Anthony Minessale II,
Giovanni Maruzzelli

FreeSWITCH 1.8

VoIP and WebRTC with FreeSWITCH: The definitive source

Packt>

< html PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
"http://www.w3.org/TR/REC-html40/loose.dtd">

FreeSWITCH 1.8

VoIP and WebRTC with FreeSWITCH: The definitive source

Anthony Minessale II
Giovanni Maruzzelli

Packt

BIRMINGHAM - MUMBAI

FreeSWITCH 1.8

First published: July 2017

Production reference: 1120717

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78588-913-4

www.packtpub.com

Credits

Authors Anthony Minessale II Giovanni Maruzzelli	**Copy Editor** Safis Editing
Reviewer Brian West	**Project Coordinator** Judie Jose
Acquisition Editor Rahul Nair	**Indexer** Aishwarya Gangawane

Content Development Editor Abhishek Jadhav	**Graphics** Kirk D'Penha
Technical Editor Aditya Khadye Manish Shanbhag	**Production Coordinator** Aparna Bhagat

About the Authors

Anthony Minessale II is the primary author and founding member of the FreeSWITCH Open Source Soft-Switch. Anthony has spent around 20 years working with open source software. In 2001, Anthony spent a great deal of time contributing code to the Asterisk PBX and has authored numerous features and fixes to that project. In 2005, Anthony started coding a new idea for an open source voice application. The FreeSWITCH project was officially open to the public on January 1 2006. In the years that followed, Anthony has been actively maintaining and leading the software development of the FreeSWITCH project. Anthony also founded the ClueCon Technology Conference in 2005, and he continues to oversee the production of this annual event.

Anthony has been the author of several FreeSWITCH books, including *FreeSWITCH 1.0.6*, *FreeSWITCH 1.2*, *FreeSWITCH Cookbook*,*FreeSWITCH 1.6 Cookbook*, and *Mastering FreeSWITCH*

I'd like to thank my wife Jill and my kids, Eric and Abbi, who were in grade school when this project started and are now grown up. I'd also like to thank everyone who took the time to try FreeSWITCH and submit feedback. I finally thank my coauthor Giovanni Maruzzelli for working on this book.

Giovanni Maruzzelli (gmaruzz@OpenTelecom.IT) is heavily engaged with FreeSWITCH. In it, he wrote a couple of endpoint modules, and he is specialized in industrial grade deployments and solutions. He's the curator and coauthor of *FreeSWITCH 1.6 Cookbook* (Packt Publishing, 2015), and of *Mastering FreeSWITCH* (Packt Publishing, 2016)

He's a consultant in the telecommunications sector, developing software and conducting training courses for FreeSWITCH, SIP, WebRTC, Kamailio, and OpenSIPS.
As an Internet technology pioneer, he was the cofounder of Italia Online in 1996, which was the most popular Italian portal and consumer ISP. Also, he was the architect of its Internet technologies Italia Online (IOL). Back then, Giovanni was the supervisor of Internet operations and the architect of the first engine for paid access to il Sole 24 Ore, the most-read financial newspaper in Italy, and its databases (migrated from the mainframe). After that, he was the CEO of the venture capital-funded company Matrice, developing telemail unified messaging and multiple-language phone access to e-mail (text to speech). He was

also the CTO of the incubator-funded company Open4, an open source managed applications provider. For 2 years, Giovanni worked in Serbia as an Internet and telecommunications investment expert for IFC, an arm of the World Bank.

Since 2005, he has been based in Italy, and he serves ICT and telecommunication companies worldwide.

I'd like to thank the Open Source VoIP and RTC Community (that's our ecosystem, where FreeSWITCH is thriving because of the others). The whole FreeSWITCH Community, and each single individual that day in and day out tests, documents, and helps each other in implementing FreeSWITCH.

And I like to thank Tony, Brian, Mike, Ken, Italo, and Seven for the incredible efforts they put into FreeSWITCH, and for the incredible results those efforts bears for us all to enjoy. Thanks my guys!

Anthony Minessale II, you get a special personal thank you for bearing your role with so much grace.

About the Reviewer

Brian West is a founding member of the FreeSWITCH team. He has been involved in open source telephony since 2003. Brian was heavily involved in the Asterisk open source PBX Project as a Bug Marshal and developer. In 2005, Brian joined the initiative that eventually lead to the FreeSWITCH Open Source Soft-Switch. Today, Brian serves as the general manager of the FreeSWITCH project and keeps the software moving forward. Brian has countless skills as a developer, tester, manager, and technologist, and he fills a vital role in the FreeSWITCH Community

Customer Feedback

Table of Contents

Preface

This is the third book in a Trilogy: *FreeSWITCH 1.6 Cookbook, Mastering FreeSWITCH,* and yours truly, *FreeSWITCH 1.8*. Each one of them has its own role in FreeSWITCH's Universe. The call of this one you're reading is to be the reference and foundation, the exhaustive source of FreeSWITCH knowledge.

We've worked hard, we started from the efforts of our co-writers in *FreeSWITCH 1.2 book* (they were Michael S. Collins, Darren Schreiber and Raymond Chandler), we've updated, rewritten and written anew and we're now proud to introduce you to all you need to know about latest FreeSWITCH.

So much happened in the world of Real Time Communication and VoIP in the last years. And FreeSWITCH was leading innovation, implementation and interoperability in the industry: WebRTC, Video MCU, High Definition Audio, OPUS, VP8/9, JSON, Encryption and Security, you name it, FreeSWITCH was there.

We're about to announce FreeSWITCH 1.8: an evolutionary step from 1.6, that enhance performances and changes a lot of under the hood plumbing that will be the nervous system of The Next Big Things (TM) in FreeSWITCH Universe.

From an operation, administration and application programming point of view 1.8 is completely compatible with 1.6, and this book cover them both.

What this book covers

Chapter 1, *Architecture of FreeSWITCH*, gives a brief, but thorough introduction to the underlying architecture of FreeSWITCH.

Chapter 2, *Building and Installation*, shows how to download and install FreeSWITCH on Windows and Unix-like operating systems.

Chapter 3, *Test Driving the Example Configuration*, provides a hands-on look at the powerful and feature-rich example FreeSWITCH configuration.

Chapter 4, *User Directory, SIP, and Verto*, offers an introduction to the concept of users and the directory as well as a brief look at SIP and Verto user agents.

Chapter 5, *WebRTC, SIP, and Verto*, introduce WebRTC and its FreeSWITCH implementation, for both SIP and Verto signaling protocol, with an example web video conferencing application written for both protocols.

Chapter 6, *XML Dialplan*, explains the basic dialplan concepts and how to create and edit extensions to add advanced functionality to a FreeSWITCH install.

Chapter 7, *Phrase Macros and XML IVRs*, discusses how to create menus and sound phrases for interacting with callers as well as the useful Phrase Macro system.

Chapter 8, *Lua FreeSWITCH Scripting*, introduces the concept of advanced call handling using the lightweight scripting language Lua.

Chapter 9, *Dialplan in Deep*, builds upon the foundation laid in Chapter 6, and shows how to handle more challenging routing scenarios.

Chapter 10, *Dialplan, Directory, and ALL via XML_CURL and Scripts*, discusses how to configure and control FreeSWITCH dynamically, such as from a database.

Chapter 11, *ESL - FreeSWITCH Controlled by Events*, introduce Event System and Event Socket Library, how to tap into FreeSWITCH internal messaging bus to interact with calls in real time.

Chapter 12, *HTTAPI - FreeSWITCH Asks Webserver Next Action*, explains how to have FreeSWITCH to interrogate a webserver in real time at each step of a call interaction: a handy way to create IVRs.

Chapter 13, *Conferencing and WebRTC Video-Conferencing*, discusses how to create and manage audio and video conferences that allows for legacy, mainstream and WebRTC users to interact and collaborate. Also shows the special advanced features of Verto Communicator, our sleek Web Real Time Communication client.

Chapter 14, *Handling NAT*, provides much needed insight into understanding how NAT causes issues with SIP and how to work around them.

Chapter 15, *VoIP Security*, offers suggestions on how to secure VoIP communications from prying eyes as well as securing a FreeSWITCH server against various attacks.

Chapter 16, *Troubleshooting, Asking for Help, and Reporting Bugs*, shows how to find out what's wrong when you encounter a problem, how to interact with the wonderful FreeSWITCH community for getting help, and what to do when you think you've hit a bug.

What you need for this book

At the very least you will need a computer on which you can run FreeSWITCH. Typically this is a server although that isn't an absolute requirement. You will also need at least a couple SIP devices, be them softphones or desk phones, and a couple browsers as WebRTC clients.

Who this book is for

This book is for prospective FreeSWITCH administrators and application programmers as well as VoIP professionals who wish to learn more about how to set up, configure, and extend a FreeSWITCH installation. If you are already using FreeSWITCH, you will find that the information in this book compliments what you have already learned from your personal experience.

A solid understanding of basic networking concepts is very important. Previous experience with VoIP is not required, but will certainly make the learning process go faster.

Conventions

In this book, you will find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning. Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can see FreeSWITCH status information by issuing the `stuts` command at the FreeSWITCH console."

A block of code is set as follows:

```
<extension name="get voicemail">
<condition field="destination_number" expression="^\*98$">
<action application="answer"/>
<action application="voicemail"
     data="check auth default ${domain_name}"/>
</condition>
</extension>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
<extension name="get voicemail">
<condition field="destination_number" expression="^\*98$">
<action application="answer"/>
<action application="voicemail"
     data="check auth default ${domain_name}"/>
</condition>
</extension>
```

Any command-line input or output is written as follows:

```
/usr/local/freeswitch/bin/fs_cli -x version
```

New terms and **important words** are shown in bold.

Warnings or important notes appear like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book-what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of. To send us general feedback, simply e-mail`feedback@packtpub.com`, and mention the book's title in the subject of your message. If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you. You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the SUPPORT tab at the top.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on Code Download.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

The code bundle for the book is also hosted on GitHub at https://github.com/PacktPublishing/FreeSWITCH-1.8. We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from https://www.packtpub.com/sites/default/files/downloads/FreeSWITCH1.8_ColorImages.pdf.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books-maybe a mistake in the text or the code-we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title. To view the previously submitted errata, go to https://www.packtpub.com/books/content/supportand enter the name of the book in the search field. The required information will appear under the Errata section.

Piracy

Questions

If you have a problem with any aspect of this book, you can contact us at`questions@packtpub.com`, and we will do our best to address the problem.

Architecture of FreeSWITCH

Welcome to FreeSWITCH! If you are reading this, then you're interested in things such as real-time communication, WebRTC, telecommunications, and **Voice over Internet Protocol** (**VoIP**). FreeSWITCH is a complete application server and a flexible toolset, used worldwide to build advanced and future-proof communication applications. Before looking at the architecture of this powerful software, let's take a look at the world of telecommunications. This will help to put FreeSWITCH into perspective.

In this chapter, we will cover:

- A communication double revolution
- Advantages of FreeSWITCH
- Endpoint and Dialplan modules
- How FreeSWITCH simplifies complex applications such asvoicemail and video-conferences

Real-time communication without mysteries

Real-time communication (**RTC**) began with telephone during the second half of the 1800s, and almost immediately evolved into a big, worldwide, interconnected network of big companies and infrastructure.

Until a few years ago, telephony was a walled castle strictly guarded by huge corporations and almost nobody was able to completely understand how it was actually working. You got a chance to gain that esoteric knowledge by attending internal technical seminars and in-house schools, and even that knowledge was limited to the part of the system you were to work on (central office, last mile, PBX, and so on). Both the infrastructure carrying and routing the calls, and the applications answering and managing them, were ad hoc, rigid, not compatible with each other, and required huge investments.

Then two revolutions completely obliterated that old world. The first one was the VoIP telephony revolution, which brought an open protocol (SIP) to power, first in the application/PBX realm, and then in the infrastructure too. First to go was that steel box containing a proprietary PBX that could only be expanded by changing internal hardware cards made by that same company, serving only its own kind and model of hardware phones. It was substituted by a standard server with standard PC architecture, using off-the-shelf expansion cards, able to support any kind of standard-compliant phones. SIP then climbed from bottom to top, up to the very core of Big Telcos infrastructure. Today, all telecommunication infrastructures including those of Big Telcos and Carriers are running some version of the SIP protocol.

And the second revolution, which is ongoing right now and will take some more years to complete and bear fruits: **WebRTC**. This is a very misleading name; WebRTC does not require web pages and browsers at all. WebRTC is a set of standards for the encrypted interconnections of communication endpoints. This WebRTC standard happened to first be implemented in browsers.In the mean time it has become the mainstream way to communicate in the **Internet of Things**, spanning from smartphone apps to cars, lifts, shop cashiers, and point of sale.

Nowadays it's possible to build communication systems that outperform traditional voice, video, and conference services and offer advanced features for a relatively low cost. FreeSWITCH has been designed to make all of this easier, and we will go over its architecture to get a better understanding of how it works.

Don't be concerned if you don't grasp it all in one swoop. Learning takes time, especially RTC and VoIP. In fact, we recommend that you read this chapter more than once. Absorb as much as you can on the first pass, then come back after you complete Chapter 6, *XML Dialplan*. You will be surprised at how much your understanding of FreeSWITCH has improved. Then come back and skim it a third time after you have completed Chapter 10*, Dialplan, Directory, and ALL via XML_CURL and Scripts*; at this point, you will have a firm grasp of FreeSWITCH concepts. Give yourself time to digest all of these new concepts, and soon you will find that you are a skilled FreeSWITCH administrator.

Today, we live in an ecosystem of multiple **RTC** technologies, all coexisting at the same time: telephony systems (such as telephone switches and PBXs), the traditional analog phone (**POTS** lines or **Plain Old Telephone Service**), traditional telephone networks operated by carriers (PSTN or Public Switched Telephone Network), mobile phones (CDMA, GSM, LTE, and so on), faxes, WebRTC, smartphone apps, VoIP phones, and Enterprise systems.

FreeSWITCH's niche is right in the center of this ecosystem: it connects and accepts connections from all those technologies, it bridges and mixes them together, it provides interactive applications and services to users whatever their endpoint happens to be. FreeSWITCH is able to connect to external data services and to legacy in-house systems, to computer programs and business procedures. FreeSWITCH runs on Linux, Windows, and Mac OS X, as well as *BSD and Solaris. We have plenty of hardware choices, from big multiprocessor servers to Raspberry Pi. So, you can develop on your laptop, and then deploy on datacenter, or in an embedded device. Installing FreeSWITCH is discussed in detail in Chapter 2, *Building and Installation*.

The FreeSWITCH design - modular, scalable, and stable

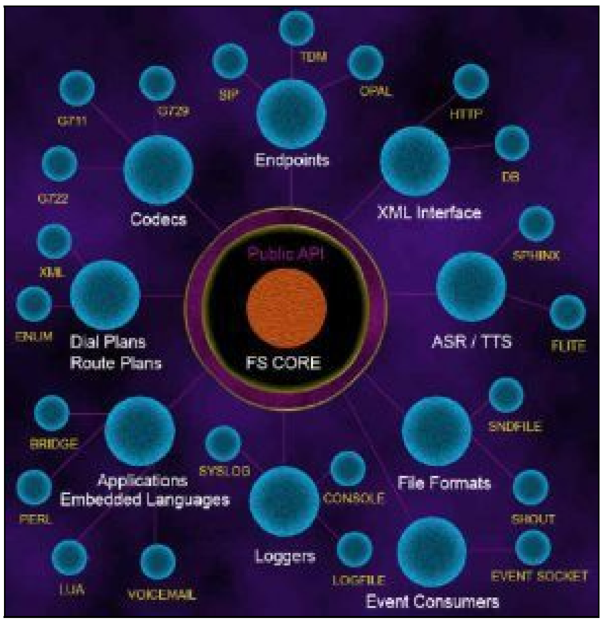
The design goal of FreeSWITCH is to provide a modular, scalable system around a stable switching core, and to provide a robust interface for developers to add to and control the system. Various elements in FreeSWITCH are independent of each other and do not have much knowledge about how the other parts are working, other than what is provided in what are called "FreeSWITCH APIs". The functionality of FreeSWITCH can be extended with loadable modules, which tie a particular functionality or an external technology into the core.

FreeSWITCH has many different module types that surround the central core, like sensors and interfaces connect a robot *brain* to the external environment. The list includes the following:

Module type	**Purpose**
Endpoint	Telephone protocols such as SIP and PSTN lines.
Application	Performs a task such as playing audio or sending data.
Automated Speech Recognition (ASR)	Interfaces with speech recognition systems.
Chat	Bridges and exchanges various chat protocols.
Codec	Translates between audio formats.
Dialplan	Parses the call details and decides where to route the call.
	Connects directory information services, such as LDAP, to a

Directory	common core lookup API.
Event handlers	Allows external programs to control FreeSWITCH.
File	Provides an interface to extract and play sound from various audio file formats.
Formats	Plays audio files in various formats.
Languages	Programming language interfaces used for call control.
Loggers	Controls logging to the console, system log, or log files.
Say	Strings together audio files in various languages to provide feedback to say things such as phone numbers, time of day, spellings of words, and so on.
Text-To-Speech (TTS)	Interfaces with text-to-speech engines.
Timers	POSIX or Linux kernel timing in applications.
XML Interfaces	Uses XML for **Call Detail Records** (**CDRs**), CURL, LDAP, RPC, and so on.

The following diagram shows what the FreeSWITCH architecture looks like and how the modules orbit the core of FreeSWITCH:



By combining the functionalities of the various module interfaces, FreeSWITCH can be configured to connect IP phones, POTS lines, WebRTC, and IP-based telephone services. It can also translate audio formats and provides an **Interactive Voice Response** (**IVR**) system with custom menus. A FreeSWITCH server can also be controlled from another machine. Let's start by taking a closer look at a pair of important module types.

Important modules - Endpoint and Dialplan

Endpoint modules are critically important and add some of the key features that make FreeSWITCH the powerful platform it is. The primary role of endpoint modules is to take certain common communication technologies and normalize them into a common abstract entity, which we refer to as a **session**. A session represents a connection between FreeSWITCH and a particular protocol. There are several Endpoint modules that come with FreeSWITCH, which implement several protocols such as SIP, H.323, Jingle, Verto, WebRTC, and some others. We will spend some time examining one of the more popular modules named `mod_sofia`.

Sofia-SIP (http://sofia-sip.sourceforge.net) is an open source project originally developed by Nokia, which provides a programming interface for the Session Initiation Protocol (SIP). FreeSWITCH developers have worked a lot evolving and fixing its original code base, further enhancing its robustness and features. We use our version of this library in FreeSWITCH (/usr/src/freeswitch/libs/sofia-sip) in a module named `mod_sofia`. This module registers to all the hooks in FreeSWITCH necessary to make an Endpoint module, and translates the native FreeSWITCH constructs into Sofia SIP constructs and vice versa. Configuration information is taken from FreeSWITCH configuration files, which makes for `mod_sofia` to load user-defined preferences and connection details. This allows FreeSWITCH to accept registrations from SIP phones and devices, register itself to other SIP servers such as service providers, send notifications, and provide services to the phones such as blinking lights and voicemail.

When a SIP audio/video call is established between FreeSWITCH and another SIP device, it will show up in FreeSWITCH as an active session. If the call is inbound, it can be transferred or bridged to IVR menus, hold music (and/or video), or to one or more extensions. Or, it can be bridged to a newly created outbound call, toward a PSTN subscriber, or a WebRTC browser, for example. Let's examine a typical scenario where an internal SIP phone registered as extension 2000 dials extension 2001 with the hope of establishing a call.

First, the SIP phone sends a call setup message to FreeSWITCH over the network (`mod_sofia` is listening for such messages). After receiving the message, `mod_sofia` in turn parses the relevant details, builds the abstract call *session* data structure the core understands, and passes that call into the core state machine in FreeSWITCH. The state machine (in the FreeSWITCH core) then puts the call into the **ROUTING** state (meaning looking for a destination).

Core's next step is to locate the Dialplan module based on the configuration data for the calling Endpoint. The default and most widely used Dialplan module is the XML Dialplan module. This module is designed to look up a list of instructions from the XML tree within FreeSWITCH's memory. The XML Dialplan module will parse a series of XML objects using regular expression pattern-matching.

As we are trying to call 2001, we are looking around for an XML extension where the `destination_number` field matches 2001. Also, let's remember the Dialplan is not limited to matching only a single extension. An incoming call can match more than one extension in the Dialplan, and in Chapter 6, *XML Dialplan*, you will get an expanded definition of the term extension. The XML Dialplan module builds a TODO list for the call. Each matched extension will have its actions added to the call's TODO list.

Assuming FreeSWITCH finds at least one extension with a matching condition, the XML Dialplan will insert instructions into the session object with the information it needs to try and connect the call to 2001 (the TODO list for this call). Once these instructions are in place, the state of the call session changes from ROUTING to EXECUTE, where the core drills down the list and executes the instructions piled during the ROUTING state. This is where the API comes into the picture.

Each instruction is added to the session in the form of an application name and a data argument that will be passed to that application. The one we will use in this example is the bridge application. The purpose of this application is to create another session with an outbound connection, then connect the two sessions for direct audio exchange. The argument we will supply to bridge will be user/2001, which is the easiest way to generate a call to an internal registered phone at extension 2001. A Dialplan entry for 2001 might look like this:

```
<extension name="example">
<condition field="destination_number"
                expression="^2001$">
<action application="bridge" data="user/2001"/>
</condition>
</extension>
```

This extension is named example, and it has a single condition to match. If that condition is matched, it has a single application to execute. It can be understood as: If the caller dialed 2001, then establish a connection between the caller and the endpoint (that is, the phone) at 2001.

Once we have inserted the instructions into the TODO list for the session, the session's state will change to EXECUTE, and the FreeSWITCH core will start to use the data collected to perform the desired action(s). First, it will parse the list and find it must execute bridge on user/2001, then it will look up the bridge application and pass the user/2001 data to it. This will cause the FreeSWITCH core to create a new outbound session (that is, a call) of the desired type. Let's assume that user 2001 is registered to FreeSWITCH using a SIP phone. So user/2001 will resolve into a SIP dialstring, which will be passed to `mod_sofia` to ask it to create a new outbound session (toward user 2001's SIP phone).

If the setup for that new session is successful, there will be two sessions in the FreeSWITCH core, the new session and the original session (from the caller phone). The bridge application will take those two sessions and call the bridge function on it. This will make the audio and/or the video stream to flow in both directions once the person at extension 2001 answers the phone. If that user is unable to answer or is busy, a timeout (that is, a failure) would occur and a failure message will be sent back to the caller's phone. If a call is unanswered or an extension is busy, many reactions can be built in the Dialplan, including call forwarding or voicemail.

FreeSWITCH takes all the complexity of SIP and reduces it to a common (internal to its core) denominator. Then it reduces the complexity further by allowing us to use a single instruction in the Dialplan for connecting the phone at 2000 to the phone at 2001. If we also want to allow the phone at 2001 to be able to call the phone at 2000, we can add another entry in the Dialplan going the other way:

```
<extension name="example 2">
<condition field="destination_number" expression="^2000$">
<action application="bridge" data="user/2000"/>
</condition>
</extension>
```

In this scenario, the Endpoint module (`mod_sofia`) turned incoming SIP call into a FreeSWITCH session and the Dialplan module (`mod_dialplan_xml`) turned XML into an extension. The bridge application (from `mod_dptools`) turned into a simple application/data pair the complex code of creating an outbound call and connecting its media streams. Both the Dialplan module and the application module interface are designed around FreeSWITCH sessions. Not only does the abstraction make life easier for us at the user level, it also simplifies the design of the application and the Dialplan because they can be made agnostic of the actual endpoint technology involved in the call. It is because of this abstraction that when tomorrow we will write a new Endpoint module to the 3D holographic communication network we will be able to reuse all the same applications and Dialplan modules. To the FreeSWITCH core, and to all of the FreeSWITCH modules, an holographic 3D call will then appear as just one more standard abstract session.

It is possible that you may want to work with some specific data provided by the

Endpoint's native protocol. In SIP, for instance, there are several arbitrary headers as well as several other bits of interesting data inside the SIP packets. You may need or want to access those specific bits of information. We solve this problem by adding variables to the channel (that is, the part of the session structure that interact with the endpoint). Using channel variables, `mod_sofia` can create these arbitrary values as they are encountered in the SIP data, and you can retrieve them from the channel by variable name, in your Dialplan or application. Those were originally specific (and arbitrary) parts of the low-level SIP messages. However, the FreeSWITCH core just sees them as arbitrary channel variables that the core can ignore. There are also several special reserved channel variables that can influence the behavior of FreeSWITCH in many useful ways. If you have ever used a scripting language or configuration engine that uses variables (sometimes called **attribute-value pairs** or **AVP**, channel variables are pretty much the same concept. There is simply a name and a value that are passed to the channel, and the data is set.

The application interface for this, the set application, lets you set your own variables from the Dialplan:

```
<extension name="example 3">
<condition field="destination_number" expression="^2000$">
<action application="set" data="foo=bar"/>
<action application="bridge" data="user/2000"/>
</condition>
</extension>
```

This example is almost identical to the previous example, but instead of just placing the call, we first set the variable `foo` equal to the value `bar`. This variable will remain set throughout the call and can even be referenced at the end of the call in the detail logs (CDRs).

The more we build things by small pieces, the more the same underlying resources can be reused, making the whole system simpler to use. For example, the codec interface knows nothing else about the core, other than its own isolated world of encoding and decoding audio and video packets. Once a proper codec module has been written, it becomes usable by any Endpoint interface capable of carrying that codec in one of its media streams. This means that if we get a *text-to-speech* module working, we can generate synthesized speech on any and all Endpoints that FreeSWITCH supports. The TTS module becomes more useful because it can use more codecs; the codecs have become more useful because we added a new function that can take advantage of them. The same idea applies to applications. If we write a new application module, the existing endpoints will immediately be able to run and use that application.

Complex applications made simple

FreeSWITCH removes much of the complexity from advanced applications. Let's look at two examples of a more complex application.

Voicemail

The first application we will discuss is the voicemail application. This application is useful to add right after the bridge application as a second option, executed in cases where the call was not completed. We can do this with one of those special variables that we were discussing earlier. Let's look at a version of our last extension that also allows us to leave a voicemail:

```
<extension name="example 4">
<condition field="destination_number" expression="^2000$">
<action application="set"
              data="hangup_after_bridge=true"/>
<action application="bridge" data="user/2000"/>
<action application="voicemail"
            data="default ${domain} 2000"/>
</condition>
</extension>
```

Here, we see two uses of channel variables. First, we set `hangup_after_bridge=true` telling the system to hang up once we have have successfully bridged the call to another phone and to disregard the rest of the instructions. We are using the *domain* variable in brackets prefixed with a dollar sign, `${domain}`. This is a special variable that defaults to the auto-configured domain name, which comes from the XML configuration.

In this example, we check if someone is dialing `2000`. We then try to bridge the call to the user which endpoint is registered to extension `2000`. If the call fails or if there is no answer (for example, if the bridge attempt has failed, so we do not execute the `hangup` after the bridge), we will continue to the next instruction, which is to execute the voicemail application. We provide the information the application needs to know (for example, which domain the voicemail belongs to) and which extension the voicemail is for, so the application knows how to handle the situation. Next, the voicemail application plays the pre-recorded greeting or generates one using the *Say* module's interface, which we briefly discussed earlier. It then plays short sound files one after each other to make a voice say something like *The person at extension 2 0 0 0 is not available, please leave a message*. Next, `mod_voicemail` prompts you to record a message. As an additional feature, if you are not convinced with your recording, you can listen and re-record it as many times as you wish. Once you finally commit, a FreeSWITCH MESSAGE_WAITING event is fired into the core event system queue, which is picked up by `mod_sofia` by way of an event consumer, and the event information is translated into SIP-in this case a SIP NOTIFY message that lets the SIP phone know that there is a message waiting. A blinking lamp (**Message Waiting Indicator** (**MWI**)) lights upon the receiving phone.

In this example, not only have we seen how to play a greeting, record a message, and transform it into a voicemail for a user, we have also met an important part of the FreeSWITCH core-the event system. The FreeSWITCH event system is not an additional module interface like the previous examples, it is a core engine feature that you can use to bind to named events and react accordingly when an event is received. In other words, throughout the FreeSWITCH core, there are events that are sent and received. Modules can bind to (that is listen for) various events. They can also fire events into the event engine; other modules can listen for those events. You can think of it as similar to other queuing systems such as **RabbitMQ** (actually, there is a module to interface the internal event system of a FreeSWITCH server with RabbitMQ, so you can integrate it into an enterprise queuing system, and/or having multiple FreeSWITCH servers be parts of the same big, distributed queue). As we discussed, the Sofia SIP module binds to (subscribes to) the event designated for MESSAGE_WAITING information. This allows our `mod_voicemail` module to interact with `mod_sofia` without either system having any knowledge about the other's existence. The event is blindly fired by `mod_voicemail` (fire and forget, in military parlance), intercepted (received, because has subscribed to) by `mod_sofia`, and translated into the proper SIP message-all courtesy of the event system.

There are several challenges with such a complex system of concatenating sounds when considering all of the possible languages it may need to support, as well as what files to play for the automated messages and how they are strung together. The *Say* module supplies a nice way to string files together, but it is limited to something specific, such as spelling a word, counting something, or saying a certain date. The way we overcome this is by defining a more complex layer on top of the *Say* module called **Phrase Macros**. Phrase Macros are a collection of XML expressions that pull out a list of arguments by matching a regular expression and executing a string of commands. This is very similar to how the XML Dialplan works, only custom-tailored for IVR scenarios. For example, when `mod_voicemail` asks you to record your message, rather than coding in the string of files to make it say what you want, the code just calls a phrase macro called `voicemail_record_message`. This arbitrary series of sound bites is defined in the Phrase Macro section in the XML configuration allowing us, the administrators, to edit the phrase without modifying the Voicemail IVR program:

```
<macro name="voicemail_record_message">
<input pattern="^(.*)$">
<match>
<action function= "play-file"
        data="voicemail/vm-record_message.wav"/>
</match>
</input>
</macro>
```

When `mod_voicemail` executes the `voicemail_record_message` macro, it first matches the pattern, which, in this case, is to match everything, because this particular macro has no use for input (that is, whatever input you give it, is not used). If the macro did use the input, the pattern matching could be used to play different sound bites based on different input. Once a match is found, the XML *match* tag is parsed for action tags just like in our Dialplan example. This macro only plays the file `vm-record_message.wav`, but more complicated macros, like the ones for verifying your recording or telling you how many messages you have in your inbox, may use combinations of various *Say* actions and play many different audio files. Phrase Macros are discussed in detail in Chapter 6, *XML Dialplan*, and used extensively in Chapter 8, *Lua FreeSWITCH Scripting*.

Here too, we can see co-operation between various parts of FreeSWITCH architecture: the phrase system, the audio file, and the *Say* modules loaded by the core are used together to enable powerful functionalities. The *Say* modules are written specifically for a particular language or voice within a language. We can programmatically request to say the current time and have it translated into Spanish or Russian sounds by the proper Say module based on input variables. The Phrase Macro system is a great way to put a layer of abstraction into your code, which can be easily tweaked later by system administrators. For example, if we wanted to make a small IVR that asks us to dial a four-digit number, then reads it back and hangs up, we could make one macro called `myapp_ask_for_digits` and the other called `myapp_read_digits`. In our code, we would execute these macros by name-the former when it is time to ask for the digits and the later to read back the digits by passing in the value we entered. Once this is in place, a less-experienced individual (for example, a local administrator) could implement the XML files to play the proper sounds. She can use the *Say* modules to read back the number, and it should all be working in multiple languages with no further coding necessary. Voicemail is just one example of using FreeSWITCH as an application server. There are endless possibilities when we use FreeSWITCH to connect real-time communication with computers.

Multi-party audio/video conferencing

Another important feature of FreeSWITCH is delivered by the `mod_conference` conferencing module. The `mod_conference` provides dynamic conference rooms that can bridge together the audio and video from several users. It may mix video streams together, applying CG (computer graphics) transformations to them, such as composing a live feed of different conference participants together, over imposing a caption with the name and role to each users' video stream, sharing the screen of each participant computer (for example, a PowerPoint presentation), and so on. Also, a real-time chat can be added to the conference, so participants can exchange text messages out of band from the main audio/video stream. Obviously, this same module can also be used for plain regular audio conference calls.

Each new session that connects to the same conference room will join the others, and instantly be able to talk and see all of the other participants at the same time (as per the whim of the conference admin, who can choose who to display, who can talk, and so on). Using an example similar to the one we used for bridging to another phone, we can make an extension to join a conference room:

```
<extension name="example 4">
<condition field="destination_number"  expression="^3000$">
<action application="conference" data="3000@default"/>
</condition>
</extension>
```

This is as simple as bridging a call, but with a conference application many callers can call the same extension (`3000` in this case) and join the same conference room. If three people joined this conference and one of them decides to leave, the other two would still be able to continue their conversation.

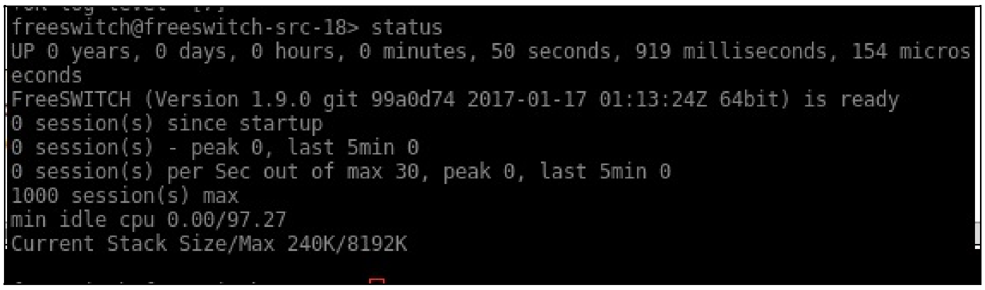
The conference module also has other special features, such as the ability to play sound or video files or text-to-speech to the whole conference, or even to a single member of the conference. As you may have guessed, we are able to do this by using the TTS and video/sound file interfaces provided by their respective modules. The smaller pieces come together to extend the functionality without needing knowledge of each other.

The conference module also uses the event system in an additional way, employing what are called custom events. When it first loads, a module can reserve a special event namespace called a subclass. When something interesting happens, such as when a caller joins or leaves a conference, it fires those events on the CUSTOM event channel in the core queue. When we are interested in receiving such events, all we have to do is subscribe to the CUSTOM event by supplying an extra subclass string, which specifies the specific CUSTOM events we are interested in. In this case, it is `conference::maintenance`. This makes it possible to look out for important things such as when someone joins or leaves the conference, when they start and stop talking, when they are displayed on video, or what video layout (screen disposition) is currently in use. Conferencing is discussed in detail in Chapter 13*, Conferencing and WebRTC Video-Conferencing*.

FreeSWITCH API commands (FSAPI)

Another very powerful FreeSWITCH concept is the FSAPI. Most API commands are implemented in mod_commands, and almost all other modules add some to the commands that are executable via FSAPI. FSAPI mechanism is very simple-it takes a single string of text as input, which may or may not be parsed, and performs a particular action. The return value is also a string that can be of any size, from a single character up to several pages of text, depending on the function that was called by the input string. One major benefit of FSAPI functions is that a module can use them to call routines in another module without directly linking into the actual compiled code (thus avoiding sudden incompatibilities and crashes). The most egregious example is the command-line interface of FreeSWITCH or CLI, which uses FSAPI functions to pass FreeSWITCH API commands.

Here is a small example of how we can execute the status FSAPI command from the FreeSWITCH CLI:



```
freeswitch@freeswitch-src-18> status
UP 0 years, 0 days, 0 hours, 0 minutes, 50 seconds, 919 milliseconds, 154 micros
econds
FreeSWITCH (Version 1.9.0 git 99a0d74 2017-01-17 01:13:24Z 64bit) is ready
0 session(s) since startup
0 session(s) - peak 0, last 5min 0
0 session(s) per Sec out of max 30, peak 0, last 5min 0
1000 session(s) max
min idle cpu 0.00/97.27
Current Stack Size/Max 240K/8192K
```

What's really happening here is that when we type status and press the Enter key, the word `status` is used to look up the status FSAPI function from the module in which it is implemented. The underlying function is then called (passing it the arguments if they were typed, in this case none), and the core is queried for its status message. Once the status data is obtained, the output is written to a stream that prints a string.

We have already learned that a module can create and export FSAPI functions that can be executed from anywhere such as the CLI. But there's more. Modules can also be written to execute commands via the FSAPI interface and then send the results over a specific protocol. There are two modules included in FreeSWITCH that do just that-`mod_xml_rpc` and `mod_event_socket` (discussed in Chapter 10, *Dialplan, Directory, and ALL via XML_CURL and Scripts*, and Chapter 11, *ESL - FreeSWITCH Controlled by Events* respectively). Consider the example of `mod_xml_rpc`. This module implements the standard XML-RPC protocol (Remote Procedure Call via XML strings) as a FreeSWITCH module. Clients using whatever standard XML-RPC interface can connect to FreeSWITCH and execute FSAPI commands. So a remote client could execute an RPC call to status, and get a similar status message to the one we saw in the previous example. This same module also provides FreeSWITCH with a listening web server, which allows FSAPI commands to be accessed froma direct URL link. For example, one could point a browser to `http://example.freeswitch.box:8080/api/status` to execute the `status` command directly over HTTP. By using this technique, it's possible to create FSAPI commands that work similar to a CGI, providing a dynamic web application that has direct access to FreeSWITCH internals (for a more advanced HTTP integration, you may want to check the HTTAPI module in Chapter 12, *HTTAPI - FreeSWITCH Asks Webserver Next Action*).

As we have shown, the FSAPI interface is very versatile. Now we know it can be used to provide a CLI interface, a way for modules to call functions from each other, and a way to export HTTP or XML-RPC functions. There is still one more use for FSAPI functions that we have not covered. We touched briefly on the concept of channel variables earlier, noting that we can use the expression `${myvariable}` to get the value of a certain variable. FSAPI functions can also be accessed this way in the format `${myfunction()}`. This notation indicates that the FSAPI command `myfunction` should be called, and that the notation should be replaced with the output of that function call. Therefore, we can use `${status()}` anywhere when variables are expanded to gain access to the output of the `status` command. For example:

```
<action application="set" data="my_status=${status()}"/>
```

The value placed in the `my_status` variable will be the string output from the `status` command.

Most FSAPI commands can be easily accessed using all of the ways we have discussed. Some commands only make sense when accessed via a particular method. For instance, if we made an FSAPI command that produced HTML intended to be accessed with a web browser, we would probably not want to access it from the CLI or by referencing it as a variable. But, never say never, there are cases where it can be useful, and you have the flexibility to do it.

The XML registry

We discussed many of the fundamental components of the FreeSWITCH core and how they interact with each other. We have seen how the event system can carry information across the core to the modules, and how the XML Dialplan can query the XML registry for data. This would be a good time to explain the XML registry a bit more. The XML registry is the XML tree document that holds all of the critical data that FreeSWITCH needs to operate properly. FreeSWITCH builds that document by loading a file from your hard drive and passing it to its own pre-processor. This pre-processor can include other XML documents and execute other special operations, such as setting global variables. Global variables will then be resolved by FreeSWITCH when they're used further down in the document tree.

Once the entire document and all of the included files are parsed, replaced, and generated in a static XML document, this final static document (with all global variables substituted for) is loaded into memory. The XML registry (tree) is divided into several sections- configuration, dialplan, directory, chat plan, languages, phrases, etc. The core and the modules draw their configuration from the configuration section. The XML Dialplan module draws its Dialplan data from the dialplan section. SIP and Verto authentication, user lookup, and the voicemail module read their account information from the directory section. The Phrase Macros pull their configuration from the phrases section. If we make a change to any of the XML files on the disk, we can reload the changes into memory by issuing the `reloadxml` command from the CLI If we change the values assigned to one of the global variables, we will need to restart FreeSWITCH to apply the new value, `reloadxml` will not be enough.

Scripting language modules

Scripting language modules embed a programming language like Lua, JavaScript, Perl, C#, and so on, into FreeSWITCH, and transfer functionality between the core and the language's runtime. This allows things like IVR applications to be written in that scripting language, with a simple interface back to FreeSWITCH for all the heavy lifting. Language modules usually register into the core with the application interface and the FSAPI interface and are executed from the Dialplan. Language modules offer lots of opportunities and are very powerful. Using language modules, you can build powerful real-time communication applications in a standard programming language you already know, using its libraries for data manipulations and legacy interfacing.

The demo configuration

Understanding all of these concepts right off the bat is far from easy, and as maintainers of the software, we do not expect most people to have everything just click. This is the main reason that every new layer we put on top of the core makes things simpler and easier to learn. The demonstration configuration of FreeSWITCH is the last line of defense between new users of the software and all of the crazy, complicated, and sometimes downright evil stuff better known as Real Time Communication. We try very hard to save the users from such things.

The main purpose of the demonstration configuration in FreeSWITCH is to showcase all of the hundreds of parameters there are to work with. We present them to you in a working configuration that you could actually leave untouched and play with before trying your own hand at changing some of its options. Think of FreeSWITCH as a Lego set. FreeSWITCH and all of its little parts are like a brand new bucket Lego bricks, with plenty of parts to build anything we can imagine. The demonstration configuration is like the sample spaceship that you find in the instruction booklet. It contains step-by-step instructions on exactly how to build something you know will work. After you pick up some experience, you might start modifying your Lego ship to have extra features, or rebuild the parts into a car or some other creation. Obviously, you can leave outmany, or most, of the features built in that configuration and use only what is useful in your specific deployment. The good news about FreeSWITCH is that it comes out of the box already assembled. Therefore, unlike the bucket of Lego bricks, if you get frustrated and smash it to bits, you can just re-install the defaults and you won't have to build it again from scratch. The demonstration configuration is discussed in Chapter 3, *Test Driving the Example Configuration*.

Once FreeSWITCH has been installed, you only need to start its executable without changing one line in the configuration file. You will be immediately able to point a SIP telephone or software-based SIP softphone to the address of your server (be it your laptop, a virtual machine, a 48-core server, a Raspberry Pi, or an Amazon instance), make a test call, and access all of the functionalities of FreeSWITCH. Interfacing with other protocols will require additional configurations (such as installing SSL certificates for WebRTC and the like), but the end results will be exactly the same. If you have more than one phone, using the default configuration you should be able to configure them to each having an individual extension in the range 1000-1019, which is the extension number range that is predefined in the demonstration configuration. Once you get the phones registered, you will be able to make calls across them or have them meet in a conference room in the 3000-3399 range. If you call an extension that is not registered, or let the phone ring on another extension for too long, the voicemail application will use the phrase system to indicate that the party is not available, and ask you to record a message. If you dial 5000, you can see an example of the IVR system at work, presenting several menu choices demonstrating various other neat things FreeSWITCH can do. There are a lot of small changes and additions that can be made to the demonstration configuration while still leaving it intact.

For example, using the pre-processor directives we went over earlier, the demonstration configuration loads a list of files into the XML registry from certain places, meaning that every file in a particular folder will be combined into the final XML configuration document. The two most important points where this takes place are where the user accounts and the extensions in the Dialplan are kept. Each of the 20 extensions that are preconfigured with the defaults are stored into their own file. We could easily create a new file with a single user definition, drop it into place to add another user, and issue the `reloadxml` command at the FreeSWITCH CLI. The same idea applies to the example Dialplan. We can put a single extension into its own file and load it into place whenever we want.

Summary

FreeSWITCH is a complex system of moving parts that are integrated to produce a solid, stable core with flexible and easy-to-extend add-ons. The core extends its interfaces to modules. The modules also can bring outside functionalities into FreeSWITCH by translating various communication protocols into a common, abstract, internal format. We looked at the various module types, and demonstrated how they revolve around the core and interact with each other to turn simple abstract concepts into higher-level functionalities. We described a few of the more popular applications in FreeSWITCH-the conferencing and voicemail modules and how they, in turn, make use of other modules without ever knowing it. This agnosticism is accomplished by means of the event system. We also saw how the demonstration configuration provides several working examples to help take the edge off of an otherwise frightening feat of staring down the business end of a full-featured soft-switch.

In the following chapter, we will take our first steps towards getting a FreeSWITCH system up and running.

Building and Installation

FreeSWITCH is open-source software. So, you will always be able to obtain its source code for free. Also, it is a special kind of open source: you can modify it (or ask some hired consultant to modify it on your specs), build your product on top of it, and sell it, with no need to distribute your modifications or to pay any royalties (it is covered by the BSD like license). In fact, with or without your modifications, building and installing FreeSWITCH from source code is simple. We'll look into it in this chapter.

FreeSWITCH can be compiled and installed on Linux, Windows, *BSD, and OSX, on a range of hardware that spans 96 cores big datacenter servers to Raspberry Pi, from VMWare to KVM virtual machines, and from LXC containers to AWS instances.

For each and every platform, FreeSWITCH depends on a lot of pre-requisites, many libraries, tools, and programs, both for being compiled and for running. Obviously, for your comfort, we have automated those pre-requirements' installation on the most popular platforms.

Also, we make available ready-made packages for the most popular Linux distributions (Debian, CentOS/RHEL, Ubuntu). We'll look into this too.

In this chapter, we'll look at how to download and install FreeSWITCH from source code, and how to install it from ready-made packages.

Finally, we will explain how to launch FreeSWITCH and how to run it in the background (as *daemon*, or *service*).

In this chapter, we will cover the following topics:

- Choosing the platform on which to install FreeSWITCH
- Choosing between installing from source code or from ready-made packages
- Installing FreeSWITCH on Linux and Windows
- Launching FreeSWITCH and running it in the background

Where to Install FreeSWITCH

FreeSWITCH can be installed on almost anything, which is good for testing, developing, and opens up a wealth of different use cases (think embedded systems and Internet of Things).

For production systems and, particularly for mission-critical systems (think the PBX of your company, or a taxi dispatching service, or your hot startup WebRTC-enabled commercial videoconferencing-cum-translation service), you have three choices: use the platforms we counsel (the most tried and true), or know exactly what you're doing (hint: it is impossible you know so much about FreeSWITCH internals), or ask for a qualified consultancy. That's important not only to save on time, money, efforts, and frustrations, but so much more for the ongoing support, bug fixing, features upgrade, and platform evolution.

Choice of Operating System

Which operating system is best for running FreeSWITCH in production? Choose the one that is the most used in your company, the one in which you have most expertise, and the one you feel comfortable with and where you know where to look for support. As long as you choose one of the counseled operating systems, we've got your back covered.

Please note that on Linux it may appear easy to compile and run FreeSWITCH on a distro different from the supported ones. It's still Linux, isn't? Wroooong! FreeSWITCH is a complex server application that, particularly for video mixing and massaging, has a lot of very specific dependencies. So, you may find it easier to compile and run FS on a different Linux distro (and we'd love to hear of your success) if you avoid video-related modules.

If there are no special considerations (for example, embedded systems) always use a 64-bit operating system; they're the most optimized for modern servers' hardware, both at the kernel level and at the libraries level. Particularly, avoid using 32-bit OS and virtual machines on 64-bit hardware. There have been multiple reports of incompatibility, and plainly speaking, it's not worth looking for unknown unknowns.

Linux

Debian 8 (Jessie) 64bit is the most counseled, deployed, and tested Linux distro, and it's the platform on which most FreeSWITCH development is primarily done. If you're into Linux, and in doubt about which distro to use, definitely use Debian Jessie. All libraries, dependencies, kernel features, and so on are tested each minute for FreeSWITCH usage in every possible configuration, use-case, and work load. Debian (Raspbian) is used on Raspberry Pi too, and the Pi3 has enough horsepower to handle video. Immediate runners up as Linux distros for FreeSWITCH are CentOS/RHEL 7 and Ubuntu 16.04 (Xenial). Please be aware that Ubuntu tends to be a little too much on the bleeding edge, to update packages and libraries often, even in **Long Term Support (LTS)** versions. We support Ubuntu, we even distribute ready-made packages, but if you don't have a strict company policy, go for Debian. If you come from Ubuntu you'll find yourself completely at home, and on the safe, conservative side. FreeSWITCH 1.8 announcement stated there will be support for Debian 9 "Stretch", that over time will become the preferred Linux distro.
Support for Stretch will probably be backported to FreeSWITCH 1.6, but please check our http://freeswitch.org/confluence official documentation for up to the minute updates.

Windows

Windows is the most popular OS for running FreeSWITCH after Linux. In a lot of Windows shops, FS happily chugs along with the whole Microsoft ecosystem, and it has supported C# since forever. We provide `.msi` installers, both for 64- (recommended for server class machines) and 32-bit Windows. Also, you can compile from source code using the free (as in beer) versions of Microsoft Visual Studio (for example, Express and Community).

OS X (Mac)

Many developers compile and run FreeSWITCH on their Macs, and there is an automated macOS FreeSWITCH Installer (macFI) that makes all the many procedures involved as easy as click and go to lunch. OS X is supported for versions 10.12 (Sierra), 10.11 (El Capitan), and 10.10 (Yosemite). Running FreeSWITCH in production on Mac servers would be *original*, but... why not?

*BSD

FreeSWITCH should compile and install cleanly on FreeBSD. Now and then there are rumors about FreeSWITCH on OpenBSD and other BSD derivatives. At least the core and the most popular modules (except video modules) would probably compile easily with little modification from the FreeBSD port.

Packages or Source?

Installing executables from packages or installers is the easiest way to go and requiresfewer dependencies (you will need no source-code downloading, no code-compilation tools, or development libraries. And that's a lot of stuff!). This is by far less time consuming.

One downside is that you end up with a directory structure that's different from what you will read in this book, or in anyFreeSWITCH documentation. That is because each Operating System, and each Linux distribution, requires packages to install stuff in a specific directories layout, for example, executables in `/usr/bin/` and documentation in /`usr/share/doc/freeswitch/`. Also, each Linux distribution may have its own specific locations for particular files. Contrast this with the results of installing FreeSWITCH from source with default "configur" values: all will be installed in a tree under `/usr/local/freeswitch/`.

We'll see below that at the end of installation from packages, we'll create symlinks to mimic the default source installation and conform to documentation.

Installing FreeSWITCH

Here we'll cover in brief the steps needed for a successful FreeSWITCH installation on two of the most popular platforms. Please be sure to check our documentation at http://www.freeswitch.org/confluence for instructions updated to the minute (FS is evolving!).

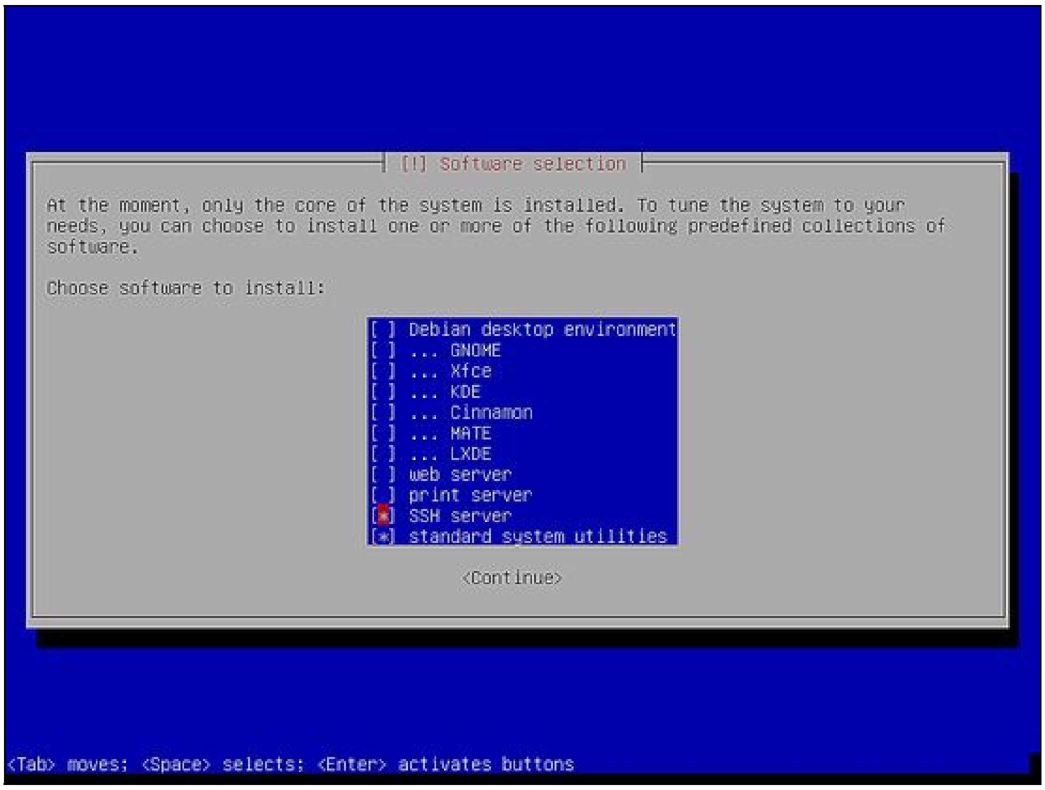
Debian 8 Jessie

This is the reference platform, the most deployed and the one on which most development is first done before being ported to other OSs and distros. If in doubt about which Linux distro to use, use this one. On Debian, both installing from source and installing from packages is supported and is mostly automated.

FreeSWITCH 1.8 announcement stated there will be support for Debian 9 "Stretch", that over time will become the preferred Linux distro.
Support for Stretch will probably be backported to FreeSWITCH 1.6, please check our http://freeswitch.org/confluence official documentation for up to the minute updates.

Be sure to start from a freshly deployed, updated, clean, bare Debian 8 (Jessie) Server 64 bit. You may want to use the `netinst` mini-image to minimize the initial download from Debian servers (because the packages will likely need to be updated anyway), or you can start from a regular Debian 8 installation medium.

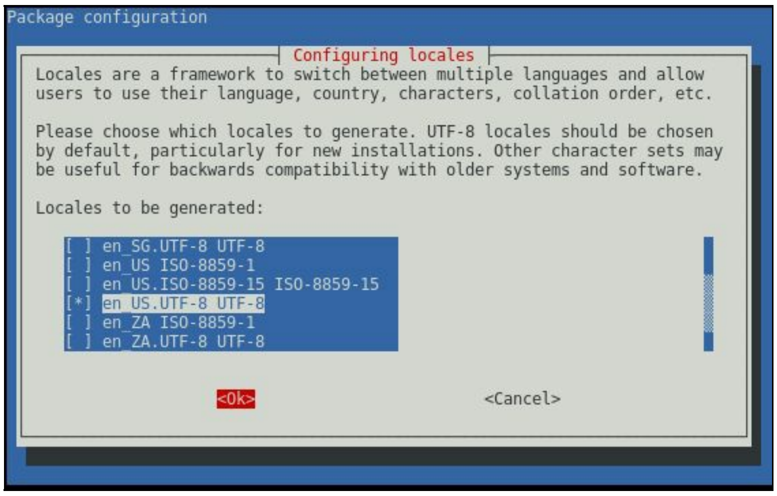
If you don't have specific requirements, you can accept all default choices proposed by the instructions on screen. At the end of Debian deployment, when asked about software selection, choose only (pressing spacebar on the list items) the SSH server and the standard system utilities; we will install all other needed packages during the following steps ourselves:


```
[!] Software selection

At the moment, only the core of the system is installed. To tune the system to your
needs, you can choose to install one or more of the following predefined collections of
software.

Choose software to install:

[ ] Debian desktop environment
[ ] ... GNOME
[ ] ... Xfce
[ ] ... KDE
[ ] ... Cinnamon
[ ] ... MATE
[ ] ... LXDE
[ ] web server
[ ] print server
[*] SSH server
[*] standard system utilities

<Continue>

<Tab> moves; <Space> selects; <Enter> activates buttons
```

After software selection, finish Debian deployment in the usual way, following the instructions on screen.

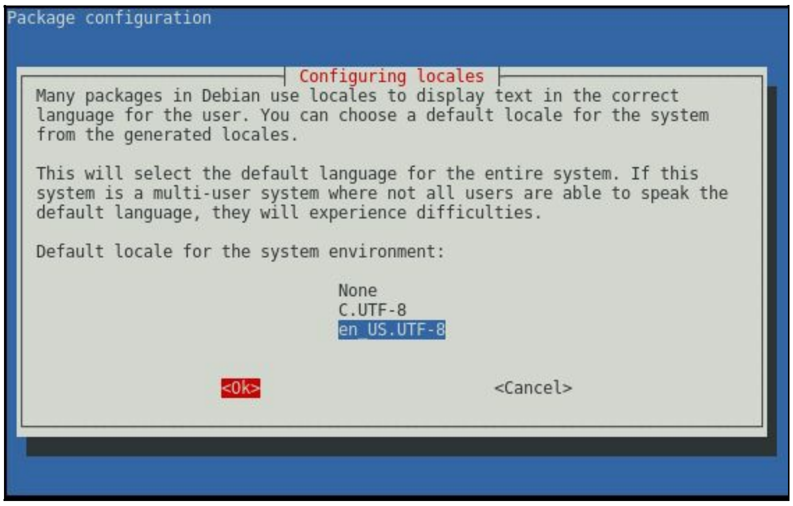
Then connect via ssh, and install and set the default locale to your preference (`en_US.UTF-8UTF-8` can be a good default), using the CLI command

> **`dpkg-reconfigure locales:`**



```
Package configuration

                  ┤ Configuring locales ├
 Locales are a framework to switch between multiple languages and allow
 users to use their language, country, characters, collation order, etc.

 Please choose which locales to generate. UTF-8 locales should be chosen
 by default, particularly for new installations. Other character sets may
 be useful for backwards compatibility with older systems and software.

 Locales to be generated:

    [ ] en_SG.UTF-8 UTF-8
    [ ] en_US ISO-8859-1
    [ ] en_US.ISO-8859-15 ISO-8859-15
    [*] en_US.UTF-8 UTF-8
    [ ] en_ZA ISO-8859-1
    [ ] en_ZA.UTF-8 UTF-8


               <Ok>                        <Cancel>
```

Then click OK.



```
Package configuration

                  ┤ Configuring locales ├
 Many packages in Debian use locales to display text in the correct
 language for the user. You can choose a default locale for the system
 from the generated locales.

 This will select the default language for the entire system. If this
 system is a multi-user system where not all users are able to speak the
 default language, they will experience difficulties.

 Default locale for the system environment:

                               None
                               C.UTF-8
                               en_US.UTF-8


               <Ok>                        <Cancel>
```

Then logout and login again, or better yet, reboot the machine altogether.

We can now start installing FreeSWITCH.

From Packages

In a new, bare, clean Debian 8 64-bit machine, execute the following as root, from the command line (following is repository for FreeSWITCH 1.6; for 1.8, please search for latest instructions on http://freeswitch.org/confluence):

```
apt-get update
apt-get -y upgrade
apt-get install -y wget ca-certificates
wget -O - https://files.freeswitch.org/repo/deb/debian/freeswitch_archive_g0.pub | apt-ke
echo "deb http://files.freeswitch.org/repo/deb/freeswitch-1.6/ jessie main" > /etc/apt/sc
apt-get update
apt-get install -y freeswitch-meta-all
```

A FreeSWITCH installation, complete with all its accessories, files, and default configuration, is now deployed on the machine.

We still have an (optional, but very useful) thing to do, because the layout on hard disk after a package installation is different from what you will read in this book and in all documentation. We create the `/usr/local/freeswitch` directory that is referred to everywhere in docs, and symlinks from there to the various FreeSWITCH components. Again, as root, from the command line, execute the following:

```
mkdir -p /usr/local/freeswitch/bin
ln -s /usr/bin/freeswitch /usr/local/freeswitch/bin/
ln -s /usr/bin/fs_* /usr/local/freeswitch/bin/
ln -s /usr/bin/fsxs /usr/local/freeswitch/bin/
ln -s /usr/bin/tone2wav /usr/local/freeswitch/bin/
ln -s /usr/bin/gentls_cert /usr/local/freeswitch/bin/
ln -s /etc/freeswitch /usr/local/freeswitch/conf
ln -s /var/lib/freeswitch/* /usr/local/freeswitch/
ln -s /var/log/freeswitch /usr/local/freeswitch/log
ln -s /var/run/freeswitch /usr/local/freeswitch/run
ln -s /etc/freeswitch/tls /usr/local/freeswitch/certs
ln -s /usr/lib/freeswitch/mod /usr/local/freeswitch/
rm /usr/local/freeswitch/lang
```

We have now installed from packages a complete FreeSWITCH system, stable, always up to date with the latest security, bug fixes, and features, and completely compatible with official documentation, community knowledge, and broscience.

Now, reboot the machine and you'll find FreeSWITCH is automatically run as *freeswitch user* (and *freeswitch* group).

We update the whole platform periodically using the command line (as root):

```
apt-get clean
apt-get update
apt-get dist-upgrade
```

From Source

Again, on a new, bare, clean Debian 8 64-bit machine, connect via ssh and execute as root, from the command line:

```
apt-get update
apt-get -y upgrade
apt-get install -y wget ca-certificates git
wget -O - https://files.freeswitch.org/repo/deb/debian/freeswitch_archive_g0.pub | apt-ke
echo "deb http://files.freeswitch.org/repo/deb/freeswitch-1.6/ jessie main" > /etc/apt/sc
apt-get update
apt-get install -y --force-yes freeswitch-video-deps-most
```

At this point, we'll see that an enormous amount of software is installed on our machine. Those are all dependencies needed to build FreeSWITCH from source. You'll end up with a complete developer suite, tools, libraries, and languages; all that's needed for FS development is there, already installed.

After a while, the mega-dependencies-galore has finished, and we can go forward to the next steps: downloading, compiling, and installing FreeSWITCH source code, demo configuration, and related accessory files:

```
git config --global pull.rebase true
cd /usr/src/
git clone https://freeswitch.org/stash/scm/fs/freeswitch.git -bv1.6 freeswitch
cd freeswitch
./bootstrap.sh -j
./configure
make
make install
make cd-sounds-install cd-moh-install
```

All those steps will take a while to finish, and you'll end up with a complete FreeSWITCH system, stable, always up to date with the latest security, bug fixes, and features, and completely compatible with official documentation, community knowledge, and broscience. (YES, you end up with the same stuff as if you had installed from packages. But maybe you want to be able to modify the sources and recompile).

We need one more step for running FreeSWITCH in the recommended way, avoiding to have it all, files, executables, processes owned by root. Let's create a specific *freeswitch user*, which will own the files that belong to the FreeSWITCH server. That same *freeswitch* user will also run the FS processes (see the `systemd` paragraph):

```
cd /usr/local
groupadd freeswitch
adduser --disabled-password--quiet --system --home /usr/local/freeswitch --gecos "FreeSWI
chown -R freeswitch:freeswitch /usr/local/freeswitch/
chmod -R ug=rwX,o= /usr/local/freeswitch/
chmod -R u=rwx,g=rx /usr/local/freeswitch/bin/*
```

Now we are ready to blend FreeSWITCH into the Debian automation and management framework, by letting `systemd` manage the start, stop, and restart of its processes. First, we install from source the `systemd` *unit* for FreeSWITCH, edit it, and then we enable it to be run at each boot:

```
cp /usr/src/freeswitch/debian/freeswitch-systemd.freeswitch.service /etc/systemd/system/f
```

Now edit that new file to reflect the locations under the `/usr/local/freeswitch/` directory. Specifically, edit the following lines:

```
PIDFile=/usr/local/freeswitch/run/freeswitch.pid
ExecStart=/usr/local/freeswitch/bin/freeswitch -u freeswitch -g freeswitch -ncwait -nonat -rp
```

Then tell `systemd` to reload its configs, start FreeSWITCH, and enable it to start automatically at boot:

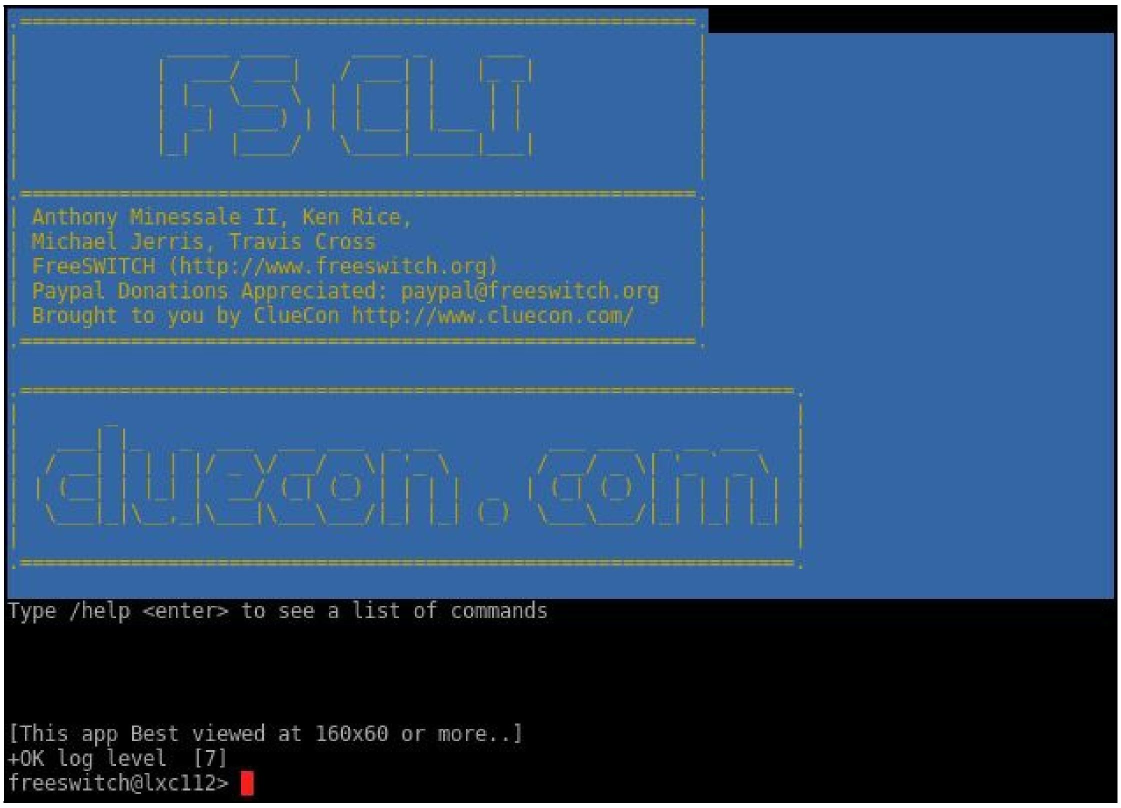
```
systemctl daemon-reload
systemctl start freeswitch
systemctl enable freeswitch
```

Now, reboot the machine and you'll find FreeSWITCH is automatically run as *freeswitch* user (and *freeswitch* group).

We can update the whole platform periodically via the command line (as root):

```
apt-get update
apt-get dist-upgrade
cd /usr/src/freeswitch
make current
```

Whatever the method we choose for installation, from ready-made packages or source, after FreeSWITCH has been started, we can connect to and manage it via the `fs_cli` executable, found in `/usr/local/freeswitch/bin`:

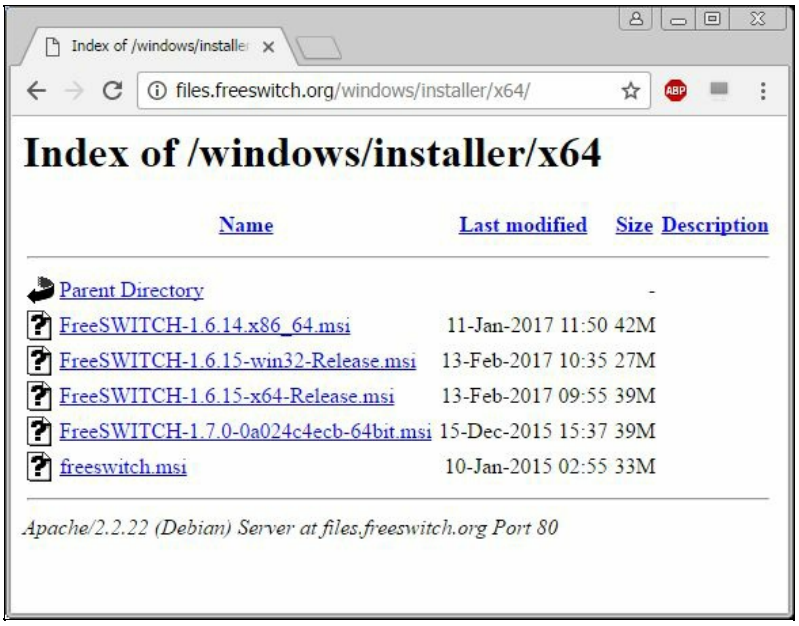


```
| Anthony Minessale II, Ken Rice,
| Michael Jerris, Travis Cross
| FreeSWITCH (http://www.freeswitch.org)
| Paypal Donations Appreciated: paypal@freeswitch.org
| Brought to you by ClueCon http://www.cluecon.com/

Type /help <enter> to see a list of commands



[This app Best viewed at 160x60 or more..]
+OK log level  [7]
freeswitch@lxc112>
```

Windows

FreeSWITCH is Windows-native by design, since the beginning. Windows is not a port or an afterthought, but an integral part of FreeSWITCH architecture, whose core makes heavy use of the APR libraries, the same that allows for the huge scalability and cross-platform performance of Apache.

The recommended way to deploy FreeSWITCH in Windows is by using the MSI installer:



files.freeswitch.org/windows/installer/x64/

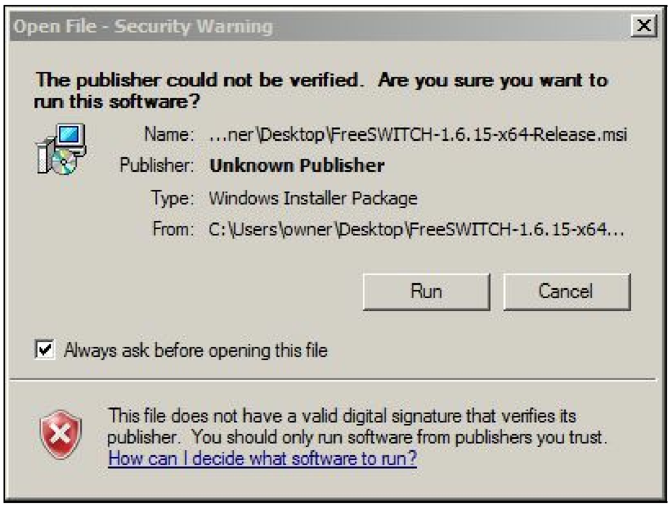
Index of /windows/installer/x64

Name	Last modified	Size	Description
Parent Directory		-	
FreeSWITCH-1.6.14.x86_64.msi	11-Jan-2017 11:50	42M	
FreeSWITCH-1.6.15-win32-Release.msi	13-Feb-2017 10:35	27M	
FreeSWITCH-1.6.15-x64-Release.msi	13-Feb-2017 09:55	39M	
FreeSWITCH-1.7.0-0a024c4ecb-64bit.msi	15-Dec-2015 15:37	39M	
freeswitch.msi	10-Jan-2015 02:55	33M	

Apache/2.2.22 (Debian) Server at files.freeswitch.org Port 80

So, in a browser, go to http://files.freeswitch.org/windows/installer/, choose your architecture (x64 is recommended), choose release, right-click on it, and save as on your desktop:

Then, double click on the FreeSWITCH Installer icon, and give it authorization to run:


Open File - Security Warning

The publisher could not be verified. Are you sure you want to run this software?

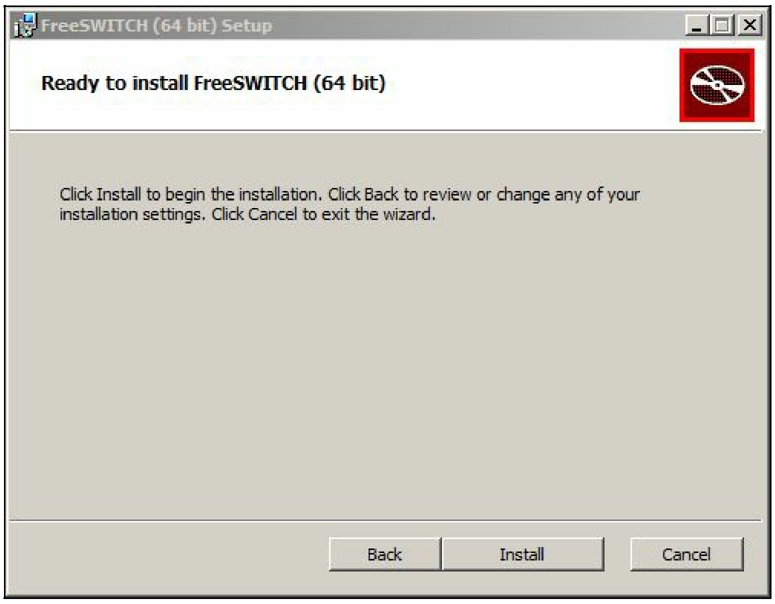
Name: ...ner\Desktop\FreeSWITCH-1.6.15-x64-Release.msi
Publisher: Unknown Publisher
Type: Windows Installer Package
From: C:\Users\owner\Desktop\FreeSWITCH-1.6.15-x64...

Always ask before opening this file [✓]

This file does not have a valid digital signature that verifies its publisher. You should only run software from publishers you trust.
How can I decide what software to run?

After accepting the license, choose a Typical installation.

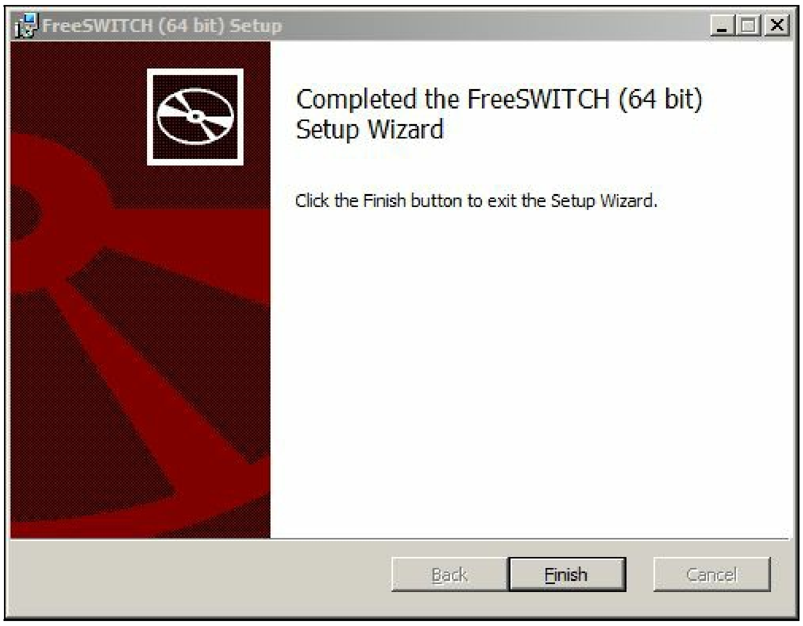
Then, click on Install:


FreeSWITCH (64 bit) Setup

Ready to install FreeSWITCH (64 bit)

Click Install to begin the installation. Click Back to review or change any of your installation settings. Click Cancel to exit the wizard.

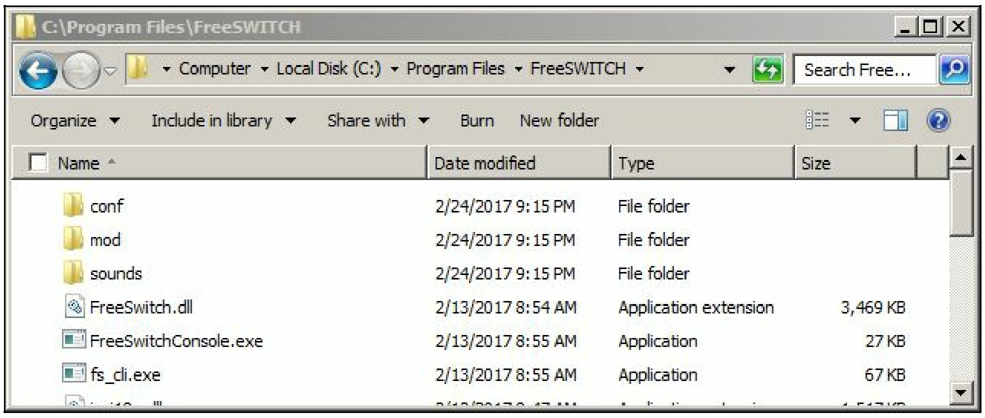
After a while, the installer will finish working its magic:



Completed the FreeSWITCH (64 bit) Setup Wizard

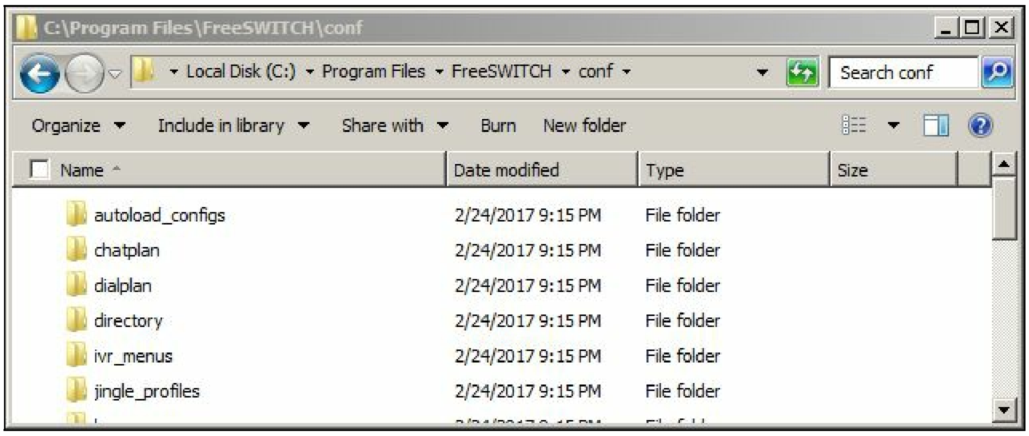
Click the Finish button to exit the Setup Wizard.

You will then find a neat standard FreeSWITCH directory tree under `C:\Program Files\FreeSWITCH\`, complete with all standard demo sound files, and executables:


C:\Program Files\FreeSWITCH
Computer ▸ Local Disk (C:) ▸ Program Files ▸ FreeSWITCH ▸
Search Free...
Organize
Include in library
Share with
Burn
New folder

Name	Date modified	Type	Size
conf	2/24/2017 9:15 PM	File folder	
mod	2/24/2017 9:15 PM	File folder	
sounds	2/24/2017 9:15 PM	File folder	
FreeSwitch.dll	2/13/2017 8:54 AM	Application extension	3,469 KB
FreeSwitchConsole.exe	2/13/2017 8:55 AM	Application	27 KB
fs_cli.exe	2/13/2017 8:55 AM	Application	67 KB

Also, in directory "conf" you'll find already installed the demo configuration with all the files we'll talk about in next chapters.

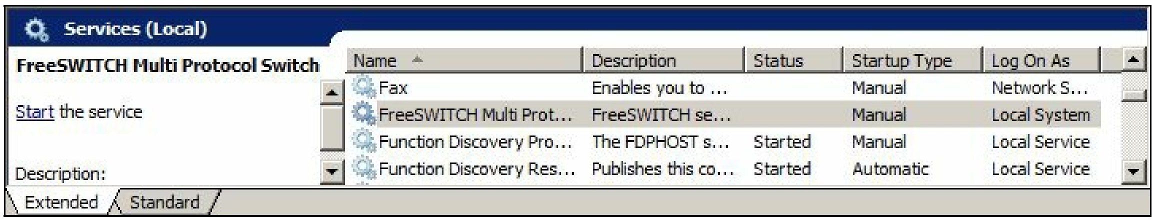

C:\Program Files\FreeSWITCH\conf
Local Disk (C:) ▸ Program Files ▸ FreeSWITCH ▸ conf ▸
Search conf
Organize
Include in library
Share with
Burn
New folder

Name	Date modified	Type	Size
autoload_configs	2/24/2017 9:15 PM	File folder	
chatplan	2/24/2017 9:15 PM	File folder	
dialplan	2/24/2017 9:15 PM	File folder	
directory	2/24/2017 9:15 PM	File folder	
ivr_menus	2/24/2017 9:15 PM	File folder	
jingle_profiles	2/24/2017 9:15 PM	File folder	

You'll feel completely at home following this book and the other FreeSWITCH documentation: the only things that are different from Linux are the root path (in this case, `C:\Program Files\FreeSWITCH` instead of `/usr/local/freeswitch`) and the path separator (backslash `\` instead of slash `/`). Your FreeSWITCH deployment is still configured by the same XML files and uses the same set of sound files as its Linux sibling.

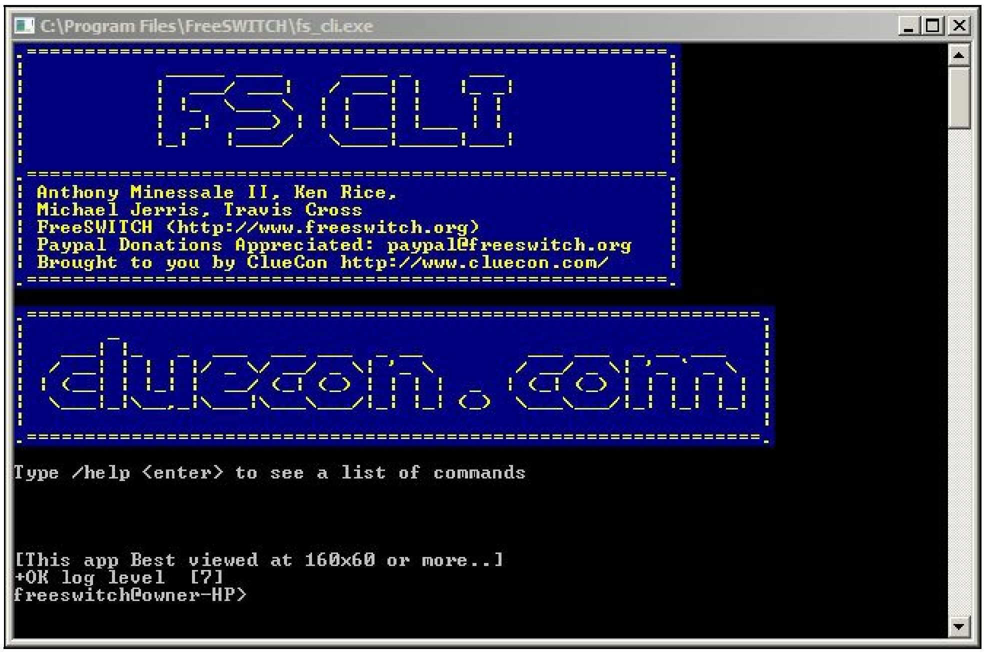
Now it is time to register FreeSWITCH with the Windows Firewall, so it will be allowed to accept incoming connections (network packets). For this, use the executable `FreeSwitchConsole.exe` found in the installation root directory (in our case, `C:\Program Files\FreeSWITCH`). Double-click on it. This will start FreeSWITCH in the foreground, as a *normal* program (as opposed to a background *service* program), and will cause Windows to ask you if FS can be authorized to accept and make network connections. Give it all authorizations. Then, type `fsctl shutdown` in the console.

FreeSWITCH has been registered as a service by the installer, so you can start and stop it from the Services applet, and you can enable it to be run at boot (Automatic Startup Type):



Name	Description	Status	Startup Type	Log On As
Fax	Enables you to ...		Manual	Network S...
FreeSWITCH Multi Prot...	FreeSWITCH se...		Manual	Local System
Function Discovery Pro...	The FDPHOST s...	Started	Manual	Local Service
Function Discovery Res...	Publishes this co...	Started	Automatic	Local Service

Once FreeSWITCH is started as a service, you connect to it and interact with it using the Windows equivalent of Linux `fs_cli`, named `fs_cli.exe`, found in the installation root directory (in our case, `C:\Program Files\FreeSWITCH`). Double-click on it:



```
Anthony Minessale II, Ken Rice,
Michael Jerris, Travis Cross
FreeSWITCH (http://www.freeswitch.org)
Paypal Donations Appreciated: paypal@freeswitch.org
Brought to you by ClueCon http://www.cluecon.com/

Type /help <enter> to see a list of commands


[This app Best viewed at 160x60 or more..]
+OK log level  [7]
freeswitch@owner-HP>
```

Warm and fuzzy, isn't it?

Summary

In this chapter, we covered a lot of ground:

- On which platform FreeSWITCH can be installed
- Preferred Platforms
- How to Install FreeSWITCH on Debian Linux
- How to install FreeSWITCH on Microsoft Windows
- How to start FreeSWITCH as a background server (for example, daemon, service)
- How to connect to it for management via `fs_cli`

In the next chapter, we will delve into the features available in the demo configuration that has been installed by default.

Test Driving the Example Configuration

After a default installation, out of the box, you'll find that FreeSWITCH is already able to do an astonishing amount of things! We've put a lot of effort into writing the mother of all demos, and the configuration that gets installed by default is just that, a big FreeSWITCH demo. When we add a new feature to FreeSWITCH, when we write a new module, or when someone comes out with a novel, clever way to use FreeSWITCH, we add some lines to the example configuration. Please note, we'll repeat that again later, that the example demo configuration is not intended at all to be used in a production system. In it, you'll find a mine of snippets waiting for you to reuse, techniques to do things, and many different ways to organize information about users, gateways, applications, context, and scripts. It is an enormous repository of FreeSWITCH applied knowledge, and is working right at the end of installation. So, please use it until you feel acquainted with the basics of FreeSWITCH, and then come back to reading it when you're looking for a way to add a feature or execute a task.

In this chapter, we will cover the following topics:

- Important VoIP and FreeSWITCH concepts
- Using the FreeSWITCH command-line interface (`fs_cli`)
- Configuring a phone to work with FreeSWITCH
- Calling various extensions in the system

Important concepts to understand

FreeSWITCH is the ultimate toolset for any kind of RealTime Communication service and action: from WebRTC-enabled videoconference cum chat & screen sharing to enterprise unified communication, from multitenant PBX in the Cloud to Carrier Grade VoIP Media Server. This implies a lot of flexibility. FreeSWITCH can do a great number of things, has lot of ready-made features, and many tasks can be accomplished by FreeSWITCH in more than one way.

We implemented many ways to harness this flexibility. One important aspect is that FreeSWITCH follows the good practice of *behaving as expected*. There are many conflicting requirements in the Real Time Communication world. Also, there are a lot of software and devices out there, and most of them have their own quirks. They bend and extend the official protocols (for example, IETF's RFCs). Or, they are plainly bugged, wrong, and subtly incompatible. Remember, our interoperable world is the heir of the many proprietary worlds where each vendor was (no more?) trying to lock in his customers with features and implementations not compatible with those of the competition. FreeSWITCH implements a policy of being very open-minded and forgiving in what it accepts as input,and of being very strict and protocol-observing in what it outputs. So, we're totally tolerant of the many interpretations of SIP, WebRTC, TDM, and all other protocols, standards, and media out there. We implemented a lot of workarounds to automatically and transparently manage all the quirks each device or software has. At the same time, we adhere meticulously to the letter and the spirit of the open standards, striving for universal compatibility and interoperability. Most of the time, FreeSWITCH features *just work* with each and any software and device you connect it to. And in the rare cases this does not happen transparently right out of the box, you can be sure there is a configuration tweak that can solve your problem; just check the docs, or ask online.

In this chapter, we'll embark in a journey through the example demo configuration and we'll get a taste of what is possible with FreeSWITCH, and a grasp of some basic concepts. The rest of this book will build up starting from here and expand/deepen the various aspects involved into implementing and managing your own production FreeSWITCH platform.

The most important concepts we will encounter throughout this book are those of XML configurations, regular expressions, and call legs.

XML Configuration

FreeSWITCH understands its configuration in terms of an all-encompassing XML tree. That XML tree is continuously searched and queried, as long as FreeSWITCH is running. Each time the FreeSWITCH core, or one of the modules, needs whatever information from the configuration, they ask (again and again) for that specific snippet of configuration. For example, let's imagine a SIP phone (or a WebRTC client) originates a call. The call is originated by user 1010@mydomain.org. When the first signaling of the originating call reaches FreeSWITCH, then the `mod_sofia` module (responsible for SIP) or `mod_verto` (responsible for... Verto!) will need to check the user authorization to make that call. The module will ask: *in the users' directory, what is the SIP password corresponding to user 1010@mydomain.org?* This question will be made each time via an internal FreeSWITCH API that will go and get the requested value from the indicated section of the XML configuration tree (in this case, the users' directory). There are many ways this task of fetching the XML configuration snippet can be accomplished: querying a database, via HTTP, through scripting languages, and so on. All of those methods return the requested XML snippet. The one used by the demo configuration out of the box is reading an XML main file from the filesystem, and then all the XML files included by it. This is done once at FreeSWITCH startup. So, each time you change one of those XML files, you must tell FreeSWITCH to *refresh* the XML tree it has built in its memory at startup. The command `reloadxml` is usually enough. The XML snippet fetching is executed each time information is requested by FreeSWITCH, but the fetching will be executed against the tree built in memory at startup, so it needs to *rebuild* it if you edit any XML file. Other methods such querying databases or HTTP servers are completely dynamic, they directly query the database or the web server each and every time.

Regular Expressions

Regular expressions (regexes) are almost omnipresent in FreeSWITCH configuration. They are "formulas" used to describe a string of text. For example, with a regular expression (regex) you can denotes all of: "cats", "cat", "supercat", "cAt" and "anycAts". That regex would be: ^.*c[aA]t.?$.. Explanation: caret ("^") means the beginning of the string. Then the point means "whatever character". The asterisk denotes an indefinite quantity (zero or more) of the preceding character - in this case of the "whatever character". The character "c" denotes... the character "c". Then the square brackets (and their content) denote a character that can be one of the list enumerated inside the square brackets (so, in this case the character can be "small a" or "capital a"). Then the character "t" is a placeholder for... the character "t". Then the point for "whatever character".

Then the question mark that establish the quantity of the preceding character in "zero or one instances", in our case means that after "t" can be one optional (eg, one or zero instances) of "whatever character". The dollar sign means the end of the string. Obviously you can have simpler regexes: "cat.*" would describes both "cat", "cats", "catharsis", etc.

Also, you can select one or more substrings from a string, by applying a regex to that string. Suppose you have the string "francesca bella ballerina" (eg: francesca, space, bella, space, ballerina), you can apply the regex "([a-z]*) ([a-z]*) ([a-z]*)" to that string. That regex (if successful) will split the string into three tokens, each one composed by small characters a to z. Eg, you have split using the space character as token delimiter, and you retained the sequences of a-z characters. Those sequences (that is, the tokens you split from the original string) are automatically put inside the $1, $2, $3 placeholders. So, you can use those placeholders and another regex to build a different string: "$1 $3 molto $2". That would results in the new string: "francesca ballerina molto bella", and she's actually a very beautiful dancer.

This micro intro gives you an idea of how powerful regexes can be for FreeSWITCH configuration. You can describe patterns of extensions, DIDs, users, callers, callees, gateways, etc. You can select, slice, dice and mix them. Logic can be built if that regex matches (eg, if it describes the string) or if that other regex is unsuccessful (it does not match the string). We'll see much more about all this.

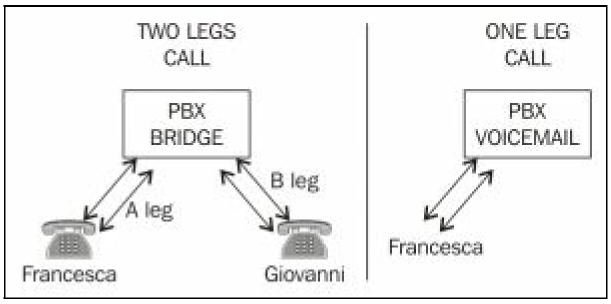
Call Legs

Call legs are what we people in the Real Time Communication field live and die for. It is a concept that can be confusing at first, but will become a second nature as time goes and you continue to work with FreeSWITCH.

Let's start with an example: we are in an office, and we have a PBX. Francesca wants to phone Giovanni. She dial Giovanni's extension. A call is established between Francesca's phone and the PBX, Francesca will hear the ringback (or a message if Giovanni is away). This call, from Francesca to the PBX, is known as "A" leg in telco jargon. Then, the PBX knows from its configuration that it must connect this A leg to Giovanni's phone, so Francesca and Giovanni can talk. PBX will originate a new call, known in lingo as "B" leg, from itself to Giovanni's phone. If Giovanni answers his ringing phone, PBX will join ("bridge" in telco lingo) the A leg with the B leg, and let the media (audio, video, whatever) flow back and forth from Francesca to Giovanni.

So: from the point of view of FreeSWITCH "A leg" is the original, initial, inbound, incoming call. "B leg" is the derivative, PBX originated, outbound call. From the point of view of the end user (Francesca), her call is successful (that is, she talks with Giovanni), if PBX successfully bridges A leg and B leg, after B leg has been answered by Giovanni. They can talk, eventually!

Let's suppose Giovanni is not there, and Francesca gets connected to the voicemail system (so, she is still only in touch with the office PBX): this is a "call" that is composed only by an "A leg", there is only one set, the inbound Francesca<=>PBX, of media streams.



There are cases where there are more than two legs, for example if Francesca calls Giovanni, and then she invites a third person to join, in a three-way call. In this case the lingo goes: from Francesca to the PBX, is the "A leg". All others legs that are originated because of Francesca call, in this case both the call leg from PBX to Giovanni and the call leg from PBX to the third person, all those are B legs (that is, there is no concept of a "C leg", "D leg", etc. They are all "B" legs).

IVRs, Interactive Voice Response (Systems)

IVRs are those voice (and soon video) menu systems we're so used to when we try to reach the customer care of our cellphone carrier, like "For knowing our latest offers, please press 1, for listening to the Xmas greetings from our CEO, please press 2, to speak with our technical department please press 3, to be transferred to our administrative department for billing and invoices, please press 4. To replay this menu, please press 5". OK, IVRs can be obnoxious, but are also useful.

A call to an IVR is at least initially a one leg call, but after a way can become a two or three legs, if you are transferred to an operator, that maybe after a while invite a sales representative to the call.

IVRs "interactivity" with the caller is almost universally based on the caller to press digits on the dialpad. There is also a growing use of ASR (Automatic Speech Recognition) systems, that are able to understand what the caller says. This recognition is usually made in a limited knowledge domain, for example, after asking "from what State are you calling from?", an ASR enabled IVR can be able to understand the caller saying the name of the State. In the future will probably be released ASR providing good reliability in "free form" conversation, but technology' is not yet there.

In any case, the IVR logic is implemented as a tree, where from the initial root the caller is able to navigate the branches back and forth by choosing (through the dialpad or by voice) an option between those that are presented to him.

Extensions, Contexts and Profiles

This is another source of great confusion. If you are new to FreeSWITCH, forget about the meaning you used to associate to the word "extension". In FreeSWITCH "extension" is an immaterial concept. It can be understood as "a group of instructions into a context". OK, this leave us with the problem of what a context is. A context is a named collection of extensions. So, in a way, we have a circular definition here, in which contexts and extensions define each others.

Bear with me and in a couple of paragraph we'll be over all this.

FreeSWITCH has "profiles". A profile is a collection of configurations that is related to a specific IP address and port couple. For example 192.168.1.12:5060 is a profile, 192.168.1.13:5060 is another profile, 192.168.1.12:5061 is a third profile. Each one profile can contains configurations that are completely different from that of other profiles. One very important such configuration contained in a profile is to which context the incoming call will go. A call is always incoming to a profile, eg, to an address:port couple. So, let's say the address:port couple is the enter gate to a specific profile. Once the call has entered the profile, first thing FreeSWITCH do is to look up in that profile the configuration that defines which context the call must go into.

Let's say the call is incoming at 192.168.1.12:5060. We called this profile "myprofile", and in the XML configuration of that profile is defined that the context to be used is "mycontext". This means that the incoming call will only be processed by the extensions (instructions sets) contained in the "mycontext" extensions collection (context). That is, the incoming call will go in the context defined in the profile, and it has no way to access anything that is outside that context.

Context is the jail of the call. This has obvious security, logical, and practical advantages: for example the calls that are incoming to a profile that is accessible by our coworkers from the internal LAN will be directed to a context that gives the feature of internal office calls, and allows also for reaching the PSTN, because we want our coworkers to be authorized to place national and international calls. On the other hand, the calls that are incoming to a profiles with a public Internet address, that is the calls that are reaching us passing through the router from the outside of our internal LAN, those calls are directed to a context that only allows for the call to be connected to one of our coworkers' phone. This specific profile will have only the feature to allow for external incoming calls to reach an internal phone, without any possibility to originate PSTN calls. That is, strangers cannot use our paid gateway to reach the PSTN.

So, different address:port couple, different profile, different context. It is actually so simple!

Now is time to see what are those "extensions". We know from above each extension is a set of instructions. When a call is incoming into a context (a context is a collection of extensions) the most important thing in each extension is: do this extension apply to the incoming call? Or, from the point of view of the call: do I enter this extension? Do this extension apply to me?

Each extension has one property, the "condition", that defines if that extension applies or not to the call. Eg, if the call will enter that extension, and pass through the steps defined by the instructions contained into the extension. An extension can be viewed as an instructions container, and the call must satisfy the extension "condition" to enter it and pass through the instructions inside it.

So, extension==list of instructions. To enter an extension, a call must satisfy a "condition".

And a "condition", what is a condition? A condition is a regular expression (regex) to be applied to one attribute of the call. If that regular expression matches (eg if it successfully describes the attribute of the call) the call will enter the extension and will execute the instructions.

Let's see an example. A call is incoming on the 192.168.1.12:5060, "myprofile". In "myprofile" XML configuration is defined "mycontext" context. In "mycontext" are specified many extensions, each one of them with its own "condition".

The first one of those extensions has a very simple condition: regex "^1010$" to be checked upon the "destination_number". So, this regex will be successful, it will matches, if the "destination_number" attribute of the call is the exact string "1010" (caret means string beginning, dollar sign means string end. So, "1010" is exactly what must be between begin and end of the string).

If the incoming call was meant to be directed to the destination number 1010, then the call will enter this extension and execute the instructions therein.

Those instructions are called "actions" in FreeSWITCH lingo, and can be whatever. For example: create a variable and assign a value to it. Read another variable, apply a regex to it so it convert to all capitals, create another variable and assign that value to it. Answer the incoming call, eg go off hook. Play a prerecorded message to the caller. Originate another, new, outbound call, and then join it (eg: bridge it) with the incoming call, so caller from incoming call and callee from outbound call can talk to

each other. Hangup the call. Query a database. Access a web server. Read or write a file. etc etc.
The actions (that are always contained inside an extension, "loose" actions do not exist) can truly do anything.

All the rest, profiles, contexts, extensions, conditions, regular expressions, are neat ways to have your actions pretty well organized, and to build logics and flows of executions.
Also, a call can matches many extensions (eg, can satisfy the condition of more than one extension) and in many cases will be allowed to enter many or all of the matched extensions, and to execute the actions contained by each one of those extensions.
We'll see much more about that.

Variables

Variables in FreeSWITCH are associated to almost anything. You can think at variables as attributes of something. For example, before we defined the "destination_number" as an attribute of an incoming ("A leg") call, and we used it as a string to which we applied the "condition" (regex) of an extension, to decide if that extension's actions will be executed by the call, or if the extension (and its actions) has to be skipped because its condition "do not matches" the string contained by the variable "destination_number". "Variables" is the official name for all those call or session attributes.

For example, an incoming call has a big number of associated variables, referred to as "channel variables", or "call leg variables", or simply "variables". Those channel variables spans from "destination number" (we saw this one) to "caller source network", from "which audio codec is in use" to "at what time the call has been received", from "what kind of user agent (eg, phone model) is calling" to "what SIP headers were part of the incoming INVITE network packet". And not only those let's say "natural", "descriptive", "objective" variables. We can also create and assign all kind of arbitrary variables, and associate them to our call. For example the call has been originated by the "sales group". Or the call is directed to the "support group". Or, the cost per minute of the call is 5 cents. Or, the max allowed duration for this call is 600 seconds. etc etc

All of the variables (both "original" and "assigned") can be checked by regular expressions in an extension's condition. So, we can have a logic that if it is a working day, during working hours, and the call is destined to the sales group, then made John's, Rehan's and Giovanni's phone ring, and connect the call to the first one that answers.

Caller profile Variables

Some variables we always find in a call (we also say those variables are part of the "caller profile"):

- username
- dialplan
- caller_id_name
- caller_id_number
- callee_id_name
- callee_id_number
- network_addr
- ani
- aniii
- rdnis
- destination_number
- source
- uuid
- context

Some of the variables' names in the caller profile may be obscure to you right now, but you'll find plenty of information later in this book, and in all the online documentation (we developers too we check the online docs, we do not memorize all and every variables, they're hundreds ;)).

What's specific about the caller profile variables' is that you use them in the "field" of a "condition" by their plain names, without special syntax.
Why we specify that? Because variables in FreeSWITCH are indicated with the special syntaxes ${varname} or $${othervarname} (we'll see later the difference between the two syntaxes).
So, as a bonus for your fingers, you can save some typing and have a more pleasant and readable dialplan when you use caller profile variables in the condition's field:

```
<extension name="Local_Extension_Skinny">

<condition field="destination_number" expression="^(11[01][0-9])$">

<action application="bridge" data="skinny/internal/${destination_number}"/>

</condition>
```

```
</extension>
```

As you can see we wrote "destination_number" in the condition's field, while we had to type it entirely as "${destination_number}" when used elsewhere (here, in an action executed if the condition is matched).

All caller profile variables are read only (that is, you cannot change them) and are assigned automatically at the beginning of the call. They describe the call itself.

"Normal" Variables

Most FreeSWITCH variables are indicated by the syntax ${varname}. Variables can be created, assigned and re-assigned. Actually, assigning a value to a variable creates it, if it does not already exists. If it exists, it reassign it to the new value.

```
<action application="set" data="mystring='Rehan Allahwala'"/>

<action application="set" data="mystring2=SuzanneBowen"/>

<action application="set" data="mystring2=${mystring}"/>
```

The application "set" assign variables. Here we see how it works in dialplan "actions".
When we create or reassign a variable, we use its name (without the ${} syntax). Inside the "data" argument of the "set" action, we use the equal sign (=) to separate the variable name from its (new) value. The single quotes are used to enclose a value when it contains spaces.

When we want to use the variable's value, we use the dollar syntax.
So, in previous example, we first assigned a value to the variable named "mystring" (creating it if it does not exists). Then we assign a value to the "mystring2" variable. In the last line we reassign "mystring2" variable, giving it the value contained into the "mystring" variable. The value of ${mystring2} is now 'Rehan Allahwala'.

All variables can be used by all modules and parts of FreeSWITCH. What does it means? That, for example, when a user that is member of the "sales" group originates a call, you can set a variable on that call. You can then have one of the Call Detail Record modules to track that variable, putting it in CDRs and allowing your company to identify cost centers.

Variables can be created by us out of nothing, or they can be automatically created by FreeSWITCH, from information inherent the calls, the environment, the time of day, etc.

Some variables reflect values from the various configuration files. Most of the variables from configuration files are "writable". For example, we can set a default value for our Station Identifier (that is, assign our own company name) when sending faxes, and then we can override it reassigning it to "ACME co" in dialplan when we use our same FreeSWITCH server as send fax service for a customer.

"Preprocessor" Variables

Those are the variables indicated by the syntax $${varname}. In a way, those are not actual variables, but they act as read-only ones. You assign them in one of the FreeSWITCH configuration files (/usr/local/freeswitch/conf/vars.xml), and then they are literally substituted at startup when encountered. This is exactly the same as the #define construct in C preprocessor, or a "Find and Replace" command in a text editor.

```
<X-PRE-PROCESS cmd="set" data="hold_music=local_stream://moh"/>
```

In this example, by editing the vars.xml configuration file you alter the value that will be substituted in all configuration files (dialplan included) when "$${hold_music} " is encountered. If you then write in dialplan:

```
<action application="set" data="mystring=$${hold_music}"/>
```

it would be exactly the same as if you wrote:

```
<action application="set" data="mystring=local_stream://moh"/>
```

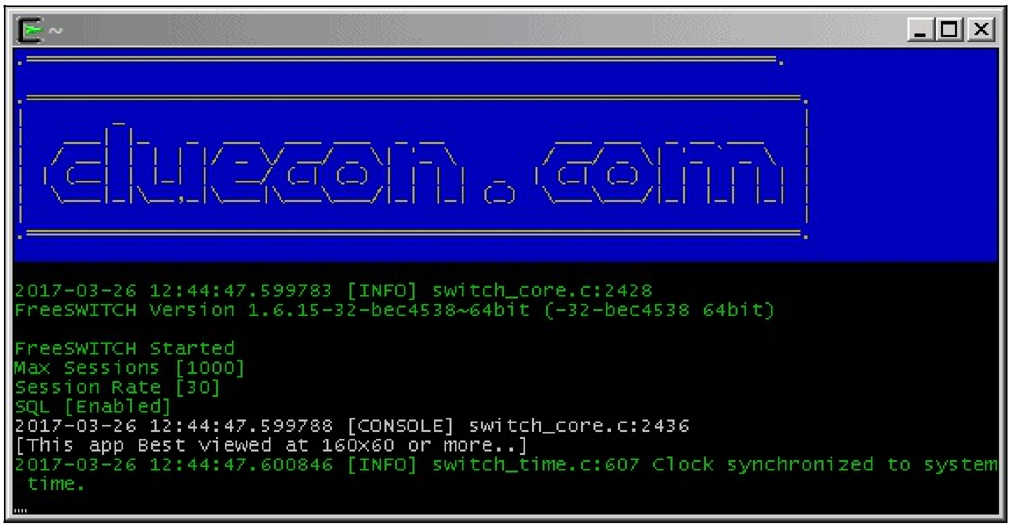
Because this is actually a text search and replace procedure that is done once at FreeSWITCH startup, if you change a preprocessor variable in vars.xml you MUST restart FreeSWITCH in order to activate the changes (in this case reloading the configuration without restarting FreeSWITCH is not enough).

FS_CLI and Console, Controlling FreeSWITCH

There are so many ways to control FreeSWITCH in real time and make it to do what you want. You can write applications, scripts, have FreeSWITCH interact with databases and legacy systems... Whatever means you'll use to integrate your platform, you'll find yourself coming back now and then to the FreeSWITCH Command LIne (CLI). And a lot of FreeSWITCH admins use only CLI for their day to day tasks.

```
/usr/local/freeswitch/bin/freeswitch -u freeswitch -g freeswitch -c
```

If you start FreeSWITCH in foreground (eg, using the "-c " option), after a brief pause all the startup messages are scrolled away, and you end up at the command of the FreeSWITCH Console:



```
2017-03-26 12:44:47.599783 [INFO] switch_core.c:2428
FreeSWITCH Version 1.6.15-32-bec4538~64bit (-32-bec4538 64bit)

FreeSWITCH Started
Max Sessions [1000]
Session Rate [30]
SQL [Enabled]
2017-03-26 12:44:47.599788 [CONSOLE] switch_core.c:2436
[This app Best viewed at 160x60 or more..]
2017-03-26 12:44:47.600846 [INFO] switch_time.c:607 Clock synchronized to system
 time.
```

Much more often, let's say almost always if you're not debugging startup messages, you start FreeSWITCH in a different way, so it goes into background as a proper server (daemon), without a controlling terminal.

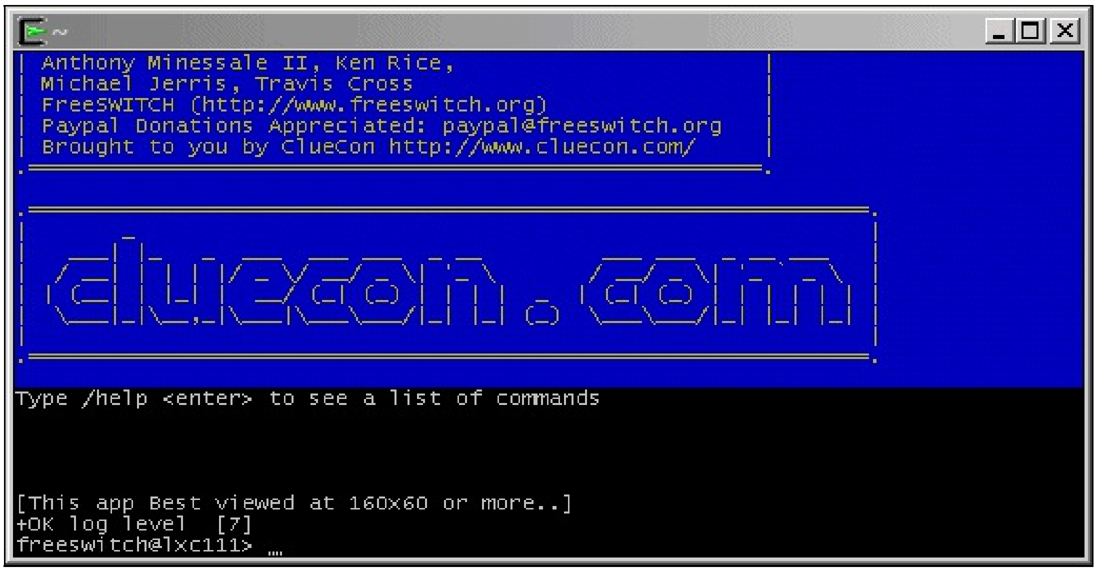
A production system is usually started like that:

```
    /usr/local/freeswitch/bin/freeswitch -u freeswitch -g freeswitch
    -nonat -ncwait
```

After some seconds, this command ends, we're told of the FreeSWITCH Process ID and then... nothing happens. Yes, is OK, FreeSWITCH is running in background and ready to accept and manage calls, and we can interact with it using the fs_cli utility, mimicking the native FreeSWITCH console.

```
/usr/local/freeswitch/bin/fs_cli
```

Using fs_cli on a system where FreeSWITCH has already started brings us immediately in control:



```
| Anthony Minessale II, Ken Rice,                        |
| Michael Jerris, Travis Cross                           |
| FreeSWITCH (http://www.freeswitch.org)                 |
| Paypal Donations Appreciated: paypal@freeswitch.org    |
| Brought to you by ClueCon http://www.cluecon.com/      |
```

```
Type /help <enter> to see a list of commands


[This app Best viewed at 160x60 or more..]
+OK log level  [7]
freeswitch@lxc111>
```

Exiting from Console kills FreeSWITCH

Most important difference between fs_cli and the actual FreeSWITCH console (eg, when you start FreeSWITCH in foreground) is that the only way to exit from the actual FreeSWITCH console is by shutting down FreeSWITCH. This is tricky, because for almost all other aspects they (fs_cli and FreeSWITCH actual console) look and behave in the same way. Problem is, if you exit fs_cli, FreeSWITCH continues to run in background, accepting calls, etc. And you can always re-launch fs_cli and reattach to it (like an ssh terminal to a remote server). BEWARE, if you exit the actual FreeSWITCH console you are terminating FreeSWITCH itself, and you will need to restart it.

Some useful CLI commands

You can use most of those commands both from the actual console and from fs_cli. After each command, press ENTER key.

command	meaning
... (three consecutive points)	exit fs_cli. If used from console it shutdown FreeSWITCH
ctrl-d	exit fs_cli
fsctl shutdown	shutdown FreeSWITCH
hupall	hangup all calls
fsctl loglevel [0-7]	change which log messages are visualized
status	tells some basic statistics
version	which code base
show channels	visualize all call legs one by one, individually
show calls	visualize all call legs grouped by complete bridged calls (A+B)
help	strangely confusing and overwhelming until you know what it is printing about. Try it yourself!

You will end up using often "fsctl loglevel" command, that allows you to choose the "level" of FreeSWITCH logging. FreeSWITCH server is able to generate an enormous amount of information on its own inner working, most of which is of no interest during normal running. You will normally use a loglevel of 3: all messages categorized as "ERROR" or even more important (eg, level 2, "CRITICAL") are visualized. Many admins like to run their FreeSWITCHes at loglevel 4, visualizing messages starting at level "WARNING". But when you want to understand the cause of a problem, or report a bug, you pump up the loglevel to 7, "DEBUG" and then copy and paste all that will scroll by in your terminal (or that gets written in logfile).

Log lines are by default colorized by loglevel, you will soon train your eyes to spot the red (ERROR or more important) and purple (WARNING) messages from the less disquieting blue (NOTIFY), green (INFO) and yellow (DEBUG) lines.

Configure SIP and make the first calls in demo dialplan

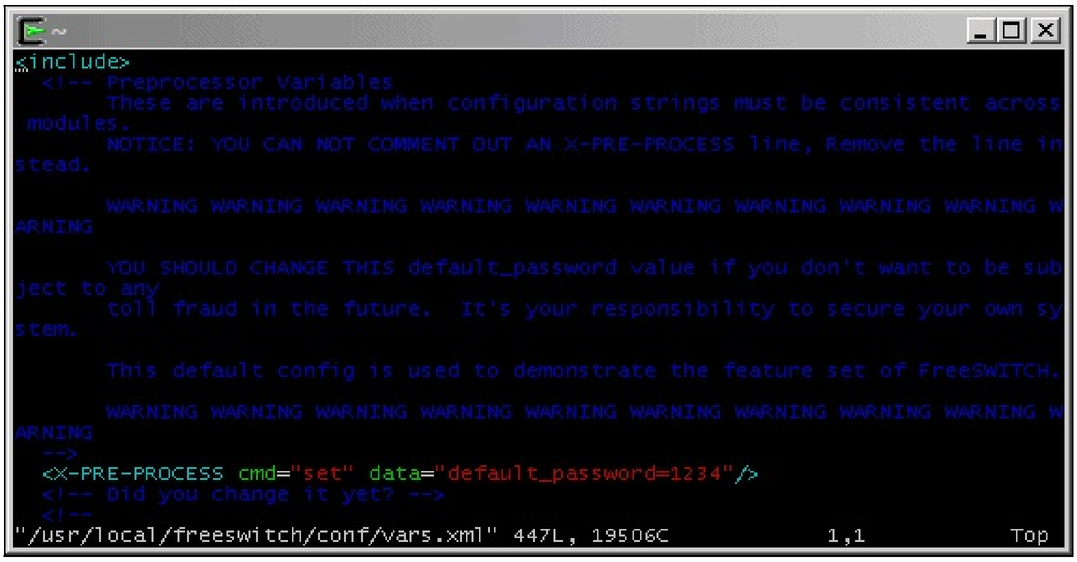
We're almost ready for the real emotions, when something rings and blinks. Right after installation FreeSWITCH gets a demo example configuration complete with a lot of features. A very complex dialplan has been laid out for you to play with, and feel the power of FreeSWITCH.

NB: That dialplan is just a demo, don't use it in production. Is overcomplex and has too much features, you do not know it, and so is outside your control.

Demo example dialplan and configuration are just that: demo examples. When you finish play with them, delete them and build something that's right for your use case. If you are not able, ask someone else, a friend or a consultant, to do it for you. Any other behavior would be stupid and dangerous from a legal and financial standpoint. You have been warned! :)

Change demo example password

First thing to do with a new installation is to change the default password for the demo users (users, passwords, and all other concepts will be clarified in Chapter 4,*User Directory, SIP, and Verto* and during the rest of this book). Edit the file /usr/local/freeswitch/conf/vars.xml :



```
<include>
  <!-- Preprocessor Variables
       These are introduced when configuration strings must be consistent across
 modules.
       NOTICE: YOU CAN NOT COMMENT OUT AN X-PRE-PROCESS line, Remove the line in
stead.

       WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING W
ARNING

       YOU SHOULD CHANGE THIS default_password value if you don't want to be sub
ject to any
       toll fraud in the future.  It's your responsibility to secure your own sy
stem.

       This default config is used to demonstrate the feature set of FreeSWITCH.

       WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING W
ARNING
  -->
  <X-PRE-PROCESS cmd="set" data="default_password=1234"/>
  <!-- Did you change it yet? -->
  <!--
"/usr/local/freeswitch/conf/vars.xml" 447L, 19506C              1,1           Top
```

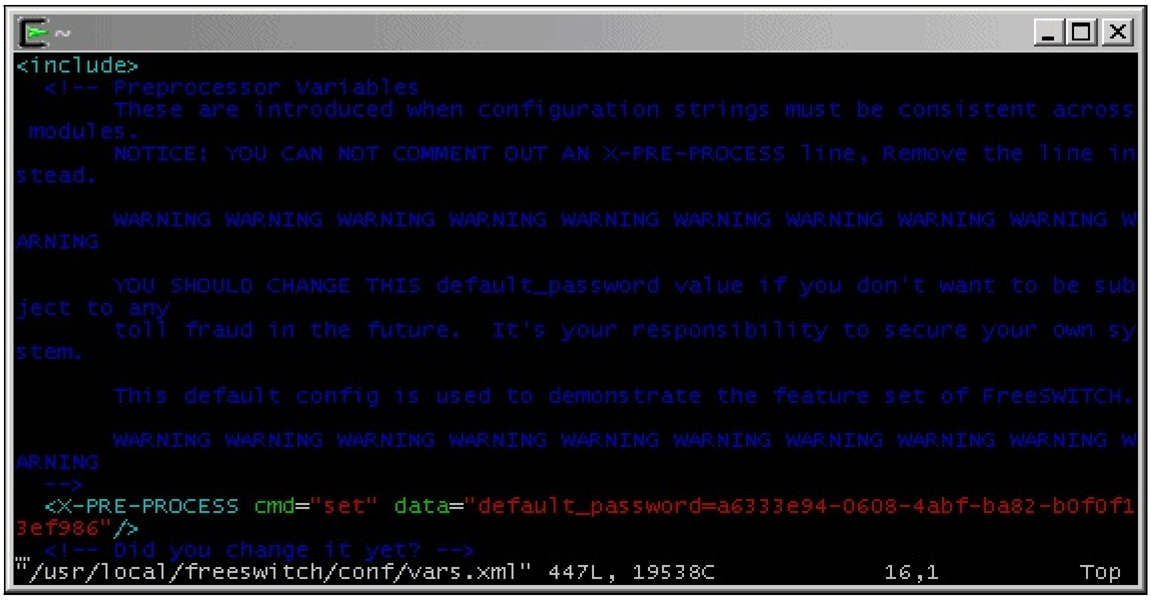
Change "1234" to something else. If you change it to something easy, or only moderately difficult, you will be hacked in next 20 minutes. It probably does not matter financially until you add a **PSTN gateway** they can use **atYOUR EXPENSE**, but please consider the legal implications if someone uses your server to originate another kind of traffic, maybe some **terrorist** SIP calls? Then, take your time explaining it all to the FBI during the next 3 years. Don't rush, they've got time. So, choose a random string of robust length. In doubt, install and use uuidgen, it makes very good passwords. In Debian, from root command line:

```
apt-get install uuid-runtime
```

Then you execute uuidgen:

```
uuidgen
   a6333e94-0608-4abf-ba82-b0f0f13ef986
```

Is a good password, isn't it? Anyway, you don't have to remember it, you just cut and paste it:



```
<include>
  <!-- Preprocessor Variables
       These are introduced when configuration strings must be consistent across
 modules.
       NOTICE: YOU CAN NOT COMMENT OUT AN X-PRE-PROCESS line, Remove the line in
stead.

       WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING W
ARNING

       YOU SHOULD CHANGE THIS default_password value if you don't want to be sub
ject to any
       toll fraud in the future.  It's your responsibility to secure your own sy
stem.

       This default config is used to demonstrate the feature set of FreeSWITCH.

       WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING WARNING W
ARNING
  -->
  <X-PRE-PROCESS cmd="set" data="default_password=a6333e94-0608-4abf-ba82-b0f0f1
3ef986"/>
  <!-- Did you change it yet? -->
"/usr/local/freeswitch/conf/vars.xml" 447L, 19538C           16,1          Top
```

Then save the file, and restart FreeSWITCH (this is a preprocessor variable, you MUST restart FS for the change to take effect).

From fs_cli, or console, type:

```
fsctl shutdown
```

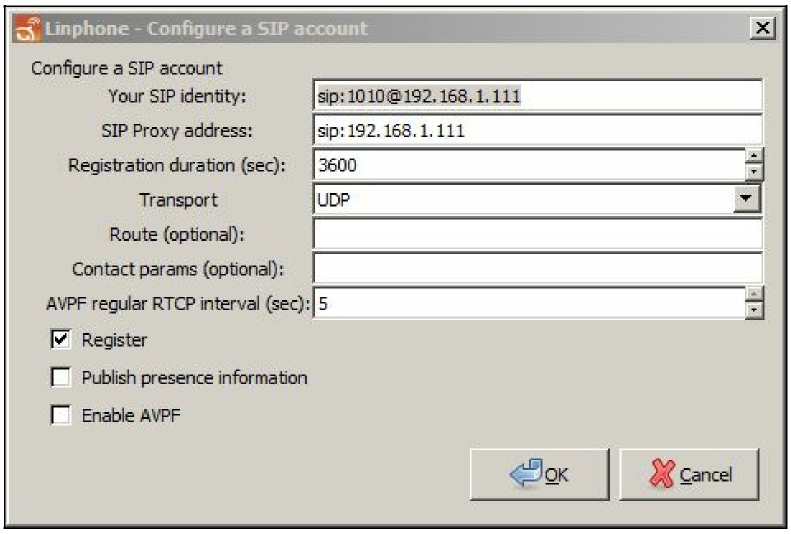
And then restart FreeSWITCH as you've done previously.

Configure Linphone on a Desktop

Linphone is a wonderful SIP application, completely cross platform (Windows, OSX, Linux, Android, iOS), battle tested in millions of installations, open-source, and mainly developed and commercially supported by the nice people at the French company Belledonne.

Example demo users are named by the 20 consecutive numbers 1000-1019. That is, 1001, 1002...1019. Their collective password is the one we just edited in vars.xml: a6333e94-0608-4abf-ba82-b0f0f13ef986. To configure a SIP account in our Linphone we need the IP address of our FreeSWITCH server, in our case the address is 192.168.1.111. Please note we will run both FreeSWITCH server and all the SIP clients on the same local network, and we open all firewalls on all machines involved. You'll learn later how to do more complex topologies.

Create a SIP account in Linphone (Menu -> Options -> Preferences -> Manage SIP Accounts), user must be between 1000 and 1019, in our case we choose 1010 (note you MUST write "sip:" before SIP identity and SIP Proxy Address):

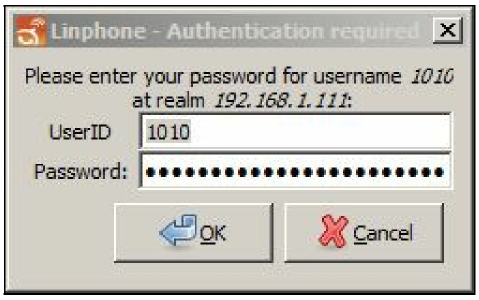

Linphone - Configure a SIP account

Configure a SIP account

Your SIP identity:	sip:1010@192.168.1.111
SIP Proxy address:	sip:192.168.1.111
Registration duration (sec):	3600
Transport	UDP
Route (optional):	
Contact params (optional):	
AVPF regular RTCP interval (sec):	5

[✓] Register
[ ] Publish presence information
[ ] Enable AVPF

This is a Windows example, on other platforms the configuration interface will be more or less the same. As soon you click OK, another dialog will ask you for the password. Be ready to cut and paste the password in the dialog, before it timeout. If it does timeout, edit the SIP account again, save it, and you will be asked again for the password.

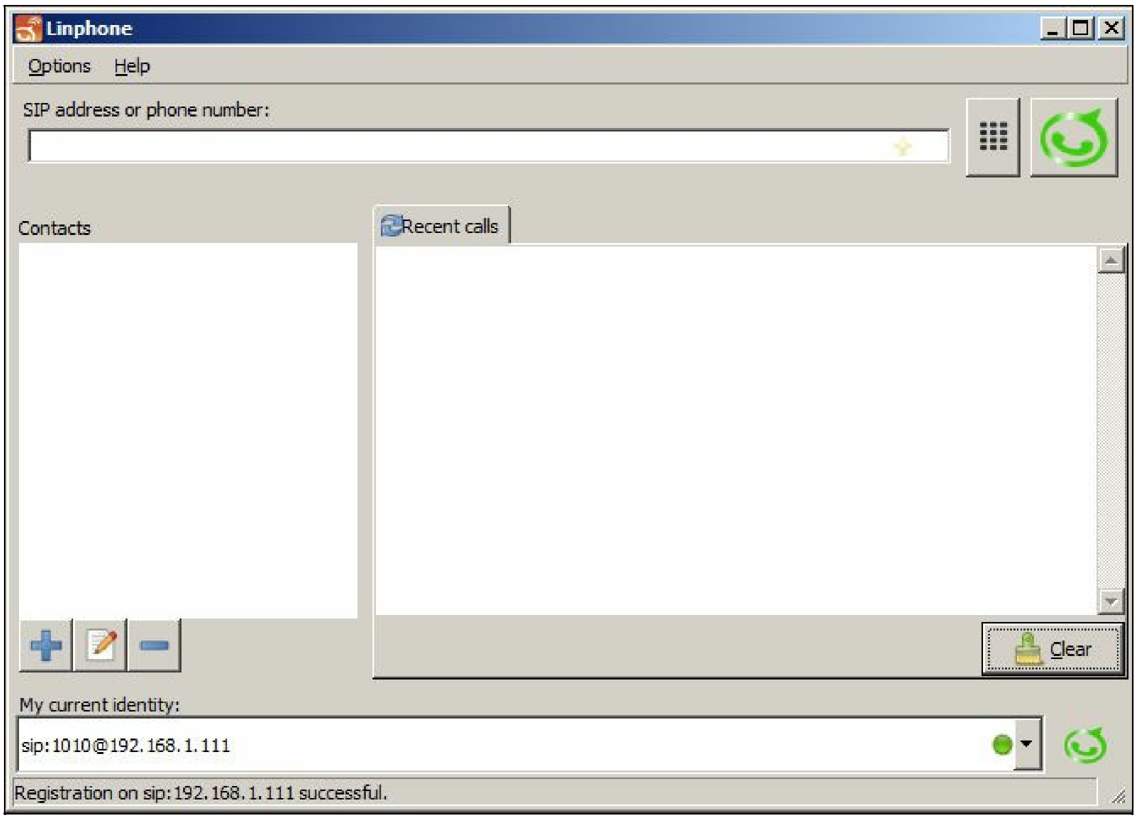


Please enter your password for username *1010* at realm *192.168.1.111*:

UserID 1010

Password: ●●●●●●●●●●●●●●●●●●●●●●●

If all has worked well, your Linphone will register to the FreeSWITCH server and will be ready to make and receive calls.

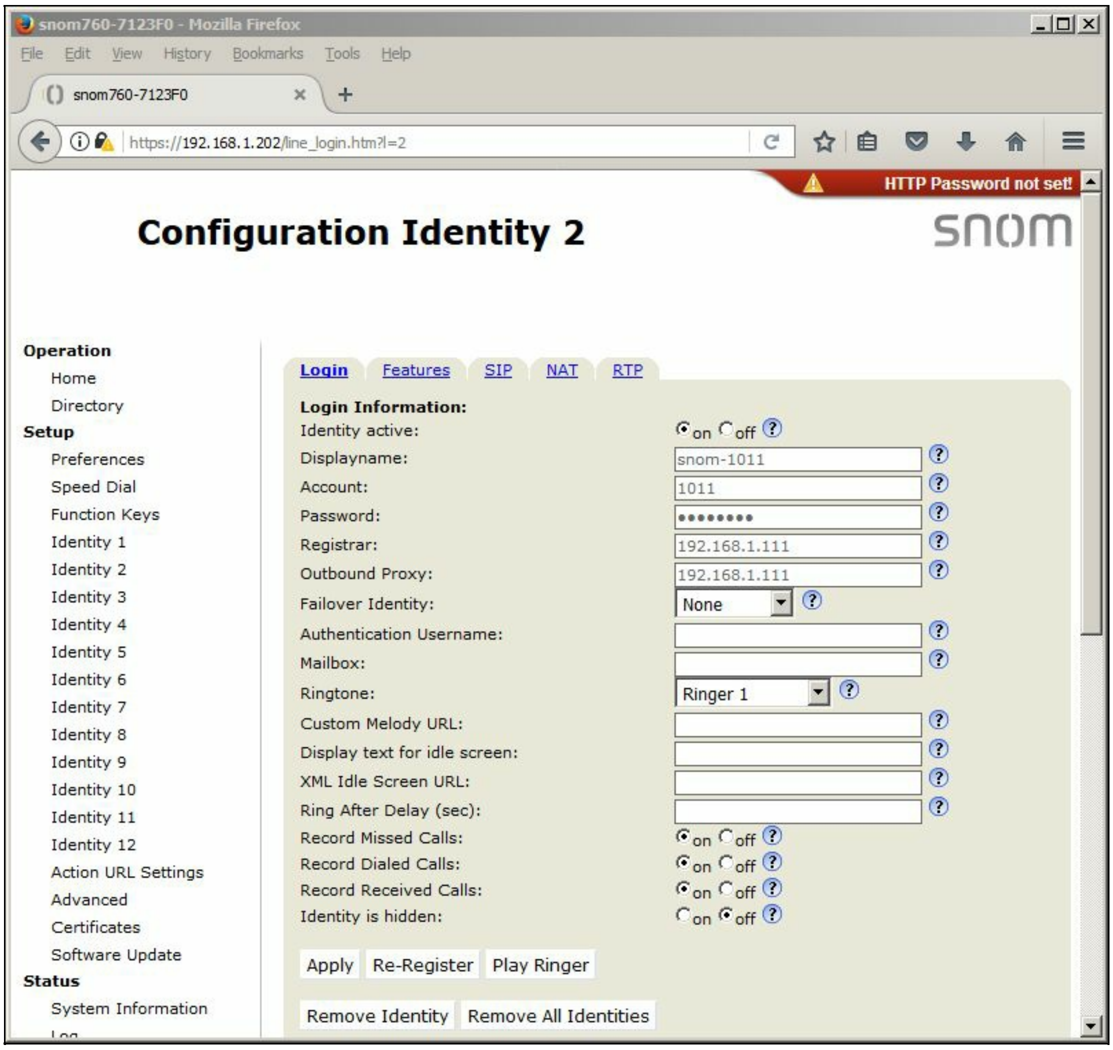


sip:1010@192.168.1.111

Registration on sip:192.168.1.111 successful.

Configure a SNOM SIP "real" phone

When a SNOM (or other SIP) phone boots, it displays its own IP address. Use a web browser and configure a SIP account ("Identity" in SNOM parlance)


snom760-7123F0 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
snom760-7123F0
HTTP Password not set!
snom

https://192.168.1.202/line_login.htm?l=2

Configuration Identity 2

Operation
- Home
- Directory

Setup
- Preferences
- Speed Dial
- Function Keys
- Identity 1
- Identity 2
- Identity 3
- Identity 4
- Identity 5
- Identity 6
- Identity 7
- Identity 8
- Identity 9
- Identity 10
- Identity 11
- Identity 12
- Action URL Settings
- Advanced
- Certificates
- Software Update

Status
- System Information
- Log

Login | Features | SIP | NAT | RTP

Login Information:

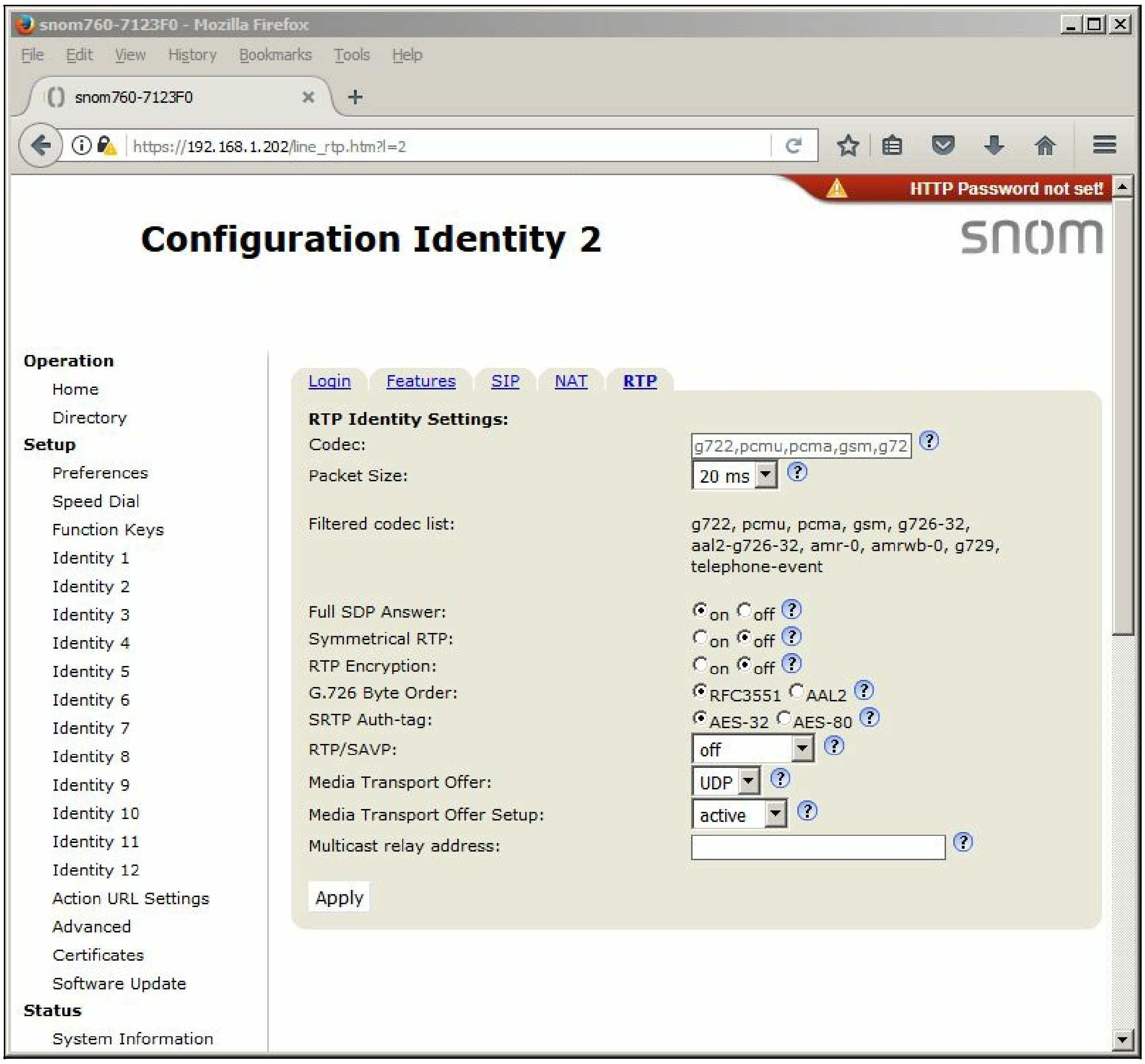
Identity active:	[✓ on] off
Displayname:	snom-1011
Account:	1011
Password:	••••••••
Registrar:	192.168.1.111
Outbound Proxy:	192.168.1.111
Failover Identity:	None
Authentication Username:	
Mailbox:	
Ringtone:	Ringer 1
Custom Melody URL:	
Display text for idle screen:	
XML Idle Screen URL:	
Ring After Delay (sec):	
Record Missed Calls:	[✓ on] off
Record Dialed Calls:	[✓ on] off
Record Received Calls:	[✓ on] off
Identity is hidden:	on [✓ off]

Apply | Re-Register | Play Ringer

Remove Identity | Remove All Identities

Then press "Apply", on the bottom.

Now, go to the "RTP" tab, and switch off RTP encryption (we'll look into encryption later in the book). Press "Apply".



Configuration Identity 2

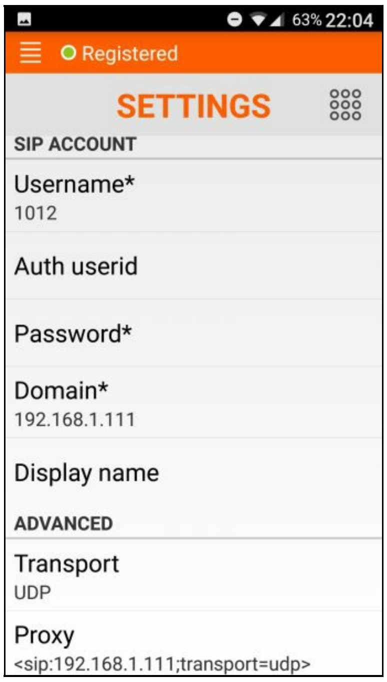
RTP Identity Settings:

Codec:	g722,pcmu,pcma,gsm,g72
Packet Size:	20 ms
Filtered codec list:	g722, pcmu, pcma, gsm, g726-32, aal2-g726-32, amr-0, amrwb-0, g729, telephone-event
Full SDP Answer:	[✓ on] off
Symmetrical RTP:	on [✓ off]
RTP Encryption:	on [✓ off]
G.726 Byte Order:	[✓ RFC3551] AAL2
SRTP Auth-tag:	[✓ AES-32] AES-80
RTP/SAVP:	off
Media Transport Offer:	UDP
Media Transport Offer Setup:	active
Multicast relay address:	

If all has worked well, the SNOM phone is now registered to the FreeSWITCH server. Don't forget to "Save" permanently the whole SNOM configuration when asked, or you'll have to begin again next time the phone boots.

Configure Linphone on Android

Create a new account by touching the menu, or the "Assistant". Fill in all the required fields.



Registered

SETTINGS

SIP ACCOUNT

Username*
1012

Auth userid

Password*

Domain*
192.168.1.111

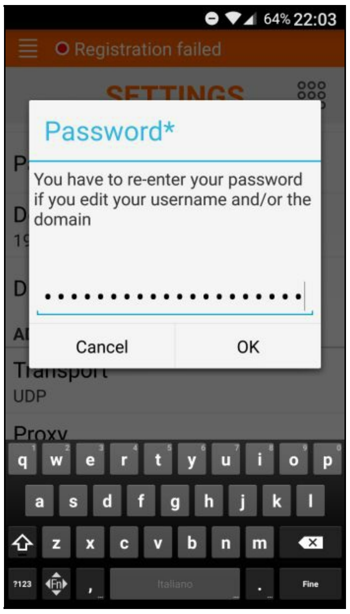
Display name

ADVANCED

Transport
UDP

Proxy
<sip:192.168.1.111;transport=udp>

The "Proxy" field will be filled in automatically for you. Then, click on the "Password" field, and insert the password (I put it in a file, transferred on the smartphone, and then... copied and pasted).



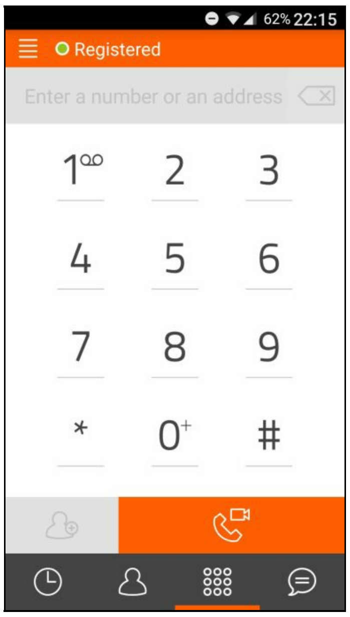
Registration failed

Password*

You have to re-enter your password if you edit your username and/or the domain

••••••••••••••••••••

Then click OK.



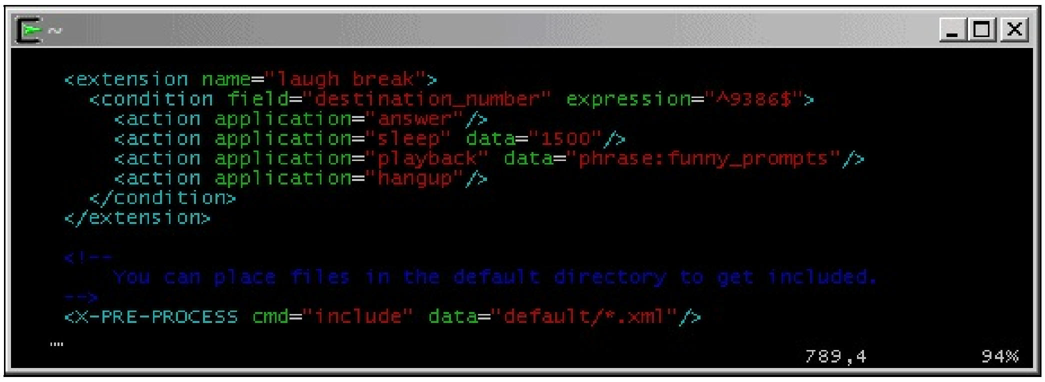
If all has worked well, the Android Linphone will register to the FreeSWITCH server, and will be ready at your commands.

And now... Fun, fun, fun with the Demo Dialplan

You'll find the example demo dialplan main file in /usr/local/freeswitch/conf/dialplan/default.xml . It makes for interesting reading. Obviously you'll not be able to understand most of the details if this is your first exposure to FreeSWITCH, but comments and your "gut feeling" will make you to figure out a lot of it.

9386, Funny Prompts

Open the dialplan main file with a text editor, and search for "9386".



```
    <extension name="laugh break">
      <condition field="destination_number" expression="^9386$">
        <action application="answer"/>
        <action application="sleep" data="1500"/>
        <action application="playback" data="phrase:funny_prompts"/>
        <action application="hangup"/>
      </condition>
    </extension>

    <!--
        You can place files in the default directory to get included.
    -->
    <X-PRE-PROCESS cmd="include" data="default/*.xml"/>
  ""
                                                          789,4          94%
```

You'll find it toward the end of the file. We see the "condition" is to exactly match the "destination_number" with "9386", no prefixes, no trailing digits (caret is beginning of string, dollar is end of string). If the destination_number (it is a variable in "caller profile", so we can use it in condition's "field" without the ${} syntax) of the incoming call matches 9386, FreeSWITCH will answer the call (eg, go off hook and begin the media streams packet exchange), sleep for a second and half, then play to the caller an entity called "phrase:funny_prompts" (we'll see later in the book that a "phrase" is a named collection of sound files). After all the playing, if the caller has not already closed the call, FreeSWITCH will hangup.

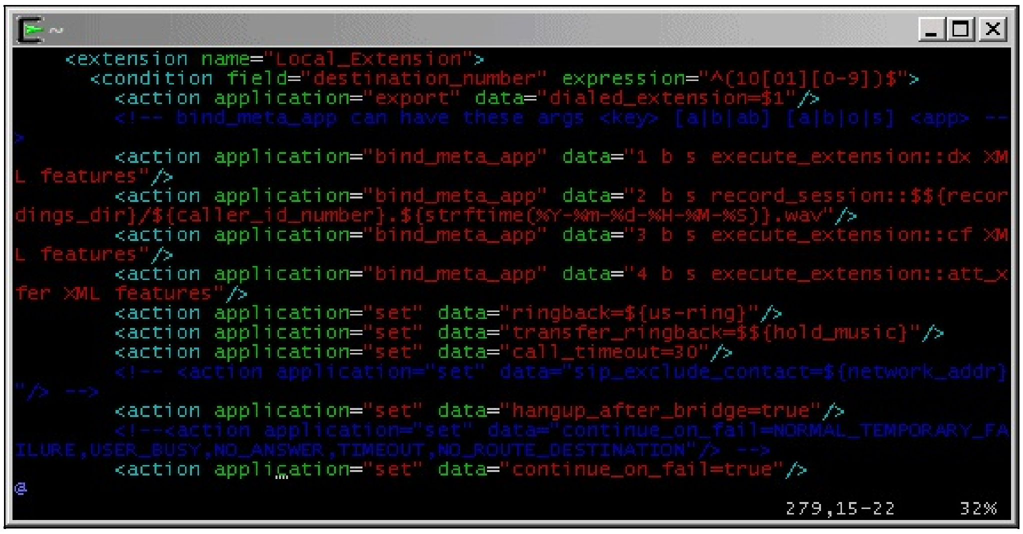
(Last line in the picture means that all files with an xml extension that are contained in the subdirectory "default" will be read in and "added" to the main dialplan. This is handy for organization, we'll see).

So, this is the first call to our freshly installed FreeSWITCH, let's call 9386 from one of our registered phones, and listen to a series of funny prompts played to us one after another.

1000...1019, Phone to Phone Local Call

The next natural step is calling from one of our phones to another. The "extension" dealing with this kind of call between demo users is very complex, because it packs a lot of features.

Open the example demo dialplan main file in /usr/local/freeswitch/conf/dialplan/default.xml with an editor, and search for "Local_Extension".



```
    <extension name="Local_Extension">
      <condition field="destination_number" expression="^(10[01][0-9])$">
        <action application="export" data="dialed_extension=$1"/>
        <!-- bind_meta_app can have these args <key> [a|b|ab] [a|b|o|s] <app> --
>
        <action application="bind_meta_app" data="1 b s execute_extension::dx XM
L features"/>
        <action application="bind_meta_app" data="2 b s record_session::$${recor
dings_dir}/${caller_id_number}.${strftime(%Y-%m-%d-%H-%M-%S)}.wav"/>
        <action application="bind_meta_app" data="3 b s execute_extension::cf XM
L features"/>
        <action application="bind_meta_app" data="4 b s execute_extension::att_x
fer XML features"/>
        <action application="set" data="ringback=${us-ring}"/>
        <action application="set" data="transfer_ringback=$${hold_music}"/>
        <action application="set" data="call_timeout=30"/>
        <!-- <action application="set" data="sip_exclude_contact=${network_addr}
"/> -->
        <action application="set" data="hangup_after_bridge=true"/>
        <!--<action application="set" data="continue_on_fail=NORMAL_TEMPORARY_FA
ILURE,USER_BUSY,NO_ANSWER,TIMEOUT,NO_ROUTE_DESTINATION"/> -->
        <action application="set" data="continue_on_fail=true"/>
@
                                                            279,15-22     32%
```

We'll not follow this extension actions step by step, it's too complex and advanced for this part of the book. But look at the condition. The regex in the "expression" field matches the "destination_number" value for 1000...1019, exactly the same series we know are the demo users. So, if from a phone you call one of the other phones, the incoming call leg (the A leg, the call you are originating from the phone) will reach FreeSWITCH, go to the dialplan, matches this extension, and the machinery inside this extension will check if the callee user phone is registered to FreeSWITCH, if it is registered FreeSWITCH will originate a new call leg (the B leg) from itself to the callee phone.

If the callee answers, FreeSWITCH will join (bridge) the two call legs and starts moving the media streams data packets back and forth between caller and callee. If the callee do not answers, or if its phone is not even registered, FreeSWITCH will connect the incoming call (the A leg) to the callee voicemail (so it will not originate a second leg, the voicemail is a FreeSWITCH module, no external entities are involved).

Now you know the plot, you can test the various scenarios, call the other phone, answer, don't answer, call a wrong number, call a non registered phone, call one of the standard users (1000...1019) for which you have not configured a phone. Have fun, already!

5000, Calling the Demo Example IVR

IVR, Interactive Voice Response, self attendant, personal attendant, voice menu, voice tree, is one of the things this bizarre society of last decades has made all of us extremely familiar with. From call centres to voice mail systems, from the wakeup call of our youth (our fathers' youth?) to airport information systems, we're used to interact with a (synthetic?) voice through the digits we press on the dialpad.

So, call 5000 from one of the phones and interact with a simple but complete IVR, with many options, menus and submenus. Some of the choices you can make from the demo IVR are also extensions in their own right.

For example, if you press "3" on the dialpad under the hood you'll be transferred to the "9664" extension, listening to music on hold. If you press "2", you'll be transferred to the "9196" extension, that will echo your voice back to you (useful to establish how much network delay for the back and forth voice trip).

We'll see more on how to build an IVR, and how the demo IVR has been built, later in the book. For the time being, let's play with it.

3000, Conference Call

If you call 3000 from different phones, they all will be connected to the same "conference room". There is no limit to the number of callers that can connect to the same conference rooms, and you can have any number of different conference rooms going on concurrently. Callers in the same conference rooms will be able to ear each others, but will not ears callers in different rooms. Also, conference rooms have a lot of features, moderators, optional pin authorization, etc. In the basic room we made up for the demo, you can play with your dialpad and see what's happen. More on conferences later in this book.

Example Dialplan quick reference

Consult the following table for a list of extension numbers and their functions:

Extension	Function
1000 - 1019	Local extensions
** + extension number	Intercepts a ringing phone (that is, call pickup)
2000	Samples call group: Sales
2001	Samples call group: Support
2002	Samples call group: Billing
3000 - 3399	Samples conference rooms
4000 or *98	Retrieves voicemail
5000	Demo IVR
5900	FIFO queue park
5901	FIFO queue retrieve
6000	Valet park/retrieval, manual
6001-6099	Valet park/retrieval, automatic

7243	RTP multicast page
0911	Group intercom example #1
0912	Group intercom example #2
0913	Emergency outbound conference example
9178	Example fax receive
9179	Example fax transmit
9180	Ring test, far end generates ring tone
9181	Ring test, send U.K.ring tone
9182	Ring test, send music as ring tone
9183	Answer, then send U.K ring tone
9184	Answer, then send music as ring tone
9191	ClueCon registration
9192	Information dump
9195	Delayed echo test

9196	Echo test
9197	Milliwatt tone (test signal quality)
9198	Tetris
9664	Music on hold

Summary

In this chapter, we were introduced to the default configuration of FreeSWITCH. Among the topics we discussed were the following:

- The important concepts behind how FreeSWITCH behaves when you make calls, and why
- Basic use of `fs_cli`, the FreeSWITCH command-line interface utility
- How to configure SIP devices to connect to FreeSWITCH using the predefined user accounts
- Testing the default dialplan by dialing a number of different extensions

We now turn our attention to another important aspect of FreeSWITCH: the user directory.

In the next chapter, we will take a closer look at the FreeSWITCH user directory.

User Directory, SIP, and Verto

SIP and Verto in FreeSWITCH both use the same User Directory machinery and concepts. FreeSWITCH's User Directory (or, in FreeSWITCH parlance, simply the "directory") is the central registry for all data related to users' authentication and authorization.
After a default installation, out of the box, you'll find that FreeSWITCH is already provided with 20 users, with a default password, each one of them belonging to one or more groups. FreeSWITCH is able to send calls to a specific user or to an entire group. Also, in User Directory you can set variables linked to the user, or to the group. Those variables are then used by FreeSWITCH both in managing incoming calls and when originating calls. We'll see how those variables can be used to select an outbound telephony provider (ITSP) gateway specific for a user or a users' group, or for letting all phones belonging to the same group to ring concurrently for an incoming call, for adding a special value to the Call Detail Record accounting log, of for choosing to which dialplan context a call must be sent to.
We'll also see how to add users and groups to the User Directory.
In SIP, we can send calls to gateways (gateways are SIP servers that belongs to ITSPs or to other in-house services), and then will be the gateway that will take care of call routing and termination (connecting the call to the national or international PSTN). We'll see how to set up a gateway in a way that automatically registers our FreeSWITCH machine to the remote server, and send it authorization credentials when originating outbound calls.

In this chapter we will cover the following topics:

- User Directory concepts
- Authentication, Authorization and Accounting (AAA) in FreeSWITCH
- Exploring and Using the Example User Directory
- Adding Users, by Modifying the User Directory and Dialplan
- SIP gateways, DID numbers, Dialplan Contexts and ITSPs

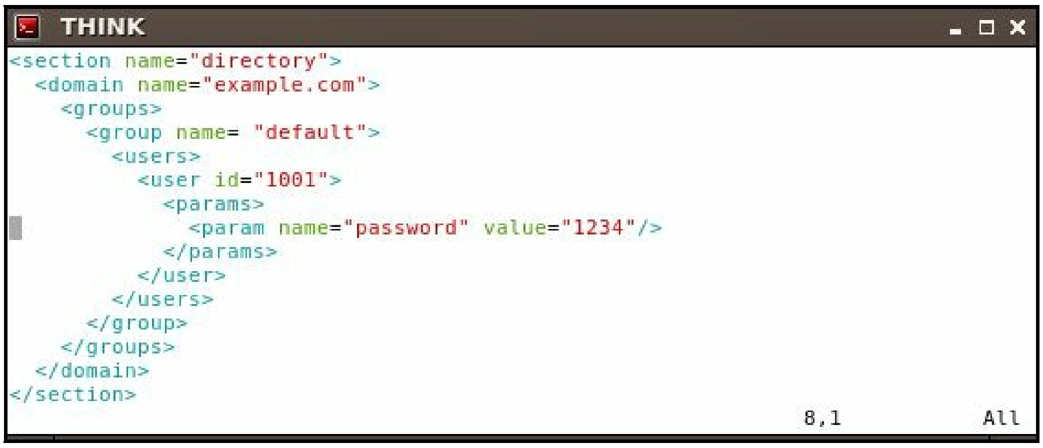
User Directory concepts

The User Directory is a (possibly big and complex) XML document that is accessed by FreeSWITCH and all of its modules whenever they need to know about users' attributes. It is in the User Directory that the user's password, which groups (if any) the user belongs to, various and arbitrary variables and parameters related to the user or to the group, and so on are defined.

Each time the FreeSWITCH core or one of its modules needs such an information, it will issue an internal request. That internal request will use one of many different methods to end up with an XML snippet that defines the information requested. So, that XML snippet can be the result of a query to a database or to a webserver, or it can be built starting from the XML tree FreeSWITCH composed in its own memory at startup based on the configuration XML documents found on the filesystem.

In this chapter we'll mostly look at the User Directory we find on the filesystem after a default fresh install, in the example demo configuration. The same concepts will apply also when the User Directory registry is accessed via database query or HTTP request.

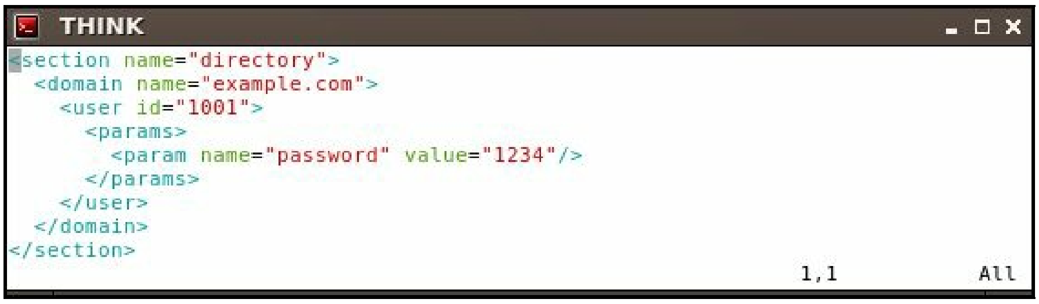
The User Directory, or more colloquially in FreeSWITCH parlance simply the "directory", usually has this schema:



```
<section name="directory">
  <domain name="example.com">
    <groups>
      <group name= "default">
        <users>
          <user id="1001">
            <params>
              <param name="password" value="1234"/>
            </params>
          </user>
        </users>
      </group>
    </groups>
  </domain>
</section>
                                                8,1           All
```

It contains one or more "domain", each "domain" contains "groups", each "groups" contains one or more "group" items. Each "group" contains the "users" item, which contains one or more "user" item. The "user" item contains the "params" item, which contains one or more "param" items. Being XML the eXtendable Markup Language, the schema can actually be more complex, we'll see it with "variables" and "variable" items interspersed here and there, but you sure got the knack of it.

By the way, the only required XML items are "domain" and "user". So, the simplest possible complete "directory" XML tree (or, quite possibly, the XML snippet returned by mod_xml_curl) would be similar to the following:



```
<section name="directory">
  <domain name="example.com">
    <user id="1001">
      <params>
        <param name="password" value="1234"/>
      </params>
    </user>
  </domain>
</section>
                                                              1,1           All
```

AAA: Authentication, Authorization, Accounting

"AAA" is an acronym that rings familiar to all old hands of telecommunication since time immemorial. Particularly, is the base itself of businesses like PSTN carriers, but also to Internet Providing since the early days of dialup modems.

Simply put, AAA is the whole operation of:

* Authentication - knowing an user is actually who he is declaring to be (eg: login and password do match)

* Authorization - knowing what "rights" that user has (eg: he can call national and international numbers, excluded premium and for-pay numbers)

* Accounting - knowing what the user has just done and his history (eg: he started with a credit of 835 units, and the call just finished is rated 35 units)

As you can see from this souped-up definition (google around for better ones), AAA is a pretty abstract requirement, one that can be implemented in many different ways, for many different aims, business logics and technologies. And it is different from billing, even if it is providing the billing operation with its basic accounting data, and is itself influenced by billing (if you don't pay your bills you will probably end up being no longer authorized to use services).

In the old days, and still used by many ISPs, RADIUS was the main protocol and server for centralization of AAA. FreeSWITCH supports RADIUS and can be integrated in such legacy environments.

In the example demo configuration from a fresh install, FreeSWITCH handles AAA in a static way, tailored for a company's internal PBX:

* **Authentication** is checked against user id and password in the **User Directory**

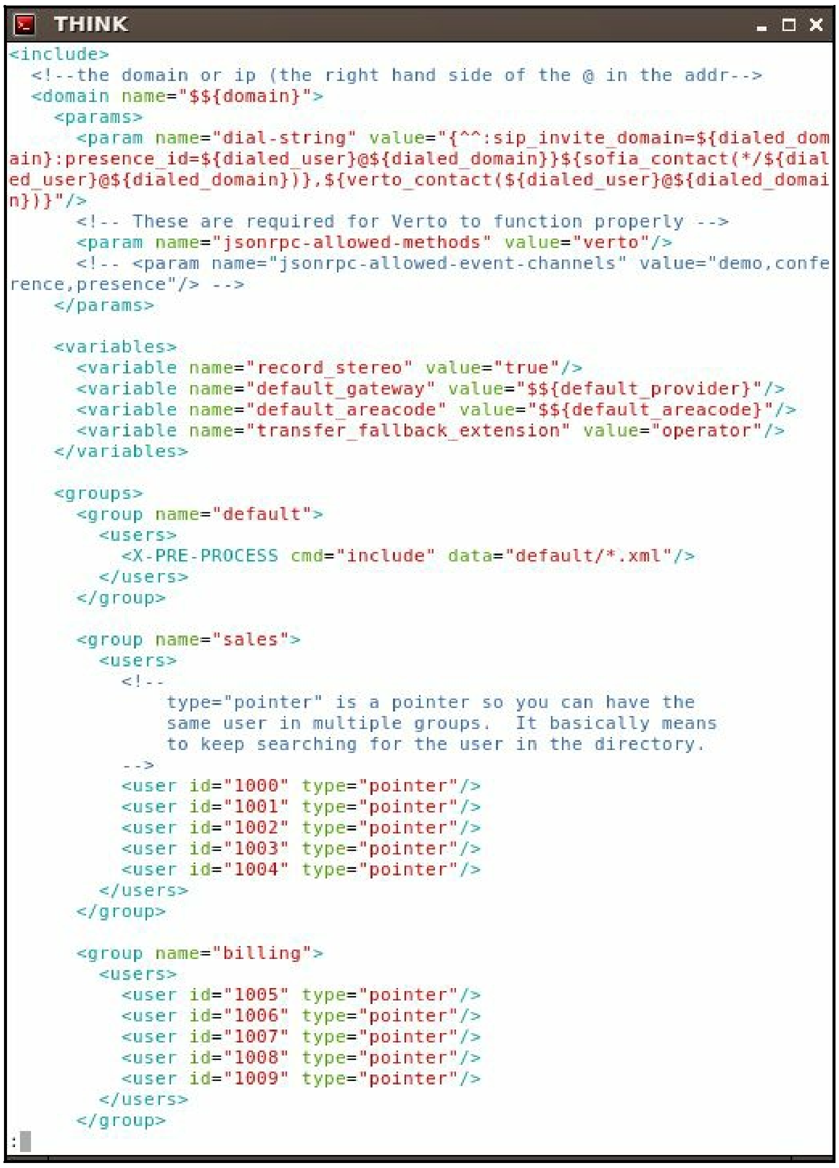
* **Authorization** is checked in **dialplan** against variables

* **Accounting** data is provided in the form of Comma Separated Values (**CSV**) rows (directly processable by spreadsheets and databases) and/or **XML** entities (richer in data dimensions, for further elaboration)

Exploring and Using the Demo Example User Directory

After a fresh install, out of the box, you'll find the "directory" main XML file located at /usr/local/freeswitch/conf/directory/default.xml . That main file will then include many other files, as we'll see.

Let's understand its most important lines.



THINK

```
<include>
  <!--the domain or ip (the right hand side of the @ in the addr-->
  <domain name="$${domain}">
    <params>
      <param name="dial-string" value="{^^:sip_invite_domain=${dialed_dom
ain}:presence_id=${dialed_user}@${dialed_domain}}${sofia_contact(*/${dial
ed_user}@${dialed_domain})},${verto_contact(${dialed_user}@${dialed_domai
n})}"/>
      <!-- These are required for Verto to function properly -->
      <param name="jsonrpc-allowed-methods" value="verto"/>
      <!-- <param name="jsonrpc-allowed-event-channels" value="demo,confe
rence,presence"/> -->
    </params>

    <variables>
      <variable name="record_stereo" value="true"/>
      <variable name="default_gateway" value="$${default_provider}"/>
      <variable name="default_areacode" value="$${default_areacode}"/>
      <variable name="transfer_fallback_extension" value="operator"/>
    </variables>

    <groups>
      <group name="default">
        <users>
          <X-PRE-PROCESS cmd="include" data="default/*.xml"/>
        </users>
      </group>

      <group name="sales">
        <users>
          <!--
              type="pointer" is a pointer so you can have the
              same user in multiple groups.  It basically means
              to keep searching for the user in the directory.
          -->
          <user id="1000" type="pointer"/>
          <user id="1001" type="pointer"/>
          <user id="1002" type="pointer"/>
          <user id="1003" type="pointer"/>
          <user id="1004" type="pointer"/>
        </users>
      </group>

      <group name="billing">
        <users>
          <user id="1005" type="pointer"/>
          <user id="1006" type="pointer"/>
          <user id="1007" type="pointer"/>
          <user id="1008" type="pointer"/>
          <user id="1009" type="pointer"/>
        </users>
      </group>
:
```

Domains and groups

First things first, we set the "domain" we're talking about. In SIP and in Verto, addresses are in the form "userid@domain", very much like in email (actually SIP borrowed this format from SMTP, the mail protocol, and Verto borrowed it from SIP).
We can have multiple "domain" items, they would be added after the closing </domain> XML tag (not in the screenshot).

Then, a "params" XML container starts, surrounding the "param" items that belongs directly to the "domain" we're defining. Those "param" items will be linked to both the "users" that are contained into this domain, and the "group" items (then, the "user" items that belong to the "group" items) contained into the "groups" container of this domain.

The "param" named "dial-string" is of param-ount importance (am I good with words, huh?). It defines the way the user will be reached, both the protocol(s) (sip and verto, in this case) to be used, and the specific addresses to send the call(s) to. Don't bother to understand the details right now. As you can see, it sets both a sip and a verto address, actually using internal APIs to know whether the user is registered in those protocols, and if it is registered, at which addresses.

Then there is another param, to allow the verto protocol to interact with procedure calls, for conference management and such.

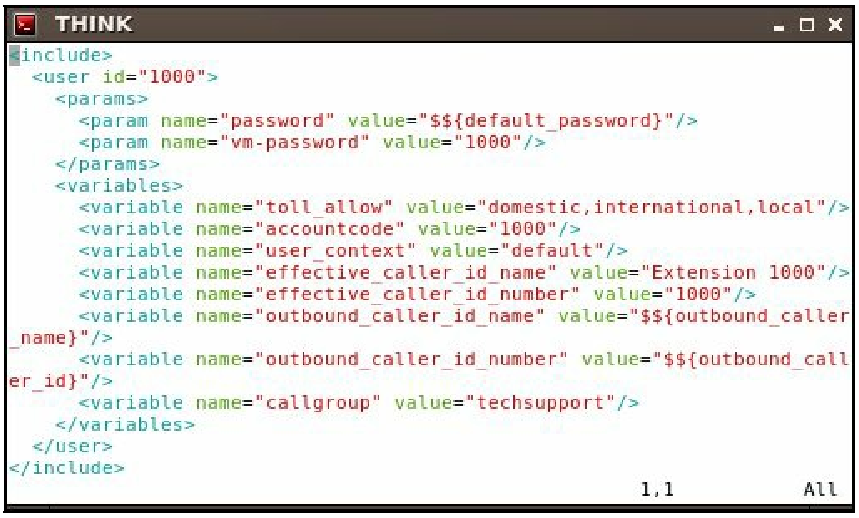
Then, we have a container of "variables" that will be created or set for users in this domain. Here we set many useful defaults, to be overridden both in dialplan and/or in the user or group definitions.

Then the "groups" container, and its "group" items. One of those "group" items is named "default", and inside it we can find a "users" container that includes all the XML files in the default/ subdirectory (relative to the User Directory location, eg: /usr/local/freeswitch/conf/directory/default/*.xml). Those XML files would reasonably contain XML snippets defining "user" items. We'll see that later.

We then can see other "group" items, each of them with a "users" item that actually contains some "user" items. But in this case those "user" items are not defining user's data and variables, instead they just indicate to look into the rest of the domain's XML definition for a user with a particular "id". This is useful to assign users to more than a group. Eg: they all belong to group "default", but some of them belong to the "sales" group too, while other belong both to "default" and to "billing" groups.

Users

Now, let's look at a true "user" item definition (eg, not to a "pointer"). Go inside /usr/local/freeswitch/conf/directory/default/ and open the file 1000.xml:



```
<include>
  <user id="1000">
    <params>
      <param name="password" value="$${default_password}"/>
      <param name="vm-password" value="1000"/>
    </params>
    <variables>
      <variable name="toll_allow" value="domestic,international,local"/>
      <variable name="accountcode" value="1000"/>
      <variable name="user_context" value="default"/>
      <variable name="effective_caller_id_name" value="Extension 1000"/>
      <variable name="effective_caller_id_number" value="1000"/>
      <variable name="outbound_caller_id_name" value="$${outbound_caller
_name}"/>
      <variable name="outbound_caller_id_number" value="$${outbound_call
er_id}"/>
      <variable name="callgroup" value="techsupport"/>
    </variables>
  </user>
</include>
                                                      1,1           All
```

At the beginning, inside the very same "user" tag, is declared the "id" of the user (this is the same "id" that will be looked upon by the "pointer" kind of "user" item we saw in previous section). The id is specific to a "domain", eg: you can have a FreeSWITCH server that serves multiple domains, let's say, "domainA" and "domainB".

In both of them can exist one user whose "id" is "1000". Each one of those two "1000" is a completely different user, not related in any way to the other one. So, those two users will probably have different passwords, can make and/or receive calls both of them at the same time, etc. They are completely different users, they just happen to have the same "id". They can coexist on the same FreeSWITCH server because they belong to different domains. So, they would be "1000@domainA" and "1000@domainB". See? Different! When there is more than one domain in a FreeSWITCH platform, telco lingo sez: "is multi-tenant".

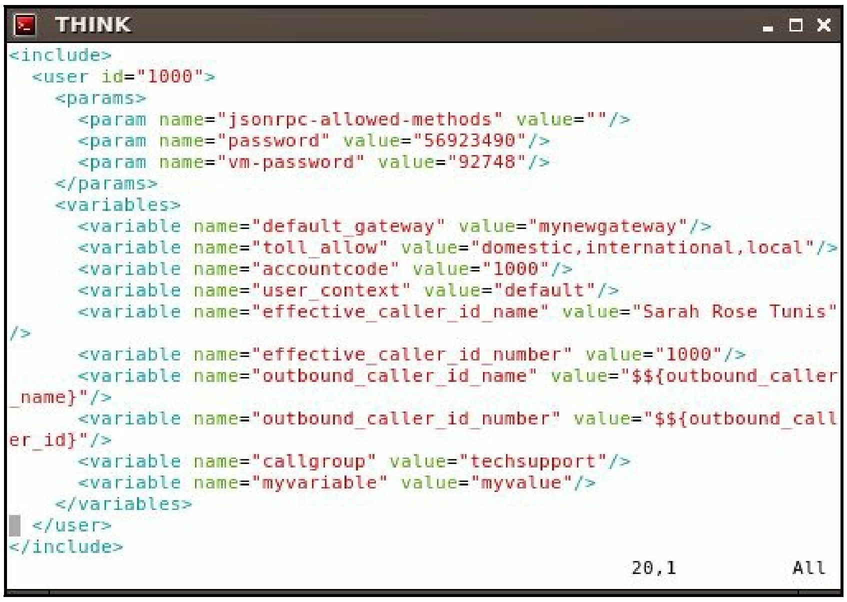
Inside the "user" container item, in this example we find both a "params" and a "variables" container item.

Inside the "params" container there are "param" items that define the parameters that belongs exclusively to this user. Possibly those "param" items override the ones set in the "domain" container, or in the "group" container. For example, you can repeat here the "jsonrpc-allowed-methods" param we found before in the domain definition, so this particular user will have its value explicitly set for him (perhaps to "", empty string, so no methods are allowed) instead of inheriting the one defined in domain (that was the string "verto").

Other ones you most probably want to modify are "password", obviously, so to set a specific password for this user (instead of inheriting the default one set into vars.xml file), and "vm-password".
And there are cases where it can be convenient to override the "dial-string" param item.

Exactly the same happens for the "variables" container, with "variable" items. Here, you may want to override "default_gateway", so to use a specific SIP gateway for this user. And very often you want to personalize some of the "*id_name" and "*id_number" variable items.

Also, you may want to set your own variables. There are some already set in the demo example user definitions 1000.xml...1019.xml. All of them can then be referenced in dialplan and scripts, for example you want to check what kind of call a user is allowed to originate, and you choose to test the content of the variable ${toll_allow} for this purpose. Remember, all those variables are available both to core FreeSWITCH and to all modules. So, the modules generating CDRs will be able to reference them, you can have the content of the ${callgroup} reported into your Call Detail Records (for accounting purposes):



```
<include>
  <user id="1000">
    <params>
      <param name="jsonrpc-allowed-methods" value=""/>
      <param name="password" value="56923490"/>
      <param name="vm-password" value="92748"/>
    </params>
    <variables>
      <variable name="default_gateway" value="mynewgateway"/>
      <variable name="toll_allow" value="domestic,international,local"/>
      <variable name="accountcode" value="1000"/>
      <variable name="user_context" value="default"/>
      <variable name="effective_caller_id_name" value="Sarah Rose Tunis"
/>
      <variable name="effective_caller_id_number" value="1000"/>
      <variable name="outbound_caller_id_name" value="$${outbound_caller
_name}"/>
      <variable name="outbound_caller_id_number" value="$${outbound_call
er_id}"/>
      <variable name="callgroup" value="techsupport"/>
      <variable name="myvariable" value="myvalue"/>
    </variables>
  </user>
</include>
                                                     20,1          All
```

The following table lists each variable and what it is used for:

Variable	Purpose
toll_allow	Specifies which types of calls the user can make

accountcode	Arbitrary value that shows up in CDR data
user_context	The dialplan context that is used when this person makes a call
effective_caller_id_name	Caller ID name displayed on called party's phone when calling another registered user
effective_caller_id_number	Caller ID number displayed on called party's phone when calling another registered user
outbound_caller_id_name	Caller ID name sent to provider on outbound calls
outbound_caller_id_number	Caller ID number sent to provider on outbound calls
callgroup	Arbitrary value that can be used in dialplan or CDR

Adding a user to the demo example directory and dialplan

Users are defined in User Directory. Where a call must go (eg, where it is routed to) is defined in dialplan. In dialplan, there are regular expressions (regexes) that check variables for string patterns. The most checked on variable is "destination_number", whose value is the number the caller dialed.

So, if we add a user in the "directory", we must then be sure the dialplan allows for calls to reach that newly added user. This often, but not always, means that we must modify the dialplan too. We'll see about this later in this section.

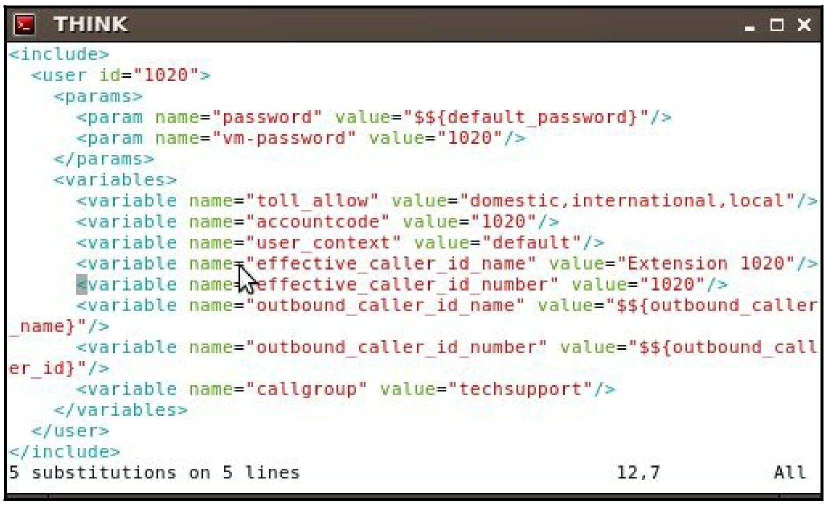
Adding a new user to the directory

The demo example configuration file layout is organized to make it easy to add and remove users. We saw that in the /usr/local/freeswitch/conf/directory/default.xml file, inside the predefined default domain, in group "default", there is an instruction to the FreeSWITCH configuration pre-processor to read in all of the XML files contained in the /usr/local/freeswitch/conf/directory/default/ directory. Those files contain the users' definitions.

We found the files 1000.xml ... 1019.xml, the 20 demo predefined users with "id" 1000 to 1019. There you can find also some other XML files, very well commented, that can be source of inspiration.

In our demo configuration case, adding a user is done simply by copying 1000.xml to, let's say, 1020.xml, and then editing the newly created XML file.

In your text editor, search and replace all string occurrences of "1000" to be "1020":



```
<include>
  <user id="1020">
    <params>
      <param name="password" value="$${default_password}"/>
      <param name="vm-password" value="1020"/>
    </params>
    <variables>
      <variable name="toll_allow" value="domestic,international,local"/>
      <variable name="accountcode" value="1020"/>
      <variable name="user_context" value="default"/>
      <variable name="effective_caller_id_name" value="Extension 1020"/>
      <variable name="effective_caller_id_number" value="1020"/>
      <variable name="outbound_caller_id_name" value="$${outbound_caller
_name}"/>
      <variable name="outbound_caller_id_number" value="$${outbound_call
er_id}"/>
      <variable name="callgroup" value="techsupport"/>
    </variables>
  </user>
</include>
5 substitutions on 5 lines                              12,7          All
```

That's it. We just created, by adding this file in that directory, a new user with all the same attributes of the user with "id" 1000. That is, same "callgroup", same "toll_allow", and so on. We only changed its "id", its voice mail password, its "accountcode" and "effective_caller_id_name" all of them to be the "1020" string. From FreeSWITCH command line (fs_cli), we need to reload configuration for the new user to become known to FreeSWITCH:

```
reloadxml
```

Modifying the dialplan for the new user

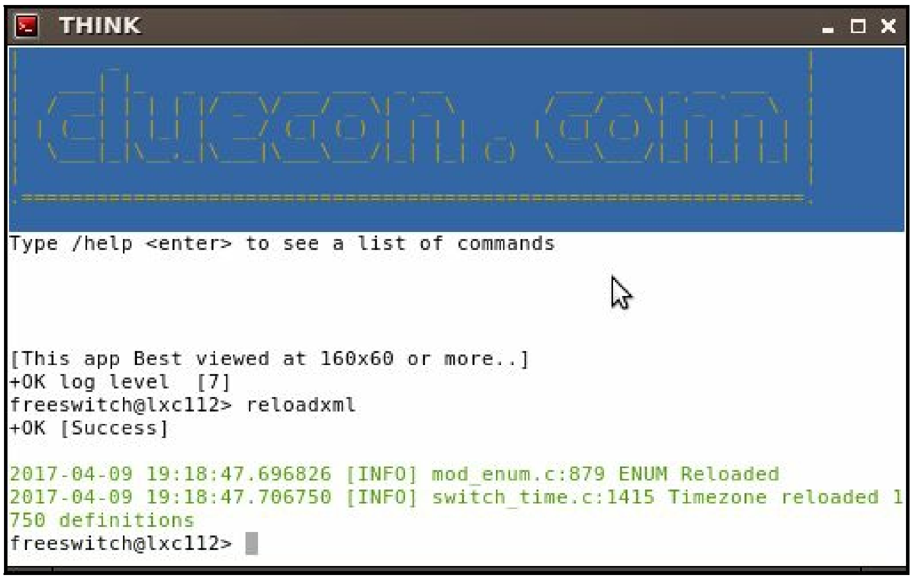
We still have a problem: we saw in the previous chapter how the "Local_Extension" in dialplan define how to route calls whenever "destination_number" (eg, what the caller dialed in hope to reach the callee) is matching the regular expression "^(10[01][0-9])$". This regex matches 1000...1019, and then send the call to the user with an "id" corresponding to the dialed number.

We must do one of two things: we can modify that already existing "Local_Extensions" so its condition matches 1020 too, or we can create a new extension in dialplan.

We can create a new extension in dialplan by copying all the "Local_Extension" XML snippet (from the opening "extension" tag to the closing "/extension" tag, both included). We may modify the new extension name attribute, but that's not required. What is required is to change the "condition" on "destination_number" expression. We can change it to be: "^1020$", and it will match exactly and only a destination_number string variable equal to "1020" (caret is beginning of string placeholder, dollar sign is end of string placeholder).

Or, we can edit the already existing "Local_Extensions" so its "destination_number" condition matches 1020 too. There is more than one way to do this (as always with anything PERL related, and FreeSWITCH regexes are of the PERL flavor).
The original regex is "^(10[01][0-9])$", matching 1000...1019.
We can modify it to be "^(10[01][0-9]|1020)$", and it will match 21 dialed numbers: 1000...1019 and 1020.
Or, we can make it "^(10[012][0-9])$", and it will match 30 dialed numbers: 1000...1029.

Again, from FreeSWITCH command line (fs_cli), we need to reload configuration for the modified dialplan to become known to FreeSWITCH:



```
Type /help <enter> to see a list of commands




[This app Best viewed at 160x60 or more..]
+OK log level  [7]
freeswitch@lxc112> reloadxml
+OK [Success]

2017-04-09 19:18:47.696826 [INFO] mod_enum.c:879 ENUM Reloaded
2017-04-09 19:18:47.706750 [INFO] switch_time.c:1415 Timezone reloaded 1
750 definitions
freeswitch@lxc112> 
```

Call to a Group

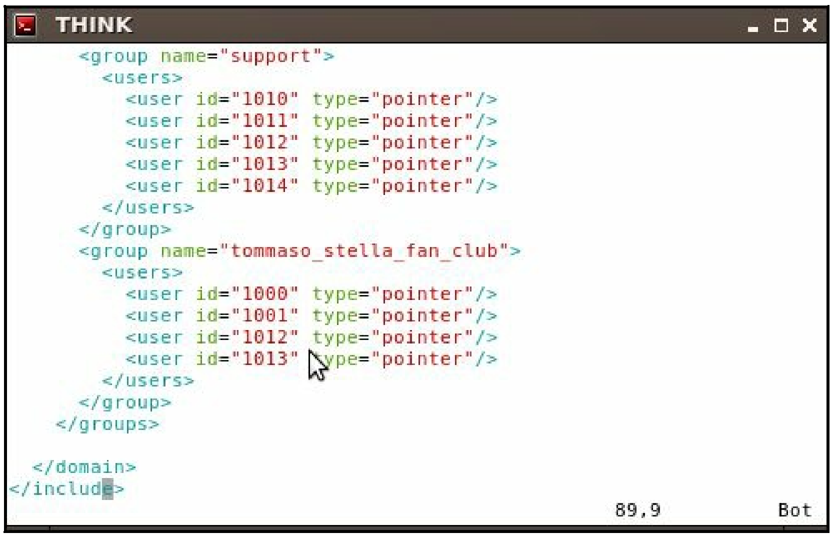
In section "Users and Groups" we saw how FreeSWITCH users can belong to "groups" in the User Directory. Also, we saw how in demo example configuration there is one group "default" to which all users belong to, and many other groups that gather only some users. Btw, this is just a demo example, it does not need to be this way, eg: group "default" could be non-existant, or gather only one user.

User groups are useful not only for assigning variables and parameters: a group could be used as "callee" to originate (bridge) multiple calls.

In demo configuration groups are defined into /usr/local/freeswitch/conf/directory/default.xml XML file.

Let's edit it, add a new group, and test a call to it.

Toward the end of the file, before the closing </groups> tag, insert a new "group" container and the "user pointer" items you want to reference:

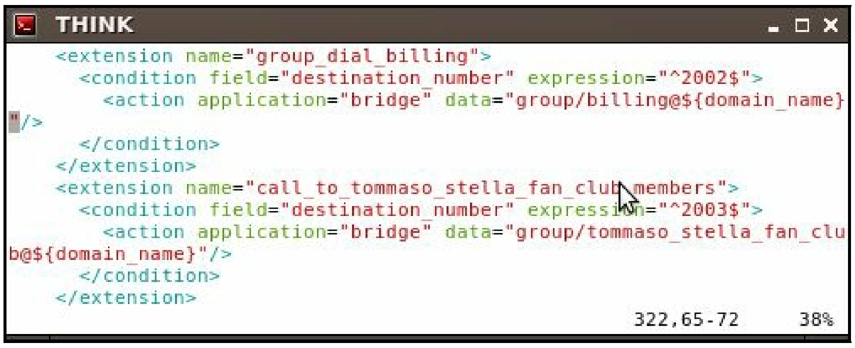


```
      <group name="support">
        <users>
          <user id="1010" type="pointer"/>
          <user id="1011" type="pointer"/>
          <user id="1012" type="pointer"/>
          <user id="1013" type="pointer"/>
          <user id="1014" type="pointer"/>
        </users>
      </group>
      <group name="tommaso_stella_fan_club">
        <users>
          <user id="1000" type="pointer"/>
          <user id="1001" type="pointer"/>
          <user id="1012" type="pointer"/>
          <user id="1013" type="pointer"/>
        </users>
      </group>
    </groups>

  </domain>
</include>
                                              89,9           Bot
```

In our case we just copied and pasted the "support" group, and edited both the group name and the contained "user" items.Save the file, and issue a "reloadxml" from FreeSWITCH command line.

Now, let's add a group call extension to the dialplan: edit the XML file /usr/local/freeswitch/conf/dialplan/default.xml

Locate the extension "group_dial_billing", and copy and paste it immediately after its </extension> closing tag:



```
    <extension name="group_dial_billing">
      <condition field="destination_number" expression="^2002$">
        <action application="bridge" data="group/billing@${domain_name}
"/>
      </condition>
    </extension>
    <extension name="call_to_tommaso_stella_fan_club_members">
      <condition field="destination_number" expression="^2003$">
        <action application="bridge" data="group/tommaso_stella_fan_clu
b@${domain_name}"/>
      </condition>
    </extension>
                                                  322,65-72     38%
```

In our case we edited the extension's name, the destination_number "expression", and the "data" passed as arguments to the "bridge" action.

The regular expression to be matched by "destination_number" value is now "^2003$" (caret indicates beginning of string, dollar sign indicates end of string), so the incoming call will match if the caller has dialed exactly "2003".

Then, in the "bridge" action data, we use the construct "group/groupname" and the ${domain_name} variable, to originate concurrent calls (parallel calls) to all members of the "tommaso_stella_fan_club". By the way, Tommaso Stella is a famous Italian sailor.

We can have groups with same name in different domains (eg, many domains in a multitenant installation may have a group named "sales"), so we reference the domain_name variable, which contains the domain the incoming call has been identified as being routed to.

Please note that group members (that are, the "user" items) will be reached using their "dial_string", defined as global, domain, group, or item parameter. So, for example, a user can be reachable via a SIP call, another via a Verto call, a third via a TDM call, and a fourth via three parallel calls by all means necessary.

After issuing another "reloadxml" from FreeSWITCH command line, we can then call "2003" extension, and have all of Tommaso Stella fandom reached by parallel calls. All phones and browsers will begin ringing, and as soon as someone answers, ringing on the other phones will stop.

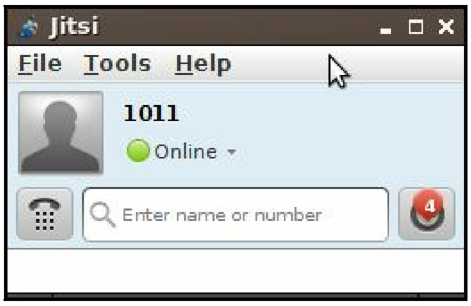
Voicemail

One of the finest features of the example demo configuration is the ready made voicemail system, brought to you by mod_voicemail. The last four actions in "Local_Extension" in demo dialplan are as follows:

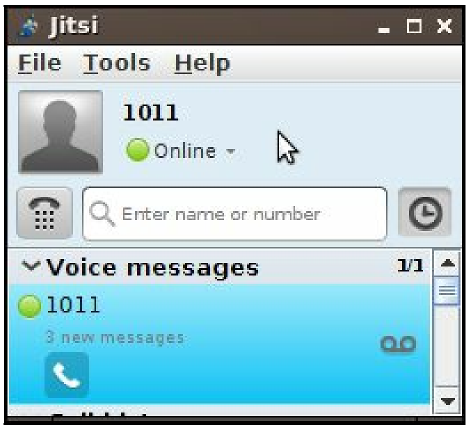
```
        <action application="bridge" data="user/${dialed_extension}@${domain_name}"/>
        <action application="answer"/>
        <action application="sleep" data="1000"/>
        <action application="bridge" data="loopback/app=voicemail:default ${domain_name} ${di
```

This means: try to connect the incoming call to the callee (that is, to the dialed_extension, that was derived from destination_number), if you fail to connect it (eg, the phone(s) at dialed_extension is/are not answered before the 30 second timeout), then answer the incoming call (eg, go off hook), wait for 1 second doing nothing, then connect that incoming call to the voicemail application (using the loopback "fake" endpoint).

After an incoming call gets connected to the voicemail application, and the caller left a message, FreeSWITCH is smart enough to light up (or blink) the MWI (Message Wait Indicator) on the callee phone(s). Wooot, that kind of magic gives a lot of satisfaction to us telco geeks. (Also, when the voice message has been heard or deleted by callee, FreeSWITCH will have those MWIs to shut. Yay!):



Let's see an example using Jitsi, a fine free and open source softphone. If the callee, when she's back at the softphone, clicks on the red icon with the number 4 (eg, the MWI says: "you have 4 messages waiting"), she will see some details about the voicemail, eg only 3 of the 4 messages are new:



Then, when she try to hear the message, she hit a bug in Jitsi (version 2.10.5550 for Linux). Do to that bug, when she clicks the blue handset icon, she actually call back herself (that is, she call back the account the messages where left for).

So, if she actually want to hear the messages, she have to call *98 (or 4000) from the dialpad. She will be asked for her voicemail password. In default demo configuration, voicemail password is the same as the extension number. So, in this case is 1011. She will then have the whole panoply of a super featured voicemail system. And after all messages will be listened to, MWI icon will disappear. But let's see only some of the available options when she call *98 or 4000 to check her own voice mailbox:

```
Main menu:
1-Listen to new messages
2-Listen to saved messages
5-Options menu (record name, record greetings, manage greetings, and so on)
#-Exit voicemail
While listening to a message:
1-Replay message from the beginning
2-Save message
4-Rewind
6-Fast-forward
After listening to a message:
0-Replay message from the beginning
2-Save message
4-Send to e-mail (requires configuration)
5 - Return the call
7-Delete message
8 - Forward the voicemail message
```

All those menus, and much more, are completely configurable by editing the XML file /usr/local/freeswitch/conf/autoload_configs/voicemail.conf.xml.

Also if you don't want to change it, give it a look. You'll discover much more options than described here. Also, don't forget to check our online documentation at http://freeswitch.org/confluence for more configuration, for setting up voicemail to mail (that is, voice message delivered to your mailbox as a mail attachment), and many more interesting features.

Communicate with the World via SIP Gateways and DIDs

Yeah yeah, all nice and dandy, but... really you want all this beautiful communication platform to only be useful for calls between internal registered users?

Let's see how to add a SIP gateway to an Internet Telephony Service Provider (ITSP), the easiest way to expand our communication platform to have a global reach.
An ITSP provides its customers with SIP "trunks", eg: capabilities to connect outbound calls to the world phone network. Also, an ITSP can provide DIDs (Direct Inward Dialing) numbers: telephone numbers that when called originate an inbound SIP call to our communication platform. So: SIP trunks are for us to send out calls to the world. DIDs are for gathering calls from the world to us.

SIP Gateway from Us to the ITSP to the World

FreeSWITCH simplifies SIP for us, with the concept of "gateway".
That "gateway" in FreeSWITCH lingo is just a glorified automation provided to us, so we can use a simplified syntax meaning: "check whether this particular Internet Telephony Service Provider (ITSP) is up and reachable, send this call out to it, give the ITSP the login and password to authorize our call if needed, and wait for the ITSP to connect our call to the wide world".

All of those mechanisms and actions are performed transparently to us, and the monitoring of the ITSP availability is performed routinely even when we don't use it, so if ITSP is down we don't even try it, but we'll be able to use it when it is back.

By the way, what happens under the hood is that FreeSWITCH uses the SIP concept of "proxy" to ask our ITSP to route our outbound call. The ITSP then use some method opaque to us to terminate, eg to connect to final destination, our call. That termination method can be a direct link ITSP has with the final or intermediate carrier, or maybe ITSP has hardware to interface to PSTN, such as Sangoma cards and Patton boxes, or even our ITSP is then rerouting our traffic to another upstream ITSP. We don't know.

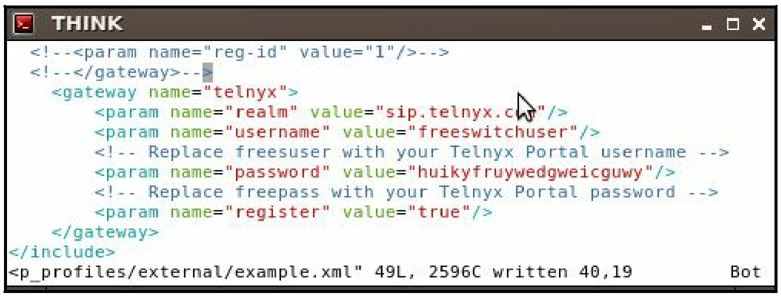
Each ITSP has slightly different methods for accepting our requests for them to route our outbound calls. It is important that ITSPs take security seriously. **We must TAKE SECURITY in connecting to our telephony provider VERY SERIOUSLY: it MAY COST US THOUSANDS OF DOLLARS IN FRAUD**. It is as easy as: someone else, maybe a rogue ITSP, route its outbound calls (its "traffic") using our account at our telephony provider. We end up paying the bill, that can be dramatically high.

So, before setting up our first gateway, let me repeat this: we must take security and fraud prevention very seriously.
Telephony fraud is a huge industry, an incredible number of frauds are continually perpetrated, and someone is definitely going to pay for that traffic. Keep our password complex and unguessable, ask our ITSP for the traffic to be allowed only from our IP address, ask for whitelisting destination countries (that is, traffic will only be accepted if directed to countries we authorized, such as USA, Europe, and Asia, but not to Oceania and satellite phones), block for pay numbers (for example, 188 and similar). If possible at all, use SIPS via TLS and SSL certificates to be sure only traffic actually generated by us will be accepted and acted upon by our ITSP.

Let's assume we've done our due diligence and secured ourselves against fraud at the best of our possibilities.

Edit the /usr/local/freeswitch/conf/sip_profiles/external/example.xml XML file, and add a "gateway" container before the ending </include> tag. We can search FreeSWITCH documentation at http://freeswitch.org/confluence looking for an example configuration of a gateway to our ITSP, or we can look into our ITSP website, send ITSP's support a mail, or ask in FreeSWITCH mailing list. Anyway, usually is very easy, and boils down to a user/password couple, and to specify whether registration is needed or not.

In our example we'll use Telnyx, a sponsor of 2017 ClueCon conference (eg, our conference, each year in August in beautiful Chicago). First of all we create a "connection" username/password using the Telnyx website. Our newly created "connection" has "freeswitchuser" as username and "huikyfruywedgweicguwy" as password. Then, we edit the /usr/local/freeswitch/conf/sip_profiles/external/example.xml XML file, and add a new "gateway" container before the ending </include> tag:

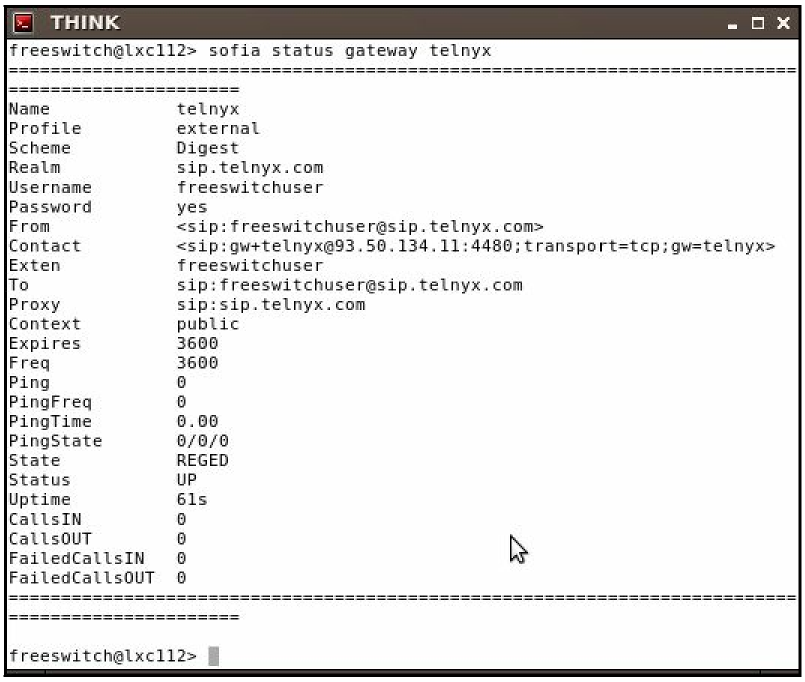


```
  <!--<param name="reg-id" value="1"/>-->
  <!--</gateway>-->
    <gateway name="telnyx">
        <param name="realm" value="sip.telnyx.com"/>
        <param name="username" value="freeswitchuser"/>
        <!-- Replace freesuser with your Telnyx Portal username -->
        <param name="password" value="huikyfruywedgweicguwy"/>
        <!-- Replace freepass with your Telnyx Portal password -->
        <param name="register" value="true"/>
    </gateway>
</include>
<p_profiles/external/example.xml" 49L, 2596C written 40,19          Bot
```

That's it. We can restart FreeSWITCH, or issue "reload mod_sofia" from command line, and FreeSWITCH will register itself to the Telnyx SIP proxy, and we'll be ready to send calls to the outside world.

We can check how our new gateway is going by issuing "sofia status gateway telnyx" from FreeSWITCH command line.

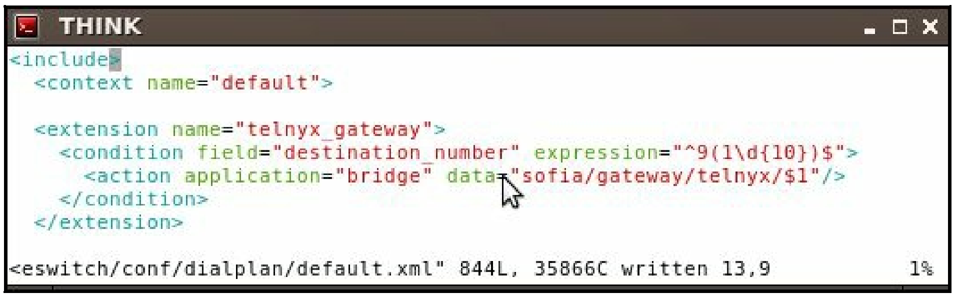
In our case, we see we've been successful in registration ("State" line is "REGED", eg registered), that this gateway is up since 61 seconds, we'll register again in 1 hour, and so on:



```
freeswitch@lxc112> sofia status gateway telnyx
===========================================================================
======================
Name            telnyx
Profile         external
Scheme          Digest
Realm           sip.telnyx.com
Username        freeswitchuser
Password        yes
From            <sip:freeswitchuser@sip.telnyx.com>
Contact         <sip:gw+telnyx@93.50.134.11:4480;transport=tcp;gw=telnyx>
Exten           freeswitchuser
To              sip:freeswitchuser@sip.telnyx.com
Proxy           sip:sip.telnyx.com
Context         public
Expires         3600
Freq            3600
Ping            0
PingFreq        0
PingTime        0.00
PingState       0/0/0
State           REGED
Status          UP
Uptime          61s
CallsIN         0
CallsOUT        0
FailedCallsIN   0
FailedCallsOUT  0
===========================================================================
======================

freeswitch@lxc112>
```

Now, let's edit the /usr/local/freeswitch/conf/dialplan/default.xml dialplan XML file, so we will be able to send calls through Telnyx gateway.

At the beginning of the file, we add a new extension that will route our calls through Telnyx to the USA PSTN:



```
<include>
  <context name="default">

  <extension name="telnyx_gateway">
    <condition field="destination_number" expression="^9(1\d{10})$">
      <action application="bridge" data="sofia/gateway/telnyx/$1"/>
    </condition>
  </extension>

<eswitch/conf/dialplan/default.xml" 844L, 35866C written 13,9            1%
```

Here, look the regular expression cleverness: "destination_number" must begin with 9, then 1, then ten digits, and no more. Of the whole "destination_number" string, we use parenthesis to select starting from the 1 after the 9, up and included the following ten digits (eg, we "discard" the initial 9 from selection).
We will find our selection in the next line, inside the "data" arguments passed to the "bridge" action": will be substituted to the "$1" placeholder (representing what was selected by the first couple of parenthesis. If there was a second parenthesis couple, we would be able to refer to what this second couple had selected using the "$2" placeholder).

After issuing a "reloadxml" from FreeSWITCH command line, we can dial 918005551212 from one of our registered phones, and have a call to 18005551212 (9 is discarded by dialplan) sent to USA via Telnyx.

SIP DID from World to the ITSP to Us

A DID (Direct Inward Calling) number is a phone number that in some way when it is called then originates a SIP call toward a SIP server (the SIP server, in our case FreeSWITCH, will then route the call to its final destination, often an extension phone).

The logical flow is: CALLER -> PSTN -> DID -> ITSP -> SIP -> FreeSWITCH

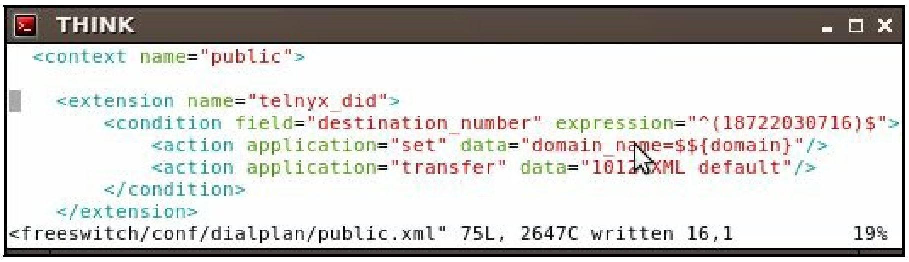
There are many DIDs providers, maybe the most known pure DID provider is DIDx.net, that can provide local numbers from all around the world. Many ITSPs also can lease you numbers, local to the country they are located and international numbers, and usually they get those numbers from upstream DID providers.

So, we buy (actually, lease, nobody can sell you the ownership of a phone number, you can only pay for its use month by month) a local number from our ITSP, in this case Telnyx. We choose a number in Chicago, USA.
Telnyx is smart enough to use the data from our FreeSWITCH gateway registration (eg our IP address and port) to send us the calls coming from the number we "purchased".

In our case we purchased the DID number "+1-872-203-0716".

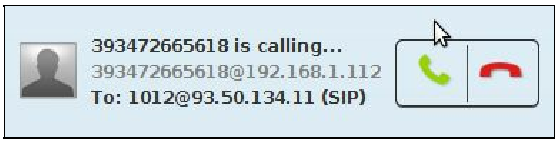
We need to add an extension to the incoming "public" dialplan ("public" is the dialplan context where we receive inbound calls from the outside world), so we will be able to connect the incoming call Telnyx send us from our DID number to one of our registered phones .

Edit /usr/local/freeswitch/conf/dialplan/public.xml , and at file beginning add the "telnyx_did" extension, which will transfer the incoming call to the 1012 extension in default dialplan (which will be actually served by the "Local_Extension", as we saw before, that provides a lot of features, voicemail included).



```
 <context name="public">

   <extension name="telnyx_did">
       <condition field="destination_number" expression="^(18722030716)$">
           <action application="set" data="domain_name=$${domain}"/>
           <action application="transfer" data="1012 XML default"/>
       </condition>
   </extension>
<freeswitch/conf/dialplan/public.xml" 75L, 2647C written 16,1          19%
```

After issuing a "reloadxml" from FreeSWITCH command line, we can use our cellphone to call "+1-872-203-0716" and the call will be routed to our softphone registered as "1012" user. It will start to ring, and if we do not answers, call will go to voicemail to record a voice message.



393472665618 is calling...
393472665618@192.168.1.112
To: 1012@93.50.134.11 (SIP)

SIP and Verto Profiles, Dialplan Contexts, ITSPs

In previous section we edited the "public" context of dialplan for servicing incoming calls from DID numbers. And, in the section before it, we added an ITSP gateway to the "external" SIP profile definition, and added an extension to the "default" dialplan context, so we can use the gateway to call outside world.

Actually, in example demo configuration we have two SIP profiles, named "internal" and "external" (those are just names, you can change them). And we have two major dialplan contexts, named "default" and "public" (there is also another, "features", but is not relevant here).

Seems a little confusing at first, but it makes perfect sense. Read on.

SIP Profiles

A SIP profile is just a couple address/port to which FreeSWITCH is listening.

Let's say our server has IP address 194.20.24.11.
The "internal" SIP profile in example configuration will have FreeSWITCH listen to port 5060. So, the "internal" SIP profile is listening at 194.20.24.11:5060. This is the address/port to which we will make our internal phones to connect, register, authenticate, and send calls to.
Also, in the "internal" SIP profile definition, there is a "param" item, which name is "context" and value is "public". Calls arriving at this address/port will be routed to the "public" context in dialplan.

The "external" SIP profile in demo configuration will be listening at 194.20.24.11:5080. This is the address/port to which we will ask our ITSP to connect and send calls to us (ITSPs do not register themselves, usually they just send calls to us). So, if you need to set this explicitly in your ITSP account, remember to set the "external" SIP profile port, that is 5080 in demo configuration.
Also, in the "external" SIP profile definition, there is a "param" item, which name is "context" and value is "public". Calls arriving at this address/port will be routed to the "public" context in dialplan.

Ouch! What's wrong? Both SIP profiles use the "public" dialplan context, but we saw before that internal users and calls are serviced by the "default" dialplan context. We'll see about this in a moment.

Dialplan Contexts

The "public" dialplan context contains extensions that can be safely reached by calls that are incoming from the wild external world and by non authenticated users. Those extensions do not give access to costly services (eg, outbound calls) or to exploitable features.

The "public" context is the one servicing inbound calls from DID numbers, sent to us by ITSPs. Also, this same "public" context is servicing whatever call comes from non-authenticated users.

It has very few features and extensions, just enough to connect an inbound call to one of our registered internal phones, or to services like voicemail and IVRs.
And it usually provides those features and services by "transferring" in an absolute and final way the call to an extension in the "default" dialplan context. We saw before how we set the call incoming from DID number to be "trasferred" to the "1012" extension in the "default" context of the "XML" dialplan (dialplan is always "XML" in example demo configuration). No way to go around, "transfer" is going to an absolute extension.

The "default" dialplan is where in example configuration most of the action is, and is servicing authenticated internal users. When a phone connects to the "internal" SIP profile and register itself, it is challenged to provide username and password. Those are checked against the User Directory. If the user is authenticated, it is then associated to all variables and parameters contained in its User Directory user definition, and in the groups and the domain it belongs to.

Amid those variables and parameters associated to an authenticated user in demo example configuration, we find the variable "user_context". That variable is set to "default". THAT'S IT!!!

So, an internal phone registers itself as a user, has its credentials checked against the User Directory, is successfully authenticated. If that user has a variable "user_context" defined in the User Directory, then all calls originated by that user (phone) will be routed by that defined dialplan context (overriding the context set in SIP profile).

Verto Profiles and Dialplan Contexts

Verto, in demo example configuration, always ask for authentication to phones (eg, browsers) that are registering. So, it behave like the "internal" SIP profile: the "user_context" variable associated to the authenticated user will be the dialplan context that will route the calls originated by that user (browser).

So, an incoming call from a DID number, sent by our ITSP to our FreeSWITCH server via SIP, can then be routed to the authenticated user registered via its browser as a Verto WebRTC client. Also, a call made by an authenticated Verto user connected via WebRTC browser will be routed by "default" dialplan context, and so it will be able to send outbound calls via the SIP gateway to the external world. This is the simplest possible way to implement a fully featured SIP/WebRTC bidirectional gateway.

ITSPs

Our Internet Telephony Service Provider (ITSP) will send the calls incoming from DID numbers to us on our "external" SIP profile (port 5080 in demo configuration) . We will send to the ITSP outbound calls in need of a SIP routing to the external world, using the "external" SIP profile. In dialplan we will use a SIP "gateway" syntax that refers to the authentication needed by our ITSP to accept our outbound traffic.

Summary

In this chapter, we discussed the following:

- How AAA (Authentication, Authorization, Accounting) concepts are implemented in FreeSWITCH
- How FreeSWITCH collects users into a directory
- How FreeSWITCH uses information fro User Directory and Dialplan to route calls from users to each other, and to the world
- Employing various user features such as voicemail
- Adding a new user and modifying the Dialplan accordingly
- Connecting to the outside world with gateways
- SIP and Verto profiles

In this chapter, we made modifications to the `default` XML Dialplan, and we learned how to set up users and domains within the XML user directory. Also, we saw how to setup SIP gateways and DID numbers to interconnect with the external world. In the next chapter we'll have a first look at WebRTC and Verto, powerful technologies that promise to seamlessly integrate browsers, mobile apps and the Internet of Things (IoT) into our FreeSWITCH communication platform.

WebRTC, SIP, and Verto

WebRTC, Real Time Communication in video and audio, is already working on more than a billion browsers. Soon mobile apps will multiply. And WebRTC is poised to be the multimedia communication layer adopted by the Internet of Things (IoT).

FreeSWITCH is a complete WebRTC solution, enabling preexisting and legacy applications to serve users via new channels. Also, FreeSWITCH being an uber-programmable application server, is the bedrock on which you can create the killer service that define a new concept of "get rich quick"!

WebRTC is a group of technologies and standards for peer-to-peer multimedia streaming. WebRTC needs an additional layer of signling and session management for actually be useful: user directory, user locations, failure management, etc etc

FreeSWITCH supports both SIP on WebRTC (the session signaling protocol that runs the telecommunication world) and VERTO, an open source new protocol designed to be easy for JavaScript developers.

We'll see how to install, configure and manage your FreeSWITCH server as an integrated WebRTC platform, we'll have a look at Verto Communicator (VC) the amazingly advanced and easy browser client for videoconferencing and telepresence.

Then we'll go through the development of the same application, WebRTC Rooms, in two indistinguishable flavors: one based on SIP.js and the other on verto.js.

In this chapter we will cover the following topics:

- WebRTC concepts
- Encryption in WebRTC
- WebRTC in FreeSWITCH
- SIP and Verto protocols rationale
- Installing and configuring a complete FreeSWITCH WebRTC platform
- Verto Communicator's amazing features
- How to write a test application in Verto and SIP.js

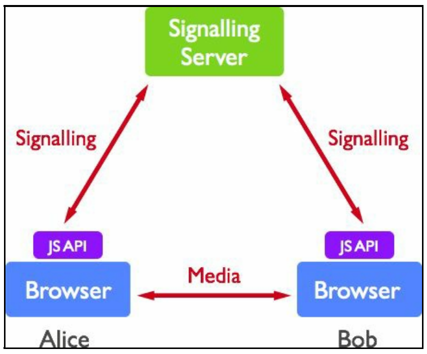
WebRTC concepts

WebRTC is a bundle of standards and technologies that enable **peer-to-peer** audio, video and data acquisition, streaming and exchange. Its first "killer app" is video calls and videoconferencing. Its first implementation is in web browsers (more than one billions WebRTC compatible browsers out there), its technology is already in use by many of smartphones' apps, and is predicated to be the base for Internet of Things (IoT) multimedia communication (will be many billions IoT devices WebRTC enabled).

In the most straightforward and popular implementation, the **browser** will access a website and load a page that contains **Javascript**. That script functions use **WebRTC APIs** (see below) to interact with the local computer multimedia hardware (microphone and camera) and to stream audio and video to/from peers.

The emphasis here is on **peer-to-peer**.WebRTC does not prescribes or support anything that goes beyond grabbing, sending, receiving, and playing streams. No signaling, no user directory or location services, no protocol to negotiate a communication session establishment and tear down, etc.

WebRTC is NOT a protocol, nor a service. Using only WebRTC technologies you must hardcode addresses in your application, on both the peers, and hope to find them listening, and the application will only work for those peers.



For anything but the simplest demo **WebRTC needs some form of signaling and session management protocols**. For your WebRTC service **you can adopt a ready made protocol, or invent your own**.

The most established signaling session management protocol is obviously **SIP**, which runs the entire world telecommunication network, from PSTN carriers to mobile operators to PBXs and calling services. Another signaling protocol in wide usage is **XMPP**, mostly for the management of instant messaging and presence services. There are plenty of **proprietary closed protocols**, most notably the one behind

Skype.
And there are **open source** alternatives geared toward Web Developers and JavaScript programmers such as **FreeSWITCH's Verto**. You will have to choose one, or write your own.

WebRTC technologies are embodied in three groups of Application Programming Interfaces (**APIs**) software developers use to build applications: GetUserMedia (GUM), RTCPeerConnection (RTCPC or PC), and RTCDataChannels (RTCDC or DC).

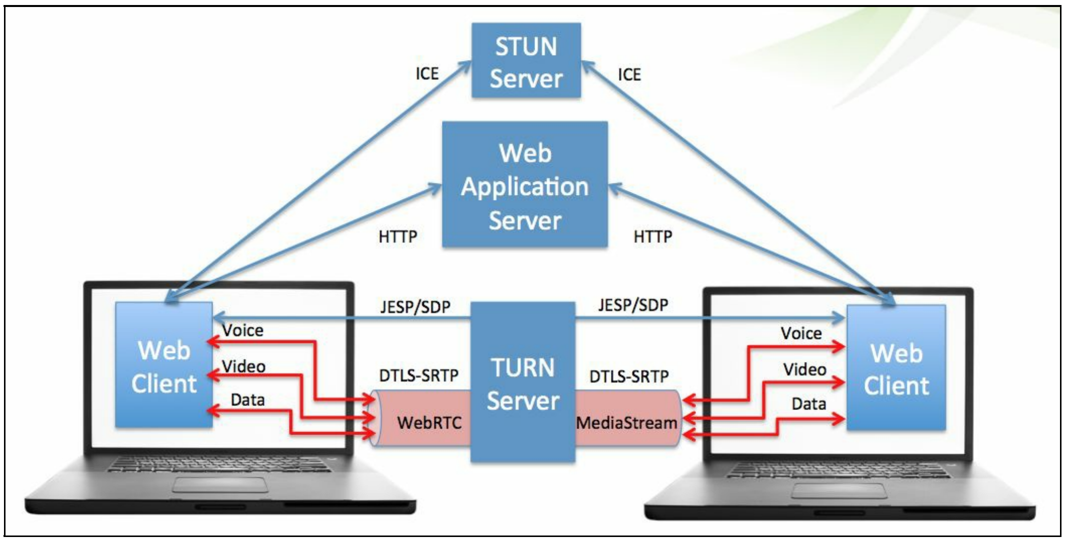
* **GetUserMedia** is about enumerating multimedia hardware on the local machine, asking user permission to control the microphone and camera, getting their streams, and playing streams through local screen and speakers

* **RTCPeerConnection** is about moving those audio and video streams between defined peers

* **RTCDataChannels** is about moving arbitrary data (even structured data) between defined peers

WebRTC RTCPeerConnection can (and must) use **ICE**, **STUN** and **TURN** to pass through firewalls and NATs, and eventually reach its peer.

ICE is a technique that has been popularized by WebRTC itself, and is gaining enormous traction. It is the real silver bullet for all networking problems, and works kind of perfect, being the distilled wisdom of many years of engineering VoIP services.



ICE will use a **STUN** server to tell the application what is its public address as seen from the Internet (eg, from outside NAT and firewall), and will use a **TURN** server as in-the-middle relay in the case peer-to-peer communication is not possible or fails.

So, a functional WebRTC application will need to:

* Use GetUserMedia API to control mic and camera

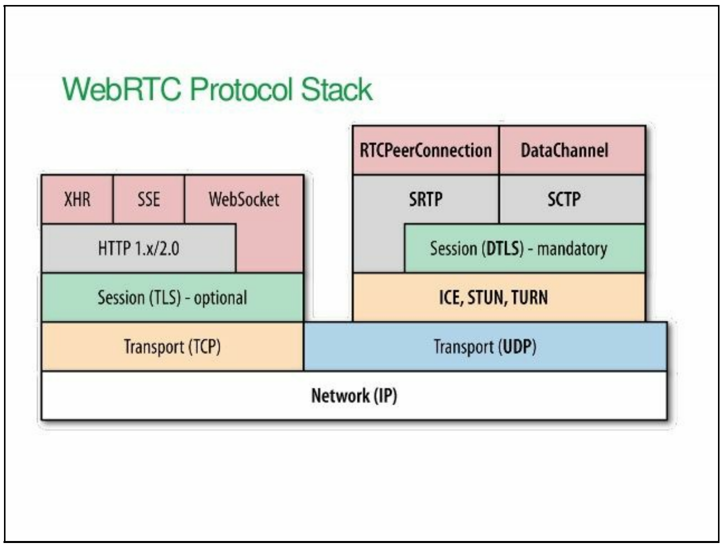
* Use a session protocol and possibly external auxiliary servers to locate the peer and establish a session

* Use ICE (and STUN and TURN) to find the network path to the peer

* Use RTCPeerConnection to stream audio and video to/from the peer

Encryption Everywhere

Encryption of media and data streams is mandatory. Full stop. There is no way you can have a WebRTC communication without encrypting the streams that are exchanged. Those would be the media streams (audio+video) and, if present, the data streams. Media streams are encrypted as **SRTP** with key exchange via **DTLS**. That is done for you by the RTCPeerConnection API.



But, while you're at it, it makes only sense to have the session protocol's signaling encrypted too. The transport of choice for such signaling is usually Secure WebSocket. You will see almost everywhere the URI prefix **WSS://** (WebSocket Secure) when defining WebRTC signaling servers and endpoints. **WSS** signaling exchange is encrypted by **TLS** (eg, like web **HTTPS**). The alternative, unencrypted, is simply **WS**, but is rarely used beyond testing purposes.

WebRTC in FreeSWITCH

OK, enough with abstractions, let's see under our belated hood, how WebRTC is implemented by FreeSWITCH.

FreeSWITCH is a complete WebRTC platform, and can act as both WebRTC gateway and WebRTC Application Server. Eg, FreeSWITCH is able to connect WebRTC clients to external "normal" SIP and PSTN services (or PBXes), enabling already existing or legacy services to be accessed by one billion browsers.

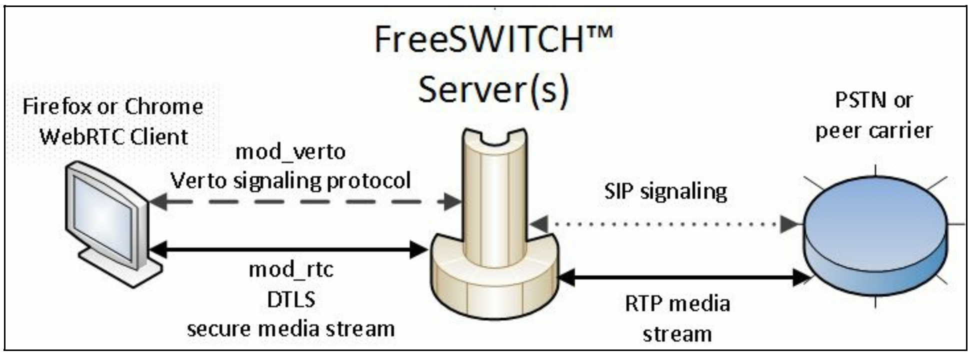
At the same time (and this can be literally true: while acting as gateway for some external services) FreeSWITCH is able to directly serve to WebRTC clients (browsers and apps) the exact same audio/video applications is serving to "normal" SIP and PSTN clients, such as PBX, IVR, Call Center, Conferencing System, Calling Card, etc applications.

To FreeSWITCH**, WebRTC is just another kind of communication channel, along with SIP,** TDM, MGCP, H323, Skype, Jingle, etc. All FreeSWITCH features are available via WebRTC, natively.

WebRTC Signaling in FS: SIP and Verto

As we've already seen in this chapter, WebRTC needs a session protocol to complement its own streaming capabilities. That protocol will use signaling to find the peer location, establish and tear down sessions, manage presence and messaging, etc. FreeSWITCH supports two such protocols: SIP and Verto.

These two different protocols only affect how is written the client application (client application is the web page and Javascript the browser load in order to access FreeSWITCH) and how it "talks" to FreeSWITCH. Both protocols give full access to FreeSWITCH features and services. Also, both signaling protocols can be used at the same time by different clients accessing the same service, and even interacting with each other (eg, we can have a videoconference in FreeSWITCH that's concurrently accessed by one WebRTC client "speaking" SIP and another "talking" Verto, in addition to other clients coming from TDM and "normal" (non-WebRTC) SIP:



Why SIP and why Verto?

Remember: this choice only impacts how you write the Javascript on the webpage the browser loads to access FreeSWITCH server. We'll see later in this chapter we can have the same exact application written for each one of protocols, indistinguishable from the end user's point of view and experience.

SIP albeit complex will be intimately known to all programmers coming from a telecom background, with its concepts of session, dialog, transaction, the familiar REGISTER-INVITE-BYE methods, etc. It's the perfect choice to integrate WebRTC clients in a complex environment, and enjoys 20 years of experience in scaling, monitoring, deploying.

Verto is a completely different experience: is very simple and it uses programming constructs already familiar to web programmers, such as JSON data structures. Also, Verto optionally provides for interaction with FreeSWITCH internal server side APIs and bidirectional synchronization of data structures.

For example, with Verto you can have a data structure in FreeSWITCH (let's say the list of videoconference participants) that's synchronized on all Verto clients transparently and in real time, so if a new participant comes in (even from PSTN) its presence will be immediately visualized by all Verto clients' browsers, without any need for any client side action (eg, no polling).
You can replicate this behavior in a SIP based JavaScript (a clever programmer can do whatever she wants, are not them all just bytes?) using data streams, special messaging, push techniques, and so on, but in Verto it's native and takes just a few lines of code.

WebRTC Media Streaming in FS: SRTP

Irrespectively of which session signaling protocol is used by WebRTC clients (SIP or Verto), their media streams will always be encrypted as mandated by WebRTC specs. That would be Secure RTP (**SRTP**), with encryption keys sent via **DTLS**. Actually inside FreeSWITCH the same module and source code will be used to encrypt and serve media streams to both kind of WebRTC clients.
As seen previously, WebRTC mandates how to stream and encrypt media, leaving the session protocol to the whim of the application implementer.

Installing and Configuring WebRTC in FS

For a quick test, Tristan Mahe contributed a script that installs all that's needed forWebRTC, from FreeSWITCH itself to certificates to webserver, to Verto Communicator, on a freshly deployed Debian 8 minimal Jessie server.
Install the base OS on hardware or on a virtual machine, ssh on that machine, execute thescript and you're ready to connect to your new WebRTC Conference Server:

https://freeswitch.org/confluence/display/FREESWITCH/Debian+8+Jessie#Debian8Jessie-QuickStartFreeSWITCHDemoWithVertoCommunicator
In following sections we will not use the automated install script. Instead we'll look at all the steps needed to build a complete WebRTC FreeSWITCH platform starting from a standard FS installation (eg, right after installing FreeSWITCH).

Actually there is no module to compile or install for adding WebRTC support on top of the standard FreeSWITCH deployment. All modules are already installed and ready to run. FreeSWITCH is WebRTC capable out of the box.

We'll need to edit several configuration files, so that FreeSWITCH finds the mandatory SSL certificates, know which ports to listen at, and many other details.

Certificates

This is one stumbling block for many FreeSWITCH/WebRTC newcomers. We'll need many certificate-related files, and different combination of them, to help encrypt different parts of FreeSWITCH traffic: "traditional" SIP on TLS (eg, "sips" as in classic RFC 3261), SIP on WSS, Verto on WSS, and SRTP media.

We need certificate files for the HTTPS server too, it will be serving the webpages and JavaScript loaded by browsers as our WebRTC clients.

We need the Certification Authority root and chain files.

And we'll need the private key too.

Also, original certificate files need to be concatenated in different ways to form the different ready-to-use certs we'll need.

No wonder many find this confusing.

I use one easy solution: I put all the certificate-related files in one only directory, and have both the different FreeSWITCH modules and the webserver (Apache or Nginx) to fetch them from there.

I also have a script to concatenate them in the correct way. You can use it directly (if your certificate comes from the same provider as mine), or easily modify it to suit the files sent you by your certificate provider.

First of all: forget about self-signed certificates, and similar souped-up solutions. Self-signed certificates "may" work for testing purposes, but is such a PITA to have all the moving parts correctly coordinated, it is not worth at all, and a sure recipe for a lot of time wasted and frustration. Simply put: DON'T. Use real, valid certificates for real, valid domain names (eg, no IP addresses).

If you are going to have your server(s) to answer requests for one or a few domains, you're lucky. Certificates can be obtained instantly and for free from letsencrypt.org.

If you want your server(s) to answer for all possible subdomains of a main domain (eg www.mydomain.com, sip.mydomain.com, support.mydomain.com, sip.mycustomername.mydomain.com, www.mysecondcustomername.mydomain.com, etc) you need to buy a wildcard certificate. No free options exist at the moment for wildcard certificates, but this may change in future.

In case of free (and perfectly valid) certificates made via www.letsencrypt.org (check their website for instructions):

```
#!/bin/sh
cp /etc/letsencrypt/live/my.fqdn.com/* /usr/local/freeswitch/certs/
cd /usr/local/freeswitch/certs/
cat fullchain.pem privkey.pem > wss.pem
cat cert.pem privkey.pem > agent.pem
cat chain.pem > cafile.pem
```

In case of PositiveSSL wilcard certificates issued by Comodo (google around for rock bottom prices), note you must find and download the *addtrustexternalcaroot.crt* file:

```
#!/bin/sh
cd /usr/local/freeswitch/certs/
cp myserver.key privkey.pem
cp STAR_mydomain_com.crt cert.pem
cp STAR_mydomain_com.ca-bundle chain.pem
cat cert.pem chain.pem addtrustexternalcaroot.crt > fullchain.pem
cat cert.pem privkey.pem fullchain.pem > wss.pem
cat fullchain.pem privkey.pem > agent.pem
cat chain.pem > cafile.pem
```

Installing and Configuring Apache or NginX for HTTPS serving

We need a webserver able to feed pages via TLS/SSL to our WebRTC browser clients. Whatever webserver that's able to speak HTTPS will do. The two most popular webserver at the moment are Apache and NginX. Let's look at how to install them and enable HTTPS/SSL, on Debian 8 (Jessie).

We will then be ready to put the files we want to serve via HTTPS in the directory /var/www/html/, both in Apache and NginX case.

You only need Apache or NginX, no need to install both of them.

Apache

The first step will be to update the package database, and install the apache2 and ca-certificates packages. The latter package will install the Certification Authorities certs, which may be useful when dealing with SSL.

Then we enable the SSL Apache module, and the default SSL site served via HTTPS.

In the following two lines (here broken over three lines each for easy reading) we use PERL to substitute the filenames and path of SSL/TLS certificate and private key.

In final line we restart Apache.

```
apt-get update
apt-get install apache2 ca-certificates
a2enmod ssl
a2ensite default-ssl.conf
perl -i -pe \
's|/etc/ssl/certs/ssl-cert-snakeoil.pem|/usr/local/freeswitch/certs/cert.pem|g' \ /etc/apache
perl -i -pe \
's|/etc/ssl/private/ssl-cert-snakeoil.key|/usr/local/freeswitch/certs/privkey.pem|g' \
/etc/apache2/sites-enabled/default-ssl.conf
service apache2 restart
```

NginX

Similar to the Apache installation, we update the package database, and install the nginx and ca-certificates packages. The latter package will install the Certification Authorities certs, which may be useful when dealing with SSL.

Then we enable listening to the 443 port (SSL), and include the snippet for the default SSL site served via HTTPS.

In the following two lines (here broken over three lines each for easy reading) we use PERL to substitute the filenames and path of SSL/TLS certificate and private key.

In last line we restart NginX.

```
apt-get update
apt-get install nginx ca-certificates
perl -i -pe 's/# listen/listen/g' /etc/nginx/sites-enabled/default
perl -i -pe \
's/# include snippets\/snakeoil.conf/include snippets\/snakeoil.conf/g' \
/etc/nginx/sites-enabled/default
perl -i -pe \
's|/etc/ssl/certs/ssl-cert-snakeoil.pem|/usr/local/freeswitch/certs/cert.pem|g' \
/etc/nginx/snippets/snakeoil.conf
perl -i -pe \
's|/etc/ssl/private/ssl-cert-snakeoil.key|/usr/local/freeswitch/certs/privkey.pem|g' \
/etc/nginx/snippets/snakeoil.conf
service nginx restart
```

Configuring VERTO

On a default FreeSWITCH installation you find the mod_verto configuration in /usr/local/freeswitch/conf/autoload_configs/verto.conf.xml. We will edit the profile "default-v4", for IPv4, eg, normal Internet connectivity. You want to check the **TCP port** where VERTO will listen for signaling on WSS transport. Note that **secure="true"indicates WSS** (as opposed to simple WS).

```
        <param name="bind-local" value="$${local_ip_v4}:8082" secure="true"/>
        <param name="ext-rtp-ip" value="93.58.44.181"/>
        <param name="debug" value="0"/>
```

The value of ext-rtp-ip sets the IP address VERTO will tell the WebRTC clients they must connect to in order to exchange media streams. It must be set to FreeSWITCH IP address as seen from the WebRTC clients. So for clients coming from the Internet, ext-rtp-ip must be set to the external side of the NAT, eg to the routable IP address (often same as the webserver IP address).

You may want at least temporarily bring the value of "debug" to 10, so all VERTO exchanges will be visible (in red!) on the FreeSWITCH console and on fs_cli. Bringing up mod_verto debug, together with the javascript console on the client browser, will give you complete trace of what signaling is exchanged.

Configuring SIP on WebRTC (WSS)

On a default FreeSWITCH installation you only need to edit the "internal" SIP profile, in /usr/local/freeswitch/conf/sip_profiles/internal.xml. You want to check **"wss-binding"** for the **TCP port** where SIP (mod_sofia) will listen for signaling on WSS transport.

```
<param name="wss-binding" value=":7443"/>
<param name="ext-rtp-ip" value="93.58.44.181"/>
<param name="ext-sip-ip" value="93.58.44.181"/>
```

The values of ext-rtp-ip and ext-sip-ip sets the IP address SIP will tell the WebRTC clients they must connect to in order to exchange signaling and media streams. It must be set to FreeSWITCH IP address as seen from the WebRTC clients. So for clients coming from the Internet, ext-rtp-ip must be set to the external side of the NAT, eg to the routable IP address (often same as the webserver IP address).

Writing WebRTC Clients

A WebRTC client, in its most popular implementation, is an HTML webpage(s) that loads a JavaScript(s). Together, HTML and JavaScript define the GUI and the behavior of the WebRTC client.

Usually the JavaScript part, loaded by the HTML, leverages one or more JavaScript libraries. Those libraries implement the signaling protocol of choice (in our case SIP or VERTO) and its interaction with WebRTC APIs.
We have already seen how WebRTC APIs provide for accessing the local computer multimedia hardware (microphone and camera), manage the peer-to-peer streaming of audio and video with the peer, and a bidirectional data channel.

The session signaling protocol will leverage and complement those WebRTC P2P capabilities, so they become useful for much more than a connection to a pre-known address and port.

SIP and JavaScript

SIP for WebRTC has been notably implemented in theJsSIP JavaScript Open Source library. JsSIP was written by José Luis Millán, Iñaki Baz Castillo, and Saúl Ibarra Corretgé aka the **Basque VoIP Mafia**, also co-authors of RFC 7118 that defines the use of WS and WSS for SIP:

JsSIP was then forked and further developed by Will Mitchell and its team, while working at OnSIP. The result as been released as open source **SIP.js** library. SIP.js as of today has widespread adoption, and is the most used Javascript WebRTC SIP library in FreeSWITCH Community:

Another notable and independent (actually, the first SIP client for HTML5) opensource SIP library implementation in JavaScript is sipml5, developed by Doubango Telecom.

Each one of these libraries is readily available on the Internet, with full documentation, example clients, etc. The most actively developed and supported seems to be SIP.js. We'll look at SIP.js in action later in this chapter, implementing the WebRTC Rooms application.

Verto and JavaScript

The VERTO protocol is a quintessential JSON protocol, and is available to JavaScript programmers by including one of the following library "flavors".

First flavor is verto-min.js the already minified base library distributed together with FreeSWITCH sources, a plain "raw" JavaScript library, depending on jquery and jquery.json.
The other flavor is a "verto" plugin for JQuery, available through the "npm" Node Package Manager.

The VERTO library has been written by Anthony Minessale II, the Lead Developer and Primary Author of FreeSWITCH in parallel with mod_verto, during the development of FreeSWITCH 1.5 (1.5 was the "master" git code leading to the release of FreeSWITCH 1.6).

The **Verto JQuery plugin** and its documentation were further developed by Italo Rossi and its team at Evolux.net.br (they're developers of Verto Communicator too, see below), and contributed by open source wiz Chad Phillips (hunmonk on IRC). You'll find complete Verto JQuery plugin documentation, and step by step programming instruction with client examples at http://evoluxbr.github.io/verto-docs/.

We'll use the "raw" original **verto-min.js** library in our **WebRTC Rooms** application, later, for the sake of simplicity.

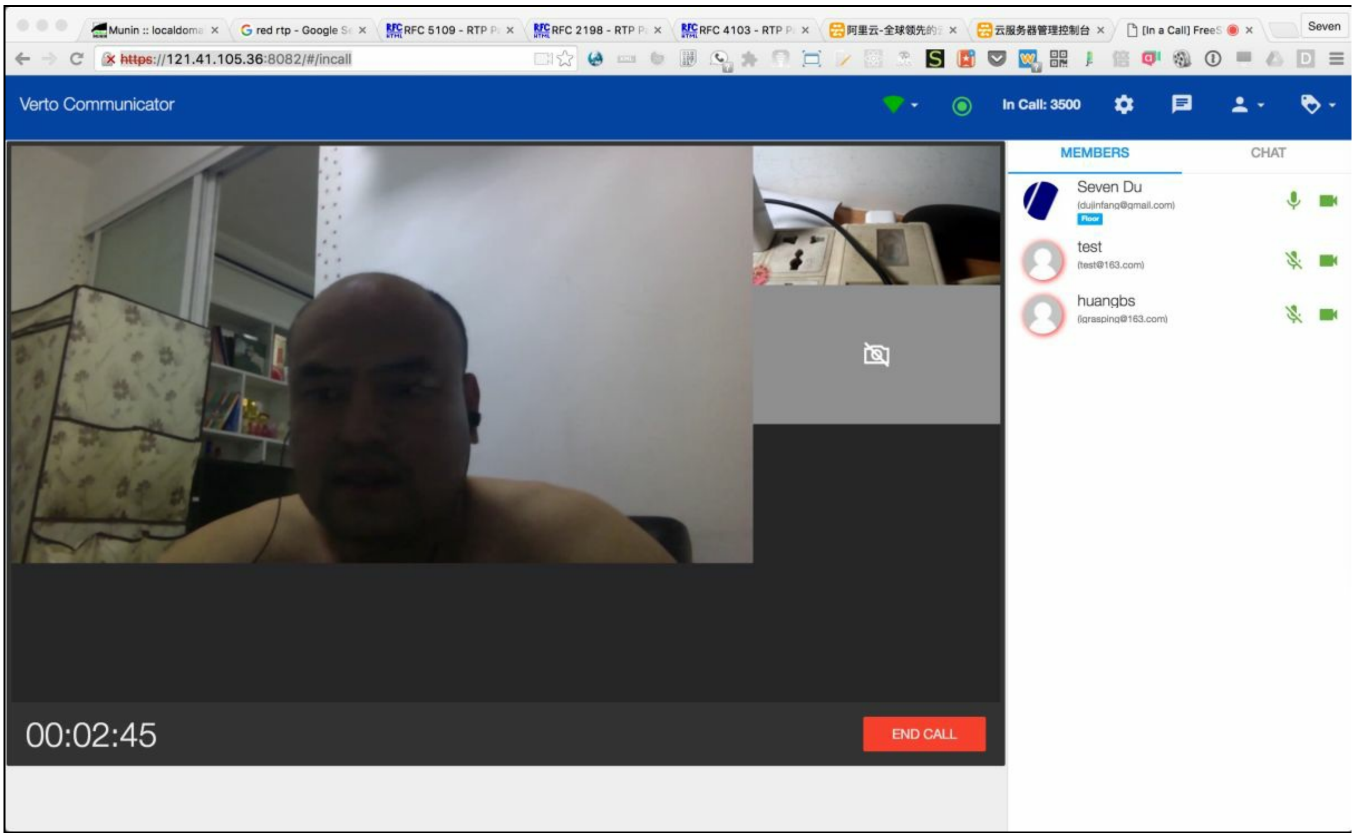
Verto Communicator, an Advanced WebRTC Client

Verto Communicator is a cool example of a complete advanced opensource WebRTC VERTO client with additional features for Videoconferencing, written by **Italo Rossi** and its team at Evolux.net.br.

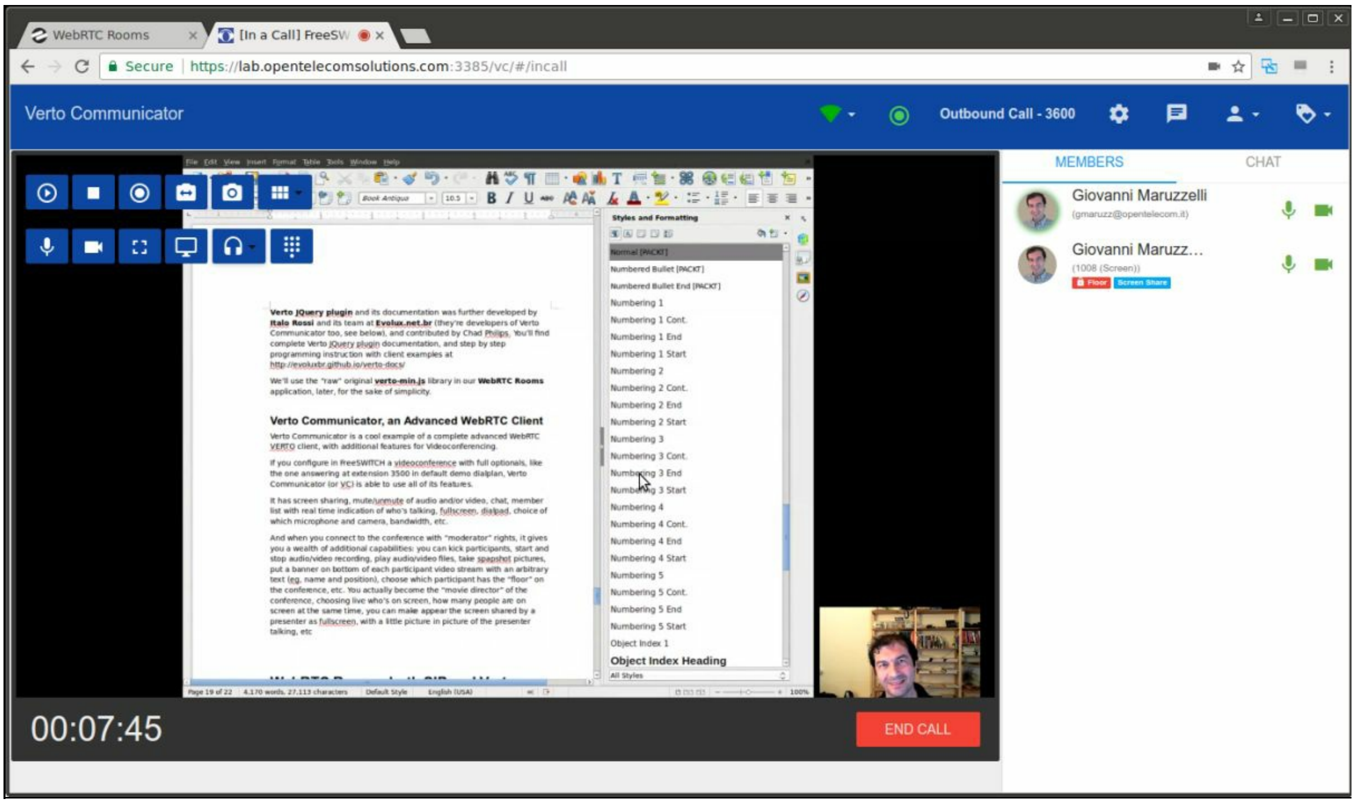
If you configure in FreeSWITCH a videoconference with full optionals, like the one answering at extension **3500** in default demo dialplan, Verto Communicator (or **VC**) is able to use all of its features.

It has screen sharing, mute/unmute of audio and/or video, chat, member list with real time indication of who's talking, automatic visualization of gravatar, fullscreen, dialpad, choice of which microphone and camera, bandwidth, etc.

In picture you can see a VC screenshot made by **Seven Du Jin Fang**, FreeSWITCH guru, writer of FS video code among many other things, and leader of the burgeoning FreeSWITCH Chinese community. He's on floor, other participants begin to arrive and are automatically positioned around him:



And when you connect to the conference with "**moderator**" rights (eg, "|moderator" flag), it gives you a wealth of additional capabilities: you can kick participants, start and stop audio/video recording, play audio/video files, take spapshot pictures, put a banner on bottom of each participant video stream with an arbitrary text (eg, name and position), choose which participant has the "floor" on the conference, etc. You actually become the "movie director" of the conference, choosing live who's on screen, how many people are on screen at the same time, you can make appear the screen shared by a presenter as fullscreen, with a little picture in picture of the presenter talking, etc. Picture made right now :).



Verto JQuery plugin and its documentation was further developed by **Italo Rossi** and its team at **Evolux.net.br** (they're developers of Verto Communicator too, see below), and contributed by Chad Philips. You'll find complete Verto JQuery plugin documentation, and step by step programming instruction with client examples at http://evoluxbr.github.io/verto-docs/

We'll use the "raw" original **verto-min.js** library in our **WebRTC Rooms** application, later, for the sake of simplicity.

Verto Communicator, an Advanced WebRTC Client

Verto Communicator is a cool example of a complete advanced WebRTC VERTO client, with additional features for Videoconferencing.

If you configure in FreeSWITCH a videoconference with full optionals, like the one answering at extension 3500 in default demo dialplan, Verto Communicator (or VC) is able to use all of its features.

It has screen sharing, mute/unmute of audio and/or video, chat, member list with real time indication of who's talking, fullscreen, dialpad, choice of which microphone and camera, bandwidth, etc.

And when you connect to the conference with "moderator" rights, it gives you a wealth of additional capabilities: you can kick participants, start and stop audio/video recording, play audio/video files, take spapshot pictures, put a banner on bottom of each participant video stream with an arbitrary text (eg, name and position), choose which participant has the "floor" on the conference, etc. You actually become the "movie director" of the conference, choosing live who's on screen, how many people are on screen at the same time, you can make appear the screen shared by a presenter as fullscreen, with a little picture in picture of the presenter talking, etc

To install Verto Communicator on your Debian 8 Jessie server, git clone FreeSWITCH sources, and:

```
cd /usr/src
git config --global url."https://".insteadOf git://
git clone https://freeswitch.org/stash/scm/fs/freeswitch.git -bv1.6 freeswitch
cd /usr/src/freeswitch/html5/verto/verto_communicator/
./debian8-install.sh
ln -s /usr/src/freeswitch/html5/verto/verto_communicator/dist /var/www/html/vc
```

You then want to edit /var/www/html/vc/config.json as per your values (edit password to be the same as in /usr/local/freeswitch/conf/vars.xml):

```
{
      "login": "1008",
      "password": "mydefaultpasswordfromvars.xml",
      "wsURL": "wss://lab.opentelecomsolutions.com:8082"
}
```

You can find a more complete example of config customization in /usr/src/freeswitch/html5/verto/verto_communicator/src/config.json.sample, made by our precious Ken Rice, FreeSWITCH Core Team developer.

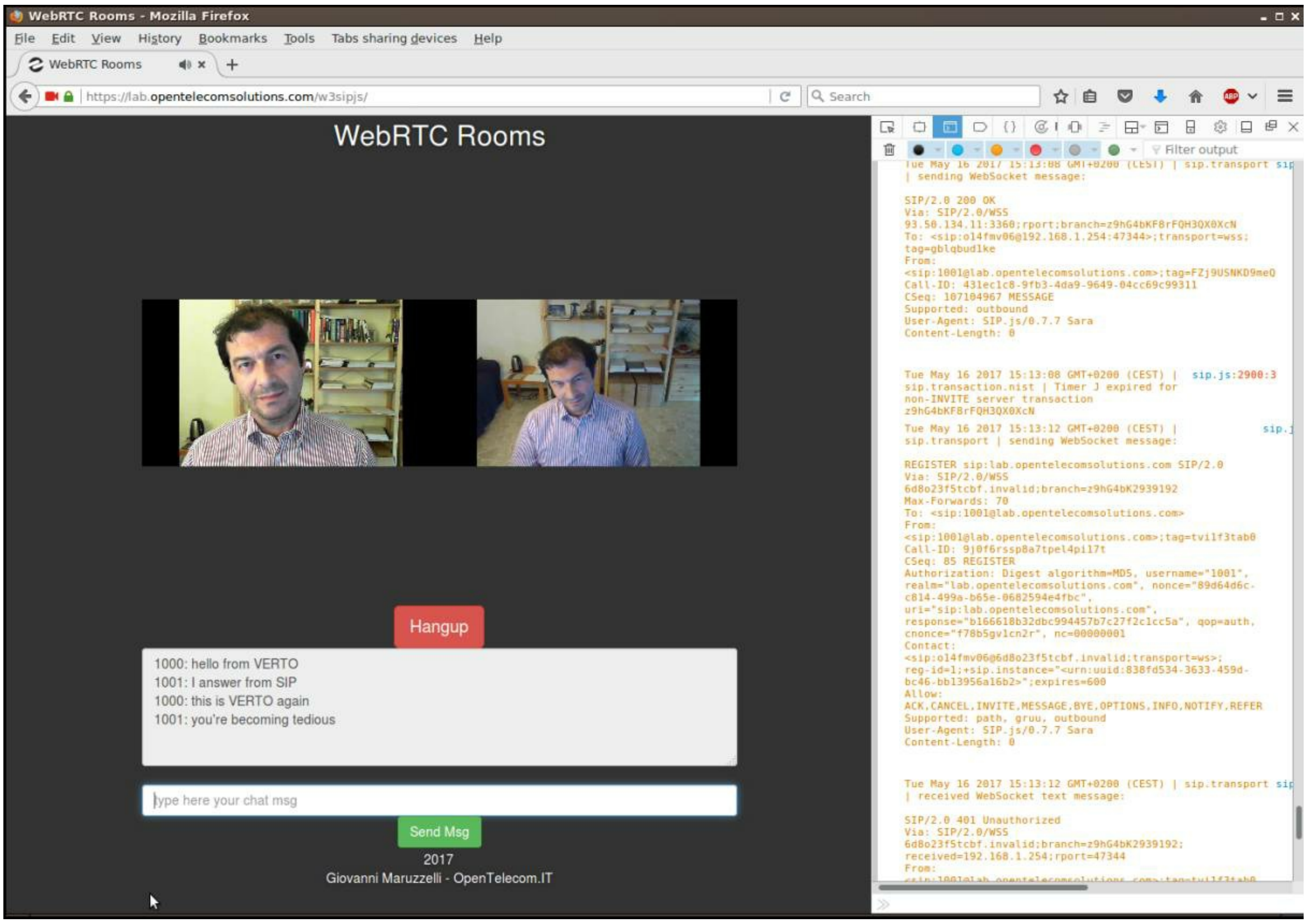
VC is covered again in Chapter 13, *Conferencing and WebRTC Video-Conferencing*.

WebRTC Rooms, both SIP and Verto Clients

Let's have a look at a simple project, that I hope you will be able to use for tests, production, and as a base for further developments.
Btw, an earlier version of WebRTC Rooms (from "Mastering FreeSWITCH" 2016 Packt book) has been adapted by Len Graham as VERTO client plugin for the excellent FusionPBX (www.fusionpbx.com : Mark Crane is the Lead Developer of FusionPBX, a complete web interface for configuring and managing a FreeSWITCH based PBX, industrial grade).

WebRTC Rooms is a basic web client able to make and receive videocalls, send DTMFs during the calls, and interface with the chatting system of FreeSWITCH conferences.
There are two versions of it, differentiated by which signaling session protocol is implemented by the JavaScript part. One version uses SIP and the other uses VERTO. The interesting bit is that both versions have the exact same HTML, user interface, behavior, user experience, etc. Actually, they are indistinguishable if you don't fire up the JavaScript console in the browser.

Here's a screenshot from the SIP.js flavor (you can see SIP messages in javascript console), later you'll see the VERTO flavor:


WebRTC Rooms - Mozilla Firefox
File Edit View History Bookmarks Tools Tabs sharing devices Help
WebRTC Rooms
https://lab.opentelecomsolutions.com/w3sipjs/
Search
Filter output
Hangup
Send Msg

WebRTC Rooms

1000: hello from VERTO
1001: I answer from SIP
1000: this is VERTO again
1001: you're becoming tedious

type here your chat msg

2017
Giovanni Maruzzelli - OpenTelecom.IT

```
Tue May 16 2017 15:13:08 GMT+0200 (CEST) | sip.transport sip
| sending WebSocket message:

SIP/2.0 200 OK
Via: SIP/2.0/WSS
93.50.134.11:3360;rport;branch=z9hG4bKF8rFQH3QX0XcN
To: <sip:o14fmv06@192.168.1.254:47344>;transport=wss;
tag=gblqbudlke
From:
<sip:1001@lab.opentelecomsolutions.com>;tag=FZj9USNKD9meQ
Call-ID: 431ec1c8-9fb3-4da9-9649-04cc69c99311
CSeq: 107104967 MESSAGE
Supported: outbound
User-Agent: SIP.js/0.7.7 Sara
Content-Length: 0



Tue May 16 2017 15:13:08 GMT+0200 (CEST) |  sip.js:2900:3
sip.transaction.nist | Timer J expired for
non-INVITE server transaction
z9hG4bKF8rFQH3QX0XcN
Tue May 16 2017 15:13:12 GMT+0200 (CEST) |          sip.j
sip.transport | sending WebSocket message:

REGISTER sip:lab.opentelecomsolutions.com SIP/2.0
Via: SIP/2.0/WSS
6d8o23f5tcbf.invalid;branch=z9hG4bK2939192
Max-Forwards: 70
To: <sip:1001@lab.opentelecomsolutions.com>
From:
<sip:1001@lab.opentelecomsolutions.com>;tag=tvi1f3tab0
Call-ID: 9j0f6rssp8a7tpel4pil7t
CSeq: 85 REGISTER
Authorization: Digest algorithm=MD5, username="1001",
realm="lab.opentelecomsolutions.com", nonce="89d64d6c-
c814-499a-b65e-0682594e4fbc",
uri="sip:lab.opentelecomsolutions.com",
response="b166618b32dbc994457b7c27f2c1cc5a", qop=auth,
cnonce="f78b5gv1cn2r", nc=00000001
Contact:
<sip:o14fmv06@6d8o23f5tcbf.invalid;transport=ws>;
reg-id=1;+sip.instance="<urn:uuid:838fd534-3633-459d-
bc46-bb13956a16b2>";expires=600
Allow:
ACK,CANCEL,INVITE,MESSAGE,BYE,OPTIONS,INFO,NOTIFY,REFER
Supported: path, gruu, outbound
User-Agent: SIP.js/0.7.7 Sara
Content-Length: 0



Tue May 16 2017 15:13:12 GMT+0200 (CEST) | sip.transport sip
| received WebSocket text message:

SIP/2.0 401 Unauthorized
Via: SIP/2.0/WSS
6d8o23f5tcbf.invalid;branch=z9hG4bK2939192;
received=192.168.1.254;rport=47344
From:
```

Conference chat in WebRTC Rooms

Usually the conference chat feature is only available in VERTO WebRTC clients, where is given almost for free by the capability of VERTO protocol to keep data structures in sync and distribute events to all connected clients. Verto clients just implements callbacks when a chat event is received.

SIP protocol does not include provisions for syncing of data structures or for events distribution. SIP is a completely different world.

The SIP WebRTC Rooms client has been implemented to only exchange "traditional" SIP SIMPLE messages, it does not use WebRTC or JavaScript techniques to get the chat events, but only uses SIP techniques. Actually it uses the same SIP instant messaging techniques as "traditional" SIP hard and soft phones (and we'll see with WebRTC Rooms that the conference chat messages can be received and sent by "traditional" SIP hard and soft phones too).

As a "bridge" to exchange the conference chat messages with the SIP conference participants we'll use the internal FreeSWITCH CHAT API, and also mod_sms and its chatplan.

Mod_sms, as we'll see later, is able to intercept and route (according to its chatplan) all messages related to the internal CHAT API interface of FreeSWITCH. The conference FreeSWITCH module (mod_conference) send out the chat messages to conference participants using the internal FreeSWITCH CHAT API. Also, SIMPLE SIP messages sent by SIP clients can be received by mod_sms and routed by its chatplan (as incoming voice and video calls are managed by the FreeSWITCH dialplan).

So, it was enough to register the SIP clients with the CHAT API of the conference so they receive chat messages from the conference, and set a chatplan so mod_sms will route the incoming SIMPLE SIP messages sent by SIP clients into the CHAT API of the conference.

By the way, in principle this same mechanism is working also for conference participants coming via mod_dingaling, mod_gsmopen and other modules that implement the CHAT FreeSWITCH API.

FreeSWITCH Conferences

Conferences in FreeSWITCH have oh, so many features. In particular, in the demo configuration there is one conference "**profile**" that's fully packed with all the goodies. It's called "**video-mcu-stereo**", and is defined in /usr/local/freeswitch/conf/autoload_configs/conference.conf.xml.

Conferences using the video-mcu-stereo profile are available at extensions 3500-3599 of demo configuration, as "**cdquality_conferences**".

The most important characteristics are the ability to mix the video streams incoming from participants, allow for screen sharing, apply video effects such as captions, chroma keys, picture in picture, video playings, and so on. Also, it sports a complete system for chatting and for displaying real time presence/status of participants.

We'll mainly use **extension 3500 for testing**, but if you're looking for something more frugal and light on your server CPU, look for extensions 3000-3099 ("nb_conferences" extensions, "default" conference profile).

mod_sms and chatplan

Chatplan is the dialplan for chats. Chatplan is the new black! First of all you must edit /usr/local/freeswitch/conf/autoload_configs/modules.conf.xml and **uncomment** mod_sms loading at FS startup, as follows:

```
    <load module="mod_sms"/>
```

Then you restart FreeSWITCH, or manually load mod_sms from fs_cli:

```
    load mod_sms
```

Now, we must edit the chatplan in /usr/local/freeswitch/conf/chatplan/default.xml :

```
<?xml version="1.0" encoding="utf-8"?>
<include>
  <context name="public">
    <extension name="demo">
      <condition field="to" expression="^(3\d{3})@.*$">
        <action application="info"/>
        <action application="set" data="to=${to_user}-${to_host}"/>
        <action application="set" data="dest_proto=conf"/>
        <action application="set" data="chat_proto=conf"/>
        <action application="set" data="skip_global_process=true"/>
        <action application="set" data="from=${from_user}"/>
        <action application="set" data="final_delivery=true"/>
        <action application="set" data="from_full=<sip:${from_user}@${from_host}>"/>
        <action application="info"/>
        <action application="send"/>
        <action application="stop"/>
        <action application="info"/>
      </condition>
    </extension>
  </context>
</include>
```

This chatplan will intercept the SIMPLE SIP instant messages sent to the conference extension (regular expression beginning with "3", then three times a number, then "@" something) and route them to the conference chat itself (the "to" field mimics the internal FreeSWITCH routing of conference chat events).
We then shape the "from" fields, so they look better when displayed in conference chat.

Before and after those transformations, we output to the FreeSWITCH console all the message variables (same as the "info" application in dialplan): before and after the treatment.
This chatplan will allow for a bridge between incoming SIP instant messages and conference chat events.

Don't forget to reload the FreeSWITCH configuration from fs_cli:

```
reloadxml
```

Dialplan "chat_proto" extension

Inside mod_conference (conference_event.c file) there is a mechanism that broadcasts all chat messages in a conference to all conference participants (avoiding double send).

Messages are sent to the "presence_id" of the participants clients, but only if that client channel has a "chat_proto" variable set to a valid value (eg, mod_conference needs to know which argument must be given to the CHAT FS API for that particular "presence_id"). Many endpoint modules have their own chat protocol, you can use it even from fs_cli:

```
freeswitch@lxc111> show api chat
name,description,syntax,ikey
chat,chat,<proto>|<from>|<to>|<message>|[<content-type>],mod_dptools
```

Anyway, the "chat_proto" variable needed by mod_conference to broadcast messages is automatically set only if channel is coming from mod_verto. So? So, without that chat_proto variable the conference participants coming from SIP does not get chat messages sent to them. We need to redress this discrimination.

Let's edit the dialplan in /usr/local/freeswitch/conf/dialplan/default.xml , and insert the following near the beginning of the file:

```
    <extension name="chat1" continue="true">
      <condition field="${source}" expression="^mod_sofia$">
        <action application="set" data="chat_proto=sip"/>
      </condition>
    </extension>
```

This extension will add the variable "chat_proto" with value "sip" to all call incoming from mod_sofia, that is to all SIP calls, included the WebRTC calls on WSS SIP transport.
Then, thanks to the "continue=true" parameter, the dialplan goes on,looking for other extensions to evaluate.

HTML

Hah, now we're entering the inner sanctum of our applications. Actually the HTML (index.html) is the same for both the SIP.js and the VERTO version. The only thing to change is which JavaScript files to load.

The VERTO flavor is as follows:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!-- The above 3 meta tags *must* come first in the head; any other head content must com
    <meta name="description" content="A WebRTC client for Verto FreeSWITCH module">
    <meta name="author" content="Giovanni Maruzzelli">
    <link rel="icon" href="favicon.ico">
    <title>WebRTC Rooms</title>
    <!-- Bootstrap core CSS -->
    <link href="css/bootstrap.min.css" rel="stylesheet">
    <!-- Custom styles for this template -->
    <link href="high.css" rel="stylesheet">
</head>
<body>
    <div id="conference">
        <input type="hidden" id="passwd" value="mysecretpassword_LOL" />
        <input type="hidden" id="cidnumber" value="WebRTC" />
        <div class="form-signin">
            <h2 class="form-signin-heading">WebRTC Rooms</h2>
            <div id="content" class="form-signin-content">
                <input type=number id="login" min=1000 max=1019 step=1 class="form-control" p
                <button class="btn btn-lg btn-primary btn-success" data-inline="true" id="log
                <input id="ext" class="form-control" placeholder="Insert the Extension to Cal
                <button class="btn btn-lg btn-primary btn-success" data-inline="true" id="cal
                <button class="btn btn-lg btn-primary btn-danger" data-inline="true" id="back
            </div>
            <div id="video1" align="center" class="embed-responsive embed-responsive-4by3">
                <video id="webcam" autoplay="autoplay" class="embed-responsive-item"> </video
            </div>
            <button class="btn btn-lg btn-primary btn-danger" data-inline="true" id="hupbtn">
            <br id="br" />
            <textarea id="chatwin" class="form-control" rows="5" readonly></textarea>
            <br id="br" />
            <input id="chatmsg" class="form-control" rows="1" placeholder="type here your cha
            <button class="btn btn-primary btn-success" data-inline="true" id="chatsend">Senc
        </div>
        <div align="center" class="inner">
            <p>2017<br/>Giovanni Maruzzelli - OpenTelecom.IT</p>
        </div>
    </div>
    <script type="text/javascript" src="js/jquery.min.js"></script>
    <script type="text/javascript" src="js/jquery.json-2.4.min.js"></script>
    <script type="text/javascript" src="js/verto-min.js"></script>
    <script type="text/javascript" src="js/md5.js"></script>
    <script type="text/javascript" src="high2.js"></script>
</body>
</html>
```

This file uses the bootstrap CSS framework, so it works perfectly on mobile and is reactive to different display format and sizes. The first few lines serve just for this, to setup the minimalist but quick GUI interface.

Then, in the HTML body, we have a lot of DIVs, BUTTONs, INPUTs, TEXTAREAs, each of them with its own "id".

We will use that "id" value from Javascript, to refer to each single HTML element when we want to show or hide the item (eg, the login button disappear after you register to the server), or to read or write its input and output area (we want to read what extension the user want to call, and we want to display chat messages in the chat textarea).

We have the special "video" element, which bootstrap.css displays in a beautyful way on any device. Its "id" will be referenced by both the VERTO and SIP.js initialization scripts (eg, they need to know where to display the video stream).

The last few lines are the scripts to load.

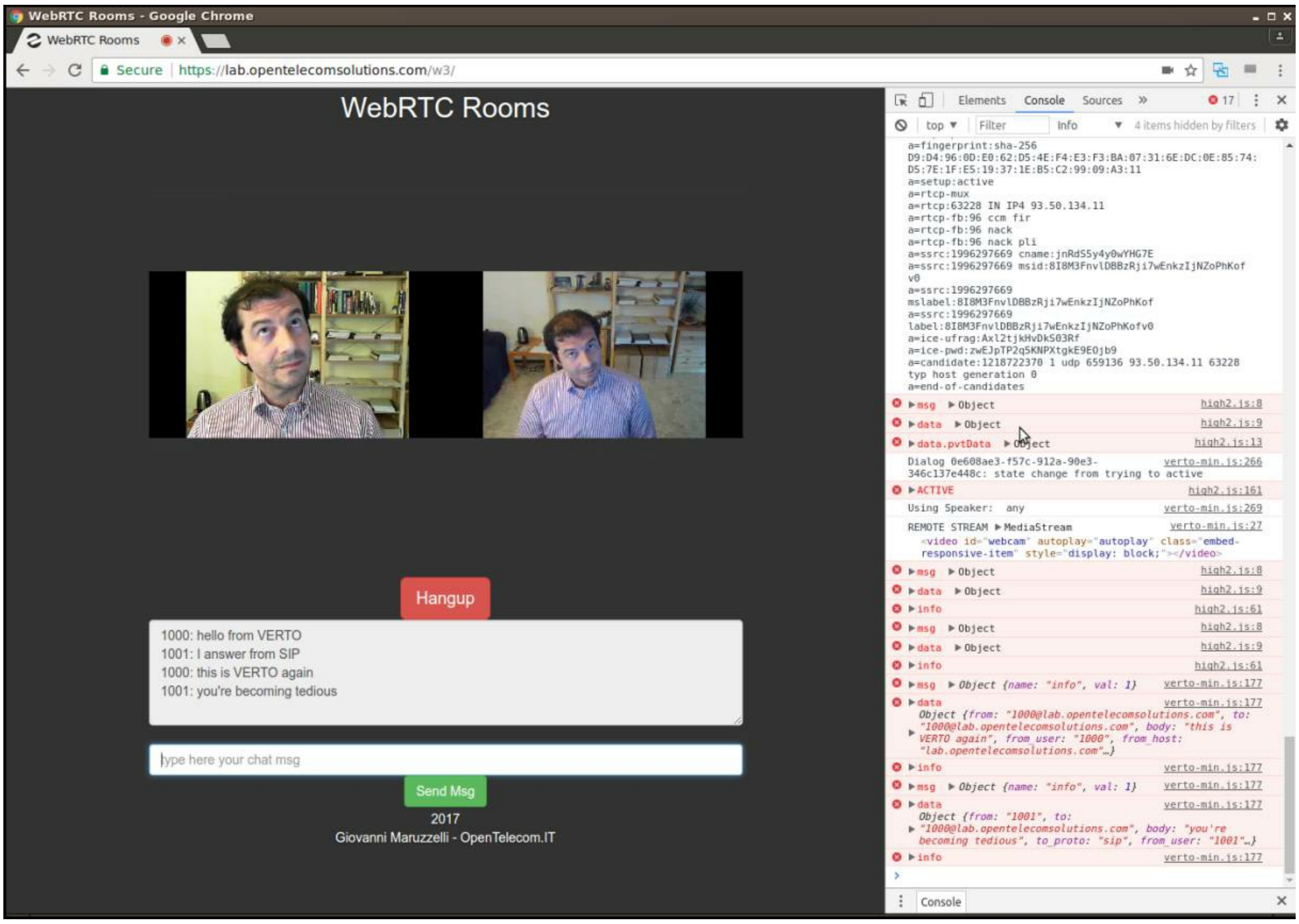
We recognize verto-min.js from /usr/src/freeswitch/html5/verto/video_demo/js/verto-min.js, one flavor of the VERTO JavaScript library distributed in FreeSWITCH sources.
Our script, the one that interact with the HTML elements' "id" is named high2.js.

The final lines of index.html SIP.js version load different Javascript:

```
    <script type="text/javascript" src="js/jquery.min.js"></script>
    <script type="text/javascript" src="js/jquery.json-2.4.min.js"></script>
    <script type="text/javascript" src="js/sip.js"></script>
    <script type="text/javascript" src="js/md5.js"></script>
    <script type="text/javascript" src="highsipjs2.js"></script>
</body>
</html>
```

We recognize the **sip.js** library (linked to the latest release, available at http://sipjs.com/download/ , while our script is highsipjs2.js

Here's a screenshot of the VERTO flavor, with Verto objects in Javascript console


WebRTC Rooms - Google Chrome
WebRTC Rooms
Secure
https://lab.opentelecomsolutions.com/w3/
Elements
Console
Sources
17
top
Filter
Info
4 items hidden by filters
Hangup
Send Msg

WebRTC Rooms

1000: hello from VERTO
1001: I answer from SIP
1000: this is VERTO again
1001: you're becoming tedious

type here your chat msg

2017
Giovanni Maruzzelli - OpenTelecom.IT

```
a=fingerprint:sha-256
D9:D4:96:0D:E0:62:D5:4E:F4:E3:F3:BA:07:31:6E:DC:0E:85:74:
D5:7E:1F:E5:19:37:1E:B5:C2:99:09:A3:11
a=setup:active
a=rtcp-mux
a=rtcp:63228 IN IP4 93.50.134.11
a=rtcp-fb:96 ccm fir
a=rtcp-fb:96 nack
a=rtcp-fb:96 nack pli
a=ssrc:1996297669 cname:jnRdS5y4y0wYHG7E
a=ssrc:1996297669 msid:8I8M3FnvlDBBzRji7wEnkzIjNZoPhKof
v0
a=ssrc:1996297669
mslabel:8I8M3FnvlDBBzRji7wEnkzIjNZoPhKof
a=ssrc:1996297669
label:8I8M3FnvlDBBzRji7wEnkzIjNZoPhKofv0
a=ice-ufrag:Axl2tjkHvDkS03Rf
a=ice-pwd:zwEJpTP2q5KNPXtgkE9E0jb9
a=candidate:1218722370 1 udp 659136 93.50.134.11 63228
typ host generation 0
a=end-of-candidates
▶msg  ▶Object                                          high2.js:8
▶data  ▶Object                                         high2.js:9
▶data.pvtData  ▶Object                                 high2.js:13
Dialog 0e608ae3-f57c-912a-90e3-                   verto-min.js:266
346c137e448c: state change from trying to active
▶ACTIVE                                              high2.js:161
Using Speaker:  any                               verto-min.js:269
REMOTE STREAM ▶MediaStream                         verto-min.js:27
  <video id="webcam" autoplay="autoplay" class="embed-
  responsive-item" style="display: block;"></video>
▶msg  ▶Object                                          high2.js:8
▶data  ▶Object                                         high2.js:9
▶info                                                 high2.js:61
▶msg  ▶Object                                          high2.js:8
▶data  ▶Object                                         high2.js:9
▶info                                                 high2.js:61
▶msg  ▶Object {name: "info", val: 1}              verto-min.js:177
▶data                                             verto-min.js:177
  Object {from: "1000@lab.opentelecomsolutions.com", to:
▶ "1000@lab.opentelecomsolutions.com", body: "this is
  VERTO again", from_user: "1000", from_host:
  "lab.opentelecomsolutions.com"…}
▶info                                             verto-min.js:177
▶msg  ▶Object {name: "info", val: 1}              verto-min.js:177
▶data                                             verto-min.js:177
  Object {from: "1001", to:
▶ "1000@lab.opentelecomsolutions.com", body: "you're
  becoming tedious", to_proto: "sip", from_user: "1001"…}
▶info                                             verto-min.js:177
>
```

JavaScript

Both in **VERTO** and in **SIP**.js flavor, the WebRTC Rooms application **script is very similar**.
Actually, VERTO flavor was the one written first, as a companion to the "Mastering FreeSWITCH" Packt book published on occasion of ClueCon 2016. In that book you can read an almost line by line discussion of it. This new version has some fixes and added features (automatic answer to incoming calls, better management of DTMF sending during a call, possibility to call non numeric extensions, and various polish).

```
$(window).load(function() {
    cur_call = null;
    chatting_with = false;
    $("#conference").show();
    $("#ext").hide();
    $("#backbtn").hide();
    $("#cidname").hide();
    $("#callbtn").hide();
    $("#hupbtn").hide();
    $("#chatwin").hide();
    $("#chatmsg").hide();
    $("#chatsend").hide();
    $("#webcam").hide();
    $("#video1").hide();
    $("#login").keyup(function(event) {
        if (event.keyCode == 13 && !event.shiftKey) {
            $("#loginbtn").trigger("click");
        }
    });
    $("#ext").keyup(function(event) {
        if (event.keyCode == 13 && !event.shiftKey) {
            $("#callbtn").trigger("click");
        }
    });
});
```

At startup both flavor execute the "load" function, that sets which HTML elements are visible, and how to react when the user presses "Enter" while writing in the login or extension input area.

When the user fill the login input and presses Enter or click on the login button, the init() function is executed. Inside init() we create the main JavaScript object, our own WebRTC User Agent client. Let's look at the SIP.js flavor, VERTO is very much similar:

```
function init() {
    cur_call = null;
    chatting_with = false;

    var nameHost;
    var which_server;

    nameHost = window.location.hostname;

    which_server = "wss://" + nameHost + ":" + "3384";
    console.error("which_server=", which_server);
    ua = new SIP.UA({
        wsServers: which_server,
        uri: $("#login").val() + "@" + nameHost,
        password: $("#passwd").val(),
        userAgentString: 'SIP.js/0.7.7 Sara',
        traceSip: true,
    });

    $("#cidname").keyup(function(event) {
        if (event.keyCode == 13 && !event.shiftKey) {
            $("#callbtn").trigger("click");
        }
    });

    setupChat();

    $(document).keypress(function(event) {
        var key = String.fromCharCode(event.keyCode || event.charCode);
        var i = parseInt(key);
        var tag = event.target.tagName.toLowerCase();

        if (tag != 'input') {
            if (key === "#" || key === "*" || key === "0" || (i > 0 && i <= 9)) {
                cur_call.dtmf(key);
            }
        }
    });
}
```

We first made sure there are no calls and chat data structures left dangling, then establish which server we're connecting to (useful in case of massive scaling with domain based partitioning), then we use values we got from HTML to create our main object, in this case "ua".

We prepare the input field for the callee extension to react at Enter press.

Then we call the setupChat() function, which is slightly different in each flavor. This is because, in the VERTO flavor, we fill the "from" fields while in SIP.js those are automatically setup by the main object. The following is VERTO:

```
function setupChat() {
    $("#chatwin").html("");

    $("#chatsend").click(function() {
        if (!cur_call && chatting_with) {
            return;
        }
        cur_call.message({
            to: chatting_with,
            body: $("#chatmsg").val(),
            from_msg_name: cur_call.params.caller_id_name,
            from_msg_number: cur_call.params.caller_id_number
        });
        $("#chatmsg").val("");
    });

    $("#chatmsg").keyup(function(event) {
        if (event.keyCode == 13 && !event.shiftKey) {
            $("#chatsend").trigger("click");
        }
    });
}
```

We then have the management of many buttons and situations: hangup, back, call, and so on.

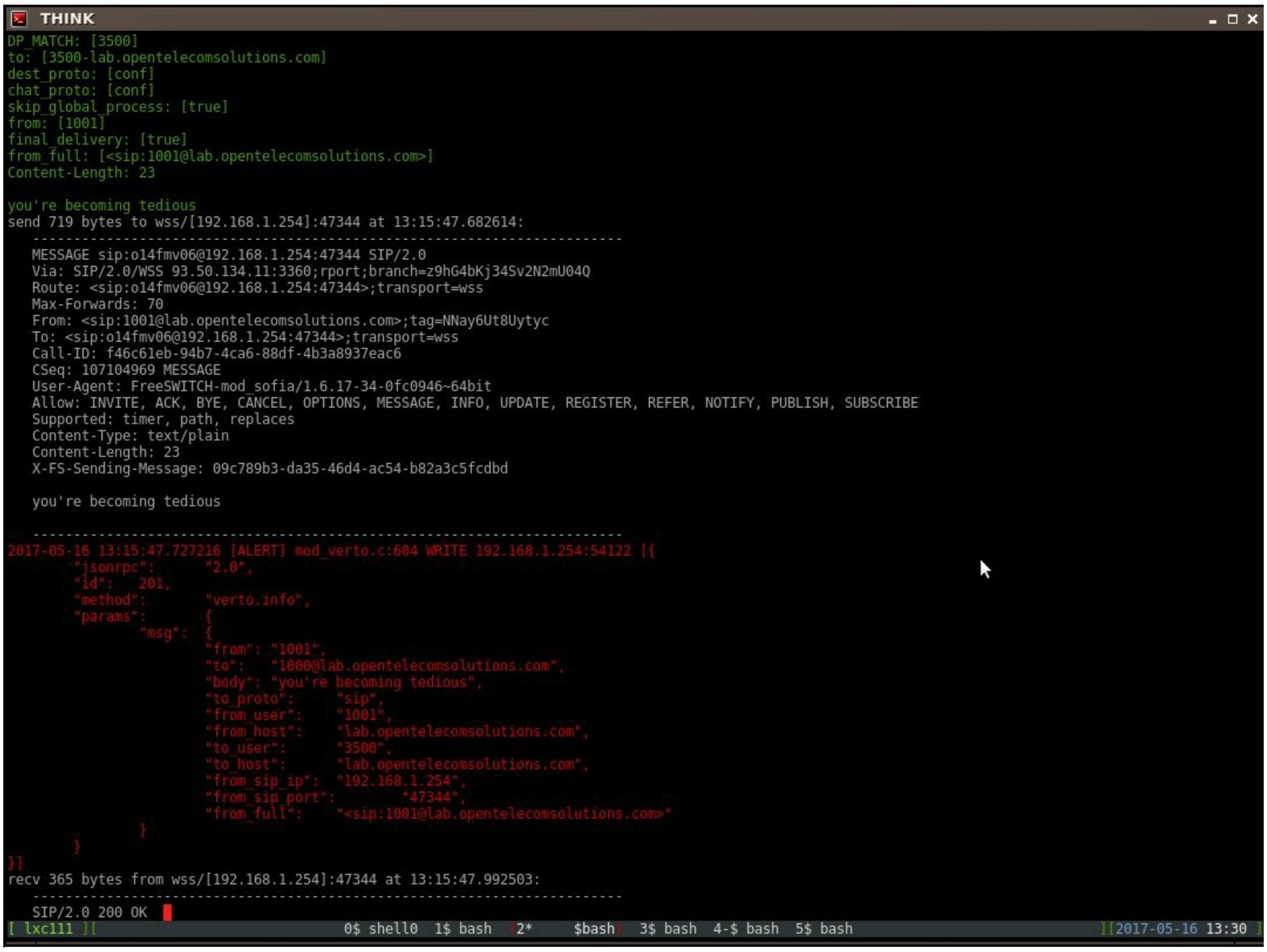
The "call" button execute the function docall(), again, very similar in both flavors: you use the INVITE method in SIP.js (already!) and the newCall method on VERTO. They get more or less similar arguments: in VERTO you have setup callbacks for events when the main object is initialized, while in SIP.js you setup your callbacks now.

Let's look at the VERTO flavor:

```
function docall() {
    if (cur_call) {
        return;
    }
    cur_call = verto.newCall({
        destination_number: $("#ext").val(),
        caller_id_name: $("#login").val(),
        caller_id_number: $("#login").val(),
        useVideo: true,
        useStereo: true,
        useCamera: 'any',
        useSpeak: 'any',
        useMic: 'any'
    });
}
```

The remaining parts of the scripts both setup their callbacks to react to incoming chat messages, to hangups, and to the establishment of the session (eg, remote peer answered).

Here's a screenshot of the FreeSWITCH console, showing chatplan info, SIP SIMPLE message, and Verto message. Yesss!



```
THINK
DP_MATCH: [3500]
to: [3500-lab.opentelecomsolutions.com]
dest_proto: [conf]
chat_proto: [conf]
skip_global_process: [true]
from: [1001]
final_delivery: [true]
from_full: [<sip:1001@lab.opentelecomsolutions.com>]
Content-Length: 23

you're becoming tedious
send 719 bytes to wss/[192.168.1.254]:47344 at 13:15:47.682614:
   ------------------------------------------------------------------------
   MESSAGE sip:o14fmv06@192.168.1.254:47344 SIP/2.0
   Via: SIP/2.0/WSS 93.50.134.11:3360;rport;branch=z9hG4bKj34Sv2N2mU04Q
   Route: <sip:o14fmv06@192.168.1.254:47344>;transport=wss
   Max-Forwards: 70
   From: <sip:1001@lab.opentelecomsolutions.com>;tag=NNay6Ut8Uytyc
   To: <sip:o14fmv06@192.168.1.254:47344>;transport=wss
   Call-ID: f46c61eb-94b7-4ca6-88df-4b3a8937eac6
   CSeq: 107104969 MESSAGE
   User-Agent: FreeSWITCH-mod_sofia/1.6.17-34-0fc0946~64bit
   Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, INFO, UPDATE, REGISTER, REFER, NOTIFY, PUBLISH, SUBSCRIBE
   Supported: timer, path, replaces
   Content-Type: text/plain
   Content-Length: 23
   X-FS-Sending-Message: 09c789b3-da35-46d4-ac54-b82a3c5fcdbd

   you're becoming tedious

   ------------------------------------------------------------------------
2017-05-16 13:15:47.727216 [ALERT] mod_verto.c:604 WRITE 192.168.1.254:54122 [{
        "jsonrpc":      "2.0",
        "id":   201,
        "method":       "verto.info",
        "params":       {
                "msg":  {
                        "from": "1001",
                        "to":   "1000@lab.opentelecomsolutions.com",
                        "body": "you're becoming tedious",
                        "to_proto":     "sip",
                        "from_user":    "1001",
                        "from_host":    "lab.opentelecomsolutions.com",
                        "to_user":      "3500",
                        "to_host":      "lab.opentelecomsolutions.com",
                        "from_sip_ip":  "192.168.1.254",
                        "from_sip_port":        "47344",
                        "from_full":    "<sip:1001@lab.opentelecomsolutions.com>"
                }
        }
}]
recv 365 bytes from wss/[192.168.1.254]:47344 at 13:15:47.992503:
   ------------------------------------------------------------------------
   SIP/2.0 200 OK
[ lxc111 ][                    0$ shell0  1$ bash  2*    $bash  3$ bash  4-$ bash  5$ bash                  ][2017-05-16 13:30 ]
```

Let's look at the different ways to react to an incoming call (both flavors answer automatically for you).
SIP.js registered the function handleInvite as callback for when a call is incoming.
Here's the implementation:

```
function handleInvite(s) {

    cur_call = s;

    s.accept({
        media: {
            constraints: {
                audio: true,
                video: true
            },
            render: {
                remote: document.getElementById('webcam')
            }
        }
    });
    $("#ext").val(s.remoteIdentity.uri.toString());
    cur_call.on('accepted', onAccepted.bind(cur_call));
    cur_call.once('bye', onTerminated.bind(cur_call));
    cur_call.once('failed', onTerminated.bind(cur_call));
    cur_call.once('cancel', onTerminated.bind(cur_call));
    $("#chatwin").html("");
}
```

It accepts the call, gives its own constraints to the media negotiation, sets some callback for call events (success, remote hangup, and so on), then sets the contents of a couple of HTML fields.

The VERTO flavor script also registered callbacks to the incoming events and state changes (similar to a GUI message pump programming style):

```
onDialogState: function(d) {
    if (!cur_call) {
        cur_call = d;
    }
    switch (d.state) {
        case $.verto.enum.state.ringing:
            console.error("RINGING");

            $("#webcam").show();
            $("#video1").show();
            cur_call.answer({
                callee_id_name: "ciao",
                callee_id_number: "1234567",
                useVideo: true,
                useStereo: true,
                useCamera: true,
                useMic: true
            });
            break;
```

TODO

A lot! For example, implementing direct chat (in normal calls, outside conferences) in VERTO.

In SIP.js, print in the chat textarea the message sent during "normal" calls (sending, receiving, and displaying of the received message is already working).

Polishing and homogenize it all

Adding button, menus, and commands to manage conferences layouts, moderator duties, screen sharing, and the like.

Have a more robust management of call failures, and a robust clean it up when things go awry.

Summary

In this chapter, we discussed the following:

- What is WebRTC and why it is very much relevant
- Mandatory security in WebRTC
- WebRTC implementation in FreeSWITCH
- The role of signaling protocols (SIP and Verto) in WebRTC
- Complete installation and configuration instruction for both FreeSWITCH and supporting WebRTC components (certificates, web servers)
- A super advanced and fully featured browser client for WebRTC videoconferencing and telepresence: Verto Communicator
- How to write a client web application using VERTO or SIP.js

In this chapter, we looked at both the theory and the nitty gritty details, and we learned how to setup real world WebRTC services for our user base.

In the next chapter we'll have a first look at the XML dialplan, the call routing engine at the core of FreeSWITCH. We'll look at how to implement the most diverse services and leverage the wealth of examples in the demo default configuration

XML Dialplan

The XML Dialplan is at the heart of a FreeSWITCH installation. For many old hands of telecommunication, coming from a different background, it can be confusing, and almost intimidating. Truth is: is very simple and logical. You just need to look at it with fresh eyes.

Let's dispel myths:

- XML is not difficult at all, it reads exactly like a plain text configuration file
- It does not need any kind of special editor: Notepad, Vim, Emacs, Nano, anything, will do
- It is not esoteric: it is text logically structured

In this chapter we will not even talk about XML in itself, because there is no need to.

Instead, we'll delve into the dialplan structure: depending on its characteristics, an incoming call will land to a specific "context" in dialplan. Contexts are like completely separated "jail" (in the BSD or chroot meaning of the word), they are the "virtual machines", or the "sandboxes" of dialplan. You can go multitenant using multiple contexts.

Each context contains extensions, where conditions are checked, and based on the result of the check, actions are executed or not.

We'll look at the multiple flow controls in dialplan, the "continue" attribute of extensions, and the "break" attribute of conditions.

We'll review what call "legs" are, and their funny terminology of A-leg and B-legs.

Then we'll look at how to interpret the demo configuration dialplan, and how to write new extensions, and all the building blocs (applications, dialstrings, etc) for a basic FreeSWITCH dialplan.

Forget all you know

Here's general dialplan concepts:

- **Dialplan** decides what FreeSWITCH will do with/to an **incoming** (audio/video) **call**
- Dialplan is a **list of unrelated patterns**
- Each individual pattern has its own individual **list of actions**
- The call **"enters"** the dialplan **from beginning**, and proceeds toward dialplan end
- The call **tries** to "match" **each pattern in turn**, from first to last
- If the call **"matches"** a pattern, then the list of **actions** of this pattern **is added** to the **call TODO list**
- **At the end** of dialplan **traversal**, the TODO **list is executed**
- **Matching (or non matching)** an individual pattern can be cause for **stopping** traversing the dialplan (that is, not try to match any successive pattern)
- The name **"EXTENSION"**, in FreeSWITCH parlance, **designatesanindividual PATTERN** complete with its own **set of ACTIONS**
- **Examples of individual patterns**: (call destination number is between 1000 and 1019 and time of call is between 10 and 11 in the morning), (call is incoming from network 192.168.1.0/24), (call is not coming from network 192.168.1.0/24), (call is from a specific DID), (call is coming from a Firefox Browser and is a WebRTC call and destination number begins with 852), (call is coming from an internal SIP phone)
- **Examples of action lists**: (answer, play a sound file to caller, hangup), (answer, play a sound to caller, monitor which DTMFs the caller presses, react to those DTMFs by doing a database query and playing different sounds based on query results, hangup), (answer, monitor for fax tone, if tone is present then receive fax, else transfer call to operator), (answer, join the call to a videoconference)

Contexts

Dialplan can have different sections, named **contexts**, which are **completely separated** from each other. Because of this separation, contexts can be used to implement a **multitenant** system (one only FreeSWITCH serving both companyA and companyB, without worries about extensions overlapping).

Based on **where** the call **come from** (that is, which protocol, network interface, or port) or based on an explicit **"context attribute" of the user** originating the call, the incoming call is "sent" to a specific dialplan context. At the **end of that context the call exits** the dialplan.

Contexts are so much separated one from each other that often **people talks** about **different dialplans**, while actually **they mean different contexts**. Formally **there is only one dialplan, the XML one**. For all practical purposes, you too can think of different contexts as different dialplans. That's OK, we all do.

There are **three contexts in the demo configuration** you get out of the box after a fresh FreeSWITCH installation (you can have as many you want):

- **default**: calls originated by internal trusted users are sent to this
- **public**: all other incoming calls are sent to this
- **features**: no calls are sent to it, but the other two contexts "borrow" features from it

Default context

The default context is the workhorse of the demo configuration: all calls from internal phones are sent to "default" context. Is the most complex context of the demo configuration, with an awful number of extensions.

Most of its extensions (see below for the exact meaning of the term "extension" in FreeSWITCH) are not "final destinations" for the call, instead they simply attach some variables to the call representation, based on the call matching or not matching certain criteria (patterns). Some extensions lead to services (IVRs, voicemail, conferences, etc). Only one of all the extensions in the default context is a bridge to internal phones, that is, what traditionally has been designated with the word "extension".

The XML dialplan's default context in demo configuration is in /usr/local/freeswitch/conf/dialplan/default.xml file, and includes all XML files inside the /usr/local/freeswitch/conf/dialplan/default/ directory.

Public context

In the demo configuration dialplan, "public" is the context to which all the calls coming from "outside", or from unauthorized users, are sent. It is a very simple context, containing just two extensions. Unauthorized users and calls coming from outside are totally limited in what they can do or which services can reach (actually zero).

The first extension print some debug info about the call on FreeSWITCH console, while the second extension is a transfer to the "internal phones" extension in the default context.

The XML dialplan's public context in demo configuration is in /usr/local/freeswitch/conf/dialplan/public.xml file, and includes all XML files inside the /usr/local/freeswitch/conf/dialplan/public/ directory.

DIDs are usually managed right there, by dialplan extensions with patterns that match calls incoming from ITSP provided phone numbers, in XML files inside the /usr/local/freeswitch/conf/dialplan/public/ directory.

Eg, in an XML file there we will define that a call coming from our ITSP with number +1212555444 will be bridged to Sara's registered internal deskphone, while +1212555445 will instead be connected to our IVR ("Hello, you reached FreeSWITCH Solutions, your call is important to us", and so on).

Features

Calls are not sent directly to the "features" context of the demo configuration. Instead, the extensions listed in features context are there to be "executed" by other contexts.

Let's say the features context is a repository for, uhm, features ready for consumption.

For example, extensions in demo features context are able to emulate "call transfer" feature even in phone that does not have a "call transfer" button.

The XML dialplan's features context in demo configuration is in /usr/local/freeswitch/conf/dialplan/features.xml file

Extensions

In FreeSWITCH parlance, an extension is NOT a phone or a service.
An extension is a configuration compound made of a pattern (criterion) and a list of actions.

An incoming call traverses the dialplan one extensions after another. The call has many characteristics (destination number, time of day, kind of transport, etc). At each extension those characteristics (represented by "variables") are checked against the pattern (the criterion). If the pattern "matches" the characteristics, the list of actions will be added to the call's TODO list. At the end of dialplan traversal, all the actions stacked into the TODO list will be executed, one after another.

An extension is contained between the XML tags <extension> and </extension>. Extensions can have two attributes: **name** and **continue**.

Attribute **"name"** has no usage beyond being printed in debug output. Helps you to follow through what happens. Forget about it, is just a name.

Attribute **"continue"** is of the utmost importance. It determines if the call, having matched the pattern of this extension and stacked its actions into the TODO list, will then continue to the next extension or will exit the dialplan now (and begin the execution of the TODO actions list).

By default, a call exits the dialplan at the first extension match.

If you set <extension name="myextensionname" continue="true">, the call will check if it matches the next extension ("and so on, and so on", as Zizek said).

Conditions (and "patterns")

Conditions are, broadly speaking, the building blocks for the "patterns" we talked about before in this chapter.

Conditions are contained by "**extension**". Conditions contain "**action**". Conditions have the following generic appearance:

```
<extension name="this_extension">
   <condition field="destination_number" expression=""^1234$">
         <!-- action -->
         <!-- action -->
   </condition>
</extension>
```

Inside a dialplan "**context**" (that is, it would be inside "default" or "public" context in demo configuration) we have a opening "extension" tag, then conditions begin. Between the opening and closing condition tag, we find actions.

In this case the condition will check if the "destination_number" call variable is matched by the "expression" (regular expression, or regex) "^1234$". If the number called was actually 1234, then the action(s) contained between this one condition opening and closing tags will be added to the TODO list (at the end of dialplan traversal all actions in the TODO list will be executed).

```
<extension name="that_extension">
   <condition field="ani" expression=""^5551212$"/>
   <condition field="destination_number" expression=""^1234$">
         <!-- action -->
         <!-- action -->
   </condition>
</extension>
```

In "that_extension" we actually have two conditions, one after the other (conditions are "stacked"). The first one has no "action" between the opening and closing tags (closing tag is implied by the trailing "/", in XML).

This means: first condition will check if the "ani" variable (eg, calling party phone number) is 5551212. If that first condition was evaluated to "true", then the second condition will be evaluated. If destination number is 1234, then this second condition's actions will be added to call's TODO list.

Stacked conditions are in AND relation, they must both be true, for the second condition's actions to be eventually executed. This because in a condition is implied by default the "break='on-false'" attribute.

```
<extension name="other_extension">
   <condition field="ani" expression=""^5551212$" break="on-true"/>
   <condition field="destination_number" expression=""^1234$">
         <!-- action -->
         <!-- action -->
   </condition>
</extension>
```

In "other_extension", we see the "**break**" attribute at work. Here the condition breaks the flow if the expression evaluate to true. So, in this case: if calling party number is different from 5551212, then second condition will be evaluated.

Following are the possible values for the **break** (meaning: it breaks out from checking further conditions - it breaks out of the extension) attribute:

- **on-false**: this is default, if expression evaluate false, it will not check next conditions
- **on-true**: if expression evaluate to true, it will not check next conditions
- **never**: will never break out, it will do check next conditions
- **always**: will sure break out, will not check next conditions

Call legs (channels)

Each industry has its own parlance and magic words. In telecommunication, whatever the underlying technology (SIP, WebRTC, TDM, etc), you will very often read about call "**legs**" and "**channels**".

First confusing fact: each "call leg" is actually a call in its own right. That is: a call is often said to be made by an "A-leg" and a "B-leg". In fact, "A-leg" and "B-leg" are proper calls.

Second confusing fact: each leg is a channel. So, most of the "calls" are consisting of two channels(A-leg and B-leg), while some (to IVRs, voicemail, and the like) are consisting of one only channel (A-leg).

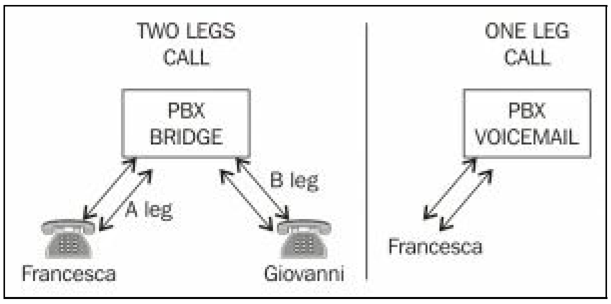
The reason for this funny terminology is the fact that when people talk about "a call", they usually mean an end-to-end voice or video connection, from the caller to the callee. Also, and maybe more importantly, calls were billed this way, as a complete "circuit" established between caller and callee.

Such a complete "call" circuit, when you have a server in the middle (for example, FreeSWITCH) is actually made by two completely separate and independent calls (also known as channels): one from the caller to FreeSWITCH, and one from FreeSWITCH to callee.

To better understand this, think at when you check your voicemail. The call "circuit" is between you and the voicemail server. This is the "A-leg" of the call (and in this case, there is no "B-leg").

From the server (that is, FreeSWITCH) point of view, the "A-leg" is the "incoming" call, the call originated by the caller.

After an incoming call is received, depending on the dialplan, a second call leg (in fact, another independent channel) will be originated by FreeSWITCH. Is also possible that dialplan will originate many B-legs (for example, when you want one person to be called in parallel to its cellphone, deskphone and home phone). This newly originated call(s) is(are) the **B-leg(s)**.



Consider a situation where an incoming call is arriving into FreeSWITCH. Inside FreeSWITCH this is allocated as one channel. This is the A-leg. Let's say destination_number of that call is 1010, it matches the "Local_Extension" extension in demo configuration dialplan.

FreeSWITCH originates another call(s) (allocated as channel(s)) from itself to each of the internal phone(s) registered as "1010". This is/are B-leg(s) calls/channels.

If one phone answers, then FreeSWITCH will join (eg mix, bridge) the media streams of the two channels/calls, and they (the caller and callee) can talk to each other. Now the ongoing complete "call", is actually composed by two calls/channels: A-leg (from caller to FreeSWITCH) and B-leg (from FreeSWITCH to callee).

If nobody answers, after a timeout the B-leg(s) (calls/channels originated by FreeSWITCH to 1010 phones) will die (canceled by FreeSWITCH).

The A-leg will go to voicemail. The complete "call" is now composed by only one channel (also known as call), the A-leg, from the caller to FreeSWITCH.

Channel Variables

Each individual channel (call) in FreeSWITCH has a number of associated characteristics and values, known as **"channel variables"**.

You use variables to get informations about the channel internals and to control the channel behavior.

Some of channel variables are assigned from the inception of the channel. Some of them change value during the life of the channel. Some variables are read-only. Some channel variables are writable, this means we can alter their value. You can (and often do) create variables in a channel, via dialplan or scripting. Specific channel variables modify the behavior of the channel, and you can take control of it by writing ("set") a variable value.

There are an incredible number of channels variables that are already set automatically when a call is processed.

For a first impact, connect via ssh to a FreeSWITCH server with demo configuration, execute /usr/local/freeswitch/bin/fs_cli, and then call 9192

```
<extension name="show_info">
  <condition field="destination_number" expression="^9192$">
    <action application="answer"/>
    <action application="info"/>
    <action application="sleep" data="250"/>
    <action application="hangup"/>
  </condition>
</extension>
```

This is a simple extension that checks destination_number, if it matches 9192 then it answers the call, execute the "info" application, then sleeps for a quarter of a second, and hangups.

As we'll see later in this chapter, the "info" application dumps on console every channel variable and its value. It's a precious debug tool!

This is an extract of what will be printed out in green characters on your terminal (actually this is a very small part. Try it for yourself!):

```
2017-05-25 17:34:15.625917 [INFO] mod_dptools.c:1743 CHANNEL_DATA:
Channel-State: [CS_EXECUTE]
Channel-Call-State: [ACTIVE]
Channel-State-Number: [4]
Channel-Name: [sofia/internal/1011@lab.opentelecomsolutions.com]
...
variable_direction: [inbound]
variable_uuid: [12cd1ed7-d04a-40a8-b1b5-495af077c69c]
variable_session_id: [2]
```

```
variable_sip_from_user: [1011]
variable_sip_from_uri: [1011@lab.opentelecomsolutions.com]
...
variable_accountcode: [1011]
variable_user_context: [default]
variable_effective_caller_id_name: [Extension 1011]
variable_effective_caller_id_number: [1011]
variable_outbound_caller_id_name: [FreeSWITCH]
variable_outbound_caller_id_number: [0000000000]
variable_callgroup: [techsupport]
variable_user_name: [1011]
variable_domain_name: [lab.opentelecomsolutions.com]
...
variable_endpoint_disposition: [ANSWER]
variable_current_application: [info]
```

All the output lines with "variable_" prefix can be accessed in dialplan using the ${} construct. For example, the last output line, "variable_current_application: [info]", means: the channel variable "${current_application}" has value "info". We can write a new dialplan extension to demonstrate it.

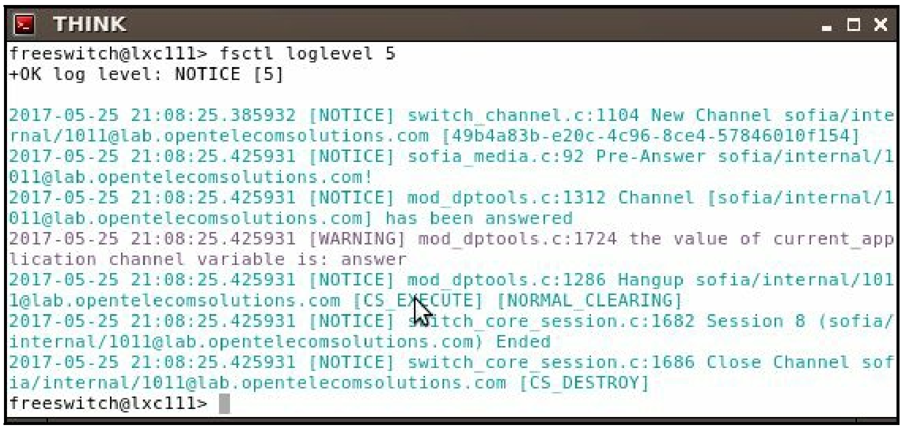
```
<extension name="giovanni_01">
   <condition field="destination_number" expression="^290864$">
         <action application="answer"/>
         <action application="log" data="WARNING the value of current_application channel var
         <action application="hangup"/>
   </condition>
</extension>
```

We insert this new extension at the beginning of the /usr/local/freeswitch/conf/dialplan/default.xml file, just after

```
<include>
  <context name="default">
```

Then from fs_cli, be sure to type: "fsctl loglevel 5", and then "reloadxml". Now, call 290864 (numerology: 29 August 1964 is my birthday).

Your terminal will show something similar to the following:



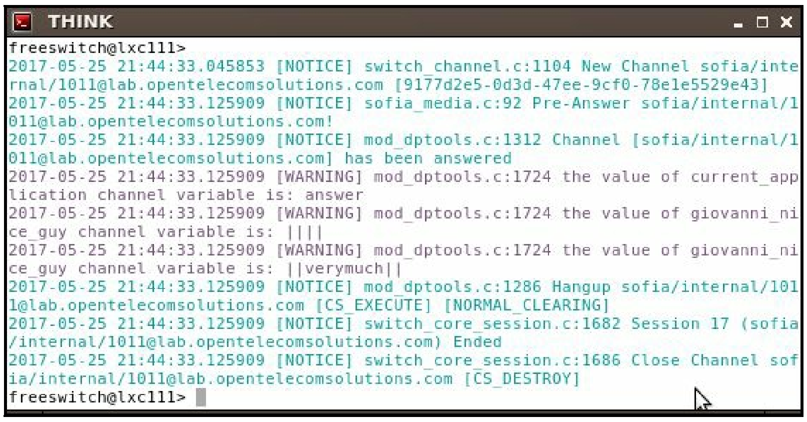
```
freeswitch@lxc111> fsctl loglevel 5
+OK log level: NOTICE [5]

2017-05-25 21:08:25.385932 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@lab.opentelecomsolutions.com [49b4a83b-e20c-4c96-8ce4-57846010f154]
2017-05-25 21:08:25.425931 [NOTICE] sofia_media.c:92 Pre-Answer sofia/internal/1
011@lab.opentelecomsolutions.com!
2017-05-25 21:08:25.425931 [NOTICE] mod_dptools.c:1312 Channel [sofia/internal/1
011@lab.opentelecomsolutions.com] has been answered
2017-05-25 21:08:25.425931 [WARNING] mod_dptools.c:1724 the value of current_app
lication channel variable is: answer
2017-05-25 21:08:25.425931 [NOTICE] mod_dptools.c:1286 Hangup sofia/internal/101
1@lab.opentelecomsolutions.com [CS_EXECUTE] [NORMAL_CLEARING]
2017-05-25 21:08:25.425931 [NOTICE] switch_core_session.c:1682 Session 8 (sofia/
internal/1011@lab.opentelecomsolutions.com) Ended
2017-05-25 21:08:25.425931 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@lab.opentelecomsolutions.com [CS_DESTROY]
freeswitch@lxc111>
```

Now, let's set a variable and read it back. Add three lines:

```
<extension name="giovanni_02">
```

```
    <condition field="destination_number" expression="^290864$">
          <action application="answer"/>
          <action application="log" data="WARNING the value of current_application channel var
          <action application="log" data="WARNING the value of giovanni_nice_guy channel varia
          <action application="set" data="giovanni_nice_guy=verymuch"/>
          <action application="log" data="WARNING the value of giovanni_nice_guy channel varia
          <action application="hangup"/>
    </condition>
</extension>
```

From the console, reloadxml, and then call 290864 again.



```
freeswitch@lxc111>
2017-05-25 21:44:33.045853 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@lab.opentelecomsolutions.com [9177d2e5-0d3d-47ee-9cf0-78e1e5529e43]
2017-05-25 21:44:33.125909 [NOTICE] sofia_media.c:92 Pre-Answer sofia/internal/1
011@lab.opentelecomsolutions.com!
2017-05-25 21:44:33.125909 [NOTICE] mod_dptools.c:1312 Channel [sofia/internal/1
011@lab.opentelecomsolutions.com] has been answered
2017-05-25 21:44:33.125909 [WARNING] mod_dptools.c:1724 the value of current_app
lication channel variable is: answer
2017-05-25 21:44:33.125909 [WARNING] mod_dptools.c:1724 the value of giovanni_ni
ce_guy channel variable is: ||||
2017-05-25 21:44:33.125909 [WARNING] mod_dptools.c:1724 the value of giovanni_ni
ce_guy channel variable is: ||verymuch||
2017-05-25 21:44:33.125909 [NOTICE] mod_dptools.c:1286 Hangup sofia/internal/101
1@lab.opentelecomsolutions.com [CS_EXECUTE] [NORMAL_CLEARING]
2017-05-25 21:44:33.125909 [NOTICE] switch_core_session.c:1682 Session 17 (sofia
/internal/1011@lab.opentelecomsolutions.com) Ended
2017-05-25 21:44:33.125909 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@lab.opentelecomsolutions.com [CS_DESTROY]
freeswitch@lxc111>
```

As we can see from the output, the **variable was not existing before** we assigned ("set") her.

For more in deep coverage of channel variables, their characteristics, their usage to control FreeSWITCH, and much more, see the Chapter 9, *Dialplan in Deep*.

Regular Expressions

Regular expressions (regex, regexes) are at the heart of dialplan, and used in many other parts of FreeSWITCH configuration.

They are an uber-clever way to analyze, slice, dice, and massage text strings. We in FreeSWITCH we use the best and brightest, the greatest regular expressions of them all: Perl Compatible Regular Expressions (PCRE). Isn't that beautiful? It is.

A regular expression checks if a string matches a pattern (the pattern being the regex). A regular expression can also substitute parts of the strings with something else. Also, regular expressions can "select" a part of a string, and reuse it in returning the result.

In dialplan regexes are used to define the "expression" criterion of the "condition" tag.

The construct most used in FreeSWITCH demo dialplan is as follows:

```
<extension name="giovanni_03">
   <condition field="destination_number" expression=""^(1234)$">
         <action application="log" data="WARNING this is $1"/>
   </condition>
</extension>
```

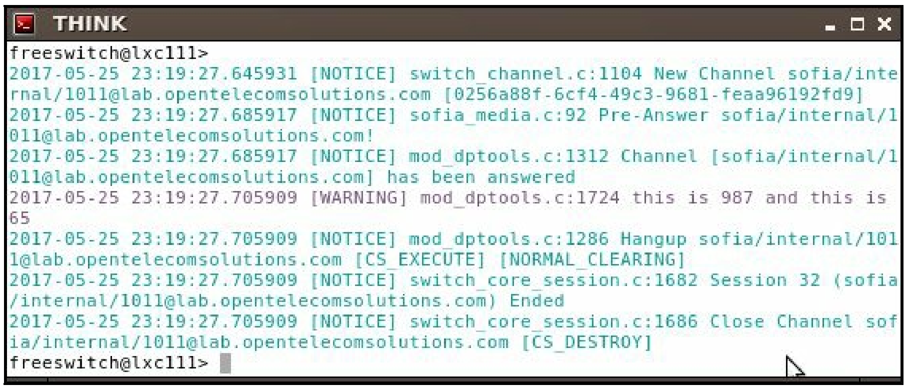
If we insert this extension at begin of default.xml demo dialplan, reloadxml, and then call 1234, we have a condition that matches if the destination_number channel variable string value is exactly "1234" (will not match a longer string). This regex has caret (^) as first character, meaning "beginning of string", dollar sign ($) as last character, meaning end of string, and some characters in between, enclosed by round parenthesis. Round parenthesis grab what's inside them, and put it in a "register". In this case, they put 1234 in register $1 (if there were a successive couple of parenthesis, they would put the grabbed text into the $2 register. And so on, and so on.

Let's look at a more complex use of regular expression:

```
<extension name="giovanni_04">
   <condition field="destination_number" expression="^1800(\d{3})(\d{2}).*$">
            <action application="answer"/>
         <action application="log" data="WARNING this is $1 and this is $2"/>
         <action application="hangup"/>
   </condition>
</extension>
```

Insert the extension at beginning of default.xml dialplan, then reloadxml and call

18009876543.
Your output will be similar to the following:



```
freeswitch@lxc111>
2017-05-25 23:19:27.645931 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@lab.opentelecomsolutions.com [0256a88f-6cf4-49c3-9681-feaa96192fd9]
2017-05-25 23:19:27.685917 [NOTICE] sofia_media.c:92 Pre-Answer sofia/internal/1
011@lab.opentelecomsolutions.com!
2017-05-25 23:19:27.685917 [NOTICE] mod_dptools.c:1312 Channel [sofia/internal/1
011@lab.opentelecomsolutions.com] has been answered
2017-05-25 23:19:27.705909 [WARNING] mod_dptools.c:1724 this is 987 and this is
65
2017-05-25 23:19:27.705909 [NOTICE] mod_dptools.c:1286 Hangup sofia/internal/101
1@lab.opentelecomsolutions.com [CS_EXECUTE] [NORMAL_CLEARING]
2017-05-25 23:19:27.705909 [NOTICE] switch_core_session.c:1682 Session 32 (sofia
/internal/1011@lab.opentelecomsolutions.com) Ended
2017-05-25 23:19:27.705909 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@lab.opentelecomsolutions.com [CS_DESTROY]
freeswitch@lxc111>
```

The expression we used in condition, "^1800(\d{3})(\d{2}).*$", can be read as: this condition matches if the field (destination_number) string value begins with 1800 then three digits then two digits then whatever until end of string. The first three digits after 1800 will be put into $1 register (first parenthesis pair), the following two digits into the $2 register.

The content of the registers persist only inside the present condition opening and closing tags.

The following are some sample regular expressions and their meanings:

Pattern	Meaning
123	Match any string containing the sequence "123"
^123	Match any string beginning with the sequence "123"
123$	Match any string ending with the sequence "123"
^123$	Match any string that is exactly the sequence "123"
\d	Match any single digit (0-9)
\d\d	Match two consecutive digits

^\d\d\d$	Match any string that is exactly three digits long
^\d{7}$	Match any string that is exactly seven digits long
^(\d{7})$	Match any string that is exactly seven digits long, and store the matched value in a special variable named $1
^1?(\d{10})$	Match any string that optionally begins with the digit "1" and contains an additional ten digits; store the ten digits in $1
^(3\d\d\d)$	Match any four-digit string that begins with the digit "3", and store the matched value in $1

We can test regular expressions from the FreeSWITCH command line, to be sure they do what we want.

The `regex` FreeSWITCH console command needs at least two arguments: the data to test and the pattern to match against. The arguments are separated by a `|` (pipe) character. The `regex` command will return `true` if the data and the pattern match, otherwise it will return `false`. You can try the following examples at `fs_cli`:

```
freeswitch@internal> regex 1234|\d

true

freeswitch@internal> regex 1234|\d\d\d\d

true

freeswitch@internal> regex 1234|\d{4}

true

freeswitch@internal> regex 1234|\d{5}

false
```

freeswitch@internal> regex 1234|^1234$

true

freeswitch@internal> regex 1234|234

true

freeswitch@internal> regex 1234|^234

false

The `regex` command also has a capture syntax that will store and return the registers' values. The expression `%0` contains the entire matched value whereas `%1` contains the first register, `%2` contains the second, and so forth.

```
freeswitch@internal> regex 18005551212|1?(\d\d\d)(\d\d\d)(\d\d\d\d)
true
freeswitch@internal> regex 18005551212|1?(\d\d\d)(\d\d\d)(\d\d\d\d)|%0
18005551212
freeswitch@internal> regex 18005551212|1?(\d\d\d)(\d\d\d)(\d\d\d\d)|%1
800
freeswitch@internal> regex 18005551212|1?(\d\d\d)(\d\d\d)(\d\d\d\d)|%2
555
freeswitch@internal> regex 18005551212|1?(\d\d\d)(\d\d\d)(\d\d\d\d)|%3
1212
freeswitch@internal> regex 18005551212|1?(\d\d\d)(\d\d\d)(\d\d\d\d)|%1%2%3
8005551212
freeswitch@internal> regex 18005551212|1?(\d\d\d)(\d\d\d)(\d\d\d\d)|%1-%2-%3
800-555-1212
freeswitch@internal> regex 18005551212|^7|%0
18005551212
```

For more regex documentation, Internet is full of tutorials on how to use Perl Regular Expressions (regexes), and there is a beautifully crafted page in Confluence especially for use in FreeSWITCH. Search http://freeswitch.org/confluence for "regular expression".

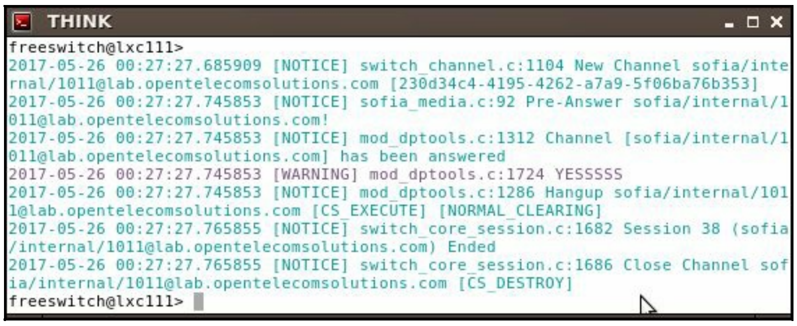
Actions and anti-actions

Actions and anti-actions live inside conditions. Actions are executed if the condition expression matches. Anti-actions are executed if the condition don't match.

Let's look at an "hello world" example, insert the following at the beginning of default.xml dialplan:

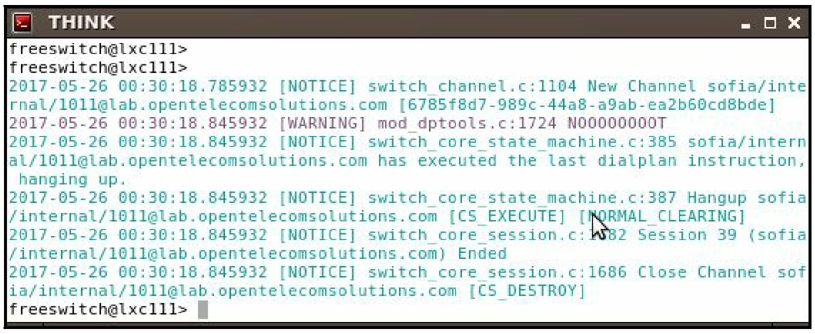
```
<extension name="giovanni_05">
   <condition field="destination_number" expression="^4321$">
            <action application="answer"/>
         <action application="log" data="WARNING YESSSSS"/>
         <anti-action application="log" data="WARNING NOOOOOOOOT"/>
         <action application="hangup"/>
   </condition>
</extension>
```

Then, as usual, reloadxml, and call 4321. Your terminal will show the following:



```
freeswitch@lxc111>
2017-05-26 00:27:27.685909 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@lab.opentelecomsolutions.com [230d34c4-4195-4262-a7a9-5f06ba76b353]
2017-05-26 00:27:27.745853 [NOTICE] sofia_media.c:92 Pre-Answer sofia/internal/1
011@lab.opentelecomsolutions.com!
2017-05-26 00:27:27.745853 [NOTICE] mod_dptools.c:1312 Channel [sofia/internal/1
011@lab.opentelecomsolutions.com] has been answered
2017-05-26 00:27:27.745853 [WARNING] mod_dptools.c:1724 YESSSSS
2017-05-26 00:27:27.745853 [NOTICE] mod_dptools.c:1286 Hangup sofia/internal/101
1@lab.opentelecomsolutions.com [CS_EXECUTE] [NORMAL_CLEARING]
2017-05-26 00:27:27.765855 [NOTICE] switch_core_session.c:1682 Session 38 (sofia
/internal/1011@lab.opentelecomsolutions.com) Ended
2017-05-26 00:27:27.765855 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@lab.opentelecomsolutions.com [CS_DESTROY]
freeswitch@lxc111>
```

Now, call instead 5555 (or whatever):



```
freeswitch@lxc111>
freeswitch@lxc111>
2017-05-26 00:30:18.785932 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@lab.opentelecomsolutions.com [6785f8d7-989c-44a8-a9ab-ea2b60cd8bde]
2017-05-26 00:30:18.845932 [WARNING] mod_dptools.c:1724 NOOOOOOOT
2017-05-26 00:30:18.845932 [NOTICE] switch_core_state_machine.c:385 sofia/intern
al/1011@lab.opentelecomsolutions.com has executed the last dialplan instruction,
 hanging up.
2017-05-26 00:30:18.845932 [NOTICE] switch_core_state_machine.c:387 Hangup sofia
/internal/1011@lab.opentelecomsolutions.com [CS_EXECUTE] [NORMAL_CLEARING]
2017-05-26 00:30:18.845932 [NOTICE] switch_core_session.c:1682 Session 39 (sofia
/internal/1011@lab.opentelecomsolutions.com) Ended
2017-05-26 00:30:18.845932 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@lab.opentelecomsolutions.com [CS_DESTROY]
freeswitch@lxc111>
```

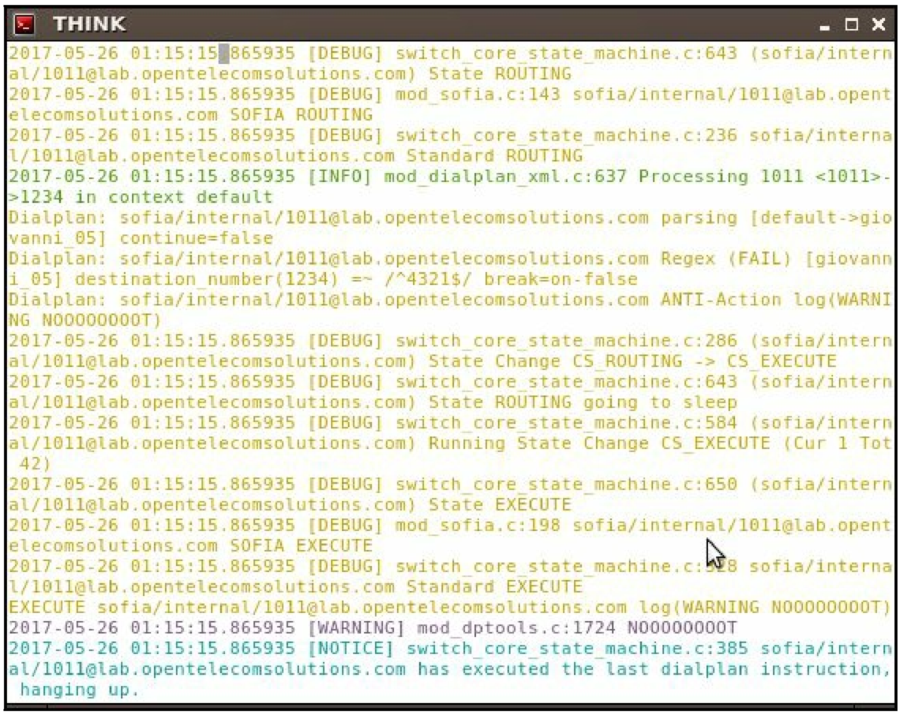
As you can see, this time only the anti-action has been executed. There is no "answer", and no explicit hangup (the hangup there is because when there are no more (anti-)actions in the call's TODO list, then... hangup. What else?).

How dialplan processing works

A call is incoming to FreeSWITCH.
It lands in a specific dialplan context based on its technology, network characteristics (profiles), or (if the call is originated by a registered user) based on explicit assignment in the User Directory. Starting from the beginning of the context, the call traverse the context, trying to match each extension, one after another.
Inside an extension, if a condition matches, actions and anti-actions contained in that condition are added to the call's TODO list. Based on the value of the "break" condition parameter, at each condition the call decides to stop, or to proceed to the next condition.
Based on the value of the "continue" extension parameter, at each extension the call decides to stop, or to proceed to the next extension.
After the call finishes traversing the dialplan context (because of "continue" or because there are no more extensions to check) then all (anti-)actions stored in its TODO list will be executed.

So, there are two separate phases: first phase is Dialplan Parsing (also known as ROUTING phase), where the call traverses the dialplan and accumulates actions in the TODO list (without executing them). The second phase is the EXECUTE phase, in which all accumulated actions are batch-executed.

This can be clearly seen from the FreeSWITCH console if we pump up the loglevel, type "fsctl loglevel 7", and then call 1234 again:



```
2017-05-26 01:15:15.865935 [DEBUG] switch_core_state_machine.c:643 (sofia/intern
al/1011@lab.opentelecomsolutions.com) State ROUTING
2017-05-26 01:15:15.865935 [DEBUG] mod_sofia.c:143 sofia/internal/1011@lab.opent
elecomsolutions.com SOFIA ROUTING
2017-05-26 01:15:15.865935 [DEBUG] switch_core_state_machine.c:236 sofia/interna
l/1011@lab.opentelecomsolutions.com Standard ROUTING
2017-05-26 01:15:15.865935 [INFO] mod_dialplan_xml.c:637 Processing 1011 <1011>-
>1234 in context default
Dialplan: sofia/internal/1011@lab.opentelecomsolutions.com parsing [default->gio
vanni_05] continue=false
Dialplan: sofia/internal/1011@lab.opentelecomsolutions.com Regex (FAIL) [giovann
i_05] destination_number(1234) =~ /^4321$/ break=on-false
Dialplan: sofia/internal/1011@lab.opentelecomsolutions.com ANTI-Action log(WARNI
NG NOOOOOOOOT)
2017-05-26 01:15:15.865935 [DEBUG] switch_core_state_machine.c:286 (sofia/intern
al/1011@lab.opentelecomsolutions.com) State Change CS_ROUTING -> CS_EXECUTE
2017-05-26 01:15:15.865935 [DEBUG] switch_core_state_machine.c:643 (sofia/intern
al/1011@lab.opentelecomsolutions.com) State ROUTING going to sleep
2017-05-26 01:15:15.865935 [DEBUG] switch_core_state_machine.c:584 (sofia/intern
al/1011@lab.opentelecomsolutions.com) Running State Change CS_EXECUTE (Cur 1 Tot
 42)
2017-05-26 01:15:15.865935 [DEBUG] switch_core_state_machine.c:650 (sofia/intern
al/1011@lab.opentelecomsolutions.com) State EXECUTE
2017-05-26 01:15:15.865935 [DEBUG] mod_sofia.c:198 sofia/internal/1011@lab.opent
elecomsolutions.com SOFIA EXECUTE
2017-05-26 01:15:15.865935 [DEBUG] switch_core_state_machine.c:[illegible]8 sofia/interna
l/1011@lab.opentelecomsolutions.com Standard EXECUTE
EXECUTE sofia/internal/1011@lab.opentelecomsolutions.com log(WARNING NOOOOOOOOT)
2017-05-26 01:15:15.865935 [WARNING] mod_dptools.c:1724 NOOOOOOOOT
2017-05-26 01:15:15.865935 [NOTICE] switch_core_state_machine.c:385 sofia/intern
al/1011@lab.opentelecomsolutions.com has executed the last dialplan instruction,
 hanging up.
```

As you can see, the call enters the ROUTING phase, where it checks the extensions of a specific dialplan context (in this case the "default" context) looking for matches. Here the first extensions has a condition with an expression that fails to match. So, at first thought we would expect to proceed to check the next extension. But this failed condition has an anti-action inside, so actually we have a match! A negative match still is a match, not only in romantic affairs.

So, after adding the anti-action to the TODO list, the call stops traversing the dialplan context, and go forward to the EXECUTE phase.

If we delete (or comment out) the anti-action from the condition, and reloadxml, when we call again we'll see that all the extensions in the context will be checked.

NB: It is important to remember that if you SETa channel variable in dialplan, and THEN CHECK, it WILL NOT WORK as expected!

```
<extension name="giovanni_06">
   <condition field="destination_number" expression="^4321$">
            <action application="answer"/>
            <action application="set" data="myvariable=abc"/>
      </condition>
      <condition field="${myvariable}" expression="abc">
         <action application="log" data="WARNING YESSSSS"/>
         <anti-action application="log" data="WARNING NOOOOOOOOT"/>
   </condition>
</extension>
```

This is because the "action" of setting the variable will be executed in the EXECUTE phase, while the checking of the variable is done at the previous ROUTING (parsing) phase.

To make it work, you must "inline" the set action, so it will be executed at ROUTING phase. The following extension works:

```
<extension name="giovanni_07">
   <condition field="destination_number" expression="^4321$">
            <action application="answer"/>
            <action application="set" data="myvariable=abc" inline="true"/>
      </condition>
      <condition field="${myvariable}" expression="abc">
         <action application="log" data="WARNING YESSSSS"/>
         <anti-action application="log" data="WARNING NOOOOOOOOT"/>
   </condition>
</extension>
```

Also, the following will never work (you cannot inline the result from play_and_get_digits. Use a script instead):

<extension name="play_and_get_digits with say">

```
<condition field="destination_number" expression="^(6500)$">

<action application="play_and_get_digits" data="1 1 1 3000 # say:'press one for technical support' silence_stream://250 choice \d+" />

</condition>

<condition field="${choice}" expression="1">

<action application="log" data="TECH SUPPORT!"/>

<anti-action application="log" data="OPERATOR"/>

</condition>

</extension>
```

Important dialplan applications

FreeSWITCH has many (hundreds) applications available for dialplan and scripting usage. Most of them are provided by **mod_dp_tools**, but almost every FreeSWITCH modules adds some applications to your toolbox.

Let's review some of the most popular building blocks, the ones you can start to experiment with immediately.

answer

The `answer` application picks up the phone by establishing a media stream to the calling party. In SIP terms, this causes FreeSWITCH to send a "200 OK" message, negotiate the audio/video codecs to use (if not already determined), and begin the flow of RTP packets.

Example:

```
<action application="answer"/>
```

bridge

The `bridge` application originate new call(s) as B-leg(s), and connects (mixes, bridges) the media streams of the newly originated B-leg(s) to the streams of the incoming call (also known as A-leg)

Argument syntax:

```
bridge data="<dialstring>[,<dialstring>][|<dialstring>]"
```

Dialstrings separated by **commas** are executed **simultaneously**. Dialstrings separated by **pipes** are executed **sequentially**. The **first** phone **to answer receives** the call, at which time dialing to **all other** phones is **canceled**.

Examples:

```
<action application="bridge" data="user/1000"/>
<action application="bridge" data="sofia/gateway/my_gateway_name/$1"/>
```

See the *Dialstring formats* section later in this chapter.

hangup

The `hangup` application disconnects the media streams and ends the call.

Argument syntax: Optional hang up cause.

Examples:

```
<action application="hangup"/>
<action application="hangup" data="USER_BUSY"/>
```

ivr

The `ivr` application sends the caller to a predefined IVR.

Argument syntax: Name of IVR to execute.

Example:

```
<action application="ivr" data="ivr_demo"/>
```

play_and_get_digits

The `play_and_get_digits` application will play a sound file to the caller, while at the same time listening for DTMFs dialed by the caller, and act on them.

This is especially useful when scripting FreeSWITCH, allowing for full application logic and control flow.

In dialplan, you can use it for simple interactions without invoking the ivr application and its XML IVR language.

Argument syntax:

```
<min> <max> <tries> <timeout> <terminators> <file> <invalid_file> <var_name> <regex> [<digit_
```

Arguments:

- `min`: Minimum number of digits to collect
- `max`: Maximum number of digits to collect
- `tries`: Number of attempts to play the file and collect digits
- `timeout`: Number of milliseconds to wait before assuming the caller is done entering digits
- `terminators`: Digits used to end input if less than `min` digits have been pressed (typically #)
- `file`: Sound file to play while digits are fetched
- `invalid_file`: Sound file to play when digits don't match the `regex` argument
- `var_name`: Channel variable that digits should be placed in
- `regex`: Regular expression to match valid digits
- `digit_timeout` (optional): Number of milliseconds to wait in between digits (defaults to timeout value)
- `failure_ext` (optional): Destination extension to which the call should be transferred if the caller does not enter a valid input value
- `* failure_dp` (optional with `failure_ext`): Destination Dialplan type when using `failure_ext (almost always "XML")`
- `failure_context` (optional with `failure_dp`): Destination Dialplan context when using `failure_dp`

Example:

```
<action application="play_and_get_digits" data="2 5 3 8000 # /path/to/sound_file.wav /path/tc
<action application="log" data="User entered these digits: ${my_digits}"/>
```

This example executes `play_and_get_digits` with the following parameters:

- Looks for a minimum of two digits
- Looks for a maximum of five digits
- Tries three times
- Waits 8 seconds before assuming the caller is done entering digits
- Uses the # key as the terminator
- Plays the sound file `/path/to/sound_file.wav` while collecting digits
- Plays the sound file `/path/to/invalid_sound.wav` if invalid digits are dialed
- Stores the dialed digits in the channel variable `my_digits`
- Matches against the pattern `\d+`

playback

The `playback` application plays an audio file to the caller. Files can be in many formats as supported by FreeSWITCH modules. The sound and music files included in FreeSWITCH demo are all `.wav` files.

Argument syntax: absolute path to a sound file or relative path to the default sound directory.

Examples:

```
<action application="playback"
  data="/absolute/path/to/sound.wav"/>
<action application="playback" data="voicemail/vm-goodbye.wav"/>
```

set

The `set` application sets a channel variable

Argument syntax:

```
<variable_name=value>
```

Example:

```
<action application="set" data="my_chan_var=example_value"/>
<action application="log" data="INFO my_chan_var contains ${my_chan_var}"/>
```

sleep

The `sleep` application pauses dialplan execution for the specified number of milliseconds.

Argument syntax: Number of milliseconds to sleep.

Example:

```
<action application="sleep" data="1000"/>
```

transfer

The `transfer` application sends the call back traversing the dialplan. This causes an entirely new parse (ROUTING) phase and EXECUTION phase to take place.

Argument syntax:

```
<destination number> [destination dialplan [destination context]]
```

Example:

```
<action application="transfer" data="9664"/>
<action application="transfer" data="12345 XML custom"/>
```

Dialstring formats

Dialstrings serve as arguments when originating (eg, creating new, outbound from the FreeSWITCH server) calls.

Each endpoint module has its own dialstring syntax. The most important endpoint modules are mod_sofia (supporting SIP signaling protocol) and mod_verto (supporting VERTO protocol).

Call creation (origination) is made in dialplan by the application "bridge". If you are in dialplan you are processing an incoming call, and the newly originated call(s) will be "bridged" to the incoming.

All registered users can get their dialstring from User Directory (that's what happen in demo configuration). In User Directory you can set a generic dialstring for all users, and specially crafted dialstrings for a users' group and/or for an individual user. Each value can be overridden the more specific you get.

Also, you can originate calls from command line (or from scripting), with the "originate" command (see later in this section).

SIP (sofia)

The basic Sofia dialstring takes two different formats, depending on wheter we use a gateway or not:

- sofia/<profile name>/<user@domain>
- sofia/gateway/<gateway name>/<user>

As we learned in Chapter 4, *User Directory, SIP, and Verto*, we can dial another SIP Endpoint, either with or without a gateway. When using Sofia to dial through a FreeSWITCH SIP profile, it is necessary to specify both the user and domain. However, when dialing through a gateway it is not possible to include the domain because this is already defined in the gateway configuration. Therefore, the following is not allowed:

```
<!-- Wrong -->
<action application="bridge" data="sofia/gateway/my_gateway/user@1.2.3.4"/>
```

The correct syntax is as follows:

```
<!-- Correct -->
<action application="bridge" data="sofia/gateway/my_gateway/user">
```

The equivalent for dialing through the `internal` profile would look like the following:

```
<!--Also correct -->
<action application="bridge" data="sofia/internal/user@1.2.3.4 /">
```

When dialing a user who is registered on your FreeSWITCH server there is a shortcut available:

```
user/<user id>[@domain]
```

This syntax makes it very easy to dial another phone registered on your system. In fact, `Local_Extension` in `conf/dialplan/default.xml` uses this method to connect calls to registered users:

```
<action application="bridge" data="user/${dialed_extension}@{domain_name}"/>
```

The `@domain` parameter is optional if you have only one domain defined on your FreeSWITCH server.

VERTO

Verto is much simpler than SIP, it connects users directly registered to the same FreeSWITCH server. So, from FreeSWITCH dialplan point of view, is very similar to the "user" SIP shortcut, and you don't have to bother too much:

```
<action
application="bridge"
data="${verto_contact ${user id}@${domain}}"/>
```

It actually uses the verto_contact API command to find the underlying RTC data needed to originate the call to the VERTO registered user.

Other dialstring formats

The following are a few more types of dialstrings:

- `loopback/<destination number>`: This creates a call leg and puts it in the dialplan at `<destination_number>`
- `freetdm/<span>/<channel>/<phone number>`: This creates a call leg on a telephony interface card (see http://wiki.freeswitch.org/wiki/FreeTDM for more information on using traditional telephony hardware with FreeSWITCH)
- `group/<group name>[@domain]`: This calls a group of users defined in User Directory

Originate Command

The basic syntax of `originate` is as follows:

```
originate <dialstring> <destination number>
```

It gets exactly the same dialtrings as the bridge application in dialplan. Actually, as "**destination number**" argument it accepts dialplan applications, so you can have this kind of command line:

```
originate sofia/internal/userA@firstdomain.com &bridge(sofia/internal/user@4.3.2.1)
```

This will originate a call to userA, and if userA answers, will originate an additional call to userB. If userB answers, it will bridge the two calls and the two users will be able to talk.

Summary

In this chapter, we discussed the following:

- The basic structure of XML dialplan:
- dialplan is divided into contexts
- contexts contains extensions
- extensions contains one or more conditions
- conditions decides about the execution of contained actions and anti-actions

We then reviewed the meaning of "call legs" (A-leg is the incoming call, B-legs are the outbound calls originated by FreeSWITCH in response to A-leg dialplan processing).

We also looked at channel variables, how to set and check them, followed by an explanation of how dialplan is traversed by accumulating "actions" in a TODO list that's executed after dialplan traversal.

And the basic building blocks of useful extensions: applications and dialstrings.

In next chapter we'll see how to use XML to build much more powerful IVRs than would be possible from dialplan.

Phrase Macros and XML IVRs

Interacting with the caller without unnecessary personnel involved is an age-old telecommunications industry specialty.

Automated Attendants and Interactive Voice Response systems are two of the most important and popular services almost any organization requires for its communication platform.

Be it a simple call dispatcher, as in an Automated Attendant that routes a call, "press 1 for sales, press 2 for support", or a complex IVR that gathers proof of caller identity and then queries a database for their bank balance status, FreeSWITCH can serve it.

But what about the messages that will be read to the caller? Are we going to record every possible phrase the customer will hear?

Nope, FreeSWITCH has the "phrase" construct, a flexible mechanism to compose arbitrary phrases from short sound files, which are combined in real time, dynamically, as needed. And then, yes, there is text-to-speech (TTS) too!

Phrase Macros and Voice Prompts

Phrase macros are FreeSWITCH's smart way to reuse, concatenate, and combine sound files into voice prompts to be played in calls.

Voice prompts are what you hear when you call into any kind of IVR, digital assistant, and so on. The most infamous of them is, "All our operators are busy at the moment, please hold on."

Voice Prompts history

Voice prompts used to be of two kinds: the very costly kind, recorded by professional actors, using high-end audio gear in professional studios, cut and polished by sound engineers in post-production. The other kind was the free-as-in-beer, do-it-yourself, very similar to the answering machine message recorded by the company secretary, unprofessional, with background noises, wavering volume, and so on.

There was nothing between those two extremes. Also, a lot of work went into recording and re-recording new prompts, reading text aloud. Such effort (and cost) made totally impractical for voice applications to read an often updated text to the caller.

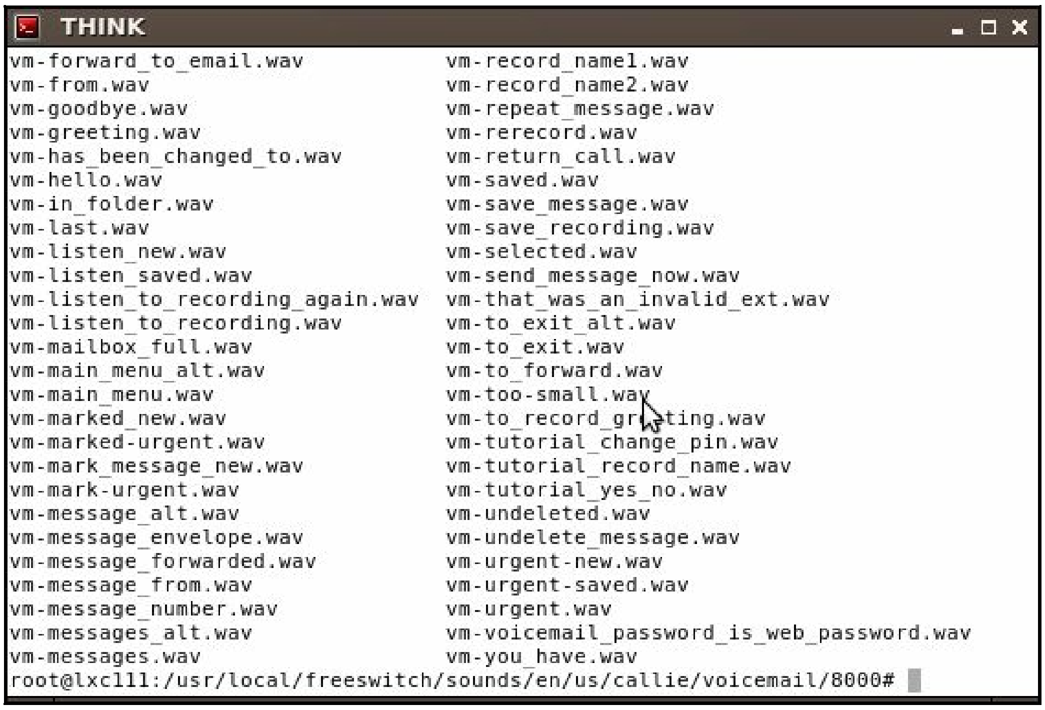
Then a few text-to-speech applications appeared that sounded mechanical and fun, like a robot in a Sixties B-movie, but for some kind of usage they were perfectly adequate. People was quick to get used to "robot" voices. Alas, it was still a very costly option, licensed per-concurrent channel, its implementation was non-trivial, and so on.

Sound files in FreeSWITCH

One of the most popular features of FreeSWITCH is its ability to build almost any kind of vocal interaction with callers, starting from a finite (and not so big) number of sound files. Those files are located under the directory /usr/local/freeswitch/sounds (audio files of music are there too, under the "music" subdirectory).

Those sound files are hyper-professionally recorded in super audio studios by top voice artists, and we even had a complete new set beautifully interpreted by the Goddess of Telecom Voices, Allison Smith, presented at ClueCon 2017 (ClueCon is the annual "from developers to developers" conference about VoIP, WebRTC, FreeSWITCH, and all real-time communication and IoT, held in Chicago during August). Yes, you get them all for free when you install FreeSWITCH.

The sound files in FreeSWITCH distribution contain single-words utterances, numbers, short word associations, and the like:



```
vm-forward_to_email.wav           vm-record_name1.wav
vm-from.wav                       vm-record_name2.wav
vm-goodbye.wav                    vm-repeat_message.wav
vm-greeting.wav                   vm-rerecord.wav
vm-has_been_changed_to.wav        vm-return_call.wav
vm-hello.wav                      vm-saved.wav
vm-in_folder.wav                  vm-save_message.wav
vm-last.wav                       vm-save_recording.wav
vm-listen_new.wav                 vm-selected.wav
vm-listen_saved.wav               vm-send_message_now.wav
vm-listen_to_recording_again.wav  vm-that_was_an_invalid_ext.wav
vm-listen_to_recording.wav        vm-to_exit_alt.wav
vm-mailbox_full.wav               vm-to_exit.wav
vm-main_menu_alt.wav              vm-to_forward.wav
vm-main_menu.wav                  vm-too-small.wav
vm-marked_new.wav                 vm-to_record_greeting.wav
vm-marked-urgent.wav              vm-tutorial_change_pin.wav
vm-mark_message_new.wav           vm-tutorial_record_name.wav
vm-mark-urgent.wav                vm-tutorial_yes_no.wav
vm-message_alt.wav                vm-undeleted.wav
vm-message_envelope.wav           vm-undelete_message.wav
vm-message_forwarded.wav          vm-urgent-new.wav
vm-message_from.wav               vm-urgent-saved.wav
vm-message_number.wav             vm-urgent.wav
vm-messages_alt.wav               vm-voicemail_password_is_web_password.wav
vm-messages.wav                   vm-you_have.wav
root@lxc111:/usr/local/freeswitch/sounds/en/us/callie/voicemail/8000#
```

Sound files locations, sampling rate and fallbacks

In FreeSWITCH configuration (for example in dialplan, IVRs, phrases, scripts, whatever), sound files are located by a full absolute path (that it, beginning with a slash "/"), or by a relative path. When the path is relative, the file resolves to $${sound_prefix}/relative_path, where relative_path is interpolated with the sampling rate (frequency in Hertz) of the sound file (so FreeSWITCH do not have to resample).

You can find the value of the $${sound_prefix} global variable in /usr/local/freeswitch/conf/vars.xml file, or by typing the following into the FreeSWITCH console:

```
eval $${sound_prefix}
```

For example, you call 5000 in demo configuration. You are answered by an IVR, defined in /usr/local/freeswitch/conf/ivr_menus/demo_ivr.xml. That IVR starts with the "demo_ivr_main_menu" phrase macro (defined in /usr/local/freeswitch/conf/lang/en/demo/demo-ivr.xml). The first item in that macro is as follows:

```
<action function="play-file" data="ivr/ivr-welcome_to_freeswitch.wav"/>
```

If your call is using PCMU audio codec (for example, g711), because that audio codec is sampled at 8khz, then relative path "ivr/ivr-welcome_to_freeswitch.wav" will be read from the file /usr/local/freeswitch/sounds/en/us/callie/ivr/8000/ivr-welcome_to_freeswitch.wav.

If that file is not available, FreeSWITCH will first try to use the 48khz file with the same filename (located in /usr/local/freeswitch/sounds/en/us/callie/ivr/48000/ivr-welcome_to_freeswitch.wav) because the internal FreeSWITCH audio mixing format is 48khz. If that is also unavailable, it will try 32khz, then 16khz. If they too are unavailable, FreeSWITCH will look for the file in /usr/local/freeswitch/sounds/en/us/callie/ivr/ivr-welcome_to_freeswitch.wav (for example, with no sampling frequency interpolated in the path), and will play it by autosensing its sampling rate. If nothing is there, FreeSWITCH will log an error and continue with the next audio file.

Phrase Macros: Concatenating Sound files

Phrase is a FreeSWITCH configuration XML construct that allows you to build complex voice prompts from basic sound files. The individual sound files are read one after another, interspersed with the chosen amount of silence.

Phrase macros definitions are found in files included by /usr/local/freeswitch/conf/lang/en/en.xml ("en" is for English. Substitute "en" with another international language code, for example, "fr" for French).

Let's have a look at the first lines of the most popular phrase macro from the demo configuration, located at /usr/local/freeswitch/conf/lang/en/demo/demo-ivr.xml:

```
<macro name="demo_ivr_main_menu" pause="100">
  <input pattern="(.*)">
    <match>
      <action function="play-file" data="ivr/ivr-welcome_to_freeswitch.wav"/>
      <action function="play-file" data="ivr/ivr-this_ivr_will_let_you_test_features.wav"/
      <action function="play-file" data="ivr/ivr-you_may_exit_by_hanging_up.wav"/>
 ...
 ...
    </match>
  </input>
</macro>
```

macro name (pause)

Macro "name" is what you will use in phrase invocations from dialplan or script. For example: "phrase:demo_ivr_main_menu". The optional parameter "pause" will be the number of milliseconds to sleep after playing each individual sound file that constitutes the phrase. If you omit "pause," audio files will be read one after the other seamlessly. Or, you can use the action "sleep" to insert a pause after "play-file."

input pattern

The regular expression in "input pattern" is checked against the optional input passed as an argument to "phrase" (see Input Patterns and Registers, below). If that expression matches, the phrase macro will execute the actions contained in the "match" section, else if the input pattern is not matched, the phrase macro will execute the actions contained in the "nomatch" section. An input pattern of ".*" will match whatever input, and even no input at all (in most cases, in fact, you do not pass any input to phrases).

match - nomatch

See *input pattern* before

action

With this XML tag, you define what the phrase macro will do. Most often, you will "play-file" or "sleep" for a number of milliseconds. Those two actions are the workhorses of audio file concatenation. Another useful action is "say," to correctly pronounce numbers, dates, quantities, currencies, ordinals, spellings, and so on, with consideration of the specific language (for example English, Chinese, and so on) rules. Using the "execute" action, you can invoke any FreeSWITCH API.

Phrase Macros: Invocation, Patterns and Registers

XML and regular expressions gives a lot of power to the phrase macro mechanism. The "input pattern" we saw before is useful not only to select which actions the phrase will execute (eg, those contained by "match" instead of those contained by "nomatch" tags) but also to build dynamic phrases. Read on.

Phrase Macro Invocation

Phrase macros can be invoked with or without input arguments. So, the general format is as follows:

phrase:macroname[:one or more arguments]

If present, the entire string (spaces included) after the phrase name (the phrase name is terminated by the optional second colon) will be passed as an argument to the macro.

Input Pattern and Registers

If the pattern matches, the phrase macro will find in registers the parts of input between the pattern parentheses.

It is exactly the same as with the "expression" patterns in dialplan: let's say the input pattern is "^\d(\d{3})(\d)(.*)$"; if the input is "1234567ciao", the phrase macro will then find "234" in $1, "5" in $2, and "67ciao" in $3.

Registers enable a single macro to be used to read variable contents to the caller. Let's look at how a macro can be used to read the number of messages (new or saved) in the caller's mailbox:

```
<macro name="voicemail_message_count">
  <input pattern="^(1):(.*)$" break_on_match="true">
    <match>
      <action function="play-file" data="voicemail/vm-you_have.wav"/>
      <action function="say" data="$1" method="pronounced" type="items"/>
      <action function="play-file" data="voicemail/vm-$2.wav"/>
      <action function="play-file" data="voicemail/vm-message.wav"/>
    </match>
  </input>
  <input pattern="^(\d+):(.*)$">
    <match>
      <action function="play-file" data="voicemail/vm-you_have.wav"/>
      <action function="say" data="$1" method="pronounced" type="items"/>
      <action function="play-file" data="voicemail/vm-$2.wav"/>
      <action function="play-file" data="voicemail/vm-messages.wav"/>
    </match>
  </input>
</macro>
```

In case the caller has only one new message, the phrase macro is invoked as "phrase: voicemail_message_count:1:new." The first input patterns will match because the first character of the argument string is "1." Also, that first character is enclosed by parentheses in the pattern, and the first parentheses' content (that is, the character "1" in this case) will go into the $1 register. The rest of the input, after a colon, used in this case as separator, is again enclosed within parentheses. Its content will go into the $2 register. In this case, $2 will contain the string "new."The phrase macro will execute the actions contained in "match" tags and then exit - break-on-true. Registers' values are used in actions: they give the argument to the "say" action, and they compose the name of the file in "play-file." Also note that, in this case, that is , matching a pattern with one message (new or saved), the phrase macro will correctly play the voicemail/vm-message.wav audio file, singular.

Let's look at what happens if the phrase macro is invoked for a caller with three saved messages: "phrase: voicemail_message_count:3:saved."The first input pattern will not match (the first argument character is not a "1"). The second input pattern is "^(\d+):(.*)$" and will match. All digits up to the colon will end up in the $1 register, while the $2 register will contain the string "saved." Also, please note that in the "match" section of this input pattern the file played to the caller is voicemail/vm-messages.wav, plural. Clever, huh?

XML IVRs

IVRs are those Interactive Voice Response systems we all know and love (NOT): "To learn about our special offer, press 2. To speak with an operator, please insert the 48 digits of your membership card number, then press pound."The FreeSWITCH XML IVR engine allows for building voice-menu-driven applications without programming, so even your marketing drone colleague, after some short Pavlovian conditioning, will be able to modify it himself to the latest fad.

FreeSWITCH XML IVR is basically a complex parameterized skeleton that takes a lot of arguments, and an arbitrary number of voice menu items, to create a complete application you can execute from a single dialplan extension.

Each voice menu item will react to DTMFs dialed by the caller, and will typically execute a dialplan application (bridge, transfer, playback, and so on).

XML IVR allows for timeouts, multi-digit input, text-to-speech integration, max wrong input acceptable attempts, submenus, back to main menu, menu repeat, and many more features.

IVR in demo config

Let's first have a look at the IVR that answers extension 5000 of the demo configuration. In /usr/local/freeswitch/conf/dialplan/default.xml dialplan it is defined as follows:

```
    <extension name="ivr_demo">
      <condition field="destination_number" expression="^5000$">
        <action application="answer"/>
        <action application="sleep" data="2000"/>
        <action application="ivr" data="demo_ivr"/>
      </condition>
    </extension>
```

This is easily one of the most exercised single snippets of FreeSWITCH configuration, because is so easy to use for testing purposes after a fresh installation. The relevant line is <action application="ivr" data="demo_ivr"/>, which will execute the "ivr" dialplan application, giving the string "demo_ivr" as argument.

The dialplan application "ivr" will look into FreeSWITCH configuration, searching for a voice menu where the "name" field has the value "demo_ivr". In this case, it will find it in the XML file /usr/local/freeswitch/conf/ivr_menus/demo_ivr.xml, located in the default voice menus configuration directory. The filename can be completely different; what is important is the value of the "name" field of the menu.

The voice menu "demo_ivr" has been written to exercise many features that constitute the building blocks of an interactive system: originating and bridging a call, doing an echo test, playing music on hold, connecting to one of the internal registered phones, playing a voice sub-menu, returning to the main menu, transferring to another extension in dialplan, and so on:

```
<include>
      greet-long="phrase:demo_ivr_main_menu"
  <menu name="demo_ivr"
      greet-short="phrase:demo_ivr_main_menu_short"
      invalid-sound="ivr/ivr-that_was_an_invalid_entry.wav"
      exit-sound="voicemail/vm-goodbye.wav"
      confirm-macro=""
      confirm-key=""
      tts-engine="flite"
      tts-voice="rms"
      confirm-attempts="3"
      timeout="10000"
      inter-digit-timeout="2000"
      max-failures="3"
      max-timeouts="3"
      digit-len="4">
    <entry action="menu-exec-app" digits="1" param="bridge sofia/$${domain}/888@conference.fr
    <entry action="menu-exec-app" digits="2" param="transfer 9196 XML default"/>    <!-- FS e
    <entry action="menu-exec-app" digits="3" param="transfer 9664 XML default"/>    <!-- MOH
    <entry action="menu-exec-app" digits="4" param="transfer 9191 XML default"/>    <!-- Clue
    <entry action="menu-exec-app" digits="5" param="transfer 1234*256 enum"/>       <!-- Scre
    <entry action="menu-sub" digits="6" param="demo_ivr_submenu"/>
    <entry action="menu-exec-app" digits="/^(10[01][0-9])$/" param="transfer $1 XML features'
    <entry action="menu-top" digits="9"/>          <!-- Repeat this menu -->
  </menu>
<!-- ========================================================= -->
  <menu name="demo_ivr_submenu"
      greet-long="phrase:demo_ivr_sub_menu"
      greet-short="phrase:demo_ivr_sub_menu_short"
      invalid-sound="ivr/ivr-that_was_an_invalid_entry.wav"
      exit-sound="voicemail/vm-goodbye.wav"
      timeout="15000"
      max-failures="3"
      max-timeouts="3">
    <entry action="menu-top" digits="*"/>
   </menu>
</include>
```

The general workingsare probably already clear on first read. Also, please test it; you'll see that as time passes, it will become very familiar. In the next section, we'll look at the usage of each argument field that determines the voice XML "menu" behavior, and how to add "entries" to it (for example, actions that will be executed by the voice menu when the caller dials some DTMFs).

XML voice menu fields

You can think of a "menu" in XML IVR as a general frame whose different elements change behavior based on parameters in the menu definition. That menu frame defines what will be told to the caller at first iteration - which usually include an initial greetingsuch as "thanks for calling ACME," followed by instructions on IVR usage: "press 1 for conference, press 2 for echo test," and so on. Another field defines what will be played to the caller in subsequent passes (for example, if you press 9 to hear the menu again, you will not be thanked for calling, you'll be read the instructions directly). Fields define what will be read before hanging up on the caller, or after an invalid input (for example, pressing 4 when only allowed to dial 1-3, 5-7, *, and #).

All fields that define what will be read can be a "phrase" construct, a sound file in the default sound directory, a sound file with an absolute path, or a "say" TTS construct (that is, using the defined Text-To-Speech engine)

name

This is the name that will be referred to by the dialplan "ivr" app. It has nothing to do with the filename. A single XML file can contain multiple menu definitions, and they will be found by the "name" field (as long as the file is in the ivr-menus directory).

greet-long

This is what will be read to the caller at first pass; you may want to include your company name and a salutation to the caller, followed by the instructions on how to use the IVR, "press 1 for sales, press 2 for support," and so on.

greet-short

Here you define what will be read to the caller on subsequent passes, when she comes back from a submenu, or when she wants to re-listen to the menu. Usually, instructions here will be terse; no cheers, and so on.

invalid-sound

This is read when the caller inputs a DTMF combination that is not reflected in the menu "entries". Here you want to read the caller something like, "the number you have entered is not a valid option."

exit-sound

This is read when the menu is exiting, before hanging up on the caller (because of too many wrong inputs, timeouts waiting for inputs, and so on).

timeout

This is the time in milliseconds the voice menu will wait for the caller to begin entering DTMFs, after greet-long or greet-short has been read. After expiration, the greeting will be re-read until max-timeouts times.

inter-digit-timeout

This defines how many milliseconds to wait for the next digit, when multi-digit input is possible (for example digit-len is higher than 1), digit-len has not been reached, and the caller has not pressed confirm-key. When expired, the digits entered so far are considered the complete input, and checked against the menu entries.

max-failures

This defines how many invalid inputs will cause the invalid-sound to be played, and then the greeting to be re-read, before the menu goes to the exit-sound and then hangs up.

max-timeouts

This defines how many times the absence of input will cause the invalid-sound to be played, and then the greeting to be re-read, before the menu goes to exit-sound, and then hangs up.

digit-len

This is the maximum number of digits the caller is allowed to enter as a single input. For example, you may want this to be 4 to be able to accept as input the digit sequences 1001 to 1019, corresponding to "Local_Extension" registered phones in the demo dialplan. The caller will be allowed to enter shorter digit sequences by terminating the sequence using "confirm-key" (usually #), or by waiting for "inter-digit-timeout."

tts-engine

This is the name of the Text-To-Speech engine registered in FreeSWITCH configuration, often "flite" for testing, or "cepstral" for production.

tts-voice

This is the name of the "voice" parameter that will be passed to TTS engine. TTS engines can use different voices, for example male, female, with regional accents, and so on. As a curiosity, the "rms" voice of the flite TTS engine has been sampled from Richard Marshall Stallman, the founder and main champion of the GNU project.

confirm-key

The DTMF caller will send to terminate an input sequence before entering all digit-len digits. The default is #.

XML voice menu entries and actions

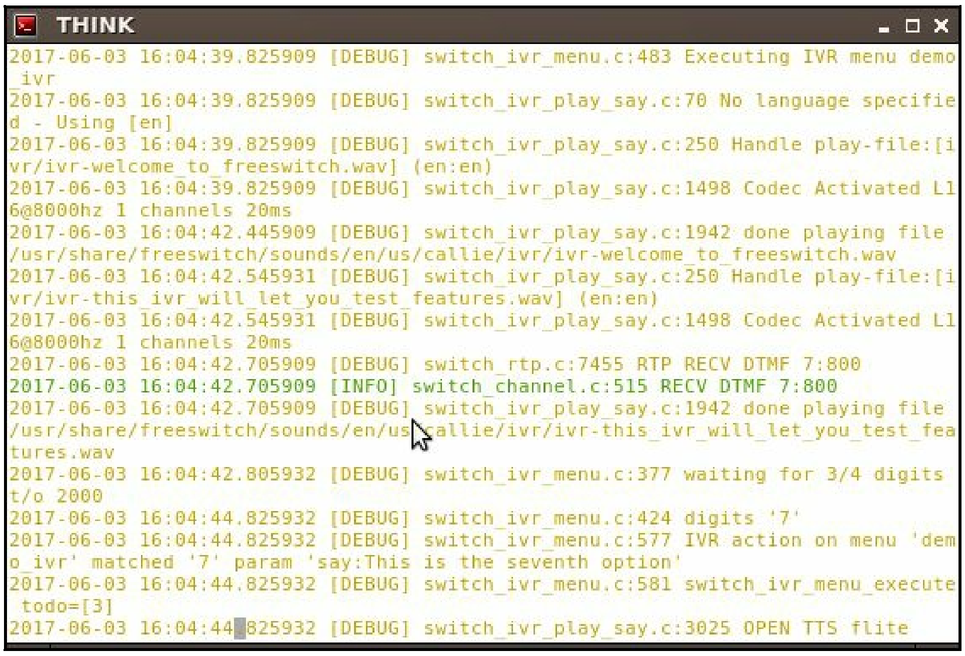
Between the opening and closing "menu" XML tags, you may want to put the real meat on the bones, that is: what will happen when the caller sends one or more DTMFs.

Those are the menu entries, in the form of an XML "entry" tag, with "action" and "digits" as mandatory attributes, and an optional "param" attribute:

```
<entry action="menu-play-sound" digits="7" param="say:This is the seventh option"/>
```

We'll look at the possible values for "action" in the following paragraphs; "digits" is the complete DTMF sequence the caller must send for the menu to execute the action (see before digit-len and confirm-key), while the "param" value is obviously passed as a parameter to the action.

In the following screenshot,we can see the debug output from the FreeSWITCH console when we call "5000" in the demo configuration and then dial 7 (we have added the entry for the digit 7, as in the previous entry example):



```
2017-06-03 16:04:39.825909 [DEBUG] switch_ivr_menu.c:483 Executing IVR menu demo
_ivr
2017-06-03 16:04:39.825909 [DEBUG] switch_ivr_play_say.c:70 No language specifie
d - Using [en]
2017-06-03 16:04:39.825909 [DEBUG] switch_ivr_play_say.c:250 Handle play-file:[i
vr/ivr-welcome_to_freeswitch.wav] (en:en)
2017-06-03 16:04:39.825909 [DEBUG] switch_ivr_play_say.c:1498 Codec Activated L1
6@8000hz 1 channels 20ms
2017-06-03 16:04:42.445909 [DEBUG] switch_ivr_play_say.c:1942 done playing file
/usr/share/freeswitch/sounds/en/us/callie/ivr/ivr-welcome_to_freeswitch.wav
2017-06-03 16:04:42.545931 [DEBUG] switch_ivr_play_say.c:250 Handle play-file:[i
vr/ivr-this_ivr_will_let_you_test_features.wav] (en:en)
2017-06-03 16:04:42.545931 [DEBUG] switch_ivr_play_say.c:1498 Codec Activated L1
6@8000hz 1 channels 20ms
2017-06-03 16:04:42.705909 [DEBUG] switch_rtp.c:7455 RTP RECV DTMF 7:800
2017-06-03 16:04:42.705909 [INFO] switch_channel.c:515 RECV DTMF 7:800
2017-06-03 16:04:42.705909 [DEBUG] switch_ivr_play_say.c:1942 done playing file
/usr/share/freeswitch/sounds/en/us/callie/ivr/ivr-this_ivr_will_let_you_test_fea
tures.wav
2017-06-03 16:04:42.805932 [DEBUG] switch_ivr_menu.c:377 waiting for 3/4 digits
t/o 2000
2017-06-03 16:04:44.825932 [DEBUG] switch_ivr_menu.c:424 digits '7'
2017-06-03 16:04:44.825932 [DEBUG] switch_ivr_menu.c:577 IVR action on menu 'dem
o_ivr' matched '7' param 'say:This is the seventh option'
2017-06-03 16:04:44.825932 [DEBUG] switch_ivr_menu.c:581 switch_ivr_menu_execute
 todo=[3]
2017-06-03 16:04:44.825932 [DEBUG] switch_ivr_play_say.c:3025 OPEN TTS flite
```

menu-exec-app

This is the workhorse of the IVR menu. It allows for executing a FreeSWITCH dialplan application, passing param as arguments. See also "action" in the Chapter 6, *XML Dialplan*.

menu-play-sound

This plays a sound file (located by an absolute path or by a path relative to the default sound directory), a "phrase:" construct, or a "say:" construct for TTS.

menu-sub

This will execute (that is, it will go to) another XML voice menu, identified by its name.

menu-back

From a sub-menu, this goes back to the previous XML voice menu.

menu-top

From a sub-menu, this goes back to the originally called-into, "top", XML voice menu.

menu-exit

You may not believe this, but menu-exit actually exits the menus!

Nesting XML IVRs

You have two choices for nesting menus (that is, for having one entry in an XML voice menu that sends the caller to another voice menu, with its own different entries, possibly with the same "digits" attribute as the previous voice menu).

You can use the "menu-sub" action, which gives you, for free, the complementary "menu-back" action. This is very handy for simple applications where only a couple voice menus are connected, and no complex inter-menus logic is involved.

In all other cases, and in cases where you cannot anticipate future development, use the "transfer" dialplan application as param in a menu-exec-app action. So, you will transfer to another extension in the dialplan, and you edit the dialplan so that your other voice menu is waiting there in an "ivr" dialplan application. This technique is much easier to use in a complex application, where there can be many logical paths between voice menus, and where you may expect to add or modify menus.

Summary

In this chapter, we looked at two FreeSWITCH mainstays for interacting with callers:

- The "phrase:" construct, which allows you to concatenate pre-recorded sound bites to build meaningful and dynamic sentences, in real time
- The "ivr" dialplan application and XML "menu" construct, which allow for easy deployment of Automatic Attendant and Interactive Voice Response systems without programming

In the next chapter, we'll look at how to use Lua scripting to bring the interaction with the caller (and the call itself) to new, incredible heights.

Lua FreeSWITCH Scripting

Executing scripts to answer incoming calls is a common way to implement complex FreeSWITCH applications. When you feel you are putting too much of your brain power into constructing complex conditional XML dialplan extensions, it's time to start scripting.

The gist is that you create an extension in the dialplan, and this extension will only be one line: execute the script. Then the script will do it all. As they say: *You do your best, script do the rest*.

The main advantage of executing a script is that you are using a proper programming language, with all the usual programming constructs required to manage whatever complexity with ease (that's what programming languages are made for).

In this chapter, we'll first have a look at the extensive FreeSWITCH scripting capabilities, and then we'll delve into Lua FreeSWITCH scripting, examining concepts and constructs that will also be useful for the other scripting languages supported by FreeSWITCH.

Many scripting languages

FreeSWITCH supports many scripting languages, both from the dialplan and command line. For each language, a specific FreeSWITCH module implements the dialplan and command line interface.

The best thing is that support for all of them comes from the same source code base: we use the **Simplified Wrapper and Interface Generator** (**SWIG**, http://www.swig.org/) to make the core FreeSWITCH library accessible to scripting.

The main advantage of using SWIG is that, in each resulting scripting language, the FreeSWITCH objects, methods, and functions all look the same, with the same arguments and return values. In fact, all the scripting languages access the same APIs from the FreeSWITCH core library (the same library used by the FreeSWITCH executable itself). SWIG ensures that this library interfacing is done in the same way for all scripting languages, instead of duplicating efforts, avoiding confusing different ways to call into the APIs.

So, there is basically only one documentation effort on how to script FreeSWITCH, and then each specific scripting language will add its own syntactic sugar. This even makes it very easy to rewrite an application from one scripting language to another, should the need arise.

The most popular languages supported by FreeSWITCH via SWIG are as follows:

- Lua
- Perl
- JavaScript
- Python
- All .NET, Mono, and Managed languages (C# and so on)
- Java

Anyway, after initializing language-specific objects, you will end up calling FreeSWITCH APIs (the same as you use from the FreeSWITCH console command line), executing dialplan applications, or interacting with FreeSWITCH Events.

Why Lua?

So, which language is better for FreeSWITCH scripting?

There is no science at all, here - all here is broscience. Yes, Lua is definitely more embeddable and faster than other languages; so what?

Scripting in FreeSWITCH has a precise role: to complement the dialplan for complex applications. That is, you use the dialplan for high-cps call dispatching. Or you use a custom C FreeSWITCH module. I would bet that, in any real-world scenario where scripting is used in its correct role, there is no meaningful performance gap between different scripting languages. Network delays, database queries, and caller reaction times will always be much more important than whatever script initialization/execution time.

That said, you balance your language choice between two sides: the language(s) you already know (this will make you immediately productive), and the popularity of its use with FreeSWITCH (so you will have immediate answers when you ask the community in IRC, HipChat, or mailing list).

The most popular scripting language in the FreeSWITCH community, by a large margin, is Lua. Obviously, this has many benefits, such as a lot of easily accessible examples, and so on. If you are in any doubt about which language to use, use Lua. Other popular choices are Perl, JavaScript, and Python.

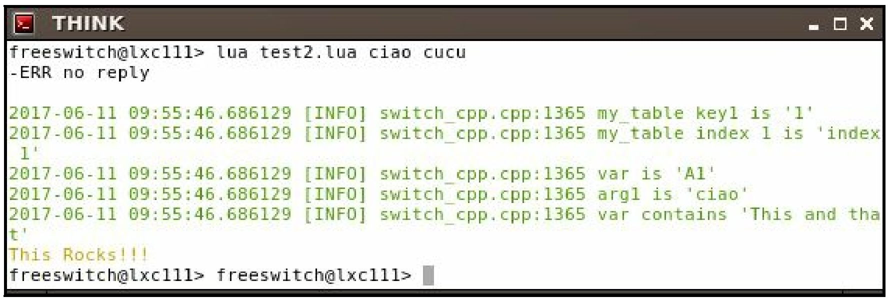
Even if you choose another language, this chapter will serve you well: you will have to change the grammar and constructs for those of your preferred language, but all the concepts will still apply, and the examples will be easily translated.

Basic Lua syntax

Lua has a simple syntax that is easy to both learn and read. The following is a simple script:

```
-- This is a sample Lua script
-- Single line comments begin with two dashes
--[[
  This is a multi-line comment.
  Everything between the double square brackets
    is part of the comment block.
]]
-- Lua is loosely typed
var = 1         -- This is a comment
var ="alpha"   -- Another comment
var ="A1"      -- You get the idea...
-- Lua makes extensive use of tables
-- Tables are a hybrid of arrays and associative arrays
val1 = 1
val2 = 2
my_table = {
     key1 = val1,
     key2 = val2,
    "index 1",
    "index 2"
}
--[[
  When the Lua script is called from FreeSWITCH
  you have a few magic objects. The main one is
  the 'freeswitch' object:
  freeswitch.consoleLog("INFO","This is a log line\n")

  If script is executed from dialplan (eg: there is
  an incoming call to manage) you have the  'session'
  object which lets you manipulate the call:
  session:answer()
  session:hangup()
]]
freeswitch.consoleLog("INFO","my_table key1 is '" .. my_table["key1"] .."'\n")
freeswitch.consoleLog("INFO","my_table index 1 is '" .. my_table[1] .."'\n")
-- Access arguments passed in
arg1 = argv[1]      -- First argument
arg2 = argv[2]      -- Second argument
-- Simple if/then
if ( var =="A1" ) then
  freeswitch.consoleLog("INFO","var is 'A1'\n")
end
--  Simple if/then/else
if ( arg1 =="ciao" ) then
  freeswitch.consoleLog("INFO","arg1 is 'ciao'\n")
else
  freeswitch.consoleLog("INFO","arg1 is not 'ciao'!\n")
end
-- String concatenation uses ..
var ="This" .." and " .. "that"
freeswitch.consoleLog("INFO","var contains '" .. var .."'\n")
freeswitch.consoleCleanLog("This Rocks!!!\n");
-- The end
```

Save this script as `/usr/local/freeswitch/scripts/test2.lua`, and then call it from the FreeSWITCH console, as `lua test2.lua ciao cucu`:



```
freeswitch@lxc111> lua test2.lua ciao cucu
-ERR no reply

2017-06-11 09:55:46.686129 [INFO] switch_cpp.cpp:1365 my_table key1 is '1'
2017-06-11 09:55:46.686129 [INFO] switch_cpp.cpp:1365 my_table index 1 is 'index
 1'
2017-06-11 09:55:46.686129 [INFO] switch_cpp.cpp:1365 var is 'A1'
2017-06-11 09:55:46.686129 [INFO] switch_cpp.cpp:1365 arg1 is 'ciao'
2017-06-11 09:55:46.686129 [INFO] switch_cpp.cpp:1365 var contains 'This and tha
t'
This Rocks!!!
freeswitch@lxc111> freeswitch@lxc111> 
```

Lua and FreeSWITCH

Calling a Lua script from FreeSWITCH makes one object available: the "**freeswitch**" object (from which you can create other basic FreeSWITCH-related objects).

You can execute a Lua script from FreeSWITCH console in two ways: `lua` and `luarun`. If you execute the script with `lua`, it will block your console until the script returns, as though it was the console thread itself to execute the script. Only after the script has exited will you see console messages. If instead you use `luarun`, a new thread will be created that will run the script (in this case the script will have no access to the `stream` object) completely independently from the console.

If the Lua script has been called from dialplan (an incoming call matches an extension where a `lua` action is executed), then an additional object is automatically already available: `session`. The object `session` represents the call leg and lets you interact with it (answer, play media, get DTMFs, hangup, and so on).

Let's play a little with the `session` Lua object. First, create a new dialplan extension that will execute the Lua script when a user dials `9910`. Edit the file `/usr/local/freeswitch/conf/dialplan/default/01_lua_example.xml` to be:

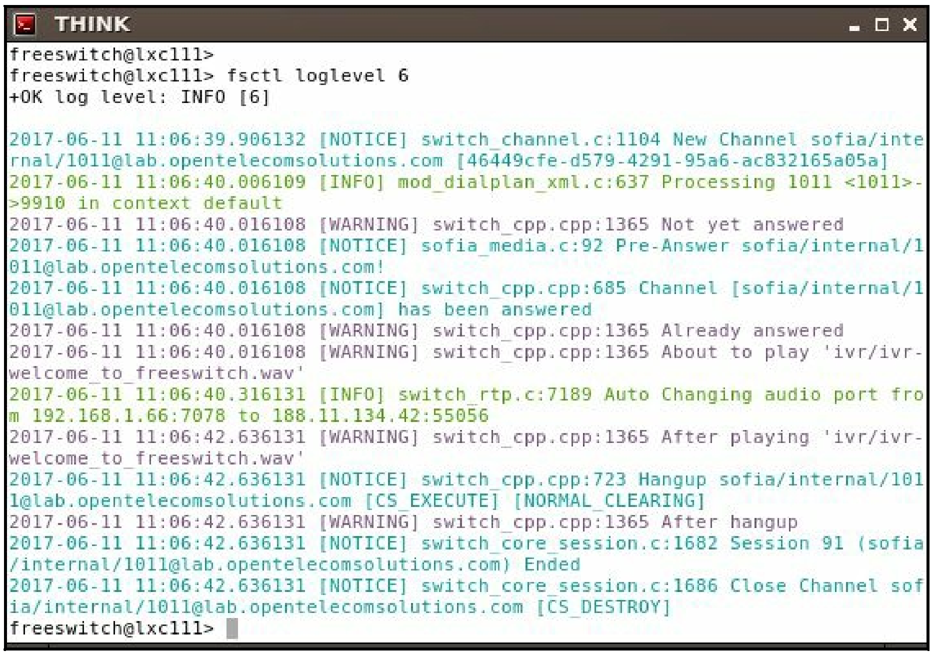
```
<extension name="Simple Lua Test">
  <condition field="destination_number" expression="^(9910)$">
    <action application="lua" data="test3.lua"/>
  </condition>
</extension>
```

Save the file, launch `fs_cli`, and issue `reloadxml`. Our dialplan is now ready to call the Lua script named `test3.lua`. Then create the `test3.lua` script in the `freeswitch/scripts/` directory, and add the following code lines:

```
-- test3.lua
-- Answer call, play a prompt, hang up
-- Set the path separator
pathsep = '/'
-- Windows users do this instead:
-- pathsep = '\'
-- Answer the call
freeswitch.consoleLog("WARNING","Not yet answered\n")
session:answer()
freeswitch.consoleLog("WARNING","Already answered\n")
-- Create a string with path and filename of a sound file
prompt ="ivr" ..pathsep .."ivr-welcome_to_freeswitch.wav"
-- Play the prompt
freeswitch.consoleLog("WARNING","About to play '" .. prompt .."'\n")
session:streamFile(prompt)
freeswitch.consoleLog("WARNING","After playing '" .. prompt .."'\n")
-- Hangup
session:hangup()
freeswitch.consoleLog("WARNING","Afterhangup\n")
```

Save the script. There is no need to `reloadxml` or anything else: scripts are automatically reloaded from filesystem if they have been modified. Using a phone that is registered on your FreeSWITCH server, dial `9910`. You will hear the sound prompt being played, and then FreeSWITCH will hangup the call.

Before placing the call, you set the loglevel to 6 or 7, to see what's happening inside FreeSWITCH (level 7 is DEBUG, and will show you the complete internals. Here we use level 6, INFO, for the sake of the screenshot):



```
freeswitch@lxc111>
freeswitch@lxc111> fsctl loglevel 6
+OK log level: INFO [6]

2017-06-11 11:06:39.906132 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@lab.opentelecomsolutions.com [46449cfe-d579-4291-95a6-ac832165a05a]
2017-06-11 11:06:40.006109 [INFO] mod_dialplan_xml.c:637 Processing 1011 <1011>-
>9910 in context default
2017-06-11 11:06:40.016108 [WARNING] switch_cpp.cpp:1365 Not yet answered
2017-06-11 11:06:40.016108 [NOTICE] sofia_media.c:92 Pre-Answer sofia/internal/1
011@lab.opentelecomsolutions.com!
2017-06-11 11:06:40.016108 [NOTICE] switch_cpp.cpp:685 Channel [sofia/internal/1
011@lab.opentelecomsolutions.com] has been answered
2017-06-11 11:06:40.016108 [WARNING] switch_cpp.cpp:1365 Already answered
2017-06-11 11:06:40.016108 [WARNING] switch_cpp.cpp:1365 About to play 'ivr/ivr-
welcome_to_freeswitch.wav'
2017-06-11 11:06:40.316131 [INFO] switch_rtp.c:7189 Auto Changing audio port fro
m 192.168.1.66:7078 to 188.11.134.42:55056
2017-06-11 11:06:42.636131 [WARNING] switch_cpp.cpp:1365 After playing 'ivr/ivr-
welcome_to_freeswitch.wav'
2017-06-11 11:06:42.636131 [NOTICE] switch_cpp.cpp:723 Hangup sofia/internal/101
1@lab.opentelecomsolutions.com [CS_EXECUTE] [NORMAL_CLEARING]
2017-06-11 11:06:42.636131 [WARNING] switch_cpp.cpp:1365 After hangup
2017-06-11 11:06:42.636131 [NOTICE] switch_core_session.c:1682 Session 91 (sofia
/internal/1011@lab.opentelecomsolutions.com) Ended
2017-06-11 11:06:42.636131 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@lab.opentelecomsolutions.com [CS_DESTROY]
freeswitch@lxc111>
```

Now, let's have a look at the various objects you'll find or create in a Lua FreeSWITCH script.

freeswitch

This is the main object, and you use its methods to create all the other FreeSWITCH-related objects (see following). Also, it has a number of non-object-creating methods, the most important of which are as follows:

- freeswitch.consoleLog("warning","lua rocks\n")
- freeswitch.bridge(session1, session2)
- freeswitch.msleep(500)
- my_globalvar = freeswitch.getGlobalVariable("varname")

freeswitch.Session and session

This object represents a call leg. You create an outbound call leg, giving freeswitch. Session a dialing string as an argument:

```
new_leg = freeswitch.Session("user/1011")
```

Let's test an example script that originates a call to a phone registered as `1011`, then originates another leg to another phone (registered as `1010`), and bridge the two legs so they can talk. Save this script as `/usr/local/freeswitch/scripts/test4.lua`, and then call it from the FreeSWITCH console, typing `lua test4.lua`, then run it again typing `luarun test4.lua` (hint: with `luarun` the console will not block):

```
freeswitch.consoleLog("WARNING","Before first call\n")
first_session = freeswitch.Session("user/1011")
if (first_session:ready()) then
        freeswitch.consoleLog("WARNING","first leg answered\n")
        second_session = freeswitch.Session("user/1010")
        if (second_session:ready()) then
                freeswitch.consoleLog("WARNING","second leg answered\n")
                freeswitch.bridge(first_session, second_session)
                freeswitch.consoleLog("WARNING","After bridge\n")
        else
                freeswitch.consoleLog("WARNING","second leg failed\n")
        end
else
        freeswitch.consoleLog("WARNING","first leg failed\n")
end
```

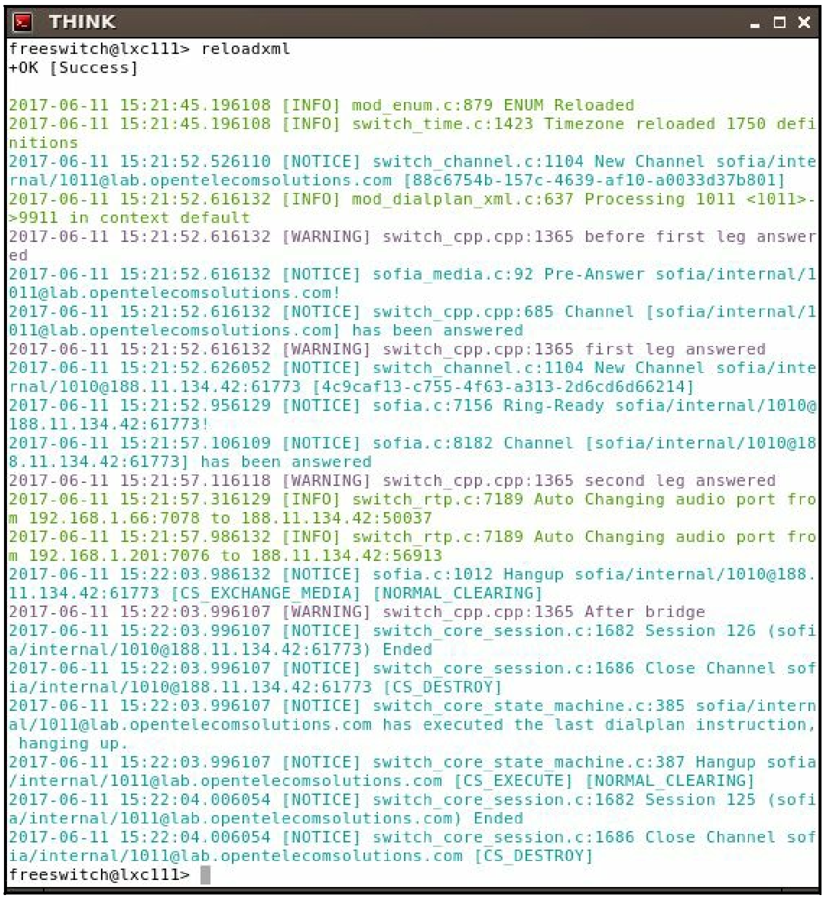
When the FreeSWITCH Lua script has been executed from dialplan (as opposed to executing the script from the console), you have a session object named `session` already existing, representing the inbound A-leg. Let's rewrite the previous example to be called from dialplan, edit the file `/usr/local/freeswitch/scripts/test5.lua`:

```
freeswitch.consoleLog("WARNING","before first leg answered\n")
session:answer()
if (session:ready()) then
        freeswitch.consoleLog("WARNING","first leg answered\n")
        second_session = freeswitch.Session("user/1010")
        if (second_session:ready()) then
                freeswitch.consoleLog("WARNING","second leg answered\n")
                freeswitch.bridge(session, second_session)
                freeswitch.consoleLog("WARNING","After bridge\n")
        else
                freeswitch.consoleLog("WARNING","second leg failed\n")
        end
else
        freeswitch.consoleLog("WARNING","first leg failed\n")
end
```

Then, create the dialplan extension, in the `/usr/local/freeswitch/conf/dialplan/default/02_lua_example.xml` file:

```
<extension name="Another Lua Test">
  <condition field="destination_number" expression="^(9911)$">
    <action application="lua" data="test5.lua"/>
  </condition>
</extension>
```

Type `reloadxml` from the console to activate the new extension. This time we'll call 9911 from a registered phone. The script will answer our call, then originate another call to the phone registered as `1010`, and will bridge the two legs together so we can talk:



```
freeswitch@lxc111> reloadxml
+OK [Success]

2017-06-11 15:21:45.196108 [INFO] mod_enum.c:879 ENUM Reloaded
2017-06-11 15:21:45.196108 [INFO] switch_time.c:1423 Timezone reloaded 1750 defi
nitions
2017-06-11 15:21:52.526110 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@lab.opentelecomsolutions.com [88c6754b-157c-4639-af10-a0033d37b801]
2017-06-11 15:21:52.616132 [INFO] mod_dialplan_xml.c:637 Processing 1011 <1011>-
>9911 in context default
2017-06-11 15:21:52.616132 [WARNING] switch_cpp.cpp:1365 before first leg answer
ed
2017-06-11 15:21:52.616132 [NOTICE] sofia_media.c:92 Pre-Answer sofia/internal/1
011@lab.opentelecomsolutions.com!
2017-06-11 15:21:52.616132 [NOTICE] switch_cpp.cpp:685 Channel [sofia/internal/1
011@lab.opentelecomsolutions.com] has been answered
2017-06-11 15:21:52.616132 [WARNING] switch_cpp.cpp:1365 first leg answered
2017-06-11 15:21:52.626052 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1010@188.11.134.42:61773 [4c9caf13-c755-4f63-a313-2d6cd6d66214]
2017-06-11 15:21:52.956129 [NOTICE] sofia.c:7156 Ring-Ready sofia/internal/1010@
188.11.134.42:61773!
2017-06-11 15:21:57.106109 [NOTICE] sofia.c:8182 Channel [sofia/internal/1010@18
8.11.134.42:61773] has been answered
2017-06-11 15:21:57.116118 [WARNING] switch_cpp.cpp:1365 second leg answered
2017-06-11 15:21:57.316129 [INFO] switch_rtp.c:7189 Auto Changing audio port fro
m 192.168.1.66:7078 to 188.11.134.42:50037
2017-06-11 15:21:57.986132 [INFO] switch_rtp.c:7189 Auto Changing audio port fro
m 192.168.1.201:7076 to 188.11.134.42:56913
2017-06-11 15:22:03.986132 [NOTICE] sofia.c:1012 Hangup sofia/internal/1010@188.
11.134.42:61773 [CS_EXCHANGE_MEDIA] [NORMAL_CLEARING]
2017-06-11 15:22:03.996107 [WARNING] switch_cpp.cpp:1365 After bridge
2017-06-11 15:22:03.996107 [NOTICE] switch_core_session.c:1682 Session 126 (sofi
a/internal/1010@188.11.134.42:61773) Ended
2017-06-11 15:22:03.996107 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1010@188.11.134.42:61773 [CS_DESTROY]
2017-06-11 15:22:03.996107 [NOTICE] switch_core_state_machine.c:385 sofia/intern
al/1011@lab.opentelecomsolutions.com has executed the last dialplan instruction,
 hanging up.
2017-06-11 15:22:03.996107 [NOTICE] switch_core_state_machine.c:387 Hangup sofia
/internal/1011@lab.opentelecomsolutions.com [CS_EXECUTE] [NORMAL_CLEARING]
2017-06-11 15:22:04.006054 [NOTICE] switch_core_session.c:1682 Session 125 (sofi
a/internal/1011@lab.opentelecomsolutions.com) Ended
2017-06-11 15:22:04.006054 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@lab.opentelecomsolutions.com [CS_DESTROY]
freeswitch@lxc111>
```

Another important hint you get from the execution of those scripts is that, when two legs are bridged, the script will stop at the "bridge" script line until one of the two legs hangs up (so, the bridge ends). While the bridge is up, our script simply waits for the bridge to end. If you want the script to continue immediately after bridging, from a main script you would need to spawn another thread, find the uuid of each call leg from the new thread, and use the bridge action from it (this is a very advanced topic; we will not further elaborate here). Session objects, perhaps together with API objects, are the ones you will use most often in your FreeSWITCH scripting. They have a lot of methods, corresponding to the various manipulations and interactions you can have with a call leg. Many of them are parallel/similar to dialplan applications.

Following is a list of them; you'll find them commented and explained when you search for Lua API Reference in our docs at https://freeswitch.org/confluence. We'll see many of them in action later, in "XML demo IVR in Lua" section in this chapter:

- session:answer
- session:answered
- session:bridged
- session:check_hangup_hook
- session:collectDigits
- session:consoleLog
- session:destroy
- session:execute
- session:flushDigits
- session:flushEvents
- session:get_uuid
- session:getDigits
- session:getState
- session:getVariable
- session:hangup
- session:hangupCause
- session:hangupState
- session:insertFile
- session:mediaReady
- session:originate
- session:playAndGetDigits
- session:preAnswer
- session:read
- session:ready
- session:recordFile
- session:sayPhrase
- session:sendEvent
- session:setAutoHangup
- session:setHangupHook
- session:setInputCallback
- session:setVariable
- session:sleep
- session:speak
- session:say
- session:streamFile
- session:transfer
- session:unsetInputCallback
- session:waitForAnswer

freeswitch.API

This is the second workhorse of FreeSWITCH Lua scripting. The API object allows you to send API commands to FreeSWITCH exactly as if you were at the console.

API commands are provided by `mod_commands` (look it up in https://freeswitch.org/confluence) and by many other modules (in fact, almost all modules provide additional API commands). You can see a list of all API commands available in your FreeSWITCH server by issuing the following from the FreeSWITCH console (help will do the same):

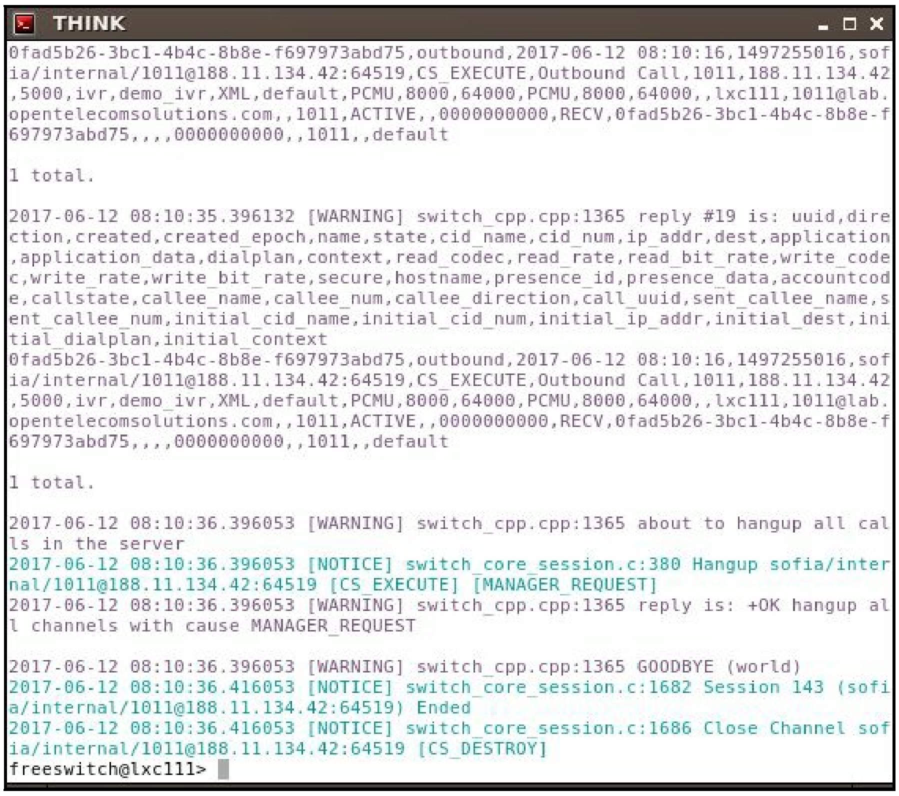
```
show api
```

Let's look at the `freeswitch.API` object doing its things: edit the file `/usr/local/freeswitch/scripts/test6.lua`:

```
freeswitch.consoleLog("WARNING","before creating the API object\n")
api = freeswitch.API()
freeswitch.consoleLog("WARNING","after creating the API object\n")
reply = api:executeString("version")
freeswitch.consoleLog("WARNING","reply is: " .. reply .."\n")
reply = api:executeString("status")
freeswitch.consoleLog("WARNING","reply is: " .. reply .."\n")
reply = api:executeString("sofia status")
freeswitch.consoleLog("WARNING","reply is: " .. reply .."\n")
reply = api:executeString("bgapi originate user/1011 5000")
-- reply = api:executeString("originate user/1011 5000")
freeswitch.consoleLog("WARNING","reply is: " .. reply .."\n")
counter = 0
while(counter < 20) do
        reply = api:executeString("show channels")
        freeswitch.consoleLog("WARNING","reply #" .. counter .. " is: " .. reply .."\n")
        counter = counter + 1
        api:executeString("msleep 1000")
end
freeswitch.consoleLog("WARNING","about to hangup all calls in the server\n")
reply = api:executeString("hupall")
freeswitch.consoleLog("WARNING","reply is: " .. reply .."\n")
freeswitch.consoleLog("WARNING","GOODBYE (world)\n")
```

This script will first create the API object, then use it to interact with the FreeSWITCH server as if it were you at the console (it issues the commands `version`, `status`, and `sofia status`, then print back on the console what the server returned after each API command). After those informative actions, our script takes a more decisive route, and originates an outbound call leg toward a phone registered as `1011` to the server. Then it shows the result of the `show channels` command twenty times, pausing for one second, and eventually hangs up all the calls on the FreeSWITCH server.

You can execute it from the console by issuing `lua test6.lua`; it will block your console until the script exits. Or you can execute it issuing `luarun test6.lua` and you'll see the entire process in real time:



```
0fad5b26-3bc1-4b4c-8b8e-f697973abd75,outbound,2017-06-12 08:10:16,1497255016,sof
ia/internal/1011@188.11.134.42:64519,CS_EXECUTE,Outbound Call,1011,188.11.134.42
,5000,ivr,demo_ivr,XML,default,PCMU,8000,64000,PCMU,8000,64000,,lxc111,1011@lab.
opentelecomsolutions.com,,1011,ACTIVE,,0000000000,RECV,0fad5b26-3bc1-4b4c-8b8e-f
697973abd75,,,,0000000000,,1011,,default

1 total.

2017-06-12 08:10:35.396132 [WARNING] switch_cpp.cpp:1365 reply #19 is: uuid,dire
ction,created,created_epoch,name,state,cid_name,cid_num,ip_addr,dest,application
,application_data,dialplan,context,read_codec,read_rate,read_bit_rate,write_code
c,write_rate,write_bit_rate,secure,hostname,presence_id,presence_data,accountcod
e,callstate,callee_name,callee_num,callee_direction,call_uuid,sent_callee_name,s
ent_callee_num,initial_cid_name,initial_cid_num,initial_ip_addr,initial_dest,ini
tial_dialplan,initial_context
0fad5b26-3bc1-4b4c-8b8e-f697973abd75,outbound,2017-06-12 08:10:16,1497255016,sof
ia/internal/1011@188.11.134.42:64519,CS_EXECUTE,Outbound Call,1011,188.11.134.42
,5000,ivr,demo_ivr,XML,default,PCMU,8000,64000,PCMU,8000,64000,,lxc111,1011@lab.
opentelecomsolutions.com,,1011,ACTIVE,,0000000000,RECV,0fad5b26-3bc1-4b4c-8b8e-f
697973abd75,,,,0000000000,,1011,,default

1 total.

2017-06-12 08:10:36.396053 [WARNING] switch_cpp.cpp:1365 about to hangup all cal
ls in the server
2017-06-12 08:10:36.396053 [NOTICE] switch_core_session.c:380 Hangup sofia/inter
nal/1011@188.11.134.42:64519 [CS_EXECUTE] [MANAGER_REQUEST]
2017-06-12 08:10:36.396053 [WARNING] switch_cpp.cpp:1365 reply is: +OK hangup al
l channels with cause MANAGER_REQUEST

2017-06-12 08:10:36.396053 [WARNING] switch_cpp.cpp:1365 GOODBYE (world)
2017-06-12 08:10:36.416053 [NOTICE] switch_core_session.c:1682 Session 143 (sofi
a/internal/1011@188.11.134.42:64519) Ended
2017-06-12 08:10:36.416053 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@188.11.134.42:64519 [CS_DESTROY]
freeswitch@lxc111>
```

Also, note that you can use this same "execute a Lua script" technique from outside the console, that is, from a shell or a cron script, in the following two forms:

```
fs_cli -x "lua test6.lua"
fs_cli -x "luarun test6.lua"
```

You will have no console printing of logs, obviously. To get your printouts back you can delete the following line:

```
freeswitch.consoleLog("WARNING","reply #" .. counter .. " is: " .. reply .."\n")
```

and insert this one at its place:

```
stream:write("reply #" .. counter .. " is: " .. reply .."\n")
```

This will not work if you invoke the script with `luarun` (`luarun` has no access to the stream object). So, you're stuck collecting your printouts at script's exit.

freeswitch.Dbh

The Dbh object is a handle to all kinds of databases supported natively by FreeSWITCH, namely: **SQLite** (a filesystem-based database, suitable for moderate, non-heavy parallel-concurrent access), **PostgreSQL** (a heavy duty, Enterprise- and Carrier-ready, massively parallel concurrent-access database), and **ODBC** (ODBC is a middleware that gives you access to all databases that have an ODBC-compatible driver. Let's say that all databases have an ODBC driver. Yes, **MySQL** has one, and ODBC is the way you connect to MySQL and MariaDB from FreeSWITCH. By the way, I originally ported iODBC to Unix back when archaeopteryxes were roaming the skies).

An alternative way to deal with databases is to use **LuaSQL**. Search https://freeswitch.org/confluence about it.

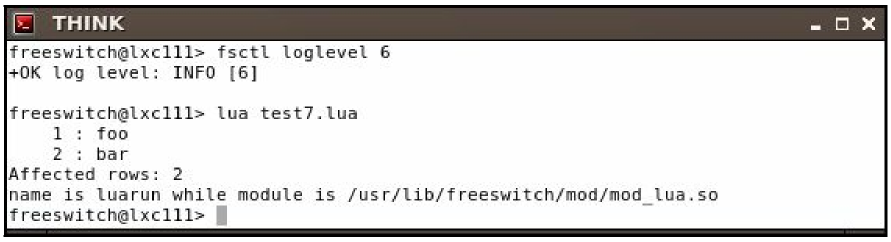
The Dbh object gives you a handle from the FreeSWITCH server's database-pooling mechanism. These are database connections that FreeSWITCH manages, so they're always available and ready. This will not incur the overhead of opening a direct database connection each time you need it (in a script that would be per execution). Instead, you connect to the FreeSWITCH server, which will multiplex you in its own pool of database connections. That is to say, the connection is already there, managed by FreeSWITCH. You only need a handle for it. FreeSWITCH will create the connection or grow the number of actual database connections in the pool as needed, in the background.

You can then use the Dbh object to execute queries and all the usual database operations. Let's look at Dbh in action: edit the `/usr/local/freeswitch/scripts/test7.lua` file as follows:

```
dbh = freeswitch.Dbh("sqlite://my_db") -- sqlite database in subdirectory "db"
assert(dbh:connected()) -- exits the script if we didn't connect properly
dbh:test_reactive("SELECT * FROM my_table",
                  "DROP TABLE my_table",
                  "CREATE TABLE my_table (id INTEGER(8), name VARCHAR(255))")
dbh:query("INSERT INTO my_table VALUES(1, 'foo')") -- populate the table
dbh:query("INSERT INTO my_table VALUES(2, 'bar')") -- with some test data
dbh:query("SELECT id, name FROM my_table", function(row)
stream:write(string.format("%5s : %s\n", row.id, row.name))
end)
dbh:query("UPDATE my_table SET name = 'changed'")
stream:write("Affected rows: " .. dbh:affected_rows() .. "\n")
dbh:release() -- optional
------
------
dbh2 = freeswitch.Dbh("sqlite://core") -- sqlite database in subdirectory "db"
assert(dbh2:connected()) -- exits the script if we didn't connect properly
dbh2:query("select name,filename from interfaces where name='luarun'", function(row)
stream:write("name is ".. row.name .. " while module is " .. row.filename)
end)
```

```
dbh2:release() -- optional
```

You can run this script from console issuing `lua test7.lua`



```
THINK
freeswitch@lxc111> fsctl loglevel 6
+OK log level: INFO [6]

freeswitch@lxc111> lua test7.lua
    1 : foo
    2 : bar
Affected rows: 2
name is luarun while module is /usr/lib/freeswitch/mod/mod_lua.so
freeswitch@lxc111>
```

The first part of this script interacts with an arbitrary database, it could be a remote PostgreSQL or a local MySQL reached via ODBC. For the sake of simplicity, the script actually creates the Dbh object by connecting to the `my_db` SQLite database in the default FreeSWITCH directory for SQLite databases (`/usr/local/freeswitch/db`). Then it uses the `test_reactive` method, which runs a query, and in case of error performs two SQL operations: in this case a "drop table", and a "create table". This method is useful to check if a database, table, or schema exists, and create it if needed. Be aware: if you delete the table while FreeSWITCH is running, you will need to restart FreeSWITCH for the test_reactive method to work properly. Then it uses the query method to interact with the database (search for "Lua API Reference" at http://freeswitch.org/confluence for details on how to print the query results, that is, the callback function that will run for each returned row), and eventually release the database handle.

The second part of the script connects to the FreeSWITCH internal core database, where FreeSWITCH keeps its own internal data. In this case we query the interface table, the same table that is queried when you issue "show interfaces" from the console.

So, yes, you can read, write, and delete from FreeSWITCH internal databases and tables, where FreeSWITCH keeps its guts and soul. Obviously, you can damage FreeSWITCH by doing that. With great power comes great responsibility. If damage you done, stop FreeSWITCH, delete all databases into the `/usr/local/freeswitch/db/` directory, or all tables in the PGSQL|ODBC databases, and restart FreeSWITCH. They will be re-created at startup.

freeswitch.Event

This object allows your script to inject an event in FreeSWITCH, which will then be a full member of the FreeSWITCH Event System:

```
--Create Custom event
event = freeswitch.Event("custom", "dial::dial-result")
event:addBody("dial_record_id: " .. dial_record_id .. "\n" ..
        "call_disposition: " .. Disposition .. "\n" ..
        "campaign_number: "  .. Campaign .. "\n" ..
        "called_number: "    .. dial_num .."\n")
event:fire();
```

Events sent this way can interact with FreeSWITCH core itself, or with modules, or be consumed by custom modules or applications. See in Chapter 11, *ESL - FreeSWITCH Controlled by Events* about the role of events in FreeSWITCH.

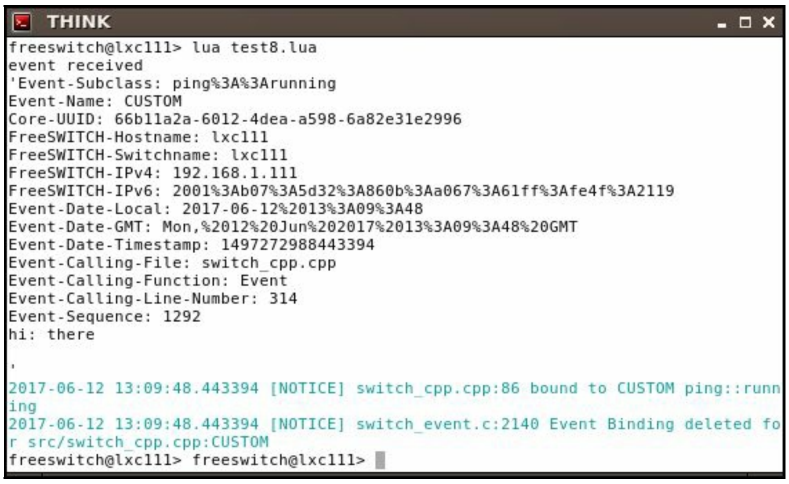
For a complete example, see the following section.

freeswitch.EventConsumer

This object creates a listener/consumer for a certain kind of event in FreeSWITCH Event System:

```
-- You may subscribe to specific events if you want to, and even subclasses
-- con1 = freeswitch.EventConsumer("CUSTOM")
-- or
-- con2 = freeswitch.EventConsumer("CUSTOM","conference::maintenance")

--listen for an event specific subclass
con3 = freeswitch.EventConsumer("CUSTOM", "ping::running")

-- create an event in that specific subclass
event = freeswitch.Event("CUSTOM", "ping::running")
event:addHeader("hi", "there")
-- fire the event
event:fire()

-- look in the listened events stack for events to "pop" out, block there for max 20 seconds
received_event = con3:pop(1, 20000)
if(received_event) then
    stream:write("event received\n")
    stream:write(received_event:serialize("text"))
else
    stream:write("no event\n")
end
```

Let's look at events in action: edit the `/usr/local/freeswitch/scripts/test8.lua` file:



```
freeswitch@lxc111> lua test8.lua
event received
'Event-Subclass: ping%3A%3Arunning
Event-Name: CUSTOM
Core-UUID: 66b11a2a-6012-4dea-a598-6a82e31e2996
FreeSWITCH-Hostname: lxc111
FreeSWITCH-Switchname: lxc111
FreeSWITCH-IPv4: 192.168.1.111
FreeSWITCH-IPv6: 2001%3Ab07%3A5d32%3A860b%3Aa067%3A61ff%3Afe4f%3A2119
Event-Date-Local: 2017-06-12%2013%3A09%3A48
Event-Date-GMT: Mon,%2012%20Jun%202017%2013%3A09%3A48%20GMT
Event-Date-Timestamp: 1497272988443394
Event-Calling-File: switch_cpp.cpp
Event-Calling-Function: Event
Event-Calling-Line-Number: 314
Event-Sequence: 1292
hi: there

'
2017-06-12 13:09:48.443394 [NOTICE] switch_cpp.cpp:86 bound to CUSTOM ping::runn
ing
2017-06-12 13:09:48.443394 [NOTICE] switch_event.c:2140 Event Binding deleted fo
r src/switch_cpp.cpp:CUSTOM
freeswitch@lxc111> freeswitch@lxc111>
```

You execute this script issuing `lua test8.lua` from the FreeSWITCH console. Note that the binding messages are printed on console after the output of stream:write(), that is, after our script has exited.

It would be interesting to split the script into two, one that listens and one that sends, and execute them using two different console instances on the same FreeSWITCH server.

freeswitch.IVRMenu

This object allows you to create XML IVR menus on the fly, with all the XML IVR trappings such as greet_long, greet_short, and so on. See previous chapter.

Lua scripting IVR Applications

The concept of an Automated Attendant or IVR is to answer an incoming call, then present some options for the caller to choose from via DTMFs, loop reading the menu while waiting for the caller to input their choice, execute the chosen option or tell the caller their input is wrong, and continue to loop until a timeout or a maximum number of loops is reached. Options can be as complex as desired, involving any kind of external interaction with databases, webservers, or other services. One form of option can be to gather a long string of DTMFs from the caller, for example, a numerical membership ID. Also, the caller must be able to jump from one menu to another containing different options, by choosing the appropriate option. Heck, you got the idea.

playAndGetDigits

The diva of IVRs in FreeSWITCH scripting is the Session object method **playAndGetDigits** (initial "p" letter is small cap, rest is camel case). She's the one all other things revolves around, like "Le Roi Soleil" in 1700 France.

Syntax

```
digits = session:playAndGetDigits (
          min_digits, max_digits, max_attempts, timeout, terminators,
          prompt_audio_files, input_error_audio_files,
          digit_regex, variable_name, digit_timeout,
          transfer_on_failure)
```

Arguments

min_digits	The minimum number of digits required.
max_digits	The maximum number of digits allowed.
max_attempts	The number of times this function waits for digits and replays the prompt_audio_file when digits do not arrive.
timeout	The time (in milliseconds) to wait for a digit.
terminators	A string containing a list of digits that cause this function to terminate.
prompt_audio_file	The initial audio file to play. Playback stops if digits arrive while playing. This file is replayed after each timeout, up to max_attempts.
input_error_audio_file	The audio file to play when a digit not matching the digit_regex is received. Received digits are discarded while this file plays. Specify an empty string if this feature is not used.
digit_regex	The regular expression used to validate received digits.
variable_name	**(Optional)** The channel variable used to store the received digits.
digit_timeout	**(Optional)** The inter-digit timeout (in milliseconds). When provided, resets the timeout clock after each digit is entered, thus giving users with limited mobility the ability to slowly enter digits without causing a timeout. If not specified, digit_timeout is set to timeout.

transfer_on_failure	**(Optional)** In the event of a failure, this function will transfer the session to an extension in the dialplan. The syntax is "*extension-name* [*dialplan-id* [*context*]]".

Behavior

- This method returns an empty string when all timeouts and retry counts are exhausted.
- When the maximum number of allowable digits is reached, the function returns immediately, even if a terminator was not entered.
- If the user forgets to press one of the terminators, but has made a correct entry, the digits are returned after the next timeout.
- The session has to be answered before any digits can be processed. If you do not answer the call you, the audio will still play, but no digits will be collected.

XML demo IVR in Lua

The following is an educational rewrite, in Lua, of the demo XML IVR shipped in the demo configuration, the one that answers when you call `5000` from a registered phone. Write it in the `/usr/local/freeswitch/scripts/test9.lua` file:

```
conference_call_session = NIL
first_time_main_menu = false
first_time_submenu = false

api = freeswitch.API()
is_a_demo_user_extension = "false"
-----------------------------------------------------------------------
session:answer()
session:execute("playback", "silence_stream://1000")

if(not session:ready())then goto END end
digits = session:playAndGetDigits (1,4,1,10000,
         '#','phrase:demo_ivr_main_menu',
         'ivr/ivr-that_was_an_invalid_entry.wav',
         '^\\d$|^10[01][0-9]$','digits',2000)
first_time_main_menu = true
goto MAIN_MENU_OPTIONS
-----------------------------------------------------------------------
::MAIN_MENU_SECOND_TIME::
if(not session:ready())then goto END end
if(conference_call_session) then goto END end
digits = session:playAndGetDigits (1,4,2,10000,
         '#','phrase:demo_ivr_main_menu_short',
         'ivr/ivr-that_was_an_invalid_entry.wav',
         '^\\d$|^10[01][0-9]$','digits',2000)
first_time_main_menu = false
goto MAIN_MENU_OPTIONS
-----------------------------------------------------------------------
::MAIN_MENU_OPTIONS::
if(not session:ready())then goto END end
if(conference_call_session) then goto END end
if(digits == "") then
        freeswitch.consoleLog("WARNING", "No digits!\n")
elseif(digits == "1") then
        conference_call_session = freeswitch.Session(
         "sofia/internal/888@conference.freeswitch.org")

        if(conference_call_session:ready()) then
                conference_call_session:execute("playback", "silence_stream://1000")
                if(not session:ready())then goto END end
                if(not conference_call_session:ready())then goto END end
                freeswitch.bridge(session, conference_call_session)
        end
elseif(digits == "2") then
        session:execute("transfer", "9196 XML default")
elseif(digits == "3") then
        session:execute("transfer", "9664 XML default")
elseif(digits == "4") then
        session:execute("transfer", "9191 XML default")
elseif(digits == "5") then
        session:execute("transfer", "1234*256 enum")
elseif(digits == "6") then
        goto SUBMENU
else
        is_a_demo_user_extension = api:executeString(
         "regex " .. digits .. "|^(10[01][0-9])$")

        if(is_a_demo_user_extension == "true") then
                session:execute("transfer", digits .. " XML features")
        end
end
if(first_time_main_menu) then goto MAIN_MENU_SECOND_TIME end
goto END
-----------------------------------------------------------------------
::SUBMENU::
if(not session:ready())then goto END end
digits = session:playAndGetDigits (1,1,1,5000,
         '#','phrase:demo_ivr_sub_menu',
         'ivr/ivr-that_was_an_invalid_entry.wav',
         '\\*','digits',2000)
first_time_submenu = true
goto SUBMENU_OPTIONS
-----------------------------------------------------------------------
::SUBMENU_SECOND_TIME::
if(not session:ready())then goto END end
digits = session:playAndGetDigits (1,1,2,5000,
         '#','phrase:demo_ivr_sub_menu_short',
         'ivr/ivr-that_was_an_invalid_entry.wav',
         '\\*','digits',2000)
first_time_submenu = false
goto SUBMENU_OPTIONS
-----------------------------------------------------------------------
::SUBMENU_OPTIONS::
if(not session:ready())then goto END end
if(digits == "") then
        freeswitch.consoleLog("WARNING", "No digits in submenu!\n")
elseif(digits == "*") then
        goto MAIN_MENU_SECOND_TIME
end
if(first_time_submenu) then goto SUBMENU_SECOND_TIME end
goto END
-----------------------------------------------------------------------
::END::
if(session:ready())then
        session:execute("playback", "voicemail/vm-goodbye.wav")
end
freeswitch.consoleLog("WARNING", "END!\n")
```

Then, create the dialplan extension, in the `/usr/local/freeswitch/conf/dialplan/default/02_lua_example.xml` file:

```
<extension name="XML IVR in Lua">
  <condition field="destination_number" expression="^(9912)$">
    <action application="lua" data="test9.lua"/>
  </condition>
</extension>
```

Type `reloadxml` from the console to activate the new extension, and call `9912` from a registered phone.

As you can see, we need a way to simulate the XML IVR method to have different prompts if it's the first time the menu is read to the caller, or if it's a successive pass.

We do that here by executing `playAndGetDigits()` the first time with the long prompt and only one repetition as arguments, while in successive passes `playAndGetDigits()` gets the short prompt and two repetitions (for a total of three, like in the demo XML IVR) as arguments.

The regex we set in the main menu `playAndGetDigits` arguments, `'^\\d$|^10[01][0-9]$'`, means: a single digit, or 1000...1019 (1000...1019 are the extensions that correspond to local users in the demo configuration). If anything else is entered, it is considered invalid input. We also set the min digits to 1, the max digits to 4, and the inter-digits timeout to 2 seconds.

When a user extension is entered (1000...1019), we then perform a completely unnecessary further check, which is there just to show you how to use FreeSWITCH's Perl Compatible Regular Expressions via an API object, instead of juggling with the Lua's own quirky (in my opinion) regular expressions. Be aware that `is_a_demo_user_extension` is a string value, not a boolean. That is, it contains the string `true` or the string `false` as returned by FreeSWITCH when you issue the `regex` API command.

In the submenu, we accept as `playAndGetDigits` input only the asterisk, and the min and max digits are set to `1`.

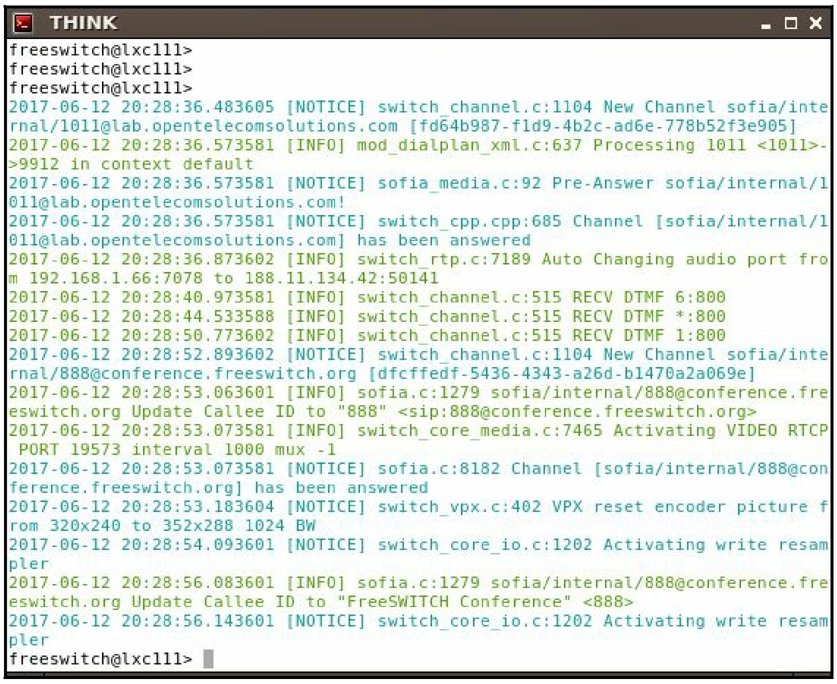
Also, you will see the extensive use of GOTO and labels, so we can safely jump from one menu to another, and have a completely modular script where in future we can insert additional menus or options as needed.

Always checking session:ready()

In this example script there is an almost compulsive checking of the state of the Session objects - are they `ready()`? This is to check if the call has been answered, if it has not been hungup; in a word: if someone is still there.

Checking `session:ready()` is important, not only for knowing at which point the caller left your application, but also to avoid getting stuck in loops, waiting for the return of a database query, or (worse) for the caller to give input.

You can end up with your script stuck, and FreeSWITCH will not be able to free the associated resources (script and call data structures, and so on). Beware!



```
freeswitch@lxc111>
freeswitch@lxc111>
freeswitch@lxc111>
2017-06-12 20:28:36.483605 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@lab.opentelecomsolutions.com [fd64b987-f1d9-4b2c-ad6e-778b52f3e905]
2017-06-12 20:28:36.573581 [INFO] mod_dialplan_xml.c:637 Processing 1011 <1011>-
>9912 in context default
2017-06-12 20:28:36.573581 [NOTICE] sofia_media.c:92 Pre-Answer sofia/internal/1
011@lab.opentelecomsolutions.com!
2017-06-12 20:28:36.573581 [NOTICE] switch_cpp.cpp:685 Channel [sofia/internal/1
011@lab.opentelecomsolutions.com] has been answered
2017-06-12 20:28:36.873602 [INFO] switch_rtp.c:7189 Auto Changing audio port fro
m 192.168.1.66:7078 to 188.11.134.42:50141
2017-06-12 20:28:40.973581 [INFO] switch_channel.c:515 RECV DTMF 6:800
2017-06-12 20:28:44.533588 [INFO] switch_channel.c:515 RECV DTMF *:800
2017-06-12 20:28:50.773602 [INFO] switch_channel.c:515 RECV DTMF 1:800
2017-06-12 20:28:52.893602 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/888@conference.freeswitch.org [dfcffedf-5436-4343-a26d-b1470a2a069e]
2017-06-12 20:28:53.063601 [INFO] sofia.c:1279 sofia/internal/888@conference.fre
eswitch.org Update Callee ID to "888" <sip:888@conference.freeswitch.org>
2017-06-12 20:28:53.073581 [INFO] switch_core_media.c:7465 Activating VIDEO RTCP
 PORT 19573 interval 1000 mux -1
2017-06-12 20:28:53.073581 [NOTICE] sofia.c:8182 Channel [sofia/internal/888@con
ference.freeswitch.org] has been answered
2017-06-12 20:28:53.183604 [NOTICE] switch_vpx.c:402 VPX reset encoder picture f
rom 320x240 to 352x288 1024 BW
2017-06-12 20:28:54.093601 [NOTICE] switch_core_io.c:1202 Activating write resam
pler
2017-06-12 20:28:56.083601 [INFO] sofia.c:1279 sofia/internal/888@conference.fre
eswitch.org Update Callee ID to "FreeSWITCH Conference" <888>
2017-06-12 20:28:56.143601 [NOTICE] switch_core_io.c:1202 Activating write resam
pler
freeswitch@lxc111>
```

Summary

In this chapter, we've looked at the rationale for scripting in FreeSWITCH (is a complement to XML dialplan for anything complex), and the many languages that FreeSWITCH supports.

Then we used Lua, because of its popularity in the FreeSWITCH community, as an example scripting language.

We looked at all the objects FreeSWITCH makes available to Lua scripting, and how to use them, then we rewrote in Lua the XML demo IVR we looked at in the previous chapter, as an example of a real-world extensible application.

In the next chapter, we'll look at how to go the next level with XML dialplan, having a deep look into the concepts we barely scratched before.

Dialplan in Deep

We have now a more than basic understanding on how the XML configuration of FreeSWITCH and its powerful XML dialplan work.

It's time to go beyond that feeling of "I know how to do things, but not quite why they work that way".

Bear with me for this long and difficult chapter, and you will be rewarded with a good understanding of all the machineries that contribute to FreeSWITCH flexibility and versatility.

We'll see:

- How to combine conditions
- How to avoid dialplan pitfalls
- All about variables and their usage
- How to execute API commands from dialplan like we're on console
- Specific idiolects to slice, dice and rinse variables and strings
- Applications for interacting with live calls (barge and hotkeys)
- A cookbook style trove of useful dialplan recipes

Where are the previous Dialplan seasons?

We saw a lot in previous chapters, *Test Driving the Demo Configuration* and *XML Dialplan*. We visited all the foundational concepts, and the basic techniques that make FreeSWITCH ticking and spinning, happily chugging along your Real Time Communication traffic growth.

Don't cheat on yourself; if you've not been there, go now. It will take little reading time, even less if you're an experienced old FreeSWITCH hand, and it will ensure we're on the same page using the same concepts and terms, and will make you happy.

I will not repeat in this chapter the bulk of those chapters, so pretty please, read those chapters already.

XML Dialplan Recap

Following, in bullet format, how the dialplan works, its various components, their interaction, and all the important terms' definitions

- **incoming call** lands at the beginning of one of the **dialplan contexts**. This is the "routing phase" of the call, phase during which the TODO list is built
- to **which context** the call lands on is determined by modules' **configuration** (eg, which **SIPprofile** - IPaddress/port pair - the call comes from) or by **User Directory** (if call has been authorized)
- a **context is a list of extensions**, call enters at beginning, and proceed toward end of list. If call goes through the **end**, is **hungup**
- if an **extension** has been evaluated as "**matching**" (see later), then the call continue to the next extension depending on the value of the "**continue**" extension's **parameter**
- **each extension** is composed by **one or more conditions**, **each condition may** contain one or more **actions/anti-actions**
- the **characteristics** of the **call** (eg, dialed number), of the **configuration** (you can set variables in vars.xml and elsewhere) and of the **system** (eg, time of day) are all **available via variables**
- each **condition checks** some specified **variable** against some (regular) **expression**
- a **condition** is evaluated **true or false**, if the expression matches or not the specified variable(s)
- **if condition is true its actions** will be pushed in the **TODO** list stack, it its false its anti-actions will go in the TODO list
- in a **condition**, the parameter "**break-on**" determines if checks will **continue** with the next condition **or stops** here
- **actions** (and anti-actions) **are dialplan applications** (provided by almost each FreeSWITCH module, most of them provided by mod_dptools)
- **after** the **traversing** of **the dialplan** context, and the checking against the extensions, is completed then the **call enters its "execution phase"**
- in **"execution phase"all actions** that were pushed **in TODO** list stack are executed in order (FIFO)
- the separation between 'routing phase" and "execution phase" implies that you **cannot set and check variables** in the same dialplan pass (eg, not in next conditions of same extension, nor in conditions inside any following extensions). If you need that, use the **"inline" parameter** of the "set" application
- the **original incoming call** that enters the dialplan is called **"A-leg"**. FreeSWITCH-originated calls to be bridged with the A-leg, are called **"B-**

legs"

More about conditions

Conditions are the control elements in XML dialplan. Inside each extension there is at least one "condition" tag where the characteristics of the configuration/system/incoming call, represented by variables, are checked against a (regular) expression. If the condition is evaluated as true, the actions inside the condition are added to the call's TODO list. If evaluated as false, anti-actions (if present) are added to the TODO list. The general format is:

```
<extension name="is_looking_for_2900" continue="true">
  <condition field="destination_number" expression="^2900$" break="on-true">
    <action application="log" data="WARNING first TRUE"/>
    <anti-action application="log" data="ERR first FALSE"/>
  </condition>
  <condition field="destination_number" expression=".*">
    <action application="log" data="WARNING second TRUE"/>
    <action application="set" data="var01=value01" inline="true"/>
    <action application="set" data="var02=value02"/>
    <anti-action application="log" data="ERR second FALSE"/>
  </condition>
  <condition field="${var01}" expression="^value01$">
    <action application="log" data="WARNING third TRUE"/>
    <anti-action application="log" data="ERR third FALSE"/>
  </condition>
  <condition field="${var02}" expression="^value02$">
    <action application="log" data="WARNING fourth TRUE"/>
    <anti-action application="log" data="ERR fourth FALSE"/>
  </condition>
</extension>
<extension name="after_is_looking_for_2900">
  <condition field="destination_number" expression="^2900$">
    <action application="log" data="WARNING first of after TRUE"/>
    <anti-action application="log" data="ERR first of after FALSE"/>
  </condition>
</extension>
```

Let's have a quick look at this example, and please re-read previous chapters for more details.

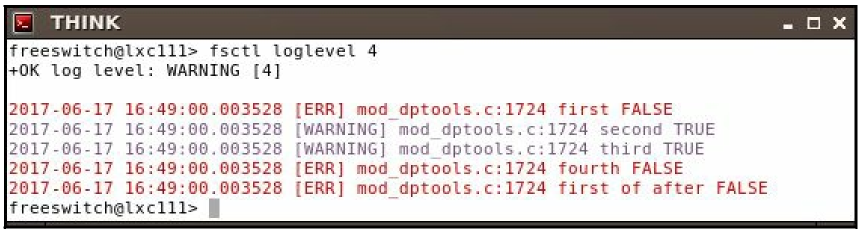
The first extension has a "continue" parameter set to true, so if this extension is found "matching" (that is, its first condition is evaluated as true), the next extension will be checked for a match. The default is the "continue" parameter set to false, so traversing the dialplan looking for matches will stop at the first matching extension. The first condition has a parameter "break" set to "on-true". So, if the expression is evaluated as true, the next condition(s) will not be checked. The default is to break "on-false" (eg, if the condition is false, do not check the next condition); other options include "always" and "never".

The first condition expression can be read as "a string exactly matching '2900'", because caret means "beginning-of-string" and dollar sign means "end-of-string".

The second condition will only be checked if the first condition is evaluated as false. The second condition's anti-action will never be sent to the TODO list (because the second condition expression ".*" will always match whatever the destination number is).

The third condition will be true if the destination number is anything different from 2900 (because the variable in expression is set by the second condition and the second condition is always true but is only tested if the first condition is false) while the fourth condition will never be true: this because only the variable checked by the third condition was set using the "inline" parameter. Your action of setting a variable takes place after dialplan traversing is completed, if you don't use the "inline" parameter of the "set" action.

The second extension is there just to show you that the first extension, because of the "continue" parameter, will not stop dialplan traversing even when it is matching. The following screenshot shows what happens when you call something other than "2900".



```
freeswitch@lxc111> fsctl loglevel 4
+OK log level: WARNING [4]

2017-06-17 16:49:00.003528 [ERR] mod_dptools.c:1724 first FALSE
2017-06-17 16:49:00.003528 [WARNING] mod_dptools.c:1724 second TRUE
2017-06-17 16:49:00.003528 [WARNING] mod_dptools.c:1724 third TRUE
2017-06-17 16:49:00.003528 [ERR] mod_dptools.c:1724 fourth FALSE
2017-06-17 16:49:00.003528 [ERR] mod_dptools.c:1724 first of after FALSE
freeswitch@lxc111>
```

regex operator (multiple variables and/or expressions)

You can go to great lengths with a regular expression checking a variable. I mean, you can have a regular expression so complex as to be indistinguishable from the line noise picked up by old modems. But there are cases where you want to check the relationship between two or more variables, or check multiple expressions which are not easily combined.

In these cases you use the "regex" operator to combine multiple fields and expressions into one only condition. You combine them with one only criterium:

- **all**: all expressions must be evaluated as true for the condition to be true (logical AND)
- **any**: at least one expression must be evaluated as true for the condition to be true (logical OR)
- **xor**: one and only one of the expressions must be evaluated as true for the condition to be true (e.g. if two expressions are evaluated as true, the condition is then false)

Let's see an OR on two different fields:

```
<extension name="Regex OR example">
  <condition regex="any">
    <!-- If one is true then add actions in TODO-->
    <regex field="caller_id_name" expression="Sara Adasi"/>
    <regex field="caller_id_number" expression="^1001$"/>
    <action application="log" data="WARNING At least one matched!"/>
    <!-- If *none* are true then add anti-actions -->
    <anti-action application="log"
      data="ERR Call is neither from Sara Adasi nor from 1001!"/>
  </condition>
</extension>
```

nested conditions

Conditions can be nested one inside each other. Nested conditions is hard, very very hard to get right. You may want to use a script, instead (yes, yes, you want to use a script). See previous chapter for Lua scripting.

When a (main) condition has nested conditions, **first its (main) expression is evaluated**, and its (main) actions are pushed in TODO list, **then nested conditions** are evaluated, and their actions pushed.

Each condition has an implied parameter "**require-nested**", that is by **default** set to "**true**".

When we have a condition with **require-nested** parameter set to "**true**", then **ALL** its **nested** conditions must evaluate to **true** (**AND** its **expression** must evaluate to **true**) for the condition to be evaluated to **true**.

If "**require-nested**" is set to **false**, then **only the expression** must evaluate to true for the condition to be true, whatever the nested conditions are.

Conditions are checked the usual way (respecting the "**break**" parameter), and actions inside conditions added to TODO list.

Let's see an **example extension** with nested conditions **you can play with**, checking all possible cases and corner cases.

The expression of the (main) condition checks if the destination number of the incoming call is "2901". If it is, then proceed to examine the nested conditions.

The first nested condition checks if the caller_id_number user's profile field (equivalent to the form ${caller_id_number}) evaluate to "1011".

The second nested condition checks if the ${caller_id_name} variable (equivalent to the form caller_id_name) evaluate to "Giovanni".

```
<extension name="nested_example">
    <condition field="destination_number" expression="^2901$" require-nested="false">
        <action application="log" data="ERR 00 CIDnum is ${caller_id_number} CIDname is ${cal
        <action application="set" data="var_01=N/A" inline="true"/>
        <action application="set" data="var_02=N/A" inline="true"/>
        <action application="set" data="var_03=N/A" inline="true"/>
        <action application="set" data="var_04=N/A" inline="true"/>
        <action application="set" data="var_05=N/A" inline="true"/>
        <action application="log" data="ERR 01 I'm before..."/>
        <action application="set" data="var_01=01" inline="true"/>
        <action application="log" data="ERR 02 I'm before  ${var_01} ${var_02} ${var_03} ${va
        <condition field="caller_id_number" expression="1011" break="on-false">
```

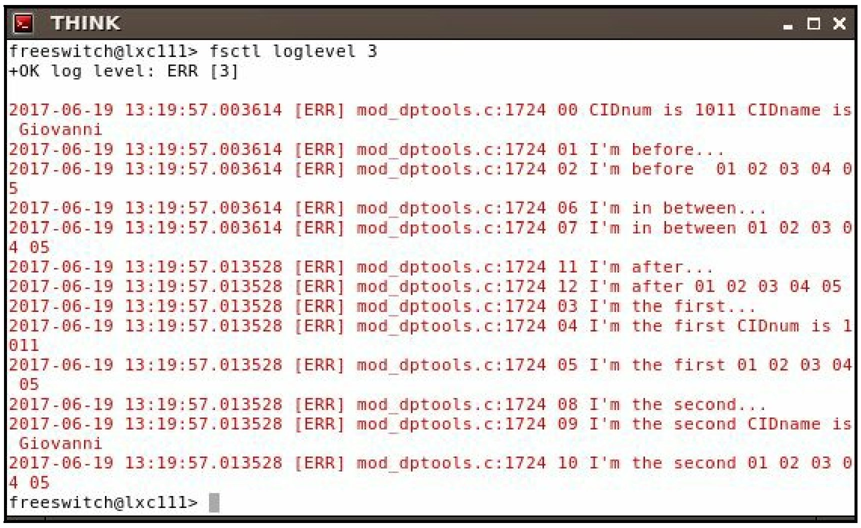
```
                <action application="log" data="ERR 03 I'm the first..."/>
                <action application="log" data="ERR 04 I'm the first CIDnum is ${caller_id_number
                <action application="set" data="var_02=02" inline="true"/>
                <action application="log" data="ERR 05 I'm the first ${var_01} ${var_02} ${var_0
            </condition>
            <action application="log" data="ERR 06 I'm in between..."/>
            <action application="set" data="var_03=03" inline="true"/>
            <action application="log" data="ERR 07 I'm in between ${var_01} ${var_02} ${var_03} $
            <condition field="${caller_id_name}" expression="Giovanni" break="on-false">
                <action application="log" data="ERR 08 I'm the second..."/>
                <action application="log" data="ERR 09 I'm the second CIDname is ${caller_id_name
                <action application="set" data="var_04=04" inline="true"/>
                <action application="log" data="ERR 10 I'm the second ${var_01} ${var_02} ${var_
            </condition>
            <action application="log" data="ERR 11 I'm after..."/>
            <action application="set" data="var_05=05" inline="true"/>
            <action application="log" data="ERR 12 I'm after ${var_01} ${var_02} ${var_03} ${var_
        </condition>
</extension>
<extension name="call_has_not_stopped_before_here">
    <condition field="destination_number" expression=".*">
        <action application="log" data="ERR NOT STOPPED BEFORE HERE"/>
    </condition>
</extension>
```

There are a lot of things you can change to test this example:

- Obviously, all the "expression": 2901, 1011, Giovanni
- In the first condition, require-nested can be set to "true" or "false"
- The "inline" parameter of the "set" actions can be set to "true" or "false"
- The "break" parameter of the nested conditions can be set to "on-false", "on-true", "never", "always"

Let's begin testing with "**require-nested**" set to "**false**", all the "set" actions with the "**inline**" attribute equal "**true**", and the "**break**" parameters of conditions set to "**on-false**".

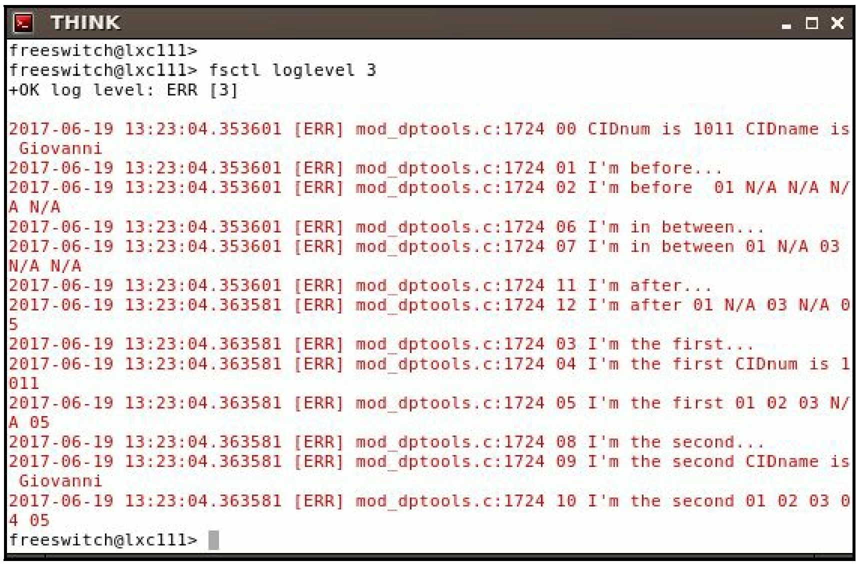
If we call **2901** with **caller_id_number=1011** and **caller_id_name=Giovanni** (eg, when all conditions are true) we get:



```
freeswitch@lxc111> fsctl loglevel 3
+OK log level: ERR [3]

2017-06-19 13:19:57.003614 [ERR] mod_dptools.c:1724 00 CIDnum is 1011 CIDname is
 Giovanni
2017-06-19 13:19:57.003614 [ERR] mod_dptools.c:1724 01 I'm before...
2017-06-19 13:19:57.003614 [ERR] mod_dptools.c:1724 02 I'm before  01 02 03 04 0
5
2017-06-19 13:19:57.003614 [ERR] mod_dptools.c:1724 06 I'm in between...
2017-06-19 13:19:57.003614 [ERR] mod_dptools.c:1724 07 I'm in between 01 02 03 0
4 05
2017-06-19 13:19:57.013528 [ERR] mod_dptools.c:1724 11 I'm after...
2017-06-19 13:19:57.013528 [ERR] mod_dptools.c:1724 12 I'm after 01 02 03 04 05
2017-06-19 13:19:57.013528 [ERR] mod_dptools.c:1724 03 I'm the first...
2017-06-19 13:19:57.013528 [ERR] mod_dptools.c:1724 04 I'm the first CIDnum is 1
011
2017-06-19 13:19:57.013528 [ERR] mod_dptools.c:1724 05 I'm the first 01 02 03 04
 05
2017-06-19 13:19:57.013528 [ERR] mod_dptools.c:1724 08 I'm the second...
2017-06-19 13:19:57.013528 [ERR] mod_dptools.c:1724 09 I'm the second CIDname is
 Giovanni
2017-06-19 13:19:57.013528 [ERR] mod_dptools.c:1724 10 I'm the second 01 02 03 0
4 05
freeswitch@lxc111>
```

All variables are there since the beginning.

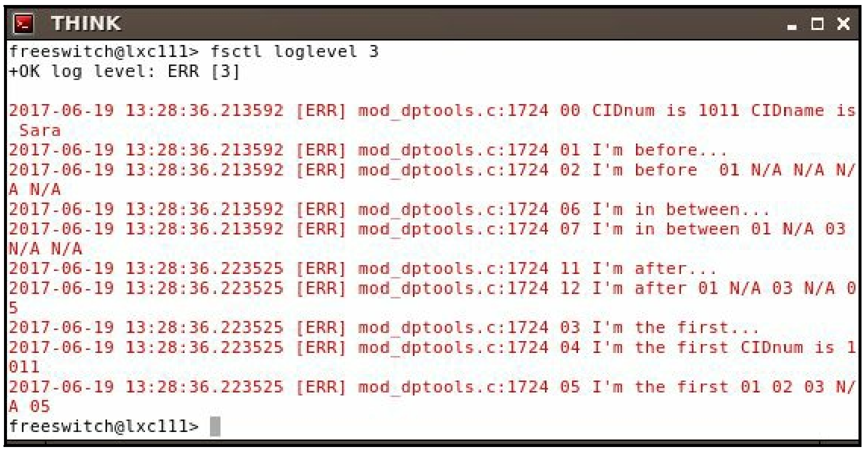
If we edit the extension and makes all the "inline" attributes equal "false", and reload the dialplan, then we get:



```
freeswitch@lxc111>
freeswitch@lxc111> fsctl loglevel 3
+OK log level: ERR [3]

2017-06-19 13:23:04.353601 [ERR] mod_dptools.c:1724 00 CIDnum is 1011 CIDname is
 Giovanni
2017-06-19 13:23:04.353601 [ERR] mod_dptools.c:1724 01 I'm before...
2017-06-19 13:23:04.353601 [ERR] mod_dptools.c:1724 02 I'm before  01 N/A N/A N/
A N/A
2017-06-19 13:23:04.353601 [ERR] mod_dptools.c:1724 06 I'm in between...
2017-06-19 13:23:04.353601 [ERR] mod_dptools.c:1724 07 I'm in between 01 N/A 03
N/A N/A
2017-06-19 13:23:04.353601 [ERR] mod_dptools.c:1724 11 I'm after...
2017-06-19 13:23:04.363581 [ERR] mod_dptools.c:1724 12 I'm after 01 N/A 03 N/A 0
5
2017-06-19 13:23:04.363581 [ERR] mod_dptools.c:1724 03 I'm the first...
2017-06-19 13:23:04.363581 [ERR] mod_dptools.c:1724 04 I'm the first CIDnum is 1
011
2017-06-19 13:23:04.363581 [ERR] mod_dptools.c:1724 05 I'm the first 01 02 03 N/
A 05
2017-06-19 13:23:04.363581 [ERR] mod_dptools.c:1724 08 I'm the second...
2017-06-19 13:23:04.363581 [ERR] mod_dptools.c:1724 09 I'm the second CIDname is
 Giovanni
2017-06-19 13:23:04.363581 [ERR] mod_dptools.c:1724 10 I'm the second 01 02 03 0
4 05
freeswitch@lxc111>
```

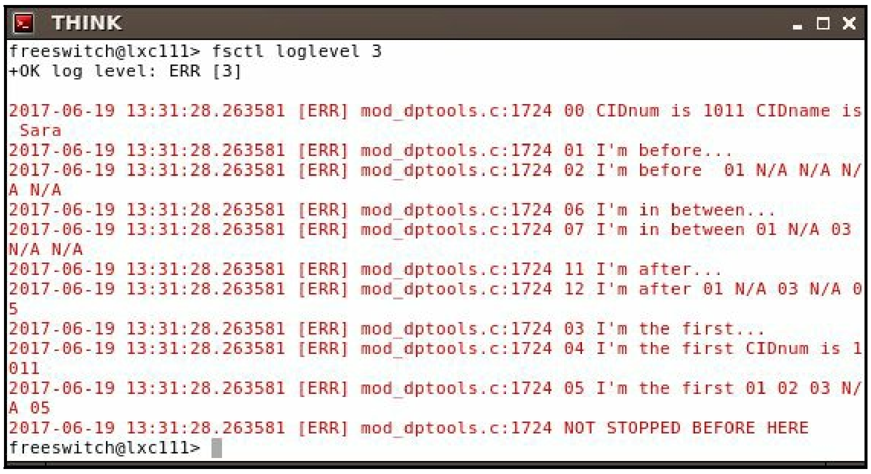
Variables are set in each step.

Let's further explore what happens if **only one nested condition** is evaluated to be **true**.
If we call **2901** with **caller_id_number=1011** and **caller_id_name=Sara** we get:



```
freeswitch@lxc111> fsctl loglevel 3
+OK log level: ERR [3]

2017-06-19 13:28:36.213592 [ERR] mod_dptools.c:1724 00 CIDnum is 1011 CIDname is
 Sara
2017-06-19 13:28:36.213592 [ERR] mod_dptools.c:1724 01 I'm before...
2017-06-19 13:28:36.213592 [ERR] mod_dptools.c:1724 02 I'm before  01 N/A N/A N/
A N/A
2017-06-19 13:28:36.213592 [ERR] mod_dptools.c:1724 06 I'm in between...
2017-06-19 13:28:36.213592 [ERR] mod_dptools.c:1724 07 I'm in between 01 N/A 03
N/A N/A
2017-06-19 13:28:36.223525 [ERR] mod_dptools.c:1724 11 I'm after...
2017-06-19 13:28:36.223525 [ERR] mod_dptools.c:1724 12 I'm after 01 N/A 03 N/A 0
5
2017-06-19 13:28:36.223525 [ERR] mod_dptools.c:1724 03 I'm the first...
2017-06-19 13:28:36.223525 [ERR] mod_dptools.c:1724 04 I'm the first CIDnum is 1
011
2017-06-19 13:28:36.223525 [ERR] mod_dptools.c:1724 05 I'm the first 01 02 03 N/
A 05
freeswitch@lxc111>
```

But if we edit the extension and make the "**require-nested**" parameter of the first condition to be **true**, then "reloadxml", and call again, we get:



```
freeswitch@lxc111> fsctl loglevel 3
+OK log level: ERR [3]

2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 00 CIDnum is 1011 CIDname is
 Sara
2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 01 I'm before...
2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 02 I'm before  01 N/A N/A N/
A N/A
2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 06 I'm in between...
2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 07 I'm in between 01 N/A 03
N/A N/A
2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 11 I'm after...
2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 12 I'm after 01 N/A 03 N/A 0
5
2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 03 I'm the first...
2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 04 I'm the first CIDnum is 1
011
2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 05 I'm the first 01 02 03 N/
A 05
2017-06-19 13:31:28.263581 [ERR] mod_dptools.c:1724 NOT STOPPED BEFORE HERE
freeswitch@lxc111>
```

Exactly like the one before, but this time the extension was not evaluated to be true (because not all its nested conditions were true), and the next extension has been evaluated. Note that also in this case, the actions belonging to conditions evaluated to true have been pushed in TODO list.

There are another zillion permutations you can check by changing the various parts of this example extension.

Bottom line: you may have the perfect use case for nested conditions, I would use a Lua script instead.

Pitfalls to avoid

The most frequent Dialplan error made by people new to FreeSWITCH (and by distracted old hands) is without any doubt **setting a variable and expecting to be able to check it** some lines down for the assigned value. NOOOOOOOOOOT.

In this book, I am repeating the concept very (too much?) often, trying to leave a mark in your collective consciousness, but let's do it again here:

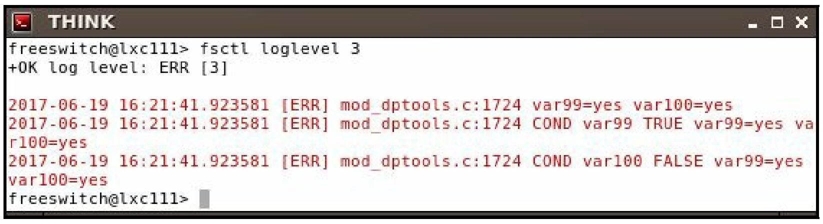
- Dialplan is first "traversed" by the call, checking all extensions, conditions, and so on for matches. Only after all the "traversing" is done and completed, are the actions in the TODO list executed.
- If there are condition matches, then actions will be put in the TODO list.
- Some of those actions can be "set" actions, actions that assign variables.
- So, we just added a variable setting action in the TODO list.
- We then check the variable. Is it STILL OLD (or nonexistent) VALUE.
- That's because we are still in the traversing phase, we have not yet executed the "set" action.
- Got it?
- If you want your "set" action to take place at the moment it is encountered during traversal, so the variable can be checked in the next line, then use the inline="true" parameter in the "set" action:

```
<action application="set" data="myvar=false" inline="true"/>
```

Also, when debugging assignments, you may be tempted to go the sure way, the "right" way, and put a printf, ahum, a "log" application, in dialplan, just before or after checking the variable you previously set.

```
<extension name="check_var">
  <condition field="destination_number" expression="^2902$">
     <action application="set" data="var99=yes" inline="true"/>
     <action application="set" data="var100=yes"/>
     <action application="log" data="ERR var99=${var99} var100=${var100}"/>
  </condition>
  <condition field="${var99}" expression="yes">
     <action application="log" data="ERR COND var99 TRUE var99=${var99} var100=${var100}"/>
     <anti-action application="log" data="ERR COND var99 FALSE var99=${var99} var100=${var10
  </condition>
  <condition field="${var100}" expression="yes">
     <action application="log" data="ERR COND var100 TRUE var99=${var99} var100=${var100}"/>
     <anti-action application="log" data="ERR COND var100 FALSE var99=${var99} var100=${var1
  </condition>
</extension>
```

And, you fall right in the pit: again, the "log" application is executed after the evaluation (traversal) phase. You need the "inline" attribute in the action "set" tag, to be able to check later in dialplan the variable you set. Try it, note the "**inline**" attribute, and see this mind-boggling output:



```
freeswitch@lxc111> fsctl loglevel 3
+OK log level: ERR [3]

2017-06-19 16:21:41.923581 [ERR] mod_dptools.c:1724 var99=yes var100=yes
2017-06-19 16:21:41.923581 [ERR] mod_dptools.c:1724 COND var99 TRUE var99=yes va
r100=yes
2017-06-19 16:21:41.923581 [ERR] mod_dptools.c:1724 COND var100 FALSE var99=yes
var100=yes
freeswitch@lxc111> 
```

Variables

Variables are everywhere in FreeSWITCH configuration and in FreeSWITCH working. Almost all things in FreeSWITCH happen because you set or change a variable. Also, all things that happen set and change FreeSWITCH variables.

An important aspect of channel variables (and of FreeSWITCH variables by and large) is that you use them not only to check the state of the system/call/user and so on,but you also use variables to affect the behavior of FreeSWITCH! (see later this chapter in "*setting variables and call setup*").

Here we will add and build on top of our previous expositions, passing in review various kind of variables you can use in dialplan conditions.

Global variables

Some global variables are set (and possibly calculated) automatically by FreeSWITCH at startup, like all the default directories in configuration, the IP addresses in use, NAT related infos, and so on.

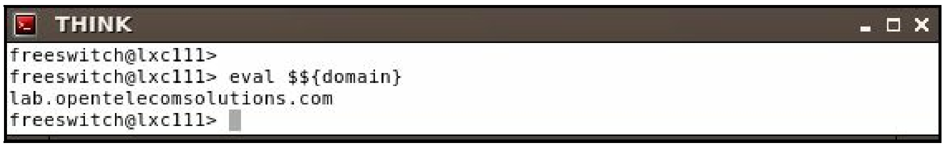
These are:

- hostname
- local_ip_v4
- local_mask_v4
- local_ip_v6
- switch_serial
- base_dir
- recordings_dir
- sound_prefix
- sounds_dir
- conf_dir
- log_dir
- run_dir
- db_dir
- mod_dir
- htdocs_dir
- script_dir
- temp_dir
- grammar_dir
- certs_dir
- storage_dir
- cache_dir
- core_uuid
- zrtp_enabled
- nat_public_addr
- nat_private_addr
- nat_type

Other global variables are set by the XML configuration pre-processor, and are usually found in /usr/local/freeswitch/conf/vars.xml. (see "pre-processor variables" in Chapter 3, *Test Driving the Example Configuration*).

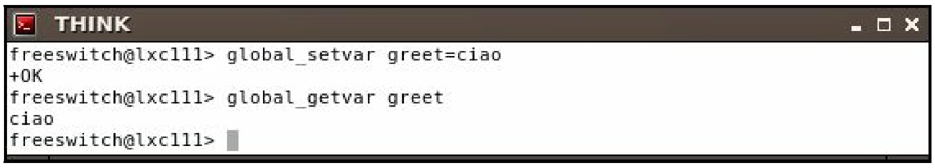
All global variables can be referenced everywhere in configuration (so, even in dialplan), and are used often to set defaults for modules.

You can reference global variables in dialplan (and elsewhere) by the construct $${globalvar} (note the double dollar sign). Also, you can check their value by using the "eval" command from the FreeSWITCH console:




```
freeswitch@lxc111>
freeswitch@lxc111> eval $${domain}
lab.opentelecomsolutions.com
freeswitch@lxc111>
```

You can also set and get global variables using the APIs (commands) global_setvar and global_getvar:




```
freeswitch@lxc111> global_setvar greet=ciao
+OK
freeswitch@lxc111> global_getvar greet
ciao
freeswitch@lxc111>
```

In dialplan, you can use them in expressions and in data (and everywhere else):

```
<extension name="check_global">
  <condition field="destination_number" expression="^2906$">
     <action application="set" data="var101=lab.opentelecomsolutions.com" inline="true"/>
  </condition>
  <condition field="${var101}" expression="$${domain_name}">
     <action application="log" data="ERR YES, var101 is $${domain_name}"/>
     <anti-action application="log" data="ERR NOPE, var101 is NOT $${domain_name}"/>
  </condition>
</extension>
```

Time of Day, Day of Week, Holidays

All conditions related to time at large can be expressed directly, with a specific syntax that checks the variable itself (no ${} construct here). You can combine more than one of them in the same condition, and use ranges:

```
<conditionwday="6"hour="8-12">
```

The available "time" variables are:

- **year** Calendar year, 0-9999
- **yday** Day of year, 1-366
- **mon** Month, 1-12 (Jan = 1, etc.)
- **mday** Day of month, 1-31
- **week** Week of year, 1-53
- **mweek** Week of month, 1-6
- **wday** Day of week, 1-7 (Sun = 1, Mon = 2, etc.) or "sun", "mon", "tue", etc.
- **hour** Hour, 0-23
- **minute** Minute (of the hour), 0-59
- **minute-of-day** Minute of the day, (1-1440) (midnight = 1, 1am = 60, noon = 720, etc.)
- **time-of-day** Time range formatted: hh:mm[:ss]-hh:mm[:ss] (seconds optional) Example: "08:00-17:00"
- **date-time** Date/time range formatted: YYYY-MM-DD hh:mm[:ss]~YYYY-MM-DD hh:mm[:ss] (seconds optional, note tilde between dates) Example: 2010-10-01 00:00:01~2010-10-15 23:59:59

The following extensions are adapted from the demo configuration dialplan.

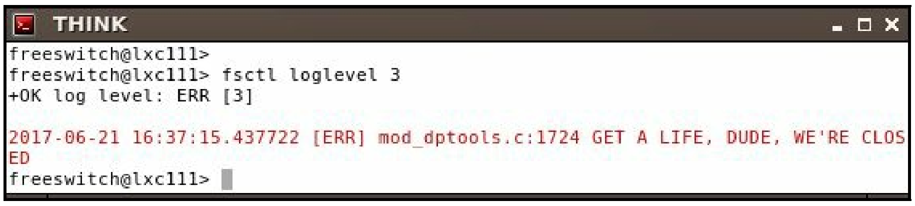
Please note the added **inline="true"**, as you need it if you want to test the variable ${open} in "its_open_or_not" last example's extension.

Also, be aware of a **common error** (at least: it got me, so it MUST be common): when you check for, let's say, hour="9-18" the **condition evaluates** to true starting from 09:00 up to 18:59. It evaluates **the numeric value** of the current hour. Ta-daaa: I blindly thought it would be 09:00 to 18:00. Don't ask me why.

<extension name="tod_example" continue="true" >

```
    <condition wday="2-6" hour="9-18">
      <action application="set" data="open=true"/>
    </condition>
  </extension>
```

```
         <extension name="holiday_example" continue="true">
      <condition mday="1" mon="1">
        <!-- new year's day -->
        <action application="set" data="open=false" inline="true"/>
      </condition>
      <condition wday="2" mweek="3" mon="1">
        <!-- martin luther king day is the 3rd monday in january -->
        <action application="set" data="open=false" inline="true"/>
      </condition>
      <condition wday="2" mweek="3" mon="2">
        <!-- president's day is the 3rd monday in february -->
        <action application="set" data="open=false" inline="true"/>
      </condition>
      <condition wday="2" mon="5" mday="25-31">
        <!-- memorial day is the last monday in may (the only monday between the 25th and the
        <action application="set" data="open=false" inline="true"/>
      </condition>
      <condition mday="4" mon="7">
        <!-- independence day -->
        <action application="set" data="open=false" inline="true"/>
      </condition>
      <condition wday="2" mday="1-7" mon="9">
        <!-- labor day is the 1st monday in september (the only monday between the 1st and th
        <action application="set" data="open=false" inline="true"/>
      </condition>
      <condition wday="2" mweek="2" mon="10">
        <!-- columbus day is the 2nd monday in october -->
        <action application="set" data="open=false" inline="true"/>
      </condition>
      <condition mday="11" mon="11">
        <!-- veteran's day -->
        <action application="set" data="open=false" inline="true"/>
      </condition>
      <condition wday="5-6" mweek="4" mon="11">
        <!-- thanksgiving is the 4th thursday in november and usually there's an extension fo
        <action application="set" data="open=false" inline="true"/>
      </condition>
      <condition mday="25" mon="12">
        <!-- Christmas -->
        <action application="set" data="open=false" inline="true"/>
      </condition>
    </extension>


<extension name="is_open_or_not">
  <condition field="destination_number" expression="^2904$"/>
  <condition field="${open}" expression="true">
     <action application="log" data="ERR YES, WE'RE OPEN FOR BUSINESS"/>
     <anti-action application="log" data="ERR GET A LIFE, DUDE, WE'RE CLOSED"/>
  </condition>
</extension>
```



```
freeswitch@lxc111>
freeswitch@lxc111> fsctl loglevel 3
+OK log level: ERR [3]

2017-06-21 16:37:15.437722 [ERR] mod_dptools.c:1724 GET A LIFE, DUDE, WE'RE CLOS
ED
freeswitch@lxc111>
```

Channel Variables

For an introduction to channel variables, please see the section "Channel Variables" in Chapter 6, *XML Dialplan* .

There are various pages in official documentation about channel variables, and the one where we are trying to centralize all info is:

https://freeswitch.org/confluence/display/FREESWITCH/Channel+Variables

Below is a list of some of them, grouped "thematically" for your future reference (for details please search https://freeswitch.org/confluence).

Also, please note that **if you need a variable** (an information on some aspect of the channel, call, system, user, or a way to tweak a parameter) is **almost certain** that **it exists**: if the information or parameter exists, it is probably available as a variable. Search our documentation site, both for the specific variable and for the possibly related modules, search the mailing list archives, google for it, or ask the community (mailing list, IRC, HipChat, etc).

- CDR related
  - process_cdr
  - accountcode
  - hold_events
  - hangup_complete_with_xml
  - skip_cdr_causes
  - transfer_to
- Hangup Causes
  - bridge_hangup_cause
  - disable_q850_reason
  - hangup_cause
  - hangup_cause_q850
  - sip_hangup_disposition
  - proto_specific_hangup_cause
  - last_bridge_hangup_cause
  - last_bridge_proto_specific_hangup_cause
- DTMF Related
  - pass_rfc2833
  - dtmf_type
  - drop_dtmf
  - drop_dtmf_masking_digits

    - drop_dtmf_masking_file
- Media Handling
    - monitor_early_media_fail
    - monitor_early_media_ring
- Timeout Related
    - call_timeout
    - leg_timeout
    - originate_continue_on_timeout
    - park_timeout
- Music On Hold Related
    - hold_music
    - temp_hold_music
- Locale Related
    - default_language
    - timezone
    - tod_tz_offset
- Bridge Related
    - api_after_bridge
    - auto_hunt
    - bridge_early_media
    - bridge_filter_dtmf
    - bridge_terminate_key
    - continue_on_fail
    - transfer_on_fail
    - enable_file_write_buffering
    - failure_causes
    - force_transfer_context
    - force_transfer_dialplan
    - hangup_after_bridge
    - hold_hangup_xfer_exten
    - last_bridge_to
    - loopback_bowout_on_execute
    - loopback_export
    - outbound_redirect_fatal
    - originate_timeout
    - park_after_bridge
    - signal_bond
    - no_throttle_limits
    - rtp_jitter_buffer_during_bridge
    - uuid_bridge_continue_on_cancel
- Conference Related

    - conference_auto_outcall_announce
    - conference_auto_outcall_caller_id_name
    - conference_auto_outcall_caller_id_number
    - conference_auto_outcall_flags
    - conference_auto_outcall_prefix
    - conference_auto_outcall_timeout
    - conference_auto_outcall_maxwait
    - conference_controls
    - conference_enter_sound
    - conference_last_matching_digits
    - last_transferred_conference
    - conference_member_id
    - conference_uuid
    - hangup_after_conference
- Code Execution Related
    - api_hangup_hook
    - bridge_pre_execute_aleg_app
    - bridge_pre_execute_aleg_data
    - bridge_pre_execute_bleg_app
    - bridge_pre_execute_bleg_data
    - exec_after_bridge_app
    - exec_after_bridge_arg
    - origination_nested_vars
    - The execute_on family
    - The api_on family
    - failed_xml_cdr_prefix
    - fail_on_single_reject
    - intercept_unbridged_only
    - intercept_unanswered_only
    - session_in_hangup_hook
- Caller ID Related
    - caller_id_name
    - caller_id_number
    - effective_caller_id_name
    - effective_caller_id_number
    - sip_cid_type
    - effective_sip_cid_in_1xx
- Callee ID Related
    - initial_callee_id_name
    - origination_callee_id_name
    - origination_callee_id_number

- Call Recording Related
    - Audio File Metadata
    - record_fill_cng
    - RECORD_HANGUP_ON_ERROR
    - RECORD_DISCARDED
    - record_post_process_exec_api
    - record_post_process_exec_app
    - record_restart_limit_on_dtmf
    - record_sample_rate
    - record_waste_resources
    - recording_follow_transfer
    - Example
- Codec Related
    - absolute_codec_string
    - codec_string
    - inherit_codec
    - media_mix_inbound_outbound_codecs
    - read_codec
    - write_codec
    - passthru_ptime_mismatch
    - sip_renegotiate_codec_on_reinvite
    - conference_enforce_security
    - suppress-cng
- IVR related
    - ivr_menu_terminator
    - detect_speech_result
- SIP related variables
    - rtp_disable_hold
    - sip_acl_authed_by
    - sip_acl_token
    - sip_copy_multipart
    - sip_invite_params
    - sip_invite_domain
    - sip_from_display
    - sip_invite_from_params
    - sip_invite_to_params
    - sip_invite_contact_params
    - sip_invite_tel_params
    - sip_network_destination
    - sip_auth_username
    - sip_auth_password

  - sip_auto_simplify
  - sip_callee_id_name
  - rtp_force_audio_fmtp
  - sip_invite_req_uri
  - sip_invite_record_route
  - sip_invite_route_uri
  - sip_invite_full_from
  - sip_invite_full_to
  - sip_handle_full_from
  - sip_handle_full_to
  - sip_force_full_from
  - sip_force_full_to
  - sip_recover_contact
  - sip_recover_via
  - sip_invite_from_uri
  - sip_invite_to_uri
  - sip_ignore_reinvites
  - sip_has_crypto
  - rtp_sdes_suites
  - rtp_secure_media
  - rtp_secure_media _inbound
  - rtp_secure_media _outbound
  - rtp_secure_media_suites
  - sip_wait_for_aleg_ack
  - timer_name
  - ignore_display_updates
  - deny_refer_requests
- SDP Manipulation
  - rtp_append_audio_sdp
  - sdp_m_per_ptime
  - switch_r_sdp
  - switch_l_sdp
  - switch_m_sdp
  - verbose_sdp
  - sip_local_sdp_str
  - sip_recovery_break_rfc
  - sip_mirror_remote_audio_codec_payload
  - sip_enable_soa
  - sdp_secure_savp_only
- FIFO related variables
  - fifo_bridged

    - fifo_caller_consumer_import
    - fifo_consumer_caller_import
    - fifo_manual_bridged
    - fifo_position
    - fifo_role
    - transfer_after_bridge
- Playback related variables
    - playback_terminators
    - sound_prefix
    - playback_terminator_used
    - playback_ms
    - playback_samples
    - playback_last_offset_pos
    - playback_sleep_val
    - playback_delimiter
    - sleep_eat_digits
    - playback_timeout_sec
- Originate related variables
    - execute_on_originate
    - leg_delay_start
    - originate_disposition
    - originate_retries
    - originate_retry_sleep_ms
    - originate_timeout
    - originating_leg_uuid
    - origination_channel_name
    - origination_caller_id_name
    - origination_caller_id_number
    - origination_cancel_key
    - origination_privacy
    - origination_uuid
    - originator
    - originator_codec
- RTP/media related variables
    - bypass_media
    - bypass_media_after_bridge
    - bypass_keep_codec
    - proxy_media
    - rtp_autoflush
    - rtp_autoflush_during_bridge
    - disable_rtp_auto_adjust

    - progress_timeout
    - bridge_answer_timeout
    - ignore_early_media
    - ringback
    - instant_ringback
    - transfer_ringback
    - rtp_disable_hold
    - rtp_negotiate_near_match
- RTCP Related
    - rtcp_packet_count
    - rtcp_octet_count
    - N/Artcp_audio_interval_msec
    - rtcp_mux
    - rtp_assume_rtcp
- Camp-on related variables
    - campon
    - campon_retries
    - campon_timeout
    - campon_sleep
    - campon_fallback_exten
    - campon_fallback_dialplan
    - campon_fallback_context
    - campon_hold_music
    - campon_stop_key
    - campon_announce_sound
- Answer confirmation variables
    - group_confirm_file
    - group_confirm_key
    - group_confirm_cancel_timeout
- Voicemail Related Variables
    - voicemail_alternate_greet_id
    - voicemail_greeting_number
    - vm_message_ext
    - vm_cc
    - skip_greeting
    - skip_instructions
    - voicemail_authorized

Channel Variables and Field shortcuts

Some (not all!) channel variables can be used in "field" with or without the ${} construct.

- **username** Directory User who has originated this call
- **dialplan** Which kind of dialplan is in use, almost always "XML"
- **caller_id_name**
- **caller_id_number**
- **callee_id_name**
- **callee_id_number**
- **network_addr** address from which this call has been originated
- **ani** Automatic Number Identification (actual caller number)
- **aniii** Automatic Number Identification 2 (actual caller number version 2)
- **rdnis** Redirected Dialed Number Information Service, previous destination_number of a transferred call
- **destination_number**
- **source** what module was used for the call, e.g. for a SIP call, mod_sofia
- **uuid** Universal Unique Identifier that denotes this call
- **context** which context inside dialplan

```
<extension name="check_sip" continue="true">
      <condition field="source" expression="mod_sofia">
            <action application="log" data="ERR YES, THIS CALL IS
A SIP CALL"/>
            <anti-action application="log" data="ERR NOPE, NOT A
SIP CALL"/>
      </condition>
   </extension>
```

Channel Variables and Caller Profile fields

Those are variables that are set when the call is authenticated, typically when is originated by a registered phone. Some of the Caller Profile fields are filled by the values coming from the User Directory. Please check Chapter 4, *User Directory, SIP, and Verto*.

You set a variable in User Directory:

```
<user id="1011">
  <variables>
    <variable name="myvariable" value="1234"/>
  </variables></user>
```

And then you use it in dialplan:

```
 <extension name="check_caller_profile" continue="true">
      <condition field="${myvariable}" expression="1234">
            <action application="log" data="ERR myvariable exists
and is 1234"/>
            <anti-action application="log" data="ERR myvariable
has NOT been set or is NOT 1234"/>
      </condition>
  </extension>
```

Channel Variables and call setup

An important aspect of channel variables (and of FreeSWITCH variables by and large) is that you use them not only to check the state of the system/call/user and so on, but you also use **variables** to **affect the behavior of FreeSWITCH**!

Most of what FreeSWITCH will do with a call is set by its configuration, and can be overridden at will on a call by call basis.

Let's see the most basic characteristic of a SIP or Verto call: which codec FreeSWITCH will propose to a remote peer.

You set a default list in the /usr/local/freeswitch/conf/vars.xml file line:

```
<X-PRE-PROCESS cmd="set" data="outbound_codec_prefs=OPUS,G722,PCMU,PCMA,VP8"/>
```

This line will be used by pre-processor to set the global variable $${outbound_codec_prefs}, which will then be used by SIP and Verto profiles when negotiating media with remote party (callee) during the setup of FreeSWITCH originated calls.

So, if you use a simple dialplan extension, that list will be used verbatim:

```
    <extension name="bridge_test_01">
      <condition field="destination_number" expression="^2908$">
        <action application="bridge"
                 data="sofia/internal/888@conference.freeswitch.org"/>
      </condition>
    </extension>
```

Extension "bridge_test_01" will use the default outbound codec preferences list set in vars.xml.

setting channel variables in dialstrings

You can alter this behavior by setting variables in dialstring, using the construct "{variable=value,variable01=value02}dialstring".

In case you want to override outbound_codec_prefs you need to set the "codec-string" variable on the dialstring (note the usage of single quotes to escape value strings containing commas. Comma would delimit variables-values pairs):

```
    <extension name="bridge_test_02">
       <condition field="destination_number" expression="^2908$">
         <action application="bridge"  data="{codec_string='PCMA,PCMU,VP8'}user/1011"/>
       </condition>
     </extension>
```

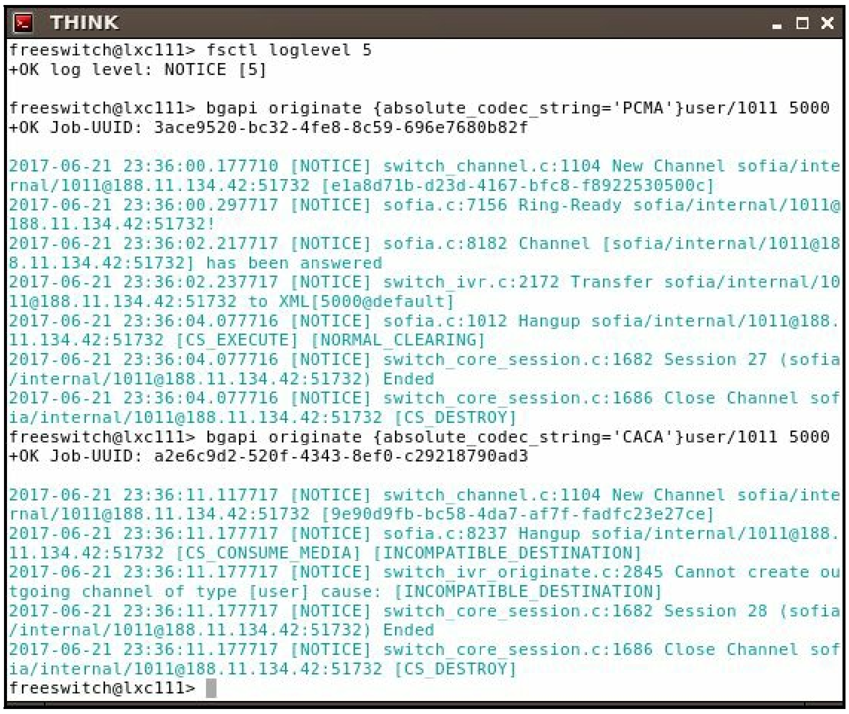
If instead of changing the preferences list, you want to force FreeSWITCH to use exactly a codec list, and nothing else, you set the "absolute-codec-string":

```
<extension name="bridge_test_02">
       <condition field="destination_number" expression="^2909$">
         <action application="bridge"  data="{absolute_codec_string='PCMA,PCMU,VP8'}user/1011'
       </condition>
 </extension>
```

You can use the same technique from the FreeSWITCH console command line:

```
bgapi originate {absolute_codec_string='PCMA,PCMU,VP8'}user/1011 5000
```

In the following example, we first set the codec to PCMA in dialstring, and the call succeeds. Then we try again, setting the codec to something obviously not compatible with callee, and the call fails.



```
freeswitch@lxc111> fsctl loglevel 5
+OK log level: NOTICE [5]

freeswitch@lxc111> bgapi originate {absolute_codec_string='PCMA'}user/1011 5000
+OK Job-UUID: 3ace9520-bc32-4fe8-8c59-696e7680b82f

2017-06-21 23:36:00.177710 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@188.11.134.42:51732 [e1a8d71b-d23d-4167-bfc8-f8922530500c]
2017-06-21 23:36:00.297717 [NOTICE] sofia.c:7156 Ring-Ready sofia/internal/1011@
188.11.134.42:51732!
2017-06-21 23:36:02.217717 [NOTICE] sofia.c:8182 Channel [sofia/internal/1011@18
8.11.134.42:51732] has been answered
2017-06-21 23:36:02.237717 [NOTICE] switch_ivr.c:2172 Transfer sofia/internal/10
11@188.11.134.42:51732 to XML[5000@default]
2017-06-21 23:36:04.077716 [NOTICE] sofia.c:1012 Hangup sofia/internal/1011@188.
11.134.42:51732 [CS_EXECUTE] [NORMAL_CLEARING]
2017-06-21 23:36:04.077716 [NOTICE] switch_core_session.c:1682 Session 27 (sofia
/internal/1011@188.11.134.42:51732) Ended
2017-06-21 23:36:04.077716 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@188.11.134.42:51732 [CS_DESTROY]
freeswitch@lxc111> bgapi originate {absolute_codec_string='CACA'}user/1011 5000
+OK Job-UUID: a2e6c9d2-520f-4343-8ef0-c29218790ad3

2017-06-21 23:36:11.117717 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@188.11.134.42:51732 [9e90d9fb-bc58-4da7-af7f-fadfc23e27ce]
2017-06-21 23:36:11.177717 [NOTICE] sofia.c:8237 Hangup sofia/internal/1011@188.
11.134.42:51732 [CS_CONSUME_MEDIA] [INCOMPATIBLE_DESTINATION]
2017-06-21 23:36:11.177717 [NOTICE] switch_ivr_originate.c:2845 Cannot create ou
tgoing channel of type [user] cause: [INCOMPATIBLE_DESTINATION]
2017-06-21 23:36:11.177717 [NOTICE] switch_core_session.c:1682 Session 28 (sofia
/internal/1011@188.11.134.42:51732) Ended
2017-06-21 23:36:11.177717 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@188.11.134.42:51732 [CS_DESTROY]
freeswitch@lxc111>
```

setting channel variables on B-legs: export and nolocal

You may want to set or override variables in dialplan, earlier, instead of resorting to last moment dialstring.

When you want to set a variable in both the current (A-leg) call and any future originated B-legs (typically originated later with one or more bridge applications) you "**export**" a variable (note: **no single quotes to escape the variable's value string**):

```
<extension name="bridge_test_03">
     <condition field="destination_number" expression="^2910$">
       <action application="export" data="absolute_codec_string=PCMA,PCMU"/>
       <action application="bridge" data="user/1011"/>
     </condition>
</extension>
```

In this specific case, we made a subtle impropriety: **we exported the variable to both A-leg and any future B-legs**. But this has no effect on the current A-leg, because A-leg has already been established (e.g. there will not be codec negotiation between FreeSWITCH and the caller. Maybe this can affect future re-invites, and maybe we do not want this to happen).

So, this extension would be better written in a way that **exports** the variable **only to future B-legs**, using the **nolocal** parameter:

```
<extension name="bridge_test_04">
      <condition field="destination_number" expression="^2911$">
        <action application="export" data="nolocal:absolute_codec_string=PCMA,PCMU"/>
        <action application="bridge" data="user/1011"/>
      </condition>
 </extension>
```

exporting variables to SIP custom (X-) Headers

Sometimes you need to pass variables to other SIP entities in your own or in remote networks. This is typically sought for accounting or authorization or operational purposes.

You can add arbitrary headers to FreeSWITCH generated SIP messages by setting or exporting channel variables which names begin with "sip_h_". If you are setting nonstandard SIP headers (you better not use standard headers) you can follow the popular convention of naming the header "X-Something", so your variable would be "sip_h_X-Something"

```
<extension name="bridge_test_05">
      <condition field="destination_number" expression="^2912$">
        <action application="set" data="sip_h_X-AccountCode=4321"/>
        <action application="bridge" data="sofia/gateway/gw03/2125551212"/>
      </condition>
 </extension>
```

This extension will originate a B-leg. This new INVITE SIP message will contain an header like:

```
X-AccountCode=4321
```

The gateway will then be able to read the custom header and act accordingly. SIP proxies have their own functions for reading a custom X-header, while other FreeSWITCHes will find it in the channel variable ${sip_h_X-AccountCode}.

API commands from Dialplan

All commands (APIs) you can execute from cli in the FreeSWITCH console are also available to be used in dialplan.

Many API commands are provided by mod_commands (search for it in http://freeswitch.org/confluence), but usually all modules add their own API commands you can execute from the console command line. From the console, type "help" for a quick overview.

The general form to execute an API command from dialplan is:

```
${api_command(argument01 argument02)}
```

You may use the "set" action, and assign to a channel variable (that you maybe will not use) the string returned from the console command line.

API commands' arguments are between parenthesis and separated by spaces; if there are no arguments, use empty parenthesis.

The third action line of this example extension will try to unload mod_verto from FreeSWITCH, and the fifth action line will give us the SIP dialstring to call the caller (if the call was originated by a registered phone):

```
<extension name="API">
      <condition field="destination_number" expression="^2913$">
        <action application="log" data="ERR ${status()}"/>
        <action application="log" data="ERR ${sofia(status profile internal reg)}"/>
        <action application="set" data="api_result=${unload(mod_verto)}"/>
        <action application="log" data="ERR ${module_exists(mod_verto)}"/>
        <action application="log" data="ERR ${sofia_contact(${username}@$${domain_name})}"/>
      </condition>
</extension>
```

FreeSWITCH string interpreter functions

There are couple of constructs we can use to enhance the expressive power of FreeSWITCH configuration. They are borrowed from programming languages, and are particularly handy when, for the sake of efficiency, we don't want to use a scripting language like Lua or Perl.

cond, C-like "expr ? truevalue : falsevalue"

The "**cond**" construct will be **substituted** by its **truevalue if expr** is evaluated to **true**, or by its falsevalue in the other case.

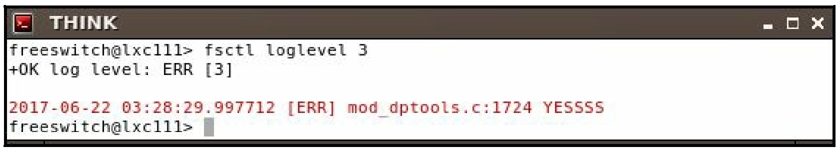
The general format of the cond function is (**note the spaces around question mark and around colon**):

```
${cond(<expr> ? <truevalue> : <falsevalue>)}
```

Let's see an example extension using "cond" construct:

```
<extension name="COND">
      <condition field="destination_number" expression="^2914$">
        <action application="log" data="ERR ${cond(${username} == 1011 ? YESSSS : NOOOOT)}"/>
      </condition>
</extension>
```

Calling 2914 from a phone registered as user 1011 will output:



```
freeswitch@lxc111> fsctl loglevel 3
+OK log level: ERR [3]

2017-06-22 03:28:29.997712 [ERR] mod_dptools.c:1724 YESSSS
freeswitch@lxc111> 
```

Comparison operators are:

- == equality
- != not equal
- > greater than
- >= greater than or equal to
- < less than
- <= less than or equal to

You can compare strings with strings and numbers with numbers, but if you compare a string to a number, they will be compared as strlen(string) and the number.

variables' substrings

You can select a portion of a variable's value, (just like a substr function in many programming languages) by wrapping the variable in ${var:offset:length} tags. The arguments are:

- var: A string variable. It can be a literal string or a variable such as ${caller_id}.
- offset: The location to start copying data. The value 0 indicates the first character.
- length: How many characters to select. It is optional and if omitted, the remainder of the string is copied.

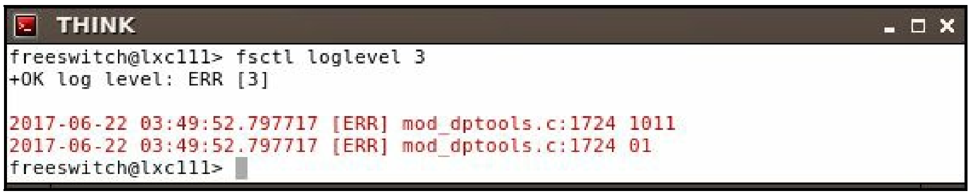
Some examples of the arguments are as follows:

```
set data="varname=1234567890"
${varname:offset:length}
${varname:0:1}  // 1
${varname:1}    // 234567890
${varname:-4}   // 7890
${varname:-4:2} // 78
${varname:4:2}  // 56
```

An extension showing a variable and its substring:

```
<extension name="SUBSTR">
      <condition field="destination_number" expression="^2915$">
        <action application="log" data="ERR ${username}"/>
        <action application="log" data="ERR ${username:1:2}"/>
      </condition>
</extension>
```

Calling 2915 from a phone registered as user 1011 will output:



```
freeswitch@lxc111> fsctl loglevel 3
+OK log level: ERR [3]

2017-06-22 03:49:52.797717 [ERR] mod_dptools.c:1724 1011
2017-06-22 03:49:52.797717 [ERR] mod_dptools.c:1724 01
freeswitch@lxc111>
```

db and hash: simple key/value stores

You can arbitrarily insert, delete, select, and update values from the internal FreeSWITCH database.

db will work on persistent tables, be they SQLite files in /usr/local/freeswitch/db, or ODBC or PGSQL remote tables.

hash will work instead on memory resident simple hash tables, extremely fast and efficient, not persistent after a FreeSWITCH restart.

The general format for database commands is:

```
${db(command/table/key/value)}
```

or

```
${hash(command/table/key/value)}
```

And you have also the corresponding dialplan application "db" and "hash".

Commands can be insert, select, or delete, followed by the table, followed by a key and (when appropriate) a value.

```
<action application="hash" data="insert/${domain_name}-last_dial/${caller_id_number}/${destir
```

This action (taken from "global" extension in demo dialplan) will insert into a table named from the SIP domain in use the caller_id that originated the call, and the destination number. This can easily be used afterwards to build a "redial" extension.

You would get at the last dialed number by the caller with an extension like this:

```
<extension name="LASTDIALED">
      <condition field="destination_number" expression="^2916$">
        <action application="log" data="ERR ${hash(select/${domain_name}-last_dial/${caller_i
      </condition>
</extension>
```

"Hotkeys", Listening, Barging

Nobody is counting the number of applications you can use in Dialplan, but I would bet there are hundreds. Anyway, I can count: "1, 2, 3, 4, 5, MANY". OK, there are MANY of them.

Most dialplan applications are provided by the aptly named mod_dptools (Dialplan Tools), but almost all other FreeSWITCH modules add their own applications to the heap of what is available in dialplan.

Let's have a quick look at a couple of dialplan applications that allow for going beyond the simple calling and bridging.

bind_meta_*, DTMFs to actions outside menus

bind_meta_app: This command binds an application to the specified call leg(s). During a bridged call, the DTMF sequence on the bound call leg will trigger the execution of the application. The call leg that is not bound will not hear the DTMF sequence being dialed. You can only bind a single digit, and the binding is usually proceeded with a * key press. As an example, let's say you want to set *2 to begin a call recording.

When the calling party presses *2, the recording will begin:

```
<action application="bind_meta_app" data="2 a s
record_session::recording.wav"/>
<action application="bridge"
data="sofia/sipprovider/+14158867900">
```

Note that unless otherwise specified, bind_meta_app will use * as the "meta key". Set the **bind_meta_key** channel variable to a different value to modify this behavior. For example, to use # instead of * you can do this:

```
<action application="set" data="bind_meta_key=#"/>
```

Notice the format of the bind_meta_app parameters:

```
<action application="bind_meta_app" data="KEY LISTEN_TO FLAGS APPLICATION[::PARAMETERS]"/>
```

Explanation of parameters

- KEY is the button you want to respond to after the * button is pressed. If you wanted to respond to *1, you would put 1 in place of KEY. You are limited to a single digit in the range 0-9 while * or # will be translated to 0.
- LISTEN_TO specifies which call leg(s) to listen to for digits. Acceptable parameters are '''a''', '''b''' or '''ab'''.
- FLAGS modifies the behavior. The following flags are available:
  - a - Respond on A leg
  - b - Respond on B leg
  - - Respond on opposite leg
  - s - Respond on same leg
  - i - Execute inline (see below)
  - 1 - Unbind this meta_app after it is used one time
- APPLICATION specifies the application to execute.
- PARAMETERS specify the arguments to provide to the APPLICATION. You

must put :: after the APPLICATION for these arguments to be parsed properly.

Once bound to a call leg, the application binding will persist for the lifetime of the call leg.

eavesdrop (call barge)

This dialplan application allows for listening and interacting with other channels/calls.

It's a very flexible application; search on http://freeswitch.org/confluence for all the details and features.

The format of invocation is:

```
eavesdrop UUID|all
```

A couple of example dialplan actions would be:

```
<action application="eavesdrop" data="${db(select/spymap/$1)}"/>
<action application="eavesdrop" data"all"/>
```

The first one, adapted from demo dialplan, assumes that elsewhere we inserted into a persistent table (see "db and hash" before in this chapter) the UUID of the ongoing call originated by the user we get in the "$1" regular expression register. We then get the UUID value from the db table, and execute eavesdrop on it.

The second one will connect in turn to all ongoing calls.

eavesdrop DTMF commands

- 2 to speak with the uuid
- 1 to speak with the other half
- 3 to engage a three way
- 0 to restore eavesdrop.
- * to next channel.

If we are listening to an UUID (and not to "all" channels) then "*" will terminate the eavesdrop.

Dialplan Cookbook

We present here a few scenarios that you may need to refer to from time-to-time because they are relatively common. The examples presented in this section are in the mold of the traditional *cookbook* full of *recipes* for the reader to try. Feel free to use and modify these recipes in your custom Dialplans.

Match by IP address and call a number

In the following example, the particular extension will be selected only if the IP address of the calling endpoint is `192.168.1.1`. In the second condition, the dialed number is extracted in variable `$1` and put in the data of the `bridge` application, in order to dial out to IP address `192.168.2.2`.

```
<extension name="Test1">
  <condition field="network_addr"
    expression="^192\.168\.1\.1$"/>
  <condition field="destination_number" expression="^(\d+)$">
    <action application="bridge"
      data="sofia/profilename/$1@192.168.2.2"/>
  </condition>
</extension>
```

The first condition field is terminated by a slash. The last condition field that contains the `action` tag is terminated by a regular `</condition>` tag. Also, note that the preceding example is not the same as this example:

```
<extension name="Test1Wrong">
  <condition field="destination_number" expression="^(\d+)$"/>
  <condition field="network_addr"
    expression="^192\.168\.1\.1$">
    <action application="bridge"
      data="sofia/profilename/$1@192.168.2.2"/>
  </condition>
</extension>
```

The `Test1Wrong` example will not route the call properly because the variable `$1` will not have any value, since the destination number was matched in a different condition, field.

You can also solve the `Test1Wrong` example by setting a variable in the first condition which you then use inside the second condition's action:

```
<extension name="Test1.2">
  <condition field="destination_number" expression="^(\d+)$">
    <action application="set" data="dialed_number=$1"/>
  </condition>
  <condition field="network_addr"
    expression="^192\.168\.1\.1$">
    <action application="bridge"
      data="sofia/profile/${dialed_number}@192.168.2.2"/>
  </condition>
</extension>
```

You cannot use a variable set inside an extension for further conditions/matches as the extension is evaluated when the action is called.

If you need to do different actions based on a variable set inside an extension, you need to either use `execute_extension` to transfer the call for the variable to be set, or use inline processing. (See the section *Inline execution* earlier in this chapter.)

Match an IP address and Caller ID

In this example, we need to match a called number beginning with the prefix `1` and match the incoming IP address at the same time.

```
<extension name="Test2">
  <condition field="network_addr"
    expression="^192\.168\.1\.1$"/>
  <condition field="destination_number" expression="^1(\d+)$">
    <action application="bridge"
      data="sofia/profilename/$0@192.168.2.2"/>
  </condition>
</extension>
```

Here, although we match with the rule `^1(\d+)$`, we don't use the variable `$1`, which would contain only the rest of the dialed number with the leading `1` stripped off. Instead, we use the variable `$0` that contains the original destination number.

Match a number and strip digits

In this example we need to match a called number beginning with `00`, but we also need to strip the leading digits. Assuming that FreeSWITCH receives the number `00123456789` and we need to strip the leading `00` digits, then we can use the following extension:

```
<extension name="Test3.1">
  <condition field="destination_number"
    expression="^00(\d+)$">
    <action application="bridge"
      data="sofia/profilename/$1@192.168.2.2"/>
  </condition>
</extension>
```

On the other hand, if you anticipate receiving non-digits, or you want to match on more than just digits, use `.+` instead of `\d+`, because `\d+` matches numeric digits only, whereas a `.+` will match all characters from the current position to the end of the string:

```
<extension name="Test3.2">
  <condition field="destination_number" expression="^00(.+)$">
    <action application="bridge"
      data="sofia/profilename/$1@192.168.2.2"/>
  </condition>
</extension>
```

Match a number, strip digits, and add a prefix

In this example we need to strip the leading digits as shown, but we also need to place a new prefix before the called number. Assuming that FreeSWITCH receives the number `00123456789` and we need to replace the `00` with `011`, we can use the following extension:

```
<extension name="Test4">
  <condition field="destination_number"
    expression="^00(\d+)$">
    <action application="bridge"
      data="sofia/profilename/011$1@x.x.x.x"/>
  </condition>
</extension>
```

Call a registered device

This example shows how to bridge to devices that have registered with your FreeSWITCH system. In this example we assume that you have set up a Sofia profile called `local_profile` and your phones are registering with the domain `example.com`. Note the `%` instead of `@` in the dial string:

```
<extension name="internal">
  <condition field="source" expression="mod_sofia"/>
  <condition field="destination_number" expression="^(4\d+)$">
    <action application="bridge"
      data="sofia/local_profile/$0%example.com"/>
  </condition>
</extension>
```

The use of `%` instead of `@` is a FreeSWITCH-specific feature. Using the form `user%domain` tells FreeSWITCH that a user is registered with domain, and that domain is being serviced by the FreeSWITCH directory.

Try party A, then party B

The following example shows how it is possible to call another action if the first action fails.

If the first action is successful, the call is bridged to `1111@example1.company.com` and will exist until one of the parties hangs up. After this, no other processing will be done because the caller's channel is closed. (In other words, `1111@example2.company.com` is not called.)

If the initial call to `1111@example1.company.com` was not successful, the channel will not be closed and the second action will be called.

```
<extension name="find_me">
  <condition field="destination_number" expression="^1111$">
    <action application="set"
      data="hangup_after_bridge=true"/>
    <action application="set" data="continue_on_fail=true"/>
    <action application="bridge"
        data="sofia/local_profile/1111@example1.company.com"/>
    <action application="bridge"
        data="sofia/local_profile/1111@example2.company.com"/>
  </condition>
</extension>
```

Route DIDs to extensions

To route incoming calls that come in to a certain DID via the context public to a fixed extension in the context `inhouse`, do something like the following:

```
<context name="public">
   <extension name="test_did">
     <condition field="destination_number"
       expression="^\d{6}(\d{4})$">
       <action application="transfer" data="$1 XML inhouse"/>
     </condition>
   </extension>
</context>
```

This will capture only the last four digits of a ten-digit number and transfer the caller to that number via the `inhouse` context. Note the parentheses around `\d{4}` that allow us to capture only the last four digits.

Alternate outbound gateways

In this example we send ten-digit outbound calls from `OfficeA` to `gateway1` and from `OfficeB` to `gateway2`. This assumes `OfficeA` and `OfficeB` are both using the same FreeSWITCH box but need different routing for outbound calls. It assumes both offices have 4-digit extensions, and that `OfficeA`'s extensions start with `2` and `OfficeB`'s extensions start with `3`.

```
<extension name="officeA_outbound">
  <condition field="caller_id_number"
             expression="^2\d{3}$"/>
  <condition field="destination_number"
             expression="^(\d{10})$">
      <action application="set"
        data="effective_caller_id_number=8001231234"/>
      <action application="set"
        data="effective_caller_id_name=Office A"/>
      <action application="bridge"
              data="sofia/gateway/myswitch.com/$1"/>
  </condition>
</extension>
<extension name="officeB_outbound">
  <condition field="caller_id_number"
             expression="^3\d{3}$"/>
  <condition field="destination_number"
             expression="^(\d{10})$">
      <action application="set"
         data="effective_caller_id_number=8001231235"/>
      <action application="set"
         data="effective_caller_id_name=Office B"/>
      <action application="bridge"
         data="sofia/gateway/otherswitch.com/$1"/>
  </condition>
</extension>
```

Multiple endpoints with enterprise originate

Consider this example: a customer wants to know if they can route a call to two different people. Person number one (`Alice`) prefers that her desk phone ring first, and then her mobile phone. Person number two (`Bob`) prefers that his desk phone and his mobile phone ring simultaneously. Whoever answers first-Alice or Bob-will take the call and all the other outbound calls will stop ringing.

This complicated scenario requires the use of FreeSWITCH's enterprise originate. The basic idea of an enterprise originate is that there are individual originates that are connected in a larger "enterprise". In our example, Alice's phones might be dialed like this:

```
<action application="bridge"
data="[leg_timeout=10]user/Alice|
[leg_timeout=20]sofia/gateway/my_gw/${alice_mobile}"/>
```

And for Bob:

```
<action application="bridge"
data="[leg_timeout=10]user/Bob|
[leg_timeout=20]sofia/gateway/my_gw/${bob_mobile}"/>
```

Each of these individual bridge attempts would work for calling either Alice or Bob, but not both at the same time. To accomplish this, put both of these into a single `bridge` action and separate them with the special `:_:` sequence. Here is an example:

```
<action application="bridge"
data="<ignore_early_media=true>[leg_timeout=10]user/Alice|
[leg_timeout=20]sofia/gateway/my_gw/${alice_mobile}:_:
[leg_timeout=10]user/Bob|
[leg_timeout=20]sofia/gateway/my_gw/${bob_mobile}"/>
```

In this case, when the inbound leg hits this bridge app, FreeSWITCH will initiate two separate "originates"-one for reaching `Alice` and the other for reaching `Bob`. The effect here is that FreeSWITCH tries to reach `Alice`, by calling her desk phone and then her mobile while at the same time calling `Bob` at his desk phone and his mobile phone. If anyone answers then the whole "enterprise" stops and the call is connected to the endpoint that answered.

Note that we are forced to use `ignore_early_media=true`, because we are creating so many call legs. There is no way to pick just one source of early media (ringing) and use it. Be sure to set the `ringback` or `transfer_ringback` variables if you need to supply some kind of ringing signal to the calling party.

Summary

In this chapter we delved very deeply into the operation of the FreeSWITCH Dialplan. Building upon the foundation laid in Chapter 6, *XML Dialplan*, we discussed many advanced Dialplan concepts:

- How Dialplan parsing works
- Using global variables and channel variables
- Advanced use of regular expressions
- Various advanced routing concepts

The Dialplan system in FreeSWITCH is one of the most important concepts you can learn. The power of FreeSWITCH is truly unleashed within the Dialplan system itself, and understanding the complexities of using various functions within FreeSWITCH is key to ensure that the FreeSWITCH performs exactly the way you want.

In the next chapter, we will lay the foundation for doing very powerful FreeSWITCH configurations that do not rely solely on the static XML files.

Dialplan, Directory, and ALL via XML_CURL and Scripts

The same XML Dialplan and User Directory (and actually all of the FreeSWITCH configuration) we saw in previous chapters can be served dynamically to FreeSWITCH, changing in real time, instead of being read from the filesystem.

Also, this is not an all or nothing choice: you can serve dynamically just the User Directory, or the dialplan, or the module configurations. And you can have fall-backs on filesystem that are read when the dynamic configuration is not covering that specific looked for item (as in "404 File not found").

You can serve that XML information via a web server to mod_xml_curl, or you can use scripts in supported languages (Lua, Perl, and so on). Scripts can do whatever to build the XML snippet to be returned to FreeSWITCH (query a database, use a template, and so on).

Each time FreeSWITCH needs to know something (a user configuration, or a dialplan extension) it will use the script (or mod_xml_curl) to retrieve the XML snippet with the relevant info.

This way, for example, you can have a cluster of FreeSWITCHes all accessing the same dynamic real time configuration.

Also, you can have that configuration, which determines FreeSWITCH behavior, changing call per call responding to your business logic, and so on.

We'll see:

- The logic behind mod_xml_curl
- How to configure mod_xml_curl
- How to set up a web server php script that mod_xml_curl will interrogate
- How to use scripts to dynamically retrieve XML snippets

mod_xml_curl rationale

mod_xml_curl uses the curl library, the same library behind the curl executable utility (https://curl.haxx.se/ great documentation!), to get a snippet of XML from a webserver in response to an internal FreeSWITCH request.

In its base configuration FreeSWITCH at startup reads its XML configuration from filesystem. FreeSWITCH XML configuration is composed by a couple of XML files that "include" many other XML files. The inclusion mechanism, and the inherent structure in XML files, results in one very big XML tree to be built into FreeSWITCH memory.

Each time FreeSWITCH needs to know something about its own configuration (module parameters, dialplan, User Directory, and so on), FreeSWITCH uses its internal XML tools to query the big in-memory XML tree. From each one of those queries, that in a busy server can be thousands at a second, FreeSWITCH receive an XML snippet response, containing the relevant data. The query/response is immediate and extremely light, like a binary tree in RAM.

mod_xml_curl can be configured to answer some or all of those internal XML queries by returning an XML snippet from a web server (local or remote).

Also, in case a first web server fails or does not contain the information, mod_xml_curl can be configured to try a second and third web server and/or letting FreeSWITCH fall back to read the XML from the local filesystem (like mod_xml_curl was not there)."Nothing to be found in this mod_xml_curl. Move along, read from filesystem":

mod_xml_curl configuration

Configuration of mod_xml_curl is where all modules' configurations are, in `/usr/local/freeswitch/conf/autoload_configs/xml_curl.conf.xml`.

Its XML tree in its simplest form is like this:

```
<configuration name="xml_curl.conf" description="cURL XML Gateway">
<bindings>
<binding name="localhostweb">
<param name="gateway-url" value="http://127.0.0.1/xml_handler.php" bindings="directory"/>
</binding>
</bindings>
</configuration>
```

Let's see its most important tags:

- **configuration** is the document container, and it specifies that this particular XML configuration snippet is "xml_curl.conf" (mod_xml_curl will look for it when loaded).
  - **bindings** contains all the "binding" tags.
    - **binding** each binding tag contains specifications about where and how mod_xml_curl will look for to retrieve a class of XML settings.
      - **gateway-url** this is the only mandatory tag inside a binding tag. Its **value** is the complete URL (protocol, server, port, resource) that will be used by mod_xml_curl to retrieve the XML snippet. It has an attribute **bindings** that specifies which class of XML snippets will be served by this gateway.

Inside each binding tag you can have one only gateway-url.

Each gateway-url can have its bindings attribute set to one or more (separated by pipe sign "|") of those classes of XML snippets:

- **directory**: the User Directory, registered users authentication and parameters/variables
- **dialplan**: contexts and extensions
- **phrases**: languages and XML IVR constructs for the "say" API
- **configuration**: all that is not covered by the other three

It can be a little confusing, because the same terms keep appearing in singular and plural quantities, as tag names and as attribute values. Be strong! You can do it!

Inside each **binding** tag you can have many optional tags that will further tell the curl library how to retrieve the XML snippet (for example, GET/POST, SSL related settings, password, timeout, and so on).

If you have **more than one binding that has a gateway-url for the same class of XML snippets**, if the first fails, the second will be tried, and so on. This can be used as "**client side high availability**" (obviously you can implement HA on the web server side, with classical web load balancing techniques).

Ifallbinding for a class of XML snippets **fail**, for example the webservers are down or timeout or the responses are invalid XML snippets, **then the XML snippet** will be looked upon in "traditional" XML configuration **on filesystem**.

If a **binding fails, an error is reported** in console and logfile, so you know and can fix it. Eg: failover works, but an error is reported.

If you do not want the error to be reported, but you want **to silently fail over to another binding or to filesystem** (because this is the expected behavior, binding is expected not to have all possible XML snippets), the **webserver must return a "not found" content** (as opposed to nothing or empty or invalid), see the example in next section "php/mysql setup for mod_xml_curl".

Let's see a couple more complex mod_xml_curl configurations, modified from our confluence documentation:

```
<configuration name="xml_curl.conf" description="cURL XML Gateway">
<bindings>
<!-- Ask example1.com for a dialplan config, if we get a valid response then don't continue c
<binding name="dialplan">
<param name="gateway-url" value="http://example1.com:80/fsapi" bindings="dialplan"/>
</binding>

<!-- This will only get called if example1.com returns an invalid or not found response. If t
<binding name="dialplan backup">
<param name="gateway-url" value="http://example2.com:80/fsapi" bindings="dialplan"/>
</binding>

<!-- Ask example1.com for directory and phrases config, if not found in case of directory the
<binding name="example1">
<param name="gateway-url" value="http://example1:80/fsapi" bindings="directory|phrases"/>
</binding>

<!-- Ask example1.com for a configuration config, if not found then revert to disk -->
<binding name="configuration">
<param name="gateway-url" value="http://example1:80/fsapi" bindings="configuration"/>
</binding>

<!-- Ask example2.com for a phrases config, if not found then revert to disk -->
<binding name="example2">
<param name="gateway-url" value="http://example2:80/fsapi" bindings="phrases"/>
</binding>
</bindings>
</configuration>
```

Also, you want to check the comments in the example config file installed by demo configuration:

```
<binding name="example">
<!-- Allow to bind on a particular IP for requests sent -->
<!--<param name="bind-local" value="$${local_ip_v4}" />-->
<!-- The url to a gateway cgi that can generate xml similar to
what's in this file only on-the-fly (leave it commented if you dont
need it) -->
<!-- one or more |-delim of configuration|directory|dialplan -->
<!-- <param name="gateway-url" value="http://www.freeswitch.org/gateway.xml" bindings="dialp
<!-- set this to provide authentication credentials to the server -->
<!--<param name="gateway-credentials" value="muser:mypass"/>-->
<!--<param name="auth-scheme" value="basic"/>-->

<!-- optional: this will enable the CA root certificate check by libcurl to
verify that the certificate was issued by a major Certificate Authority.
note: default value is disabled. only enable if you want this! -->
<!--<param name="enable-cacert-check" value="true"/>-->
<!-- optional: verify that the server is actually the one listed in the cert -->
<!-- <param name="enable-ssl-verifyhost" value="true"/> -->

<!-- optional: these options can be used to specify custom SSL certificates
to use for HTTPS communications. Either use both options or neither.
Specify your public key with 'ssl-cert-path' and the private key with
'ssl-key-path'. If your private key has a password, specify it with
'ssl-key-password'. -->
<!-- <param name="ssl-cert-path" value="$${certs_dir}/public_key.pem"/> -->
<!-- <param name="ssl-key-path" value="$${certs_dir}/private_key.pem"/> -->
<!-- <param name="ssl-key-password" value="MyPrivateKeyPassword"/> -->
<!-- optional timeout -->
<!-- <param name="timeout" value="10"/> -->

<!-- optional: use a custom CA certificate in PEM format to verify the peer
with. This is useful if you are acting as your own certificate authority.
note: only makes sense if used in combination with "enable-cacert-check." -->
<!-- <param name="ssl-cacert-file" value="$${certs_dir}/cacert.pem"/> -->

<!-- optional: specify the SSL version to force HTTPS to use. Valid options are
"SSLv3" and "TLSv1". Otherwise libcurl will auto-negotiate the version. -->
<!-- <param name="ssl-version" value="TLSv1"/> -->

<!-- optional: enables cookies and stores them in the specified file. -->
<!-- <param name="cookie-file" value="$${temp_dir}/cookie-mod_xml_curl.txt"/> -->

<!-- one or more of these imply you want to pick the exact variables that are transmitted --
<!--<param name="enable-post-var" value="Unique-ID"/>-->
</binding>
```

mod_xml_curl: Caveats and Pitfalls

So, you get excited by the endless possibilities offered by mod_xml_curl and you want to implement it yourself. Good!

But you want to take into account some things:

- FreeSWITCH makes **a lot of XML lookups**. Really: a lot! A very big fast whopping lot!
- You want your **webserver** to be **very fast** in returning the XML snippets, in failing, in timeouts, in all things: **FreeSWITCH will wait** for it
- You want to **set the timeout in each binding**. In case of webserver not responding or slow, **FreeSWITCH will wait** for it
- Particularly for directory, FreeSWITCH will not initiate a new phone registration until the previous has been completed or failed
- You may want to allow for **FreeSWITCH caching of XML snippets**. Look in mod_xml_curl confluence documentation for it
- You may want to use **PHP caching, memcached** or similar systems
- You want to **cure all aspect of web security**, both for denial of service, for hackability of server side scripts, for exposition of your database, etc
- You **DO NOT** want to have **FreeSWITCH and database/webserver on the same hardware** machine. It's a no-no. Also having FreeSWITCH in a vm and database/webserver in another vm on the same hardware is a no-no. If you expect significant traffic, don't.

php/mysql setup for mod_xml_curl

I adapted all the following example setup, including database and PHP code, from the book "Costruire centralini telefonici con FreeSWITCH", written by FreeSWITCH Italian champion Christian Bergamaschi. Look it up on Amazon and elsewhere! (in Amazon.IT you can even have it as printed paper book).

For testing, you can install webserver, php, and mysql on the same FreeSWITCH machine. For production you want them to be on a separate hardware on the same gigabit LAN.

In Debian 8 Jessie you do:

```
apt-get install mysql-server nginx php5-fpm php5-mysql
```

You'll be asked for a root password for mysql server, set it to something you remember, you'll need it again.

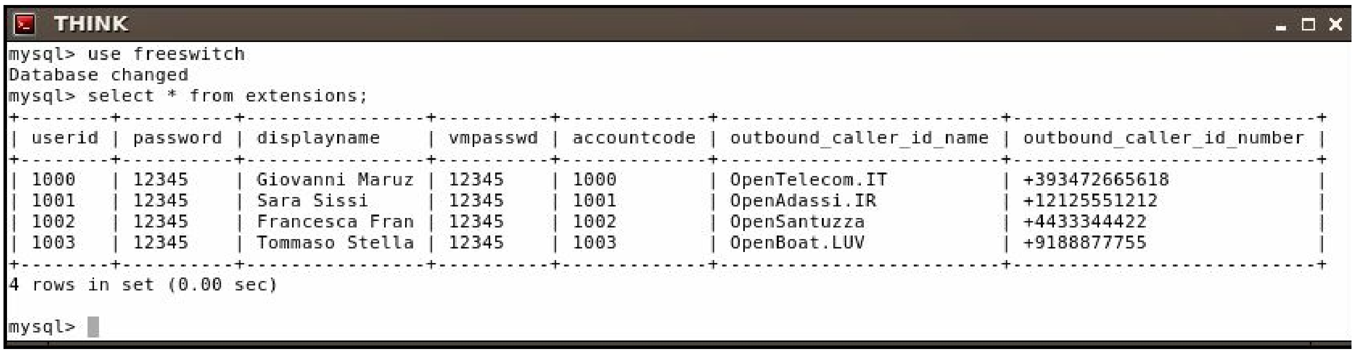
Create and populate the Database

From the Linux root command line, type:

```
mysql -A -p
```

Then, from the mysql console:

```
create database freeswitch;
use freeswitch;
CREATE TABLE `extensions` ( `userid` varchar(5) NOT NULL DEFAULT '', `password` varchar(30) N
insert into extensions (userid,password,displayname,vmpasswd,accountcode,outbound_caller_id_n
select * from extensions;
```



THINK

```
mysql> use freeswitch
Database changed
mysql> select * from extensions;
+--------+----------+----------------+----------+-------------+------------------------+--------------------------+
| userid | password | displayname    | vmpasswd | accountcode | outbound_caller_id_name | outbound_caller_id_number |
+--------+----------+----------------+----------+-------------+------------------------+--------------------------+
| 1000   | 12345    | Giovanni Maruz | 12345    | 1000        | OpenTelecom.IT         | +393472665618            |
| 1001   | 12345    | Sara Sissi     | 12345    | 1001        | OpenAdassi.IR          | +12125551212             |
| 1002   | 12345    | Francesca Fran | 12345    | 1002        | OpenSantuzza           | +4433344422              |
| 1003   | 12345    | Tommaso Stella | 12345    | 1003        | OpenBoat.LUV           | +9188877755              |
+--------+----------+----------------+----------+-------------+------------------------+--------------------------+
4 rows in set (0.00 sec)

mysql>
```

Setup NGINX and xmlhandler.php

Edit /etc/nginx/sites-enabled/default and uncomment the lines needed to activate PHP and php5-fpm, and then "service nginx restart" from the command line.

Create the file /var/www/html/xml_handler.php with the following content:

```
<?php
function not_found()
{
Header("Content-type: text/xml");
$xmlw = new XMLWriter();
$xmlw -> openMemory();
$xmlw -> setIndent(true);
$xmlw -> setIndentString('');
$xmlw -> startDocument('1.0', 'UTF-8', 'no');
$xmlw -> startElement('document');
$xmlw -> writeAttribute('type', 'freeswitch/xml');
$xmlw -> startElement('section');
$xmlw -> writeAttribute('name', 'result');
$xmlw -> startElement('result');
$xmlw -> writeAttribute('status', 'not found');
$xmlw -> endElement(); //end result
$xmlw -> endElement(); //end section
$xmlw -> endDocument(); //end document
echo $xmlw -> outputMemory();
return TRUE;
}

function directory()
{
global $_SERVER;
global $_POST;
# connect to MySQL
$connect = mysql_connect("localhost", "root", "mysql_root_password");
mysql_select_db("freeswitch", $connect) or die(mysql_error());

# Query
$query = "SELECT
userid,password,displayname,vmpasswd,accountcode,outbound_caller_id_name,outbound_caller_id_r
# perform the query
$result = mysql_query($query, $connect) or die(mysql_error());
$num_rows = mysql_num_rows($result);

if($num_rows==0){
# if no database row, fallback to filesystem, no FS error
not_found();
return TRUE;
}
Header("Content-type: text/xml");
$xmlw = new XMLWriter();
$xmlw -> openMemory();
$xmlw -> setIndent(true);
$xmlw -> setIndentString('');
$xmlw -> startDocument('1.0', 'UTF-8', 'no');
$xmlw -> startElement('document');
$xmlw -> writeAttribute('type', 'freeswitch/xml');
$xmlw -> startElement('section');
$xmlw -> writeAttribute('name', 'directory');
$xmlw -> startElement('domain');
$xmlw -> writeAttribute('name','$${domain}');
$xmlw -> startElement('params');
$xmlw -> startElement('param');
$xmlw -> writeAttribute('name', 'dial-string');
$xmlw -> writeAttribute('value', '{^^:sip_invite_domain=$
{dialed_domain}:presence_id=${dialed_user}@${dialed_domain}}$
{sofia_contact(*/${dialed_user}@${dialed_domain})}');
$xmlw -> endElement(); //end param
$xmlw -> endElement(); //end param
while( $row = mysql_fetch_array($result, MYSQL_ASSOC) ) {
$xmlw -> startElement('user');
$xmlw -> writeAttribute('id', $row['userid']);
// Params
$xmlw -> startElement('params');
$xmlw -> startElement('param');
$xmlw -> writeAttribute('name', 'password');
$xmlw -> writeAttribute('value', $row['password']);
$xmlw -> endElement();
$xmlw -> startElement('param');
$xmlw -> writeAttribute('name', 'vm-password');
$xmlw -> writeAttribute('value', $row['vmpasswd']);
$xmlw -> endElement();
$xmlw -> endElement(); //end params
// Variables
$xmlw -> startElement('variables');
$xmlw -> startElement('variable');
$xmlw -> writeAttribute('name', 'accountcode');
$xmlw -> writeAttribute('value', $row['accountcode']);
$xmlw -> endElement();
$xmlw -> startElement('variable');
$xmlw -> writeAttribute('name', 'user_context');
$xmlw -> writeAttribute('value', 'default');
$xmlw -> endElement();
$xmlw -> startElement('variable');
$xmlw -> writeAttribute('name', 'effective_caller_id_name');
$xmlw -> writeAttribute('value', $row['displayname']);
$xmlw -> endElement();
$xmlw -> startElement('variable');
$xmlw -> writeAttribute('name', 'effective_caller_id_number');
$xmlw -> writeAttribute('value', $row['userid']);
$xmlw -> endElement();
$xmlw -> startElement('variable');
$xmlw -> writeAttribute('name', 'outbound_caller_id_name');
$xmlw -> writeAttribute('value', $row['outbound_caller_id_name']);
$xmlw -> endElement();
$xmlw -> startElement('variable');
$xmlw -> writeAttribute('name', 'outbound_caller_id_number');
$xmlw -> writeAttribute('value', $row['outbound_caller_id_number']);
$xmlw -> endElement();
$xmlw -> endElement(); //end variables
$xmlw -> endElement(); //end user
} // end while
$xmlw -> endElement(); //end domain
$xmlw -> endElement(); //end section
$xmlw -> endDocument(); //end document
echo $xmlw -> outputMemory();
return TRUE;
}

if (isset($_POST['section'])) {
if($_POST['section']=='directory') {
directory();
} else {
# if section is not directory, fallback to filesystem, no FS error
not_found();
}
}
?>
```

At the end of the PHP script we see its main body that checks if the POST request has a "section" field. If the "section" web request field has the value "directory", then the directory() function is executed, or else the not_found() function is executed.

This will allow us to use this script to answer mod_xml_curl requests for whatever XML snippet class: if that class ("section") is not the expected one ("directory") we'll properly return the "not found" document and FreeSWITCH will happily fall back to the filesystem without logging errors.

If the request was for the "directory" class, then we make the database query asking for the XML snippet describing the desired user, identified by "userid" (that is, the SIP user).

If we find rows in the database we return the properly formatted XML snippet.

If in the database there is no row for that userid (zero returned rows from mysql query) then again we properly return the "not found" document, so FreeSWITCH will happily fall back to the filesystem without logging errors.

mod_xml_curl configuration

Edit the file /usr/local/freeswitch/conf/autoload_configs/xml_curl.conf.xml to be:

```
<configuration name="xml_curl.conf" description="cURL XML Gateway">
<bindings>
<binding name="localhostweb">
<param name="gateway-url" value="http://127.0.0.1:8888/xml_handler.php" bindings="directory"/
<binding name="localhostweb2">
<param name="gateway-url" value="http://127.0.0.1:80/xml_handler.php" bindings="directory"/>
</binding>
</bindings>
</configuration>
```

This configuration will first try to get the XML snippet from the non-existing web server on port 8888 of the FreeSWITCH server machine. It will fail immediately (because no one is listening at that port), log an error, and try the second binding. The second binding will try to get an answer from the listening webserver (port 80).

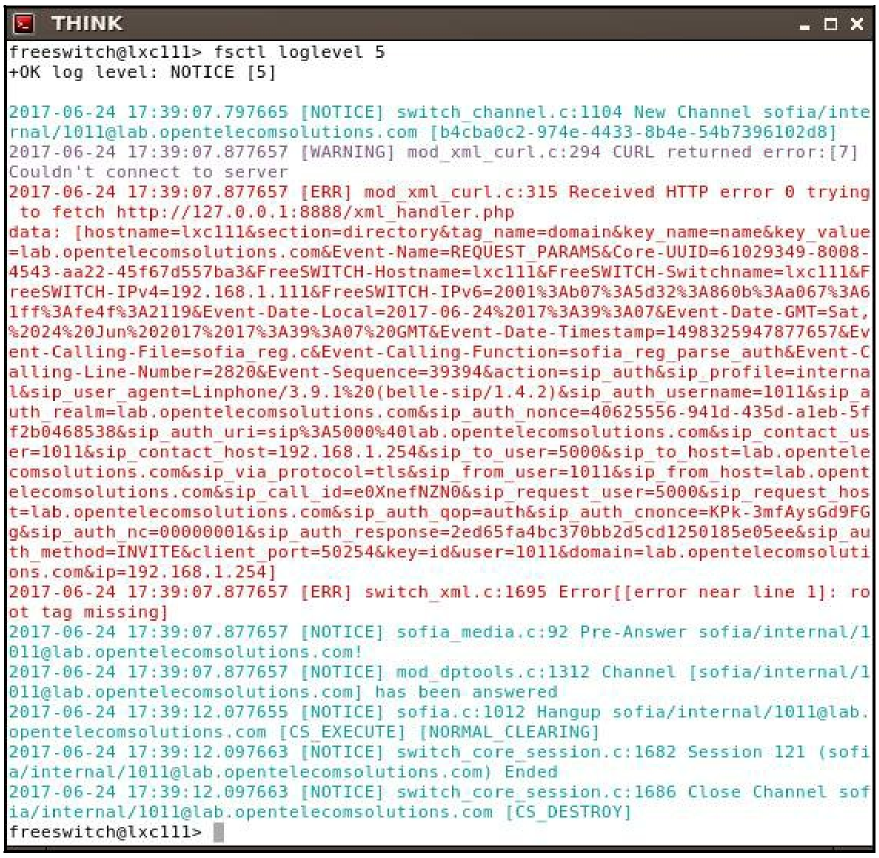
Testing php/mysql setup for mod_xml_curl

From the FreeSWITCH console command line now "load mod_xml_curl" (or "reload" it).

Each time FreeSWITCH will need to check User Directory (to authorize a phone registration in SIP or Verto, to authorize an outbound call from a registered phone, to find out details to serve an incoming call for a SIP or Verto user, and so on), mod_xml_curl will enter in action.

First mod_xml_curl will try and fail to connect to a nonexisting webserver on port 8888, and log an error.

Then will connect to port 80, ask for the XML snippet, and if that user is not in database, will silently (no error logged) fall back to the filesystem.



```
freeswitch@lxc111> fsctl loglevel 5
+OK log level: NOTICE [5]

2017-06-24 17:39:07.797665 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1011@lab.opentelecomsolutions.com [b4cba0c2-974e-4433-8b4e-54b7396102d8]
2017-06-24 17:39:07.877657 [WARNING] mod_xml_curl.c:294 CURL returned error:[7]
Couldn't connect to server
2017-06-24 17:39:07.877657 [ERR] mod_xml_curl.c:315 Received HTTP error 0 trying
 to fetch http://127.0.0.1:8888/xml_handler.php
data: [hostname=lxc111&section=directory&tag_name=domain&key_name=name&key_value
=lab.opentelecomsolutions.com&Event-Name=REQUEST_PARAMS&Core-UUID=61029349-8008-
4543-aa22-45f67d557ba3&FreeSWITCH-Hostname=lxc111&FreeSWITCH-Switchname=lxc111&F
reeSWITCH-IPv4=192.168.1.111&FreeSWITCH-IPv6=2001%3Ab07%3A5d32%3A860b%3Aa067%3A6
1ff%3Afe4f%3A2119&Event-Date-Local=2017-06-24%2017%3A39%3A07&Event-Date-GMT=Sat,
%2024%20Jun%202017%2017%3A39%3A07%20GMT&Event-Date-Timestamp=1498325947877657&Ev
ent-Calling-File=sofia_reg.c&Event-Calling-Function=sofia_reg_parse_auth&Event-C
alling-Line-Number=2820&Event-Sequence=39394&action=sip_auth&sip_profile=interna
l&sip_user_agent=Linphone/3.9.1%20(belle-sip/1.4.2)&sip_auth_username=1011&sip_a
uth_realm=lab.opentelecomsolutions.com&sip_auth_nonce=40625556-941d-435d-a1eb-5f
f2b0468538&sip_auth_uri=sip%3A5000%40lab.opentelecomsolutions.com&sip_contact_us
er=1011&sip_contact_host=192.168.1.254&sip_to_user=5000&sip_to_host=lab.opentele
comsolutions.com&sip_via_protocol=tls&sip_from_user=1011&sip_from_host=lab.opent
elecomsolutions.com&sip_call_id=e0XnefNZN0&sip_request_user=5000&sip_request_hos
t=lab.opentelecomsolutions.com&sip_auth_qop=auth&sip_auth_cnonce=KPk-3mfAysGd9FG
g&sip_auth_nc=00000001&sip_auth_response=2ed65fa4bc370bb2d5cd1250185e05ee&sip_au
th_method=INVITE&client_port=50254&key=id&user=1011&domain=lab.opentelecomsoluti
ons.com&ip=192.168.1.254]
2017-06-24 17:39:07.877657 [ERR] switch_xml.c:1695 Error[[error near line 1]: ro
ot tag missing]
2017-06-24 17:39:07.877657 [NOTICE] sofia_media.c:92 Pre-Answer sofia/internal/1
011@lab.opentelecomsolutions.com!
2017-06-24 17:39:07.877657 [NOTICE] mod_dptools.c:1312 Channel [sofia/internal/1
011@lab.opentelecomsolutions.com] has been answered
2017-06-24 17:39:12.077655 [NOTICE] sofia.c:1012 Hangup sofia/internal/1011@lab.
opentelecomsolutions.com [CS_EXECUTE] [NORMAL_CLEARING]
2017-06-24 17:39:12.097663 [NOTICE] switch_core_session.c:1682 Session 121 (sofi
a/internal/1011@lab.opentelecomsolutions.com) Ended
2017-06-24 17:39:12.097663 [NOTICE] switch_core_session.c:1686 Close Channel sof
ia/internal/1011@lab.opentelecomsolutions.com [CS_DESTROY]
freeswitch@lxc111> 
```

In this case we call from the user 1011, that is not in the database (we have userid 1000...1003 in the database).

First mod_xml_curl will try and fail to connect to port 888, and log an error. Then it will connect to port 80, get the "not found" document, and fall back to filesystem, where the user 1011 is found, and the call is authorized.

Phew!

Lua XML handler

Exactly in the same vein as mod_xml_curl, you can have FreeSWITCH retrieve its XML configuration snippets by executing a script in one of the supported scripting languages.

You must first configure the scripting language module, in this case mod_lua, to handle XML lookup requests for a class of XML snippets.

Then you create a script that sets a variable named XML_STRING, containing the XML snippet.

That's it. Because lua is embedded in FreeSWITCH (as python and perl) there is no need for anything else.

You would connect to databases, webservers, do your data massaging, and so on from the script itself. The script is just supposed to return a string containing the XML snippet. With any means necessary.

This example Lua script can be configured as xml-handler-script, it sets the variable XML_STRING to the XML document FreeSWITCH is looking for:

lua xmlhandler script

```
xml_snippet = [[
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<document type="freeswitch/xml">
<section name="directory">
<domain name="lab.opentelecomsolutions.com">
<params>
<param name="dial-string" value="{^^:sip_invite_domain=${dialed_domain}:presence_id=${dialed_
</params>
<user id="1000">
<params>
<param name="password" value="56789"/>
<param name="vm-password" value="56789"/>
</params>
<variables>
<variable name="accountcode" value="1000"/>
<variable name="user_context" value="default"/>
<variable name="effective_caller_id_name" value="Giovanni Maruz"/>
<variable name="effective_caller_id_number" value="1000"/>
<variable name="outbound_caller_id_name" value="OpenTelecom.IT"/>
<variable name="outbound_caller_id_number" value="+393472665618"/>
</variables>
</user>
<user id="1001">
<params>
<param name="password" value="56789"/>
<param name="vm-password" value="56789"/>
</params>
<variables>
<variable name="accountcode" value="1001"/>
<variable name="user_context" value="default"/>
<variable name="effective_caller_id_name" value="Sara Sissi"/>
<variable name="effective_caller_id_number" value="1001"/>
<variable name="outbound_caller_id_name" value="OpenAdassi.IR"/>
<variable name="outbound_caller_id_number" value="+12125551212"/>
</variables>
</user>
</domain>
</section>
</document>
]]
XML_STRING = xml_snippet

freeswitch.consoleLog("notice", XML_STRING .. "\n")
```

lua.conf.xml

You must add these two lines to ?usr/local/freeswitch/conf/autoload_configs/lua.conf.xml:

```
<param name="xml-handler-script" value="xml_handler.lua"/>
<param name="xml-handler-bindings" value="directory"/>
```

Lua xml_handler.lua testing

You must first restart FreeSWITCH (because you cannot unload or reload mod_lua).

Then you try a call from user 1000 or 1001. Notice that in the script they have a "56789" password. So FreeSWITCH will at first reject the call or registration, then will ask you again for the password, and eventually will authorize.

Summary

In this chapter, we looked at how to obtain XML snippets dynamically, and how to fall back to filesystem if we fail.

We used two mechanisms for the same aim: getting the same XML snippet.

We can use mod_xml_curl to interrogate a remote or local database.

We can use one of the supported scripting languages to build/retrieve the XML snippet.

We also saw the caveats: FreeSWITCH will need a very very fast execution of those lookup and timeouts to protect the users from sluggish backend applications.

In the next chapter, we will see how to control FreeSWITCH externally, via ESL, the most powerful way to micromanage what FreeSWITCH does.

ESL - FreeSWITCH Controlled by Events

FreeSWITCH is a core switching and mixing matrix with dozens of modules around it, providing functionalities and features.

Lowest level control provided by events allows us to effectively leverage the toolbox approach, where we can choose how and when to use the single tools and techniques.

We have in our full control the entire wealth of real-time information that composes all communication packets flowing in and out of FreeSWITCH.

At the same time, we can bring each functionality from any module to life, invoking it at the right moment.

Full knowledge and full control allows us to tailor FreeSWITCH to the minimum details of our project, we'll never be forced to admit: "I can't do that".

In this chapter, we'll see:

- What is the Event System and its role in FreeSWITCH
- How Events look like
- How to connect to mod_event_socket
- Why Event Socket Library is so relevant
- How to write scripts to leverage the power of ESL

Event System Fundamentals

In first chapter of this book we introduced the Event System and its role in FreeSWITCH architecture.

We'll not repeat that intro here. **Let's** have some bullets that refresh it for us, and **add important insights**:

- **FreeSWITCH is made by a core** hyper-optimized switching and mixing engine **interacting with modules** implementing all sort of features
- **Event System is an internal messaging bus** optimized for distribution of "events"
- **Anything happens** inside FreeSWITCH **generates** one or more **events**
- There are **two different categories** of events: **system events and logging events**
- **Logging events** are generated any time something has to be logged. Logging events are then listened to by the modules and applications that write logfiles, write in database, in syslog, in windows registry, etc
- **System Events** are **generated** both **by** FreeSWITCH **core**, by **modules**, and by external **application** both in reaction to "**external**" stimulus and to express "**internal**" statuses and proceedings
- From now on, **when we write "events"**, without further specification, **we mean "system events"**
- Events are the **main way for interaction betweencore** and modules, between **modules** and modules, and between FreeSWITCH and **external applications**
- Events can be used to carry "**commands**" that **will be executed by modules or core**
- Events comes in many **categories** and **subcategories**
- **Events** distributed by Event System bus **can be "listened for" by** modules and external applications **that have "subscribed" (or "binded") to** that event **category/subcategory**
- Events can be "**fired**", as in "**injected**", into the Event System bus by core, modules and external applications
- **Firing** events **is** a "**shoot and forget**" action. Shooter has no direct feedback other that the event was accepted in the bus
- **All "subscribers" or "listeners"** to the event **category/subcategory** chosen by shooter will then **receive that event, and decide how to react**
- **When all subscribers have read ("consumed") the event**, the event **is deleted** from the Event System bus.
- **If even one only subscriber do not read** the events is subscribed to **fast**

enough, those **unconsumed events queue up** until that subscriber consume them or the queue is filled

- Event System **queues are very big** (like 10 thousands events), **but not infinite**. If the queues are completely filled, **Event System bus stops, and FreeSWITCH fails**.
- **External applications** can **access** the FreeSWITCH **Event System via** interfaces provided by **modules**
- Most important one is **mod_event_socket**, that **exposes** the Event System **via a bidirectional TCP socket**, to which external applications connect and exchange events in text format

Event format - Headers and Body

Before going to more depths, let's see what an "event" is. Event is a text message in a format loosely based on those of famous Internet protocols such as SIP, SMTP, and HTTP: many "headers" (key/value pairs) and possibly one "body" (free form payload):

```
Event-Name: RE_SCHEDULE
Core-UUID: 6ddf010e-bc0b-4366-8413-6becd7f77076
FreeSWITCH-Hostname: lxc111
FreeSWITCH-Switchname: lxc111
FreeSWITCH-IPv4: 192.168.1.111
FreeSWITCH-IPv6: 2001%3Ab07%3A5d32%3A860b%3Aa067%3A61ff%3Afe4f%3A2119
Event-Date-Local: 2017-06-26%2018%3A39%3A51
Event-Date-GMT: Mon,%2026%20Jun%202017%2018%3A39%3A51%20GMT
Event-Date-Timestamp: 1498502391402930
Event-Calling-File: switch_scheduler.c
Event-Calling-Function: switch_scheduler_execute
Event-Calling-Line-Number: 71
Event-Sequence: 22105
Task-ID: 1
Task-Desc: heartbeat
Task-Group: core
Task-Runtime: 1498502411
```

This is the dump of one of the easiest events to spot, the rescheduling of the heartbeat (an internal FreeSWITCH pulse). This kinds of events, like most events, do not have a body.

We can see here the textual structure of events, composed by headername/headervalue pairs separated by ":" (colon).

In the following example, we see an event with a body, a payload. The last header before the body (separated by double newline, for example, an empty line) indicates the body length, like in SIP messages:

```
Event-Name: MESSAGE
Core-UUID: 6ddf010e-bc0b-4366-8413-6becd7f77076
FreeSWITCH-Hostname: lxc111
FreeSWITCH-Switchname: lxc111
FreeSWITCH-IPv4: 192.168.1.111
FreeSWITCH-IPv6: 2001%3Ab07%3A5d32%3A860b%3Aa067%3A61ff%3Afe4f%3A2119
Event-Date-Local: 2017-06-26%2019%3A09%3A13
Event-Date-GMT: Mon,%2026%20Jun%202017%2019%3A09%3A13%20GMT
Event-Date-Timestamp: 1498504153183007
Event-Calling-File: switch_loadable_module.c
Event-Calling-Function: switch_core_chat_send_args
Event-Calling-Line-Number: 864
Event-Sequence: 22348
proto: global
from: 1011%40lab.opentelecomsolutions.com
to: 1001%40lab.opentelecomsolutions.com
type: text/plain
skip_global_process: true
blocking: true
dest_proto: sip
Delivery-Result-Code: 200
```

```
Delivery-Failure: false
Content-Length: 12

ciao a tutti
```

Some headers always available in any event are:

- **Event-Name**: The event's name, which is a description of the type of event it is. Also known as event's **Type** or **Class**.
- **Core-UUID**: The UUID of the current object in the FreeSWITCH core related to this event.
- **Event-Date-Local**: The date/time of the event according to the system clock.
- **Event-Date-GMT**: The date/time of the event in GMT (that is UTC) time.
- **Event-Calling-File**: The C source file from which the event was fired.
- **Event-Calling-Function**: The name of the function that fired this event.
- **Event-Calling-Line-Number**: The exact line number of the C source file where this event was fired.

Apart from these common headers, each event **type** has a number of other fields. Some of them are very familiar, as in case of channel and system variables (see previous chapters). Others are specific to the event type.

There are many **types of events**, for a complete list search http://freeswitch.org/confluence for "Event Types". Some important **event types** (or **classes** or **names**) are as follows:

- CHANNEL_*
- API
- DTMF
- MESSAGE
- CODEC
- PLAYBACK_(START|STOP)
- RECORD_(START|STOP)
- CUSTOM

The last of those event types, **CUSTOM**, is a porte-manteau class where developers can add events specific to a module, or to a feature, or to a service. So in case of events of CUSTOM type, the interesting header, the header that indicates the event's role and meaning, is "**Event-Subclass**".

CUSTOM events are **generated** by most **endpoint modules** (for example, when a user logs into Verto, or a SIP phone try to register), and by **many applications**, particularly by the conference application, but also by fax management in mod_spandsp, by voicemail application, by fifo, callcenter, and so on. Also, XML

IVR will generate CUSTOM events that subclasses describe entering and exiting of menus.

For a complete list of subclasses, please search our confluence for "CUSTOM Event SUBCLASSES". Some examples of CUSTOM event subclasses from various modules are as follow:

- sofia::register_attempt
- menu::enter
- spandsp::rxfaxresult
- filestring::open
- conference::maintenance

In case of **conference::maintenance** (as in some other subclasses) there is **another interesting header**, which describes the meaning of the event. In this specific **conference case**, it is called **Action**.

Some Action headers of CUSTOM event subclass conference::maintenance are as follows:

- conference-create
- add-member
- floor-change
- start-talking
- play-file-done

Event Handling modules

Event System is circulating events around FreeSWITCH core and modules. What if you want to **tap into the event bus and listen, or inject** some events in it? There are modules called **event "handlers"** (whose source code is in /usr/src/freeswitch/src/mod/event_handlers) that are able to interface the Event System with pretty much anything:

- mod_amqp
- mod_cdr_csv
- mod_cdr_mongodb
- mod_cdr_pg_csv
- mod_cdr_sqlite
- mod_erlang_event
- mod_event_multicast
- mod_event_socket
- mod_event_test
- mod_event_zmq
- mod_format_cdr
- mod_json_cdr
- mod_kazoo
- mod_odbc_cdr
- mod_radius_cdr
- mod_rayo
- mod_smpp
- mod_snmp

All of those modules interface the Event System with the external world, specifically for Call Detail Records accounting (all the *cdr* modules) or for more general purposes.

Particularly important are the modules that interface **amqp** (for example, rabbitMQ, and so on), **zeroMQ**, and **Erlang**. The most popular interface to the Event System is provided by mod_event_socket.

mod_event_socket

This is the workhorse for FreeSWITCH's Event System interaction with the external world. **mod_event_socket** works in the most simple way: it **opens a TCP socket, and listen** for incoming connections. Multiple **clients** can **connect** and **interact** bidirectionally with the Event System via **textual commands** and responses.

In default configuration **only connections to 127.0.0.1** loopback interface, unreachable from the network, are accepted by a **network ACL**. See below how to change this.

All **communication is exchanged in clear text** on the TCP socket, so if you want to connect from a remote machine, be sure to do it via a **vpn** or an **ssh** connection. Or have some form of encrypted middleware/proxy.

mod_event_socket accepts **many different commands**, we'll see later, for displaying and filtering events, and for injecting events in the Event System bus.

Events are displayed as soon as they appear on the bus, and on **a busy FreeSWITCH server** this can be overwhelming for your terminal to display, and **can hog the machine**. **You want to filter**, or find a way to consume them fast enough (for example, write them to a file you then grep into, or feed them to an application). We'll see much more on events filtering below.

mod_event_socket configuration

The XML configuration file is, as expected, in /usr/local/freeswitch/conf/autoload_configs/event_socket.conf.xml

In **demo configuration**, mod_event_socket listens on port **8021** of all network interfaces, but its **ACLaccepts incoming connections only from 127.0.0.1** (loopback interface on localhost). All other connections are answered with an "access denied" and shut down.

You can alter which connections are accepted by tuning two knobs:

- **listen-ip** default is "**::**" for listening on all interfaces, ipv4 and ipv4. You can set this parameter to "0.0.0.0" for all interfaces, ipv4 only, or "127.0.0.1" for loopback interface on localhost, etc
- **apply-inbound-acl** by default is "**loopback.auto**", only loopback interface on localhost is allowed. You can write an acl in /usr/local/freeswitch/conf/autoload_configs/acl.conf.xml and reference it here, or you can use "any_v4.auto" for allowing all IP addresses

Other obvious configuration parameters are **port** and **password**.

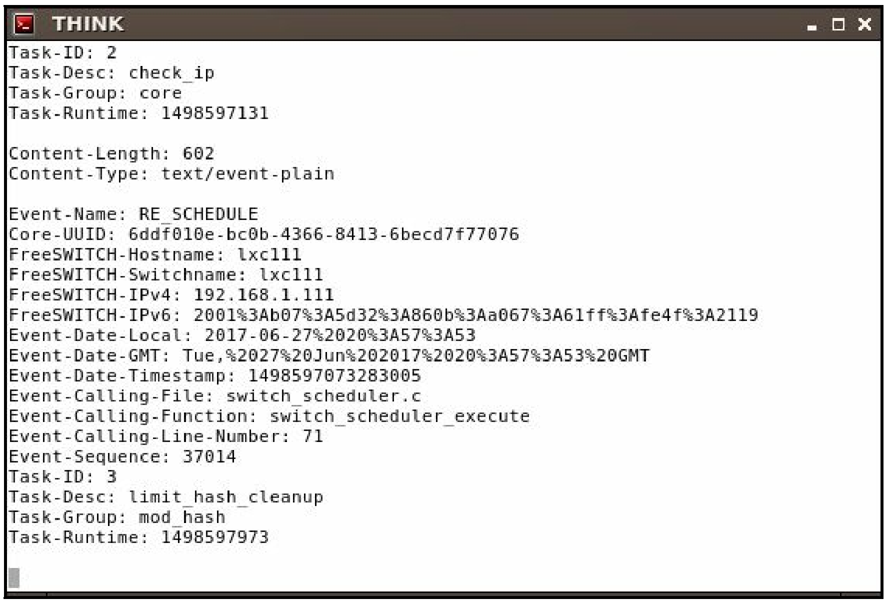
SECURITY, PLEASE NOTE: I repeat here again, **all communication is exchanged in clear text on the TCP socket**, so if you want to connect from a remote machine, be sure to do it via a **vpn** or an **ssh** connection. Or have some form of **encrypted** middleware/proxy.

Reading and sending events via telnet

The following example shows how to connect to mod_event_socket via telnet, and start viewing all events, as they come into the Event System bus. **In bold is what we typed. After each command, press "Enter" twice:**

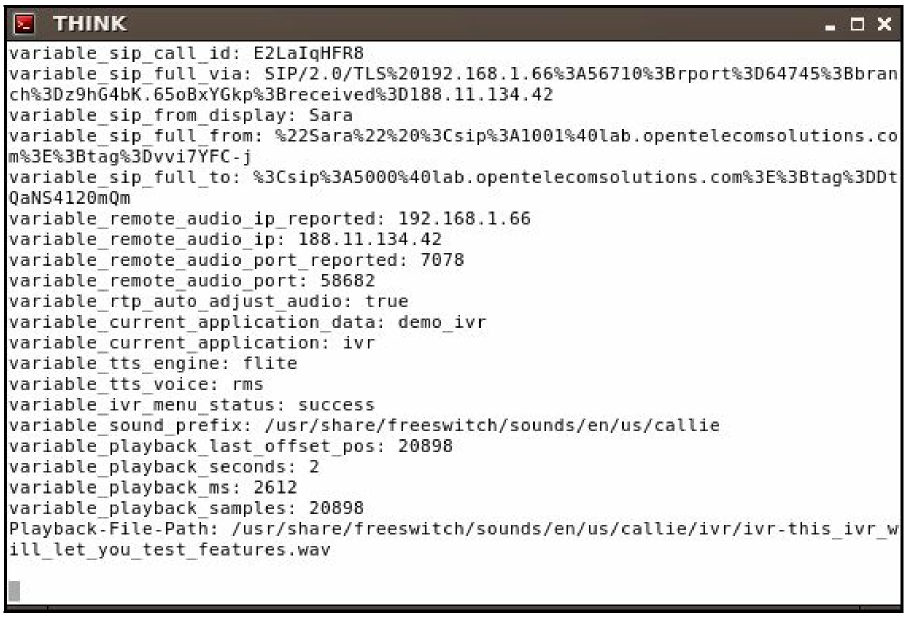
```
telnet 127.0.0.1 8021
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Content-Type: auth/request

auth ClueCon

Content-Type: command/reply
Reply-Text: +OK accepted

events plain all

Content-Type: command/reply
Reply-Text: +OK event listener enabled plain
```

From now on, we'll see the events zapping through our terminal. On an idle FreeSWITCH server, we'll see mainly HEARTBEAT and RE_SCHEDULE events. On a busy system, we'll not be able to discern in the fast flow. Remember, to escape from the telnet connection type "Ctrl-]", and then "quit" to close it.



```
Task-ID: 2
Task-Desc: check_ip
Task-Group: core
Task-Runtime: 1498597131

Content-Length: 602
Content-Type: text/event-plain

Event-Name: RE_SCHEDULE
Core-UUID: 6ddf010e-bc0b-4366-8413-6becd7f77076
FreeSWITCH-Hostname: lxc111
FreeSWITCH-Switchname: lxc111
FreeSWITCH-IPv4: 192.168.1.111
FreeSWITCH-IPv6: 2001%3Ab07%3A5d32%3A860b%3Aa067%3A61ff%3Afe4f%3A2119
Event-Date-Local: 2017-06-27%2020%3A57%3A53
Event-Date-GMT: Tue,%2027%20Jun%202017%2020%3A57%3A53%20GMT
Event-Date-Timestamp: 1498597073283005
Event-Calling-File: switch_scheduler.c
Event-Calling-Function: switch_scheduler_execute
Event-Calling-Line-Number: 71
Event-Sequence: 37014
Task-ID: 3
Task-Desc: limit_hash_cleanup
Task-Group: mod_hash
Task-Runtime: 1498597973
```

Obviously, it becomes much more interesting if we watch it during a call (use a terminal with a very big scrollback buffer, or use the "screen" utility, so you can seek up and down the events):



```
variable_sip_call_id: E2LaIqHFR8
variable_sip_full_via: SIP/2.0/TLS%20192.168.1.66%3A56710%3Brport%3D64745%3Bbran
ch%3Dz9hG4bK.65oBxYGkp%3Breceived%3D188.11.134.42
variable_sip_from_display: Sara
variable_sip_full_from: %22Sara%22%20%3Csip%3A1001%40lab.opentelecomsolutions.co
m%3E%3Btag%3Dvvi7YFC-j
variable_sip_full_to: %3Csip%3A5000%40lab.opentelecomsolutions.com%3E%3Btag%3DDt
QaNS4120mQm
variable_remote_audio_ip_reported: 192.168.1.66
variable_remote_audio_ip: 188.11.134.42
variable_remote_audio_port_reported: 7078
variable_remote_audio_port: 58682
variable_rtp_auto_adjust_audio: true
variable_current_application_data: demo_ivr
variable_current_application: ivr
variable_tts_engine: flite
variable_tts_voice: rms
variable_ivr_menu_status: success
variable_sound_prefix: /usr/share/freeswitch/sounds/en/us/callie
variable_playback_last_offset_pos: 20898
variable_playback_seconds: 2
variable_playback_ms: 2612
variable_playback_samples: 20898
Playback-File-Path: /usr/share/freeswitch/sounds/en/us/callie/ivr/ivr-this_ivr_w
ill_let_you_test_features.wav
```

Later, and in examples, we'll see how to send commands, inject events, and only see the event types we're interested in.

Let's see a quick example on how to send an event into the Event System bus. We first use the command "**sendevent**" with a **message type**, and **then we add the headers** needed by that event type (default headers are automatically added by FreeSWITCH), and last we type the **body**, followed by a double carriage return. In bold what we type, after telnetting to localhost port 8021 (hit "Enter" twice after "ClueCon" and after "World"):

```
auth ClueCon

Content-Type: command/reply
Reply-Text: +OK accepted

sendevent SEND_MESSAGE
Content-Type: text/plain
User: 1001
Profile: internal
Host: lab.opentelecomsolutions.com
Content-Lenght: 11
Hello World

Content-Type: command/reply
Reply-Text: +OK df178dbd-9b9a-4b48-bcc0-440737e4a2f6
```

This event, of type MESSAGE, will send a SIP SIMPLE chat message to phones registered as "1001" user on our "lab.opentelecomsolutions.com" domain. Linphone and almost all other SIP softphones can display SIP chat messages.
Remember, to exit your established telnet session, use the "Ctrl-]" key sequence, followed by "quit".

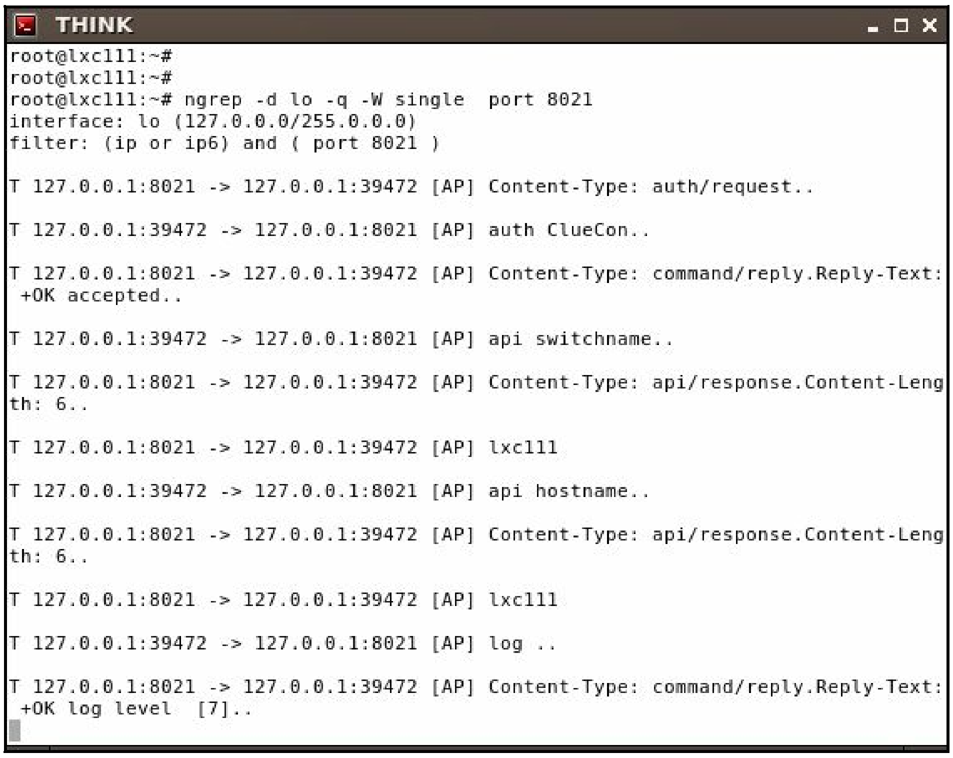
fs_cli

Surprise, surprise, fs_cli is an application that connects via TCP to mod_event_socket, on localhost port 8021. After connecting, fs_cli sends some commands and then enables receiving "logging" events.

Let's see it in action, on port 8021! Disconnect all fs_cli sessions you may have running and any other application that connects to FreeSWITCH (they're probably using this same interface), and then you can start sniffing with ngrep:

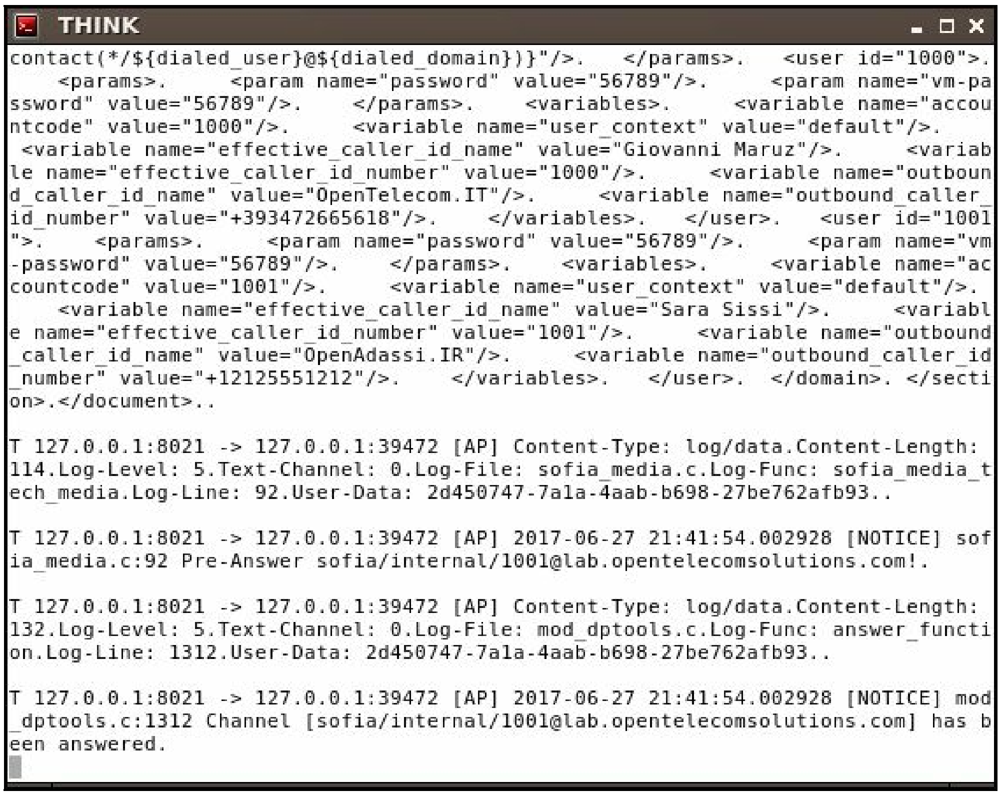
```
ngrep -d lo -q -W single port 8021
```

In another terminal, execute fs_cli. Ngrep will display the connection authorization, the "api" commands, and then the "log" command that enable displaying of "logging" events.



```
root@lxc111:~#
root@lxc111:~#
root@lxc111:~# ngrep -d lo -q -W single  port 8021
interface: lo (127.0.0.0/255.0.0.0)
filter: (ip or ip6) and ( port 8021 )

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] Content-Type: auth/request..

T 127.0.0.1:39472 -> 127.0.0.1:8021 [AP] auth ClueCon..

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] Content-Type: command/reply.Reply-Text:
 +OK accepted..

T 127.0.0.1:39472 -> 127.0.0.1:8021 [AP] api switchname..

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] Content-Type: api/response.Content-Leng
th: 6..

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] lxc111

T 127.0.0.1:39472 -> 127.0.0.1:8021 [AP] api hostname..

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] Content-Type: api/response.Content-Leng
th: 6..

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] lxc111

T 127.0.0.1:39472 -> 127.0.0.1:8021 [AP] log ..

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] Content-Type: command/reply.Reply-Text:
 +OK log level  [7]..
```

Then make a call to 5000 from a registered phone, and see what's displayed by ngrep:



```
contact(*/${dialed_user}@${dialed_domain})}"/>.   </params>.   <user id="1000">.
    <params>.     <param name="password" value="56789"/>.     <param name="vm-pa
ssword" value="56789"/>.    </params>.    <variables>.     <variable name="accou
ntcode" value="1000"/>.     <variable name="user_context" value="default"/>.
 <variable name="effective_caller_id_name" value="Giovanni Maruz"/>.     <variab
le name="effective_caller_id_number" value="1000"/>.     <variable name="outboun
d_caller_id_name" value="OpenTelecom.IT"/>.     <variable name="outbound_caller_
id_number" value="+393472665618"/>.    </variables>.   </user>.   <user id="1001
">.    <params>.     <param name="password" value="56789"/>.     <param name="vm
-password" value="56789"/>.    </params>.    <variables>.     <variable name="ac
countcode" value="1001"/>.     <variable name="user_context" value="default"/>.
    <variable name="effective_caller_id_name" value="Sara Sissi"/>.     <variabl
e name="effective_caller_id_number" value="1001"/>.     <variable name="outbound
_caller_id_name" value="OpenAdassi.IR"/>.     <variable name="outbound_caller_id
_number" value="+12125551212"/>.    </variables>.   </user>.  </domain>. </secti
on>.</document>..

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] Content-Type: log/data.Content-Length:
114.Log-Level: 5.Text-Channel: 0.Log-File: sofia_media.c.Log-Func: sofia_media_t
ech_media.Log-Line: 92.User-Data: 2d450747-7a1a-4aab-b698-27be762afb93..

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] 2017-06-27 21:41:54.002928 [NOTICE] sof
ia_media.c:92 Pre-Answer sofia/internal/1001@lab.opentelecomsolutions.com!.

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] Content-Type: log/data.Content-Length:
132.Log-Level: 5.Text-Channel: 0.Log-File: mod_dptools.c.Log-Func: answer_functi
on.Log-Line: 1312.User-Data: 2d450747-7a1a-4aab-b698-27be762afb93..

T 127.0.0.1:8021 -> 127.0.0.1:39472 [AP] 2017-06-27 21:41:54.002928 [NOTICE] mod
_dptools.c:1312 Channel [sofia/internal/1001@lab.opentelecomsolutions.com] has b
een answered.
```

Event System Commands

Once connected to a mod_event_socket port, or actually once connected in any way (for example, via other events handler modules) to the Event System, you can interact via simple commands.

A lot of information is exposed online in our http://freeswitch.org/confluence, search for "mod_event_socket", at paragraph "Command Documentation".

Let's see the most important, and don't forget to **terminate the command, when all data has been entered (all additional headers, and so on) with double CR/LF (that is, double "Enter")**.

api

Send an api command (blocking mode) and wait for completion:

```
api <command><arg>
```

Api commands are all the ones you can execute typing at the FreeSWITCH console.

Examples:

```
api originate sofia/mydomain.com/ext@yourvsp.com 1000
api sleep 5000
```

bgapi

Send an api command (non-blocking mode) this will let you execute a job in the background and the result will be sent as an event with an indicated uuid to match the reply to the command):

```
bgapi <command><arg>
```

The same API commands are available as with the **api** command, however, the server returns immediately and is available for processing more commands.

Example return value:

```
Content-Type: command/reply
Reply-Text: +OK Job-UUID: c7709e9c-1517-11dc-842a-d3a3942d3d63
```

When the command is done executing, FreeSWITCH fires an event with the result and you can compare that to the Job-UUID to see what the result was. In order to receive this event, you will need to subscribe to BACKGROUND_JOB events.

If you want to set your own custom Job-UUID over plain socket:

```
bgapi status
Job-UUID: d8c7f660-37a6-4e73-9170-1a731c442148
```

Reply:

```
Content-Type: command/reply
Reply-Text: +OK Job-UUID: d8c7f660-37a6-4e73-9170-1a731c442148
Job-UUID: d8c7f660-37a6-4e73-9170-1a731c442148
```

event

Enable or disable events by class or all (plain or xml or json output format):

```
event plain <listof events to log or all>
event xml <listof events to log or all>
event json <listof events to log or all>
```

The event commands are used to subscribe on events from FreeSWITCH. You may specify any number of events on the same line; they should be separated with space.

Examples:

```
event plain ALL
event plain CHANNEL_CREATE CHANNEL_DESTROY CUSTOM conference::maintenance sofia::register sof
event xml ALL
event json CHANNEL_ANSWER
```

Subsequent calls to 'event' won't override the previous event sets. Supposing, you've first registered for DTMF:

```
event plain DTMF
```

then you may want to register for CHANNEL_ANSWER too; it is enough to give:

```
event plain CHANNEL_ANSWER
```

and you will continue to receive DTMF along with CHANNEL_ANSWER, later one doesn't override the previous.

filter

Specify event types to listen for. Note that this is not a filter out, but rather a "filter in," that is, when a filter is applied only the filtered values are received. Multiple filters on a socket connection are allowed.

Usage:

```
filter <EventHeader><ValueToFilter>
```

Example:

The following example will subscribe to all events and then create two filters, one to listen for HEARTBEATS and one to listen for CHANNEL_EXECUTE events:

```
events plain all
filter Event-Name CHANNEL_EXECUTE
filter Event-Name HEARTBEAT
```

Now only HEARTBEAT and CHANNEL_EXECUTE events will be received. You can filter on any of the event headers. To filter for a specific channel you will need to use the uuid:

```
filter Unique-ID d29a070f-40ff-43d8-8b9d-d369b2389dfe
```

To filter multiple unique IDs, you can just add another filter for events for each UUID. This can be useful for example if you want to receive start/stop-talking events for multiple users on a particular conference:

```
filter plain all
filter plain CUSTOM conference::maintenance
filter Unique-ID $participantB
filter Unique-ID $participantA
filter Unique-ID $participantC
```

This will give you events for Participant A, B, and C on any conference. To receive events for all users on a conference you can use something like:

```
filter Conference-Unique-ID $ConfUUID
```

You can filter on any of the parameters you get in a freeSWITCH event:

```
filter plain all
filter call-direction Inbound
filter Event-Calling-File mod_conference.c
filter Conference-Unique-ID $ConfUUID
```

You can use them individually or compound them depending on whatever end result you desire for the type of events you want to receive

filter delete

Specify the events that you want to revoke the filter. "filter delete" can be used when some filters are applied wrongly or when there is no use of the filter.

Usage:

```
filter delete <EventHeader><ValueToFilter>
```

Example:

```
filter delete Event-Name HEARTBEAT
```

Now, you will no longer receive HEARTBEAT events.

```
filter delete Unique-ID d29a070f-40ff-43d8-8b9d-d369b2389dfe
```

This is to delete the filter that was applied for the given unique-id. After this, you won't receive any events for this unique-id.

```
filter delete Unique-ID
```

This deletes all the filters that were applied based on any unique-id.

sendevent

Send an event into the event system (multi line input for headers):

```
sendevent <event-name>
```

Example of sendevent with a message body, the length of the body is specified by content-length:

```
sendevent SEND_MESSAGE
Content-Type: text/plain
User: 1001
Profile: internal
Host: lab.opentelecomsolutions.com
Content-Lenght: 11
Hello World
```

exit

```
exit
```

Close the socket connection.

auth

```
auth <password>
```

You must send this command immediately after a TCP connection.

log

```
log <level>
```

Enable displaying of logging events. Level has the same values as in the console, for example: 7=DEBUG 3=ERR, and so on.

nolog

```
nolog
```

Disables logging events output previously enabled by the log command.

nixevent

```
nixevent <event types | ALL  | CUSTOM custom event sub-class>
```

Suppress the specified type of event. Useful when you want to allow 'event all' followed by 'nixevent <some_event>' to see all but one type of event.

noevents

```
noevents
```

Disable all events that were previously enabled with event.

sendmsg

```
sendmsg <uuid>
```

Control an existing call of given uuid

Needs the additional header "**call-command**", which can have value **execute** or **hangup**.

Examples:

```
sendmsg ec96830d-0ad5-4a55-afbd-11bb625db2d0
call-command: execute
execute-app-name: playback
execute-app-arg: /tmp/ivr-wakey_wakey_sunshine.wav

sendmsg ec96830d-0ad5-4a55-afbd-11bb625db2d0
call-command: hangup
hangup-cause: USER_BUSY
```

call-command

This event header can have one of the following two values:

execute

Execute dialplan applications. Need the additional header "execute-app-name", and the optional headers "execute-app-arg" and "loops" (how many time to repeat the command).

hangup

Hang up the call. Allows for the optional header "hangup-cause".

Outbound Socket - call connects to you

The dialplan application "socket" makes an outbound TCP connection to the specified ip:port and the other end can control the call exactly in the same way as in the inbound connection to mod_event_socket.

You must have a "server" application running, listening to the correct IP address and port, and ready to accept inbound connections from the "socket" dialplan application. Once the TCP connection is established, then commands and output are the same as we saw before.

When you call outbound socket, FreeSWITCH automatically puts the call in parked state.

The syntax for calling socket from the Dialplan is:

```
<ip>:<port> [<keywords>]
```

The following are examples of how to use it in the Dialplan:

```
<action application="socket" data="127.0.0.1:8084"/>
<action application="socket" data="127.0.0.1:8084 async"/>
<action application="socket" data="127.0.0.1:8084 full"/>
<action application="socket"
data="127.0.0.1:8084 async full"/>
```

The optional keywords async and full modify the behavior as follows:

- **async**: The async keyword indicates that all commands will return instantly, making it possible to monitor the socket for events while the stack of commands are executing. If the async keyword is absent, then each event socket command will block until it has finished.
- **full**: The full keyword indicates that the other end will have the full command set for event socket. This is the same command set an inbound event socket connection has, so you can execute API commands, get global events, and so on. If the full keyword is absent, then the command set and events are limited to that particular call. In other words, if full is not specified, then the commands sent on this socket connection can affect only the channel currently being processed. Likewise, the socket connection will only receive events related to this particular channel.

ESL - Event Socket Library

Actually I would not counsel to go the hard and low level way to interact with FreeSWITCH Event System: we've put a lot of effort to provide you with a higher level library, available for a lot of scripting languages and for C/C++.

ESL will save you from a lot of tedious tasks, such as managing the TCP connection, parsing the events, and so on. Many useful and popular functions are already there, so you don't have to reinvent the wheel.

ESL supported languages

ESL has been developed using only one set of SWIG interfaces, so variables, methods, and functions have the same names and arguments whatever the programming language.

- java
- lua
- managed (.NET)
- perl
- php
- python
- ruby
- tcl
- C

ESL Reference

Following are the most important functions and their arguments, as per how they are available in all ESL implementations for the various programming languages.

eslSetLogLevel

```
eslSetLogLevel($loglevel)
```

Calling this function within your program causes your program to issue informative messages related to the Event Socket Library on STDOUT.

The values for $loglevel are the usual:

- 0 is EMERG
- 1 is ALERT
- 2 is CRIT
- 3 is ERROR
- 4 is WARNING
- 5 is NOTICE
- 6 is INFO
- 7 is DEBUG

ESLevent Object

This is the object that represent an Event, can be created (with new()) or received. Its methods are as follows:

new

```
new($event_type [, $event_subclass])
```

Instantiates a new event object of that type. Optionally, in case of CUSTOM and other type of events, you specify the subclass.

serialize

```
serialize([$format])
```

Print the event into colon-separated 'name: value' pairs similar to a sip/email packet (the way it looks on '/events plain all').

$format can be:

- "xml"
- "json"
- "plain" (default)

setPriority

```
setPriority([$number])
```

Sets the priority of the event to $number.

getHeader

```
getHeader($header_name)
```

Gets the value of the header $header_name.

getBody

```
getBody()
```

Gets the body of the event.

getType

```
getType()
```

Gets the event type.

addBody

```
addBody($value)
```

Add $value to the body of the event. This can be called multiple times.

addHeader

```
addHeader($header_name, $value)
```

Add a header with key = $header_name and value = $value to an event. Can be called multiple times.

delHeader

```
delHeader($header_name)
```

Delete the header with key $header_name.

firstHeader

```
firstHeader()
```

Sets the pointer to the first header in an event object, and returns its key name. This must be called before nextHeader is called.

nextHeader

```
nextHeader()
```

Moves to the next header in an event, and returns its key name. firstHeader must be called before this method. If you're already on the last header when this method is called, then it will return NULL.

ESLconnection Object

This is the object that represent the network connection to FreeSWITCH listening socket, can be created with new(). Its methods are as follows:

new

```
new($host, $port, $password)
```

Initializes a new instance of ESLconnection, connects to the host $host on the port $port, and supplies $password to FreeSWITCH.

Intended only for an event socket in "**Inbound**" mode. In other words, this is only intended for the purpose of creating a connection to FreeSWITCH that is not initially bound to any particular call or channel.

Does not initialize channel information (since inbound connections are not bound to a particular channel). In plain language, this means that calls to getInfo() will always return NULL:

```
new($fd)
```

Initializes a new instance of ESLconnection, using the existing file number contained in $fd.

Intended only for Event Socket **Outbound** connections. It will fail on Inbound connections, even if passed a valid inbound socket.

The standard method for using this function is to listen for an incoming connection on a socket, accept the incoming connection from FreeSWITCH, fork a new copy of your process if you want to listen for more connections, and then pass the file number of the socket to new($fd).

socketDescriptor

```
socketDescriptor()
```

Returns the UNIX file descriptor for the connection object, if the connection object is connected. This is the same file descriptor that was passed to new($fd) when used in outbound mode.

connected

```
connected()
```

Test if the connection object is connected. Returns 1 if connected, 0 otherwise.

getInfo

```
getInfo()
```

When FreeSWITCH connects to an "Event Socket Outbound" handler, it sends a "CHANNEL_DATA" event as the first event after the initial connection. getInfo() returns an ESLevent that contains this Channel Data.

getInfo() returns NULL when used on an "Event Socket Inbound" connection.

send

```
send($command)
```

Sends a command to FreeSWITCH.

Does not wait for a reply. You should immediately call recvEvent or recvEventTimed in a loop until you get the reply. The reply event will have a header named "content-type" that has a value of "api/response" or "command/reply".

To automatically wait for the reply event, use sendRecv() instead of send().

sendRecv

```
sendRecv($command)
```

Internally sendRecv($command) calls send($command) then recvEvent(), and returns an instance of ESLevent.

recvEvent() is called in a loop until it receives an event with a header named "content-type" that has a value of "api/response" or "command/reply", and then returns it as an instance of ESLevent.

Any events that are received by recvEvent() prior to the reply event are queued up, and will get returned on subsequent calls to recvEvent() in your program.

api

```
api($command[, $arguments])
```

Send an API command to the FreeSWITCH server. This method blocks further execution until the command has been executed.

api($command, $args) is identical to sendRecv("api $command $args").

bgapi

```
bgapi($command[, $arguments][,$custom_job_uuid])
```

Send a background API command to the FreeSWITCH server to be executed in its own thread. This will be executed in its own thread, and is non-blocking.

bgapi($command, $args) is identical to sendRecv("bgapi $command $args")

Here is a Perl snippet that demonstrates the custom Job-UUID setting (works for all swigged languages):

```
my $command = shift;
my $args = join("", @ARGV);

my $con = new ESL::ESLconnection("127.0.0.1", "8021", "ClueCon");

my $e = $con->bgapi($command, $args, "my-job-id");
print $e->serialize("json");
```

This code returns:

```
{
"Event-Name":   "SOCKET_DATA",
"Content-Type": "command/reply",
"Reply-Text":   "+OK Job-UUID: my-job-id",
"Job-UUID":     "my-job-id"
}
```

sendEvent

```
sendEvent($event)
```

Inject an event into the FreeSWITCH Event System.

recvEvent

```
recvEvent()
```

Returns the next event from FreeSWITCH. If no events are waiting, this call will block until an event arrives.

If any events were queued during a call to sendRecv(), then the first one will be returned and removed from the queue, then the next event will be read from the connection in sequence.

recvEventTimed

```
recvEventTimed($milliseconds)
```

Similar to recvEvent(), except that it will block for at most $milliseconds.

A call to recvEventTimed(0) will return immediately. This is useful for polling for events.

filter

```
filter($header, $value)
```

Same as the "filter" Event System command.

events

```
events($event_type, $value)
```

Same as the "events" Event System command.

execute

```
execute($app[, $arg][, $uuid])
```

Execute a dialplan application, and wait for a response from the server. On socket connections not anchored to a channel (most of the time inbound), all three arguments are required -- $uuid specifies the channel to execute the application on.

Returns an ESLevent object containing the response from the server. The getHeader("Reply-Text") method of this ESLevent object returns the server's response. The server's response will contain "+OK [Success Message]" on success or "-ERR [Error Message]" on failure.

executeAsync

```
executeAsync($app[, $arg][, $uuid])
```

Same as execute, but doesn't wait for a response from the server.

This works by causing the underlying call to execute() to append "async: true" header in the message sent to the channel.

setAsyncExecute

```
setAsyncExecute($value)
```

Force async mode on for a socket connection. This command has no effect on outbound socket connections that are set to "async" in the dialplan and inbound socket connections, since these connections are already set to async mode on.

$value should be 1 to force async mode, and 0 not to force it.

Specifically, calling setAsyncExecute(1) operates by causing future calls to execute() to include the "async: true" header in the message sent to the channel. Other event socket library routines are not affected by this call.

setEventLock

```
setEventLock($value)
```

Force sync mode on for a socket connection. This command has no effect on outbound socket connections that are not set to "async" in the dialplan, since these connections are already set to sync mode.

$value should be 1 to force sync mode, and 0 not to force it.

Specifically, calling setEventLock(1) operates by causing future calls to execute() to include the "event-lock: true" header in the message sent to the channel. Other event socket library routines are not affected by this call.

disconnect

```
disconnect()
```

Close the socket connection to the FreeSWITCH server.

ESL installation

ESL is not installed by default. You must explicitly install it from packages, or compile and install it from source.

If you're installing from Debian packages, then you want to install libesl-perl, python-esl, freeswitch-mod-perl, freeswitch-mod-python, and so on.

If you're compiling from sources, then you want to check on out http://freeswitch.org/confluence and search for "Event Socket Library" for latest dependencies and installation instructions.

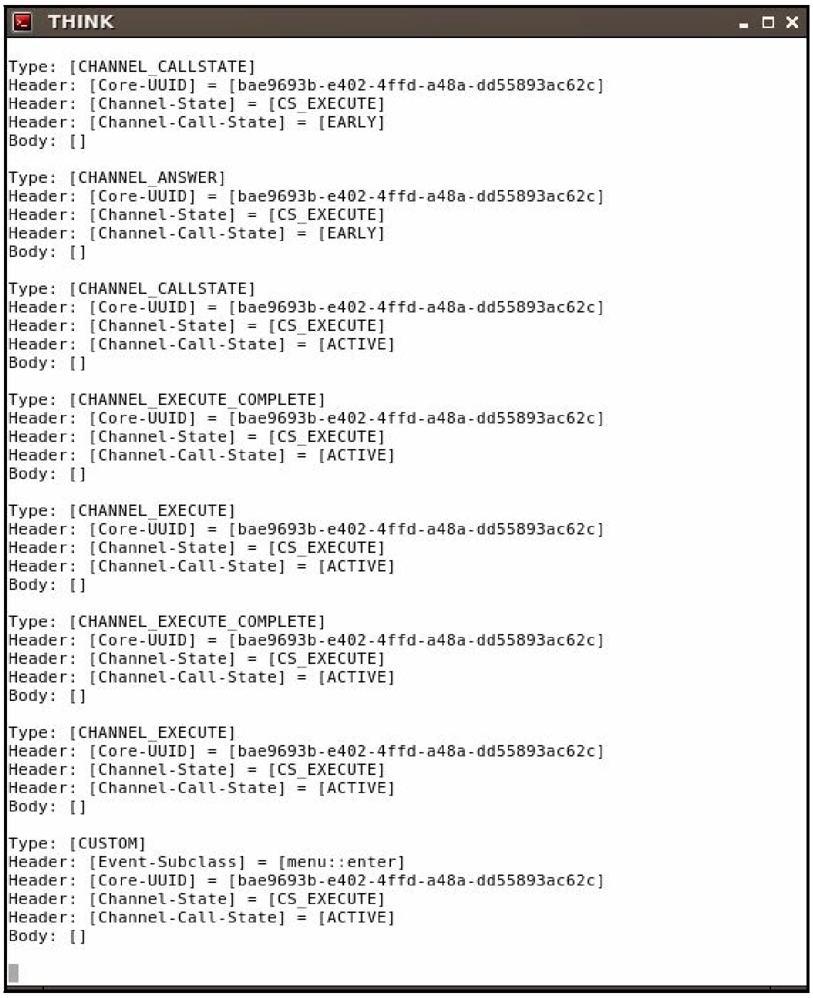
ESL: Examples in Perl

Lets see how to use the Perl ESL implementation to connect and interact with the Event System in a FreeSWITCH server.

Display Filtered Events

A command line utility that connects to FreeSWITCH listening socket, subscribe to all events (don't do this on a busy server), and then display only selected types, further filtered on values in chosen headers.

```
#!/usr/bin/perl
require ESL;

my $type = ".*";
my $header = ".*";
my $value = ".*";

#$type = "CUSTOM|CHANNEL.*|MESSAGE";
#$header = "Core-UUID|Event-Subclass|Channel-State|Channel-Call-State";
#$value = "HANGUP|ACTIVE|DOWN|CS_EXECUTE";

my $con = new ESL::ESLconnection("localhost", "8021", "ClueCon");
$con->events("plain", "all");

while($con->connected()) {
my $e = $con->recvEvent();
if ($e) {
if(! ($e->getType() =~ /^($type)$/) ) {
next;
}
printf "Type: [%s]\n", $e->getType();
my $h = $e->firstHeader();
while ($h) {
if( ( $h =~ /^($header)$/) && ($e->getHeader($h) =~ /^($value)$/) ){
printf "Header: [%s] = [%s]\n", $h, $e->getHeader($h);
}
$h = $e->nextHeader();
}
printf "Body: [%s]\n\n", $e->getBody;
}
}
```



```
Type: [CHANNEL_CALLSTATE]
Header: [Core-UUID] = [bae9693b-e402-4ffd-a48a-dd55893ac62c]
Header: [Channel-State] = [CS_EXECUTE]
Header: [Channel-Call-State] = [EARLY]
Body: []

Type: [CHANNEL_ANSWER]
Header: [Core-UUID] = [bae9693b-e402-4ffd-a48a-dd55893ac62c]
Header: [Channel-State] = [CS_EXECUTE]
Header: [Channel-Call-State] = [EARLY]
Body: []

Type: [CHANNEL_CALLSTATE]
Header: [Core-UUID] = [bae9693b-e402-4ffd-a48a-dd55893ac62c]
Header: [Channel-State] = [CS_EXECUTE]
Header: [Channel-Call-State] = [ACTIVE]
Body: []

Type: [CHANNEL_EXECUTE_COMPLETE]
Header: [Core-UUID] = [bae9693b-e402-4ffd-a48a-dd55893ac62c]
Header: [Channel-State] = [CS_EXECUTE]
Header: [Channel-Call-State] = [ACTIVE]
Body: []

Type: [CHANNEL_EXECUTE]
Header: [Core-UUID] = [bae9693b-e402-4ffd-a48a-dd55893ac62c]
Header: [Channel-State] = [CS_EXECUTE]
Header: [Channel-Call-State] = [ACTIVE]
Body: []

Type: [CHANNEL_EXECUTE_COMPLETE]
Header: [Core-UUID] = [bae9693b-e402-4ffd-a48a-dd55893ac62c]
Header: [Channel-State] = [CS_EXECUTE]
Header: [Channel-Call-State] = [ACTIVE]
Body: []

Type: [CHANNEL_EXECUTE]
Header: [Core-UUID] = [bae9693b-e402-4ffd-a48a-dd55893ac62c]
Header: [Channel-State] = [CS_EXECUTE]
Header: [Channel-Call-State] = [ACTIVE]
Body: []

Type: [CUSTOM]
Header: [Event-Subclass] = [menu::enter]
Header: [Core-UUID] = [bae9693b-e402-4ffd-a48a-dd55893ac62c]
Header: [Channel-State] = [CS_EXECUTE]
Header: [Channel-Call-State] = [ACTIVE]
Body: []
```

Use "api" and "bgapi"

Let's remember: "api" blocks and wait for result, "bgapi" returns immediately giving you an uuid you will use to recognize the results.

```
#!/usr/bin/perl
require ESL;

my $con = ESL::ESLconnection->new("localhost", "8021", "ClueCon");

$con->api('chat', "sip|1011\@lab.opentelecomsolutions.com|1001\@lab.opentelecomsolutions.com|
$con->bgapi('chat', "sip|1011\@lab.opentelecomsolutions.com|1001\@lab.opentelecomsolutions.cc
```

Use "sendEvent"

How to fire an event.

```
#!/usr/bin/perl
require ESL;

my $con = ESL::ESLconnection->new("localhost", "8021", "ClueCon");
my $e = ESL::ESLevent->new("SEND_MESSAGE");

$e->addHeader("Content-Type", "text/plain");
$e->addHeader("User", '1001');
$e->addHeader("profile", 'internal');
$e->addHeader("host", 'lab.opentelecomsolutions.com');
$e->addBody("ciao a tutti3");
$con->sendEvent($e);
```

Summary

In this chapter, we saw how to use FreeSWITCH with the maximum possible control: we tapped into its nervous system, into its internal "Internet of Things", and we were able to use its features as a real-time toolbox.

The two ways to connect to Event System, inbound and outbound, complement themselves: use inbound with a supervising client application that governs many calls; use outbound if you want your server application to control just one call.

We just scratched the surface, but the feeling is the right one: we can actually do the most complex and crazy things in this way. No need to walk the beaten path.

In the next chapter, we'll see another way to control FreeSWITCH, on a much higher level of abstraction: mod_httapi allows your application to interact via HTTP.

HTTAPI - FreeSWITCH Asks Webserver Next Action

HTTAPI allows for FreeSWITCH to ask a webserver what to do to/with a call, and after executing, ask the webserver again. So, is actually a hyper dynamic protocol, where FreeSWITCH sends the webserver all info about a call and the results of last action(s), and webserver reply with what to do as next step. Then the cycle repeat until hangup or the call has been transferred. The obvious use cases are IVRs, but other kind of applications may enjoy this approach.

In this chapter we will discuss:

- The HTTAPI main concepts
- HTTAPI dialplan action
- The structure of the document returned by webserver
- Configuration od mod_httapi
- A sample PHP library that makes HTTAPI applications easier to develop

HTTAPI Rationale

HTTAPI allows for direct **control** of the **call**, in dynamic and real time way, **step by step (action by action)**, as opposed to dialplan or mod_xml_curl that define all the steps which will be executed by one extension.

When the **httpapi dialplan action** is executed, **FreeSWITCH** will **make** an **HTTP request** to a configured webserver. In this HTTP request FreeSWITCH will send to the webserver **informations about the call**, and other variables and parameters.

Webserver will **answer** to FreeSWITCH with a short **XML HTTPAPI document** containing **what FreeSWITCH must do** with/to the call.

FreeSWITCH will execute it, **and** then will **again make a HTTP request** to the webserver, like the previous one.

The cycle repeat itself until the call is hangup or transferred.

HTTAPI dialplan

mod_httapi is invoked by the "httapi" action in dialplan.

```
<extension name="myhttapi">
  <condition field="destination_number" expression="^12345$">
    <action application="answer"/>
    <action application="httapi"/>
  </condition>
</extension>
```

data

We can pass to "httapi" the optional "data" argument. In "data" we can directly (eg, without curly braces) write the URL to be used as gateway, bypassing the one set in module configuration.

Or we can list in curly braces, separated by commas, 3 possible parameters to bypass the ones set by the configuration profile in use (same technique as in dialstring).

```
<action application="httapi" data="http://localhost/freeswitch" />
```

httapi_profile

Use this parameter to choose which configuration profile to use, bypassing the one set as default in configuration file. If this parameter is not used, and no default is set in configuration file, the profile named "default" will be used

```
<action application="httapi" data="{httpapi_profile=myprofile}" />
```

url

Use this parameter to bypass the **gateway-url** set by the configuration profile in use

```
<action application="httapi" data="{httpapi_profile=myprofile,url=http://my.webserver.com/dir
```

method

Use this parameter to bypass the method set by the configuration profile in use

```
<action application="httapi" data="{httpapi_profile=myprofile,url=http://my.webserver.com/di
```

HTTAPI document syntax

In its most basic form an XML HTTAPI document sent by the webserver in response to mod_httapi request looks like this:

```
<document type="text/freeswitch-httapi">
 <params/>
 <variables/>
 <work/>
</document>
```

As an example, this is the document returned from webserver when pressing "6" on the keypad while in a call to demo ivr in php httpapi, detailed later in this chapter:

```
<document type="text/freeswitch-httapi">
  <variables>
    <IVR_variable_01>VariableValue01</IVR_variable_01>
  </variables>
  <params>
    <IVR_param_01>ParamValue01</IVR_param_01>
  </params>
  <variables>
    <main_menu_option>6</main_menu_option>
  </variables>
  <work>
    <playback error-file="ivr/ivr-that_was_an_invalid_entry.wav" loops="3" digit-timeout="15(
      <bind>*</bind>
    </playback>
    <!-- session_id => 9c6f38d3-897a-44aa-9162-bf4f718c6d45 -->
    <!-- hostname => ip-172-31-11-17 -->
    <!-- Caller-Direction => inbound -->
    <!-- Caller-Logical-Direction => inbound -->
    <!-- Caller-Username => 1010 -->
    <!-- Caller-Dialplan => XML -->
    <!-- Caller-Caller-ID-Name => 1010 -->
    <!-- Caller-Caller-ID-Number => 1010 -->
    <!-- Caller-Orig-Caller-ID-Name => 1010 -->
    <!-- Caller-Orig-Caller-ID-Number => 1010 -->
    <!-- Caller-Network-Addr => 188.11.134.42 -->
    <!-- Caller-ANI => 1010 -->
    <!-- Caller-Destination-Number => 12345 -->
    <!-- Caller-Unique-ID => 9c6f38d3-897a-44aa-9162-bf4f718c6d45 -->
    <!-- Caller-Source => mod_sofia -->
    <!-- Caller-Context => default -->
    <!-- Caller-Channel-Name => sofia/internal/1010@52.57.248.151 -->
    <!-- Caller-Profile-Index => 1 -->
    <!-- Caller-Profile-Created-Time => 1498838395878804 -->
    <!-- Caller-Channel-Created-Time => 1498838395878804 -->
    <!-- Caller-Channel-Answered-Time => 1498838395898804 -->
    <!-- Caller-Channel-Progress-Time => 0 -->
    <!-- Caller-Channel-Progress-Media-Time => 1498838395898804 -->
    <!-- Caller-Channel-Hangup-Time => 0 -->
    <!-- Caller-Channel-Transfer-Time => 0 -->
    <!-- Caller-Channel-Resurrect-Time => 0 -->
    <!-- Caller-Channel-Bridged-Time => 0 -->
    <!-- Caller-Channel-Last-Hold => 0 -->
    <!-- Caller-Channel-Hold-Accum => 0 -->
    <!-- Caller-Screen-Bit => true -->
    <!-- Caller-Privacy-Hide-Name => false -->
    <!-- Caller-Privacy-Hide-Number => false -->
    <!-- url => http://localhost/phttapi/book/demo-ivr.php -->
    <!-- IVR_param_01 => ParamValue01 -->
    <!-- main_menu_option => 6 -->
    <!-- input_type => dtmf -->
</work>
</document>
```

An HTTAPI response **must** have an HTTP MIME **content-type** of **text/xml** (webserver must be configured to send this content-type). All HTTAPI responses **must** include the **document tag** with the **type** attribute of **text/freeswitch-httapi**. Then, may have any one, or all, of the following tags:

- **params**: These are params (that is, "parameters") the script asks FreeSWITCH to pass back to the web server on each request. You use the <params> tag to tell FreeSWITCH to pass custom POST params.
- **variables**: The <variables> tag allows you to set channel variables that can be used by the FreeSWITCH Dialplan or read back into httapi on subsequent requests.
- **work**: This is where most of the interesting stuff happens. There are many different action tags that can be used as children of the <work> tag to make FreeSWITCH do just about anything with the phone call being controlled: logging a message at the console, playing a sound file, doing Automatic Speech Recognition, collecting DTMF keypresses, and so on. The available action tags and the attributes that correspond to each action are covered in detail in the next section.

Work actions

The HTTAPI work actions are described in this section. In the following definitions, *DATA* is the content of the tag (that is, <tag>*DATA*</tag>).

All work actions have two tags that are always available:

- action: Changes the new default target URL.
- temp-action: Changes target URL to submit the next request. Subsequent requests will use the default URL or whatever is specified in the action tag.

Several of the actions can contain one or more **bind** tags that function in similar fashion to bind_digit_action

```
<bind action strip>*EXPR*</bind>
ATTRS:
action              : a specific url to go to next if the binding is dialed
strip               : a character to strip from the result string, such as #
```

The following is a list of work actions and their descriptions:

playback

```
<playback file name error-file action digit-timeout input-timeout loops asr-engine asr-gramma
```

playback plays a file and optionally collects input. It has the following attributes:

- file: The path to the file to play
- name: Param name to save result
- error-file: Error file to play on invalid input
- digit-timeout: Timeout waiting for digits after file plays (when input bindings are present)
- input-timeout: Timeout waiting for more digits in a multi-digit input
- loops: Maximum number of times to play the file (when input bindings are present)
- asr-engine: Automated Speech Recognition (ASR) engine to use
- asr-grammar: Automated Speech Recognition (ASR) grammar to use
- terminators: The keys that you want to use to immediately stop and process the digits collected

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <playback action="http://newurl/index.php"
      temp-action="http://newtempurl/index.php"
      name="playback_user_input"
      error-file="ivr/ivr-error.wav"
      file="ivr/ivr-welcome_to_freeswitch.wav"
      asr-engine="pocketsphinx"
      asr-grammar="my_default_asr_grammar"
      digit-timeout="5"
      input-timeout="10"
      loops="3"
      terminators="#">
      <bind strip="#">~\\d{3}</bind>
    </playback>
  </work>
</document>
```

The playback action is analogous to the playback Dialplan application.

record

```
<record file name error-file action digit-timeout input-timeout><bind action strip>*EXPR*</bi
```

record records a file, optionally collects input, and posts the file back to the target URL. It has the following attributes:

- file: The file path to record
- name: Param name to save result (will be a multipart form file upload)
- error-file: Error file to play on invalid input
- beep-file: File to play as an indicator to start recording message (that is, a voicemail beep)
- digit-timeout: Timeout waiting for digits after file plays (when input bindings are present)
- limit: Upper limit of number of seconds to record
- terminators: The keys that you want to use to immediately stop and process the digits collected

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <record action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      name="playback_user_input"
      error-file="ivr/ivr-error.wav"
      beep-file="tone_stream://$${beep}"
      file="12345.wav"
      digit-timeout="5"
      limit="60"
      terminators="#">
      <bind strip="#">~\\d{3}</bind>
    </record>
  </work>
</document>
```

The record action is analogous to the record Dialplan application.

pause

```
<pause name error-file action digit-timeout input-timeout loops milliseconds><bind action st
```

pause waits for input for a specific amount of time. It has the following attributes:

- milliseconds: Number of milliseconds to pause
- name: Param name to save result
- error-file: Error file to play on invalid input
- digit-timeout: Timeout waiting for digits after file plays (when input bindings are present)
- input-timeout: Timeout waiting for more digits in a multi-digit input
- loops: Maximum number of times to play the file when input bindings
  are present.
- terminators: The keys that you want to use to immediately stop and process the digits collected

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <pause action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      name="pause_user_input"
      error-file="ivr/it_was_that_bug.wav"
      digit-timeout="5"
      milliseconds="15000"
      terminators="#">
      <bind strip="#">~\\d{3}</bind>
    </pause>
  </work>
</document>
```

speak

```
<speak file name error-file action digit-timeout input-timeout loops engine voice><bind acti
```

speak reads text to the caller using the TTS (Text-to-Speech) engine, optionally collecting input. It has the following attributes:

- text: The text to be spoken to the caller
- name: Param name to save result
- error-file: Error file to play on invalid input
- digit-timeout: Timeout waiting for digits after file plays (when input bindings are present)
- input-timeout: Timeout waiting for more digits in a multi-digit input
- loops: Maximum number of times to play the file when input bindings are present
- engine: Text-to-Speech (TTS) engine to use
- voice: Text-to-Speech (TTS) voice to use
- terminators: The keys that you want to use to immediately stop and process the digits collected

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <speak action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      name="speak_user_input"
      error-file="ivr/ivr-error.wav"
      digit-timeout="5"
      engine="flite"
      voice="slt"
      text="Hello from flite text to speech engine"
      terminators="#">
      <bind strip="#">~\\d{3}</bind>
    </speak>
  </work>
</document>
```

The speak action is analogous to the speak Dialplan application.

say

```
<say file name error-file action digit-timeout input-timeout loops language type method gende
```

Use the FreeSWITCH say engine to iterate sounds to simulate a human speaker. It has the following attributes:

- text: The text to speak, spell, pronounce, and so on
- name: Param name in which the result will be saved
- error-file: Error file to play on invalid input
- digit-timeout: Timeout waiting for digits after file plays (when input bindings are present)
- input-timeout: Timeout waiting for more digits in a multi-digit input
- loops: Maximum number of times to play the file when input bindings are present
- language: Language of speech
- type: Type (say interface parameter)
- method: Method (say interface parameter)
- gender: Gender (say parameter)
- terminators: The keys that you want to use to immediately stop and process the digits collected

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <say action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      name="say_user_input"
      error-file="ivr/ivr-error.wav"
      digit-timeout="5"
      language="en"
      type="name_spelled"
      method="pronounced"
      text="This is what the caller will hear"
      terminators="#">
      <bind strip="#">~\\d{3}</bind>
   </say>
 </work>
</document>
```

The say action is analogous to the say Dialplan application. See the say entry under Important Dialplan applications in Chapter 6, *XML Dialplan*.

execute

```
<execute application data action>*DATA*</execute>
```

execute executes a FreeSWITCH Dialplan application. It has the following attributes:

- application: The Dialplan application to run
- data: Alternate source for application data
- *DATA*: The application data

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <execute action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      application="log"
      data="INFO this is an info log message"/>
   </work>
 </document>
```

sms

```
<sms to action>DATA</sms>
```

sms sends an SMS message. It has the following attributes:

- to: The destination number
- *DATA*: The message data

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <sms action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      to="sip:1007@192.168.1.101">Message text here</sms>
  </work>
</document>
```

Note: this requires mod_sms to be compiled and loaded. See http://wiki.freeswitch.org/wiki/Mod_sms for more information.

dial

```
<dial context dialplan caller-id-name caller-id-number action>*DATA*</dial>
```

dial places an outbound call or transfer. It has the following attributes:

- context: Dialplan context
- Dialplan: Dialplan type (usually XML)
- caller-id-name: Caller ID Name
- caller-id-number: Caller ID Number
- *DATA*: Number to dial or originate string

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <dial action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      caller-id-name="HTTAPI Test"
      caller-id-number="19193869900"
      context="default"
      Dialplan="XML">
        sip:2019@10.1.1.12
      </dial>
  </work>
</document>
```

The dial action will place a call through the Dialplan, and if a new call leg is created, the call being controlled by HTTAPI will be connected to it.

recordCall

```
<recordCall limit name action>
```

recordCall initiates recording of the call. The file will be posted when the call ends. It has the following attributes:

- limit: Timeout in seconds.
- name: If this starts with http:// then it must specify the URL where FreeSWITCH will PUT the file. Your web server must be set up to handle a PUT request in order to use it this way. If omitted, FreeSWITCH will record to a temporary directory.

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <recordCall action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      name="http://localhost/newfile.wav"
      limit="60"/>
  </work>
</document>
```

The recordCall action is analogous to the record Dialplan application.

conference

<conference profile action>

conference starts a conference call. It has the following attributes:

- profile: Conference profile to use
- *DATA*: The conference name in which the call will be placed

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <conference action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      profile="my_new_profile">
        My_Conference
      </conference>
 </work>
</document>
```

The conference action is analogous to the conference Dialplan application.

hangup

```
<hangup cause action>
```

hangup hangs up the call. It has the following attributes:

- cause: The hangup cause to send

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <hangup action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      cause="NORMAL_CLEARING"/>
  </work>
</document>
```

The hangup action is analogous to the hangup Dialplan application.

break

break exits the httapi application and continues in the Dialplan.

```
<document type="text/freeswitch-httapi">
  <work>
    <break/>
  </work>
</document>
```

log

```
<log level clean action>
```

log writes a log line to fs_cli, console, and log file.

- level: The log level to use
- clean: If set to a true value, then the log line will not print the log prefix

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <log action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      level="info">this is a log message with a prefix</log>
   <log level="warning"
      clean="1">and this is one without</log>
  </work>
</document>
```

The log action is analogous to the log Dialplan application. Note that the log Dialplan application does not have a clean option whereas the httapi log action does.

continue

```
<continue action>
```

continue performs no specific work actions and continues (that is, a no-op action). This is useful if you want to request a different action URL based on the results of a getVar or similar.

```
<document type="text/freeswitch-httapi">
  <work>
    <continue action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"/>
  </work>
</document>
```

getVar

```
<getVar action temp-action permanent>
```

getVar gets a channel variable's contents (depends on permissions). It has the following attributes:

- permanent: When set to a true value this variable gets sent on all subsequent HTTAPI requests for this call, otherwise it is sent only on the next request
- name: The variable name to read from the channel (for example, caller_id_name)

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <getVariable name="caller_id_name"
      action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      permanent="1"/>
  </work>
</document>
```

voicemail

```
<voicemail action temp-action check auth-only profile domain id>
```

voicemail calls the voicemail Dialplan application without requiring "execute" permissions. It has the following attributes:

- check: When set to a true value this allows the caller to check messages; that is, the mailbox user. If omitted then the caller will be prompted to leave a voice message in the mailbox.
- auth-only: Authenticate only and move on. In the event this mode is chosen, two new variables will be set on the channel upon successful authentication:
  - variable_user_pin_authenticated is set to true
  - variable_user_pin_authenticated_user is the username of the successfully authenticated user
    - profile: Voicemail profile name to use (omit for "default").
    - domain: Domain to use (omit for global domain variable).
    - id: ID to use (omit to prompt for id).

- Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <voicemail action="http://localhost/newurl.php"
      temp-action="http://localhost/newtempurl.php"
      auth-only="1"
      check="1"
      domain="192.168.1.101"
      id="1010"
      profile="default"/>
  </work>
</document>
```

The voicemail action is analogous to the voicemail Dialplan application.

vmname

vmname plays a voicemail name and optionally collects input. It has the following attributes:

- id: User's name to play passed as user@domain
- name: Param name to save result
- error-file: Error file to play on invalid input
- digit-timeout: Timeout waiting for digits after file plays (when input bindings are present)
- input-timeout: Timeout waiting for more digits in a multi-digit input (when input bindings are present)
- loops: Maximum number of times to play the file (when input bindings are present)
- terminators: The keys that you want to use to immediately stop and process the digits collected

Example:

```
<document type="text/freeswitch-httapi">
  <work>
    <vmname action="http://newurl/index.php"
      temp-action="http://newtempurl/index.php"
      name="vmname_user_input"
      error-file="ivr/ivrerror.wav"
      id="1007@192.168.1.101"
      digit-timeout="5"
      input-timeout="10"
      loops="3"
      terminators="#">
      <bind strip="#">~\\d{3}</bind>
    </vmname>
  </work>
</document>
```

exiting (param)

In the event that the user hangs up, FreeSWITCH will pass the "exiting" param to let the webserver know that the call is over, and you can tear down any sessions that you might have opened and complete any reporting that you might have been keeping track of. While you can completely ignore this request and FreeSWITCH will recover just fine, it's expecting a plain text response of "OK".

storing data across successive requests

What is the best way to store pieces of information from one request to the next? Obviously, you could **set and get channel variables**, but that **could be expensive** with having to do multiple requests per set/get operation.

You could just **store** those bits of information **in a session** for easy access later. Each request will include a **session_id** POST or GET **parameter**. With this session_id parameter value you should be able to initialize a session using its value as the session identifier. This level of control is available in just about every web programming language. An example of doing this in PHP would look similar to the following:

```
if ( array_key_exists( 'session_id', $_REQUEST ) ) {
    session_id( $_REQUEST['session_id'] );
}
session_start();
```

After you've started the session, you will have access to a session variable or object that can be used to store information that you can access on subsequent requests.

mod_httapi configuration file

The mod_httapi configuration file is found in conf/autoload_configs and is named httapi.conf.xml. It contains several **settings** parameters as well as a **profiles** section. The example configuration contains a **default** HTTAPI **profile** (you may create your own profiles).

params

Inside the profile tag you will notice a number of **param** entries. These control things such as **default settings**, and the **default URL** to use for HTTP requests.

gateway-url

The **gateway-url** parameter set the default resource that mod_httapi will use for making requests. This parameter can be overridden in dialplan, as an argument to httapi application.

method

The **method** parameter set the default method (GET|POST|PUT) that mod_httapi will use for making requests. This parameter can be overridden in dialplan, inside curly braces.

permissions

You may want to have permissions on things such as variables that shouldn't be changed, or applications and APIs that you don't want to execute inadvertently.

In the example httapi.conf.xml configuration file, within the permissions tag you'll find many different permissions that you can enable, with even more fine-grained ACL-style control over certain aspects of some of them.

One thing to note is that any of the ACL-like control lists we'll see below can be skipped by simply closing the <permission> tag without including the list. This will default to allow all as if you had created the list with default="allow". The following two examples will work exactly the same way.

Example one:

```
<permission name="set-vars" value="true"/>
```

Example two:

```
<permission name="set-vars" value="true">
  <variable-list default="allow">
  </variable-list>
</permission>
```

set-params

It is a permission to allow you to set or restrict the setting of the same parameters that can be set from within the {}'s when calling httapi from the Dialplan and have them persist throughout the lifetime of the call.

set-vars

It allows you to set or restrict the setting of channel variables. This permission goes a step further than the set-params does and allows you to specify which variables you want to allow the setting of, similar to the way access control is handled in acl.conf.xml. Notice the default policy and the ability to allow access in the following snippet from the example configuration file:

```
<permission name="set-vars" value="true">
  <variable-list default="deny">
    <!-- Variables here may be changed -->
    <variable name="caller_id_name"/>
  </variable-list>
</permission>
```

The preceding code listed says, in effect, that the ability to set variables is disabled unless they are specifically named within the variable-list. So, variables named inside the variable-list are the exception to the rule established with default:

```
<permission name="set-vars" value="true">
  <variable-list default="allow">
    <!-- Variables here may *not* be changed -->
    <variable name="caller_id_name"/>
  </variable-list>
</permission>
```

get-vars

allows you the ability to read channel variables from the call. This permission has the same fine-grained controls that the set-vars option gives.

extended-data

pass much more information to your web application than without it. The default behavior is to post a succinct overview of the channel and allow you to get the information you need via HTTAPI commands and subsequent callbacks. If you would rather have every set channel variable posted to your application upon the initial request, you only need to enable this option.

extended-data longer than max CGI length

If you turn on extended-data in configuration profile then you may start missing out on some of the data that should be passed to your application. The reason for this is that the default method is a GET request. All the information that is passed with extended-data can often exceed the maximum allowed length of CGI parameters, which results in cutting off some of the data from the end of the request. There are a couple of ways to fix this. The most permanent way is to set the method parameter in the httapi.conf.xml file as shown here:

```
<param name="method" value="POST"/>
```

However, there is a way to set the method per-request if you decide that you only want to set the method to POST on the requests that are causing issues. You can set the method in the URL string as shown here:

```
<action application="httapi"
  data="{url=http://localhost/httapi/index.php,method=POST}"/>
```

execute-apps

If you set this permission then you'll have the ability to call Dialplan applications from within your httapi web application. This will allow you to use applications with an <execute> tag as described in Work actions syntax. This permission has the same sort of ACL-like control that we have seen in a couple of the other permissions already. Let's deny access to use applications with a default deny policy, and then allow the execution of the info and hangup applications:

```
<permission name="execute-apps" value="true">
  <application-list default="deny">
    <application name="info"/>
    <application name="hangup"/>
  </application-list>
</permission>
```

expand-vars

This permission allows you to use variables in your applications like you normally could from the XML Dialplan. Variables like ${caller_id_number} would be expanded inline. The expression ${caller_id_number} would give you the number of the calling party. This also gives you a way to use API commands from within your web applications. Consider this example:

```
${sofia_contact(1010@192.168.1.100)}
```

This line will execute the sofia_contact API command with the given argument and have the result inserted in place.

ACL-like control lists are available. You can allow or disallow as many API commands or variables as you need to. Consider the following XML snippet:

```
<permission name="expand-vars" value="true">
  <variable-list default="deny">
    <variable name="caller_id_name"/>
    <variable name="caller_id_number"/>
  </variable-list>
  <api-list default="deny">
    <api name="expr"/>
    <api name="lua"/>
    <api name="sofia_contact"/>
  </api-list>
</permission>
```

The preceding code would allow the setting of the caller_id_name and caller_id_number channel variables and no others. It would allow the execution of the expr, lua, and sofia_contact API commands but no others. This example shows the fine-grained control that you as the application developer and FreeSWITCH system administrator have over HTTAPI applications running on your system.

dial

allows you to dial a number from your web application that will hit the Dialplan and be routed accordingly. Enabling any one of dial-set-context, dial-set-Dialplan, dial-set-cid-name, dial-set-cid-number, or dial-full-originate will enable the dial permission even if it is set to false elsewhere in the configuration file.

dial-set-dialplan

allow you to change the dialplan kind inside the dial tag. You want always use the "XML" dialplan, the default.

dial-set-context

allow you to change the dialplan context inside the dial tag. This would allow to use:

```
<dial context="othercontext">
```

dial-set-cid-name

allow you to change the callerid name inside the dial tag. This would allow to use:

```
<dial cid_name="Giovanni Maruzzelli">
```

dial-set-cid-number

allow you to change the callerid number inside the dial tag. This would allow to use:

```
<dial cid_number="+393472665618">
```

dial-full-originate

will allow you to dial using the full endpoint/profile/number syntax. (for example: sofia/internal/1010@192.168.1.100)

conference

will allow you to call into conferences

conference-set-profile

allow you to change the profile name inside the conference tag. This would allow to use:

```
<conference profile="wb">
```

If conference-set-profile is enabled then conference will be enabled even if it is set to false elsewhere in the configuration file.

HTTPAPI libraries for PHP and Python

As you looked at the examples earlier in the chapter, you may have thought that while there might be a lot of power in this httapi thing, you really don't want to learn another XML format to control FreeSWITCH. Also, manually printing all of that XML could be a huge hassle. That's why the httapi XML was written to be easy to implement with a helper library in the language of your choice.

A couple of such libraries exist, for PHP and for Python. Both libraries were developed by Raymond Chandler, and you can find them in the freeswitch-contrib GIT repository, under the directory intralanman. To download the libraries and the examples (and a whole lot of other good stuff):

```
cd /usr/src
git clone https://freeswitch.org/stash/scm/fs/freeswitch-contrib.git
cd freeswitch-contrib/intralanman
```

There you'll find a PHP and a Python directory. Both PHP and Python libraries are composed by one only file (phttapi.php and httapy.py), for ease of use.

The demo-ivr in PHP-HTTAPI

Before you start, you need to already have a web server set up to serve up PHP files, with the PHP XML extensions installed. Next thing you'll need to do is the PHTTAPI library.

In our demo script, named "demo-ivr.php", we have the library in the parent directory, "required" in the second line:

```
<?php
require "../phttapi.php";

if ( array_key_exists( 'session_id', $_REQUEST ) ) {
    session_id( $_REQUEST['session_id'] );
}
session_start();

if ( array_key_exists( 'exiting', $_REQUEST ) ) {
    header( 'Content-Type: text/plain' );
    print "OK";
    exit();
}

$demo = new phttapi();
$opt  = array_key_exists( 'main_menu_option', $_REQUEST ) ? $_REQUEST['main_menu_option'] : '

$demo->start_variables();
$demo->add_variable( 'IVR_variable_01', 'VariableValue01' );
$demo->end_variables();
$demo->start_params();
$demo->add_param( 'IVR_param_01', 'ParamValue01' );
$demo->end_params();

if ( preg_match( '/^10[01][0-9]$/', $opt ) ) {
    $xfer = new phttapi_dial( $opt );
    $xfer->context( 'default' );
    $xfer->dialplan( 'XML' );
    $demo->add_action( $xfer );
} else {
    switch ( $opt ) {
        case '1':
            $conf = new phttapi_dial( '9888' );
            $conf->caller_id_name( 'another book reader' );
            $conf->context( 'default' );
            $conf->dialplan( 'XML' );
            $demo->add_action( $conf );
            break;

        case '2':
            $echo = new phttapi_dial( '9196' );
            $echo->context( 'default' );
            $echo->dialplan( 'XML' );
            $demo->add_action( $echo );
            break;

        case '3':
            $moh = new phttapi_dial( '9664' );
            $moh->context( 'default' );
            $moh->dialplan( 'XML' );
            $demo->add_action( $moh );
            break;

        case '4':
            $clue = new phttapi_dial( '9191' );
            $clue->caller_id_name( 'another book reader' );
            $clue->context( 'default' );
            $clue->dialplan( 'XML' );
            $demo->add_action( $clue );
            break;

        case '5':
            $monkey = new phttapi_dial( '1234*256' );
            $monkey->dialplan( 'enum' );
            $demo->add_action( $monkey );
            break;

        case '6':
            if ( array_key_exists( 'sub_menu_option', $_REQUEST ) && $_REQUEST['sub_menu_opti
                unset( $_SESSION['first_sub_play_done'] );
                $demo->add_action( $c = new phttapi_continue() );
                break;
            }
            $demo->start_variables();
            $demo->add_variable( 'main_menu_option', 6 );
            $demo->end_variables();

            $sub = new phttapi_playback();
            $sub->error_file( 'ivr/ivr-that_was_an_invalid_entry.wav' );
            $sub->loops( 3 );
            $sub->digit_timeout( '15000' );

            if ( !array_key_exists( 'first_sub_play_done', $_SESSION ) ) {
                $_SESSION['first_sub_play_done'] = TRUE;
                $sub->file( 'phrase:demo_ivr_sub_menu' );
            } else {
                $sub->file( 'phrase:demo_ivr_sub_menu_short' );
            }

            $star = new phttapi_action_binding( '*' );
            $sub->add_binding( $star );
            $sub->name( 'sub_menu_option' );

            $demo->add_action( $sub );
            break;

        case '9':
            $continue = new phttapi_continue();
            $demo->add_action( $continue );
            break;

        default:
            $intro = new phttapi_playback();
            $intro->error_file( 'ivr/ivr-that_was_an_invalid_entry.wav' );
            $intro->loops( 3 );
            $intro->digit_timeout( '2000' );
            $intro->input_timeout( '10000' );
            $intro->name( 'main_menu_option' );

            if ( !array_key_exists( 'first_play_done', $_SESSION ) ) {
                $_SESSION['first_play_done'] = TRUE;
                $intro->file( 'phrase:demo_ivr_main_menu' );
            } else {
                $intro->file( 'phrase:demo_ivr_main_menu_short' );
            }

            $b1   = new phttapi_action_binding( 1 );
            $b2   = new phttapi_action_binding( 2 );
            $b3   = new phttapi_action_binding( 3 );
            $b4   = new phttapi_action_binding( 4 );
            $b5   = new phttapi_action_binding( 5 );
            $b6   = new phttapi_action_binding( 6 );
            $b9   = new phttapi_action_binding( 9 );
            $bext = new phttapi_action_binding( '~10[01][0-9]' );
            $bext->strip( '#' );
            $intro->add_binding( $bext );

            $intro->add_binding( $b1 );
            $intro->add_binding( $b2 );
            $intro->add_binding( $b3 );
            $intro->add_binding( $b4 );
            $intro->add_binding( $b5 );
            $intro->add_binding( $b6 );
            $intro->add_binding( $b9 );

            $demo->add_action( $intro );
    }
}
header( 'Content-Type: text/xml' );
foreach ( $_REQUEST as $key => $val ) {
    $demo->comment( " $key => $val " );
}

print $demo->output();
```

Copy the demo script in the web documents directory on your web server.

Please follow along as we discuss each line of code and what it does.

```
if ( array_key_exists( 'session_id', $_REQUEST ) ) {
    session_id( $_REQUEST['session_id'] );
}
session_start();
```

This block will start the session using the session_id like we described earlier in this chapter.

```
if ( array_key_exists( 'exiting', $_REQUEST ) ) {
    session_destroy();
    header( 'Content-Type: text/plain' );
    print "OK";
    exit();
}
```

If we see the exiting parameter, we destroy the PHP session and tell FreeSWITCH we understand and then we exit the script.

```
    $demo = new phttapi();
```

Here, we create the httapi object. This object ($demo) allows us to perform operations such as work actions.

```
    $opt = array_key_exists( 'main_menu_option', $_REQUEST ) ?
           $_REQUEST['main_menu_option'] : '';
```

This is a simple if/then/else condition that will make sure $opt is always set, even if the main_menu_option is empty. The main_menu_option, which we later bind to the options, is going to be populated with the keys that were pressed by the caller.

```
if ( preg_match( '/^10[01][0-9]$/', $opt ) ) {
    $xfer = new phttapi_dial( $opt );
    $xfer->context( 'default' );
    $xfer->Dialplan( 'XML' );
    $demo->add_action( $xfer );
} else {
```

This block will test if the option matches the extension regex. If it is then we build a new phttapi_dial object ($xfer), set the destination, and then add the action to the $demo object. If it's not an extension, we drop into a switch statement that tests for each of the single keypress options.

```
    case '1':
        $conf = new phttapi_dial( '9888' );
        $conf->caller_id_name( 'another book reader' );
        $conf->context( 'default' );
        $conf->Dialplan( 'XML' );
        $demo->add_action( $conf );
        break;
```

If option 1 was pressed, then we create a dial option that corresponds to the dial tag that we described earlier in this chapter. Each of the attributes on the tag have a corresponding method in the phttapi_dial class. For example, the context method sets the context attribute, the Dialplan method sets the Dialplan attribute, and so on. (Option 1 will send the caller to the public FreeSWITCH conference server.)

Cases 2 through 5 are all dial objects and have the same basic logic with the attributes assigned differently to achieve the desired results for each option.

```
    case '6':
        if ( array_key_exists( 'sub_menu_option', $_REQUEST ) && $_REQUEST['sub_menu_option']
            unset( $_SESSION['first_sub_play_done'] );
            $demo->add_action( $c = new phttapi_continue() );
            break;
        }
        $demo->start_variables();
        $demo->add_variable( 'main_menu_option', 6 );
        $demo->end_variables();

        $sub = new phttapi_playback();
        $sub->error_file( 'ivr/ivr-that_was_an_invalid_entry.wav' );
        $sub->loops( 3 );
        $sub->digit_timeout( '15000' );

        if ( !array_key_exists( 'first_sub_play_done', $_SESSION ) ) {
            $_SESSION['first_sub_play_done'] = TRUE;
            $sub->file( 'phrase:demo_ivr_sub_menu' );
        } else {
            $sub->file( 'phrase:demo_ivr_sub_menu_short' );
        }

        $star = new phttapi_action_binding( '*' );
        $sub->add_binding( $star );
        $sub->name( 'sub_menu_option' );

        $demo->add_action( $sub );
        break;
```

Option 6 is a little tricky and should probably be broken out into its own file, as it is technically a separate IVR. We included it in a single file for you here to make it easier to install and test. (Option 6 demonstrates an IVR submenu.)

```
        case '9':
            $continue = new phttapi_continue();
            $demo->add_action( $continue );
            break;
```

With this chunk of code, we simply do a continue, which has the effect of "repeating these options" as there are no bindings and no way to pass the main_menu_option parameter.

```
        default:
            $intro = new phttapi_playback();
            $intro->error_file( 'ivr/ivr-that_was_an_invalid_entry.wav' );
            $intro->loops( 3 );
            $intro->digit_timeout( '2000' );
            $intro->input_timeout( '10000' );
            $intro->name( 'main_menu_option' );
            $intro->terminators( '#' );

            if ( !array_key_exists( 'first_play_done', $_SESSION ) ) {
                $_SESSION['first_play_done'] = TRUE;
                $intro->file( 'phrase:demo_ivr_main_menu' );
            } else {
                $intro->file( 'phrase:demo_ivr_main_menu_short' );
            }
```

The default case action is to play the intro file. To simulate the way the ivr Dialplan application does it, we'll just store something in the session to let us know whether or not we've already played the long intro or not. (The short and long greetings are explained in IVR menu definitions in Chapter 7, *Phrase Macros and XML IVRs*)

```
        $b1   = new phttapi_action_binding( 1 );
        ...
        $bext = new phttapi_action_binding( '~10[01][0-9]' );

        $intro->add_binding( $b1 );
        ...
        $intro->add_binding( $bext );
        ...
        $demo->add_action( $intro );
```

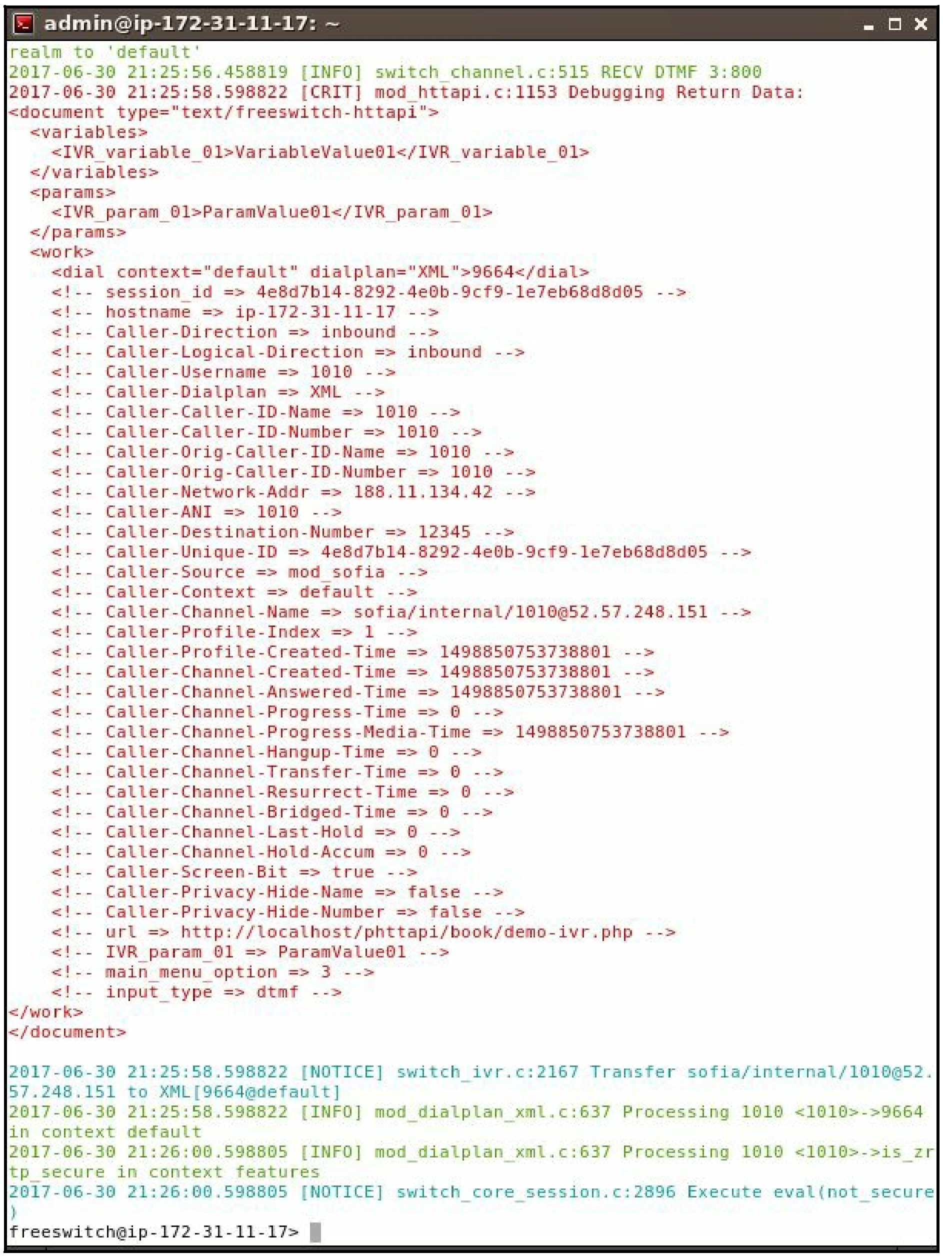
In this section the ellipses indicate that the other options were left out for brevity.

You can see here that we create a binding object for each of the digit choices and then add each binding to the playback action. Then, just as with our previous examples, we add the action to the $demo object. Obviously, we could have built out a single binding with a more complete regex that would have worked for all the digits. However, we did it this way to show that you can have multiple bindings with single digits and/or regexes and things will still work as designed.

```
header( 'Content-Type: text/xml' );
print $demo->output();
```

Here, we set the content type of the response to text/xml and print the output of the object we've been building. FreeSWITCH won't understand your responses if you use any other content-type than text/xml, so be sure to set that however it's done in your language of choice.

And this is the terminal output when we first type: "fsctl loglevel 6", and then make a call into our newly created HTTAPI extension, and press "3":



```
realm to 'default'
2017-06-30 21:25:56.458819 [INFO] switch_channel.c:515 RECV DTMF 3:800
2017-06-30 21:25:58.598822 [CRIT] mod_httapi.c:1153 Debugging Return Data:
<document type="text/freeswitch-httapi">
  <variables>
    <IVR_variable_01>VariableValue01</IVR_variable_01>
  </variables>
  <params>
    <IVR_param_01>ParamValue01</IVR_param_01>
  </params>
  <work>
    <dial context="default" dialplan="XML">9664</dial>
    <!-- session_id => 4e8d7b14-8292-4e0b-9cf9-1e7eb68d8d05 -->
    <!-- hostname => ip-172-31-11-17 -->
    <!-- Caller-Direction => inbound -->
    <!-- Caller-Logical-Direction => inbound -->
    <!-- Caller-Username => 1010 -->
    <!-- Caller-Dialplan => XML -->
    <!-- Caller-Caller-ID-Name => 1010 -->
    <!-- Caller-Caller-ID-Number => 1010 -->
    <!-- Caller-Orig-Caller-ID-Name => 1010 -->
    <!-- Caller-Orig-Caller-ID-Number => 1010 -->
    <!-- Caller-Network-Addr => 188.11.134.42 -->
    <!-- Caller-ANI => 1010 -->
    <!-- Caller-Destination-Number => 12345 -->
    <!-- Caller-Unique-ID => 4e8d7b14-8292-4e0b-9cf9-1e7eb68d8d05 -->
    <!-- Caller-Source => mod_sofia -->
    <!-- Caller-Context => default -->
    <!-- Caller-Channel-Name => sofia/internal/1010@52.57.248.151 -->
    <!-- Caller-Profile-Index => 1 -->
    <!-- Caller-Profile-Created-Time => 1498850753738801 -->
    <!-- Caller-Channel-Created-Time => 1498850753738801 -->
    <!-- Caller-Channel-Answered-Time => 1498850753738801 -->
    <!-- Caller-Channel-Progress-Time => 0 -->
    <!-- Caller-Channel-Progress-Media-Time => 1498850753738801 -->
    <!-- Caller-Channel-Hangup-Time => 0 -->
    <!-- Caller-Channel-Transfer-Time => 0 -->
    <!-- Caller-Channel-Resurrect-Time => 0 -->
    <!-- Caller-Channel-Bridged-Time => 0 -->
    <!-- Caller-Channel-Last-Hold => 0 -->
    <!-- Caller-Channel-Hold-Accum => 0 -->
    <!-- Caller-Screen-Bit => true -->
    <!-- Caller-Privacy-Hide-Name => false -->
    <!-- Caller-Privacy-Hide-Number => false -->
    <!-- url => http://localhost/phttapi/book/demo-ivr.php -->
    <!-- IVR_param_01 => ParamValue01 -->
    <!-- main_menu_option => 3 -->
    <!-- input_type => dtmf -->
</work>
</document>

2017-06-30 21:25:58.598822 [NOTICE] switch_ivr.c:2167 Transfer sofia/internal/1010@52.
57.248.151 to XML[9664@default]
2017-06-30 21:25:58.598822 [INFO] mod_dialplan_xml.c:637 Processing 1010 <1010>->9664
in context default
2017-06-30 21:26:00.598805 [INFO] mod_dialplan_xml.c:637 Processing 1010 <1010>->is_zr
tp_secure in context features
2017-06-30 21:26:00.598805 [NOTICE] switch_core_session.c:2896 Execute eval(not_secure
)
freeswitch@ip-172-31-11-17>
```

Summary

In this chapter we learned about the new functionality unleashed with mod_httapi. By combining a web server with FreeSWITCH it is now possible to do call control using the simple HTTAPI markup. Further, we also discussed a PHP library (phttapi.php) that provides an abstraction layer to make it even easier to build telephony applications on a web server. By using mod_httapi, enterprises can leverage the knowledge of their web developers to assist with creating telephony applications. Furthermore, web developers need not learn all that a FreeSWITCH administrator needs to know. Rather, they can learn just HTTAPI and still have everything they need to build feature-rich, web-controlled telephony applications.

In the next chapter, we will focus on a very important subject in today's converged world: Conferencing and WebRTC Videoconferencing

Conferencing and WebRTC Video-Conferencing

Conferencing is one of the premium services you want to offer to your users and your customers. In these times where we're between cost cuts and time constraints, the ability to have a virtual meeting is a real bonus.

So more so if you can manage the meeting, give the "talk" floor in an ordered manner to each presenter, mute and unmute each and every of the participants, etc.

And when you add to the mix the video stream from each participant webcam, the possibility to share a presentation screen, to play videos, to record the session, and to have different persons "on stage" at the same time... That's not only cheaper and faster than walking in a physical room to join a meeting, is also more productive!

We have also developed a WebRTC client that fully leverage our contemporary VERTO protocol to offer a beautiful, productive, immersive and engaging conference experience to participants and moderators!

FreeSWITCH gives you all the building blocks, and thanks to WebRTC support, your conferences' participants, moderators and presenters only need a standard browser on their computer, tablet or smartphone.

In this chapter we will discuss:

- Conferencing concepts, audio and video
- Conferencing support in FreeSWITCH
- Audio Conferencing Setup in FreeSWITCH
- Audio Conferencing Operation
- Video Conferencing Setup in FreeSWITCH
- Video Conferencing Operation
- Verto Communicator in Video Conferencing
- Video Conferencing Additional Tricks

Conferencing Concepts

Conferencing, participating in a call where more than two parties can communicate, has always been one of the most sought-of service in telecommunication.

Technology for allowing more than two parties to concurrently talk each other (eg, not in a push-to-talk service) is very much different than the one used for "normal" calls with two participants. In a "normal" call, you only care that media streams goes back and forth from caller to callee. Actually you can have (and often you have) the media streams (via RTP) going directly from caller to callee and back, bypassing FreeSWITCH or any other telecommunication server.

Conferencing, on the other hand, needs a server. The most basic function of a conferencing server is mixing, eg merging, the different media streams that are sent by participants. The resulting merged stream is broadcasted to all participants. So, each participant sends to server its own media stream, and all participants receive from server a media stream that is the "sum" of the streams the server received from them all. Participants do not communicate directly with each other, they are linked together in a "star topology" where the conferencing server is the central hub.

On top of the basic "merge all incoming streams and send the resulting stream out" function, conferencing servers can have much more advanced features. They can accept commands to alter how the incoming streams are merged, to only merge some of them (or to broadcast one only), to play media files to participants, to record the conferencing session, to alter the characteristics of the streams that are received or sent out, to ask pin code identification to participants, to consider one or more participants as "moderators" bearing powers on other participants and conference management, and many more.

As you correctly suspected, FreeSWITCH provides for all that and much more.

Conferencing Support in FreeSWITCH

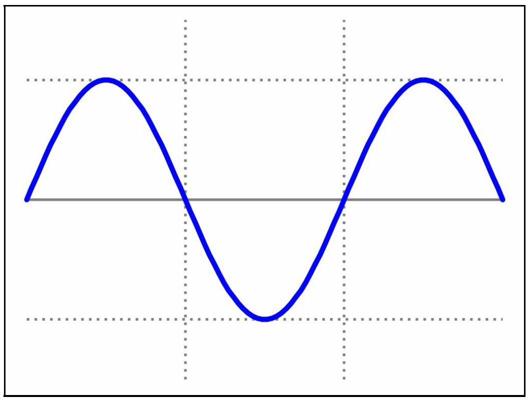
Since the first stages of its drawing board phase, FreeSWITCH has been designed to excel in mixing capabilities. Streams can be combined and massaged by the core in an hyper-efficient way.

The media processing engine in FreeSWITCH core is very flexible, and provides hooks so that core developers and contributors can add features without ever risking to compromise stability and efficiency of the system.

There are two obvious main areas of media processing: audio and video. Audio and video are treated in a complete different way because of their own completely different nature.

Audio conferencing

Audio streams are composed by a sequence of audio samples. Each audio sample is essentially a number (a quantity) representing the evolution of the sound in time. Sound can be described by those sine waves we've seen in high school textbooks, and each sample is the value in amplitude at a certain point in time.



Mixing (merging, combining) two or more sound samples is essentially about summing their numeric value, and divide the total by the number of streams we are mixing (eg: taking the average at each point in time). Obviously, is not at all so simple when the samples have been previously compressed, encoded, etc. And then you want to normalize the volume. But is exactly like that in its simplest and conceptual form (uncompressed, raw, sound samples).

Video conferencing

Video streams are another thing altogether! They are a complex sequence of values that describe how an image (the "screen") changes over time. You have a streaming movie (or a webcam feed) at 1920x1080 pixel resolution. Each one of those pixels, or most of them (very rarely all of them), can change value (of color, brightness, etc) at each point in time. So, the samples at each point in time are not representing the value of each pixel in the screen, but the result of complex calculations that analyze which pixel changed and how. You cannot add them, no meaning in that.

Also, the result you want when you combine video streams together is almost never their blending superimposition one on each other (like a "transparency" transition effect). You want to divide the screen real estate into areas, and display a different video streams (scaled) in each defined area. Or you want to display a floating logo, or a caption on each one of those screen areas. This is more similar to the "video mixer and effects" processor used by the director of a TV talk show.

There are various strategies for video conferencing (video streams combining) in today's telecommunication world, and each one has its own rationale and use cases:

- **Peer-to-Peer (MESH)** this is based on pure device to device communication (without server), where each participant send and receive video streams to and from all other participants. Streams are combined device-side (eg, in the browsers). Control commands may be broadcasted from each participant to all others if those commands must affect all devices' displays. Obvious problem of this strategy is that the number of video streams and the bandwidth grows quadratically with the number of nodes: 3 conference participants exchange 6 video streams, 4 conference participants 12 video streams, 10 participants would exchange 90 video streams, etc. This technique is popular in WebRTC services, but is not used with more than 3 or 4 conference participants.
- **Selective Forwarding Unit (SFU)** is a central server that receives the video streams from all participants, and decide which stream to forward to which participant. The central server (the SFU) is not combining streams or applying effects to them, it just routes packets in and out. Streams are combined device side (eg, in the browsers) and normally they just shows in different areas of the screen, without further processing. If multiple streams need to be combined on participants screen, all of those streams need to be routed to all participants. This can be very expensive in terms of bandwidth usage, and of client side CPU
- **Multipoint Conferencing Unit (MCU)** is a central server that receives the video streams from all participants, mixes and apply transformations and effects

to them, and then sends the one stream resulting from the processing to all participants. This has the lowest possible requirements in terms of bandwidth and client side CPU (each client sends its stream to the MCU and receive one stream from the MCU, ready to be displayed), but put a lot of stress on the MCU, that has to decode all the streams that are meant to be mixed, apply them complex Computer Graphics for captions, logos, merging and scaling and then reencode the result.

- **FreeSWITCH is an advanced MCU** (where mixing video streams and applying effects is a very heavy task) **that can also have the irrelevant CPU requirement of an SFU** in the limit case of "at each time only one participant originated video stream is showed on all participants screens" (eg, no video mixing and effects, only routing one of the incoming video streams to all participants).

Audio Conferencing setup in FreeSWITCH

FreeSWITCH includes a powerful built-in engine which allows the mixing of audio channels between callers in a multi-user audio conferencing system. mod_conference allows for full control of all audio mixing and caller interaction features, such as detection of touch-tones, management of send and receive audio paths per channel, volume controls, gain controls, and more. You can create as many conferences as you like, as long as free system resources (that is, memory, CPU cycles, and so forth) are still left in the server.

Configuration of mod_conference is located in /usr/local/freeswitch/autoload_configs/conference.conf.xml file. A conference characteristics and behavior are set by its profile. You can have many profiles in configuration file, each one with its own unique set of parameters. A conference is assigned to a profile when is created (usually in dialplan). The conference configuration file is divided into several sections. We'll see all of them below.

profiles

A conference profile is a set of settings that can be applied to a particular conference. In combination with caller-controls (discussed in this section), conference profiles allow for the complete customization of the behavior of individual conferences. You can create a couple profiles and apply them across many conferences, create a profile for each conference you intend to utilize, or you can simply utilize the defaults.

General conference profile structure is as follows:

```
<profiles>
<profile name="default">
<param name="paramName" value="paramValue"/>
</profile>
</profiles>
```

You can have any number of <profile> tags, and each <profile> tag can have any number of <param> tags. The following are some of the parameters that are available. For a full list, and extended explanations, please consult our documentation online on http://freeswitch.org/confluence, searching for "mod_conference".

rate

The rate parameter specifies the default (and highest) sampling rate that the conference bridge will utilize. All callers who call into this channel will have their audio transcoded into this sampling rate if they are not already at that rate. For the purposes of audio mixing, this defines the lowest sampling rate in relation to the system-if two callers have HD phones but call into a conference where the rate is 8000, those callers will have their audio sampled down to the lower rate.

Parameter syntax: <param name="rate" value="8000"/>

Available options: 8000, 12000, 16000, 24000, 32000, and 48000

caller-controls

This parameter specifies the caller-controls profile to use with this conference bridge.

Parameter syntax: <param name="caller-controls" value="default"/>

auto-record

This parameter specifies whether to automatically record conferences or not. Recording will begin once *min-required-recording-participants* have joined. This option, if set, must consist of a path that can be written to for the purposes of recording the conference.

Parameter syntax: <param name="auto-record" value="filename"/>

Default: off

Sample filename:
/usr/local/freeswitch/sounds/conferences/${conference_name}.wav

The sample filename listed would record conferences into a file based on the conference bridge's name.

interval

This parameter specifies the number of milliseconds per frame that are mixed. This setting is similar to how ptime works, but does not need to match the actual ptime of a caller. Higher numbers require less CPU usage, but can cause conversation quality issues, so experiment with your setup. The default is usually OK.

Parameter syntax: <param name="interval" value="20"/>

Default: 20

energy-level

This parameter specifies the energy level (or strength/volume of audio) required for audio to be sent to the other users. The energy level is a threshold that dictates the level at which a person is determined to be speaking versus the background noise received. This feature helps remove background or ambient noise from being mixed into the conference. If this option is too high, it can result in clipping at the beginning and end of people's sentences. The value 0 disables the detection and will bridge all packets even if they are only background noise.

Parameter syntax: <param name="energy-level" value="20"/>

Default: 20

Set 0 to disable completely

member-flags

This parameter allows for setting member-specific flags or parameters on individual conference members. These options include whether a participant is a **moderator**, whether to be wasteful with packet mixing (that is, send audio to individuals even when no speaking is happening in the conference), whether or not a specific member is the leader of a conference (and thus, the conference should terminate when they leave), and so on. Options should be separated by a pipe | character.

Parameter syntax: <param name="member-flags" value="waste|endconf"/>

Some of the options for the member-flags parameter follows. For a full list, and extended explanations, please consult our documentation online on http://freeswitch.org/confluence, searching for "mod_conference"

- **moderator**: Member has a different set of caller-controls, and can manage the conference: mute and unmute participants, etc
- **deaf**: Prevents the members from listening to other members in the conference by default (this can be changed after the conference has begun).
- **waste**: Sends audio to channels even when no conversation is occurring.
- **dist-dtmf**: Distributes DTMF signals to each channel. When someone sends a DTMF tone, it is normally absorbed and processed by FreeSWITCH. This option prevents that from happening and instead echoes the DTMF tone to all other members.
- **endconf**: Specifies that the conference should end when this member exits.

conference-flags

This parameter sets the conference-wide flags that dictate how the conference behaves. Options should be separated by a pipe | character.

Parameter syntax: <param name="conference-flags" value="wait-mod|audio-always"/>

Some of the options for the conference-flags parameter follows. For a full list, and extended explanations, please consult our documentation online on http://freeswitch.org/confluence, searching for "mod_conference"

- **wait-mod** force users to wait for the moderator before the conference begins. Moderators are determined via the Dialplan, when bridged to the conference, by passing an extra member flag. While waiting for the moderator to join, callers hear music on hold.
- **audio-always** Do not use energy detection to choose which participants to mix; instead always mix audio from all members
- **restart-auto-record** If the auto-record conference param is set, and recording is stopped, auto recording will continue in a new file

tts-engine

This parameter specifies the Text-To-Speech engine to utilize within this conference bridge.

Parameter syntax: <param name="tts-engine" value="cepstral"/>

tts-voice

This parameter specifies which Text-To-Speech engine voice to utilize within this conference bridge.

Parameter syntax: <param name="tts-voice" value="david"/>

pin

This parameter specifies the PIN code (that is pass code or password) that must be entered before user is allowed to enter the conference.

Parameter syntax: <param name="pin" value="12345"/>

max-members

This parameter sets the maximum number of members allowed in the conference. If this number is reached and an additional member tries to join, the max-members-sound will be played and the caller will not be allowed to enter the conference bridge.

Parameter syntax: <param name="max-members" value="20"/>

caller-id-name

This parameter instructs the caller ID name to set when making an outbound call from within this conference bridge.

Parameter syntax: <param name="caller-id-name" value="John Doe"/>

caller-id-number

This parameter instructs the caller ID number to set when making an outbound call from within this conference bridge.

Parameter syntax: <param name="caller-id-number" value="4158867900"/>

comfort-noise

This parameter instructs the volume level of background white noise to get added to the conference. Sometimes callers think they have been dropped from a conference if the audio level remains too quiet. This comfort noise setting provides white noise on the line so the caller knows the line is still connected. Note that at higher audio sampling rates, this noise can become bothersome, so you may wish to tweak this setting if you go above 8000 Hz sampling rates.

Parameter syntax: `<param name="comfort-noise" value="1000"/>`

announce-count

This parameter will speak the total number of callers in the conference when a new person joins, but only when the threshold specified in this parameter is reached. It requires a valid text-to-speech engine.

Parameter syntax: <param name="announce-count" value="5"/>

suppress-events

This parameter is for use with the FreeSWITCH event system. This parameter is a comma delimited string that specifies which events will NOT be sent to the socket when getting CUSTOM conference::maintenance events..

Parameter syntax: <param name="suppress-events" value="add-member,del-member"/>

sound-prefix

This parameter sets a default path from where to retrieve conference audio files.

Parameter syntax: <param name="sound-prefix" value="/usr/local/freeswitch/sounds/"/>

sounds

Many parameters are available for setting custom sounds to play from within the conference bridge when certain activities occur. All sounds are played to individual caller channels and not to all parties in the conference, with the exception of enter-sound and exit-sound, which are played to all participants.

All sound files are specified with the format: <param name="sound-name" value="file.wav"/>

- **muted-sound**: This sound is played when a caller has been muted.
- **unmuted-sound**: This sound is played when a caller is no longer muted.
- **alone-sound**: This sound is played to a caller when they are the only remaining party.
- **enter-sound**: This sound is played to all members when a new caller joins the conference.
- **exit-sound**: This sound is played to all members when a caller leaves the conference.
- **kicked-sound**: This sound is played when a caller is kicked out from the conference.
- **locked-sound**: This sound is played to callers who try to join a locked conference.
- **is-locked-sound**: This sound is played to conference participants when a conference is locked.
- **is-unlocked-sound**: This sound is played to conference participants when a conference is unlocked.
- **pin-sound**: This prompt is used while asking for a conference pin.
- **bad-pin-sound**: This sound is played when an invalid PIN number is entered.
- **perpetual-sound**: A special setting-this plays a sound in a continuous loop forever. This can be used to broadcast sales or emergency messages to callers.
- **moh-sound**: A file or resource handle that plays a particular music-on-hold stream to the conference, when there is only one member in the conference. When a second member joins, this audio will stop, unless the mod-wait settings have been specified (as mentioned earlier).
- **max-members-sound**: If someone tries to join a conference that already has the maximum amount of members, this file is played.

caller-controls

Conferences allow caller controls groups which specify what commands are available to participants via touch-tones from within an active conference. Commands can include modifying the volume of the conference, mute/un-mute himself, or more advanced options such as playing menus to individuals or moving people from one conference to another.

Caller controls are based on pre-configured templates that are applied when a conference is started. For example, you can specify a group of controls that are available (such as the 0 key for mute, 1 to lower the volume, and 3 to increase the volume) and then apply those controls to three different conferences. The settings are applied when the conference begins (first caller joins) and remain the same for the duration of the conference (last caller left). You cannot have one party entering the conference with one set of controls and another party with another set of controls.

Do not name your caller-controls groups "**default**" or "**none**". **Those words are reserved** for the default key mappings or no key mappings, respectively.

Following is an example of the caller-controls configuration:

```
<caller-controls>
<group name="standard-keys">
<control action="vol talk dn" digits="1"/>
<control action="vol talk zero" digits="2"/>
<control action="vol talk up" digits="3"/>
<control action="transfer" digits="5"
data="100 XML default"/>
<control action="execute_application" digits="0"
data="playback conf_help.wav"/>
<control action="execute_application" digits="#"
data="execute_dialplan conference-menu"/>
</group>
</caller-controls>
```

The preceding example shows how to create a caller-controls profile named standard-keys. The keys 1, 2, and 3 lower, normalize, and raise the volume respectively, the key 5 transfers the caller who pressed the key to an extension 100, and the keys 0 and # each execute a specific Dialplan application.

caller-controls group "default" is automatically set to:

```
<caller-controls>
<group name="default">
<control action="mute" digits="0"/>
<control action="deaf mute" digits="*"/>
<control action="energy up" digits="9"/>
<control action="energy equ" digits="8"/>
<control action="energy dn" digits="7"/>
<control action="vol talk up" digits="3"/>
```

```
<control action="vol talk zero" digits="2"/>
<control action="vol talk dn" digits="1"/>
<control action="vol listen up" digits="6"/>
<control action="vol listen zero" digits="5"/>
<control action="vol listen dn" digits="4"/>
<control action="hangup" digits="#"/>
</group>
</caller-controls>
```

advertise

The advertise section of the conference configuration file allows you to generate presence events (advertisements) to services and subscribed parties via the FreeSWITCH event system. The idea is to set up permanent room names that generate presence events just like a phone or other device would. An outside program can then monitor whether a conference room is in use or not.

Advertise settings contains a room name in each element within advertise tags, as shown in this example:

```
<advertise>
<room name="888@$${domain}" status="FreeSWITCH"/>
</advertise>
```

Video-WebRTC conferencing setup in FreeSWITCH

Video conferencing setup is a superset of what we have seen for audio conferencing. Actually both audio and video conferencing configurations are read by the same mod_conference module.

Audio only callers can join video conferences, for example from PSTN (obviously they will only be able to listen and speak to the conference, they will not send or receive video streams). SIP callers can join with video, too, particularly from softphones and communication apps. But the queens and kings of FreeSWITCH videoconferences are the WebRTC clients, and between them the VERTO clients, that are able to tap into a series of advanced interactive functionalities. The Empress, the Khaleesi of VERTO clients is Verto Communicator, our flagship conferencing client for WebRTC browsers. We'll learn more about her later.

A fully configured FreeSWITCH video conference, with all the special VERTO features enabled, is also a perfectly legit audio conference, and if no caller ever joins that has video capabilities, that conference is indistinguishable from an audio only conference, no video streams will be created and no additional CPU will be used.

profiles additional video parameters

Inside "profile" sections in /usr/local/freeswitch/autoload_configs/conference.conf.xml file, we can add many parameters used specifically to control the video behavior of the conference and the additional features available to VERTO clients. Some of them are described below, for a complete list, and extended explanations, please consult our documentation online on http://freeswitch.org/confluence, searching for "mod_conference".

video-mode

The mode to run video conferencing in. passthrough is non transcoded video follow audio. transcode allows for better switching and multiple codecs. mux allows for multiple parties on the video canvas at the same time. Allowed values: mux, transcode, passthrough. Passthrough is obviously the lightest on CPU

Parameter syntax: <param name="video-mode" value="mux"/>

Default: passthrough

video-layout-name

The layout name from conference_layouts.conf.xml or group prefixed by "group:". Setting this setting will enable the video mux. Not setting this will switch video presentation based on floor.

Value can be in the form of "layout-name" or group:"group-name" (eg, group name must be preceded by "group:")

Parameter syntax: <param name="video-layout-name" value="2x2"/>

video-canvas-size

Pixel dimensions of the video mux canvas.

Value must be in the form of "integer"x"integer"

Parameter syntax: <param name="video-canvas-size" value="1920x1080"/>

video-no-video-avatar

Path to PNG file for members without video to display.

Parameter syntax: <param name="video-no-video-avatar" value="/tmp/novideo.png"/>

video-mute-banner

Sets the video mute banner text, font, font size, font color, and background color for the member. font_scale valid values 5-50, fg/bg hex color code, all settings besides text are optional.

Value can be in the form of "layout-name" or group:"group-name" (eg, group name must be preceded by "group:")

Parameter syntax: <param name="video-mute-banner" value="{font_face=/path/to/font.ttf,font_scale=5,bg=#000000,fg=#FFFFFF}VIDEO MUTED"/>

caller-controls additional video actions

- **vmute** Video mute. Toggle video from this member into the conference
- **vmute on** Disable video from this member into the conference
- **vmute off** Enable video from this member into the conference
- **vmute snap** Take a video snapshot for this user to be used when in vmute
- **vmute snapoff** Discard the vmute video snapshot

member-flags additional video options

- **join-vid-floor** Locks member as the video floor holder
- **no-minimize-encoding** Bypass the video transcode minimizer and encode the video individually for this member
- **vmute** Enter conference video muted

conference-flags additional video options

- **minimize-video-encoding** Use a single video encoder per output codec
- **livearray-json-status** activate transmission of a machine parseable live status of conference participants

screen estate layouts

The XML file /usr/local/freeswitch/conf/autoload_configs/conference_layouts.conf.xml contains the additional configuration where are specified the different ways to organize and populate the screen real estate during video conferences.

The moderator can change a video conference layout dynamically, during the conference, to adapt it to changing situations: during presentations you may want to have a little video stream of the presenter's face superimposed ("overlapped") to the screen she is sharing, so the effect is a little talking head as a picture-in-picture over the PowerPoint slides.

Layouts are defined by a name, the coordinates that divide the screen into areas, and the proportion the video stream will be scaled for each area. All numbers, both of coordinates and of scaling factors, are in the range 0-360 and proportional to the canvas. So, a layout that divides the screen real estate into 3 rows of 3 equally sized areas (eg, a 3x3 disposition) will work the same both with a canvas of 1920x1080 and with a canvas of 800x600.

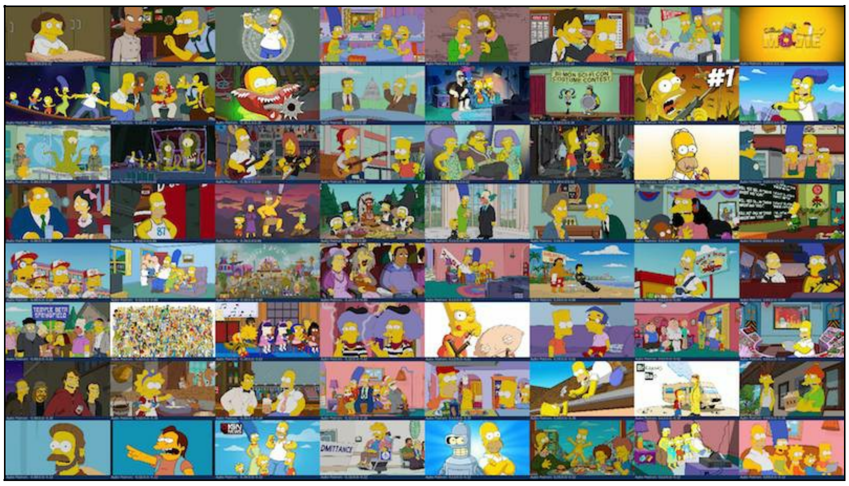
```
<layout name="3x3">
<image x="0" y="0" scale="120"/>
<image x="120" y="0" scale="120"/>
<image x="240" y="0" scale="120"/>
<image x="0" y="120" scale="120"/>
<image x="120" y="120" scale="120"/>
<image x="240" y="120" scale="120"/>
<image x="0" y="240" scale="120"/>
<image x="120" y="240" scale="120"/>
<image x="240" y="240" scale="120"/>
</layout>
```

layout groups

Layouts can be grouped in named "groups". If you give to a conference the vid-layout of "group:groupname", then FreeSWITCH will apply dinamically the group member more apt to display all participants.

```
<group name="grid">
<layout>1x1</layout>
<layout>2x1</layout>
<layout>1x1+2x1</layout>
<layout>2x2</layout>
<layout>3x3</layout>
<layout>4x4</layout>
<layout>5x5</layout>
<layout>6x6</layout>
<layout>8x8</layout>
</group>
```

Using the group in this example, when the first caller joins the conference, layout 1x1 will be applied. Then, as more members join, the layout will be dynamically chosen to accommodate all participants. When participants will grow from 36 to 37, FreeSWITCH will change layout from "6x6" to "8x8". Yay, on a 1920x1080 screen we can fit 64 conference participants live video streams!



FreeSWITCH conferencing operation

How to invoke conferences from dialplan? How to send an incoming call to join an already running conference? How to originate calls that will join the callees in the conference?
Also, conferences and video-conferences can be much more productive and interesting if participants avoid to talk one on top of each other, if video follow who is speaking, if a presenter is given possibility to show his screenshare or himself talking depending of the moment (after a while, any slide can be boring).

In next sections we'll look at how to invoke and manage running conferences.

Conference Application (dialplan)

Callers reach conferences via the conference application, which is usually invoked from the Dialplan or from the event socket via API calls. The general syntax for connecting a caller to a conference is as follows:

```
<action application="conference" data="confname@profilename"/>
```

confname is an arbitrary name for this conference room, and profilename is the profile to use from the conference configuration file (as specified earlier in this chapter). All callers that are sent to the same confname@profilename will be in the same room.

You can optionally pass specific parameters in to the conference by appending +PIN+flags at the end of a conference profile name, as shown in the following code:

```
<action application="conference" data="confname@profilename+12345+flags{
mute|deaf|waste|moderator|vmute}"/>
```

If you want to pass flags, but don't want to require a PIN, then conference invocation will be (note the double "+" sign):

```
<action application="conference" data="confname@profilename++flags{
mute|deaf|waste|moderator|vmute}"/>
```

Conferences are created on-demand when the first participant arrives in the bridge. Upon creation, the settings from the active profile, along with the specified conference PIN number, are recorded in memory with the conference. This is important to note because changes you load into memory won't take effect on in-progress conferences. For example, once a conference has been started with a PIN number, any future participants who join the conference must specify the same PIN number.

Conferences stay alive until the number of members drops to zero.

The following are some examples of values to specify in the data section when bridging a call to a conference:

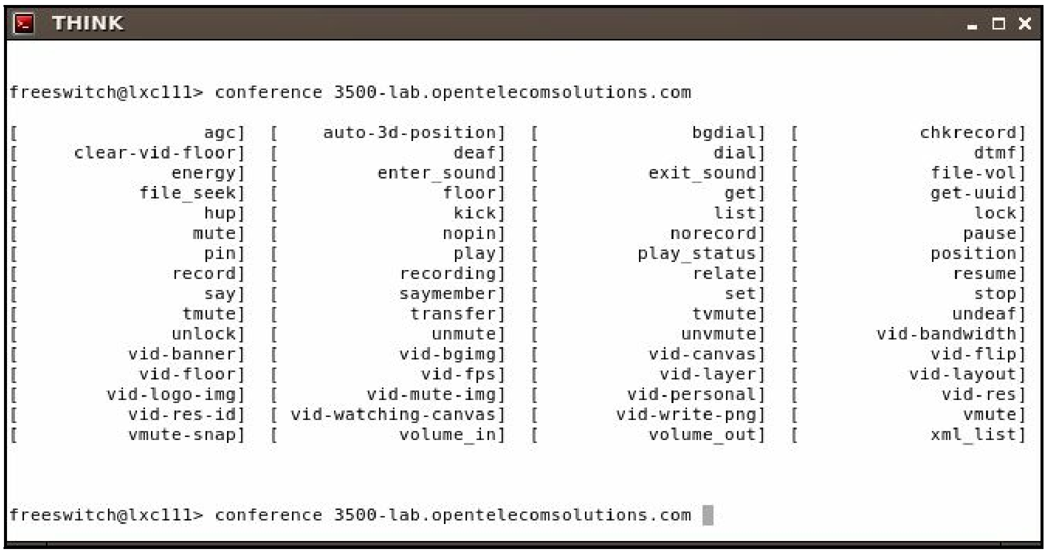
Action data	**Description**
confname	Profile is "default", no flags or PIN

confname+1234	Profile is "default", PIN is 1234
confname@profilename+1234	Profile is "default", PIN is 1234, no flags
confname@profilename++flags{mute|waste}	Profile is "default", multiple flags, no PIN
confname+1234+flags{mute|waste|vmute}	Profile is "default", multiple flags with PIN

Note that while some parameters are optional, their order is very important.

Controlling active conferences

A number of CLI and API commands exists for controlling an active conference. The most commonly used commands involve kicking members, adjusting volumes, adding a text banner to a participant video stream (eg, stating its name and role) and originating calls (to add people to a conference).



```
freeswitch@lxc111> conference 3500-lab.opentelecomsolutions.com

[                 agc]  [    auto-3d-position]  [              bgdial]  [           chkrecord]
[     clear-vid-floor]  [                deaf]  [                dial]  [                dtmf]
[              energy]  [         enter_sound]  [          exit_sound]  [            file-vol]
[           file_seek]  [               floor]  [                 get]  [            get-uuid]
[                 hup]  [                kick]  [                list]  [                lock]
[                mute]  [               nopin]  [            norecord]  [               pause]
[                 pin]  [                play]  [         play_status]  [            position]
[              record]  [           recording]  [              relate]  [              resume]
[                 say]  [           saymember]  [                 set]  [                stop]
[               tmute]  [            transfer]  [              tvmute]  [              undeaf]
[              unlock]  [              unmute]  [             unvmute]  [       vid-bandwidth]
[          vid-banner]  [           vid-bgimg]  [          vid-canvas]  [            vid-flip]
[           vid-floor]  [             vid-fps]  [           vid-layer]  [          vid-layout]
[        vid-logo-img]  [        vid-mute-img]  [        vid-personal]  [             vid-res]
[          vid-res-id]  [ vid-watching-canvas]  [       vid-write-png]  [               vmute]
[          vmute-snap]  [           volume_in]  [          volume_out]  [            xml_list]


freeswitch@lxc111> conference 3500-lab.opentelecomsolutions.com
```

In the picture are shown all the API commands available from FreeSWITCH console when we hit the TAB key twice after typing the word "conference" (first TAB will list the running conferences, just one, in our case; second TAB will show the available commands to interact with this conference). For example, you can change layout with:

```
conference 3500-lab.opentelecomsolutions.com vid-layout 3x3
```

or apply a caption on the bottom of video stream coming from participant with member_id "2" (note the single quote character to delimit the banner string):

```
conference 3500-lab.opentelecomsolutions.com vid-banner 2 'Giovanni Maruzzelli <gmaruzz@Open
```

For extended explanations, please consult our documentation online on http://freeswitch.org/confluence, searching for "mod_conference".

Calling outbound into the conference

FreeSWITCH can originate outbound calls to connect callees to a conference. This can be done with the application "conference_set_auto_outcall" and its helper channel variables.

But is probably much more useful when done by FreeSWITCH administrator (or the conference moderator, operating via some script) interactively in real time from fs_cli:

```
conference 3500-lab.opentelecomsolutions.com bgdial user/1010 +12125551212 weekly_conference
```

Verto Communicator and Video-WebRTC Conferencing

Verto Communicator is our flagship WebRTC client developed exactly to participate, manage and moderate FreeSWITCH videoconferencies. Under the slick User Interface lies all the power of VERTO protocol. VERTO, a protocol designed to be easy of use for web developers, is based on JSON. JSON + WebSockets means a bidirectional capability of real time information exchange. This is only evident when you see Verto Communicator (VC) updating its chat and participants arrays, or when you see the icons of each single participant to blink when she talk, and its microphone and camera change color when she mutes herself.

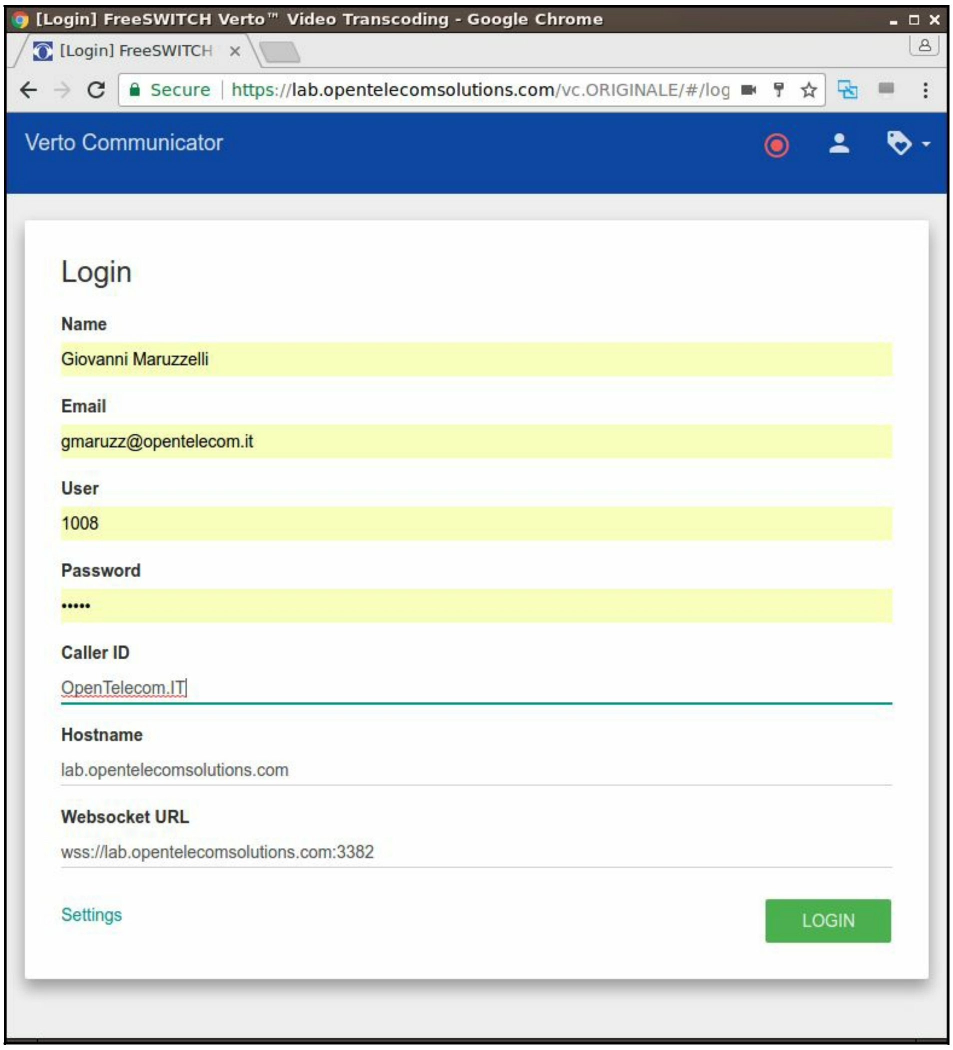
VC allows the participants of taking care of their own settings like outbound and inbound volume, audio mute, video mute, chosing which microphone and webcam to use (important when you have a laptop that doubles as workstation, lying closed on the side of the desk attached to a gorgeous USB 3.0 powered hub, and a displayport triplicator).

And VC exposes in its GUI all the administrators' tools: kicking, muting, vmuting, giving floor, adding captions, taking snapshot images, recording session, playing audio/video files, changing layouts, all with just a mouse click.

For installation of the various components needed to provide WebRTC videoconferences, and VC itself, please refer to the installation paragraphs of "WebRTC, SIP and Verto" in this book.

settings

When you first connect to a Verto Communicator page, you will be presented with a little window that asks you your name (so it can later appear in the conference participant roster list and in the chat window). If you instead click on the "setting" link, that window expands, and you can enter all connection details, included to which server connect, authorization username, and password.


[Login] FreeSWITCH Verto™ Video Transcoding - Google Chrome
[Login] FreeSWITCH
Secure
https://lab.opentelecomsolutions.com/vc.ORIGINALE/#/log
Verto Communicator

Login

Name
Giovanni Maruzzelli

Email
gmaruzz@opentelecom.it

User
1008

Password
•••••

Caller ID
OpenTelecom.IT

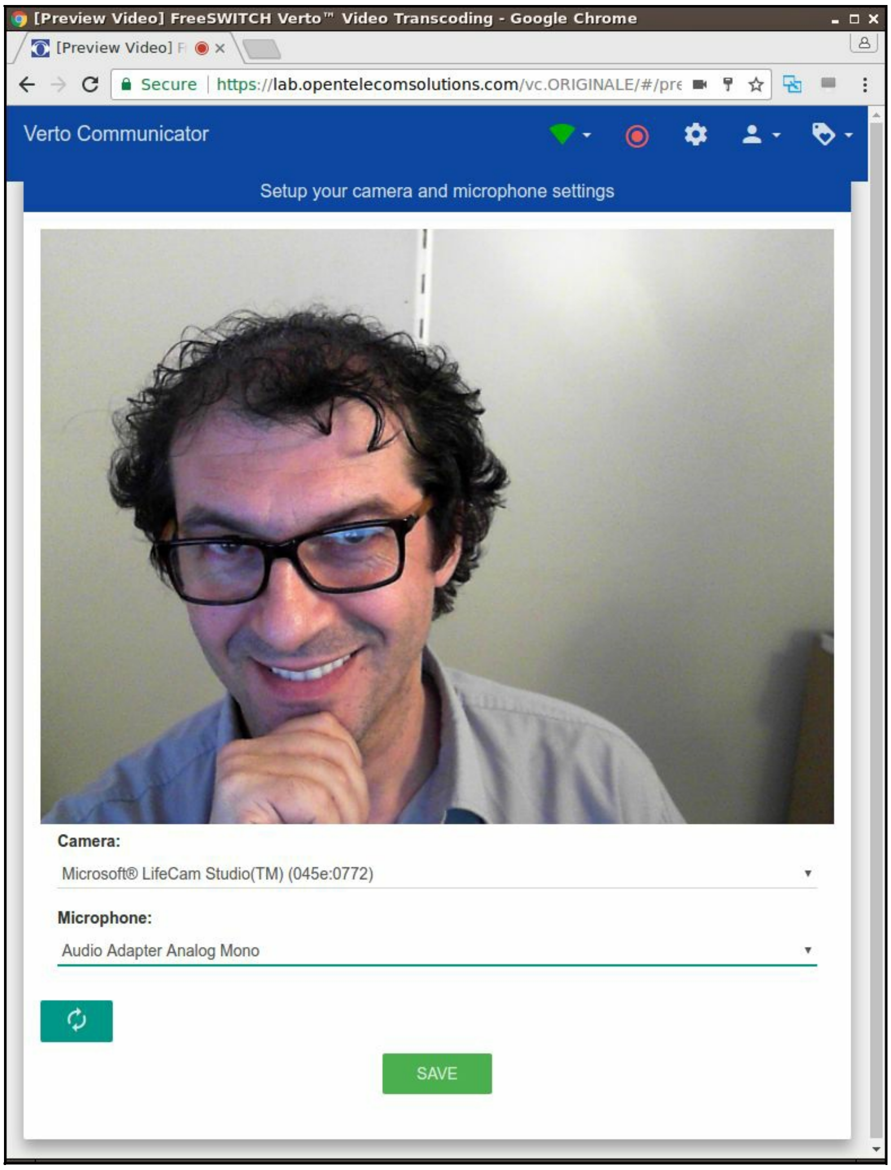
Hostname
lab.opentelecomsolutions.com

Websocket URL
wss://lab.opentelecomsolutions.com:3382

Settings

LOGIN

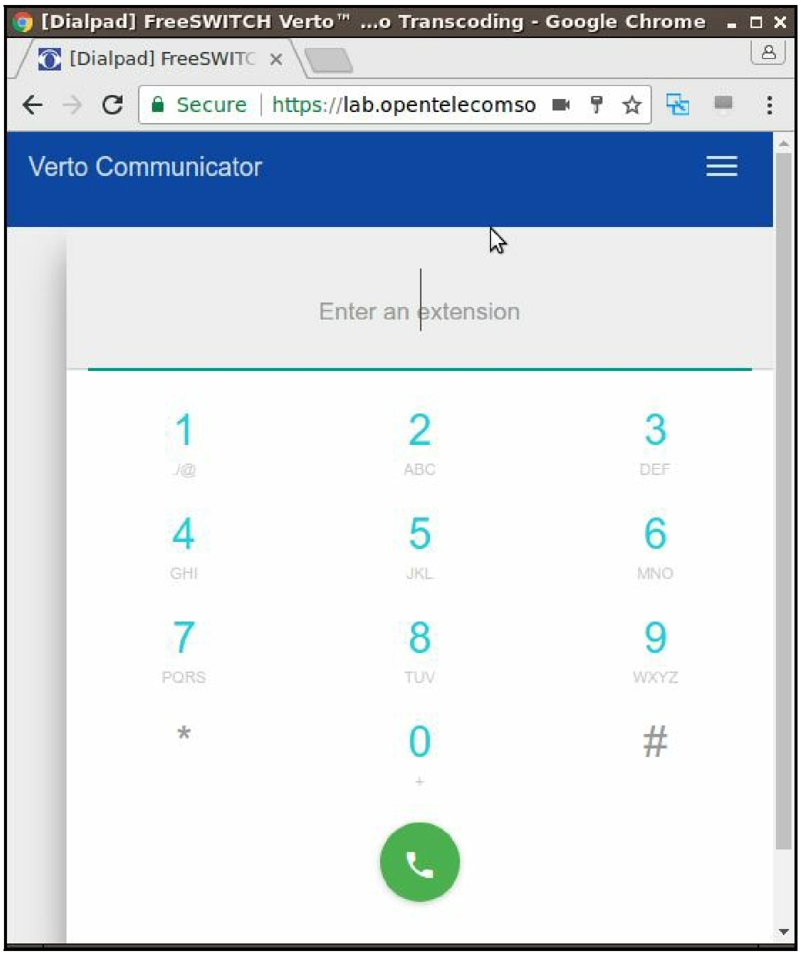
You will then be offered the possibility to choose which multimedia sources to use, as in which microfone, camera and speakers (you may easily have tree audio sources to choose from: laptop built-in, USB soundcard, microphone inside the webcam)


[Preview Video] FreeSWITCH Verto™ Video Transcoding - Google Chrome
Secure
https://lab.opentelecomsolutions.com/vc.ORIGINALE/#/pre
Verto Communicator
Setup your camera and microphone settings

Camera:
Microsoft® LifeCam Studio(TM) (045e:0772)

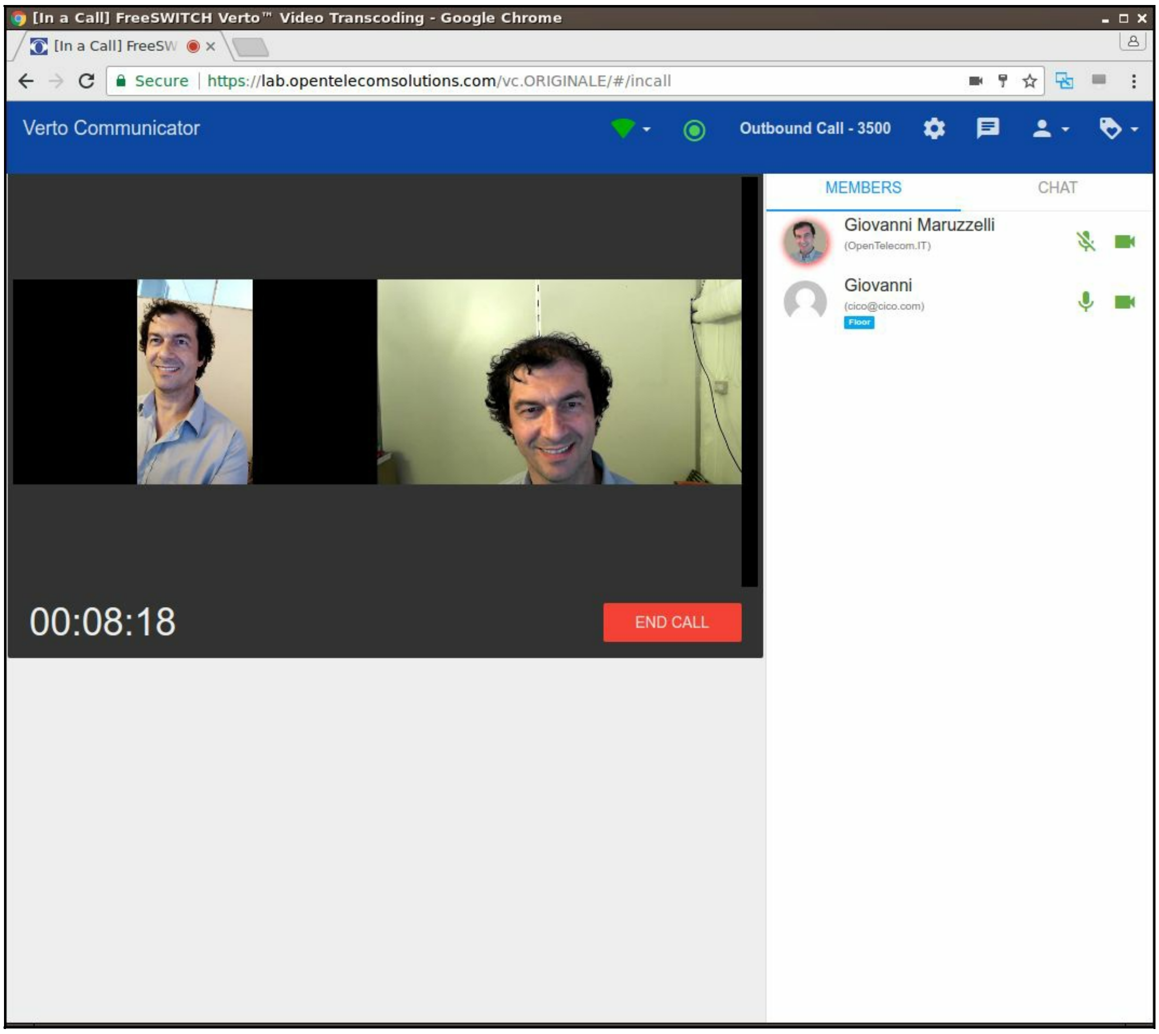
Microphone:
Audio Adapter Analog Mono

Now you're setup and logged in, you may dial:


[Dialpad] FreeSWITCH Verto™ ...o Transcoding - Google Chrome
[Dialpad] FreeSWITC
Secure
https://lab.opentelecomso
Verto Communicator
Enter an extension
1
./@
2
ABC
3
DEF
4
GHI
5
JKL
6
MNO
7
PQRS
8
TUV
9
WXYZ
*
0
+
#

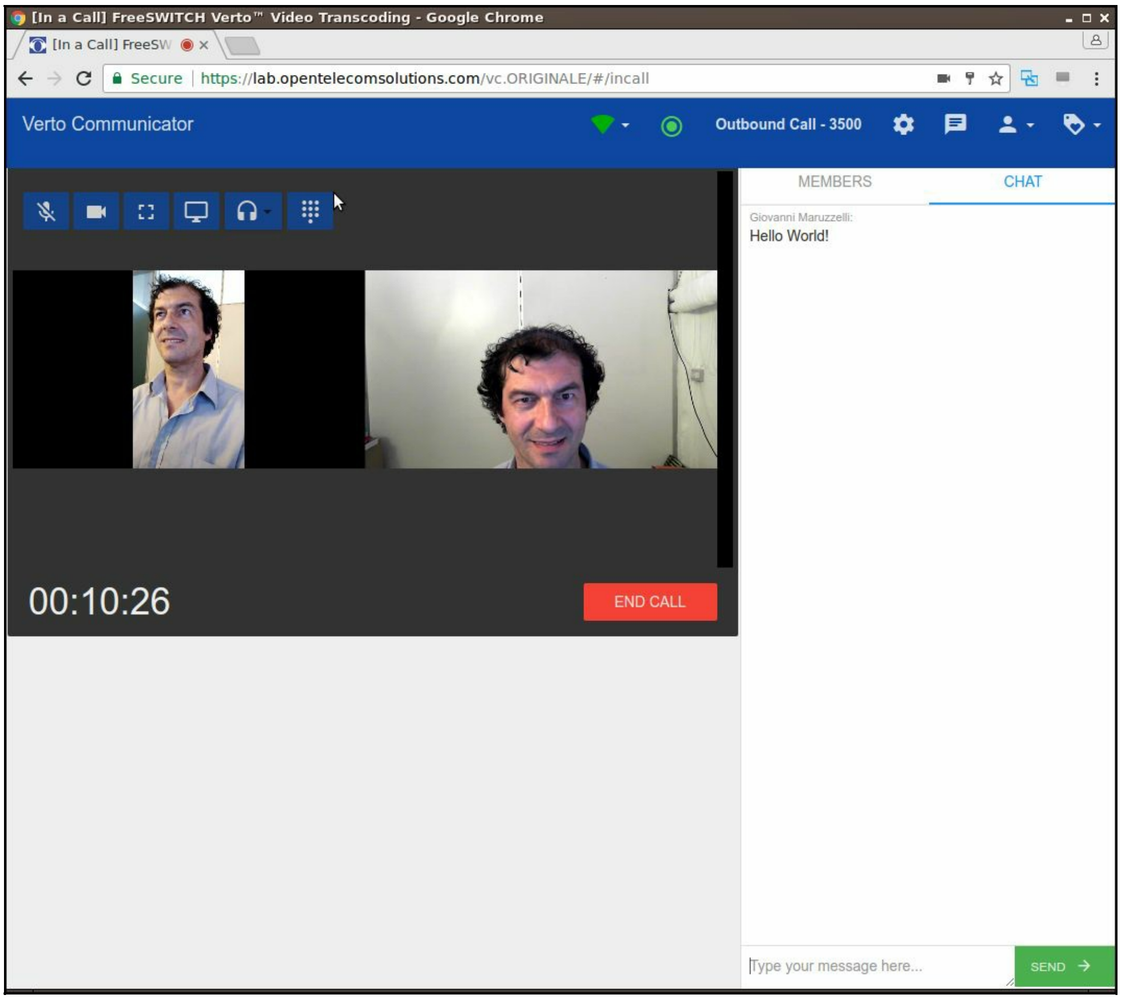
participants list

After joining the conference, you'll see on the right the member roster with interactive avatars, and live icons.



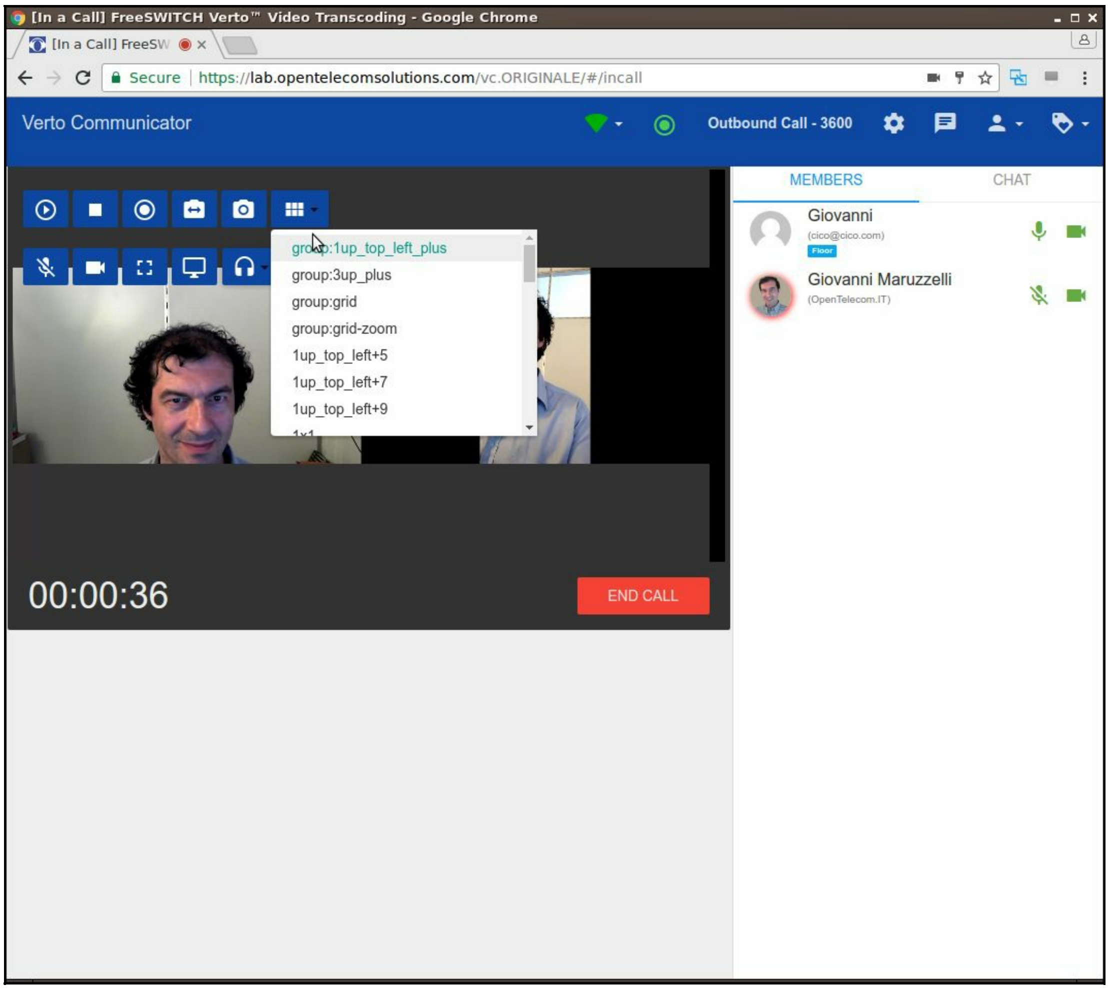
participant GUI commands and chat

Clicking on "chat" tab, you'll see the real time synched chat window, while if you hover on the main window, the user commands will apper as On Screen Display : microphone toggle, webcam toggle, fullscreen, screen share, audio output selection, and dialpad.


[In a Call] FreeSWITCH Verto™ Video Transcoding - Google Chrome
[In a Call] FreeSW
Secure
https://lab.opentelecomsolutions.com/vc.ORIGINALE/#/incall
Verto Communicator
Outbound Call - 3500
MEMBERS
CHAT
00:10:26
END CALL
Type your message here...
SEND

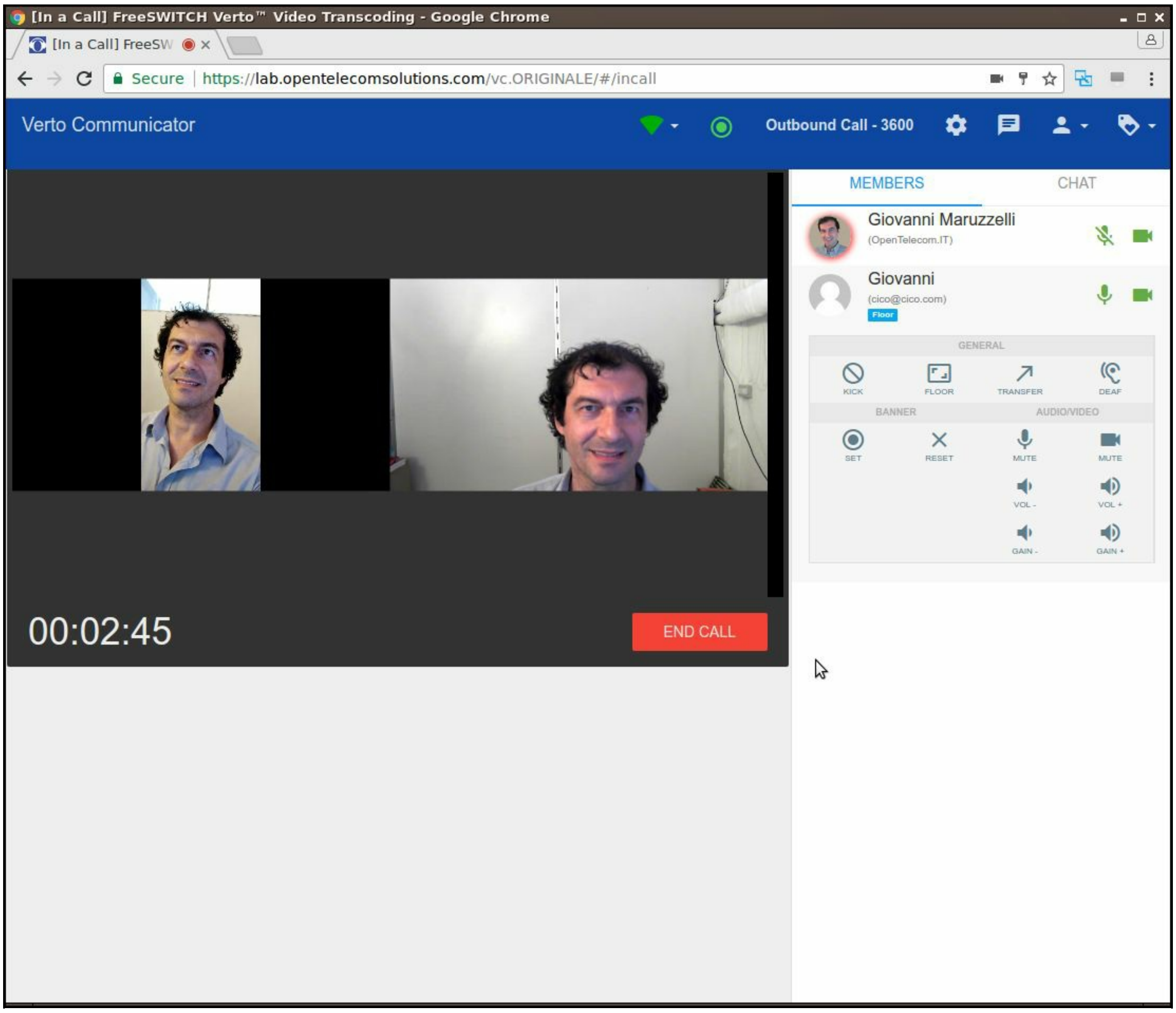
Giovanni Maruzzelli:
Hello World!

moderator GUI commands



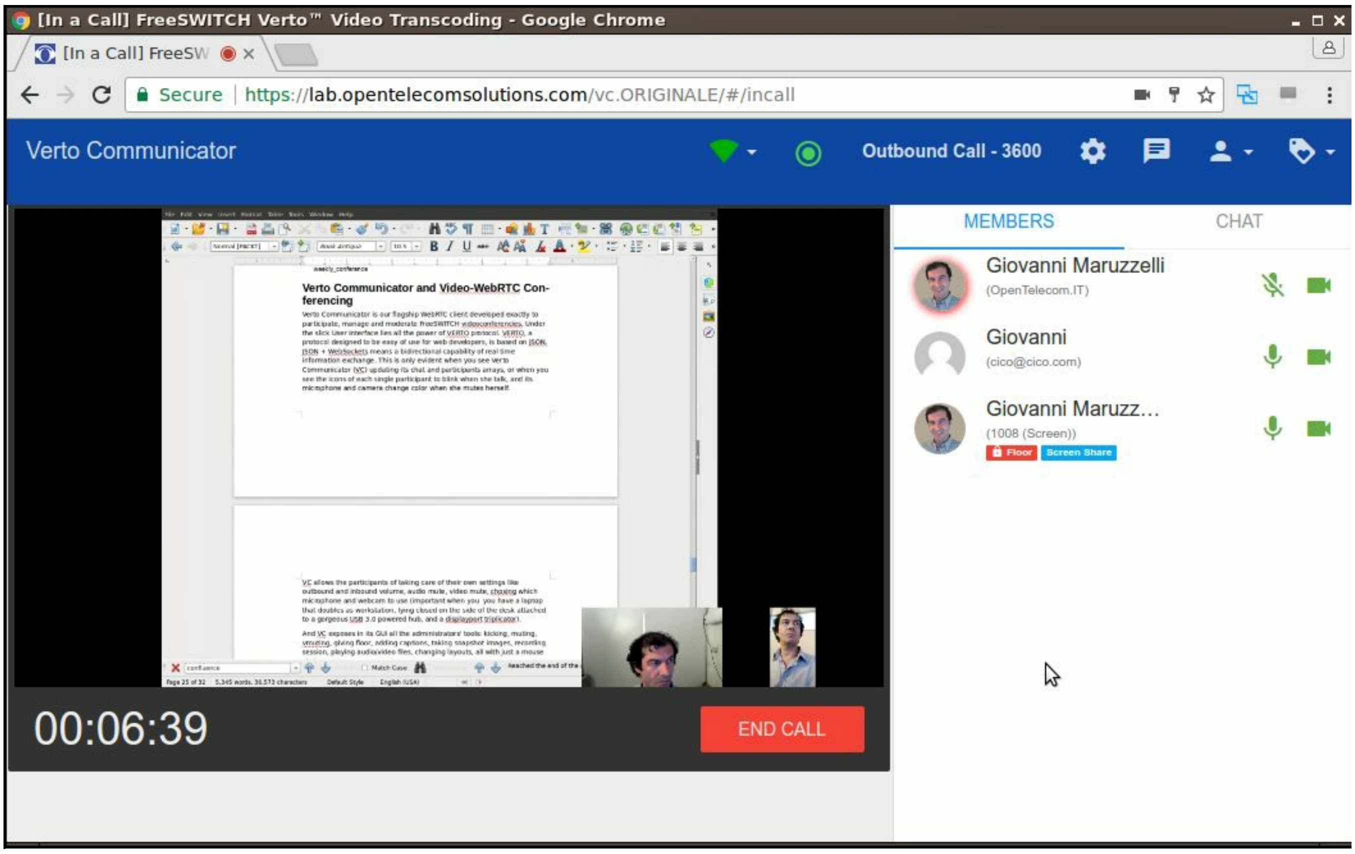
If you are connected as one of the conference moderators, then you have much more controls on screen: in addition to the participant controls, moderators have the recording/stop recording, the play media/stop playing, the take a snapshot image and a dropdown menu for changing screen layout.

Also, conference moderators can click on the icon on the right to toggle audio and video to each participant, add a caption (like name and role), give the floor (the conference focus), etc


[In a Call] FreeSWITCH Verto™ Video Transcoding - Google Chrome
[In a Call] FreeSW
Secure | https://lab.opentelecomsolutions.com/vc.ORIGINALE/#/incall
Verto Communicator
Outbound Call - 3600
MEMBERS
CHAT
Giovanni Maruzzelli
(OpenTelecom.IT)
Giovanni
(cico@cico.com)
Floor
GENERAL
KICK
FLOOR
TRANSFER
DEAF
BANNER
AUDIO/VIDEO
SET
RESET
MUTE
MUTE
VOL -
VOL +
GAIN -
GAIN +
00:02:45
END CALL

screen sharing

Sharing your own entire screen, or just one application window, is very easy. Just click on the "screenshare" participant control, and choose from the popup what you want to share. I decided to share the LibreOffice Writer window where I'm writing this chapter. Also, I changed the conference layout to "overlaps", and now conference participants are little picture-in-picture live videos superimposed to the live video from my application window (which can be a slide show, a spreadsheet, a terminal, whatever).



Verto Communicator and Video-WebRTC Conferencing

Verto Communicator is our flagship WebRTC client developed exactly to participate, manage and moderate FreeSWITCH videoconferencies. Under the slick User interface lies all the power of VERTO protocol. VERTO, a protocol designed to be easy of use for web developers, is based on JSON. JSON + WebSockets means a bidirectional capability of real time information exchange. This is only evident when you see Verto Communicator (VC) updating its chat and participants arrays, or when you see the icons of each single participant to blink when she talk, and its microphone and camera change color when she mutes herself.

VC allows the participants of taking care of their own settings like outbound and inbound volume, audio mute, video mute, choising which microphone and webcam to use (important when you you have a laptop that doubles as workstation, lying closed on the side of the desk attached to a gorgeous USB 3.0 powered hub, and a displayport triplicator).

And VC exposes in its GUI all the administrators' tools: kicking, muting, vmuting, giving floor, adding captions, taking snapshot images, recording session, playing audio/video files, changing layouts, all with just a mouse

Summary

In this chapter we learned about the conferencing capabilities of FreeSWITCH. After a brief overview of audio and video conferencing techniques, we saw how they are supported by FreeSWITCH.

We learned how to setup audio conferences in FreeSWITCH, with their multiple features. And how to extend that setup to provide the newest video capabilities, and WebRTC interactions.

Then we saw how to manage and operate a conference from FreeSWITCH console, how to call into it from dialplan, and how to call out from it interactively.

We closed this chapter with a visual presentation of Verto Communicator features and functionalities for both conference participants and moderators.

We're ready to implement leading edge conferencing services for our users!

In next chapter we'll learn all about the dreaded enemy of telco implementors: NAT, and how FreeSWITCH has been designed to pass through it.

Handling NAT

NAT (Network Address Translation) is the most notable legacy of a very different time: when the Internet was not widely adopted, and dinosaurs were roaming free. At that time, like... 20 year ago?, there was plenty of available Internet Addresses (was normal to be assigned a class C network with 254 public Internet routable addresses. I personally got two class-C, for iol.it and for matrice.it) and no one had any idea the ipv4 address pool will ever be exhausted. So, most machines on the Internet were there with their own public address. Also, no problems with security, encryption, and all that. There was no money on the Internet, so no crime.

Telecommunication was a matter of having the best way to connect two public hosts peer-to-peer. SIP was designed with that in mind, at beginning. So, it was completely out of its natural environment when in few years most of the hosts using VoIP would find themselves behind a firewall and a NAT. Different IP addresses if you see the machine from inside of the private network, or from the Internet side. Those addresses are automatically translated into each other at the lowest level level of data packets (0s and 1s) by the routers or the "modems". Problem is, SIP needs a text part (SDP) where data are written before going out in the Internet. Those written down text addresses describe the media streams. And text is not translated by routers (and when routers try, as in ALG, they make even more damages). That's why so many times you end up with a call that rings, your peer can answer, but there is no audio, or no video, or one way only, or it drops after 30 seconds (timeout).

In this chapter we will discuss:

- A brief introduction to NAT, including a little history
- The four pitfalls of NAT
- The settings in FreeSWITCH that help overcome NAT
- Troubleshooting tips

A brief introduction to NAT

A good way to explain NAT to someone who could absolutely care less about techno-babble would be with an analogy. Think of a giant office building and its mailroom. An employee on the 10th floor sends a package to you by dropping it off at the mailroom on the ground floor. The package is passed on to the Postal service and it arrives at your house. The return address on the package is actually the address of the entire office building and not the tiny office on the 10th floor. Now say you need to return the package. You put it back through the Postal system and it arrives at the building and the employees in the mailroom must figure out where to deliver the package by mapping your name or office number to the location in the building, and then they take it back up to the employee on the 10th floor. The mailroom is like a NAT router because it proxies the mail between the actual Postal system and the one inside the building. The offices are like the LAN addresses because they do not have any direct access to the mail. What if the name of the office sending the message is messed up or not present on the package and the mailroom employees have no idea which office to send the package to when you return it? This would be a NAT problem and the package may end up getting lost just like your calls. Perhaps you get the package and notice it's missing the office name or number but you know which office sent it because you were expecting this package and then when you return it you write the office number on the label? This would be an ANTI-NAT feature created by you.

When it comes to networking, NAT is basically a technique where an entire LAN or Local Area Network (meaning a network that is not directly connected to the Internet) is connected to a device that does have access to the Internet and uses a single public IP address (meaning an IP that is directly connected to the Internet) to provide Internet connectivity to the entire LAN. It's used primarily to reduce the number of public IP addresses that are necessary since we are running out of them fast. We started out with four billion of them and they are all basically used up at the time of writing this.

Putting your local network behind NAT has a side effect of constantly protecting your computers and other devices from attack since they are not visible on the Internet. Experts do not feel this is the ultimate solution for security because there are still ways to compromise devices behind NAT but consider it a bonus protection when used together with other good security practices. By the way, you'll learn more about security when it comes to VoIP in Chapter 15, *VoIP Security*.

Understanding the evolution of NAT

The demand for IPv4 addresses has grown over the years since the Internet has evolved. As the demand increased, the pool of available addresses has been depleted and there is a shortage of available IPv4 addresses. Two major attempts to deal with this situation have become popular over time - NAT and IPv6.

NAT has become a fairly popular way to take a small subset of public IP addresses and utilize them across a larger number of devices on a network. NAT took off in the 1990s as a way to mitigate the IP starvation problem until IPv6 took off, but now it's so popular that many people don't want to let go of it. Meanwhile, system administrators are forced to embrace the looming legacy of NAT and make sure that we can tolerate it across our software and equipment due to popularity.

A new standard for IP addresses called IPv6 can solve the IP starvation problem by adding so many public IP addresses that we can have trillions of IP addresses for every square inch of the surface area of the Earth. We could give a block of IPs, the size of the entire Internet as we know it today, to every creature on the planet and not even put a dent in the total available pool of IPv6 addresses. The IPv6 specification was published in 1998 and has been slowly gaining momentum. IPv6 still takes a back seat to the more widely used IPv4 that was adopted in the 1970s. Most likely, even if we fully adopt IPv6, NAT still won't go away for some time.

The key problem we are trying to solve with NAT and VoIP is that since a device (phone) behind NAT is not visible to the Internet, it becomes difficult to contact that device when you want to call it. The next big problem is that some protocols, such as SIP, may break when used over NAT.

The four pitfalls of NAT

There are four basic pitfalls of NAT that everyone should learn. Understand these pitfalls and you will be well-equipped to handle the NAT scenarios that you'll no doubt face:

- NAT can be there even when you don't know about it. The Internet does not have to be involved.
- Any two techniques to defeat NAT used together will cancel each other out.
- Some devices use a SIP ALG (Application Layer Gateway) to defeat NAT. They even do that without telling you.
- NAT correction techniques (like ALG) can falsely identify a situation and actually make things even worse.

Become familiar with these pitfalls. They are referenced frequently throughout this chapter.

Let's discuss each of these in more detail:

- **NAT can be there even when you don't know it. The Internet does not have to be involved:** If you are using home Internet service from your cable or telephone company, or even in some cases a business-class service, they may on occasion use NAT to put all of their customers in a separate network and then translate that network to other segments in their infrastructure. This could happen not just once but multiple times between your device and its destination and you have no control over it. This can cause some real problems for people trying to use VoIP. Most VoIP protocols only have basic provisions for dealing with NAT and often fall short. This is probably the first problem that most home-users will encounter when trying to use VoIP from their homes. NAT can also be used in this fashion inside a LAN connecting multiple LANs without actually reaching the Internet. Getting on the Internet is just the most popular use for NAT but it can be used just as well to isolate one LAN from another. If you're asking your friendly neighborhood VoIP guru for help and he suggests a NAT problem, don't count it out just because you are not using the Internet or because you don't know NAT is there.
- **Any two techniques to defeat NAT used together will cancel each other out:** This one is tricky and a very popular issue among VoIP users. The best way to visualize it is to picture a game of Othello. Whenever you make a move to block the NAT it flips everything around. If you make a counter move, it flips it all back. This might even be happening more than you think (see the first pitfall). As long as it's an odd number of flips and you started out with a non-working

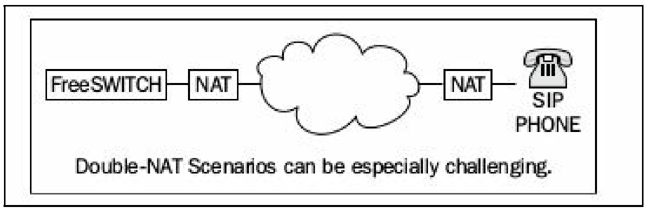
situation you should end up okay, but you should make it a point to do only the most minimal modifications possible to avoid confusion and pain. If your phone supports NAT features and you enable them and also enable them on FreeSWITCH, you may end up with one-way or no audio. What's even more confusing is that there are so many ways to cancel out NAT. Some require only changes on your phone behind the NAT, while some require changes only on FreeSWITCH and some require a change on both ends.

- **Some devices use a SIP ALG to defeat NAT**: "Arrrghh, curse you SIP ALG!". We've heard that being exclaimed countless times over IRC or on a community conference call. ALGs mean well but they usually mess things up real bad. They are like a combination of the first two pitfalls because they are usually implemented inside your provider or in your network router and enabled by default without your knowledge. They do the worst and last resort of all the ANTI-NAT techniques that is modifying the text of SIP packets as they pass through your router. This can lead to misbehaviors and misrouted traffic that will present itself to you as a complete mystery. Heed my words. If you find yourself uttering the phrase, "This makes absolutely no sense", the first thing you should check is to see if you are under the evil spell of a SIP ALG. In many cases, simply turning off a SIP ALG resolves NAT-related issues.
- **NAT correction techniques can falsely identify a situation and actually make things even worse**: It helps to understand your surroundings at least enough to know if you actually need to enable ANTI-NAT features. Some SIP agents make use of the more arcane aspects of SIP and do really fancy things with the network addresses in the packet. For those of you unfamiliar with SIP, yes I know it's all arcane but we need to keep things in perspective. So the problem is, completely legitimate packets that are just doing things in a way that resemble NAT can trigger some of the features we use to detect NAT. So you need to be careful, especially with Cisco phones that are notorious for being bad behind NAT and subject to false detection at the same time.

Demystifying NAT settings in FreeSWITCH

Now that we have reviewed the common pitfalls of NAT, we can go over the various types of NAT situations that you may encounter. Basically you will probably be in a situation where either your phone or PBX is behind NAT talking to a SIP endpoint that is not behind NAT (or vice versa). Even worse, you might end up in the dreaded double-NAT situation where both sides of a connection are independently behind their own individual NAT routers at the same time.

A double-NAT scenario looks like the following diagram:



Let's start with a sane, yet challenging situation where you have a phone at your house that can't understand NAT and you want to register to your FreeSWITCH server that is on the public Internet. The good news is that this situation is already covered for you by the example FreeSWITCH demo configuration. The core of FreeSWITCH has a feature called **ACL (Access Control Lists)**. An ACL lets you create lists of network addresses and control access to things depending on whether or not a particular device is originating from an address defined by the ACL. There is equal value in determining if an address matches or does not match a list and from there deciding if being (or not being) on the list is a good or bad thing.

This feature makes it possible to allow certain devices to authenticate based on their IP address or you can make a list of enemy devices so you can completely block anyone who is on the list. In this case we will use the ACL to determine if a device is behind NAT or not and decide what to do from there.

A device behind NAT is likely to have an IP address within a special range called RFC-1918. The easiest explanation for this is that there is a special set of IP addresses that never lead to the Internet because they are reserved for private use on LANs. This is basically any IP address that starts with 192.168.x.x, 172.16.x.x through 172.31.x.x, or 10.x.x.x. We'll just call them LAN addresses from now on. People call them also as per the RFC, eg "rfc1918 addresses".

Because these addresses are private, there can be endless networks using the same exact IP addresses internally but they can never connect to each other. Now when you connect these networks to a NAT router, all the phones on these private networks will be able to reach your FreeSWITCH server. The routers work by keeping track of all the traffic coming from the LAN addresses and sending it to the Internet as if it was coming from the public Internet IP on the router. Then, when the destination on the Internet sends a response to the NAT router, it uses the mapping to deliver the packet back to the sender. The source address that FreeSWITCH sees the traffic originating from may never be the same and this makes it very difficult to send an incoming call to the phone. This is where the ACL comes in handy.

FreeSWITCH has a configuration parameter for mod_sofia profiles called **apply-nat-acl**. This parameter can be used more than once in the same profile and expects the name of an ACL list. When mod_sofia gets SIP REGISTER or INVITE packets, it looks at the contact address and checks the IP referenced in the Contact header against the specified ACL. If there is a match, it concludes that the device must be behind NAT. It's difficult to tell which IPs represent devices behind NAT but we have a bit of a clue. Remember RFC-1918 or LAN addresses as we called them? Since it's a defined range of IP addresses, we can conclude that if you are coming from one of these addresses then you are calling from behind NAT.

Be careful! Don't forget the fourth pitfall, **it's not 100 percent safe to assume every device coming from a LAN address is behind NAT**. This may not always be the case but **more times than not it is**. It's just good to be wary. One case where it may not be true is when FreeSWITCH also has a LAN address because it's behind NAT too. Well, we have a special ACL that is created for you when FreeSWITCH starts, called **nat.auto**. This special ACL already contains the entire RFC-1918 address space but it also checks the machine's local network address and excludes that address space so you **won't get any false positives** when it gets calls from phones on the **same LAN as FreeSWITCH**. At the same time this ACL can detect a phone that is actually in a remote location behind NAT. FreeSWITCH comes **pre-configured with apply-nat-acl set to nat.auto** and can correct most typical device behind NAT versus FreeSWITCH on the public Internet situations.

How do we solve the problem? Basically, when the phone registers from behind NAT and it's detected, we save the IP and port that we saw the register originate from and store it in the internal database alongside the unreachable LAN address that the unsuspecting phone has provided us. When we need to contact this phone, we consult our database and determine the external IP:port to which the message should be sent. The SIP headers will still have the internal IP:port values that the remote phone is expecting to see. We also tell the phone to register more frequently so that the mapping stays open, since most NAT routers only hold a translation path open for a short period of time. This technique is especially effective in avoiding the other part of the fourth pitfall because we never modified the intended address at all like the evil ALG.

Here is an example of output from the FreeSWITCH CLI (Command Line Interface). The client is a softphone behind NAT registering to an instance of FreeSWITCH running on a public IP. Notice the Contact field is using the IP 10.0.1.85 that is a LAN address. The status shows that UDP-NAT was detected thanks to the nat.auto ACL list. The trick comes in at the end of the Contact. The extra parameters fs_nat and fs_path are appended to the Contact address of the phone registration so we can figure out how to circumnavigate the NAT. Consider the following:

```
fs_nat=yes
fs_path=sip%3A1006%40206.22.109.244%3A43425%3Brinstance%3Db67dbafc
 9baa9465%3Btransport%3Dudp.
```

The fs_path field is a SIP URI (Uniform Resource Identifier) that actually will lead back to the phone through NAT. It's URL-encoded, so special characters in the URI do not conflict with the real contact. The decoded version of this field is:

```
sip:1006@206.22.109.244:43425;rinstance=b67dbafc9baa9465;transport=udp
```

So even though when we call the phone we will send the INVITE network packet to 206.22.109.244:43425, we will keep it addressed (in the text of the packet) to 10.0.1.85:5060 and that is exactly where it will end up once the NAT translation takes place and the remote router delivers it to the phone. You can see the complete set of registration information using the sofia status profile internal reg command. The following is an example:

```
freeswitch@myhost> sofia status profile internal reg
Registrations:
=================================================================
Call-ID:        ZWU1MjdiZTI2MTg2MmVhNTc5NTk3MDY5YjFmOTVkMTU.
User:           1006@myhost.freeswitch.org
Contact:        "TEST"<sip:1006@10.0.1.85:10118;rinstance=b67dbafc9baa9465;transport=udp;fs_r
Agent:          eyeBeam release 1104g stamp 54685
Status:         Registered(UDP-NAT)(unknown) EXP(2012-12-09 10:18:07) EXPSECS(88)
Host:           myhost
IP:             206.22.109.244
Port:           43425
Auth-User:      1006
Auth-Realm:     myhost.freeswitch.org
MWI-Account:    1006@myhost.freeswitch.org
Total items returned: 1
=================================================================
```

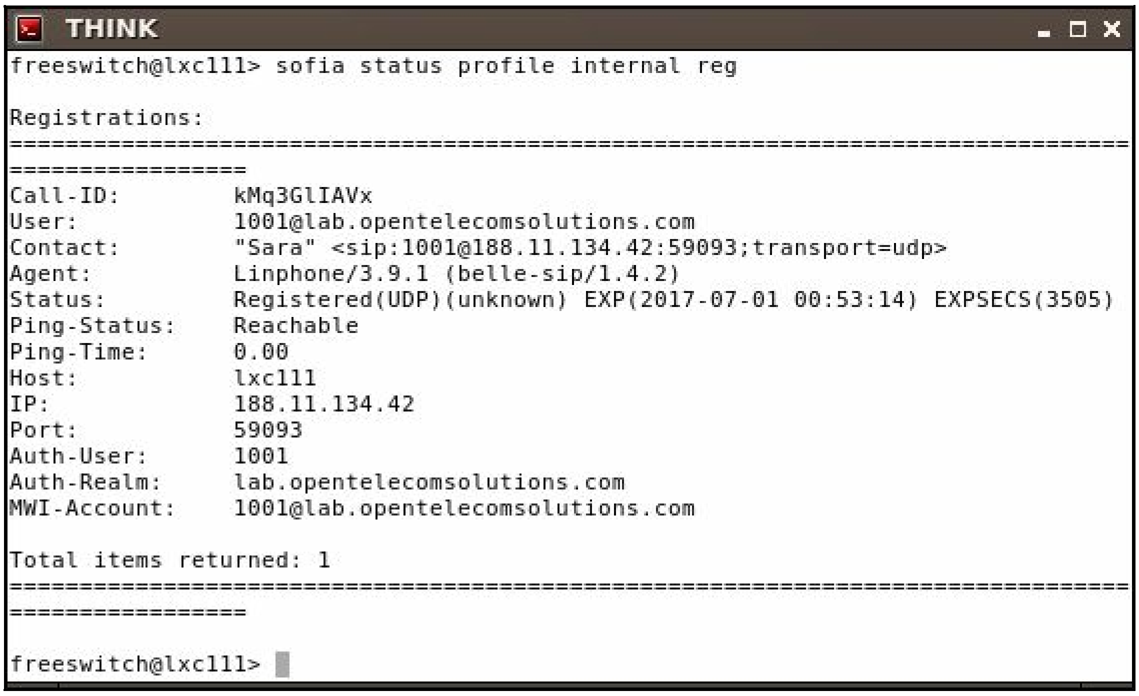
Notice that the Contact: header contains both the fs_nat and fs_path parameters. Any SIP traffic that FreeSWITCH needs to send to user 1006 will use the URI specified in the fs_path parameter.

Following we have another example, with a more modern client and in the popular situation where both the client and the server are behind different NATs. In this case the client is able to correct its own address using the data sent to him by FreeSWITCH.

Our Linphone send the following as first REGISTER SIP packet:

```
REGISTER sip:lab.opentelecomsolutions.com SIP/2.0
Via: SIP/2.0/UDP 192.168.1.200:5060;branch=z9hG4bK.znT45Avvt;rport
From: "Sara"<sip:1001@lab.opentelecomsolutions.com>;tag=cfurzFSzg
To: "Sara"<sip:1001@lab.opentelecomsolutions.com>
CSeq: 20 REGISTER
Call-ID: kMq3GlIAVx
Max-Forwards: 70
Supported: replaces, outbound
Accept: application/sdp
Accept: text/plain
Accept: application/vnd.gsma.rcs-ft-http+xml
Contact: <sip:1001@192.168.1.200;transport=udp>;+sip.instance="<urn:uuid:ba010749-6214-4a35-
Expires: 3600
User-Agent: Linphone/3.9.1 (belle-sip/1.4.2)
```

You see in the original "Contact" header the address is 192.168.1.200, the private LAN address. But then, after some packet exchanges needed to complete the SIP registration, Linphone sends FreeSWITCH the right "Contact", and we end up with:



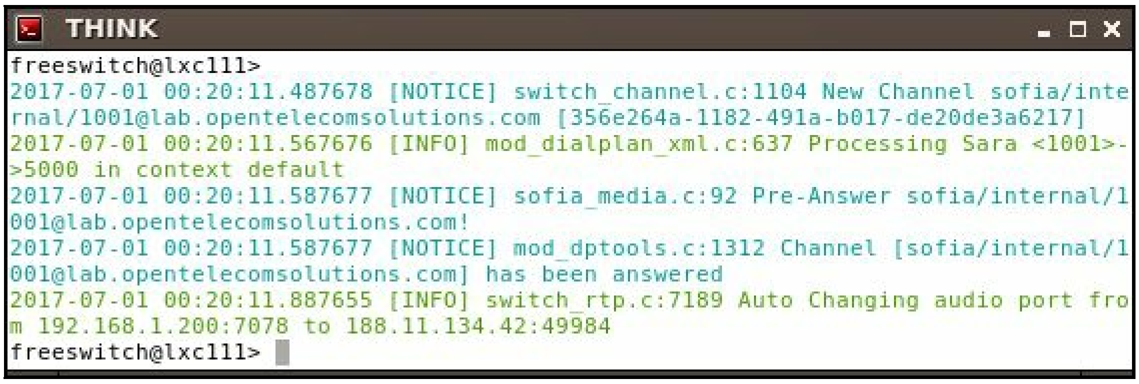
```
freeswitch@lxc111> sofia status profile internal reg

Registrations:
=================================================================================
=================
Call-ID:        kMq3GlIAVx
User:           1001@lab.opentelecomsolutions.com
Contact:        "Sara" <sip:1001@188.11.134.42:59093;transport=udp>
Agent:          Linphone/3.9.1 (belle-sip/1.4.2)
Status:         Registered(UDP)(unknown) EXP(2017-07-01 00:53:14) EXPSECS(3505)
Ping-Status:    Reachable
Ping-Time:      0.00
Host:           lxc111
IP:             188.11.134.42
Port:           59093
Auth-User:      1001
Auth-Realm:     lab.opentelecomsolutions.com
MWI-Account:    1001@lab.opentelecomsolutions.com

Total items returned: 1
=================================================================================
=================

freeswitch@lxc111>
```

Making media flow

Now that the SIP messages are flowing properly from FreeSWITCH to the phone, what about the media? A phone call or a videocall is not very eventful if you can't even hear each other, right? We had many problems where the **calls would set up properly until** the point where **NAT would strike the RTP** packets that provide the actual media of the call, rendering the call with **one-way-audio or even no-way-audio** in some cases. In light of this injustice, we created a separate feature that is always enabled and only needs to be manually disabled in a very few set of cases inspired by the fourth pitfall. This feature is called **RTP auto-adjust**. The reason we need it is because when the phone tries to call us from behind NAT, it will naively advertise its unreachable LAN private address to FreeSWITCH as to where to send the media streams (eg the **address written in the text of SDP**).

We could guess that since the device is behind NAT, we should really send the audio to the same address that we saved from the SIP message. But that is not always the case since various types of NAT have restrictions and the port mappings sometimes don't exist until the device behind NAT has actually sent a packet. So, in reality **we may have no idea** whatsoever as to **how to successfully get audio flowing to the phone**. Thanks to the auto-adjust feature, we still have a fighting chance. As long as we give the phone a valid address where it can send us audio, **we can wait until it sends us some packets and use the originating address** to determine where **to send the audio back**. This is not 100 percent guaranteed, as we know, but it's very effective in an otherwise hopeless situation. Just to be safe we only allow this magical adjustment to happen right at the beginning of the call otherwise evildoers may try to steal people's audio streams.

As previously mentioned, this RTP auto-adjust is pretty much enabled by default and will turn on automatically. The way in which you can tell that it is working is by looking for a log message like the one that follows. The message shows you the original media destination and the new one that was detected. This log line prints at the beginning of any call where auto-adjust has triggered.



```
freeswitch@lxc111>
2017-07-01 00:20:11.487678 [NOTICE] switch_channel.c:1104 New Channel sofia/inte
rnal/1001@lab.opentelecomsolutions.com [356e264a-1182-491a-b017-de20de3a6217]
2017-07-01 00:20:11.567676 [INFO] mod_dialplan_xml.c:637 Processing Sara <1001>-
>5000 in context default
2017-07-01 00:20:11.587677 [NOTICE] sofia_media.c:92 Pre-Answer sofia/internal/1
001@lab.opentelecomsolutions.com!
2017-07-01 00:20:11.587677 [NOTICE] mod_dptools.c:1312 Channel [sofia/internal/1
001@lab.opentelecomsolutions.com] has been answered
2017-07-01 00:20:11.887655 [INFO] switch_rtp.c:7189 Auto Changing audio port fro
m 192.168.1.200:7078 to 188.11.134.42:49984
freeswitch@lxc111>
```

If you do have one of the **two percent of cases** where this feature triggers the fourth pitfall and actually **breaks things** for you, **then** all you need to do is **add this parameter to your SIP profile**:

```
<param name="disable-rtp-auto-adjust" value="true"/>
```

Otherwise you can also disable it on a per-call basis by setting the channel variable rtp_auto_adjust=false at some point before the media stream has started.

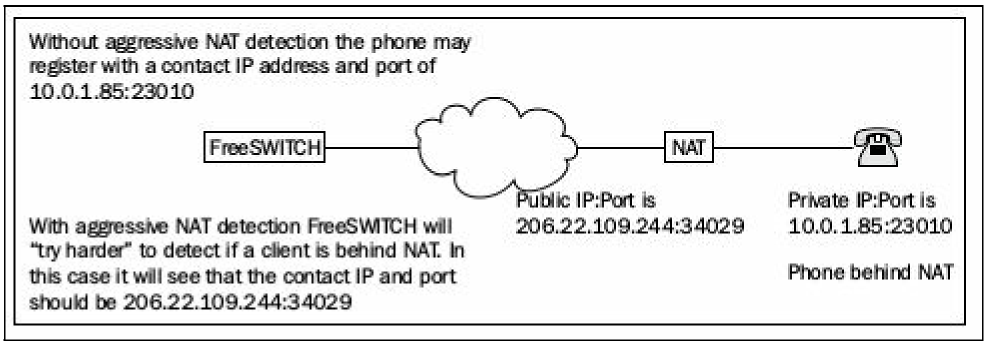
Advanced options and settings

Now that we kind of understand how it works, we can look at other ways to trigger the NAT detection. In some cases, the ACL is not enough because maybe a sneaky ALG has messed up the packet or maybe the traffic is passing over a proxy or the phone may think it's handling the NAT case itself but it's not doing it quite right.

We have another option that is not enabled by default because it laughs in the face of the fourth pitfall and basically thinks almost anything slightly out of the ordinary is NAT. **This parameter is dangerous but effective in cases where you have no other choice**. The name of the parameter is `aggressive-nat-detection` and setting it to `true` in your SIP profile will enable it for all traffic. Basically it looks at the SIP packets and if it sees a variety of IP addresses in various headers, it uses logical deduction to figure out which one is the source address. From there it does the same thing that the ACL based one does, only it may not always be the source address that it writes into the database.

```
<param name="aggressive-nat-detection" value="true"/>
```

The following diagram illustrates such a case:


Without aggressive NAT detection the phone may register with a contact IP address and port of 10.0.1.85:23010
FreeSWITCH
NAT
Public IP:Port is 206.22.109.244:34029
Private IP:Port is 10.0.1.85:23010
Phone behind NAT
With aggressive NAT detection FreeSWITCH will "try harder" to detect if a client is behind NAT. In this case it will see that the contact IP and port should be 206.22.109.244:34029

FreeSWITCH has a family of parameters we refer to as "**No Device Left Behind" or NDLB**. These parameters denote situations where we completely mimic or **accept a flaw in a device and pretend** everything is peachy or make provisions for the device so it can still work despite the fact that we are utterly offended that we actually had to make it work by modifying the code. Those are especially useful when **dealing with old clients** that are not completely conform to SIP specifications.

One such parameter that is particularly effective in the battle against NAT is called **force-rport**. The purpose of the rport attribute in SIP is a minimal attempt to conquer NAT by appending ;rport to the request. When FreeSWITCH sees this attribute, it will respond in kind with a rport=host:ip attribute so the phone will realize it's behind NAT. The funny thing is, some phones can react properly when they see the rport in the response but never request it. The force-rport parameter causes FreeSWITCH to pretend that every device we talk to has supplied an rport parameter so we respond as if they did and hence unlock the functionality that would otherwise be unobtainable.

This parameter is also not enabled by default, as is the case for most NDLB options. It also opens up vulnerability to the fourth pitfall, since **it can break many devices**. You can either set it to true to always assume rport or set it to safe to only enable it for devices where we know it's required to make things work. You can also set it to client-only or server-only to only do it depending on the direction of the call but with any luck you will never need those options.

FreeSWITCH behind NAT

We have covered some of the common cases where your phone is behind NAT talking to FreeSWITCH who isn't. Now we can move on to cases where your local copy of FreeSWITCH is behind NAT and talking to a SIP provider or another FreeSWITCH server on the public Internet. Luckily, we have also dressed up the example configuration to have the best chance to work under these NAT circumstances. We recommend you try the unaltered example configuration on a test instance of FreeSWITCH every so often just to see if there are any more new default behaviors you may be missing out on.

FreeSWITCH speaks to the router

FreeSWITCH supports two client-side NAT-busting protocols by default called **NAT-PMP** and **UPnP**. Both of these protocols use slightly different methods but the basic gist is the same.

Both methods use a network protocol to discover your NAT router and communicate with it, so rather than making the NAT mappings on-the-fly, it asks the router to open a port and actually learns the details of the port mapping, so when FreeSWITCH talks to another server it's putting the correct information in the packets for both the SIP and the media. That is cool! Beware though, now we're opening the door to the second, third, and fourth pitfalls.

We have now disguised the fact that we are behind NAT so the other side cannot detect it. An ALG might be hiding in our midst and mess with the packet thinking that it still needs to when it doesn't. The other side might be using something similar to aggressive NAT detection and spring a double anti-NAT trap. The great thing is, if you have some very strict NAT routers or firewalls, this feature will not only solve the address mapping dilemma, but also unlock otherwise blocked ports held tight by the firewall.

FreeSWITCH behind NAT, leaving the router alone

There are a set of **parameters** in the **SIP profile** called `ext-sip-ip` and `ext-rtp-ip`. These parameters are used to supply information about how to behave in regards to external IP addresses. The default configuration defines both of these parameters with `auto-nat`. This is used together with the NAT router control feature we were just talking about to just do the right thing. Some of us will not have a router that supports NAT-PMP or UPnP, or worse, it will claim to support one or the other and then not work at all because either it's inherently broken or due to some other deadly combination of the pitfalls.

We can disable this functionality when we start FreeSWITCH by supplying the `-nonat` command-line option. With this option disabled, we still have some tricks up our sleeve. You can **put an IP address in either field** when you already know what it is, for example **when you have a static external IP address**. Better yet, you can set it to `autonat:x.x.x.x` (where x.x.x.x should be replaced with your public IP) so it uses your known external IP and still does a bit of dynamic magic.

```
ext-sip-ip="auto-nat:194.20.24.1"
ext-rtp-ip="auto-nat:194.20.24.1"
```

Additional ways to set the external addresses

You can make use of dynamic DNS or STUN by setting it to host:my.domain.com or stun:stun.myhost.com (where my.domain.com or stun.myhost.com are your own dynamic domain or STUN server, respectively) to do lookups as needed. This has disadvantages because it can slow things down or stop working. As usual we have already shipped the best options by default but you still deserve to know about the other options. If you do control your own routers, you can also create permanent NAT mappings that route specific traffic right to your FreeSWITCH server or phones, and the ext-sip-ip and ext-rtp-ip can come in very handy in that case. Also, if you have the means, you could use a VPN to route two independent networks to each other when both are behind NAT, assuming you can actually control the router at both ends.

Other creative uses of FreeSWITCH in a NAT situation

FreeSWITCH can be used to conquer NAT by simply wedging it between devices. You can configure a local FreeSWITCH and register all of your phones to it, then register that instance of FreeSWITCH to the SIP provider on behalf of all of your phones, carving a hole right through the NAT and keeping everyone happy. Also, you can set up FreeSWITCH on a public IP somewhere on the Internet, then register all of your phones or local FreeSWITCH instances from multiple locations to that common server, so even if both locations are behind NAT, they can still make calls between sites without a problem.

NAT and WebRTC (SOLVED)

OK, WebRTC has been defined barely yesterday, actually is being defined right now. It's completely unencumbered by legacy shortcomings, particularly when dealing with NAT. So let's say using WebRTC as transport solves all NAT problems.

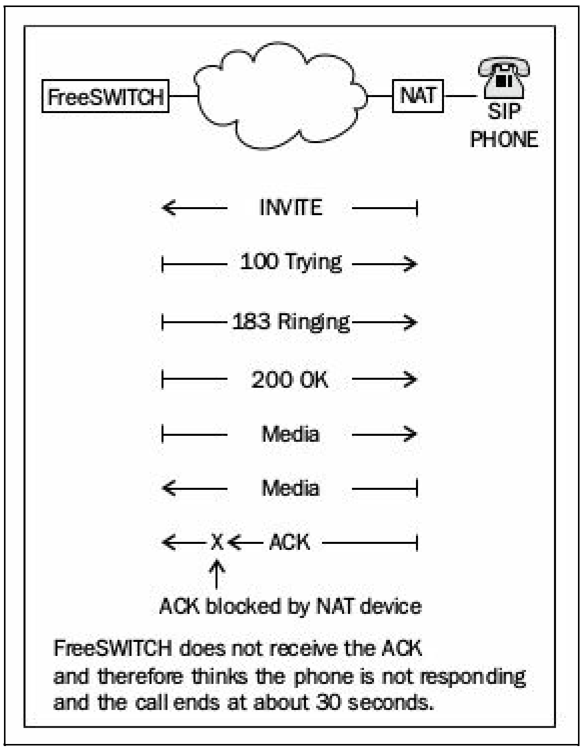
You define your **ext-rtp-ip** and **ext-sip-ip** in **SIP profiles**, then your **ext-rtp-ip** in **verto.conf.xml**, and you're all set.

The magic of ICE, used by default by all WebRTC clients to determine how to connect, and implemented on FreeSWITCH side too, is almost sure to guarantee a perfect flow of signaling and media.

If you have **a very strange situation**, for example you want to use WebRTC **inside LANs without using public addresses or DNS names,** then you will **need to implement your own ICE/STUN** server **and configure your WebRTC clients** to use that server(s). But if you have such a project, you probably already know you need your ICE infrastructure.

Conclusion

We wish you well on your journey into the world of NAT! Hopefully, this chapter will prove useful when you are stuck or prepare you in advance for a battle with an ALG .That would really be awful! If you memorize everything in this chapter and somehow still have problems, you can always come online and ask other FreeSWITCH community members for help. One final word of wisdom that can make you look smart is that if next time you notice someone complaining that their calls fail 30 seconds after they start, then yes, it's a NAT problem. This has been shown in the following diagram:


FreeSWITCH
NAT
SIP PHONE
INVITE
100 Trying
183 Ringing
200 OK
Media
Media
X
ACK
ACK blocked by NAT device
FreeSWITCH does not receive the ACK and therefore thinks the phone is not responding and the call ends at about 30 seconds.

In some cases FreeSWITCH can't fix the problem and you will need to fix or replace the NAT device.

Summary

In this chapter, we've identified the pitfalls of NAT and if even one reader is spared the searing pain caused by a first encounter with an ALG or NAT router then our journey into madness was not in vain.

We've also covered most of the options you have in FreeSWITCH to mitigate NAT-related problems. Before we move on to VoIP security, I'll leave you with a few more tips so hopefully you will get things working even when plagued by NAT. The following are the tips you should remember:

- Learn the four pitfalls of NAT and keep your eyes open for them
- It's very easy to get distracted and fall prey to one of the pitfalls. If you notice it's taking too long, start over and make sure that you haven't made a mistake somewhere that is leading you astray.
- Try to make the least changes necessary to get NAT working
- The more you mess with NAT settings, the easier it is to do something wrong or make things incompatible. It's very easy to get one end working and break the other and go back and forth for hours at the mercy of the second pitfall.
- If you have access to your own router, configure it to make conditions as favorable as possible for NAT
- The best way to make things simple is to tune your surroundings so you have the ideal environment. Choose basic NAT settings and stick with defaults that are usually tuned to working the best for the majority of cases.

In our next chapter we will change our focus from overcoming NAT issue and look at ways to improve VoIP security.

VoIP Security

VoIP Security is an increasingly important topic for protecting your FreeSWITCH system. Protection strategies include both proactive and defensive technologies. Proactive technologies in FreeSWITCH include multiple types of encryption for both SIP and RTP communication which discourages tampering or eavesdropping with phone calls. Defensive technologies in FreeSWITCH, when combined with other open source tools, can block suspicious or malicious transmissions from unknown sources and prevent abuse or fraud. The importance of combining FreeSWITCH capabilities with generally available open source VoIP tools is essential when running in a production environment.

This chapter is divided into the following four sub-sections:

- Network level protection
- Protecting signaling
- Protecting audio
- Protecting passwords

Network level protection

Most malicious individuals utilize open network ports to break into VoIP systems. They look for anything from weak passwords to known software bugs and attempt to exploit those setups to control the configuration and routing of a phone system. The general goal is to commit fraud, eavesdrop on calls, or steal information (such as voicemail messages).

Since the network is the entry point to your system, it's important to pay close attention to how your network is setup and take advantage of some of the functionality within FreeSWITCH to secure your system further.

Separating interfaces and restricting traffic

SIP is a technology that is commonly targeted for abuse on the open Internet. In most cases, malicious hackers will attempt to scan a range of IP addresses by sending UDP packets on port 5060 and look for servers that respond. Once they find a server which responds, they will attempt to brute-force common passwords or simply try to dial out. In some cases they will also simply flood the server with fake registration or other packets, crippling the system's ability to operate properly.

In average, a **SIP server** begins to be **probed by hackers and script kiddies 30 minutes** after is connected to the Internet. If you trace SIP packets, you'll see a lot of REGISTER attempts, or INVITE attempts, by user-agents like **SIPvicious**, **friendly-scanner** and **SIPcli** (and the ones that tries harder to disguise themselves as "normal" phones). Yes, they're there with a purpose.

One of the most basic ways by which you can protect your FreeSWITCH system is by separating your SIP interfaces and enforcing firewall or IPTables rules that are different on each interface.

As you've learned in previous chapters, FreeSWITCH allows you to set up different Sofia SIP interfaces so that you can send and receive SIP traffic via different IP addresses and ports on the same system. What may not be obvious is that this setup is useful for providing an extra layer of security and stability.

In terms of security, Sofia SIP profiles have default contexts for which they will route inbound calls to. Those contexts can default to fairly restrictive dialplan contexts. If you combine restrictive dialplan contexts with the relevant SIP profile, you are less likely to allow someone to send fraudulent SIP traffic through your system, even if you accidentally create a minor misconfiguration.

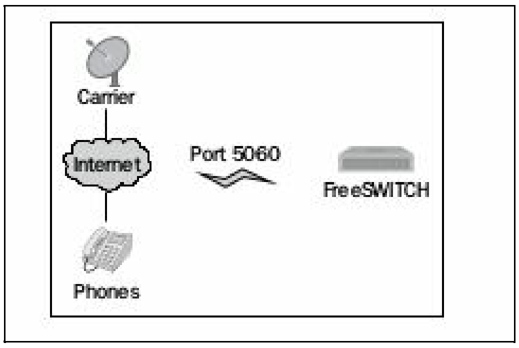
In addition, each Sofia SIP profile can have a different **Access Control List** (**ACL**). In this way, you can put more stringent restrictions on public facing SIP profiles (IP addresses) and looser restrictions on private IP addresses.

In terms of stability and performance, a little known fact about FreeSWITCH's design is that **each Sofia SIP interface is a separate thread**. It means that by having separate threads for each port and IP, you somewhat help in minimizing any disruptions someone can cause to the system. While this is by no means a foolproof way of protecting your system, any additional time you get to resolve an issue when being attacked maliciously is helpful.

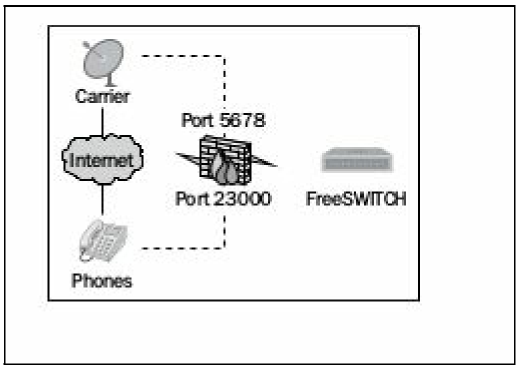
Sample setup - simple

In its most simple form, having a **SIP profile** where your carriers or **Internet Telephone Service Providers** (**ITSPs**) reach you versus a SIP profile that your own phones use is inherently beneficial. Most malicious activity starts with someone discovering that you are accepting and responding to SIP traffic on port 5060 via a port scan. At this point they will try various combinations of authentication methods until they find one that works, or otherwise abuse that port. If you **restrict access on this IP and change the port to a random number and solely allow inbound calls from carriers using ACLs**, you immediately prevent anyone from gaining access to your system, even if they have the right username and password.

The following diagram shows how people set up their FreeSWITCH system by default:



In an alternate setup, you can utilize unique and unusual ports that make it harder for hackers to find. In addition, you can use a firewall to restrict carriers to one port while keeping the other port open for phones. This has been demonstrated in the following diagram:



To achieve the previous example where carriers communicate via port 5678 and phones communicate on port 23000, you could set up a configuration like the following:

```
<profile name="incoming_from_pstn">
```

```
<settings>
<param name="auth-calls" value="true"/>
<param name="apply-inbound-acl" value="my_carriers"/>
<param name="context" value="inbound_call"/>
<param name="sip-port" value="5678"/>
... other settings here ...
</settings>
</profile>
```

In the previous Sofia profile, inbound calls from your carriers must come in on port 5678. When they hit this port, the my_carriers ACL will be applied, making sure only carriers get through. If you make an error in your my_carriers ACL, no big deal - the context the caller reaches is inbound_call and only allows for inbound calls, not outbound. These are a fairly solid set of restrictions.

In addition, since you know only your carriers should be contacting you on port 5678, you could modify your firewall or IPTables software firewall rules to allow traffic on this port only from your carrier's IP addresses. This is a fairly fool proof methodology for inbound access.

You would then create the following profile for your users to utilize:

```
<profile name="customer_access">
<settings>
<param name="auth-calls" value="true"/>
<param name="apply-inbound-acl" value="default_deny_list"/>
<param name="context" value="customer_call"/>
<param name="sip-port" value="23000"/>
... other settings here ...
</settings>
</profile>
```

In this Sofia SIP profile, calls that your customers make would need to hit SIP port 23000. This port requires authentication and uses an ACL called default_deny_list which denies all traffic by default. This will force traffic to be authenticated, meaning the user must provide a valid username and password to be able to utilize the system. Once the user provides that information they will be sent to the customer_call context where the calls will be processed.

Sample setup - complex

A more complex example indicates isolation of networks and assumes you have network routing gear that allows you to split network access either physically (separate network cables and routers) or via subnets or VLANs (see next section).

In the more complex setup, you would additionally use different IP addresses which, via your operating system, are bound to different network cards. In addition to mapping the SIP ports to different network interfaces, you would also map the event socket and other FreeSWITCH ports to a management network interface.

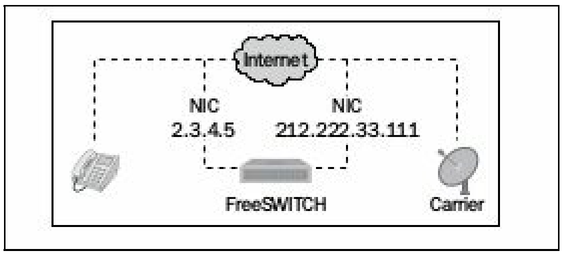
To do this, you might set up two Sofia SIP profiles like the following:

```
<profile name="incoming_from_pstn">
<settings>
<param name="auth-calls" value="true"/>
<param name="apply-inbound-acl" value="my_carriers"/>
<param name="context" value="inbound_call"/>
<param name="sip-port" value="5678"/>
<param name="sip-ip" value="2.3.4.5"/>
... other settings here ...
</settings>
</profile>
<profile name="customer_access">
<settings>
<param name="auth-calls" value="true"/>
<param name="apply-inbound-acl" value="default_deny_list"/>
<param name="context" value="customer_call"/>
<param name="sip-port" value="23000"/>
<param name="sip-ip" value="212.222.33.111"/>
... other settings here ...
</settings>
</profile>
```

In the previous scenario, note carefully the sip-ip setting which differs for each interface. One interface is 2.3.4.5 and the other is 212.222.33.111. FreeSWITCH will attempt to utilize the network interface on the physical server which represents the IP address that you have specified in the Sofia SIP profile. This allows you to use different firewalls and network links for each interface powering your FreeSWITCH system.

In this scenario, 2.3.4.5 could be an internal IP address that is not routable on the public Internet and 212.222.33.111 could be a public IP address that is routable. The IP address 212.222.33.111 would only have to be open to carriers unless you have users with phones outside your local network. As an alternative for allowing phones, **you could allow your staff to VPN into your network. This would be the most secure strategy possible.**

The previous scenario can also be illustrated with the following diagram:



In the previous diagram, the phone talks on the Network Interface Card (NIC) on the FreeSWITCH box assigned to 2.3.4.5, while the carrier talks to FreeSWITCH via the NIC on 212.222.33.111.

VLANs

VLANs are a fantastic way to isolate phones from data communications on your local network and if done correctly, they **can improve call quality while preventing malicious activity**.

You don't want **data traffic** on your local LAN (someone copying a movie, or taking a backup) to **hog the bandwidth** shared with phones and impair the quality of voice/video calls.

VLANs are often overlooked as optional but in fact, a lot of damage can be done by having phones on the same network as computers. It is simple to identify the IP address of a device and log in to it when they are on the same network. From there, it is trivial to lift the SIP username and passwords of many phones. For example, if you log in to an old Polycom phone, you can export the configuration which contains the phone's credentials and view them in plain text. It is more challenging to do this when a phone is on a private segment of the network.

In addition to manipulating a phone directly, VLANs prevent tools running on computers from sending bogus information into the voice network. This includes scenarios as simple as unauthorized softphones that are used to hijack an extension to send bogus BYE messages causing calls to hang up intentionally when they should have continued.

It's worth noting that VLANs are available on most networks as either port-based tagging, where a specific physical port on a switch is part of a virtual LAN or software-based tagging, where the network interface card or operating system tags each packet with a specific virtual LAN number in the IP header.

Setting up VLANs is beyond the scope of this book. However it should be noted that nearly all popular desk phone models and recent network gear support VLANs, including both software and port-based VLAN tagging.

Intrusion detection

Detecting intruders attempting to gain access to the system or who are intentionally creating a denial-of-service or similar type of disturbance can be a challenge. While it may seem obvious what type of traffic would be considered unusual, there are edge cases that must be considered when setting up rules for automatic detection and blocking of hacking attempts.

Registration and call attempts monitoring

Some tools overwhelm VoIP systems by sending fake authorization attempts to them without ever responding to the challenge request that is used in SIP. One popular tool is often referred to as friendly scanner or SIPvicious. These types of tools keep a system busy handling bogus requests, overloading the system, making it difficult to handle real requests, and so on. Another suspicious behavior can be detected from someone trying to make long distance or international calls repeatedly within a short time period.

FreeSWITCH provides the ability to log a warning when an attempt is made to utilize credentials in the system (recognized or not). Programs such as Fail2Ban may then be used to monitor the frequency in which this logline is produced. If the frequency hits a threshold where the traffic is suspicious, the IP address causing the traffic can be blocked for a period of time (or permanently). It is generally considered suspicious if a large number of authorization attempts occurs from the same IP address within a relatively short period of time.

To ensure that a warning is generated when FreeSWITCH receives an invalid authentication attempt, you can modify your SIP profiles and include the following setting:

```
<param name="log-auth-failures" value="true"/>
```

A log line will be generated for authentication attempts that looks as follows:

[WARNING] SIP auth challenge (REGISTER) on sofia profile 'customer_access' for [user_rdkj7h@2600hz.com] from ip 184.106.157.100

These warnings will be counted automatically by Fail2Ban and when they hit a configured threshold the IP address 184.106.157.100 will be blocked.

Fail2Ban

Fail2Ban is a third-party program that runs in the background and monitors logs. When specific loglines (such as the authentication challenge line shown previously) are seen a certain number of times, Fail2Ban takes an action. It can be programmed to e-mail you with an alert or automatically use IPTables to block an offending IP address after too many attempts occur within a certain period of time.

This book is not intended to be a complete guide for using Fail2Ban. However some sample scripts are given later in the chapter.

To configure Fail2Ban you will need to create several files which instruct Fail2Ban what to look for in your logs and what to do when it finds a match.

Fail2Ban default configuration has a folder where you can place filters. These filters contain strings which can be used to match against your logs. You can have as many filters as you want to look for different types of suspicious traffic in your logs. When combined with FreeSWITCH's error log which shows invalid login attempts, this can become a useful filter mechanism.

The second file, known as the jail configuration file applies the filters to rules such as how often an error is allowed to occur and what action to take after that threshold has been exceeded. The jail configuration file effectively specifies how to react when a filter matches.

Filter configurations

Let's first review a filter configuration file. This file would typically be placed in /etc/Fail2Ban/filter.d/ and named according to the particular filter you are looking for. The file would contain a filter to look for failed authentication attempts. The format is a standard regular expression. In this case we consider a failure in FreeSWITCH anytime someone tries to register or make a call using invalid credentials.

The fine people of Fail2Ban community maintains a configuration tailor made for this usage with FreeSWITCH. Is available in latest version at their repository (https://github.com/fail2ban/fail2ban/blob/master/config/filter.d/freeswitch.conf), save it as file /etc/Fail2Ban/filter.d/freeswitch.conf. Relevant lines at the moment of this writing are:

```
# Enable "log-auth-failures" on each Sofia profile to monitor
# <param name="log-auth-failures" value="true"/>
# -- this requires a high enough loglevel on your logs to save these messages.
#
# In the fail2ban jail.local file for this filter set ignoreip to the internal
# IP addresses on your LAN.
#

[Definition]

failregex = ^\.\d+ \[WARNING\] sofia_reg\.c:\d+ SIP auth (failure|challenge) \((REGISTER|
^\.\d+ \[WARNING\] sofia_reg\.c:\d+ Can't find user \[\d+@\d+\.\d+\.\d+\.\d+\] from <HOS

ignoreregex =

# Author: Rupa SChomaker, soapee01, Daniel Black
# https://freeswitch.org/confluence/display/FREESWITCH/Fail2Ban
# Thanks to Jim on mailing list of samples and guidance
```

This will watch the FreeSWITCH logs for failed REGISTER or INVITE messages.

Jail configurations

Now, combine this filter with a jail entry which blocks an IP address if too many failed INVITEs or REGISTERs are received within a certain period of time.

The /etc/jail.conf file may get overwritten when upgrading Fail2Ban. Create a /etc/fail2ban/jail.local file with the following data in it, setting the correct path to *your* freeswitch.log file (maybe yours is in /usr/local/freeswitch/log/freeswitch.log), and adjust the sender email address to your setup:

```
[freeswitch]
enabled  = true
port     = 5060,5061,5080,5081
filter   = freeswitch
logpath  = /var/log/freeswitch/freeswitch.log
action   = iptables-allports[name=freeswitch, protocol=all]
sendmail-whois[name=FreeSwitch, dest=root, sender=fail2ban@example.org]
maxretry = 10
findtime = 60
bantime  = 600
# "ignoreip" can be an IP address, a CIDR mask or a DNS host
ignoreip = 127.0.0.1/8 192.168.2.0/24 192.168.1.0/24
```

The earlier settings indicate the use of freeswitch filter and after 10 failed INVITE or REGISTER authorization attempts (maxretry) within a 60 second period, blocks the IP address of the offender and send an alert mail. If the filter is met (meaning 10 failed INVITE or REGISTER authorization attempts occur) within a 60 seconds period, the offending IP address will be banned in full for 600 seconds (ten minutes) and an alert mail will be sent to the configured administrator address.

Do not lock the good guys out!

The Fail2Ban configuration must be tuned so that a large site is not accidentally kicked offline just because they are coming back online, causing all their phones to reregister at once. For example, if you have a rate limit Fail2Ban entry (eg a rule about quantity of "good" traffic, as opposed to "failed attempts" numbers), you would not want to set up Fail2Ban to block IP addresses if they happen to send 50 authentication requests in a 5 second period, because if the site has 50 phones and their power goes out, when their power comes back on all phones will attempt to register at once, resulting in them being banned. This is not the intent. **Be careful in what you judge a "Denial of Service" traffic pattern.**

Care must be taken when setting up Fail2Ban to test for edge case scenarios like the power outage scenario just described.

Encryption in non-WebRTC SIP: Signaling and Media

Specific techniques to apply encryption has been developed especially for "classic", or "normal", SIP (as opposed to SIP over WebRTC, see later).

SIP encryption is based on two concepts - encrypting signaling and encrypting media (audio/video) communication. Like any standard encryption mechanism, SIP encryption utilizes standard cryptography libraries and involves key exchanges and password negotiation to securely transmit and receive information. The main encryption algorithms used in SIP (which are detailed later) are very similar to SSL over the web and key exchange is used when connecting to remote servers via SSH. In either exchange, the main goal is to end up with an encryption algorithm and a common encryption secret between the two parties that only they know, which can be used to encrypt and decrypt the actual content - the phone call.

Many people toss around the terms TLS, SSL, and SRTP without fully understanding them. It should be understood that in order to fully protect communications it is recommended to choose both an encryption strategy for signaling and an encryption strategy for audio encryption.

In the following sections, we'll review each cryptography strategy in more detail.

Choosing between encryption options

There are some encryption options available for FreeSWITCH. You can encrypt the signaling (that is, the SIP messages), the media (that is, the audio in the RTP stream), or both.

TLS (Transport Layer Security) encrypts everything over the TCP connection; this has the downside that jitter or delays due to TCP can occur. UDP is generally preferred for RTP and using TLSV1 has some additional traffic overhead.

ZRTP has the advantage that it allows for end-to-end encryption without pre-exchanging keys or certificates but is a bit more complex to set up, especially for some clients. ZRTP is a protocol that was co-designed by Phil Zimmermann, the same individual who created PGP encryption. More information can be found at http://zfone.com.

SSL (Secure Sockets Layer) v2/3; SSLv2 was found to be **insecure** and is deprecated since 2011, while SSLv3 has been **broken** in 2014 and deprecated in 2015 (google for POODLE attack). So, because there is no meaning into having a breakable encryption, **disable SSL altogether**.

Disabling SSLv3 and SSLv2

It's long time we already ship with correct configuration, but you may want to check your settings.

You do not want to have sslv2 or sslv3 into **tls-version** parameter value.

Edit /usr/local/freeswitch/conf/vars.xml (or /etc/freeswitch/vars.xml if you installed from packages), and be sure this line reads as:

```
<X-PRE-PROCESS cmd="set" data="sip_tls_version=tlsv1,tlsv1.1,tlsv1.2"/>
```

Then check into all SIP profiles if they are using this same value. It can be taken from global variables as:

```
<param name="tls-version" value="$${sip_tls_version}"/>
```

Certificates

For any kind of SIP encryption, you need a **Security Certificate** issued ("signed") by a **recognized Certification Authority**, that takes on herself the responsibility to assert that you are you and not a cat.

You can cheat at this by crafting **"self-signed" certificates**, using software that simulate a certification authority (eg: you are your own Certification Authority). **Don't. Read the section "Certificates" in chapter "WebRTC, SIP and Verto"**.

Certificates have nothing to do with using SSL as encryption method, **"SSL Certificate"** is just the old way to call a security certificate (because was then used by SSL, but it can be used by TLS too, no problem, is always the same certificate). Also **if you disable SSL, you still need the same certificates** for TLS, DTLS, and all other kind of encryptions.

"Real" certificates, from "real" CAs

You want to use real, actual certificates issued by a real, actual Certification Authority (as opposed to "self-signed" certificates). Those "real certificates" are automatically accepted by SIP clients, WebRTC clients, browsers and Internet of Things thingies.

At https://letsencrypt.org you can obtain legit certificates for free, or you can explore the market about the for-pay solution that best fits your needs (eg: wildcard certificates).

For all details about **certificates**, **how to obtain and install** them for both SIP, Verto, and WebRTC, **read the section "Certificates"** in Chapter 5, *WebRTC, SIP, and Verto*.

Protecting signaling

SIP signaling contains both authentication information your phone utilizes to make and receive calls and includes the Caller ID Name and Number of the caller and callee, by default in plain text. This is easy to sniff and to spoof. Encryption makes that harder. In addition, if you are using SRTP (Secure RTP), the SIP signaling contains the cryptography key used to keep your audio secure. Someone who observed this key in plain-text would easily be able to defeat the media encryption utilized.

Encryption with TLS

Once you've obtained your certificates, you will need to tell FreeSWITCH to use these certificates. To do this, set these variables in /usr/local/freeswitch/conf/vars.xml:

```
<X-PRE-PROCESS cmd="set" data="sip_tls_version=tlsv1,tlsv1.1,tlsv1.2"/>
<X-PRE-PROCESS cmd="set" data="internal_tls_port=3361"/>
<X-PRE-PROCESS cmd="set" data="internal_ssl_enable=true"/>
<X-PRE-PROCESS cmd="set" data="external_tls_port=3381"/>
<X-PRE-PROCESS cmd="set" data="external_ssl_enable=false"/>
<X-PRE-PROCESS cmd="set" data="sip_tls_ciphers=ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH"/>
```

Note that in **vars.xml** TLS (historically named "**ssl**") is **enabled separately for internal and external profile**.

The last line sets the ciphers algorithms to be used. If you want to be able **to decrypt** the sniffed SIP TLS packets (using the private key), then you want to enable **only NON** perfect forward secrecy (**PFS**) algorithms. So, use AES256-SHA:

```
<X-PRE-PROCESS cmd="set" data="sip_tls_ciphers=AES256-SHA"/>
```

After setting defaults in vars.xml, **check your SIP profiles and be sure the parameters are reflecting what you want** (remember: what is written between "<!--" and "-->" is commented out):

```
<!-- TLS: disabled by default, set to "true" to enable -->
<param name="tls" value="$${internal_ssl_enable}"/>
<!-- Set to true to not bind on the normal sip-port but only on the TLS port -->
<param name="tls-only" value="false"/>
<!-- additional bind parameters for TLS -->
<param name="tls-bind-params" value="transport=tls"/>
<!-- Port to listen on for TLS requests. (5061 will be used if unspecified) -->
<param name="tls-sip-port" value="$${internal_tls_port}"/>
<!-- Location of the agent.pem and cafile.pem ssl certificates (needed for TLS server) -->
<!--<param name="tls-cert-dir" value=""/>-->
<!-- Optionally set the passphrase password used by openSSL to encrypt/decrypt TLS private ke
<param name="tls-passphrase" value=""/>
<!-- Verify the date on TLS certificates -->
<param name="tls-verify-date" value="true"/>
<!-- TLS verify policy, when registering/inviting gateways with other servers (outbound) or h
<!-- set to 'in' to only verify incoming connections, 'out' to only verify outgoing connectic
<param name="tls-verify-policy" value="none"/>
<!-- Certificate max verify depth to use for validating peer TLS certificates when the verify
<param name="tls-verify-depth" value="2"/>
<!-- If the tls-verify-policy is set to subjects_all or subjects_in this sets which subjects
<param name="tls-verify-in-subjects" value=""/>
<!-- TLS version default: tlsv1,tlsv1.1,tlsv1.2 -->
<param name="tls-version" value="$${sip_tls_version}"/>
<!-- TLS ciphers default: ALL:!ADH:!LOW:!EXP:!MD5:@STRENGTH  -->
<param name="tls-ciphers" value="$${sip_tls_ciphers}"/>
```

There are a number of gotchas and snafus possible with TLS. **TLSruns on TCP**, rather than UDP. This has the downside that when FreeSWITCH needs to make a connection toward the phone (such as delivering an inbound call to the phone) and if the phone is sitting behind a firewall or NAT traversal mechanism, the phone may be unreachable. You must make sure that all **firewalls are configured to work with TCPinbound traffic**. Also, ensure that the **time is configured properly on your phones** as you will get cryptic `bad certificate` error messages if the time is too far off, and it will fail to handshake properly.

Protecting media

Encryption of the RTP streams (audio and video) ensures that the actual content of SIP calls cannot be listened on, recorded, or otherwise illegally obtained. There are multiple ways to achieve this security.

At its core, encryption requires that both sides involved agree on an encryption method and an algorithm for encrypting and decrypting the data being transmitted and received. In other words, you can't use an encryption method that isn't supported by both sides. In addition, encryption algorithms are based on key exchanges, generally at the beginning of the call. These key exchanges are similar to exchanging passwords by both parties, but in an electronic and often automated way.

There are two popular forms of encryption generally used when encrypting audio and media streams. These forms of encryption are SRTP and ZRTP.

SRTP was developed in 2004 by a small team of IP protocol and cryptographic experts from Cisco and Ericsson. SRTP defines a method of transmitting and receiving RTP with message authentication and integrity and replay protection to the RTP data. Because it is older and was developed by key IP telephony hardware players, it has seen adoption in most SIP equipment. **SRTP is available on almost all devices** available on the market.

ZRTP was developed in 2006 by Phil Zimmermann (creator of PGP). It is a newer entrant that makes key negotiation automatic, significantly simplifying the setup and operation of ensuring secure and encrypted RTP calls. It also has the added advantage of not being dependent on server-side encryption. Encryption can occur between servers that are otherwise unaware of the contents of the RTP stream. However, **a limited number of hardware support ZRTP** at this moment, most hardware manufacturers will need to implement ZRTP for this protocol to be fully successful.

Both SRTP and ZRTP technologies are supported by FreeSWITCH and are described in this chapter.

Encryption with SRTP

SRTP is an encryption mechanism that is negotiated during call setup via SIP. Both sides of the SIP conversation must agree to support RTP encryption and exchange keys for encryption in the SIP packets. The encryption key used for SRTP is exchanged inside SIP packets. This information is then used to encrypt the audio stream.

SRTP enables encryption of the RTP data with minor overhead to the RTP UDP packets. This has the benefit that the call data are encrypted but still transmit via UDP, minimizing latency and using network traversal mechanisms that would normally be used in an unencrypted stream.

Generally SRTP is the most firewall friendly strategies for existing installations since the actual work has already been done to get RTP to transmit properly over the network. SRTP is fairly easy to configure within FreeSWITCH.

Note that **unless you enable encryption of the SIP packets as well** (discussed later) **the key for the SRTP goes in the clear**. For a fully secure connection between your phone and FreeSWITCH **you should combine SIP encryption with SRTP encryption**. This prevents any snooping or man-in-the-middle attacks. If only SRTP is enabled, only payload packets of type RTP packets will be secured.

Enabling SRTP

You can enable SRTP from your Dialplan on a per-call basis by setting the following flag:

```
<action application="set" data="sip_secure_media=true"/>
```

This needs to be done on both legs and on both inbound and outbound calls to be fully effective. Of course, the ITSP that provide you gateway to PSTN will probably not support SRTP so you may only be able to enable this on legs from FreeSWITCH to the endpoint(s).

You can check if media is secured properly by checking for the variable ${sip_secure_media_confirmed} to be set. As an example, the following block will play a bong tone when SIP media is secured:

```
<extension name="is_secure">
<condition field="${sip_secure_media_confirmed}"
expression="^true$">
<action application="sleep" data="1000"/>
<action application="gentones" data="${bong-ring}"/>
</condition>
</extension>
```

Debugging SRTP

When **debugging** encryption, a helpful hint is in the **SIP packet** as to whether the phone is properly requesting encryption. You will see SIP packets that include the `a=crypto` line if you have offered encrypted RTP in your SIP setup (you can **see SIP TLS packets in plaintext** on FreeSWITCH console, by typing: "**sofia global siptrace on**").

Encryption with ZRTP

ZRTP is an **SRTP**-based encryption algorithm that differs from SRTP by **exchanging encryption keys within the media stream**, making the encryption more secure and also transparent to servers that don't understand the protocol. This allows ZRTP to be more flexible than SRTP and gives complete control to the endpoints to handle all levels and requirements of encryption **without the risk of a man-in-the-middle attack**. ZRTP also does not require a key exchange prior to media setup. The **key exchange occurs during the initial portion of the RTP** conversation. The ZRTP protocol is fully laid out in RFC 6189.

One of the major strong points of **ZRTP** is its ability to **work via proxies**. Typically with SRTP, every point communicating via the encrypted stream needs to be aware of the encryption protocol, and also able to encrypt and decrypt the audio stream. This allows for snooping the encryption mid-stream if you have access to a server where the stream traverses. With ZRTP, the opposite is true; **the servers in the middle do not need to be aware of the encryption** protocol at all. They believe they are simply passing **standard RTP packets**. Since the servers are unaware of what the content of the RTP stream contains, **only the two endpoints need to support ZRTP** in order for the conversation to be completely secure. The proxies don't need to understand, or pass encryption information.

Another major advantage of **ZRTP in FreeSWITCH is that it's enabled by default**. The ZRTP protocol itself inserts negotiation packets into every initial RTP conversation and waits for a reply. If a reply is received, ZRTP encryption is automatically enabled when the other endpoint requests it.

Because ZRTP is a less popular protocol, there has been work to not only build ZRTP-enabled phones and soft clients but also work on a **ZRTP software proxy**. This proxy allows you to simply install a ZRTP plug-in software program, known as **Zfone**, and RTP traffic will be automatically encrypted, even from/to a software that doesn't natively support ZRTP. Zfone runs in the background unobtrusively and pops up to notify you when a key exchange has occurred. There is also an SDK available to help developers build ZRTP-enabled software and hardware as well.

Enabling ZRTP

First of all, edit the file /usr/local/freeswitch/conf/autoload_config/switch.conf.xml and set rtp-enable-zrtp to "true":

```
<param name="rtp-enable-zrtp" value="true"/>
```

Then you can enable or disable ZRTP support from within your Dialplan using the following command:

```
<action application="set" data="zrtp_secure_media=[true|false]"/>
```

When ZRTP is being negotiated, you will see the following line on the FreeSWITCH console indicating ZRTP is being offered:

[DEBUG] switch_rtp.c:928 [zrtp main]: START SESSION INITIALIZATION.

The ZRTP protocol will then begin injecting ZRTP negotiation packets into the RTP stream. If ZRTP is successfully started for a session, you will see a series of ZRTP log messages followed by a confirmation message that the channel is now secure, such as the following:

[zrtp protoco]: Enter state SECURE (DH).

You will also see that a cache of the selected shared secret was auto-stored, which will be used for comparison purposes on the next call:

[zrtp cache] Storing ZRTP cache to </usr/local/freeswitch/db/zrtp.dat>...

Testing media SIP encryption

In demo configuration, when you call 9664 (or press "3" after calling the demo IVR at 5000) both SRTP and ZRTP are tried. If you call from a recent softphone with SRTP or ZRTP enabled (eg, Linphone) you will hear "This call is secure" before the music.

Protecting WebRTC SIP and VERTO Signaling and Media

WebRTC is encrypted by default, using TLS for wss signaling and DTLS (TLS on UDP) for SRTP encryption. See the section Encryption Everywhere of WebRTC Chapter.

Encryption of media and data streams is mandatory. Full stop. There is no way you can have a WebRTC communication without encrypting the streams that are exchanged. Those would be the media streams (audio+video) and, if present, the data streams. Media streams are encrypted as **SRTP** with key exchange via **DTLS**.

Session protocol's signaling (SIP or VERTO) is encrypted too. The transport of choice for such signaling is usually **Secure WebSocket**. You will see almost everywhere the URI prefix **WSS://** (WebSocket Secure) when defining WebRTC signaling servers and endpoints. **WSS** signaling exchange is encrypted by **TLS**.

Protecting passwords

Passwords are used in FreeSWITCH when **phones register**, when **phones originate a call**, when **FreeSWITCH registers to external gateways/ITSPs** and when administrators authenticate into the FreeSWITCH system via **Event Socket** (eg: **fs_cli**). Most of these areas utilize **weak plaintext passwords**.

In addition, many **users set their passwords to simple easy-to-guess** combinations. **Worse** yet, some **don't ever change** or set up their voicemail password, leaving the **defaults** in place.

These **passwords** are very often **targeted** and once gained, they are exploited **to commit fraud**.

Following are some of the mechanisms available to mitigate this.

Registration passwords as hashes

Registration credentials do not need to be passed or kept on disk in plain-text. When defining SIP credentials in your User Directoryy, instead of including the following line:

```
<param name="password" value="samiam"/>
```

replace it with a pre-calculated a1-hash of the password, like the following:

```
<param name="a1-hash" value="c6440e5de50b403206989679159de89a"/>
```

To generate a1-hash, on a linux box get the md5 of the string username:domain:password, which is your username, domain name, and password all tied together with a colon. As an example:

```
echo -n "darren:2600hz.com:pass1234" | md5sum
b62d1e3e27773ffd173c87e342a6aace
```

You would utilize the returned hash in your User Directory entry. This means you did not have to store the actual SIP registration on disk and someone who finds a way to compromise the file can't see the password either.

A full example would look something like the following:

```
<user id="darren">
<params>
<param name="a1-hash"
       value="c6440e5de50b403206989679159de89a"/>
</params>
</user>
```

Voicemail passwords

Voicemail boxes have a history of being **compromised** for a variety of reasons. Besides simply listening to someone else's messages, voice mailboxes are often exploited because they have **call-back or forward** features which can be turned on remotely. One of the most popular strategies is to hack a voicemail box and forward that person's calls **to an expensive international destination**, racking up thousands of dollars of calls in a short amount of time. This makes voicemail password hacking popular even today.

Protection against weak voicemail passwords is fairly simple. FreeSWITCH stores voicemail passwords in plain-text in the database, allowing you to scan for passwords which are weak, such as 1111 or 1234. You can also scan for people who are using their extension number as their voicemail password which is another popular (and insecure) password strategy.

To scan for weak passwords you'll need to write a script that looks for passwords in the voicemail configuration database. Assuming you are using the defaults in FreeSWITCH, the voicemail database is stored in a SQLite file in your FreeSWITCH DB folder. This folder will be in one of various locations depending on how you installed FreeSWITCH, but most commonly it is in /opt/freeswitch/db, /usr/local/freeswitch/db, or /var/lib/freeswitch/db.

A sample way to check your database could be using the following simple SQLite query:

```
sqlite3 db/voicemail_default.db "select * from voicemail_prefs
where password=1234 or password=1111"
```

This command would use the SQLite3 linux client to look in the voicemail_prefs table for any passwords that are 1111 or 1234. It will print all information about that mailbox on the screen, including the username and domain name of the user who has this password. You can then take corrective action by either resetting the password forcefully or contacting the user to advise them to change their password.

Summary

This chapter is only a brief guide to the most common VoIP security technologies prevalent today. There are a plethora of additional resources including sites such as www.hackingvoip.com, books on Hacking VoIP, and on Computer Security.

Taking the basic steps outlined in this chapter will provide you sufficient amount of security against today's most common hacks, DoS attacks, and abuses. This should allow most small to medium sized PBXes or hosted VoIP systems to operate securely and reliably.

We now move on to the last chapter in this book where we see how to react when things go wrong, we're not able to implement something, or we hit a bug in FreeSWITCH. We'll learn the basics about troubleshooting, asking for help, and reporting problems.

Troubleshooting, Asking for Help, and Reporting Bugs

SIP, WebRTC, PSTN, Dialplan, IVRs, Lua, ESL, HTTAPI, XML_CURL, NAT, Security, ITSPs, UDP, RTP, TLS, WSS, Certificates, phones, softphones, smartphones, apps, are you born learned?

In this chapter we'll saw how to investigate problems, troubleshoot your FreeSWITCH implementation and operation, how to obtain and understand debug logs and packet traces, how to identity and pinpoint the interoperability problems with your upstream and downstream providers and users.

We'll then see how to search the documentation before asking for help, both free help from community (mailing list, hipchat, IRC) and commercial support and consulting from the pros. When you find a bug, or you want to know if a problem has been encountered before and classified as bug, the tool you want to use is Jira. We'll give you full instruction on how to research and report bugs.

Then we'll close this chapter, and the book, with a description of ClueCon the yearly Real Time Communication and IoT Conference we organize in Chicago in August. ClueCon motto is "from developers to developers", and its true: you'll find us there, and all the developers' community of Open Source Real Time Communication and Internet of Things.

Troubleshooting FreeSWITCH

So many things can go wrong in Real Time Communication, and most of them are not your responsibility, nor are in your reach. Can be a faulty switch in your Internet Carrier, a misconfiguration in ITSP, a phone faulty firmware, a customer firewall, your datacenter new topology.

And, obviously, can be an error in your implementation of any of the techniques we saw in previous chapters. Or, maybe you're right, and is a bug in a FreeSWITCH module.

Following sections are about how to troubleshoot your FreeSWITCH implementation, so to understand what the problem is, and where it comes from.

For more details and techniques please read the chapter "Tracing and Debugging VoIP" in "Mastering FreeSWITCH" book, Packt Publishing, 2016.

firewall rules and network issues

Seems trivial, but the number one source of problems is a misconfigured firewall. Misconfigured for the purpose of allowing FreeSWITCH network traffic to flow, that's it. But maybe the person in charge of maintaining the firewall rules is unaware of what kind of traffic (addresses, protocols, ports) FreeSWITCH will generate and need.

When investigating a problem, a one way communication, calls dropped after 30 seconds or a minute, all those kind of failures, first be sure the traffic is not blocked.

Open all ports and all protocols on your FreeSWITCH server machine (eg, disable the server firewall completely). Edit the file /usr/local/bin/disable_iptable.sh to be:

```
#!/bin/sh
echo "Flushing iptables rules..."
sleep 1
iptables -F
iptables -X
iptables -t nat -F
iptables -t nat -X
iptables -t mangle -F
iptables -t mangle -X
iptables -P INPUT ACCEPT
iptables -P FORWARD ACCEPT
iptables -P OUTPUT ACCEPT
```

Then make it executable, and execute it (**DO NOT EXECUTE IT LINE BY LINE!** You'll risk to be locked out from your server):

```
chmod +x /usr/local/bin/disable_iptable.sh
/usr/local/bin/disable_iptable.sh
```

Now restart FreeSWITCH, and see if the problem is still there. If no joy, check also the following list:

- If behind NAT
    - what is the public address of FreeSWITCH? Is correctly reported into all ext-rtp-ip and ext-sip-ip parameters? (go to /usr/local/freeswitch/conf and execute "grep -r ext-")
    - are all the port and protocols "open" and forwarded from the Internet to FreeSWITCH?
- If an external firewall, worst of all an appliance
    - is it blocking your traffic?
    - worst, is it applying some sort of ALG (Application Level Gateway) to SIP or other protocols? You DO NOT WANT ANY SMARTNESS in your router or firewall. They're not smart enough to be smart, and they end up

doing subtle and intermittent damages. Believe. Disable all kind of smartnesses in routers and firewalls

checking FreeSWITCH configuration and operation

If you're confident (and double checked) that is not a network problem (and you verified the problem is still there when you disable all firewalls and open and forward all protocols and ports, then you want to check your FreeSWITCH configuration.

Stop FreeSWITCH (from console "fsctl shutdown"), then restart it (or watching it be restarted by systemd). On a Debian server, you can now check four most precious files (you will easily find the equivalent files on other OSes):

```
/usr/local/freeswitch/log/freeswitch.log
/var/log/syslog
/var/log/daemon.log
/usr/local/freeswitch/log/freeswitch.xml.fsxml
```

- **freeswitch.log** this is your primary source, it contains all of FreeSWITCH self-awareness. Also, you'll find here how FreeSWITCH startup has gone. Particularly look for ERR and WARNING lines. For example, maybe you misconfigured one of your sofia SIP profiles, and that profile refused to start. Or, it tried to start but found "its own" network port already taken by another (zombie?) program.

  You check this file to see how your extensions have been interpreted by FreeSWITCH: maybe a regular expression you put in a condition was not matching (maybe because what was incoming was not in the format you expected, or because the regex is wrong).

  A lot of snafus like: "I set and check this variable, but I forgot to "inline" the set action" can be caught reading freeswitch.log

- **syslog/daemon.log** contain messages generated by the Operating System, and may report the reason FreeSWITCH has not started
- **freeswitch.xml.fsxml** is a "brain dump" of FreeSWITCH itself. It contains the entire XML tree FreeSWITCH has built in its memory at startup time from the XML files in conf directory, default settings, calculated settings (eg, local IP address), and expanded macros and variables. Can be most revealing.

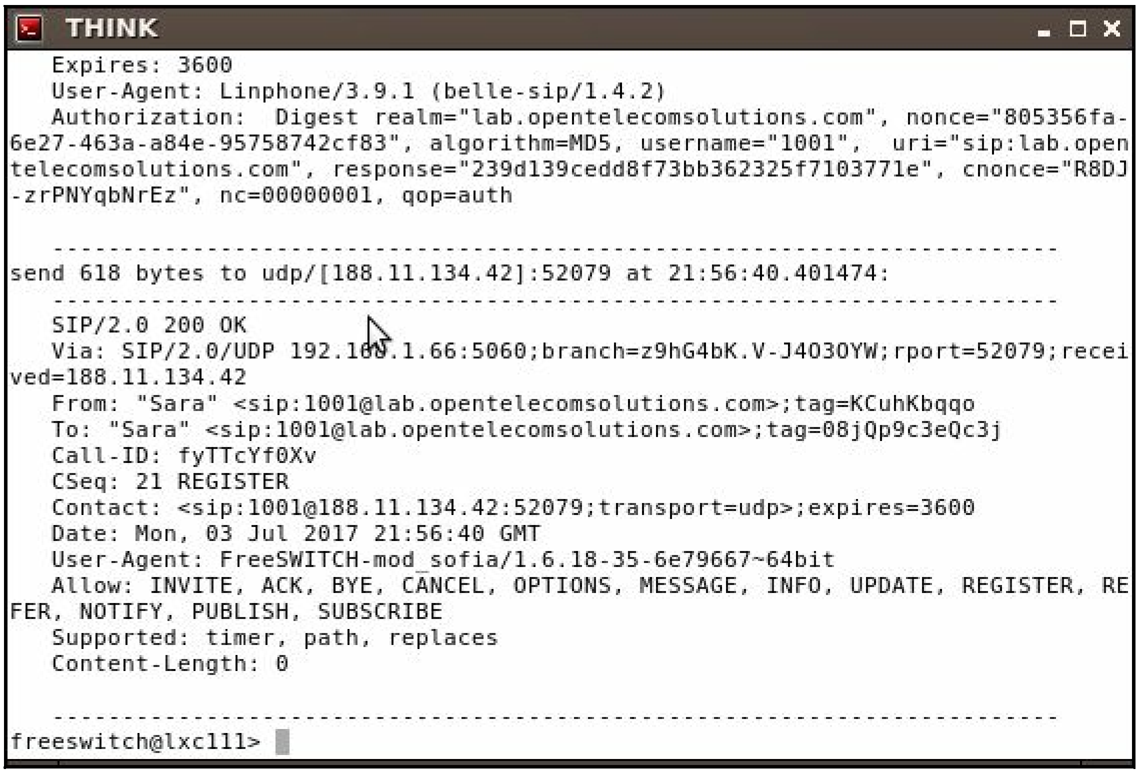
fsctl loglevel 7

For an interactive and real time check of what happens inside FreeSWITCH, nothing is so valuable as the console, or fs_cli. When connected, be sure to type "fsctl loglevel 7", so to pump up to debugging level. You'll be able to watch it all, from call creation to dialplan interpretation and execution.

Don't forget: you can filter logging on console so you only see output related to a particular call uuid, use the command "/uuid " followed by the uuid you saw with "show channels" or "show calls". If it's too late, grep freeswitch.log for the call uuid

sofia global siptrace on

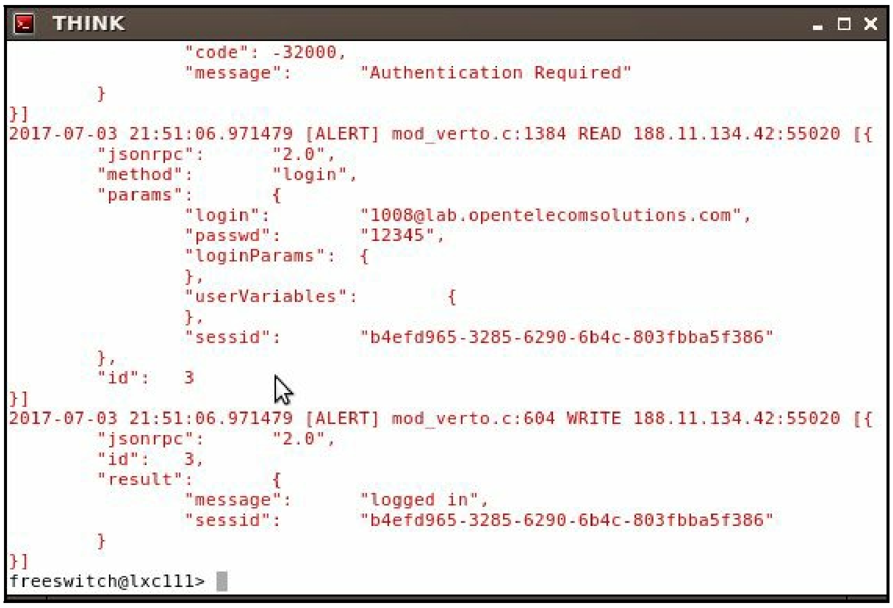
From FreeSWITCH console, you enable realtime output of SIP packets on all profiles with "sofia global siptrace on". This is particularly useful when saw in the context of a console running at loglevel 7: you will see the causal relationships between dialplan, processing, and SIP packets. Also, with a modicum of practice you'll be able to spot problems like incompatible codecs, NAT problems (wrong addresses in SIP packets), wrong passwords, non-matching domains, etc



```
   Expires: 3600
   User-Agent: Linphone/3.9.1 (belle-sip/1.4.2)
   Authorization:  Digest realm="lab.opentelecomsolutions.com", nonce="805356fa-
6e27-463a-a84e-95758742cf83", algorithm=MD5, username="1001",  uri="sip:lab.open
telecomsolutions.com", response="239d139cedd8f73bb362325f7103771e", cnonce="R8DJ
-zrPNYqbNrEz", nc=00000001, qop=auth

   ------------------------------------------------------------------------
send 618 bytes to udp/[188.11.134.42]:52079 at 21:56:40.401474:
   ------------------------------------------------------------------------
   SIP/2.0 200 OK
   Via: SIP/2.0/UDP 192.168.1.66:5060;branch=z9hG4bK.V-J4030YW;rport=52079;recei
ved=188.11.134.42
   From: "Sara" <sip:1001@lab.opentelecomsolutions.com>;tag=KCuhKbqqo
   To: "Sara" <sip:1001@lab.opentelecomsolutions.com>;tag=08jQp9c3eQc3j
   Call-ID: fyTTcYf0Xv
   CSeq: 21 REGISTER
   Contact: <sip:1001@188.11.134.42:52079;transport=udp>;expires=3600
   Date: Mon, 03 Jul 2017 21:56:40 GMT
   User-Agent: FreeSWITCH-mod_sofia/1.6.18-35-6e79667~64bit
   Allow: INVITE, ACK, BYE, CANCEL, OPTIONS, MESSAGE, INFO, UPDATE, REGISTER, RE
FER, NOTIFY, PUBLISH, SUBSCRIBE
   Supported: timer, path, replaces
   Content-Length: 0

   ------------------------------------------------------------------------
freeswitch@lxc111>
```

verto debug=10

If you set the parameter "debug" to the value "10" in /usr/local/freeswitch/conf/autoload_configs/verto.conf.xml, then all JSON protocol exchanges will be displayed in cleartext on console (they travels encrypted on the wire).



THINK

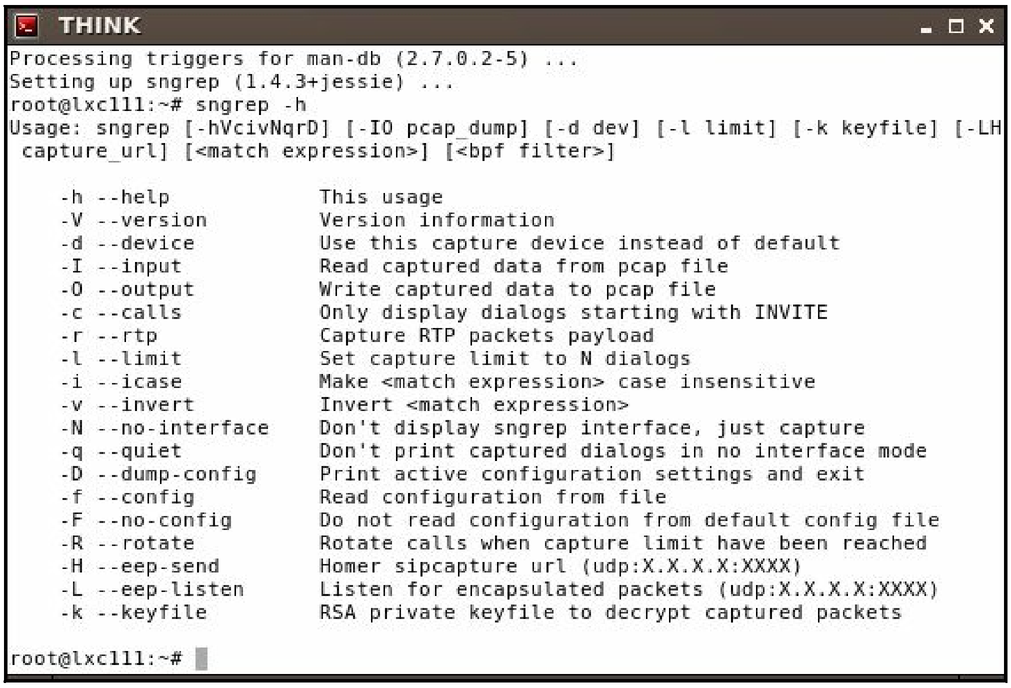
```
                "code": -32000,
                "message":      "Authentication Required"
        }
}]
2017-07-03 21:51:06.971479 [ALERT] mod_verto.c:1384 READ 188.11.134.42:55020 [{
        "jsonrpc":      "2.0",
        "method":       "login",
        "params":       {
                "login":        "1008@lab.opentelecomsolutions.com",
                "passwd":       "12345",
                "loginParams":  {
                },
                "userVariables":        {
                },
                "sessid":       "b4efd965-3285-6290-6b4c-803fbba5f386"
        },
        "id":   3
}]
2017-07-03 21:51:06.971479 [ALERT] mod_verto.c:604 WRITE 188.11.134.42:55020 [{
        "jsonrpc":      "2.0",
        "id":   3,
        "result":       {
                "message":      "logged in",
                "sessid":       "b4efd965-3285-6290-6b4c-803fbba5f386"
        }
}]
freeswitch@lxc111>
```

This will let you see any anomalies (like wrong passwords, or failed actions), and also let you learn more about the VERTO protocol itself.

sngrep: SIP dialogs visualization

This is the tool that has taken the world of telecommunication by storm. Don't miss it. It's even available as a Debian, CentOS, OSX and OpenWRT/LEDE package, ready to install! Go to https://github.com/irontec/sngrep for installation instruction, and complete documentation. We'll barely scratch the surface of what sngrep can do.

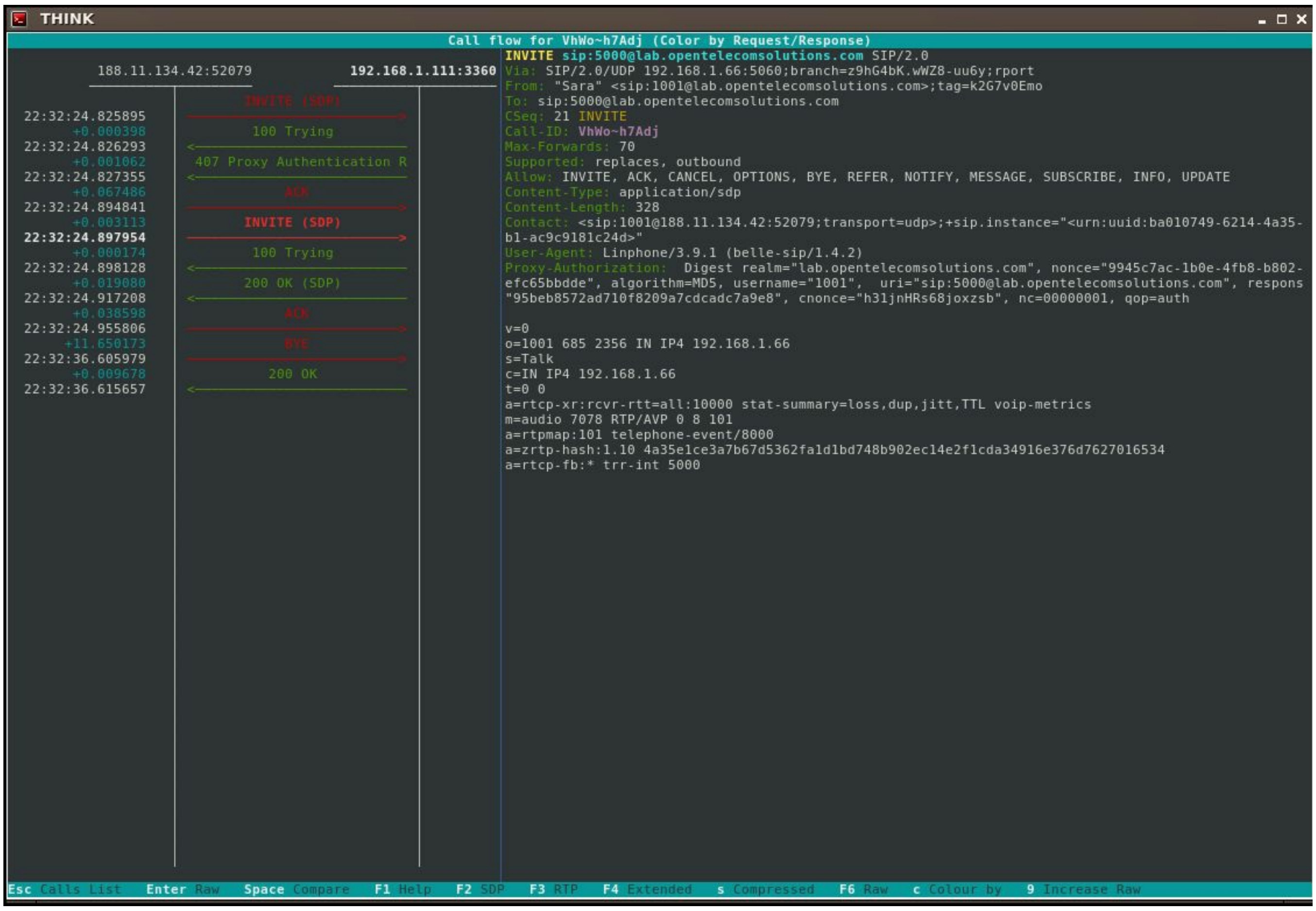
Developed by Ivan Alonso (IRC: kaian), its the perfect way to watch SIP traffic in a terminal connected to a remote machine.



```
Processing triggers for man-db (2.7.0.2-5) ...
Setting up sngrep (1.4.3+jessie) ...
root@lxc111:~# sngrep -h
Usage: sngrep [-hVcivNqrD] [-IO pcap_dump] [-d dev] [-l limit] [-k keyfile] [-LH
 capture_url] [<match expression>] [<bpf filter>]

    -h --help            This usage
    -V --version         Version information
    -d --device          Use this capture device instead of default
    -I --input           Read captured data from pcap file
    -O --output          Write captured data to pcap file
    -c --calls           Only display dialogs starting with INVITE
    -r --rtp             Capture RTP packets payload
    -l --limit           Set capture limit to N dialogs
    -i --icase           Make <match expression> case insensitive
    -v --invert          Invert <match expression>
    -N --no-interface    Don't display sngrep interface, just capture
    -q --quiet           Don't print captured dialogs in no interface mode
    -D --dump-config     Print active configuration settings and exit
    -f --config          Read configuration from file
    -F --no-config       Do not read configuration from default config file
    -R --rotate          Rotate calls when capture limit have been reached
    -H --eep-send        Homer sipcapture url (udp:X.X.X.X:XXXX)
    -L --eep-listen      Listen for encapsulated packets (udp:X.X.X.X:XXXX)
    -k --keyfile         RSA private keyfile to decrypt captured packets

root@lxc111:~# 
```

Following is a SIP dialog (call): on the left the time and delta of SIP message, the the caller source ip and port, the list of the SIP methods and responses exchanged (the one we're visualizing is in bold, the second INVITE), then the callee ip address and port, and on the right the full SIP message we're visualizing. Please note that this second INVITE contains (as per protocol) the authorization credentials (Proxy-Authorization) that were asked for by FreeSWITCH with the previous 407 response. FreeSWITCH will accept the credentials (and establish the call) with the next "200 OK" response, that will be confirmed by an "ACK" sent from the phone. The call will be terminated when the phone sends a "BYE" method, to which FreeSWITCH reply with a "200 OK" response.

Also, please note in the "body" of the INVITE message visualized in picture, we can see inside the media description (the SDP) the zrtp-hash that indicates that our client (a Linphone, actually) may attempt to negotiate ZRTP encryption of the media streams.



```
THINK
                                     Call flow for VhWo~h7Adj (Color by Request/Response)
                                                     INVITE sip:5000@lab.opentelecomsolutions.com SIP/2.0
          188.11.134.42:52079            192.168.1.111:3360 Via: SIP/2.0/UDP 192.168.1.66:5060;branch=z9hG4bK.wWZ8-uu6y;rport
                                                     From: "Sara" <sip:1001@lab.opentelecomsolutions.com>;tag=k2G7v0Emo
                          INVITE (SDP)               To: sip:5000@lab.opentelecomsolutions.com
  22:32:24.825895   ──────────────────────────>      CSeq: 21 INVITE
        +0.000398          100 Trying                Call-ID: VhWo~h7Adj
  22:32:24.826293   <──────────────────────────      Max-Forwards: 70
        +0.001062   407 Proxy Authentication R       Supported: replaces, outbound
  22:32:24.827355   <──────────────────────────      Allow: INVITE, ACK, CANCEL, OPTIONS, BYE, REFER, NOTIFY, MESSAGE, SUBSCRIBE, INFO, UPDATE
        +0.067486              ACK                   Content-Type: application/sdp
  22:32:24.894841   ──────────────────────────>      Content-Length: 328
        +0.003113         INVITE (SDP)               Contact: <sip:1001@188.11.134.42:52079;transport=udp>;+sip.instance="<urn:uuid:ba010749-6214-4a35-
  22:32:24.897954   ──────────────────────────>      b1-ac9c9181c24d>"
        +0.000174          100 Trying                User-Agent: Linphone/3.9.1 (belle-sip/1.4.2)
  22:32:24.898128   <──────────────────────────      Proxy-Authorization:  Digest realm="lab.opentelecomsolutions.com", nonce="9945c7ac-1b0e-4fb8-b802-
        +0.019080         200 OK (SDP)               efc65bbdde", algorithm=MD5, username="1001",  uri="sip:5000@lab.opentelecomsolutions.com", respons
  22:32:24.917208   <──────────────────────────      "95beb8572ad710f8209a7cdcadc7a9e8", cnonce="h31jnHRs68joxzsb", nc=00000001, qop=auth
        +0.038598              ACK
  22:32:24.955806   ──────────────────────────>      v=0
       +11.650173              BYE                   o=1001 685 2356 IN IP4 192.168.1.66
  22:32:36.605979   ──────────────────────────>      s=Talk
        +0.009678            200 OK                  c=IN IP4 192.168.1.66
  22:32:36.615657   <──────────────────────────      t=0 0
                                                     a=rtcp-xr:rcvr-rtt=all:10000 stat-summary=loss,dup,jitt,TTL voip-metrics
                                                     m=audio 7078 RTP/AVP 0 8 101
                                                     a=rtpmap:101 telephone-event/8000
                                                     a=zrtp-hash:1.10 4a35e1ce3a7b67d5362fa1d1bd748b902ec14e2f1cda34916e376d7627016534
                                                     a=rtcp-fb:* trr-int 5000

Esc Calls List   Enter Raw   Space Compare   F1 Help   F2 SDP   F3 RTP   F4 Extended   s Compressed   F6 Raw   c Colour by   9 Increase Raw
```

Using sngrep you will lower the barrier between you and SIP understanding, and a steep curve will become much more approachable.

tcpdump for saving net traffic in pcap files

When you ask for help in Real Time Telecommunication you will often (always?) be asked to provide a "pcap" file of your network traffic including the problematic session. Also, you will often want to remotely save the traffic in a file that you will later analyze (with sngrep or wireshark, etc).

The tcpdump utility is almost guaranteed to be available on any kind of operating system. Its most basic usage (save all the traffic sniffed by the Ethernet interface) is:

```
tcpdump -nq -s 0 -i eth0 -w /tmp/dump.pcap
```

You can filter what you save to only packets incoming and outbound from/to port 5060 (no protocol specified, so both TCP and UDP, full "traditional" SIP signaling):

```
tcpdump -nq -s 0 -i eth0 -w /tmp/dump.pcap port 5060
```

Asking for Help

OK, OK, we've all been there. Many of us have been there many and many times. The best ones between us are there at least once a day. Asking for help when you hit the outer border of your knowledge. This simply means you're pushing yourself beyond your limits. And this is good, if becomes an occasion to actually move your limits.

Help is there. FreeSWITCH has two incredible strengths: one is an exceptional community, extremely knowledgeable, and very respectful and caring toward newcomers. You let the community know you've done your part, you've researched the documentation and the archives, you've started from a sound configuration, but the problem is still there. Top experts will ask you for more info and will actually try to help you.

The other strength is that FreeSWITCH developers and top contributors are also professional consultants with decades of experience you can tap for commercial grade support and consulting. Your project technical success is assured!

Read the documentation first!

You will get much more convinced help, and you'll be able to better help the community help you, if you've done your part. Nothing is more irritating (eg, to me) than receiving a message or reading a mail with a question that would be answered by the first result in a Google search. That is not nice. We're all working people, and putting such a message in the stack of the ones we read (yes, we read the mailing list, the IRC and HipChat) is akin to waste our time because you think your time is such valuable you can't google around or read the online documentation.

Also, reading the available documentation will enable you to gather all information we need to answer your questions, and having a basic knowledge will allow you to ask the right questions.

confluence

http://freeswitch.org/confluence -the first source of knowledge you want to query is our Confluence documentation system. Is a web service similar to a wiki but interconnected with our development, bug marshaling and source code management environment. Has been developed by Atlassian, opensource, and its usage has no cost for open source projects like FreeSWITCH.

Confluence is a pleasure to search, and has up to the minute updated information and documentation about all that is FreeSWITCH or related to FreeSWITCH.

We put a lot of effort in our official documentation, please read it and search it.

books

Hehehe, I know you like books :). Other nice books about FreeSWITCH are also published by Packt, like this one. Their title are: "FreeSWITCH 1.6 CookBook" and "Mastering FreeSWITCH". We know the authors, we can guarantee for them.

Another book I always counsel is "VoIP Deployment for Dummies", published by Wiley. It's not for dummies at all, and (even though it was published in 2009) will give you a complete general course 400 pages long (covering topics including SIP, LAN metrics, dealing with ITSPs, and many more).

As reference works on SIP, RFCs are there for free, and the book from Johnston, "SIP, understanding the Session Initiation Protocol", Artech House, I found it complete and easy to consult when I look for details. For the architecture and the working of SIP based complex systems, "Session Initiation Protocol, Controlling Convergent Network", McGraw, is a good one.

I want to definitely signal here the book of my friends at OpenSIPS project, Bogdan-Abdrei Iancu and Flavio Goncalves, "Building Telephony Systems with OpenSIPS", again Packt publishing, because in its first part it lays down so well a working knowledge of SIP (that's needed for reading the rest of the book). Knowledge that's rooted in real implementation experience. Also, OpenSIPS as a SIP load balancer, proxy, registrar, etc is a project complementary to FreeSWITCH, and they are very often used together.

mailing list archives

I urge you to search the mailing list archives of FreeSWITCH. More than ten year of daily technical discourse about how to do things in and with FreeSWITCH. They are plenty of incredibly deep and accurate information that has been written by the brightest people discussing and solving problems. Not only you learn the answer to your question, the "how-to", but also you'll be able to follow the line of reasoning conducive to that.

Yay for the mailing list archives: http://lists.freeswitch.org/pipermail/freeswitch-users/

google

A lot of blog out there where people recorded their experiences with FreeSWITCH, and where deep technical discussions are offered about why a particular technique of application development has been chosen and how that project has been implemented. People like to describe their technical success stories. And sometimes their horror stories too.

How to ask for Community Help

So, you've tried hard to understand what's happening in your system, and the domain of the problem of your project. You've read the relevant docs and other people's experiences, but there is still something unclear, an aspect you do not understand, or something that simply does not square with the rest of your knowledge and experience.

Please, you're welcome, we're here to help. There is an entire community including hundreds of professionals that is happy to share and build knowledge. Interactively, in real time, via chatting system as IRC and HipChat, or asynchronously via mailing list.

Please be ready to provide relevant information:

- **Platform**, Operating System, Hardware, Virtualization Kind
- FreeSWITCH output of "**version**" command
- Your XML **Dialplan**
- Your **Script**
- FreeSWITCH **console debug output** (please, do not edit ip addresses, is not nice and messes things a lot)
- FreeSWITCH console **debug output** complete of **sofia siptrace** and **verto debug** output
- Network **packets capture** (**pcap** file) of relevant traffic

Those informations are not to be attached to mails, or (worst) dumped in IRC and HipChat. Those info must be made publicly available by a publicly available webserver, or by using FreeSWITCH own pastebin implementation, that has special syntax coloring, at https://pastebin.freeswitch.org/ In mails and chats you post the link to the required files and pages.

Be patient, people is working and is helping you in their free time, out of good will, and by a sense of community. Do not expect immediate answers just because **you are in a hurry**, or in an emergency. Being pushy will not getter better good will from people. Being clear and to the point, providing all relevant info, but not too much, will go a long way to help you getting the answers you looking for.

Also, keep in mind that **people lives in timezones**, and has holidays, etc. Most people in FreeSWITCH community live in North America, followed by Europe (west+est), and Asia (particularly India and Pakistan). Chinese speakers have developed their own FreeSWITCH community, and one of their leader (Seven Du Jin

Fang) is present and active also in the English speaking community. There is also a lively scene in Russian language.

IRC

This chatting system has a special place in the heart of any geek: a lot of technical discussions and mutual help have been taken place via IRC for so many years.

FreeSWITCH community IRC channel is "**#freeswitch**" (note the initial pound sign) on **irc.freenode.net**. You will need a client the like of **Pidgin** to connect to IRC, then choose to join our channel, and lurk for some time until you have "got" how it works. Then type "**~take-a-number**" followed by your question. Ask things politely but informally, without any introduction, without asking permission to ask, go straight to the point, and then **be patient**. Someone will answer your question, eventually. And you will be invited to provide more info. Be ready. If you got no answers after several hours, then ask again.

HipChat

HipChat is a new chatting system that has many advanced features not available in IRC, and is part and integrated with the Atlassian suite of software Development Management tools we are using. Particularly, HipChat chat history is always archived and is available to be searched, so nothing will be lost. We find it handy for our internal usage, and we have some rooms open also to the community at large. Best way to interact is to register for a free account on our bug tracking system (http://jira.freeswitch.org), then using that login and password with the native HipChat client for your computer. Google for HipChat client download. You will have to insert hipchat.freeswitch.org as server in settings. You can also join from the web, using this link: https://hipchat.freeswitch.org/

mailing lists

Mailing list in my perception is the main route for finding help. Asynchronous mails means that people will most probably find the right moment for answering, independent from which timezones or lifestyle they live in. You first subscribe to the mailing list, and after your registration has been approved (allow for roughly one day or so) you'll be able to post to the mailing list. Devote some time, at least at the beginning, to search and read the mailing list archives (see before), so you learn how information is exchanged. Then, without much ado, you welcome to ask your questions. Try to have an explicit and specific mail subject: "FreeSWITCH problem" or "FreeSWITCH does not work" will not get much attention from busy professionals. A mail subject like: "Timeout in playandgetdigit does not work for me" or "Registered user not reachable from failover machine" is guaranteed to grab curiosity of knowledgeable people. Eg: you're using precious time of other, busy, people. Do it in the best way.
You can subscribe to FreeSWITCH mailing list here: http://lists.freeswitch.org/mailman/listinfo/freeswitch-users

Official Commercial Support and Consulting

Professional support contracts and project consulting is available by writing to "**consulting@freeswitch.org**". You will be answered by FreeSWITCH Solutions and introduced to all commercial procedures, a person will be in charge of defining your case. Resources will be assigned to your case, and your problems and worries will fade away. FreeSWITCH Solutions, the official company of FreeSWITCH core developers, is also reachable at http://freeswitch.com/.

Reporting Problems

What if you hit a bug? First of all try to understand if actually you hit a FreeSWITCH bug. Check all the things around FreeSWITCH that can be cause of malfunctioning: network, firewalls, databases, upstream and downstream providers, etc. OK, let's say you checked it all.

Then, download and install latest stable FreeSWITCH version. You will be asked: "**is this problem affecting latest stable?**". If you're unable to answer "yes" (and give proof of it), your problem will not get any attention (maybe is fixed in latest stable).

Then, do the same with latest development version of FreeSWITCH, called "MASTER". "**is this problem affecting master?**" Be ready to answer this question too. If is solved in master, then the fix will be backported to stable, if that's possible at all.

don't report a bug on mailing list !

One things that our developers and contributors hate is the reporting of bugs to the mailing list. In some way, for some reason, people believe they can jot down a mail to mailing list, and then one of us will take care to read that mail, and then do the careful filling of the very detailed form for bug reporting. Not at all, this is not gonna happen. You will only have a slightly annoyed answer the like of: "**this is the ten thousandth time we repeat: bug reports belongs to Jira, do not report bugs in mailing list**". If we would use mails as bug tracking tool, most of the bugs reported would never be fixed, will come back (regression), and/or be forgotten. Use Jira for bugs. Not only to file a bug report, but also to search already reported bugs. Maybe someone else has already reported your problem, that was not a bug, and the Jira issue resolution will tell you how to make it work.

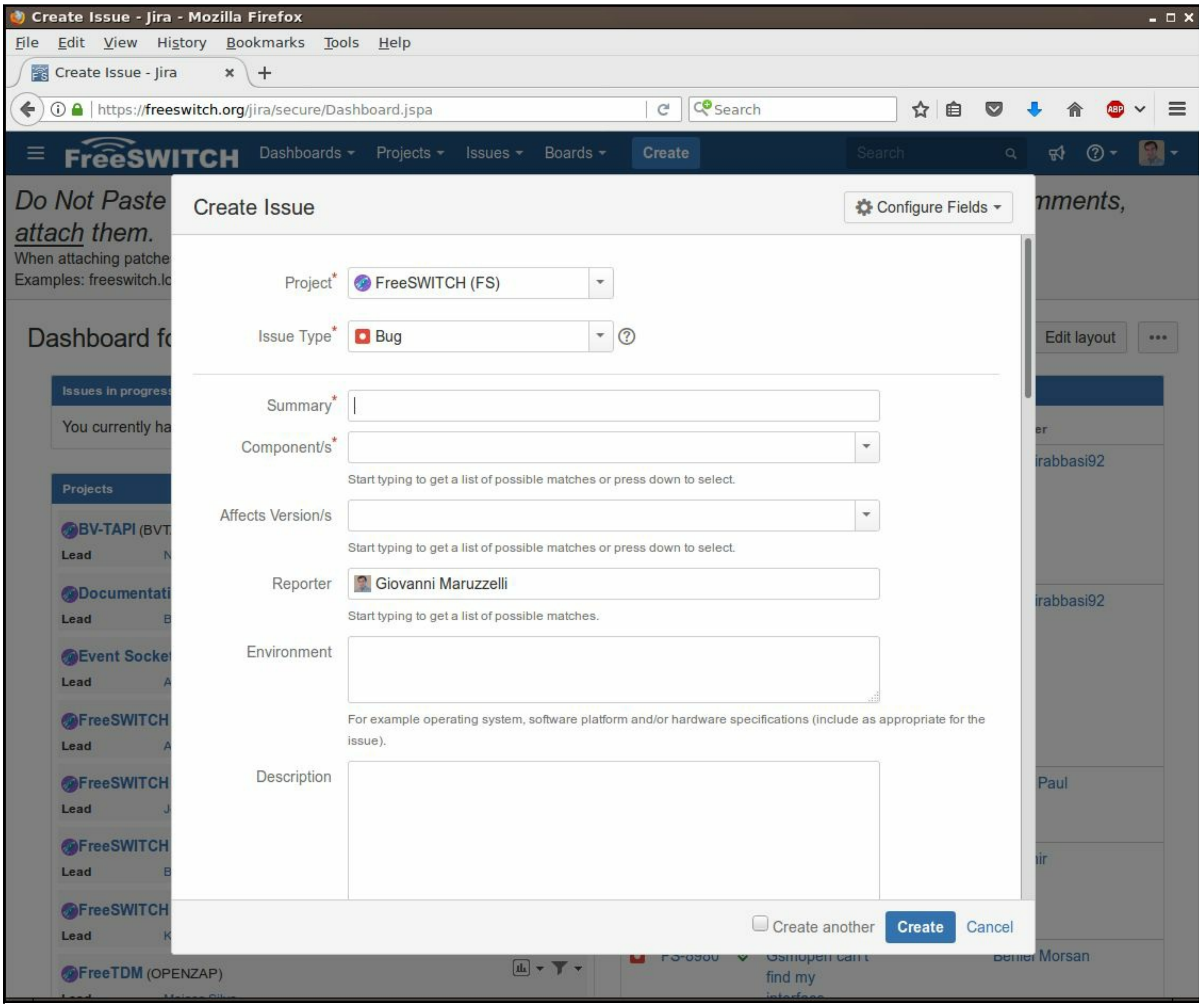
how to file a bug on Jira

First impact with http://jira.freeswitch.org can be intimidating, but its not. Listen: is easy and fast. And is also your only option, this way or the highway.

So, you register and obtain a jira account, then you can create an "issue". You will be presented with a long and daunting form, but actually what we want is to be certain that we have all the information we need to reproduce the bug, and that you have reproduced the bug on latest FreeSWITCH version.

Try to be the more complete and precise possible, while not writing too much.

Particularly all patches, logs, relevant files, scripts, etc must be **attached** (as in "**uploaded**") not cut and pasted in comments. Also take care of the filename extensions of the attachments, so they will be recognized by browser when downloaded: a patch or a diff file will have extension .diff, a logfile will have extension .log, etc


Create Issue - Jira - Mozilla Firefox
File Edit View History Bookmarks Tools Help
https://freeswitch.org/jira/secure/Dashboard.jspa
Dashboards Projects Issues Boards Create
Configure Fields
Create another
Create
Cancel

Create Issue

Project* FreeSWITCH (FS)

Issue Type* Bug

Summary*

Component/s*

Start typing to get a list of possible matches or press down to select.

Affects Version/s

Start typing to get a list of possible matches or press down to select.

Reporter Giovanni Maruzzelli

Start typing to get a list of possible matches.

Environment

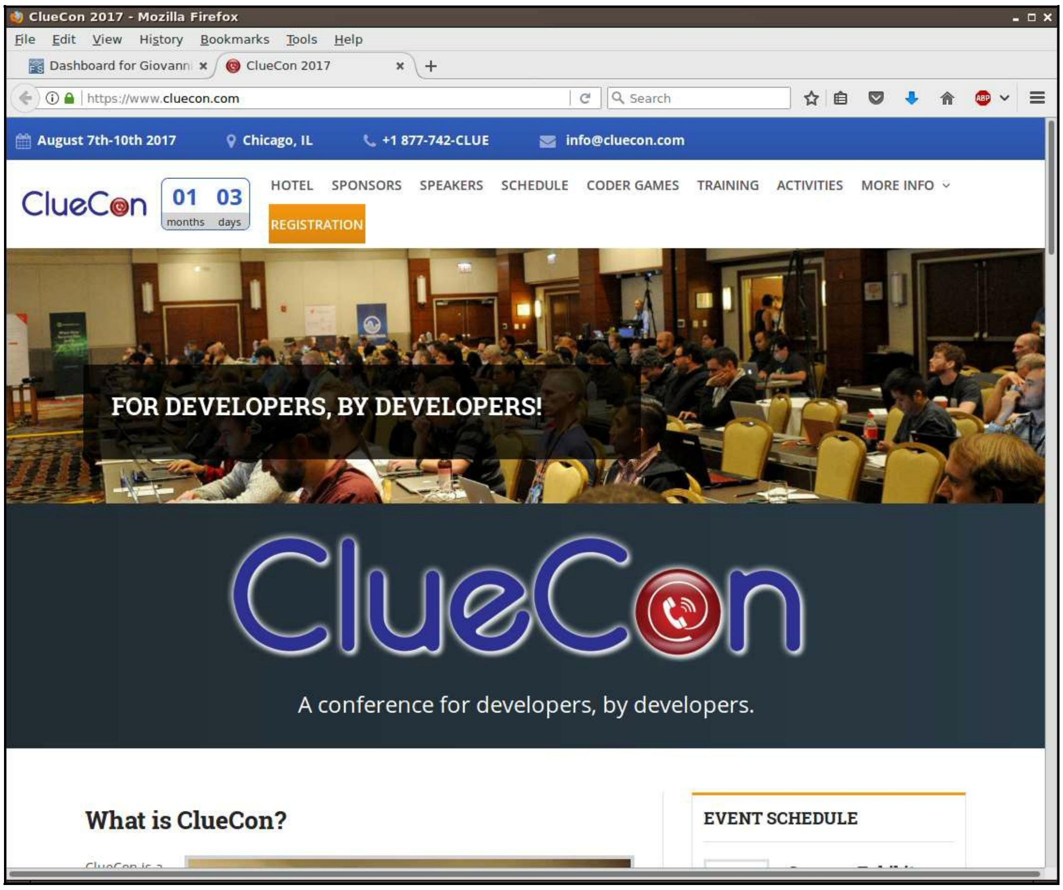
For example operating system, software platform and/or hardware specifications (include as appropriate for the issue).

Description

When you finished posting your bug report, first of all feel proud and happy because you have helped the FreeSWITCH project and community. And then be patient. You will be automatically alerted of comments, questions, issue assignment, resolution, rejection, etc If asked for more info, provide them immediately. There is nothing else you can do. Just wait. Ohum. Actually, if you have a paying interest in seeing your jira issue to have an higher priority over other jira issues, then write to consulting@freeswitch.org for costs and details.

ClueCon

ClueCon is our annual convention, the Convention of those who have a Clue in Real Time Communication and Internet of Things. August of each year, in beautiful Chicago (nicknamed "the Windy City", there is the River, is in the North, is kind of fresh and breathable. Eg, is not Phoenix, Matt. OK, is neither Fargo) all professionals VoIP, WebRTC, Makers, Visionaries and Scientists come together for a week of coding, fun, presentations, pizzas, training and games. All open source projects related to communication converge there, and you will found yourself rubbing elbows with your development heroes. Check the website https://www.cluecon.com/, see the speakers and the agenda, and register NOW for next August (the earlier the registration, the lower the cost).


ClueCon 2017 - Mozilla Firefox
File Edit View History Bookmarks Tools Help
Dashboard for Giovanni
ClueCon 2017
https://www.cluecon.com
Search
August 7th-10th 2017
Chicago, IL
+1 877-742-CLUE
info@cluecon.com
ClueCon
01 months 03 days
HOTEL SPONSORS SPEAKERS SCHEDULE CODER GAMES TRAINING ACTIVITIES MORE INFO
REGISTRATION
FOR DEVELOPERS, BY DEVELOPERS!
ClueCon
A conference for developers, by developers.
What is ClueCon?
EVENT SCHEDULE

Summary

This last chapter gave you an overview on how to troubleshoot problems in FreeSWITCH, what are the potential suspects (firewall, network, router, configuration), and how to check them.

Then we saw how to visualize the debug logs of FreeSWITCH and the signaling part of communication: SIP and Verto.

We looked at which documentation to read, where to find relevant information for solving our problems: mailing list archives, official documentation site, blogs, books.

If the problem is still there, and we've already researched it all, then we ask for help. Free, best effort help from community, commercial professional help from consulting@freeswitch.org.

If we believe to have encountered a FreeSWITCH bug, we must report it to https://freeswitch.org/jira/secure/Dashboard.jspa. Don't even bother to report it in mailing list.

As book closure we had a brief description of ClueCon, a Developer's Paradise where we annually gather in Chicago to have serious fun.