Emergency Call Handling in Open Source Soft PBX

KIRUPAKARAN PIRASATH

MASTER OF SCIENCE IN INTERNETWORKING

UNIVERSITY OF ALBERTA

2011

Abstract

This document provides complete solution to pick a telephone line to free the resources so another user can dial 911 via a congested open source based PBX. Picking the line will be based on a particular algorithm which is developed during this project. At last, a working script is provided that can be implemented in any open source PBX environment with minimal adjustments to configuration files of that open source PBX.

There were several attempts made in the past by some developers but there were no working solutions found. Therefore, this was taken as a challenge and complete solution is developed based on Java scripting. To understand this material it is adequate enough to have basic knowledge in Asterisk PBX and Java Programming language.

This project report was submitted as a part of MINT 709 requirement and I hope this will benefit all who are interested in open source PBX scripting.

Acknowledgements

I would like to thank Dr. Mike MacGregor and Mr. Shahnawaz Mir for providing this opportunity and supervising this project. They were very helpful during this period in various ways and actively involved in improving the design of this project.

Table of Contents

List of Figures and Tables

List of Abbreviations and Acronyms Used in this Report

ADM	-	Asterisk Desktop Manager
AGI	-	Asterisk Gateway Interface
AJ	-	Asterisk-Java
AMI	-	Asterisk Manager Interface
API	-	Application Programming Interface
CEO	-	Chief Executive Officer
CLI	-	Command Line Interface
DTMF	-	Dial-Tone Multi-Frequency
DAHDI	-	Digium Asterisk Hardware Device Interface
FIFO	-	First In First Out
GUI	-	Graphical User Interface
iLBC	-	internet Low Bitrate Codec
IP	-	Internet Protocol
IVR	-	Interactive Voice Response
LIFO	-	Last In First Out
MGCP	-	Media Gateway Control Protocol
OS	-	Operating System
PBX	-	Private Branch eXchange
PC	-	Personal Computer
PHP	-	Hypertext Preprocessor(recursive)
POTS	-	Plain Old Telephone System
PSAP	-	Public Safety Answering Point
PSTN	-	Public Switched Telephone Network
SIP	-	Session Initiation Protocol
TCP	-	Transmission Control Protocol

TDM	-	Time Division Multiplexing
UML	-	Unified Modelling Language
VLAN	-	Virtual Local Area Network
VoIP	-	Voice over Internet Protocol
ZAP	-	ZAPtel telephony interface

1. Introduction

Since the evolution of Plain old Telephone System (POTS), it has gone through many different phases with time. Legacy analog telephone technology had many limitations compared to digital technology which addressed many of them. But still, digital phone systems had limitations such as number of users it can handle compared to how many phone lines that were purchased from telephone service provider. Invention of fully fledged PBX systems provided answers for many of these limitations such as extension dialling, call forwarding, hunt groups, etc. Hardware PBX systems became popular during 1990s and many small, consumer-grade and consumer size PBX made available in the market. Most of the small and medium enterprises started to buy these handy compact devices to manage and optimize their telephone systems. During 1990s, there was a huge growth in data network and many people started to talk about packet switched networks. Intensive research in packet switched network paved way to transport voice over these networks efficiently. Not only that, but the availability of Internet as a global delivery system was a very crucial factor in the development of Voice over IP(VoIP) PBX systems. As a result of this, Virtual PBX systems became popular that provided solution for complex communication system. This integration of data and voice characteristics makes the communication very easy within the organization.

Research and development in open source systems was an important milestone in computing world that brought an end to vendor monopolies by eliminating vendor lock-in and provided the complete control to the owner. More and more open source applications were invented therefore its popularity increased exponentially during short span. Open source applications and systems are still very popular in computer arena. Open source application software on a Linux based entry level server gives total control at a very little cost. Nowadays, Open source PBX are replacing the legacy PBX systems at very rapid space. These software based PBX are preferred for many reasons such as low costs of ownership and maintenance, scalability and integration of many different applications and abundant support. When it comes to have a closed source PBX system, it incurs high initial and maintenance costs. An open source PBX can be built with a PC, telephony card and some effort which doesn't cost much. This provides organizations an

excellent cost advantage over legacy PBX models that charge largely for their proprietary software. Tremendous advantage in production and maintenance costs has made the vendors and developers to focus in promoting innovation into these fabulous systems. Many open source PBX developers have emerged in the past decade with great success such as Asterisk, FreeSwitch, etc. Among these, Asterisk became very popular among most organizations due to low initial and management cost. Asterisk, created by Mark Spencer of Digium, has been the leader in open source PBX domain for many years providing telephony services including PSTN and VoIP. Asterisk software provides many features that are available in traditional PBX systems: call conferencing, voice mail, automatic call distribution and many extended new features like Interactive Voice Responses (IVR), Local and Remote Agents, Macros, etc. Apart from this, Asterisk supports many voice and video codecs including the latest iLBC, Speex, Polycom and protocols like SIP, Skinny, H.323, Google Talk, Skype, MGCP and UNIStim. The complete supported features, codecs and protocols can be seen in
http://www.asterisk.org/support/features

With this increasing popularity of open source PBX systems, many applications and add-ons are being created by developers. For example, there are many applications in the market which have the capability of talking to Google Talk or Skype server from an Asterisk based PBX. Asterisk's Jingle protocol allows to embed Google Talk with Asterisk with ease. Many organizations have implemented Skype for Asterisk which enables them to make Skype-to-Skype calls, receive calls with online numbers, set and retrieve online status, etc.

Although there are several useful applications available for Asterisk, applications that are related to emergency calling are more important than anything. Emergency situations may arise at any time in our environments. First and foremost thing that comes to everyone’s mind during emergency situation is 911. When an emergency happens at home or any outdoors, we might use our home phone or mobile phones to dial out the closest Public Safety Answering Point (PSAP). In this situation, caller might get connected 911 probably within seconds, but will it be the same in a busy organization with less availability of cellular network? Mostly in large organizations, extensions are connected to a centralized or a hosted PBX system through which all the outside calls will be processed. The simultaneous outgoing calls will be determined by the trunk

availability in that organization. For example, in a small factory there are 50 extensions and 10 outgoing trunks connected to service provider. What if an emergency situation comes up and someone needs to dial 911 when all 10 outgoing lines are busy? How would this situation be handled in an open source PBX? This can be done in many ways.

1. Arbitrarily pick one line and hang up that line to proceed with 911.
2. Pick a line based on FIFO or LIFO algorithm.
3. Pick a line based on the organization business hierarchy

Option 1 may not be a suitable one for this scenario as the system is having the control in selecting an arbitrary line to disconnect which may be of a senior staff or someone who is in an important call. Even if we assign highest priority for CEO, with the option 1, still his line could be still disconnected over other staff. On the other hand, importance of a call depends on several factors such as the position of the person in the business hierarchy, the flag assigned for the call or the priority mechanism regardless of who joined the system first or last. So, the second option, LIFO or FIFO can also be eliminated here. The best pick among these will be the third because it provides control to the designer to pick the right line based on the requirements. Considering option 3, the algorithm was proposed as follows

- Assigning priorities based on user groups in an organization. This user groups can be selected based on several factors such as business hierarchy, etc.
- This assignment of priorities will be ONLY for the calls made outside of the organization; not for the extension to extension dialling and for all incoming calls through PSTN.
- Selection criteria of disconnecting the non 911 lines will be based on the lowest priority in the call stack.
- 911 will be having the highest priority over any other calls or user groups.
- Every time 911 are connected to the PSAP, the global counter EMERGENCY will be incremented by one.
- 911 calls will be never disconnected.
- When all outgoing lines are in use with 911, the next users attempt to dial 911 or non 911 will hear congestion tone or an IVR.

- Global counter, EMERGENCY, will be decreased by 1 after every call is hung up with 911.

Based on this algorithm, this below flow chart was developed and used as the base input for this project.

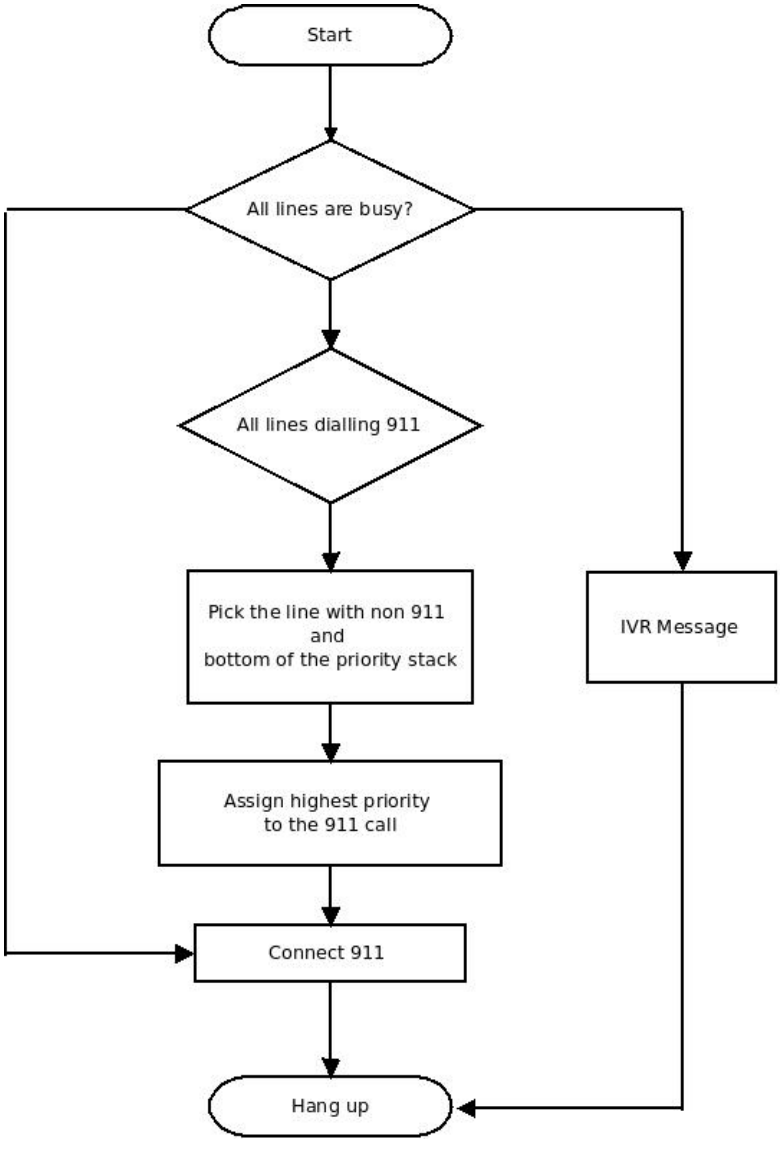


Figure 1.1: Flow chart of proposed solution

2. Literature review

2.1 Asterisk-A Brief Overview

Asterisk is controlled by dial plans that are written in Asterisk Control Language. These dial plans can control the flow of algorithms when the calls are arrived to it over the available channels. This dial plan logic instructs the Asterisk how to handle the incoming and outgoing calls, facsimile, voicemail and many other useful applications. Generally, Asterisk dial plans reside in /etc/asterisk of the UNIX file system. Extensions.conf, extensions_additional.conf and extensions_custom.conf are some of the well-known files that are built based on dial plans to control the call flow in Asterisk. A simple dial plan might look like this

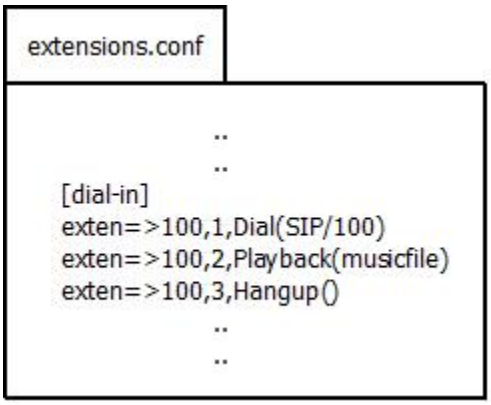


Figure 2.1: Sample of a dial-plan inside the file extensions.conf

As shown above, dial plans are built using the blocks called contexts. In Asterisk, contexts are backbone of the system and indicated by square brackets with its name around it. Typically, the first line of the dial plan starts with the keyword “exten” followed by symbol =>. This symbol informs Asterisk that it will be followed by a command. These contexts are made of three components; extension, priority and application. In the above example, the numerical value 100 indicates the extension. The terminology, priority, is having a different meaning in Asterisk than what we might think of. Voip-info.com provides the best description for this as “Priorities are numbered steps in the execution of each command that make up an extension”. Usually, priority

numbers start with the numerical value 1 and consecutively increment for each line in the contexts. Third and final piece that completes the command is the application. These applications provide different call methods to the system. If we look back the previous example, line 1 can be expressed as, “WHEN the extension 100 is dialled start this dial plan by calling the application Dial with the input parameters SIP/100 which tells the application to use SIP as the communication protocol and dial extension 100”. Then the second line tells Asterisk to load and play the music file. Finally, when playback is finished, the application Hangup will be called and disconnected with the caller. Some useful Asterisk dial plan applications can be found at http://www.voip-info.org/wiki/view/Asterisk+-+documentation+of+application+commands

2.2 Variables in Asterisk

Another powerful feature of Asterisk is the capability of assigning user-defined variables within its structure. This functionality provides the scalability to the Asterisk system. In Asterisk, variables can contain numbers, letters and strings. Asterisk variables are beneficial not only because it allows creating instructions for call flow but also it makes easier to adapt future changes in the telephony applications. Asterisk dial plans can have three types of variables assigned within its scope; global, shared and channel variables. Apart from this, Asterisk is having the capability of handling the UNIX environment variables as well. The Asterisk channel variables, which can only set values for the current, active channels while global variables are reachable in all channels even when the channels are not active. A variable name may be any alphanumeric string beginning with a letter. Usual naming convention in Asterisk is that the user-defined variables are not case sensitive while Asterisk-defined ones are case sensitive.

2.2.1 Global Variables

In Asterisk 1.6 and above, these variables can be set under the [global] category of extensions.conf file or with the keywords “Set” and “GLOBAL” within any contexts. It is a good practice to assign the global variables inside the [global] category than setting them inside the contexts as it is convenient. Creating a global variable using this method is very easy by just

using the variable name followed by the equal sign and the value, as done in most programming languages.
Global variables can also be defined inside contexts and reached with the GLOBAL () dial plan function and the set application as follows,

exten => 100,1,set(GLOBAL(Variable)=value)

2.2.2 Channel Variables

Every channel in Asterisk is having its own variable space. In other words, there will be no collision between different calls even if they have similar channel variables. All the channel variables assigned to a channel will die when that channel is hung up. Asterisk syntax to setup the channel variable is defined with the set() dial plan application.

exten => 100,1,set(Variable=value)

2.3 Previous research and works on this topic

Although similar work has been done by many asterisk developers, most of the algorithms are adopting our first case scenario in Chapter 1. In most cases, the provided solutions were not fully functional or not tested in live environments. In most of the available solutions, developers have focused on pure Asterisk based scripting than using any other scripting languages. Asterisk scripting is easy to implement but it has some limitations and low in efficiency. Now let's see some proposed methods by other developers

Solution 1

Source: http://www.voip-info.org/wiki/view/Asterisk+tips+911

This was probably the first solution provided under this subject and the script was implemented purely based on Asterisk scripting. The extensions.conf script is as follows

```
[globals]
EMERGENCY=0
LINEOUT=Zap/1

[nineoneone]
exten => 911,1,ChanIsAvail(${LINEOUT})
exten => 911,2,SetGlobalVar(EMERGENCY=1)
exten => 911,3,Dial(${LINEOUT}/911)
exten => 911,4,SetGlobalVar(EMERGENCY=0)
exten => 911,5,Hangup
exten => 911,102,GotoIf($[${EMERGENCY} = 1]?999)
exten => 911,103,SoftHangup(${LINEOUT}-1)
exten => 911,104,Wait(1)
exten => 911,105,Goto(1)
exten => 911,999,Congestion()
```

As it can be seem above, two global variables are defined in the extensions.conf before implementing the scripts. First global variable was intended to keep track of the status of emergency call in the ZAP trunk. LINEOUT, the second global variable, remembers the physical name of outgoing trunk that can be used instead of the string “Zap/1” anywhere inside the script. The [nineoneone] context provides the script to be called when 911 is dialled. This algorithm checks for the availability of the Zap trunk in its first command line. ChanIsAvail, an important Asterisk application, was introduced in Asterisk 1.4 and takes trunks or technology as its arguments and returns to the consecutive line if any channels available in that given trunk. When no channels are available, the priority will have a jump of 101 to execute the command. So in the above algorithm ChanIsAvail will work like below,

```
If (ChanIsAvail==true)
        goto priority 2;
else
        goto priority 102;
end
```

If the line is available, priority two will be implemented else it will jump to line 101+1, to implement the line GotoIf($[${EMERGENCY} = 1]?999). The knotty line here is 103 that does a soft hang-up on Zap trunk which disconnects one arbitrary channel from the group. This is not the desired output in our scenario. This solution doesn't not have a control mechanism to select the desired channel based on some algorithm; instead it randomly picks a channel to disconnect. The second issue of this script is that the global variable, EMERGENCY, is never set to zero after the disconnection of 911 calls. This will cause the system from dialling out 911 after the first time this script is called. The designer claims that he never tested this solution.

Solution 2

Source: http://www.voip-info.org/wiki/index.php?page_id=365

```
[globals]
EMERGENCY=0
EMERGENCY_TRUNK=Zap/17
; Change this for production use:
EMERGENCY_NUM=some_test_phone_number

[nineoneone]
exten => s,1,SetVar(SET_EMERG_FLAG=0)
exten => s,n(checkavail),ChanIsAvail(${EMERGENCY_TRUNK})
exten => s,n,SetGlobalVar(EMERGENCY=1)
exten => s,n,SetVar(SET_EMERG_FLAG=1)
exten => s,n(dial),Dial(${EMERGENCY_TRUNK}/${EMERGENCY_NUM})
exten => s,s+2(trunkbusy),GotoIf($[${EMERGENCY} = 1]?inprogress)
exten => s,n,SoftHangup(${EMERGENCY_TRUNK}-1)
exten => s,n,Wait(12)
exten => s,n,Goto(checkavail)
exten => s,s+2(inprogress),Congestion
exten => s,checkavail+101(notavail),Goto(trunkbusy)
```

```
exten => h,1,GotoIf($[${SET_EMERG_FLAG} = 1]?3)
exten => h,3,SetGlobalVar(EMERGENCY=0)

[your_main_context]
exten => 911,1,Goto(nineoneone,s,1)
```

This is a slightly modified solution to the previous one which sets the global variable back to its initial value of zero. Another interesting thing here is that the designer has introduced a channel variable, SET_EMERG_FLAG, to track the status of the channel that triggers the global variable. If we see the extension name of the [nineoneone] context, it reads as s at the beginning and h at the end of the context. The lowercase letter s says to implement the commands for every extensions regardless the dial patterns while letter h will be called as soon as the Hangup(not SoftHangup) application is called. But still, this solution does not fulfil our requirement. Same as the previous solution, it picks the disconnection channel randomly.

As seen in the previous works by different developers, the solutions do not match with the requirements or lack in performance. This solutions could be modified to get the desired output provided a control mechanism to select the right channel to disconnect. It can be done in several ways. For instance, we can call another context within the [nineoneone] to do the channel-pick and then return back to the [nineoneone] context with the name of the channel. But, performing arithmetic and logical operations or conditional loops inside Asterisk is neither efficient nor a cakewalk. Not only that, Asterisk lacks in complex data structure handling, database integration, error handling and most importantly object oriented design capabilities. So, integration tools were badly needed to get optimal use of Asterisk system and to develop Asterisk based applications. Generally external scripts provide very efficient and custom call handling by minimizing long lines of codes needed in pure Asterisk based solutions. Not only that, scripting languages like PHP, Perl, Python, Java, etc is easy to work with for someone who is having some programming experience and it allows the scalability by integrating with other systems. But one major challenge in writing the external scripts is the interfacing between the Asterisk and the scripting language. Bridging Asterisk and scripting languages is not an easy task for an average or intermediate programmer, thus they can always use Asterisk frameworks available in many

languages. Below are some cross platforms available for Asterisk on different scripting languages.

Framework	Language	URL
Adhearsion	Ruby	http://www.adhearsion.com/
py-Asterisk	Python	http://code.google.com/p/py-asterisk/
Asterisk-Java	Java	http://asterisk-java.org/
Asterisk-PERL	Perl	http://asterisk.gnuinter.net/
PHPAGI	PHP	http://phpagi.sourceforge.net/
MONO-TONE	.NET	http://gundy.org/asterisk
AstOCX	Active X	http://www.pcbest.net/astocx.htm

Table 2.1: List of available AGI frameworks for Asterisk scripting

2.4 The AGI-Asterisk Gateway Interface

Asterisk itself is having the capability of sending commands and receiving responses from an internal/external system. This operation is achieved via Asterisk Gateway Interface (AGI) class which is inbuilt to the Asterisk system. AGI application of Asterisk server sends the call to launch the scripts within the Asterisk dial plans. Asterisk sends and receives commands and responses through standard UNIX channels of STDIN, STDOUT and STDERR. A simple AGI call in a dial plan might look like below

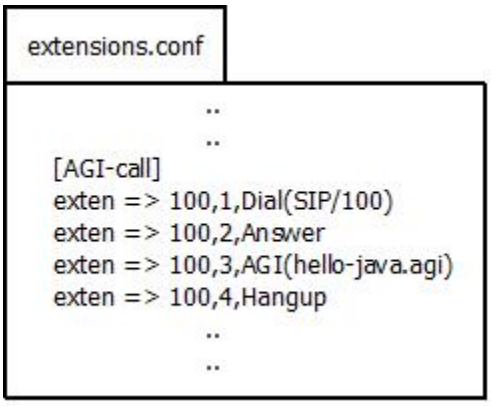


Figure 2.2: Sample AGI call inside a dial plan

In the above example, when the dial plan AGI-call is called from the main context, it starts to dial out extension 100 using the SIP technology and answers the call. Then it implements the third line, which calls the external script usually residing at /var/lib/asterisk/agi-bin/. This location might differ based on the Asterisk version and flavour. Below UML diagram explains the flow of processes during an AGI call. AGI has so many favours such as Simple AGI, Async AGI, Fast AGI, Dead AGI, etc., and these will be briefly discussed in the next chapter.

3. Designing the Script

This chapter will comprise of the design methodology for this project with all necessary diagrams, charts and processes. Further, this design sector includes both Asterisk and Java scripts that were deployed in the real environment for testing. Although there were lot of design tools available for Asterisk based systems such as Visual Dialplan GUI, Asterisk Manager Suite and Asterisk Desktop Manager (ADM), none of these tools were used to design the solution. One of the main reasons behind not relying on these tools was that those tools did not have the functionality or the additional features that were needed in this design process. For instance, the famous Visual Dialplan GUI was not suitable for this project as it does not have the ability to handle the call of external scripts. Considering these, it was decided to carry on with non-computer-aided design approach that gives freedom to designers' ideas.

3.1 Specification

Before the design, a brief list was generated based upon the project requirements. These requirements were consist of both hardware and software. It was decided to use one of the available servers in the lab as Asterisk server with Version 1.6. Although Asterisk has lot of distributions such as FreePBX, Trixbox, Callweaver, etc, Elastix was chosen over others considering the user-friendliness and its popularity in the VoIP market. Elastix was a unanimous selection considering its stability that is provided by Cent OS platform. The complete project specification is given below

- A PC with at least 512MB/20GB/1 core
- Elastix 1.6 (bundled with Cent OS and Asterisk 1.6)
- a TDM card - Wildcard TDM410P Board 1
- Cisco 7960 Phones and Yate VoIP SIP clients

SIP has been a very successful and widely adapted technology in VoIP telephony. As many organizations rely on this famous technology, it was decided to use SIP as the underlying protocol to communicate between the devices during this project.

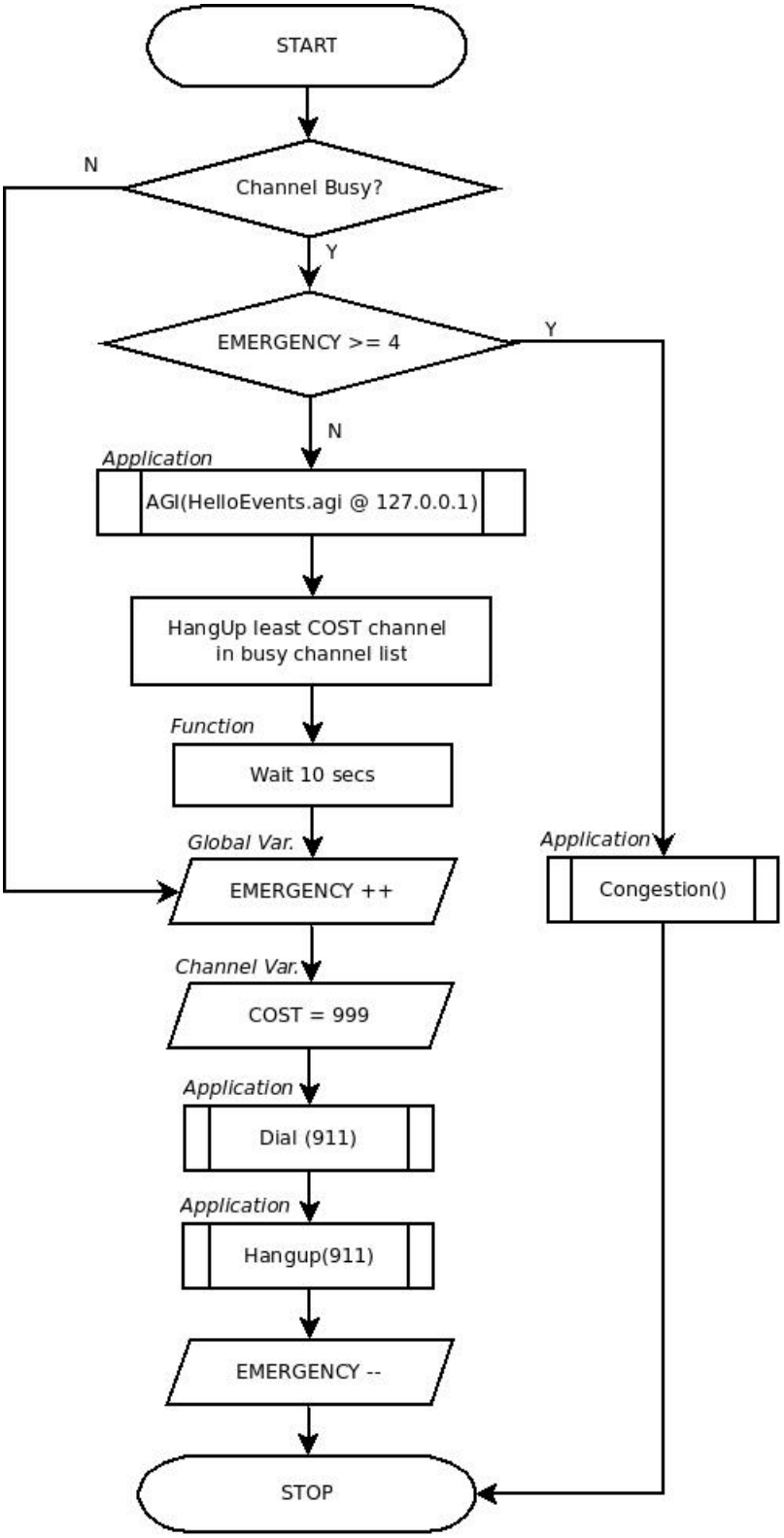


Figure 3.1: Modified Flow Chart of the Project Proposal

3.2 Design

As discussed in the previous chapter, the base algorithm was the main input for the design phase. The base algorithm was just an abstract thus a detailed flow chart was developed based on the discussions with the supervisor. The new flow chart can be seen in the previous page. This new flow chart provides more insight to the proposed solution compared to the previous flow chart. This flow chart includes some Asterisk applications and functions that should be used during the implementation stage.

The context [emergency] has to be called from the main context from which the 911 call is initiated. This script will be called only when it tries to dial out 911 using outgoing trunk. During feasibility study, it was monitored that the main contexts call another context, [macro-dialout-trunk], when they need to connect with outside world. The capture from Asterisk for this is given below

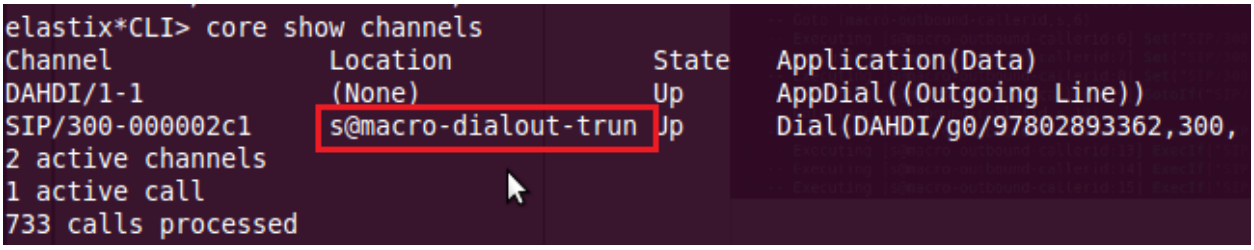


Figure 3.2: Snapshot that shows [macro-dialout-trunk] dial out of Asterisk

So, it clearly tells about the place where the priorities have to be set for user groups when they try to connect outside world. Yes, the priorities should be implemented at [macro-dialout-trunk] based on the different access groups. For instance, when CEO tries to call one of the top clients of their organization, priority of his call could be set to 600. In the same manner, rest of the user groups can be assigned with different priorities.

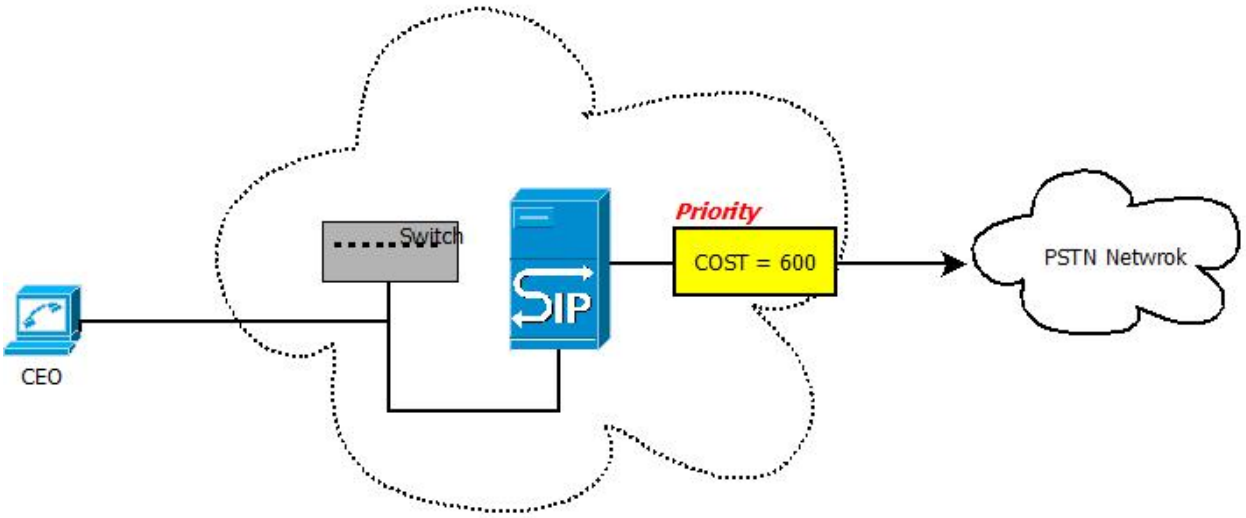


Figure 3.3: COST assignment for CEO in Asterisk

In this design, our test user groups-CEO, manager, staff, student and visitor-will be having the priority values of 600, 500, 400, 200 and 100 accordingly. For testing purposes, one extension will be assigned for each user groups as shown below.

Name of Extension/Ext. Number	Priority
CEO/300	600
Manager/400	500
Staff/500	400
Student/600	200
Visitor/700	100

Table 3.1: Assignment of extensions for user-groups and their priority mappings

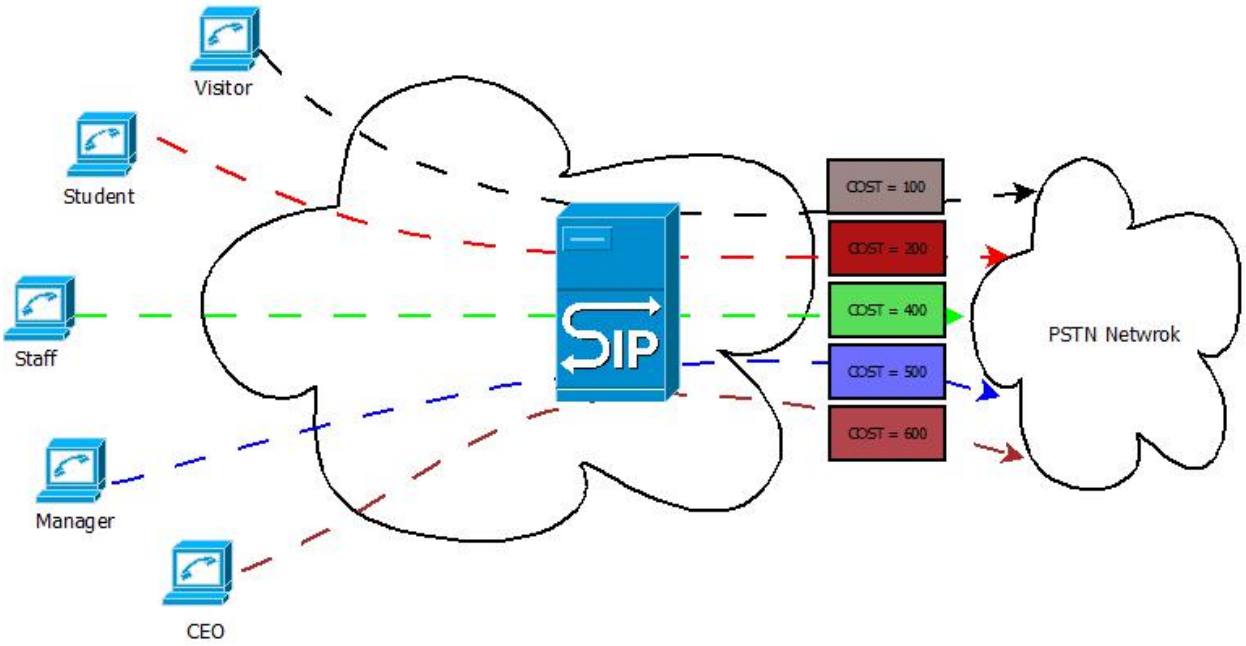


Figure 3.4: COST assignment for all user-groups via Asterisk PBX

This user groups might look like below in extensions.conf

[CEO]

include => from-internal

exten => 911,1,Goto(emergency,s,1)

[Manager]

include => from-internal

exten => 911,1,Goto(emergency,s,1)

[Student]

include => from-internal

exten => 911,1,Goto(emergency,s,1)

[Staff]

include => from-internal

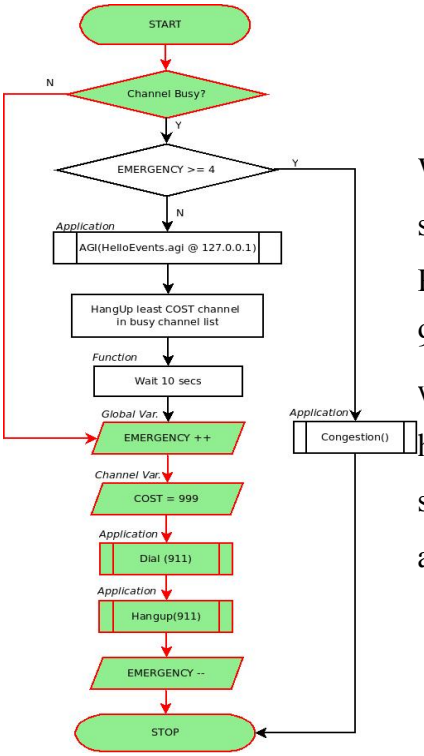
exten => 911,1,Goto(emergency,s,1)

[Visitor]
include => from-internal
exten => 911,1,Goto(emergency,s,1)

When the call is handed over to [emergency], it has to be checked whether there is any line/channel available to dial out using the trunk. Our trunk can only handle 4 simultaneous calls, thus 5 user groups were defined so the fifth caller to 911 will trigger [emergency] that will call the external AGI script. The global variable EMERGENCY keeps track of the number of 911 calls in progress through the system. This global variable is very important in this design as none of the 911 callers should be disconnected when they are online. Now let's analyze the different paths the call might take when 911 is dialled.

3.3 Case Analysis

3.3.1 Case 1: When at least one channel is available to dial 911



Whenever there is a room for at least one channel, the call should go through with priority is set to its maximum. Regardless of who initiated the 911 call, the highest priority of 999 is assigned to that channel and the EMERGENCY counter will be incremented by one. In this design, EMERGENCY can have a maximum value of 4 as the trunk can handle only 4 simultaneous outgoing calls. But how does the priority get assigned for non-911 diallers?

3.3.1.1 Priority

As seen in the previous chapter, channel variables can hold values that can remain in the system until the channels exist. Channel variables are the best approach to set priorities when they are dialled out using macro-dialout-trunk. As soon as the call is handed over to this context, the priorities should be set for that particular channel. All these channel variables will be having the same name, COST, so these values can be retrieved from the script at once and compared within it. As discussed in the previous chapter, this wont cause any conflict in call management as channel variables are unique only within that channel. So there will be 4 lines added at the very beginning of macro-dialout-trunk as shown below

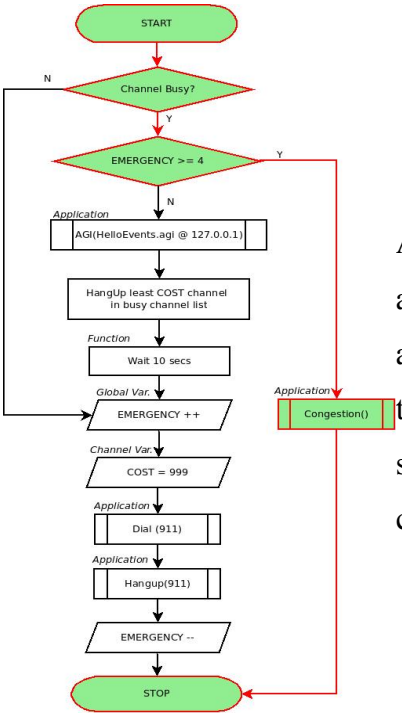
```
[macro-dialout-trunk]
exten => s,n,ExecIf($["${MACRO_CONTEXT}" = "CEO"],Set,COST=600)
exten => s,n,ExecIf($["${MACRO_CONTEXT}" = "Manager"],Set,COST=500)
exten => s,n,ExecIf($["${MACRO_CONTEXT}" = "Staff"],Set,COST=400)
exten => s,n,ExecIf($["${MACRO_CONTEXT}" = "Student"],Set,COST=200)
exten => s,n,ExecIf($["${MACRO_CONTEXT}" = "Visitor"],Set,COST=100)
```

The Asterisk defined channel variable-MACRO_CONTEXT-remembers the context that originates the call. Therefore different COST is assigned for every user groups based on the value of variable, MACRO_CONTEXT. The ExecIf() application is similar to Do-If statement in most of the programming languages. This application will be having three components as its arguments.

ExecIf(expression,application,arguments)

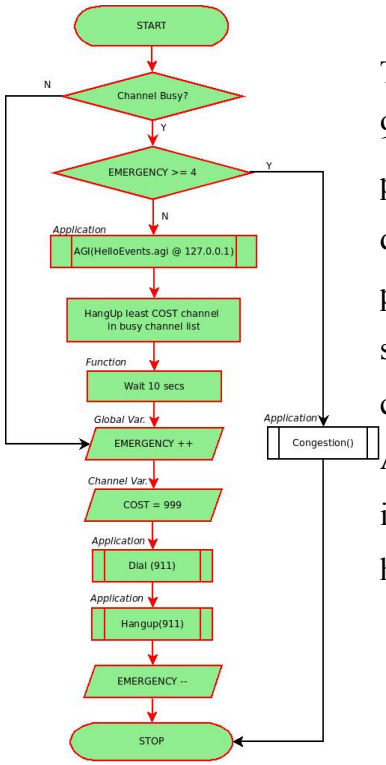
When the expression is true, the application will be called with necessary arguments passed to it. In this case, whenever the string MACRO_CONTEXT is having the value of string CEO, the application Set() will be called with COST=600 as its argument. The value 600 will be assigned as of type string; not integer. Likewise, the priorities for incoming calls from PSTN could be also configured under the context [macro-user-callerid] for every inbound routes.

3.3.2 Case 2: When no channel is available and EMERGENCY >=4



At this point, the system is fully utilized with 911 calls only. So any caller who tries to connect 911 should be denied permission and neither any lines nor resources should be freed regardless of the group that attempt to dial 911. Instead, an IVR mentioning the state of the system or a congestion tone will be sent to the 911 caller.

3.3.3 Case 3: When no channel is available and EMERGENCY < 4



This block is reached when trunk is completely occupied with non-911 calls or the system is having 911 calls of less than four. At this point, the 911 call originator should be able to get the resources to dial out 911 by hanging up one of the non-911 call based on the priority algorithm. Whenever this scenario occurs, the external script should be initiated with the inputs of names of the busy channels and its channel variable “COST”. During this time, Asterisk initiates an AGI call to a specific location where the script is idling for incoming AGI connections. This AGI script will be having the codes to

- Authenticate with Asterisk server
- Send the ACTIONs and get the EVENTs
- Get the busy channel list along with the channel variable-COST
- Compare the channels based on the priority
- Hang up the channel that is having the least priority in the stack

After disconnecting the channel, the script will be stopped and the process will jump to a certain line of [emergency] which will again try to dial 911.

3.4 Which AGI to choose?

AGI has many variants and they have their own method to communicate with Asterisk server. Therefore developers have several options to choose among many available AGI variants based on needs of the application.

3.4.1 Simple AGI

The example shown in the previous chapter is a sample for Simple AGI. This type of AGI calls the scripts by using AGI() application from dial plan. Usually, this call will comprise of the name of the script to be called and the arguments for the inputs of the script.

Syntax:
AGI(command[, arg1[,arg2 [,arg3,.....]]])

In this case, all the commands will be sent at once and nothing else will be sent until AGI response is received from Asterisk.

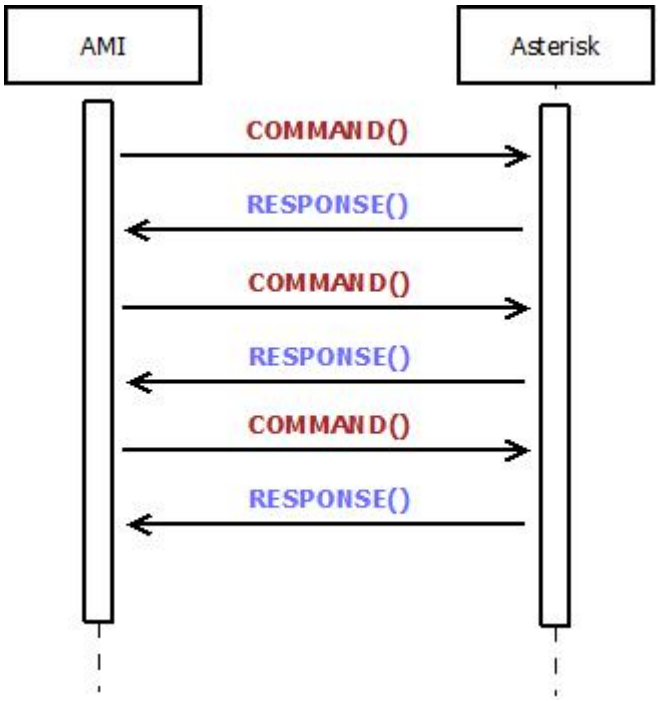


Figure 3.5: Execution Flow of Simple AGI

3.4.2 Async AGI

This type was introduced in Asterisk 1.6 and allows asynchronous AGI scripting. Unlike Simple AGI, the commands are issued by AGI's Manager Action and will be queued up in the stack for execution on that particular channel. This particular type has all three AMI message types. As soon as the Manager ACTION is initiated, there will be RESPONSE from Asterisk server about the queued up commands to be executed on that channel. Finally, Astrisk will return the EVENTs to AMI detailing the commands executed along with their corresponding Command IDs.

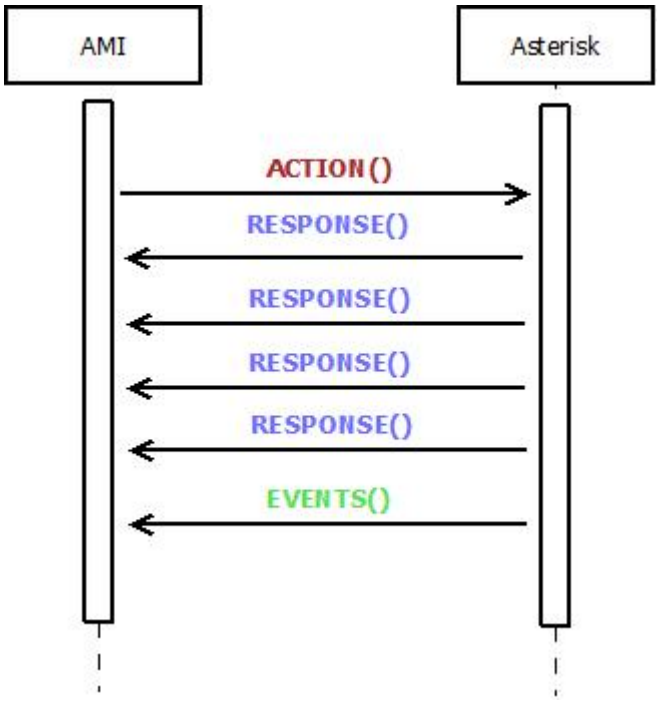


Figure 3.6: Execution Flow of Async AGI

The syntax for Async AGI will look like below

AGI(agi:async)

3.4.3 Enhanced AGI

This has the same features of Simple AGI with additional simple read-only channel audio stream. This is achieved by the fourth file descriptor defined in EAGI. First 3 file descriptors numbered through 0 to 2 are STDIN, STDOUT and STDERR respectively. The fourth channel has file descriptor ID of 3 that is used to carry linear pulse code modulated audio streams from Asterisk to AMI only. So this enables to read IVRs, play audio, read DTMF digits on the incoming channel from Asterisk to perform some action on the script.

The syntax looks like
EAGI(command[, arg1[,arg2 [,arg3,.....]]])

3.4.4 Fast AGI

As of Simple AGI, Fast AGI sends all commands in a single line and waits until it gets the response from the server before sending the send batch of commands. But, unlike other variances of AGI discussed above, Fast AGI uses TCP as the medium to transfer data between Asterisk and AMI. One big benefit in this option is that every time the commands is sent between Asterisk and AMI, standard input(STDIN) and output pipes(STDOUT) are need not to be established and torn down. Instead, the TCP session will take care of transmission between the entities and will be disconnected when all communication is done. This saves number of processes to be handled by the server and allows to use the resources efficiently. The AGI() application will be called with the logical destination address or IP address and the name of the script as the arguments.

Eg: exten => 100,1,AGI(agi://127.0.0.1/hello.agi)

In this case, DefaultAGIServer class in the Asterisk-Java library establishes the Fast AGI server and waits for incoming connections from Asterisk unlike in Simple-AGI where Asterisk's own Manager API facilitates the AGI script to communicate with Asterisk server.

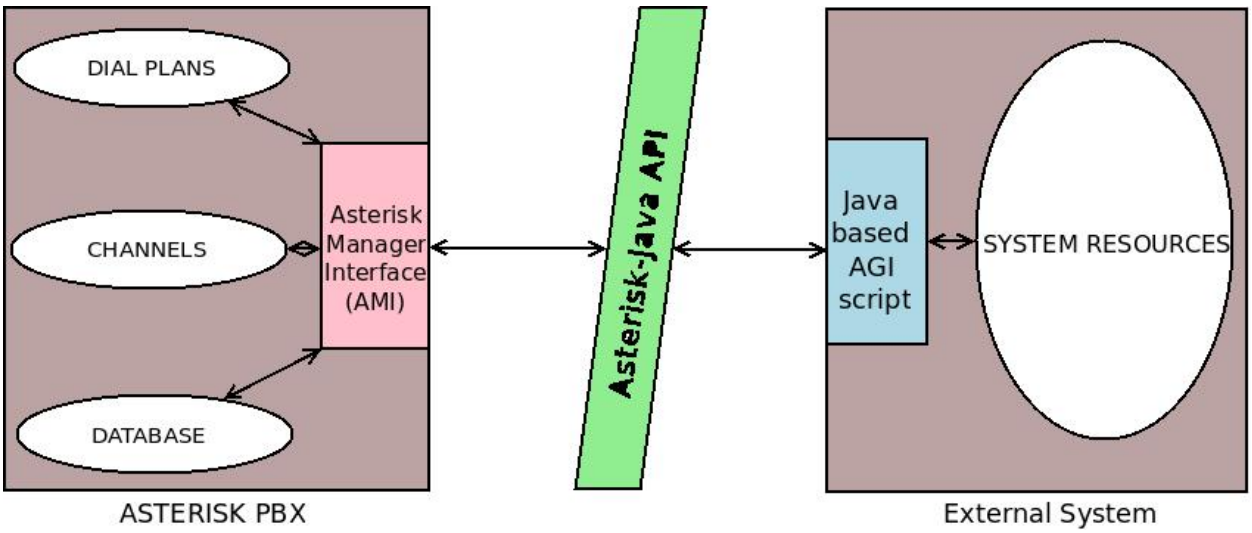


Figure 3.7: Communication between Fast AGI and Asterisk through AJ

3.5 Asterisk-Java API

Asterisk-Java currently supports Fast AGI implementation. Hence, it was decided to use Fast AGI as the protocol for this implementation with the Fast AGI server running in the same Asterisk server. As discussed so far, if an external AGI script needs to communicate with Asterisk's Manager API, there should be a mechanism that provides the channels, environments and necessary interfaces to make this communication successful. Asterisk-Java provides this platform and it carries two different packages to talk with AGI and Manager API. Asterisk-Java's Fast AGI package -org.asteriskjava.fastagi-provides necessary interfaces to deal with AGI scripts in the form of commands and replies. Fast AGI package consists of three important interfaces: AGIServer, AGIScript and MappingStrategy. Interface AGIServer provides an important class,DefaultAGIServer,in Asterisk-Java's perspective. When this class is implemented, AGIServer will wait for incoming AGI requests from Asterisk servers based on the incoming AGI requests. Interface MappingStrategy helps interface AGIServer to choose the relevent processor so the commands and replies can be correctly interpreted. Finally, interface AGIScript is responsible to process the AGI requests from Asterisk and send the AGI commands back through the established AGI channel. Classes AGIRequest, AGICommand and AGIChannel are responsible for this respectively.

When it comes to Fast AGI, Asterisk-Java's Manager API provides the interfaces to communicate between the script and Asterisk instance. The ManagerConnection, ManagerEventHandler and AsteriskManager are some of the important interfaces that provided the necessary classes to talk between these two entities. Interface ManagerConnection initiates the conversation between Asterisk and script by means of TCP/IP on the default port 5038. After establishing this connection and authentication, script may want to fire some actions towards Asterisk. ManagerAction implements this by sending actions one by one. As mentioned in previous chapter, Asterisk sends a message type called "events" to notify the applications about the changes happening in Asterisk server itself. These types of messages are very important in our case as we need to monitor the environment for the busy channels. This fuctionality is achieved by ManagerEventHandler interface.

So from above paragraphs, it is clear that this design will not only have a pure AGI solution but will also have Manager API implemented as well. In other words, Fast AGI and Asterisk Manager API will be blended in this design that will bring an important Asterisk application to this world. Therefore, in this design, application codes consist of two different tasks to be done before disconnecting the proper channel to release resources to 911 calls. Those are

1. Implementing AJ's Manager API to establish connection, fire Actions, receive Events, manage Responses and disconnect from Asterisk server.
2. Comparison of data based on the Events received from Asterisk server.

An AGI execution diagram was developed using all the inputs and outputs. Based on this detailed execution flow diagram, the emergency call handling application was developed. The execution diagram and Source code can be seen in the following pages. Source code was successfully built and executed using Java compiler. On important thing to notice in this design is that the delay period of 5000ms. This is the time period this application will wait till it gets the events from Asterisk server. For a small/medium Asterisk system 5000ms is adequate enough to get the events from server. But depending upon the configured and concurrent users of the system, this parameter might vary and could be adjusted appropriately during any time before the implementation.

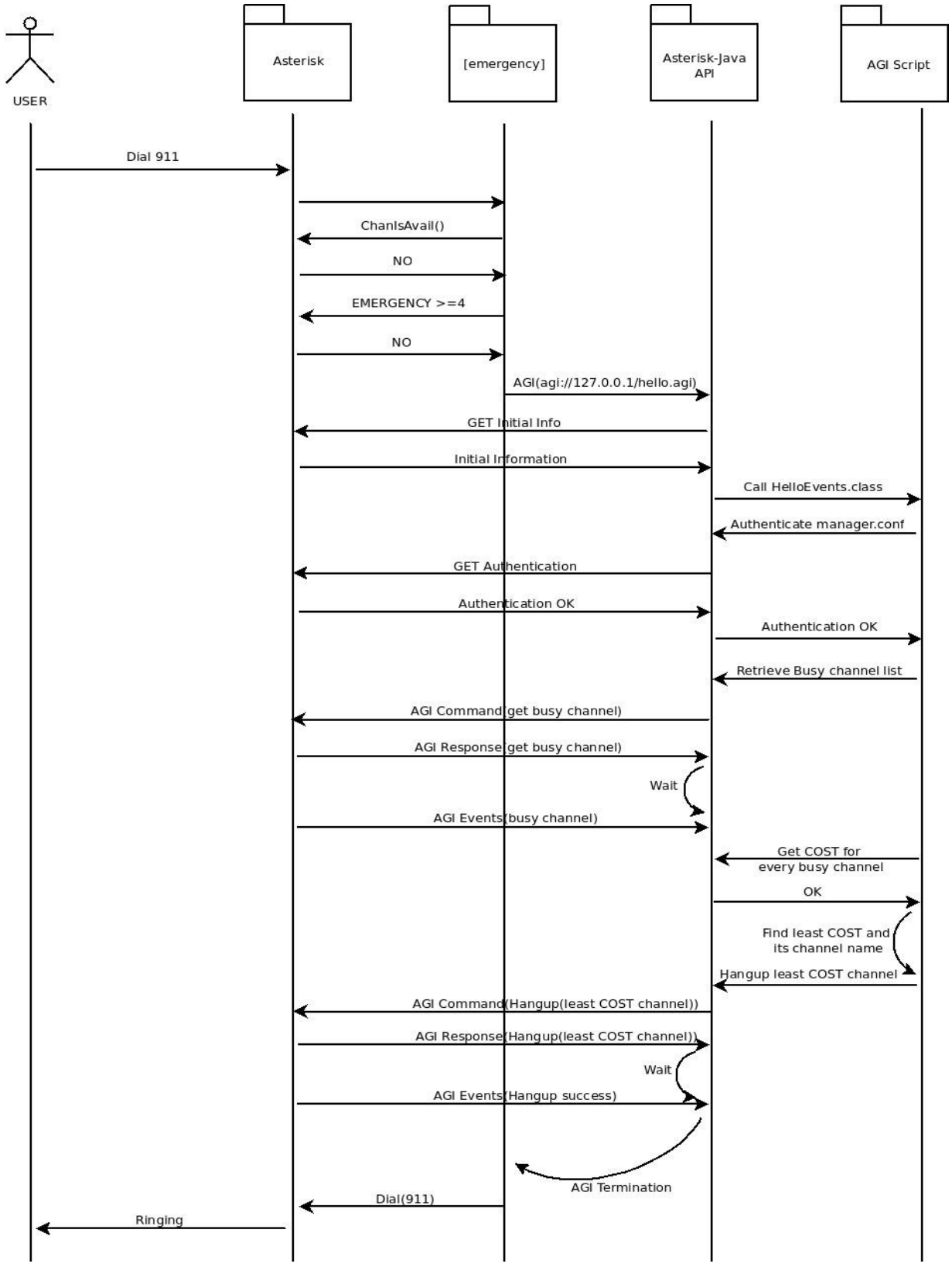


Figure 3.8: AGI Execution Flow of this Design

3.6 Source Code

```
import java.io.IOException;

import org.asteriskjava.manager.AuthenticationFailedException;

import org.asteriskjava.manager.ManagerConnection;

import org.asteriskjava.manager.ManagerConnectionFactory;

import org.asteriskjava.manager.ManagerEventListener;

import org.asteriskjava.manager.TimeoutException;

import org.asteriskjava.manager.action.HangupAction;

import org.asteriskjava.manager.action.StatusAction;

import org.asteriskjava.manager.event.ManagerEvent;

import org.asteriskjava.manager.event.StatusEvent;

import org.asteriskjava.fastagi.AgiChannel;

import org.asteriskjava.fastagi.AgiException;

import org.asteriskjava.fastagi.AgiRequest;

import org.asteriskjava.fastagi.BaseAgiScript;

import java.util.Map;



public class HelloEvents extends BaseAgiScript implements
ManagerEventListener

{

    private ManagerConnection eventGetters; // for get events

    //private ManagerConnection disconnectEvent; //for disconnect event



    String SIPChannels[]=new String[4];
```

```
    int[] variable=new int[4];
    int i=0;
    int minIndex;

    public HelloEvents() throws IOException
    {
        ManagerConnectionFactory factory = new ManagerConnectionFactory(
                "localhost", "admin", "elastix456");

        this.eventGetters = factory.createManagerConnection();
        //this.disconnectEvent = factory.createManagerConnection();

    }

    public void service(AgiRequest request, AgiChannel channel)
    throws AgiException
{
try {
        run();
} catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
} catch (AuthenticationFailedException e) {
    // TODO Auto-generated catch block
```

```
// TODO Auto-generated catch block
e.printStackTrace();
} catch (TimeoutException e) {
// TODO Auto-generated catch block
e.printStackTrace();
} catch (InterruptedException e) {
// TODO Auto-generated catch block
e.printStackTrace();
}
}
```

```
public void run() throws IOException, AuthenticationFailedException,
TimeoutException, InterruptedException
{

StatusAction statusActiona=new StatusAction();
statusActiona.setVariables("COST");

//register for events
eventGetters.addEventListener(this);

//connect to Asterisk and log in
eventGetters.login();
```

```
//request channel state

eventGetters.sendAction(statusActiona);

Thread.sleep(5000);

getMinValue(variable);

HangupAction ha = new HangupAction();

ha.setChannel(SIPChannels[minIndex]);

eventGetters.sendAction(ha);

//wait 5 seconds for events to come in

Thread.sleep(5000);


//and finally log off and disconnect-eventGetters

i=0;

minIndex=0;

eventGetters.logoff();


//connect to Asterisk and log in

//disconnectEvent.login();


//disconnect particular channel

//disconnectEvent.sendAction(ha);


//and finally log off and disconnect-disconnectEvent
```

```
//disconnectEvent.logoff();



}
      public int getMinValue(int[] numbers){
        int minValue = numbers[0];
        for(int J=1;J<i;J++){
        if(numbers[J] < minValue){
            minValue = numbers[J];
            minIndex=J;
          }
        }
        return minIndex;
      }




    public void onManagerEvent(ManagerEvent event)
    {


        if(event instanceof StatusEvent)
        {
                StatusEvent statusEvent = (StatusEvent) event;
                if(statusEvent.getChannel().substring(0,3).equals("SIP")&&sta
tusEvent.getBridgedChannel().substring(0, 5).equals("DAHDI")){
                        SIPChannels[i]=statusEvent.getChannel();
```

```
                        System.out.println("Channel " + SIPChannels[i]);
                        variable[i]=Integer.parseInt(statusEvent.getVariables
().get("COST"));
                        System.out.println("Variable: " + variable[i++]);
                }


        }
        else
        {
                // just print received events, or maybe not
                // System.out.println(event);
        }
    }
}
```

4. Implementation and Testing

When designing real systems, it should be made sure that the design fully satisfies the expected outcomes. Testing process allows designers to catch the glitches and bugs inside the codes thereby improving the quality of the product. Likewise, in this design, the developed script has to go through some testing procedures to check whether they meet the expected outcomes and standards. Below diagram shows the network diagram for this testing setup.

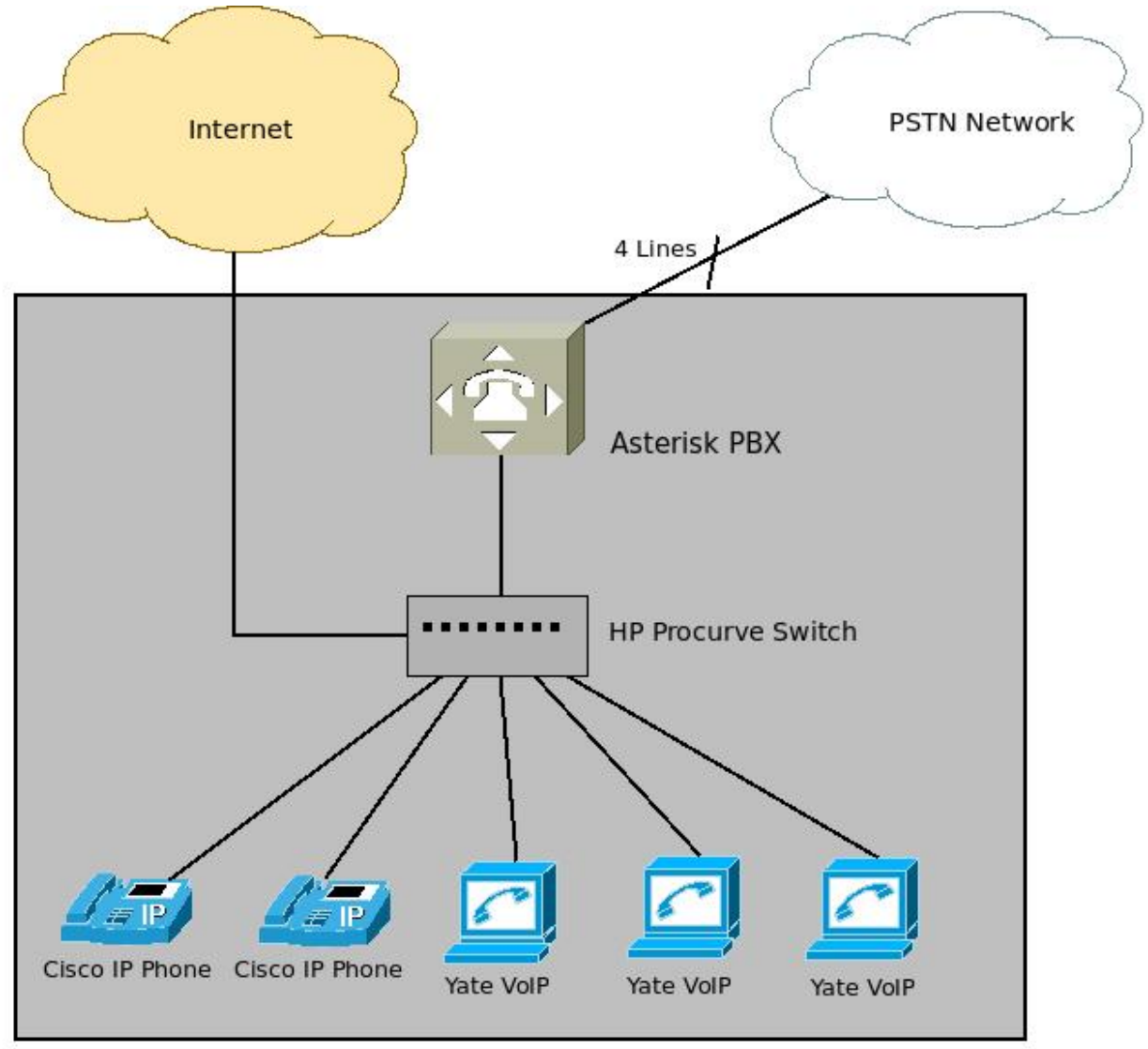


Figure 4.1: Physical Set-up for Testing

4.1 Implementation of Testing Environment

4.1.1 Hardware Installation

Couple of Cisco 7960 phones were connected to the HP ProCurve series switches under VLAN 10 and SIP accounts were created in both Asterisk and Cisco phones. It was monitored in the Asterisk CLI that the two IP phones were registered with the system and ready to use. Another soft SIP Client, Yate VoIP, was installed in one of the notebooks and three SIP accounts were registered. It was registered to the Asterisk systems as well.

```
elastix*CLI>
    -- Registered SIP '300' at 206.75.37.106 port 5060
       > Saved useragent "CISCO 7960" for peer 300
    -- Registered SIP '400' at 206.75.37.106 port 5060
       > Saved useragent "CISCO 7960" for peer 400
    -- Registered SIP '500' at 206.75.37.106 port 5060
       > Saved useragent "YATE/2.2.0" for peer 500
    -- Registered SIP '600' at 206.75.37.106 port 5060
       > Saved useragent "YATE/2.2.0" for peer 600
    -- Registered SIP '700' at 206.75.37.106 port 5060
       > Saved useragent "YATE/2.2.0" for peer 700
elastix*CLI>
elastix*CLI>
elastix*CLI>
```

Figure 4.2: SIP registration of 2 Cisco IP Phones and 3 Soft Phones

4.1.2 Software Installation

The developed script contains of native Java Language and some special packages developed by Asterisk-Java. Thus, latest stable version of Asterisk-Java 1.0.0.M3 was downloaded and installed in the same box as the Asterisk server. Installing Asterisk-Java is not a tedious task, but after the Elastix installation it was noticed that there was no Java compiler provided with Cent OS. So, Java was installed to Cent OS in order to proceed with the rest of the stages.

4.1.3 Installing the Script

The script, HelloEvents.java, was copied inside /etc/asterisk/ before compilation. But in the [emergency], AGI is trying to invoke the file hello.agi. So, obviously, there should be a mapping between the script name, hello.agi and the AGI script-HelloEvents.java-that we created. In Unix this is done by the properties file called fastagi-mapping. In this design, this file will have a single line as shown below.

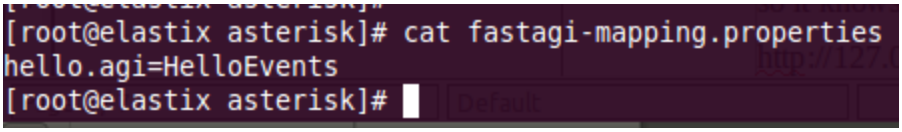


Figure 4.3: Content of fastagi-mapping.properties file

When the DefaultAGIServer is started, file fastagi-mapping.properties must be on its class path so it knows which script needs to be invoked when it gets AGI call from Asterisk with argument http://127.0.0.1/hello.agi. There should be these four files inside the /etc/asterisk/ before starting the DefaultAgiServer.

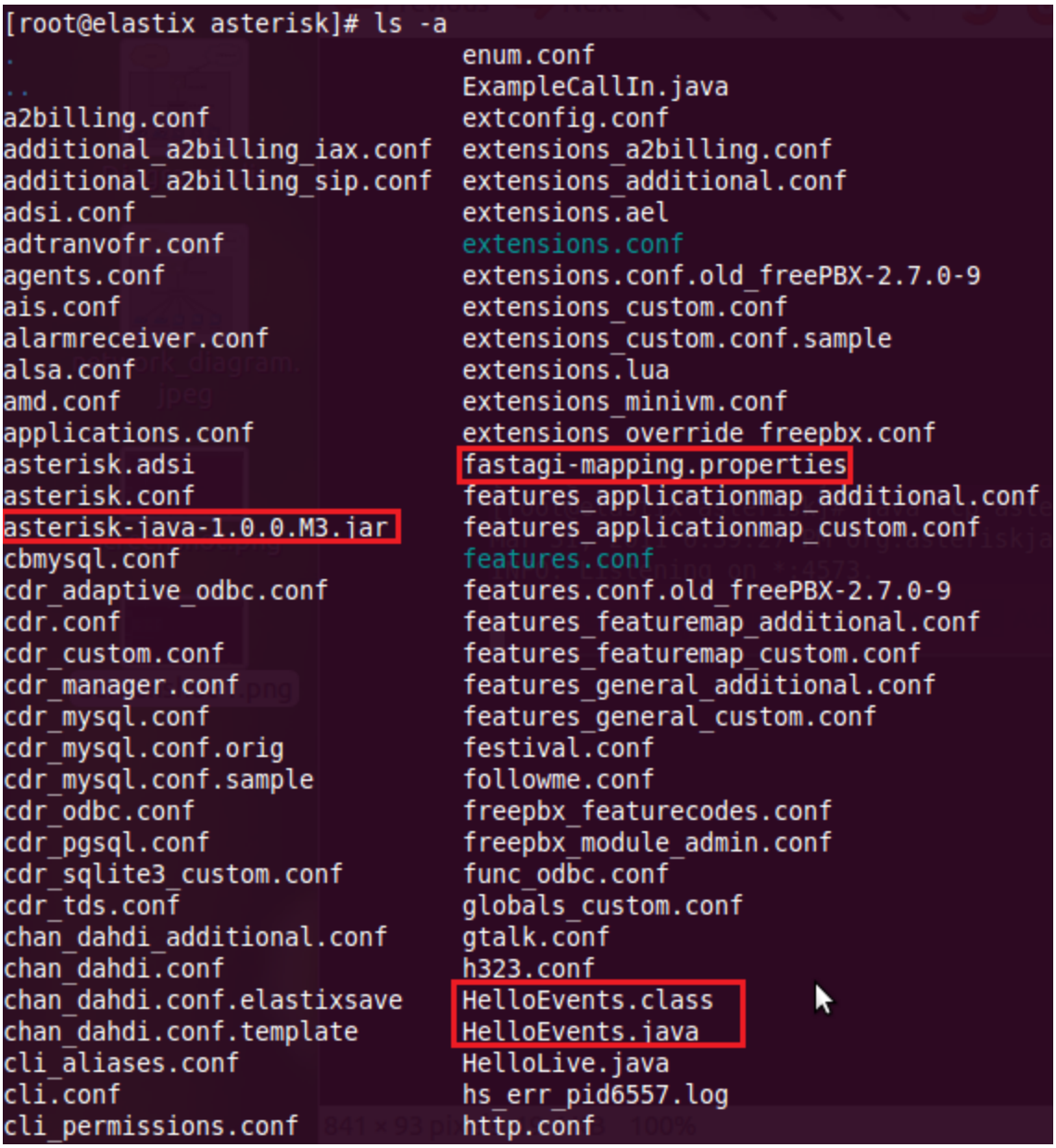


Figure 4.4:: Most important files of /etc/asterisk before starting the DefaultAGIServer

Mapping file might not be necessary inside /etc/asterisk/ during the compilation of HelloEvents.java but must reside inside when starting the DefaultAGIServer. If not, when the Asterisk ask to invoke hello.agi from Manager API, it will throw a message on the screen that there is no fastagi-mapping found. After making sure all four files are there,the DefaultAGIServer can be started by typing inside /etc/asterisk

java -cp asterisk-java-1.0.0.M3.jar:. org.asteriskjava.fastagi.DefaultAGIServer

Now it can be seen that the server is initiated and listening for incoming connections via port 4573.

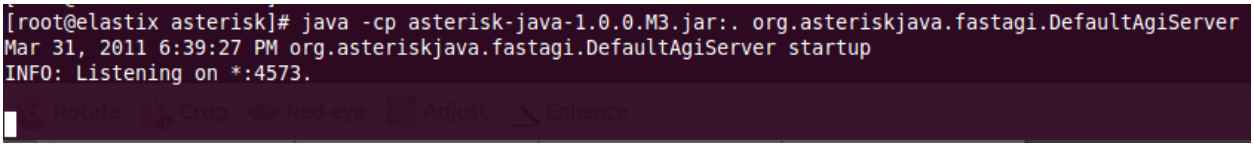


Figure 4.5: DefaultAGIServer is Listening for Incoming Connections

4.2 Testing

Some test cases were developed and tested to make sure that the script works according to the algorithm. These test cases closely match with the cases that were analyzed in section 3.3 of Chapter 3.

4.2.1 Test Case 1: All lines dialling non-911 and check the busy channel list and the COST assigned to them.

In this case, all lines were checked for proper COST assignment and the busy channel list were checked. As the script wont get activated during this process, all the status of the COST and the busy channel list were taken using Asterisk CLI commands. The Figures below provide the proof for this.

Figure 4.6: COST Assignment for CEO

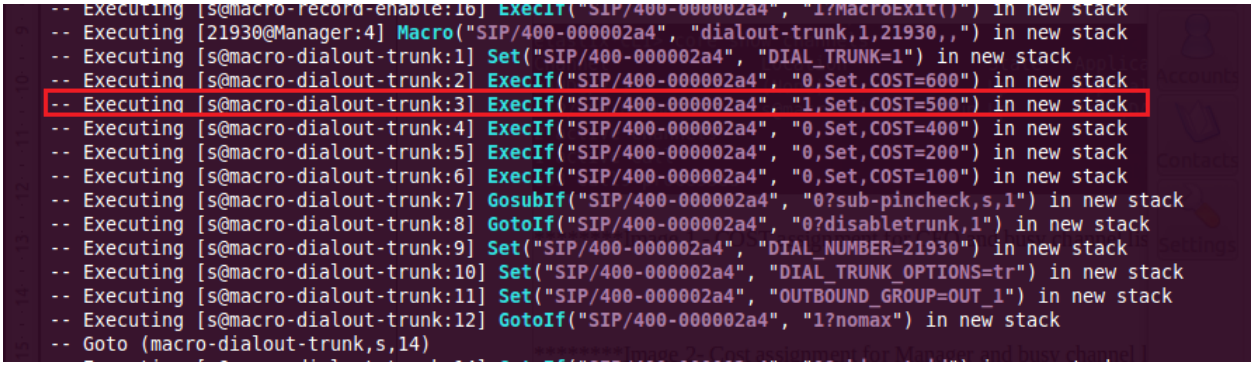


Figure 4.7: COST Assignment for Manager

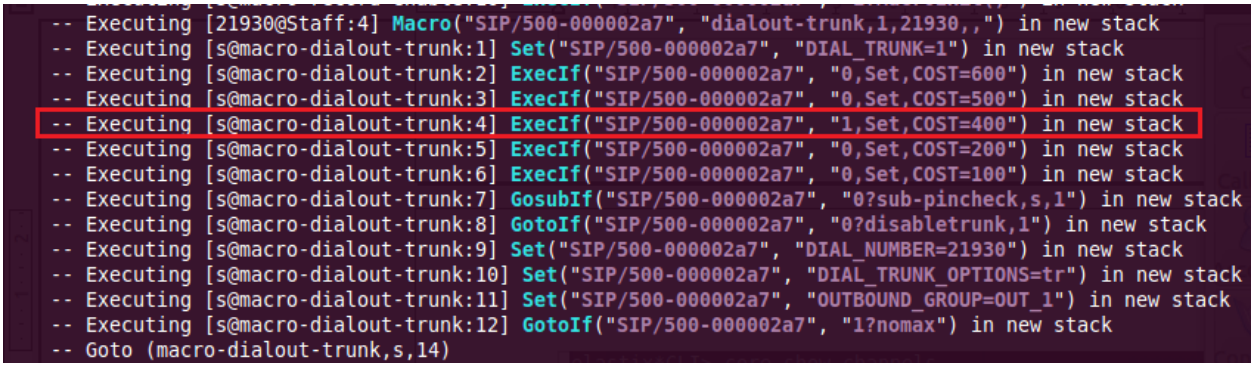


Figure 4.8: COST Assignment for Staff

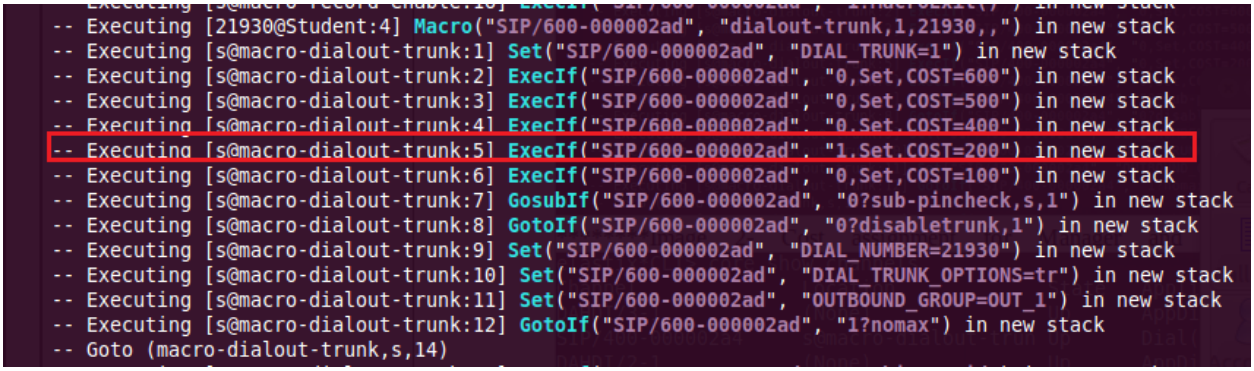


Figure 4.9: COST Assignment for Student

4.2.2 Test Case 2: Trunk is busy with non-911 calls and another user wants to dial 911

During this case, it can be seen that the script gets called by Asterisk and Asterisk API lists down the busy channel list and the COST assigned to them in one screen.

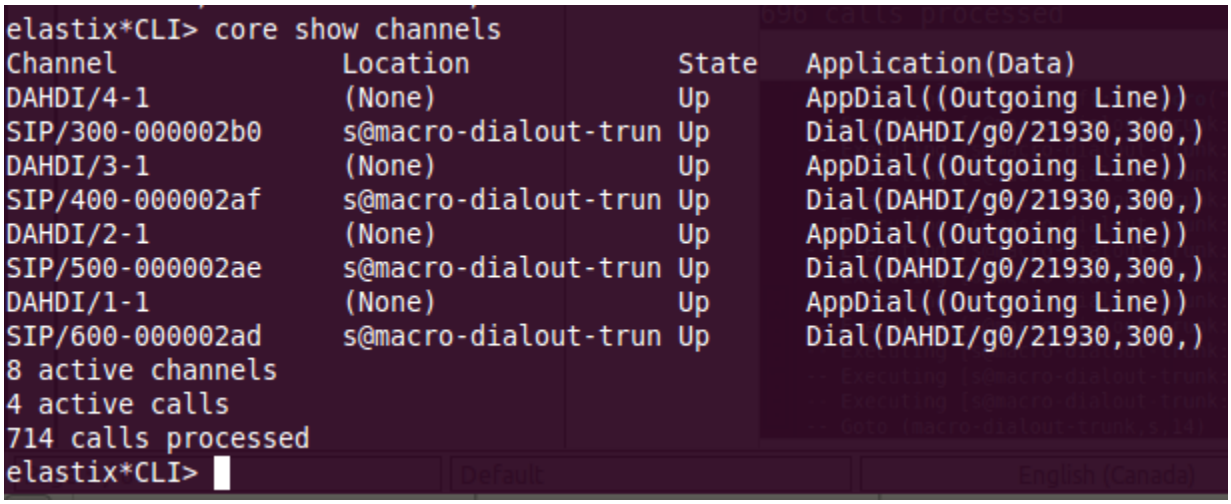


Figure 4.10: Busy Channel List of Asterisk: CEO, Manager, Staff and Student

As seen above, CEO, Manager, Staff and Student are busy in the trunk. When a new caller from Visitor user group initiates the call, Student must be removed from the system to release the resources to 911 call. Notice that highest COST of 999 will be assigned to this 911 call before dialling out. These can be seen from the Figure in the next page.

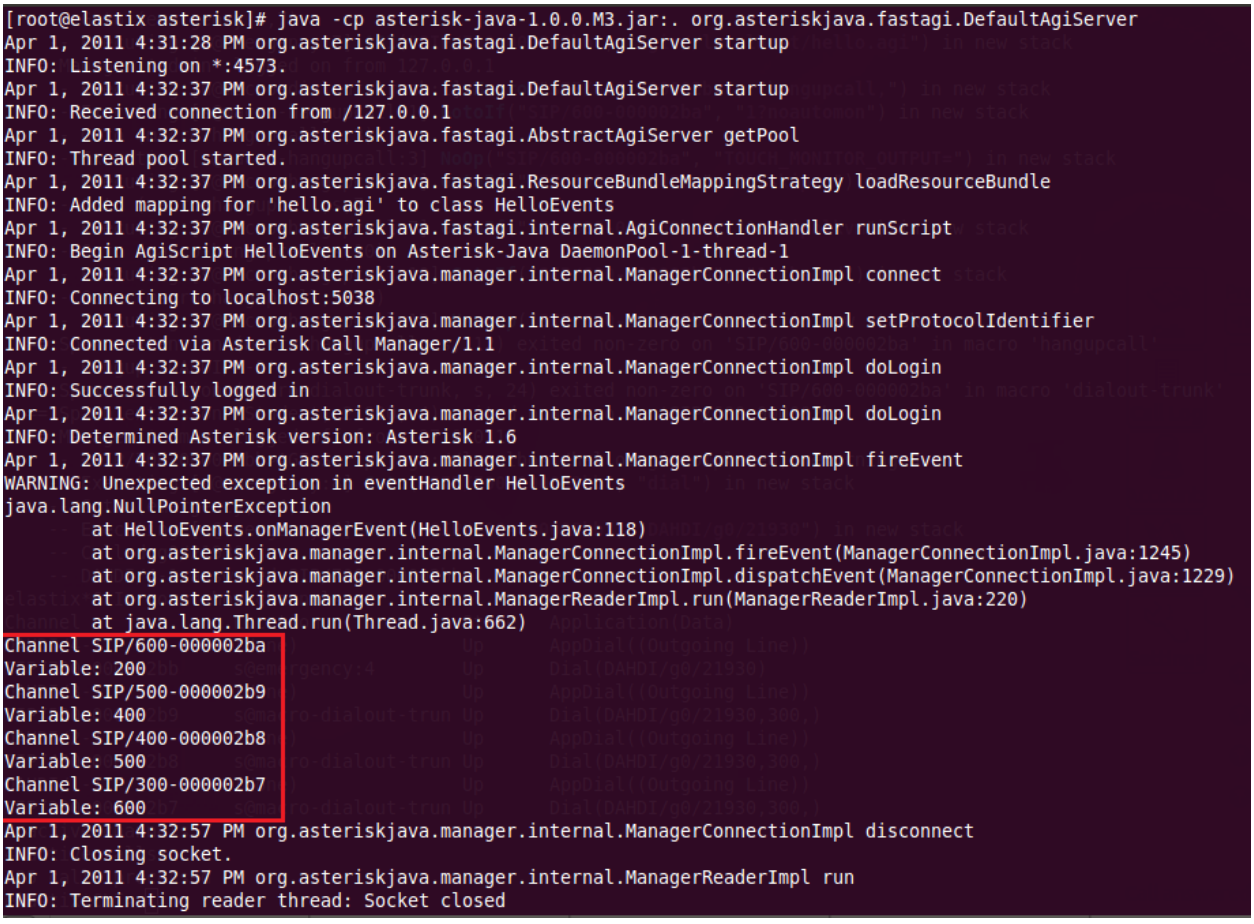


Figure 4.11: AGI Script shows that all 4 channels are busy and values of COST assigned to those channels

So According to the Figure above, associated DAHDI channel for SIP/600-000002ba should be disconnected and the Visitor(700) should be able to dial 911 using that freed channel.

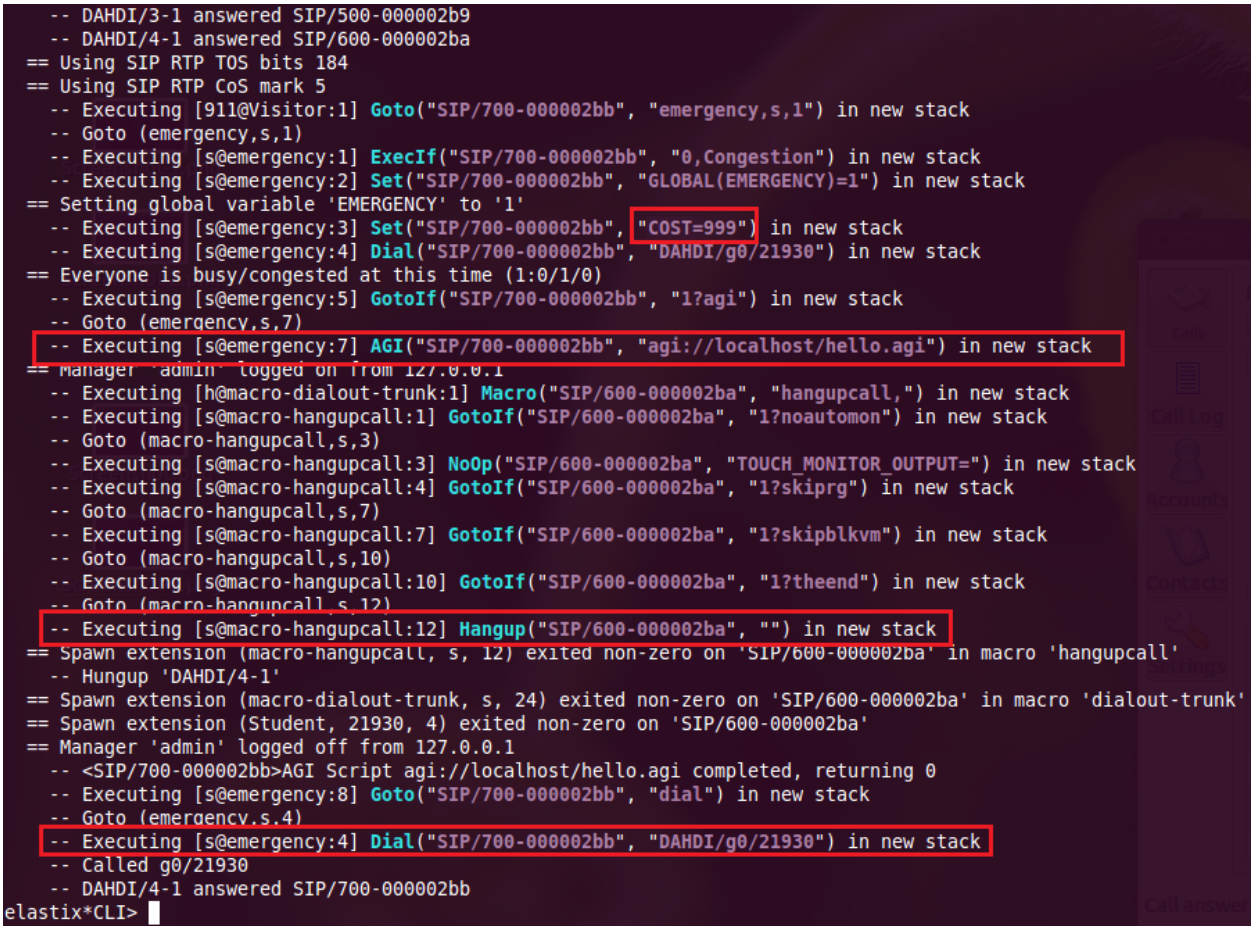


Figure 4.12: SIP-600 is freed to make room SIP-700 to dial 911

Also, it can be seen that SIP-700 is assigned the COST value of 999. So now the busy channel list will look like below.

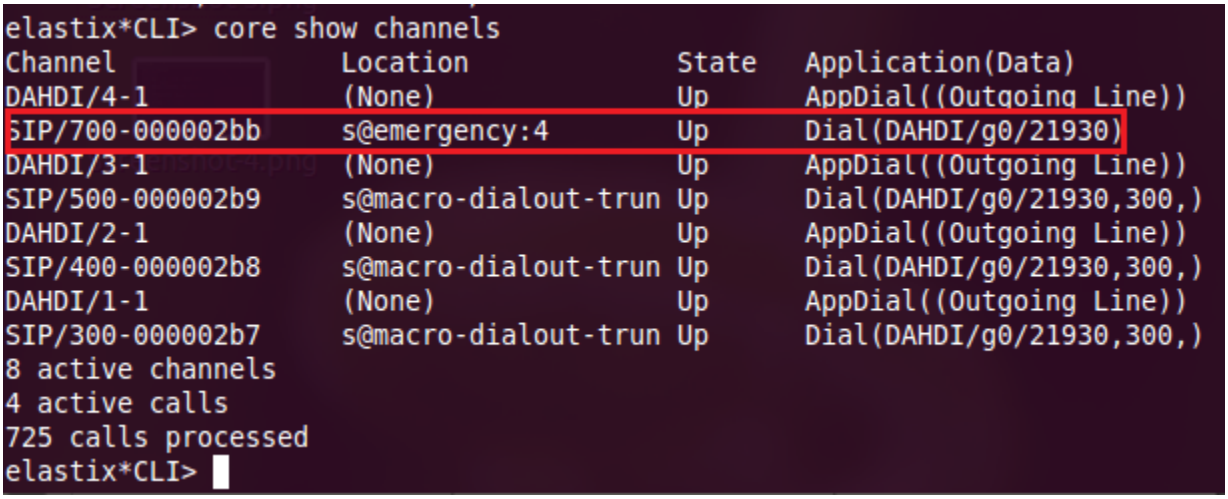


Figure 4.13: Busy Channel List of Asterisk: CEO, Manager, Staff and Visitor

It should be noted here that [emergency] dials Dial(DAHDI/g0/21930) instead of Dial(DAHDI/g0/911) to prevent dialling real 911 during testing period.

4.2.3 Test Case 3: Trunk is busy with 911 and another caller wants to dial 911.

In this scenario, when all channels are busy with 911 calls, any caller who dials any number including 911 will be rejected by the system an IVR message asking them to dial back after short while.

Figure 4.14: All channels are busy with Emergency calls and CEO is denied access to another Emergency call.

Likewise, all test cases were recursively tested and matched with expected outcomes. This indicates that the designed AGI script works according to the plan and can be implemented in any live environment to dial PSAP to get assistance during emergency situations.

5 Conclusions

Having tested all the test cases with success, this design provides a complete solution to grab a telephone line through an Asterisk server during emergency situations. This design uncovers the usefulness of integrating external scripts with Asterisk along with the Asterisk-Java framework. Although a complete solution could have been developed with pure Asterisk scripting, it might not be efficient as this for several reasons. One advantage of this design is that the external scripts provide a good structure to any system to which it will be embedded thereby providing scalability to the system.

Another feature of this design is that the number of concurrent dial-outs can be configured before the script is compiled. This provides the scalability to this design and increases the efficiency of the system by sharing resources evenly among the channels. As it can be seen in the final script, wait time of 5000 ms is set-up till all the events arrive from Asterisk server to the AGI server. This is just an arbitrary number and it is big enough to our testing scenario. But this parameter might have to be increased depending upon the number of concurrent channels in-use at any given time. Usually, 5s are more than enough to get all the events from an average Asterisk server, but this has to be found precisely to do Asterisk optimization.

From a developer's point of view, I learned and revealed a lot about Asterisk and Asterisk-Java framework and now I am in a position to write scripts for different scenarios. Currently, I am involved in the developer and user forums of Asterisk-Java and actively participating in discussions.

Bibliography

[1] Asterisk : The Future Telephony (2nd Edition) by Jim Van Meggelen, Leif Madsen and Jared Smith

[2] Elastix Without Tears(PDF Version) by Ben Sharif

[3] ProCurve Switch 2500-Configuration Guide HP2524

[4] Asterisk Gateway Interface 1.4 and 1.6 Programming by Nir Simionovich

[5] VoIP Wiki – www.voip-info.org

[6] Asterisk-Java API Docs - http://asterisk-java.org/1.0.0.M3/apidocs/

[7] Asterisk-Java Design and Tutorials - http://asterisk-java.org/1.0.0.M3/

[8] Asterisk-Java Users' Forum - https://lists.sourceforge.net/lists/listinfo/asterisk-java-users