

## Objective

To develop an application for the Asterisk voice platform working in conjunction with a network management protocol to provide a system for alert, interaction, and logging of desired events.

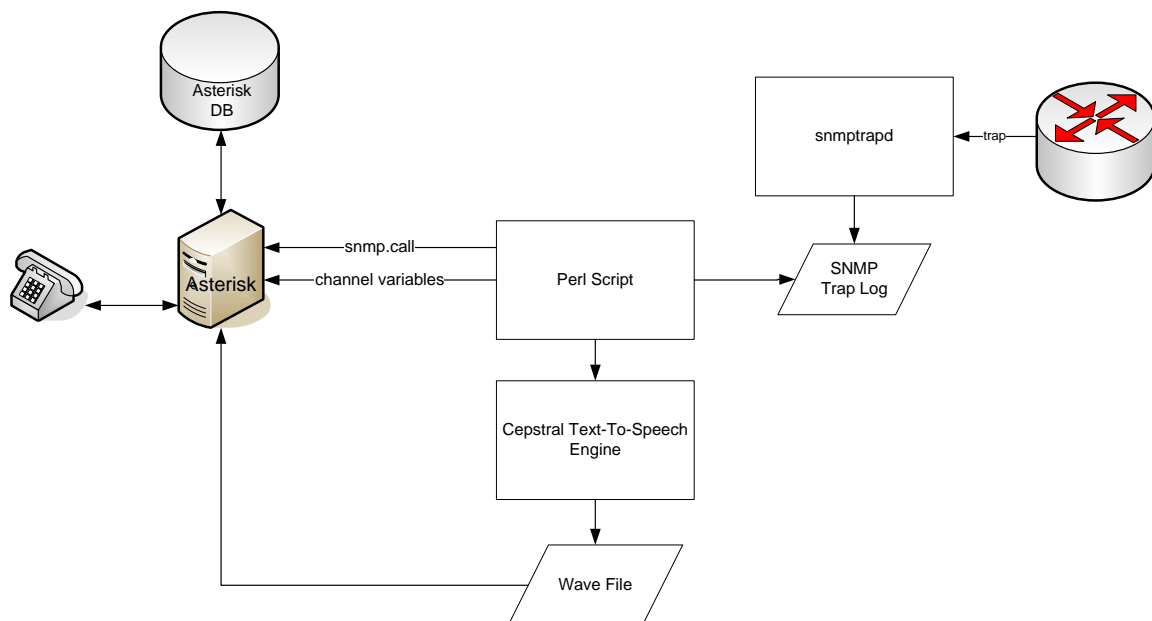
## Overview

Several kinds of technology must be tied together that do not normally interact with each other. All of these pieces are independent of the others and the fabric of these technologies will be glued together with the use of Perl, a simple yet powerful scripting language.

This paper will explore the development process required to build the management system, with each bond of the process being broken down and explained. This approach will allow the reader to understand the project as a whole by following the progress of the development from beginning to end. This will then give the reader an appreciation for why certain design decisions were made. We will then investigate some ideas which the authors have for the system if it were to continue being developed.

## Process

Our project has five main sections: SNMP traps and logging, the parsing of these logs, the creation of the text to speech wave file, the creation of call files and channel variables & the dialplan creation and Asterisk database. Figure 1-1 below illustrates how these pieces correlate to one another.



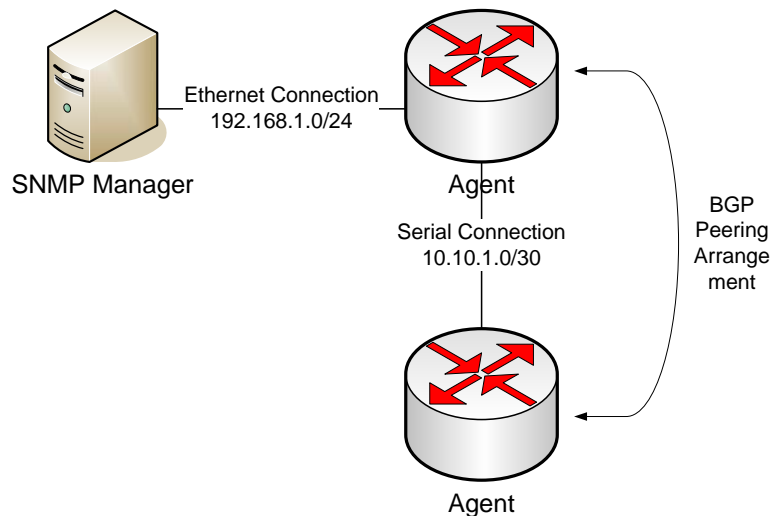
**Figure 1-1: Management System Logical Process**

## SNMP Traps and Logging

SNMP traps are definitions on the router that when a certain event happens to tell an SNMP trap daemon. The SNMP trap daemon then logs these events to a file with information about the trap. We will explore what the SNMP protocol is, what an SNMP trap is and what this information looks like when it is logged by the Linux SNMP trap daemon. These log files are where the information will be stored and what we will parse with our Perl script.

### What is SNMP?

Developed in the late 80's as a means to manage network devices, this agent/manager scheme requires little processing power and complexity on part of the agents (routers, switches). This is where the term "simple" is derived from. Most of the processing resides on the manager software used to interact with these devices.



**Figure 2-1: SNMP Manager/Agent Relationship**

This protocol has 5 basic operational conventions.

- Set
- Get
- GetNext
- GetResponse
- Trap

### What are SNMP Traps?

For the purpose of explanation we will focus for now on the trap operation, as it will allow us to explore the mechanics of SNMP. A trap is a response from an agent generated by the crossing of some threshold set for the agent to monitor. In the case below, the agent (in this case a switch) was configured to generate and deliver a message to a desired host in the case that the status of any of its links changed. (i.e. up, down, administratively up, administratively down)

This protocol was designed to be connectionless in nature, that is, the manager should continue to operate even if agent(s) may fail. As well, the agent should run independently of the manager. The point is that the agent should be able to report as little information as possible to the manager, without the need to maintain state between them. With this in mind, the SNMP information seemed suited to use UDP functionality. However, it can be implemented using TCP or mapped to an ATM cell, or direct Ethernet MAC mapping as well.

By default, all SNMP operations will use UDP as transport for delivery of its messages. Below is a captured packet generated by the agent and issued to the manager.

```

User Datagram Protocol, Src Port: 53208 (53208), Dst Port: snmptrap (162)
  Source port: 53208 (53208)
  Destination port: snmptrap (162)
  Length: 220
  Checksum: 0x74cd (correct)
Simple Network Management Protocol
  Version: 2c (1)
  Community: isystech
  PDU type: TRAP-V2 (7)
  Request Id: 0x00000002
  Error Status: NO ERROR (0)
  Error Index: 0
  Object identifier 1: 1.3.6.1.2.1.1.3.0 (iso.3.6.1.2.1.1.3.0)
  Value: INTEGER: 180577227
  Object identifier 2: 1.3.6.1.6.3.1.1.4.1.0 (iso.3.6.1.6.3.1.1.4.1.0)
  Value: OID: iso.3.6.1.6.3.1.1.5.3
  Object identifier 3: 1.3.6.1.2.1.2.2.1.1.8 (iso.3.6.1.2.1.2.2.1.1.8)
  Value: INTEGER: 8
  Object identifier 4: 1.3.6.1.2.1.2.2.1.7.8 (iso.3.6.1.2.1.2.2.1.7.8)
  Value: INTEGER: 2
  Object identifier 5: 1.3.6.1.2.1.2.2.1.8.8 (iso.3.6.1.2.1.2.2.1.8.8)
  Value: INTEGER: 2
  Object identifier 6: 1.3.6.1.2.1.2.2.1.2.8 (iso.3.6.1.2.1.2.2.1.2.8)
  Value: STRING: "FastEthernet0/8"
  Object identifier 7: 1.3.6.1.2.1.2.2.1.3.8 (iso.3.6.1.2.1.2.2.1.3.8)
  Value: INTEGER: 6
  Object identifier 8: 1.3.6.1.4.1.9.2.2.1.1.20.8 (iso.3.6.1.4.1.9.2.2.1.1.20.8)
  Value: STRING: "administratively down"

```

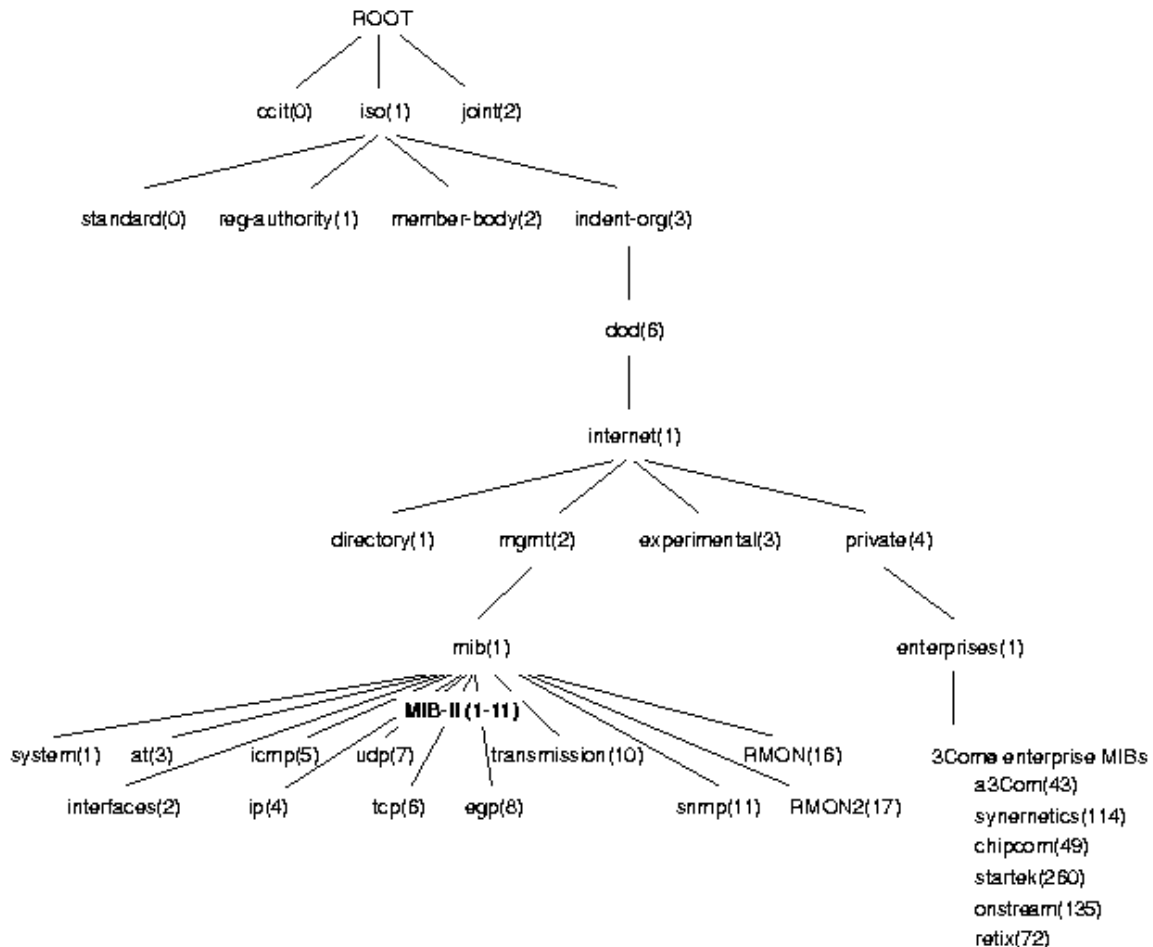
**Figure 2-2: SNMP Trap packet**

In the previous figure, it can be observed that this message is destined for port 162 UDP. By default agents and managers listen on this port for SNMP information. Trap messages will always include the version of SNMP being used for the message. This will be explained in greater detail a little later. As well it includes the community of devices to which it belongs, this can be thought of as something of a password. Finally, it will indicate the type of operation occurring in the message, in this case a trap.

After this basic information, a number of Object Identifiers and Values are carried in the message. So what are these strings of numbers? Each Object Identifier or OID is an expression indicating a relative position in a database of predefined objects that agents and managers understand. Managers have on-hand a complete database of objects and events that may be seen in a message, and agents contain some sub-set of this information. This is known as the MIB.

## MIB or Management Information Base

This database contains standardized information used to effectively manage network devices. By utilizing this universally recognized structure, manager/agent communications can be minimized to pointers indicating which object is to be negotiated and some value attached. Currently the MIB-II (extended from original) database is being used (RFC 1157).



**Figure 2-3: MIB Tree<sup>1</sup>**

In the previous diagram, the MIB tree is shown. This is a visual representation of where these OIDs are derived from. So now, some of the OIDs seen in the trap capture can be explained. The first OID in the message 1.3.6.1.2.1.1.3.0 is actually a timestamp object and is carrying with it an INTEGER value representing the time of the trap. Using an object translator provided at the Cisco website this can easily be understood.

[iso\(1\).org\(3\).dod\(6\).internet\(1\).mgmt\(2\).mib-2\(1\)](#)

|

<sup>1</sup> Image source:

<http://support.3com.com/infodeli/tools/netmgt/tncsunix/product/091500/c15snmp1.gif>

```

- -- system \(1\)
|
| -- sysDescr \(1\)
|
| -- sysObjectID \(2\)
|
| -- sysUpTime \(3\)
|
| | -- sysUpTimeInstance (0) object Details

```

**Figure 2-4: MIB Position<sup>2</sup>**

In this same way, the remaining OID values in the trap message represent different objects in the MIB, used together to generate a more comprehensive message about the nature of the trap. It was mentioned that this trap was due to some change in an interface of the agent. More specifically, this link was issued a shutdown command. Let's take a look at the last OID contained in the message; 1.3.6.1.4.1.9.2.2.1.1.20.8.

```

iso \(1\) . org \(3\) . dod \(6\) . internet \(1\) . private \(4\) . enterprises \(1\) . cisco \(9\) .
local \(2\) . interfaces \(2\)
|
| -- lifTable \(1\)
|
| -- lifEntry \(1\)

```

**Figure 2-5: MIB Position<sup>3</sup>**

The above illustrates the OID up to the Interface Entry portion of the MIB. The 20<sup>th</sup> sub-entry after ifEntry is designated to the reason for the status change of the interface. Its associated value is a STRING, and is equal to "administratively down". This is the consequence of issuing the "shutdown" command on the interface.

It is important to note that the agents are configured to use a specified version for communication. There are slight variations in output from version to version.

- V1 - This is the first, most primitive base of communication. The only security is a pairing of community and address used for authentication.
- V2 - Agents in this project were configured for this version leading to more verbose output, and the ability to apply higher security measures including MD5 hashing of data.
- V3 - The latest version offers the highest level of security and more functionality regarding remote configuration.

## SNMP in the Project

SNMP is the engine for generating and delivering information to our notification system. This gave us opportunity to use desired traps configured on agents. In this

<sup>2</sup> Image sourced from Object Translator found at <http://www.cisco.com>

<sup>3</sup> Image sourced from Object Translator found at <http://www.cisco.com>

case the agents are a pair of Cisco 2600 series routers. With this in mind, it took some consideration to decide exactly what our agents would report on. As well, our manager is a Dell box running Red Hat Linux. Because we are not running a commercial SNMP managerial suite, our manager is running a collection of SNMP daemons that give us all the functionality that may be found in commercial software. It should be mentioned that Ipswitch's WhatsUp software has messaging capabilities using SMS.

For the purposes of our project we had decided upon SNMP agents using version 2c to generate verbose output. It was our hope that with the most possible information, we may be able to form a more comprehensive notification system. We decided to configure the agents to report on link status changes, like those mentioned earlier. As well, we were interested in using BGP Finite State Machine changes as traps for notification. To see a complete listing of commands used, see the router configurations in the appendix.

## What is the Linux SNMP Trap Daemon?

To receive these traps we needed a daemon known as `snmptrapd`. This daemon receives trap information, and can format and output this information in a desired way. This daemon also initiates the Perl script used to interconnect all processes between the agents trap and the final notification placed by Asterisk. We can configure the `snmptrapd.conf` file with the following line in order to instantiate the Perl script every time the daemon receives a trap from the agent.

```
traphandle default perl /var/lib/asterisk/agi-bin/snmp-trigger-script.pl
```

## Parsing the Logged Information

Now that we know what SNMP is and how the traps are logged we need to parse this information. Before we can start writing the Perl script to do this we must ask ourselves some questions: How do we only get information about new events, what is the logged information really telling us and how do we avoid race conditions between our script and the SNMP trap daemon?

## Interpreting the Logged Information

Once we have obtained some logged information we must decide what the data is telling us. To do this we needed to create known conditions in order to activate the trap on the router. Once this happens the information would be passed to the SNMP trap daemon and logged to a file on the Linux machine. The very first trap that we triggered was an interface down by unplugging a cable from a router. Figure 3-1 below shows the output written by the SNMP trap daemon.

```
2004-12-11 13:53:49 192.168.1.101 [192.168.1.101]:
    SNMPv2-MIB::sysUpTime.0 = Timeticks: (190152) 0:31:41.52          SNMPv2-
MIB::snmpTrapOID.0 = OID: IF-MIB::linkDown          IF-MIB::ifIndex.2 = INTEGER: 2
IF-MIB::ifDescr.2 = STRING: Serial0/0          IF-MIB::ifType.2 = INTEGER:
propPointToPointSerial(22)          SNMPv2-SMI::enterprises.9.2.2.1.1.20.2 = STRING:
"down"
```

**Figure 3-1: SNMP Trap Daemon Logged Information**

As discussed previously, the SNMP MIB database contains various levels of information about devices. The trap above is the linkDown trap that is triggered when a serial interface is unplugged from a router. Some of the information above is

extraneous and not needed. What we are interested in is having something general that we can match on to determine that an interface, any interface, has either gone down or come back online. In the Perl script we look for either an IF-MIB::linkDown or IF-MIB::linkUp to determine this. Once we have matched that a link has gone down or up we need to know what interface this trap was triggered for. This is found in the IF-MIB::ifDescr.2 interface description field. Even though the IF-MIB::linkDown tells us generally that an interface has gone down, we're not sure of the reason. The SNMPv2-SMI::enterprises.9.2.2.1.1.20 tells us whether the interface was brought down physically or administratively with the value of "down" or "administratively down" string respectively.

### Obtaining Only New Information

One of the problems with having a single log file is that we don't want to be alerted of all previous traps, we wish to only be alerted of new traps. In order to accomplish this task a series of steps must be performed with the log file before we can start to parse it.

The script is run every time that a trap is encountered. We can presume that the first time we run the script that whatever traps we encounter are new and that we wish to be alerted about them. Before we get to what the script does the reader should be aware that the snmptrapd initialization script needs to be edited. This is because the script expects that both the original log file that snmptrapd logs to and a copy exists. When the script is run again (presumably the first trap that is encountered) the original log file is copied to a third log file. This third log file is then compared against the second log file. A fourth file is then created which contains the difference between the two containing only the newest traps.

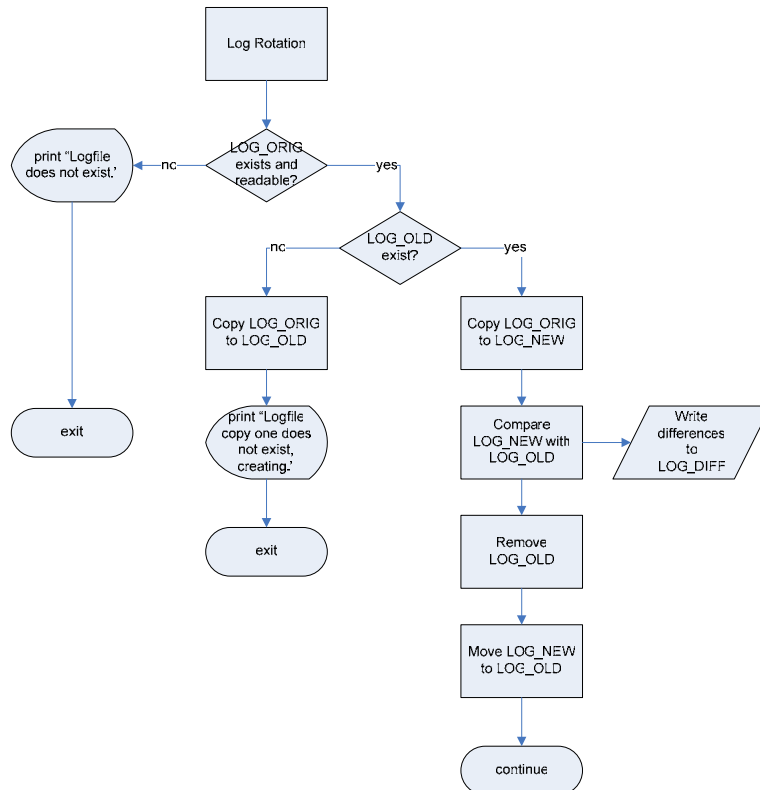


Figure 3-2: Log Rotation Logic

Figure 3-2 above shows the log rotation logic. To reiterate the previous paragraph, we have 4 possible log files during the log rotation sequence. These are the original log file (LOG\_ORIG), a copy of the original log file from the last time we parsed information (LOG\_OLD), a temporary copy of the original log file (LOG\_NEW), which we will compare against LOG\_OLD. Finally, we have LOG\_DIFF which contains the differences between LOG\_OLD and LOG\_NEW. This LOG\_DIFF file will be read into an array which we will then parse for relevant information.

When the script is run, the log parsing routine performs the following actions: LOG\_ORIG is copied to LOG\_NEW. LOG\_NEW is then compared against LOG\_OLD and any differences between the files are then written to LOG\_DIFF. After that LOG\_OLD is removed and LOG\_NEW is renamed to LOG\_OLD. This process is repeated each time the script is run. It is LOG\_DIFF which we will parse as it will contain only the newest traps which have not been reported yet.

## Parsing the Log File

In order to parse the LOG\_DIFF file, we need to pull its contents into the script. This is done through the use of a sub routine which reads in each line of the log into its own position in an array. We can then use this array in another sub routine to parse the information.

In order to parse the information we need to match certain “phrases” in the line in order to grab the relevant information and ignore others. Since it is possible to have traps logged to the file that we are not interested in, such as device system uptime, we need to create filters in order to limit the information that we want to send to the administrator. This is done through the use of regular expressions in Perl.

### What are Regular Expressions?

Regular expressions are a syntax that we can use to do string comparisons, selections and replacements. Based on this we can do complex parsing that would otherwise be impossible. Regular expressions are a huge area of knowledge bordering on an art that one could not adequately describe in this document. If the reader is interested in learning more about regular expressions, <http://www.perl.org> contains a wealth of knowledge.

For this project we use regular expressions to manipulate the data from the log file to create a string which we can then pass to the text to speech engine for the creation of the wave file.

Let's show some of the regular expressions we used in the Perl script for parsing the information related to a link going down and how we turn what appears to be random data into something meaningful. Figure 3-3 below shows snippets of regular expressions used in the interface down filter.

```
1  foreach $line (@_) {
2  if ($line =~ /^>/) { # if the line ^begins with >

3
4  # if an interface link goes down
5  if ($line =~ /\bIF-MIB::linkDown\b/) {
6  $line =~ /IF-MIB::ifDescr.[0-9] = STRING: (\b\w+\W\w\b)/;
7  $interface = $1;
8  $line =~ /SNMPv2-SMI::enterprises.9.2.2.1.1.20.[0-9] = STRING:
9  ([^.*]*)/;
```

```

8   $status = $1;
9   chomp($status);

10  # get the date and split it into 3 variables
11  $line = $_[--$i];
12  chomp($line);
13  @datesplit = split (/\s+/, $line);
14  $date = &main::DateParse($datesplit[1]);
15  $time = &main::TimeParse($datesplit[2]);
16  $ip = $datesplit[3];

17  $CALL_SETVAR = $ip;
18  print "ParseLog SETVAR: $CALL_SETVAR\n";

19  $tempstring = "Interface, $interface, current status is, $status, on,
20  $date, at, $time, from device, $ip";
21  $newstring = &main::TextParse($tempstring);
22  print "newstring\n";
23  print fileOUT "$newstring\n";
24  $i++;
25  $matched = 1;
    }

```

**Figure 3-3: A Log Filter Using Regular Expressions**

The regular expressions in Figure 3-3 have been bolded in order to more easily see them within the code. What we are doing here is going through each line of the array and checking to see if the line starts with a greater than (>) sign. The reason we do this is that when we run the 'diff' command on the log files all lines that we care about will start with a greater than sign. If not then we do not care about the line as it contains no relevant information.

Regular expressions always start and end with a front slash (/). There are also some characters which have special meaning within a regular expression. If these characters are to be used literally they must be preceded with a backslash (\).

Let's break down each of these regular expressions to find out what it is that we are trying to accomplish. The first regular expression `/^\>/` is true if the first character of a line begins with a greater than. We say that the greater than must be at the beginning of the line with the use of the caret (^) symbol. We then match the literal greater than sign by preceding it with a backslash in order to negate its special meaning.

Our next regular expression is the most general information that we can use to match a particular type of trap. Since we are trying to determine if an interface has gone down or not, we can use the `\bIF-MIB::linkDown\b/` regular expression. This line is very straight forward in that we are looking for the text IF-MIB::linkDown. We set a boundary by enclosing the string with a \b.

So far the regular expressions we've used have only returned Boolean values true or false depending on whether we have matched or not. Next we want to take some of the information from the log and use it for the creation of the string we are going to pass to the text to speech engine. We do this by placing round brackets around our regular expression. This has the effect of placing the matched contents into a variable. The first set of round brackets encountered in the regular expression is saved to variable \$1, the second variable \$2 and so on.

## Creating the String

When we create our string to send to the text to speech engine, we would like to include the time and date that the trap was received on. Unfortunately the traps do not send along a date and time with the error. Luckily the SNMP trap daemon does log it for us. As seen in Figure 3-3 we can see that the date is on a separate line, and worse yet, the line above where our trap resides. In order to combat this, we needed to implement a simple counter. This allows us to know what line we are on and when we need to grab the date for a particular trap we can just retrieve the value in the previous line. We do this at line 13 of the code in Figure 3-3 by assigning the value of the array one less than the line we are currently at. The `chomp` simply removes any carriage returns on the line. Our next line contains our old friend the regular expression again. We take the contents of the line and use the `split` function. `Split` allows you to match on any pattern you want and use that as the split point. The text that is split up is then assigned its own position within the array for later parsing. The regular expression we use is `/s+/,` which simply means to match on one or more spaces (the `+` means more than one).

The next step in the process is to convert the date, time and IP address to something which we can pass to our text to speech engine which will generate meaningful output instead of just what might sound like random numbers. This is done through the use of a couple sub routines. The date and time are parsed with the sub routines in Figure 3-4 and 3-5 respectively.

```
1 sub main::DateParse
2 {
3 # accept date in the format 2004-01-31
4 # split based on a hyphen
5 # switch month to word
6 # return in format January, 31, 2004
7 my %monthlist = (
8     "01" => "January",
9     "02" => "February",
10    "03" => "March",
11    "04" => "April",
12    "05" => "May",
13    "06" => "June",
14    "07" => "July",
15    "08" => "August",
16    "09" => "September",
17    "10" => "October",
18    "11" => "November",
19    "12" => "December"
20 );
21 my @date = split /-/, $_[0]; # split the date into separate array
    values based on hyphen
22 my $datereturn = "%monthlist{$date[1]}, $date[2], $date[0]"; #
    switch month to word
23 return $datereturn; # return in wordy formatting
24 };
```

**Figure 3-4: Date Parsing Sub Routine**

The `main::DateParse` sub routine uses a hash key/value pair to change the numeric date into a text/numeric combination which will sound more natural when converted with our text to speech engine. We accept the date in the format of YYYY-MM-DD and return it in a format more like January, 31, 2004. We do this with the `split` function and the `/-/,` regular expression. We use our `%monthlist` hash on line 22 of

Figure 3-4 to switch the numeric month with a textual month format. This is then assigned along with the original day and year to the \$datereturn variable and returned. A similar operation is performed in our main::TimeParse sub routine as shown in Figure 3-5 below.

```
1 sub main::TimeParse
2 {
3 # accept time in format 15:59:59
4 # split based on :
5 # format nicer
6 # return
7 my @time = split /:/, $_[0];
8 return "$time[0], hundred hours, $time[1], minutes";
9 };
```

**Figure 3-5: Time Parsing Sub Routine**

With the date and time now parsed there is just one more sub routine we need to make use of before our data is ready for Cepstral – main::TextParse. Its use can be seen on line 20 of Figure 3-3 and the code in the appendix. It works similar (albeit slightly more complex) to the date and time parsing sub routines. Its purpose is to allow us a place to convert common words or phrases which we would like to switch for better output from Cepstral. For example, when a connections status returns to the up state, it often sends a “KeepAlive” as well. Since Cepstral can’t interpret the KeepAlive as two words, we need to separate the words. We use a hash again to accomplish this by splitting the string into an array. The array is then searched for the key in the hash and replaced with the keys value. The entire string is then recombined and returned.

We then write our newly formed string to a file for text to speech processing and wave file creation. We set our \$matched variable to ‘1’ to indicate that we have matched one of the filters and created a string which we can pass to Cepstral.

## **Converting Text to Speech**

With the information successfully parsed from the log files we can pass this to our text to speech engine for voice synthesis. This wave file will be passed to the communication component of the system so that the device we wish to monitor can sonically communicate with the administrator.

## **How is Speech Generated from Text?**

Let’s start with a little history on speech synthesis. It has been desired to artificially generate human speech for about 800 years, of course not with electrical signals, but with mechanical devices. Interest resurged in this area in the late 18<sup>th</sup> century. Christian Kratzenstein in 1779 was able to generate vowel sounds (a, e, i, o, u). This was followed by a bellow operated “Acoustic Mechanical Speech Machine” in 1791. There were several more enhancements and discoveries, leading up to Bell Labs VOCODER in the 1930’s.<sup>4</sup> This was an electronic speech synthesizer, reflecting the more modern technology of today.

A TTS engine is comprised of a front and back end. In the front end, there is a process to take text and transform it into what is known as symbolic linguistic

---

<sup>4</sup> This information was sourced from [http://en.wikipedia.org/wiki/Speech\\_synthesis#Overview\\_of\\_speech\\_synthesis\\_technology](http://en.wikipedia.org/wiki/Speech_synthesis#Overview_of_speech_synthesis_technology)

representation. This processes the text into phonetic portions like sentences and phrases. As well it must take numerical information and transform it into written word form (i.e. 1 to one). The back end must take this symbolic linguistic representation and synthesize a waveform. The manner in which a waveform is created is an entire science of its own, with various methods to reproduce voice.

Diphone concatenation is a branch of concatenative synthesis. This is where a database of possible sounds used in a given language are stored, and strung together to create voice. The Spanish language can be represented by about 800 diphones, German by about 2500. English sits somewhere in the middle with about 1400 diphones.

There are other methods of speech synthesis, one such, is known as HMM. The Hidden Markov Model is a method which involves the spectrum of human voice, fundamental frequency and changes in correlated pitch and timing. This is a statistical model based on pattern recognition and the likelihood of these patterns occurring.

Text to Speech was an interesting element of this project to explore. It represents the frontline of the notification system. That is, after the Perl script has performed its duties, it is up to a text-to-speech engine to generate the final waveform notification.

In the early stages of the project, we had decided upon Festival (open source TTS) as our engine. This seemed the natural choice as there was support for the Asterisk platform. Festival makes use of Asterisk's built in application for launching external programs (AGI), allowing for interaction between Festival and Asterisk. What we didn't know was that support for Festival wasn't as strong as it used to be. It became problematic when using the newest build of Asterisk. We did manage to get some of the diphone voices to work, but quality was extremely poor. We needed a new TTS solution, and without direct support from Asterisk, it forced us to externalize the TTS process entirely with Perl. This ended up being rather freeing in our choice of software to use.

Cepstral was the TTS engine we chose to use, demo versions were available at the cepstral.com website, and the output this engine created was fairly high quality. It can be reasoned that this TTS engine is based on the Hidden Markov Model. Cepstrum coefficients are those that comprise the spectrum of some audible range, in our case human voice. HMM is used in conjunction with the spectrum of some sound, and the mel-cepstrum model is Fourier transform of spectrum. This is statistical in nature, and this is where the word cepstrum derives.

## Creating the Wave File

The Perl script calls the 'swift' program, which is the Cepstral binary. It's a very simple process which is relatively quick. Found in the script is the following line.

```
swift -f $sounddir/say-text-$hash.txt -p audio/channels=1,audio/sampling-rate=8000 -o $wavefile
```

The variable \$sounddir contains the path to the file containing the text we wish to convert to speech and the \$wavefile variable is the name of the file we wish to write the wave file to.

## **Call Files and Channel Variables**

Call files are plain text files which when placed in a certain directory causes Asterisk to take action. Channel variables are ways of passing information to Asterisk which are only available for the current channel.

### **Call Files**

We can use call files to make Asterisk perform an action which is most commonly to place a call by opening a channel to a device. The device could be a locally registered extension or remote extension if supplied the correct arguments. For our project the call file was used to make Asterisk call our device administrator so that we can play back the wave file generated by Cepstral. Below in Figure #-# we have an example call file that could be generated by the Perl script.

```
1 Channel: SIP/3005
2 MaxRetries: 3
3 RetryTime: 60
4 WaitTime: 30
5 Context: snmp
6 Extension: s
7 Priority: 1
8 SetVar: device=192.168.1.101
```

**Figure 4-1: Sample Call File**

Our call file in Figure 4-1 shows that Asterisk will call channel SIP/3005 and when answered, will jump to context snmp, extension s and priority 1. It will also assign the value 192.168.1.101 to the variable device which will be available only for this channel.

### **Channel Variables**

Channel variables contain information that is available only for the channel that it is assigned to. This information can be used in a variety of ways, but for our project we are using it to pass the IP address of the device that generated a trap. We can then use this information to do a lookup in the local Asterisk database to retrieve information about the device and the administrator of the device, such as administrator name, email address and extension to call.

## **The Asterisk Dialplan and Database**

The Asterisk dialplan is what defines the actions the system will take. It is the heart of Asterisk and we will explore the dialplan sections required for the system to work. As well, Asterisk includes its own database which we can use to store and retrieve values such as administrator name, extension and email address.

### **The Dialplan – extensions.conf**

The dialplan is where we are going to pull together the information we have received from the Perl script and communicate the information to the administrator. For the sake of brevity the dialplan will not be explained in detail, but some exploration is essential for the reader to fully appreciate the project. Figure 5-1 below shows the snmp context referred to previously in the sample call file (Figure 4-1).

```
1 [snmp]
2 exten => s,1,Answer
```

```

3   exten => s,2,NoOp(${office})
4   exten => s,3,Playback(tts/tts-${HASH})
5   exten => s,4,GotoIf(["${office}" != ""]?office,1)
6   exten => s,5,System(rm /var/lib/asterisk/sounds/tts/say-text-
   ${HASH}.txt)
7   exten => s,6,Wait(1)
8   exten => s,7,Playback(goodbye)
9   exten => s,8,Hangup

10  exten => office,1,ResponseTimeout(30)
11  exten => office,2,DBGet(admintoemail=device/${office}/email)
12  exten => office,3,System(cat /var/lib/asterisk/sounds/tts/say-text-
   ${HASH}.txt | mail -s "[ASTERISK]: Remote Management Alert"
   ${admintoemail})
13  exten => office,4,System(rm /var/lib/asterisk/sounds/tts/say-text-
   ${HASH}.txt)
14  exten => office,5,Playback(tts/admin-has-been-emailed)
15  exten => office,6,Background(tts/to-call-admin-of-device)
16  exten => office,7,Background(tts/press-1)
17  exten => office,8,Background(tts/to-turn-on-an-interface)

18  exten => 1,1,DBGet(admintocall=device/${office}/phonenum)
19  exten => 1,2,Dial(${admintocall},30)
20  exten => 1,3,Playback(goodbye)
21  exten => 1,4,Hangup
22  exten => *1,1,System(snmpset -v 2c -c public 192.168.1.101 IF-
   MIB::ifAdminStatus.2 i 1)
23  exten => *2,1,System(snmpset -v 2c -c public 192.168.1.101 IF-
   MIB::ifAdminStatus.2.i 2)
24  exten => #,1,Hangup
25  exten => t,1,Hangup
26  exten => i,1,Playback(invalid)
27  exten => i,2,Goto(s,2)

```

**Figure 5-1: SNMP Dialplan Context**

We will first explore the `s` extension which consists of lines two through nine. This section will first make sure that the call is answered and through the use of the playback application will stream the wave file generated by the text to speech engine. Line 5 of the dialplan performs a `GotoIf`, which checks to see if we have a value stored in the `${office}` variable. This variable is populated by the use of the `SetVar` command in the call file. We expect to see the IP address of the device in this variable, and if it is not NULL, then we go to the `office` extension. If the value does happen to be NULL, then we simple continue along in the `s` extension. We then run the `System` application which allows us to execute a command at the operating system CLI – in this case to remove a file we no longer need. Why this file is deleted here and not in the script will be explored in the next section. We then continue to fall through the rest of the priorities in the `s` extension and playback a goodbye message and finally disconnect the user from the system.

The `office` extension is reached when we detect that we have a value in the `${office}` variable. We use this value to retrieve information from the database regarding the device specified in the variable. Line 11 of the dialplan uses the database get (`DBGet`) application to retrieve the email address of the administrator of the device and assigns it to a new variable called `admintoemail`. Now it becomes clear why we haven't removed the text file that our text to speech engine used to create the wave file; we can also email the administrator of the device for information redundancy! After we have sent the email we remove the text file so that the script can create a

new text file the next time it is run. However a better way of doing this would probably be to check for the existence of this file in the script and remove it before writing again<sup>5</sup>. This simply shows the power and flexibility that the Asterisk system possesses; you can get the same results in lots of different ways.

Our system has the ability to let the administrator also interact with their device through Asterisk. We assume that the same people, or group of people, are called each time any monitored device reports a trap. However, we can also assign someone else to the device we may wish to call to take care of any problems; perhaps they are physically local to the device. We store this information in the local Asterisk database. If we wish to call the assigned administrator of the reporting device, we select that option from our menu. The dialplan then executes the DBGet application to lookup the extension for the administrator and Asterisk then connects us without requiring any extra phone numbers to be written down or dialed.

Another interesting feature we have developed is the ability to remotely turn on an interface without ever having to login to a CLI. This is done through the use of the System application to run `snmpset`. The `snmpset` program is used to issue a change to a writable option in the SNMP MIB database. This is done through the use of SNMP SET requests as mentioned previously in the SNMP section of this paper. Through the use of the `snmpset` program we can change the value of the OID in order to change the status of an interface between the up and down states.

## The Database – AstDB

The Asterisk database is an implementation of version one of the Berkeley database. It works on the bases of a key/value pair with each key having its own family. The `db.c` source code states, "DB3 is licensed under Sleepycat Public License and is thus incompatible with GPL. To avoid having to make another exception (and complicate licensing even further) we elect to use DB1 which is BSD licensed", hence the reason for version one.

For our project we used the Asterisk database to store information about a particular administrator of a device. We pass the IP address of the device which we then use as part of the family for the various keys we wish to retrieve values from. An example of this is the `/device/192.168.1.101` family which contains the keys `owner`, `email` and `phonenum`. The keys respectively contain unique values such as John Smith, [jsmith@example.com](mailto:jsmith@example.com) and SIP/3005.

In order to store the information to the database we simply used the Asterisk CLI. This is accomplished through the use of the 'database' command. To store the value SIP/3005 to the `phonenum` key for the device `192.168.1.101` we type the following.

```
*CLI> database put device/192.168.1.101 phonenum SIP/3005
```

---

<sup>5</sup> Since the writing of this paper, this functionality was included in the Perl script.

## Conclusion

We've examined all the processes that create our alert system. As well, we have seen how these processes are bound together using Perl. There are several further design considerations we have made that should be mentioned. Initially, it was our intent to use snmpget to retrieve administrator information from an agent being reported on. We later decided that if the notification was indicative of link failure or lost communication, it would be impossible to retrieve such information. Instead, a local database of administrator information is stored on the management node. This database is consulted upon receipt of notification, and appropriate information is passed.

Another potential problem was receiving multiple traps at once. This problem became evident when consecutive notifications were not being placed by asterisk, or the resultant waveform notification had been truncated. It was theorized that Cepstral was creating more than one waveform of the same name while asterisk was trying to play the message. Some alterations were made to the script to allow asterisk to finish with the file before a new one is created. In addition to this, the notification was emailed in text form for the sake of redundancy.

Some considerations for further development were also made. We discussed enhancing the system to have levels of priority based on given traps, so that notifications could be directed to different extensions. This would allow for tiered notification in an administration capacity. It may be worthwhile to extend this notification system to that of a ticketing system, allowing an administrator or technician to clear calls as they come in and store those that still require attention.

With the groundwork we have created, it will be possible to create further notifications based on any desired traps. This may include notifications for environment variables, configuration changes, and TCP connections for applicable agents. With advancements in speech-to-text technology, it may be conceivable to interact with a given agent vocally. This technology is already emerging with software such as Sphinx.

# Appendix