DRAFT

ASTERISK

A conceptual introduction

Ben Fuhrmannek <bef@eventphone.de>

May, 2007

Draft 1

TABLE OF CONTENTS

ASTERISK

A conceptual introduction

Introduction

As ironic as it might appear, this is the introduction to the conceptual introduction to asterisk. You - the reader - are probably aware of the fact that the word *asterisk*[1] refers not only to a symbol representing a star (*) in this document, but rather tries to emphasize what has fascinated millions of humans throughout the decades since the invention of the telephone itself. I am talking about an extraordinary private branch exchange[2], of course. At the end of the 19th century - where the telephone and its corresponding network had just been invented - a person had to connect two phone lines manually in order to establish a connection between two phones eventually leading to a bilateral conversation. Although this person was replaced long ago by automated techniques not involving people for each and every connection, asterisk's job can be considered almost identical to the job of this switchboard handler. Now, that you have learned to appreciate the true meaning of the word asterisk I would like to take a minute and try to describe you - the target group. There are so many forums and documentation wikis related to asterisk and the configuration of asterisk available in this write-only medium commonly referred to as the internet. As a direct consequence - and the fact that you are still with me underlines my theory - I conclude that you either stumbled over this document by accident or - which is more likely, yet not mutually exclusive - you would like to broaden your mind by adding a new perspective regarding the asterisk open source private branch exchange to your own, personal repository of memories. While I cannot promise the ultimate enlightenment it might still be illuminating to finally understand one of the concepts of modern communication. For starters, I would like you to keep in mind, that the whole point of fiddling around with asterisk is neither a better understanding of its versatile configuration nor quenching your never ending thirst for knowledge - although this comes very close to it -, but in my opinion the whole point is to enable people to communicate with each other.

Nevertheless the content of the next few pages will provide you with a quick start into asterisk starting with a conceptual overview leading to a better understanding of how to configure your PBX. This is closely followed by a section describing the most exciting ways of how to extend asterisk unleashing an idea of its true potential.

---

[1] http://www.asterisk.org

[2] private branch exchange: http://en.wikipedia.org/wiki/PBX

Understanding the PBX

Asterisk comes with such a broad range of functionalities making it exceptionally configurable however rather complex, too. In order to break down this complexity I would like you to step back and take a look at the bigger picture here. We would like to connect one phone to another. So, after commenting on asterisk's general structure and its configuration files I will change the perspective and take a look at the two phones desperately desiring to be connected from the point of view of our PBX. Here we can follow the path of an incoming call in detail along with all corresponding configuration files.

A good start in understanding asterisk is a hands-on approach. In order for this to work you should have a working copy of asterisk installed on your system ready to be configured or reconfigured respectively. I am using an asterisk version 1.2.18 for my explanations and although there is probably a newer version available when you read this introduction, the basic concepts explained here are likely not to have changed considerably.

About Files and Modules

First of all it is crucial to know that asterisk possesses the ability to be extended by modules which are loaded during runtime. Virtually every other functionality is packed into these modules. By naming convention every module is prefixed by a string representing the kind of functionality it provides for the asterisk server. You can see for yourself by typing `ls /usr/lib/asterisk/modules` into your favorite shell assuming you are using the default paths for your asterisk installation. The following table clarifies the meaning of each prefix.

PREFIX	DESCRIPTION
app_	applications which can be invoked from within the dialplan
func_	functions for use in the dialplan
chan_	channel drivers - e.g. for IAX2 or SIP
codec_	audio and video codecs
format_	audio and video formats
cdr_	call detail records
pbx_	core features of the PBX - e.g. config-file parsing
res_	resources to be used by other modules - e.g. database access

Considering that most modules can be configured by their own separate configuration file it comes as no surprise that there are quite a lot of these files. So, let's get over it and embrace the overwhelming quantity of asterisk's configuration files in all its beauty.

```
$ ls /etc/asterisk
adsi.conf           codecs.conf         logger.conf         queues.conf
adtranvofr.conf     dnsmgr.conf         manager.conf        res_odbc.conf
agents.conf         dundi.conf          meetme.conf         rpt.conf
alarmreceiver.conf  enum.conf           mgcp.conf           rtp.conf
alsa.conf           extconfig.conf      misdn.conf          sip.conf
asterisk.adsi       extensions.ael      modem.conf          sip_notify.conf
cdr.conf            extensions.conf     modules.conf        skinny.conf
cdr_custom.conf     features.conf       musiconhold.conf    telcordia-1.adsi
cdr_manager.conf    festival.conf       osp.conf            voicemail.conf
cdr_odbc.conf       iax.conf            oss.conf            vpb.conf
cdr_pgsql.conf      iaxprov.conf        phone.conf          zapata.conf
cdr_tds.conf        indications.conf    privacy.conf
```

All we need to worry about - at least for the moment - is the `iax.conf`, `sip.conf` and `extensions.*`. Most asterisk configuration files (*.conf) are easy enough to understand. Each file is separated into sections in a win.ini-style fashion, followed by a list of key/value pairs seperated by => or = like so:

```
[section]
key1 => value1
[section2]
key = value
```

If you are unfamiliar with asterisk it can be a good idea to have a look inside one or two of the configuration files using your favorite text editor, just to get a feeling for the syntax.

Peers and Users

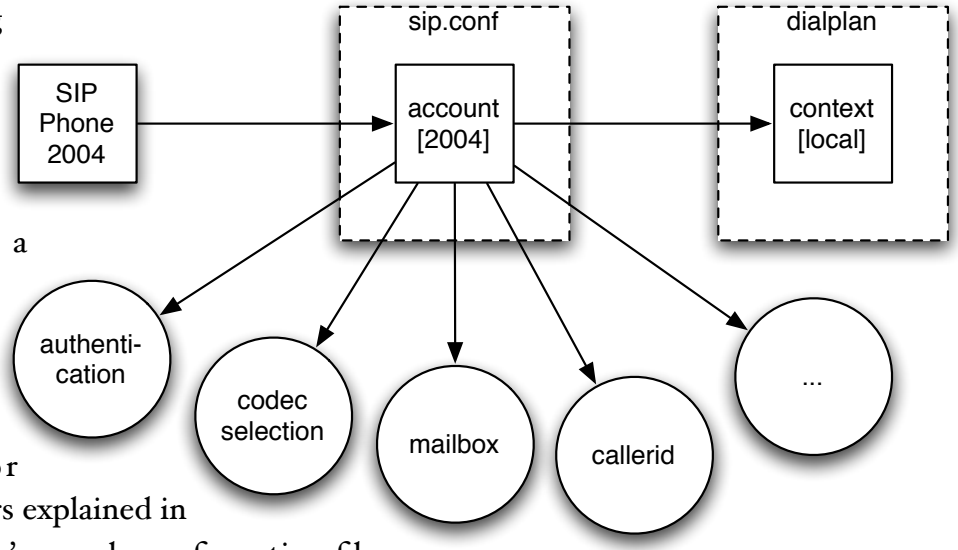
If we recall our original intension, we still need to connect two phones. Let us assume that one of the phones is a VoIP phone which has got the SIP protocol implemented. Now, in order to place or receive calls the phone must be known to our server which can be accomplished by registering in advance. The SIP account configuration typically resides inside `sip.conf` where a minimal section for our phone would look like this:

```
[2004]
type=friend
host=dynamic
secret=abc
context=local
```

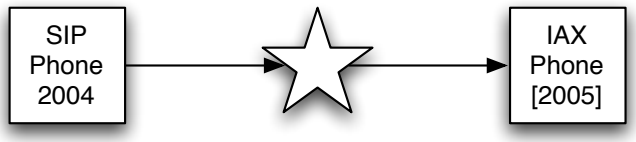
If we were using an IAX[3] phone the minimal configuration for `iax.conf` would be identical. And of course you may want to manage all your SIP and IAX accounts using a database management system which is covered by the realtime extension[4] introduced in asterisk 1.2.

Using this configuration our phone can register with the asterisk server using the username `2004`, the password `abc` and an arbitrary originating IP address (`host=dynamic`). The type of an account can either be a `peer` permitting our asterisk server to place outgoing using this account or it can be a `user` who is allowed to call us. A `friend` combines both `user` and `peer` in a single account configuration.

When registering a SIP account the corresponding account configuration can contain authentication information, a range of allowed codecs, a mailbox number, a caller ID or other parameters explained in detail in asterisk's sample configuration file.



For processing an incoming call asterisk requires an association of a context to the account the incoming call is using. Just like the SIP configuration is divided into different accounts asterisk's dialplan is divided into contexts. So the context given in the account configuration determines which part of the dialplan is to be searched for a dialed extension.



Journey of an Incoming Call

Our second phone - the one we would still like to call - could be an IAX phone which is registered to the asterisk server, too, using the account name 2005. Originating from our

[3] IAX and IAX2 both refer to the Inter Asterisk eXchange protocol version 2

[4] http://www.voip-info.org/wiki-Asterisk+RealTime

SIP phone we could simply dial 2005 and observe what's happening. Based on the SIP phone's IP address and its authentication credentials asterisk's SIP module can link the account 2004 to the incoming call. In addition we know the context in which to search for the dialed extension 2005 in the dialplan. The original way to create a dialplan, which actually had been the only way in asterisk 1.0 and before, is using the configuration file extensions.conf. A simple extension for calling our IAX phone could look like this:

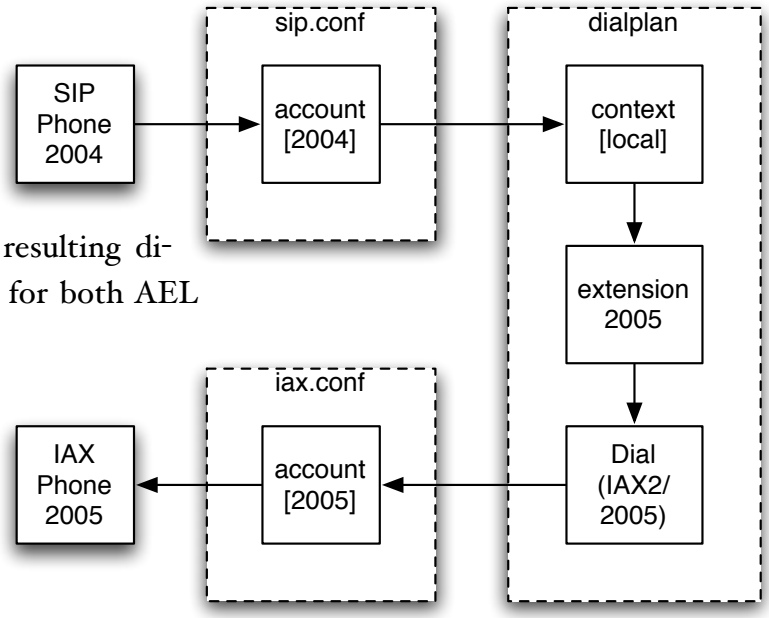
```
[local]
exten => 2005,1,Dial(IAX2/2005)
```

What we can see here is the definition of a context in brackets (`[local]`) consisting of one extension. The keyword `exten` precedes every definition of an extension. 2005 - the first of these three comma separated parameters - resembles the number of our extension, followed by a priority (1) and an application (`Dial`). Starting with 1 the priority indicates the order in which applications are executed. An accurate analogy would be a procedure in a programming language where 2005 is the procedure's name, the priority is the line number in a BASIC-like style. As anticipated the `Dial`-application is used for dialing. A complete description of the asterisk dialplan can be found in the asterisk documentation[5].

A slightly more sophisticated way of creating a dialplan is provided by the asterisk extension language (AEL). Included into the configuration file `extensions.ael` the following example generates exactly the same dialplan as the one above:

```
context local {
     2005 => Dial(IAX2/2005);
};
```

As you can see the AEL looks a bit more like a programming language and in fact it adds elements like conditions (`if`, `switch`) and loops (`while`) to the dialplan without having to revert to goto-structures. The resulting dialplan however is isomorphic for both AEL and `extensions.conf`.



So, now we can simply dial 2005 and cause the `Dial`-application to be executed with the parameter "`IAX2/2005`" which is called the di-

[5] asterisk-1.2.x.x/doc/extensions.txt

alstring. IAX2 refers to the channel type to use, 2005 is the name of an IAX peer or friend account configured in `iax.conf`. Using this information the server can easily call the IAX phone. The illustrating figure next to this paragraph concludes our journey through the PBX along with an incoming call. I suggest you sit back and let it all sink in before commencing the next section.

Command Line Interface

Everybody likes to play around and as far as asterisk is concerned you'll have plenty of opportunities to do so because the asterisk console - also known as command line interface (CLI) - provides the ultimate playground. This is where all the action takes place in terms of verbose debugging output including error reporting, realtime configuration and help acquisition and we'll jump right into the middle. Just start up asterisk in console mode using the command `asterisk -cvvvv` or attach to an already running asterisk by typing `asterisk -rvvvv`[6]. The `vvvv` means very very very verbose[7] which enables us to watch lots of useful messages passing by. Once the asterisk console is started you can see a prompt ending with `*CLI>`.

The CLI can be operated with ease using the built-in tab completion meaning you can press the tab key once to complete a word of the command you are about to type in automatically or view a list of available commands or parameters. This feature can increase the speed of your work with the CLI immensely. So let's start by pressing tab. The output should look like the following:

```
router*CLI>
!           abort       add         ael         agi         cdr
database    debug       dnsmgr      dont        dump        extensions
feature     group       help        iax2        include     indication
init        load        local       logger      meetme      misdn
mixmonitor  moh         no          pri         realtime    reload
remove      restart     rtp         set         show        sip
soft        stop        unload      zap
```

Each of these keywords either is a complete command or the beginning of a command. A few commands are particularly worth mentioning due to their relevance for everyday maintenance and debugging:

COMMAND	DESCRIPTION
quit	detach from the CLI
stop now	stop asterisk

[6] a nice abbreviation of `asterisk -r` is `rasterisk`

[7] setting the verbosity to 4 using the CLI command `set verbose 4` is equivalent

COMMAND	DESCRIPTION
reload	reload all modules
extensions reload ael reload	reload the dialplan from extensions.conf or extensions.ael
sip reload / iax2 reload	reload the SIP or IAX2 module
soft hangup ...	manually hangup a channel
show ...	show useful stuff

A complete list of CLI commands is shown by typing `help`. You may also type `help` followed by a command to retrieve a more detailed help, e.g. `help help`. Another very powerful source of information hides behind the keyword `show`. For instance, the command `show channeltypes` can get a list of available channel types which are used in a dialstring, like IAX2 seen in a previous example. Similarly a list of supported codecs is shown using the command `show codecs`, and even the current dialplan can be viewed by typing `show dialplan`.

One of the most useful capabilities of the command line interface is the debugging feature. As you might already have witnessed while playing with the CLI each call shows log messages on the console. For example, if I called number `7399` on a server listed as `server1` in `iax.conf` the output would resemble the following:

```
    -- Executing Dial("mISDN/1-1", "IAX2/server1/7399|60") in new stack
    -- Called server1/7399
    -- Call accepted by 10.0.0.1 (format alaw)
    -- Format for call is alaw
    -- IAX2/server1-1 answered mISDN/1-1
    -- Hungup 'IAX2/server1-1'
  == Spawn extension (inbound, 7399, 3) exited non-zero on 'mISDN/1-1'
```

By scanning through the output you can easily recognize that the call originated from an ISDN line and was accepted within `60` seconds by `10.0.0.1` using a-Law as codec. Eventually the channel was hung up in the end. Even more debugging output could have been seen if we switched to IAX2 debugging mode in advance (`iax2 debug`). A good starting point for logging these messages would be the corresponding configuration file `logger.conf`.

Security Considerations

Please, don't get lost in all the excitement about the newly discovered playground and take a minute to consider the proper security for your asterisk installation. When I say security I would like to protect

- the operating system running asterisk from being exploited remotely or locally,

- secrets such as passwords and account data for accessing VoIP services or databases,
- the access to my own VoIP services,
- my privacy.

OS-Exploits

Asterisk is mostly written in C and due to the nature of C it can be rather simple for a programmer to implement security risks - unintentionally of course. So, although it is next to impossible to prevent software from containing the possibility of buffer overflows, format string exploits and memory leaks we can try to minimize the risk of hostile exploitation.

For starters running asterisk as a user other than root may prevent a compromised asterisk from compromising the whole system. All you need to do is

- add a new user and group (e.g. asterisk) to your system using the preferred method (useradd, adduser, vi /etc/passwd /etc/group, ...) and
- change the ownership of a few files and directories:

```
chown -R asterisk:asterisk /var/{lib,log,run,spool}/asterisk
chown -R asterisk:asterisk /usr/lib/asterisk /dev/zap

chmod -R 750 /var/{lib,log,run,spool}/asterisk
chmod -R 750 /usr/lib/asterisk /dev/zap
```

  It may be necessary to adjust permissions for other files, too, depending on asterisk's required resources. Tools such as `lsof` and `strace` can be of help there.

- start asterisk with `asterisk -U asterisk`

Password Security

In order to protect your account passwords we have to make sure that these passwords are stored and transmitted securely. Changing permissions for the configuration files covers the first part:

```
chown -R root:asterisk /etc/asterisk
chmod -R 750 /etc/asterisk
```

Now, transmitting your account password in one or the other way cannot be avoided during the authentication process. Therefore all but plain-text authentication should be used solely: E.g. Setting `auth=md5` or `auth=rsa` for all sections in iax.conf does the trick for IAX.

Dialplan Security

When configuring the dialplan you should always keep in mind that guest users might suddenly appear in your server. Keeping your IP-address private or obfuscating SIP-Ports does count as security by obscurity and is counter-productive. If you want to keep people out of a context, implement a proper authentication. It should also be mentioned that there are dialplan applications which can jump between contexts, such as DISA and Goto.

Privacy Protection

In the PSTN it is possible to suppress a caller id granting the caller a certain level of anonymity. For a VoIP connection, however, your your asterisk is giving away tons of useful information:

- IP address
  Two computers communicating via the internet must know their counterpart's IP address. If you wanted to conceal your true IP address then you could redirect all VoIP traffic through a proxy server or an anonymity network like the tor network[8]. Redirecting time critical traffic like a realtime conversation will most certainly increase its latency - which is usually not desirable.

- Caller ID
  Like an e-mail address for email a caller id can uniquely identify a VoIP user. But unless the VoIP user was authenticated properly - e.g. by RSA or MD5 authentication - the caller id could have been altered and cannot be trusted.

- User Agent
  Most VoIP protocols transfer the name of the device or software used - aka user agent - by default. For example the default user agent for asterisk's SIP channel is "Asterisk PBX". This can be changed for SIP by modifying the `useragent` parameter in `sip.conf`.

- Password
  As mentioned previously your VoIP authentication might be using plain-text. This is insecure and can be intercepted. Simply use a different authentication mechanism.

- Codec
  There is not much an observer can learn from the used codec. However it can be speculated that a compressed codec (GSM, Speex, ...) is being used because of a bandwidth limitation and thus our line is open for a denial-of-service attack. Furthermore the order in which different codecs are negotiated between a caller and a called party can give a hint about which user agent is actually connected to each end of the connection. A-Law is dominant in Europe whereas μ-Law is more often used in Japan and USA.

---

[8] http://tor.eff.org/

The point is: Do not underestimate the value of seemingly unimportant giveaways of information.

- Spoken Word
  Apart from speaking encrypted yourself, there are ways to encrypt the VoIP traffic. The logical approach to encrypting VoIP would be a by-call negotiation of the encryption method and the keys used, comparable to the codec negotiation. SRTP would probably be the protocol of choice if it were implemented in asterisk. Alternatively there is a somewhat undocumented and unstable AES encryption available for MD5-authenticated IAX connections. One layer down, we can always encrypt our IP traffic using the usual suspects: OpenVPN, IPsec, PPTP, ...

- The mere existence of the call itself
  Can anyone determine if I'm on the line or not at any given moment? If you ever thought about this question then you are paranoid. There is simply no way anyone could set up a reliable steganographic network capable of VoIP. Just in case you manage to do so anyway, don't hesitate to call me.

Coding the PBX

Configuring a PBX is literally like programming. Even if you are not too much into programming you might want to read this chapter, just to get a feeling for your PBX's capabilities. The first section briefly explains the asterisk extension language (AEL), an alternative way of creating the dialplan. Part two can show you a way of how to encapsulate complex dialplan logic into little scripts using the asterisk gateway interface (AGI). And for remotely observing and controlling asterisk you should take a look at the asterisk manager interface (AMI).

Again, there is much programming code involved in coding the PBX, but fortunately nothing to be afraid of. Just dare to take a look anyway.

Asterisk Extension Language (AEL)

The asterisk extension language is designed to approach the programming of a dialplan in a programming style including elements of structured programming like loops and conditions. Compared to the `extensions.conf` dialplan creation AEL can be read and written with the natural ease of a program. It is like going from Basic to Python, and yet the resulting dialplans of AEL and `extensions.conf` style are isomorphic. Both methods of dialplan configuration can be used simultaneously. In fact it is possible to include one context in another regardless of its origin, like so:

```
// AEL
context ael-testcontext {
     includes {
          test;
     };
};

context n {
     1234 => Noop;
};
```

```
; extensions.conf
[test]
include => n
```

Here the AEL context `n` containing the extension `1234` is included by the `extensions.conf` context `test` which is included by the AEL context `ael-testcontext`. The special characters "//" and ";" indicate the beginning of a comment in AEL or `extenions.conf` respectively. The application named `Noop` does no operation.

For testing purposes the CLI commands `ael reload` and `extensions reload` come in quite handy.

The next little snippet of AEL code defines extension `7360` inside context `tests`. `Answer` does what you'd expect it to do, it answers the call. `Playback(beep)` plays a beep. Then there is an infinite loop constantly waiting for a #-terminated input of digits. If there is

any input within 40 seconds, the application `PlayTones` plays exactly the frequency of our input. Please note, that sequences of more than one application must be enclosed by braces (`{ app; app;... };`).

```
context tests {
     7360 => {
          Answer;
          Playback(beep);
          while (1) {
               Read(FREQ||||40);
               if ("${FREQ}" != "") {
                    PlayTones(${FREQ});
               };
          };
     };
};
```

This following code shows how to alter your caller ID based on the current value of the caller ID. When executed, this macro will set your caller ID to `7311` if it was `2008` before, and to `7310` otherwise.

```
macro cid-filter {
     switch (${CALLERID(number)}) {
     case 2008:
          CALLERID(number)=7311;
          break;
     default:
          CALLERID(number)=7310;
     };
};
```

Marcos can be executed by prefixing an ampersand (`&cid-filter;`). When dialing any number starting with `0..9` or `*` within the following context the `cid-filter` macro will alter our caller ID and move on to another context called `out-match-dst`.

```
context out-apply-cid-filter {
     _[0-9*]. => {
          &cid-filter;
          goto out-match-dst|${EXTEN}|1;
     };
};
```

The complete definition of the asterisk extension language can be found in `README.ael` of the asterisk documentation.

Asterisk Gateway Interface (AGI)

Now, that you understand the basic concepts of a dialplan, you may also have noticed its restrictions applied implicitly using only provided dialplan applications. However writing your own applications for use from within the dialplan can be as easy as the programming language you choose for this task. The Asterisk gateway interface - aka. AGI - defines a simple line-based and human readable protocol on top of standard I/O similar to CGI for the WWW. Since most programming languages support standard I/O communication you are completely free to choose the language of your choice.

AGI scripts are invoked by `AGI` application:

```
exten => 7361,1,AGI(test.sh)
```

Special derivatives of the AGI command - FastAGI, EAGI, DeadAGI - are available, too, but let's focus on the concept of AGI programming.

AGI scripts reside in /var/lib/asterisk/agi-bin by default. That's where test.sh is located. It shows a plain bash-script which will basically answer the call, stream a file called 'beep' and hangup:

```
#!/bin/sh
x=start
while [ "$x" != "" ]; do
    read x
done

echo ANSWER
read

echo "VERBOSE \"streaming file beep\" \"4\""
read

echo "STREAM FILE \"beep\" \"#\""
read

echo HANGUP
read
```

During the while loop at the beginning of the script environment variables are read and immediately discarded since we won't need them for this simple test example. One convenient possibility to observe the communication between asterisk and your AGI script is to enable the AGI debugging mode[9]. Subsequent calls of `echo` and `read` will each issue an AGI command and receive one line of response which could be evaluated for failures.

---

[9] CLI command: `agi debug`

A complete list of AGI commands of your Asterisk installation, such as ANSWER and HANGUP, is shown on CLI: `show agi`. ANSWER undoubtedly answers the call. VERBOSE can inject a log message with a specified verbosity level probably leading to the message being displayed on the asterisk console. STREAM FILE streams a file, which can be interrupted by the given symbols (e.g. #). HANGUP terminates the connection and may also terminate the AGI process.

For easier handling of the AGI protocol there are modules available for many languages [10]like Java, Perl, Python, Tcl, Ruby, C, C#. The equivalent to the previous example rewritten in Tcl using the appropriate helper module[11] could look like this:

```
#!/usr/bin/tclsh
package require agi

agi::init
agi::answer
agi::verbose "streaming file beep" 4
agi::streamfile "beep"
agi::hangup
```

As you can imagine, the integration of your own code can be quite simple at this point. You could access databases, query URLs, send messages, open garage doors, you name it.

More on AGI may be found scattered throughout the net[12], although I find reading the example scripts bundled with asterisk can be a good starting point.

Asterisk Manager Interface (AMI)

A remote control for asterisk is what we are looking at here. The CLI is for us humans what's the AMI for scripts. They can simply connect to a TCP port - 5038 by default - and issue commands in a human readable, yet machine parsable format. Commands can be processed synchronously or asynchronously, meaning they either block and wait for its completion or return immediately and signal its end and possible return values later.

Additionally the AMI can be used to subscribe to events like an incoming call or a registration timeout.

In order to get a feeling for the protocol we will analyze a simple session initiated on the command line using telnet:

---

[10] references and examples: http://www.voip-info.org/wiki-Asterisk+AGI

[11] http://tel.sf.net/

[12] http://www.google.ru/search?q=asterisk+agi

```
$ telnet localhost 5038
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Asterisk Call Manager/1.0
Action: Login
Username: test
Secret: test

Response: Success
Message: Authentication accepted

Event: Registry
Privilege: system,all
Channel: SIP
Domain: voipuser.org
Status: Registered

Event: PeerStatus
Privilege: system,all
Peer: SIP/1234
PeerStatus: Registered
```

Once the connection has been established, you should see a greeting - "`Asterisk Call Manager/1.0`". Now we can issue our commands. The first action usually resembles a login which must be configured in advance in `manager.conf`. The successful login is then being acknowledged, quickly followed by two events. At this point further commands could be issued, depending on the configured permissions for the logged-in user.

You should keep in mind, that this protocol is using an unencrypted TCP channel with plaintext authentication by default.

As always, details are to be found in `manager.txt` in the asterisk documentation and the CLI[13].

[13] CLI command: `show manager commands`

Notes

Acknowledgements

You might have noticed the fancy photo of an old-style telephone located on the title page of this document. It was used with kind permission of pixelquelle.de.

Further Reading

If you are trying to gather more information about the asterisk open source PBX a good place to start would be the asterisk homepage[14]. Apart from the obvious there is also a self-inflating conglomerate of VoIP related information aka wiki[15] available. For those of us who fancy the good old hardcopy I can recommend the O'Reilly book on Asterisk as a starting point.

A Word About Related Projects

There are numerous projects aiming to be the ultimate software PBX, each of them coming with its own pros and cons. Just to mention a few outstanding projects: Jolly's PBX4Linux, FreeSwitch, Yate, OpenPBX. The last one actually happens to be a "freed" branch of Asterisk incorporating many GPL-only modules and add-ons.

Author Contact

The author of this concatenation of random characters - Ben Fuhrmannek - can be contacted easily by using one of the following means of communication:

eMail: bef@eventphone.de

VoIP / SIP: 7310@bef.eventphone.de

VoIP / IAX2: bef.eventphone.de/7310

IM / Jabber: bef@jabber.berlin.ccc.de

Please don't hesitate to send any comments, critics, remarks, thoughts or even ideas you might encounter while enjoying this light reading directly to me. I'd be delighted.

License

---

[14] http://www.asterisk.org

[15] http://voip-info.org/wiki/

[16] http://creativecommons.org/licenses/by-nc-nd/2.5/