Asterisk: A Bare-Bones VoIP Example

by John Todd
07/03/2003

Open source software (OSS) has achieved a dominant role in the delivery of IP-based content such as web data (Apache) and email (`sendmail`), and is making serious headway in streaming media (`icecast`). As processors become less expensive and more powerful, even jobs that were once relegated to specific hardware (such as routing and load sharing) are now becoming possible on low-cost OSS platforms running Linux or BSD-based operating systems. The last bastion of hardware-specific functionality in the office environment has been the phone system, or PBX (Private Branch Exchange). PBX installations range from key systems with a few lines to large platforms fed by Primary Rate Interface ISDN (PRI) that are complex and expensive to deploy, with hundreds or thousands of extensions spanning several states or continents.

Until now, open source telephony applications have been at the periphery of the PBX, and even then, they have not been PBX-specific: fax modem software, simple voicemail software, and caller-identification software all work in conjunction with standard phone lines, but rarely together in concert as a unified platform.

Asterisk is both an open source toolkit for telephony applications and a full-featured call-processing server in itself. It can be a standalone system, or used as an adjunct to a previously existing PBX or Voice Over IP (VoIP) implementation. It can be software only, moving calls around via IP, or it can have a variety of hardware interfaces to directly tie in with existing TDM (Time Division Multiplexing) equipment. Asterisk is *not* a VoIP platform; it is a Computerized Telephony Integration platform, which just happens to have a number of very useful input/output channels through VoIP. Asterisk can just as easily be a server that has no Internet connectivity, but uses PCI-card-based analog or digital trunks to process calls -- an important distinction between Asterisk and many other systems.

It is difficult to describe the full feature set of Asterisk due to the number of fairly complex topics that are incorporated into the system: multiple VOIP channel types, hardware interfaces, a scripting language, an API, modules, and more features than can be addressed in this short article. To provide a brief introduction to Asterisk's capabilities, I will show an example that creates a very simple PBX with two extensions and voicemail on each. There will be no external connectivity to this PBX; we will simply be able to call from one line to the other. This would allow, as an example, two users to be in separate parts of the country but they could ring each other's desk phones. If the called party was unavailable, voicemail could be left.

Voice Over IP (VoIP)

VoIP has been around for a while, but has been fairly restricted to high-end users such as phone companies and large enterprise phone networks; only recently has VoIP gained momentum with end users and smaller shops. There have been a small handful of

proprietary long-distance solutions for some time, but these were closed-source systems that did not lend themselves to any extensions by the OSS community. The push of VoIP technology closer to the grasp of the Linux/*BSD end user or administrator can be attributed to a combination of low-cost, high-speed bandwidth and a recent agreement to standardize on open protocols for call delivery.

Within the last eighteen months, it has become evident that the protocol that will be leading the industry for VoIP deployments is Session Initiation Protocol (SIP) -- see RFC3265. SIP is a simple, text-based description protocol that uses interactions similar to HTTP and SMTP in order for two systems to describe a media stream (in our case, voice traffic) that needs to get from point A to point B. The description includes authentication, caller ID information, media stream parameters, and a variety of other header information that is needed to fully qualify a call between two endpoints.

While there are other VoIP protocols supported by Asterisk (such as H323 and MGCP) I will only describe SIP, as there are a growing number of phones and software stacks that support SIP as a method for call description, and for the beginner, SIP is easier to debug due to its use of plaintext headers.

Asterisk Theory

See the Configuration section below for samples of the files that are referenced here.

Call Flow: Starting Out

A call comes in on one of several channels (SIP in our case) and is "destined" for a dialed number. The Asterisk process first deals with the call via whatever channel it came in on, and learns what to do with it in that manner, and into what context to send the call in *extensions.conf*. In our example, calls inbound from both of our SIP phones are sent to the context `from-sip`, which is where we are going to start matching the dialed numbers. The called number is translated into a variable called `${EXTEN}`, and we'll refer to this variable from now on when talking about the number being dialed. It is implicitly used in any matching statements, so you don't have to worry about specifying it elsewhere. However, for ease of reference, we will use it in this article whenever we talk about the number that has been called.

Contexts

Now that *sip.conf* has told our call what context to go to, the control is handed over to the definitions created by the file *extensions.conf*. The *extensions.conf* file works by defining various "contexts," which are clusters of dialed-number matching statements. The context is the central building block of Asterisk, and, loosely, is used as one might use a subroutine. Within a context are a number of matching statements that perform match tests against the number being processed. The call is passed through the comparison list until a match is found.

A context can have "special" extensions, which are pre-defined and are reserved for special behavior. The most commonly used extension is `h`, which means hangup, and allows your dial plan to execute certain routines at the completion of a call. See the manual for a more complete listing of special extensions. None of the special extensions are used in our example.

Extension Matching

Each context has a set of extension matches, which determine what applications should be triggered by the call, and how the call should be routed. Matching is performed in numerically ascending order, which can be tricky if you have many matches that are similar; in our example we have a very simple match list. The matching examination is done on the digits following the `=>` up until the next comma. Each match definition has at least one "priority," which simply is a number that tells the server in what order to execute the applications when a match for the matching string is found. Priorities must be sequential whole numbers, which sometimes leads to headaches if you discover you need to insert an application at the top of a priority list.

Each line is in the format:

```
exten => extension,priority,application
```

Here's an example:

```
exten => 3334,1,Answer
exten => 3334,2,Playback(welcome-to-foo-inc)
exten => 3334,3,Wait(1)
exten => 3334,4,Playback(the-date-and-time-is)
exten => 3334,5,DateTime
```

The above example will match any inbound calls sent to extension 3334 and play back a short welcome followed by a verbalization of the date and time. As long as the caller hasn't hung up, the next application will be run and the results played into the channel.

To give an idea of how this works in our mini-phone system, imagine an inbound SIP call is headed towards extension 2001 from extension 2000. Thus, `${EXTEN}` would be equal to 2001. Using our example *sip.conf* file (see below), it shows that any calls coming from extension 2000 should be passed into the context `from-sip`. When the call is passed into `from-sip`, the first match statement compares `${EXTEN}` against the string "2000". That isn't a match, so the matching process jumps to the next numeric extension definition, which is "2001". In this comparison, it is true that `${EXTEN}` equals "2001" -- we have a match! At this point, the priority 1 application is executed, which is "Dial".

If the calling channel still exists and has not been terminated at the end of priority 1, then priority 2 is executed, and so on. If there are no matches for `${EXTEN}`, then the user will be sent an "Invalid Extension" result, and most likely will hear a re-order tone (fast busy) that would be generated from their phone.

Wildcards can be used in extension mapping, and match strings beginning with the underscore character (_), meaning that the following portions of the match string include wildcard characters. Commonly used wildcards are `N` (digits 2-9), `X` (any digit), `.` (any number of digits), and a variety of regular-expression matching methods. See the handbook for more detailed explanations of wildcards for matching. A valid example of a wildcarded matching string might be `exten => _301.,1,Dial(Zap/1/${EXTEN})`, which would match any of the following: 3013, 3015551212, 301543*999.

Much more extensive comparisons can be applied to the matching behaviors: caller ID of calling party, time of day, detection of fax modems, string matching, and more can all be used to determine call flow.

Also, for a quick description of some system variables and conditional expressions, take a look at the file *README.variables*, which is in the main Asterisk source directory.

Applications

After a match is made on `${EXTEN}`, the applications start to be executed in the order in which they are listed by their priority values. There are a huge number of applications that are available, and additional applications are fairly easy to integrate. There is even an application called AGI that links to external programs, and there are Perl and Python libraries that allow for easy development of tools external to the applications built into Asterisk.

The most-used application is called Dial, and that is the application that rings a remote channel and then connects the two different channels together if there is an answer. The Dial application has some special abilities due to its multiple responses. If a Dial application gets an answer on the remote channel, then the two callers are bridged together and the call proceeds. After an answer, the only options are for one or both parties to hang up. When a hangup happens, the Dial routine exits with a non-zero status, and the priority list stops executing because we have lost the call -- this is a normal call termination.

If the Dial application rings the remote phone for 20 seconds (specified by the `,20` in our Dial statement) but there is no answer, Dial will exit and the next priority will be executed -- in our case, that next priority is a command to run the Voicemail application, which sends the caller to the "unavailable" greeting for the called party. If the Dial application gets a "busy" answer back from the remote phone, or the remote phone is not on-line, then the Dial application does something special: it adds 101 to the existing priority, and jumps to that priority. In our case, this means priority 102, which sends the caller to the "busy" greeting for the called party. Dial is the only application that has this special priority incrementing ability, though there are priority control applications that can give the administrator direct control to the priority, context, and extension such as Goto, and more sophisticated versions such as GotoIf, which can be used to evaluate expressions.

To get a list of applications, just type "show applications" at the command prompt, and then "show application xxxx," where xxxx is the application for which you want more information.

Prerequisites

Asterisk currently runs on Linux > 2.4.x (various flavors) and OpenBSD 3.3. The OpenBSD version has not been tested thoroughly and should be considered "alpha" at the time of this writing (June 2003).

You will also need an SIP-capable phone, such as a Cisco 7960 or 7940, SNOM, Pingtel, or Grandstream. Alternately, you could get a software client for Windows such as the free client from Xten or a Linux SIP phone like kphone. Configuring these various SIP clients is outside of the scope of this article, but I will describe what is required to make them work with Asterisk as far as SIP login data is required.

I would strongly suggest a "hardphone" to start with, especially if you are using this as a demonstration platform. Anecdotal evidence strongly suggests that nobody except the most hardcore road-warrior wants to talk on the phone through their computer -- they want a handset that looks and acts like a phone. Plus, the SIP hardphones are usually debugged quite well, as the vendors cannot simply expect to roll out a "patch" to upgrade their customers without extensive preparation, so I have found that the hardphones are generally more reliable than softphones.

Your two phones or phone clients should have at least 80 kilobits of capacity each back to the Asterisk server, as we will be using G.711 codecs that take up quite a bit of bandwidth. There are other voice encoding methods that get as low as about 4kbps, but that's getting a bit too fancy for this article.

You will need a complete kernel source tree symlinked to */usr/src/linux* since Asterisk requires certain header files from the kernel to compile correctly. If you are running OpenBSD, this is not required. You will need to have the following packages installed to complete a full installation: `bison`, `cvs`, `gcc`, `kernel-source`, `libtermcap-devel`, `ncurses-devel`, `newt-devel`, `openssl096b`, and `openssl-devel`.

SIP Client Configuration

You will need to configure your SIP clients so that they have their SIP gateway set to be the IP address of your Asterisk server. The usernames and passwords are contained below in the *sip.conf* file -- the username is the extension, and the password is listed in the `secret=` line for each extension. The configuration of the clients is often half the battle of getting VoIP to work, but once your particular client is understood, it normally becomes plug-and-play to add more phones. Your client must support registering with the SIP server -- I would suggest telling your client to register every 15 seconds or so during your experiments, to keep things re-registering quickly after Asterisk restarts. Later, you can take this back up to 1000 seconds or higher.

Installing Asterisk:

Go to the Asterisk home page and you can follow the "Download" link to get to the correct page. I would suggest not downloading the tarball, but using CVS to create your version of the application. The project sees an enormous amount of work done to it almost daily, so getting the most recent version via CVS is strongly suggested.

To download the latest CVS repository:

```
foo# cd /usr/src
foo# mkdir asterisk
foo# export CVSROOT=:pserver:anoncvs@cvs.digium.com:/usr/cvsroot
foo# cvs login

 [type "anoncvs" without the quotes as your password]

foo# cvs checkout asterisk

 [Here you should see the package being downloaded
  and distributed into the ./asterisk directory.]

foo# cd asterisk
foo# make
foo# make install
foo# make samples
```

Now, to test, type `asterisk -vvvvgc` to start up Asterisk. If you end up with a prompt that says `*CLI>`, then you've successfully installed the demonstration configuration. If you get errors on the installation, I would suggest joining the IRC channel `#Asterisk` on FreeNode or asking the asterisk-users mailing list for advice.

Here's a compile hint. If you get "illegal instruction" errors on launching Asterisk, you probably have run across an issue with Asterisk compiling itself for the wrong processor type. Edit */usr/src/asterisk/Makefile* and uncomment your processor type and comment out all other lines starting with `PROC=` in the processor definition section.

Asterisk is normally launched with `safe_asterisk`, which will be installed in */usr/sbin*. To connect to the currently running version of Asterisk, launch a client with `asterisk -r` and you should see a command prompt.

Command-line Use

There are many instructions that are available from within Asterisk's CLI interface, and described below are several that you may find useful during your configuration. A full list may be displayed by simply hitting the "?" key, and applications may be displayed (but not executed) with "show applications."

```
reload - soft-restarts Asterisk and updates internal configs
    with changes you've made to /etc/asterisk/* - does not hang up calls
```

```
show dialplan - shows the full dialplan of how your calls will be handled

sip show peers - shows all registered SIP clients

sip show channels - shows current "live" channels that are in
    use by SIP clients (off-hook)

stop gracefully - shuts down Asterisk after all calls have hung up

stop now - shuts down Asterisk, hanging up any current calls
```

Configuration

The following configs are going to put an extremely basic configuration in place, and we will pretty much going to throw out the sample files that came with Asterisk and pare down things to an absolute minimum. If you need to find out more information about additional options, the original demonstration files are located in */usr/src/asterisk/configs* for your reference. Leave everything in place that Asterisk's "make samples" puts in for you, except for the three files below; those are the only three that need to be touched for our sample to work.

After configuration, you will have the ability to dial between your two phones, and voicemail will end up stored in the system for each extension, and will also be sent as an email attachment to the email address specified in the mailbox configurations.

Since only SIP channels are being used, we only need to modify three files for our mini-PBX two-line system: *sip.conf* (this defines the SIP peers, which are the software or hardware SIP phones), *extensions.conf* (this is where the dialplans are kept -- the meat of the system), and *voicemail.conf* (where we define the voice mailboxes for each user).

Perform a `cd /etc/asterisk`, move the existing *sip.conf*, *extensions.conf*, and *voicemail.conf* somewhere safe, and create new files with the following contents:

sip.conf

```
[general]

port = 5060           ; Port to bind to (SIP is 5060)
bindaddr = 0.0.0.0    ; Address to bind to (all addresses on machine)
allow=all             ; Allow all codecs
context = bogon-calls ; Send SIP callers that we don't know about here

[2000]

type=friend           ; This device takes and makes calls
username=2000         ; Username on device
secret=9overthruster7 ; Password for device
host=dynamic          ; This host is not on the same IP addr every time
context=from-sip      ; Inbound calls from this host go here
mailbox=100           ; Activate the message waiting light if this
                      ; voicemailbox has messages in it
```

```
[2001]               ; Duplicate of 2000, except with different auth data

type=friend
username=2001
secret=11bbanzai9
host=dynamic
context=from-sip
mailbox=101
```

extensions.conf

```
[general]

static=yes       ; These two lines prevent the command-line interface
writeprotect=yes ; from overwriting the config file. Leave them here.

[bogon-calls]

;
; Take unknown callers that may have found
; our system, and send them to a re-order tone.
; The string "_." matches any dialed sequence, so all
; calls will result in the Congestion tone application
; being called. They'll get bored and hang up eventually.
;

exten => _.,1,Congestion

[from-sip]

;
; If the number dialed by the calling party was "2000", then
; Dial the user "2000" via the SIP channel driver. Let the number
; ring for 20 seconds, and if no answer, proceed to priority 2.
; If the number gives a "busy" result, then jump to priority 102
;

exten => 2000,1,Dial(SIP/2000,20)

;
; Priority 2 send the caller to voicemail, and gives the "u"navailable
; message for user 2000, as recorded previously. The only way out
; of voicemail in this instance is to hang up, so we have reached
; the end of our priority list.
;

exten => 2000,2,Voicemail(u2000)

;
; If the Dialed number in priority 1 above results in
; a "busy" code, then Dial will jump to 101 + (current priority)
; which in our case will be 101+1=102. This +101 jump is built
; into Asterisk and does not need to be defined.
;
```

```
exten => 2000,102,Voicemail(b2000)
exten => 2000,103,Hangup

;
; Now, what if the number dialed was "2001"?
;

exten => 2001,1,Dial(SIP/2001,20)
exten => 2001,2,Voicemail(u2001)
exten => 2001,102,Voicemail(b2001)
exten => 2001,103,Hangup

;
; Define a way so that users can dial a number to reach
; voicemail. Call the VoicemailMain application with the
; number of the caller already passed as a variable, so
; all the user needs to do is type in the password.
;

exten => 2999,1,VoicemailMain(${CALLERIDNUM})
```

Now, we're almost ready to go. Actually, we've completed everything that is required for two phones to call each other, but we still need to assemble the configuration files that will know how to save messages in case a line is busy or "unavailable." Note that Asterisk treats phones that are turned off or are otherwise not registered as "Busy" and not "Unavailable" -- the status of "Unavailable" usually refers to a situation where nobody has answered the phone in the given number of seconds.

Before we can use our voicemail system, we need to create empty voicemail boxes for each user. This is done using the script located at */usr/src/asterisk/addmailbox*, which is simply a small shell script that creates a directory and puts some default greetings into place. Run `addmailbox` twice, specifying "2000" and "2001" as the mailboxes that you wish to create.

voicemail.conf

```
[general]

format=wav

[local]

;
; format: password, name, email address for attached voicemail msgs
;

2000 => 4321,John Whorfin,jwhorfin@planet10.com
2001 => 8383,Sidney Zweibel,newjersey@banzaiinstitute.com
```

Testing: Here We Go!

The user on extension 2000 should be able to dial 2001 and the other line will ring. As you watch the console, you should see a flurry of messages showing up as you dial, and after

you hang up. If you go immediately to the "Busy" message, it's probably because your phone client hasn't registered with the Asterisk server -- make sure it's sending server registry statements, as these are the "heartbeats" that Asterisk uses to ensure that a remote client is available for inbound calls. If you have your registry interval set to more than 15 seconds, you'll need to wait at least that long for the SIP client to register (once Asterisk is started) before calls can be sent to that phone.

Try leaving voicemail for extension 2000. From extension 2000, dial 2999 and you should hear a prompt for the password -- you won't need to type in the extension for which you are trying to retrieve mail, as it is automatically entered with the addition of the `${CALLERIDNUM}` variable as an argument to the `VoiceMailMain` line -- this types in the correct extension automatically. Extension 2999 has a password of 4321, so type in "4321#" -- the # sign is equivalent to "enter." Then follow the prompts to retrieve the mail. You can also change the greetings on the line -- experiment a bit with the different options within voicemail.

Hopefully this all worked for you. Often, getting the first implementation up and running is a process of trial-and-error, and the added complexity of the SIP clients can make this an exhaustive reduction in variables. Feel free to ask the asterisk-users mailing list any specific questions you might have after reviewing the archives -- both resources are very useful to the beginner.

Debugging

SIP is a bit tricky to start out with. I would strongly suggest that your Asterisk server and both phones are on the same network segment when you begin your testing; having one or more devices processed through a firewall or NAT will cause you quite a few ulcers, and will almost certainly not work during your testing.

Make sure your SIP phones are sending correct `REGISTER` statements to the server -- without valid registration, the Asterisk process will not know where to send a call destined for that extension. Try `sip show peers` to see if the IP address of your phone shows up and is valid.

If SIP registration seems to be the problem, you can try removing the `secret=` lines and specifying an IP address of the phone in the `host=` line. This will lock the phones onto specific addresses and remove any registration issues you might be having.

I have found the `tethereal` tool (part of the `ethereal` network sniffer toolset) to be immensely valuable. `tethereal -n ip and port 5060` will debug SIP packets in a human-readable form, and should show you what is happening on the wire when testing.

Another important debugging technique is to run asterisk in "full debug mode." This is done with `asterisk -vvvvvgcd` and puts all possible debugging information on your console. Also make sure that your SIP client is using the G.711 codec (either `alaw` or `ulaw`) as that is a codec that is known to work with Asterisk.

Additional Features

To list Asterisk's full feature set would take quite a while, as it is just as much a toolkit as a set of applications. Some things I've implemented for myself and other customers include:

- **Telemarketer block:** Forces callers without caller ID to enter valid info, or hangs up.
- **Zapateller:** Used with telemarketer block -- plays three-tone "disconnected number" for callers with no caller ID, which auto-removes from many phone spam networks.
- **Dual ring:** Phones at the office and at home ring at the same time, during certain hours.
- **PBX long distance gateway:** PRI interface to Asterisk box, then low-priced ITSP to get better LD rates using existing PBX.
- **IVR interface:** Use as an inbound call center pre-processor.
- **International toll avoidance:** One in the HQ in each continent allows for centrex-style dialing over the Internet.
- **Voicemail:** Stand-alone voicemail server interfaces with existing PBX -- low-cost solution to otherwise expensive upgrade.
- **Monitoring system:** Intelligent dial-out platform that can verbalize network problems and take actions based on IVR.
- **NAT traversal:** Asterisk can work with SIP clients behind NATs with no additional software (see `nat=1` in *sip.conf*).

Digium makes a low-cost ($100) analog FXO (Foreign Exchange Office -- meaning, it accepts a wire that has dialtone on it) PCI card that can be used to connect your home or office phone line into Asterisk. Add to this a Cisco ATA-186 (~$150) for two lines of FXS (handsets) use, and you can have a mini-PBX in your house with extremely powerful features for about $250, plus the cost of a reasonably decent old PC that you may have laying around. I've had acceptable experiences running with a 400mhz PIII with two lines. I've found that putting one of these systems together with an FXO card is a fast way to convince others that OSS VoIP platforms are ready to be examined as a serious possibility for the office environment, since one can attach the machine directly to the existing PBX (assuming you can get an analog line off of your PBX system).

More complex Asterisk configurations and some other SIP-related examples can be found on my web site.

Other Projects and Resources

Servers/Software

There are several SIP implementations that are OSS, but they are primarily what are known as "call proxies" instead of more full-featured PBX applications. This means that they function only to connect two endpoints together, and are basically just large, fast, directory servers. Examples of SIP Proxies are ser and Vocal. There are also other open source PBX

projects like Bayonne and OpenPBX, which have slightly different feature sets than Asterisk.

Long Distance/Free Services

There are also now retail vendors of SIP long distance service, which are called Internet Telephony Service Providers (ITSPs.) Vendors such as iconnecthere (formerly Deltathree) have rates in the $.01-per-minute category anywhere in the U.S., and I expect that some searching will find similar vendors in Europe and Asia. There are also no-cost solutions like Free World Dialup that are limited to calling users via IP only, but you can mix/match dialplans so you can reach all destinations from the same phone, if that phone is routed through Asterisk. There are other SIP long distance providers, but to my knowledge all of the others (Vonage, Packet8) require you to use their equipment, thus making use with Asterisk impossible.

SIP Hardware

Cisco ATA-186	(About $150 new)	Supplies two FXS ports (standard phones plug into FXS ports)
Cisco 7960/7940	(About $300 used)	Deskset with XML-programmable six-line LCD
Digium	($100-$500 new)	Variety of PCI cards for direct telephony connections to Asterisk servers
Pingtel	(About $550 new)	Deskset with LCD
SNOM 200	(About $290 new)	Deskset with LCD
Grandstream	(About $85 new)	Deskset with small LCD

SIP Software/Hardware

SIP Product List: Lists of hardphones, softphones, and other servers

IPTel.org: Lists of hardphones, softphones, and other servers

John Todd is a networking and VoIP consultant, specializing in Asterisk implementations.