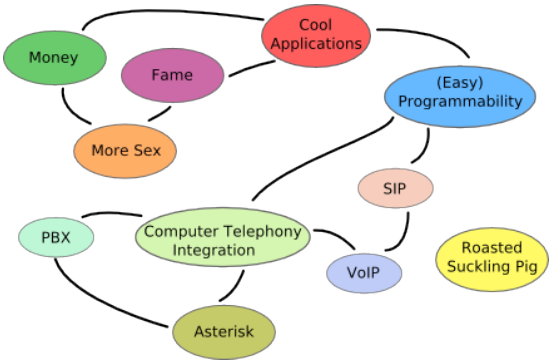
Introduction to Telephony Application Development Using Asterisk, Asterisk-Java, and SIP

by Cokorda Raka Angga Jananuraga

Putu Sutawijaya – Gerak Bagai Badai (Stormlike Movements)

I racked my minds, no gems to be found, for an impressive opening line....
So, out of frustration I write...

Frankly, I have no idea for the opening paragraphs of this booklet. All I have is a set of concepts / ideas roaming the empty space in my head.... Well, I'll just let them all out now.



Raka's Mind Map

This booklet is about all of them and none of them in particular (*huh?*). I mean, it touches the subjects drawn above to various levels of detail. Some are mentioned briefly, and some get quite an extensive coverage (for an introduction). I use a program that I wrote as a vehicle [*Res. 1*], and I'll start my explanation from the objectives (the requirements of the program). It reflects my learning style that usually starts with "*what? tell me*" , followed by "*how? show me*", and ends up in "*why? theories please*".

Without further ado: In this article I'm going to introduce you to techniques, libraries, and tools for developing telephonyservices. We'll use all open source stuffs here (*cool, huh!?*), like Asterisk, Asterisk-Java Library, Ethereal, SIPp, and NIST JAIN SIP library.

Audience

Well, I can think of at least 2 groups of people. First: those who've never been in touch with telephony and CTI stuffs. I was in that group. But once I got my hands on Asterisk, an open source PBX, I got excited. It was like: "Wow, kewl, I can do stuffs with this thing... quite easily!". I hope after reading this article, they'll find telephony application development interesting, and would like to give it a try.

The second group is: those who've been in this domain for some time, have heard of the word Asterisk and SIP buzzing around them, but not quite sure where and how to start making something out of them. I hope this article can be somekind of a guidemap.

I started from zero, I think. I studied the SIP specification [*Ref. 5*], some relevant books, installed and played around with Asterisk and related stuffs, scrolled through lots of packets captured using Ethereal, etc. I enjoyed the process, and it worked quite well for me. At least I think now I'm sufficiently oriented for my further endeavor. I'd like to share that approach with you through this article.

By the way: if you're looking for the table of contents, it's at the end of this booklet.

All You Need Is...

love..., *and* several softwares! Here's the list of things that you need to have in your computer:

✪ Asterisk, that will serve as our backend. Among other things, it switches incoming calls to registered extensions. In this occasion, we're going to monitor the *events* happening in an Asterisk so that we know when a call is started and ended.

You wouldn't find detailed information about how to install, run, and administer an Asterisk box here. References such as [*Ref. 2*] and [*Ref. 6*] do that job a lot better. However, I will give you an explanation just enough to quickly setup your asterisk box and configuring it in order to make our program works.

✪ Asterisk-Java, a programming library that we'll use to interact with Asterisk from the program that we're going to develop with Java. The kind of interactions possible are:

a) Listening to events sent by Asterisk.
   Asterisk send all *events* happening in it to all the registered listeners. It's a socket based communication. Our program will open a socket, through which it registers itself to Asterisk (via a manager interface) so that *events* generated by Asterisk will arrive on that socket.
b) Sending *actions* to Asterisk through the connection between our program and the manager interface. We'll receive a *response* for an action that we send to Asterisk.
c) Sending *commands* to Asterisk through the so called "*agi channel*". We'll get a reply for a command that we send to Asterisk.

> *Sidenote*. The distinction between *command* and *action* is not yet clear for me. I guess action is more general, in the sense that we can send an action to Asterisk any time. On the other hand, we send command through an *agi channel*, that's only available in the context of an execution of an *AGI script*. Execution of an AGI script, in turn, is triggered by a call made through the Asterisk box. So, I guess, the scope of a command is limited to a particular call.

✪ SIPp. It is a program that we can use to generate SIP signaling traffic. Typically it is used to *stress-test* a SIP-based telecommunication system. In our case, we need to make sure that our program performs correctly in a more-or-less realistic situation (calls coming in and out simultaneously). Therefore, we use SIPp to simulate multiple calls.

✪ Ethereal, an open source GUI network protocol analyzer. It lets you interactively browse packet data from a live network or from a previously saved capture file. As I told you earlier, I walked my way from the bottom up in understanding these stuffs. Hmm..., well, not exactly. Of course, first I read some references about how things work (for example: communication between SIP endpoints), then I moved on to API documentations to get some practical insights. But maybe because I'm a skeptic, I felt the need to really see what's being exchanged through the wire, the sequence, etc. For that reason, I opened my Ethereal, start sniffing, and try to confirm what I've read.

✪ In our program, particularly whenever creation of conference room is detected, we need to sneak in a dummy participant into that room for a reason that will be explained later. That dummy participant

basically is a software agent that dials in to the conference, and stays there until everybody else leave the room. I've decided to implement the establishment (and tear-down) of the call session between that software agent and Asterisk using SIP.

Sidenote. Actually we have at least 2 choices of call signalling protocol with an Asterisk box: IAX2 and SIP. I choose SIP for two reasons:

✔ At the time of writing there was no IAX programming library in Java, at least not the open sourced one. In this occasion I limit the choice only to open source softwares and libraries.

✔ The purpose of this article is to give the readers somekind of *birdview* over the world of telephony application development, using emerging open standards and open source softwares. We're not going for the *depth* right now, but instead for the *breadth*. Following the "*more is more*" principle, I bring on SIP to the table. The talk about "*which one is more suitable for particular case, SIP or IAX?*" is better left for another article, I guess.

SIP is interesting and promising, I can tell you. If you check big telecommunication vendors' websites, you'll find that they all have SIP related stuffs in their product line (be it software or hardware). A scoop of knowledge on SIP might be beneficial for you to play in that market.

We need JAIN-SIP. It is a Java-standard interface to SIP signaling stack (whatever that means :). From practical point of view, our Java program constructs, sends, receives, and reads SIP messages using classes and interfaces specified in JAIN-SIP. There's an open source implementation of JAIN-SIP specification from National Institute of Standard and Testing [*Res. 7*].

Does JAIN-SIP vaguely remind you of JDBC...? Exactly!

Mission Briefing

First, let's define an objective. We want to create something like in *Fig. 1*. Let's call it *CallWatcher*.

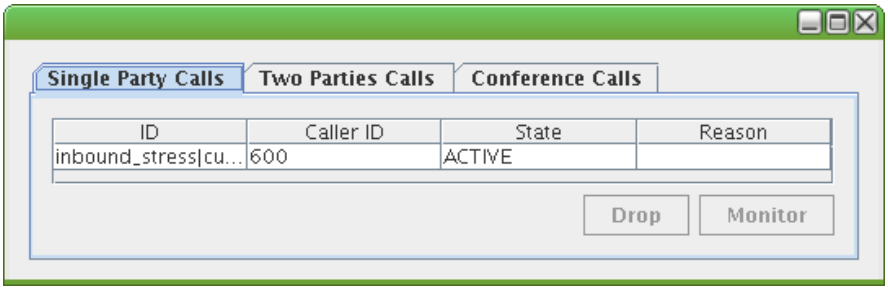


***Fig. 1**. CallWatcher's main window– single party calls*

The application has 3 panels:

- *Single Party Calls*
  Calls which are answered directly by Asterisk should show up in this panel (*Fig. 1*). For example: in our *dialplan*, we specify that calls coming in to extension 600 will be answered by an Asterisk's built-in IVR that simply echoes whatever the caller is saying. That call should show up in the *Single Party Calls* panel.
- *Two Parties Calls*
  Calls which are terminated at another *user agent*[1] (UA) outside of Asterisk should show up in this panel (*Fig. 2*). For example: we have configured in our dialplan that incoming call to extension 101 must be switched to *agent #01.* That call should show up in the *Two Parties Calls* panels. In this panel, the id of the caller – that could be her telephone number – is shown in the *Caller ID* column. The dialed extension is shown in the *Called ID* column.

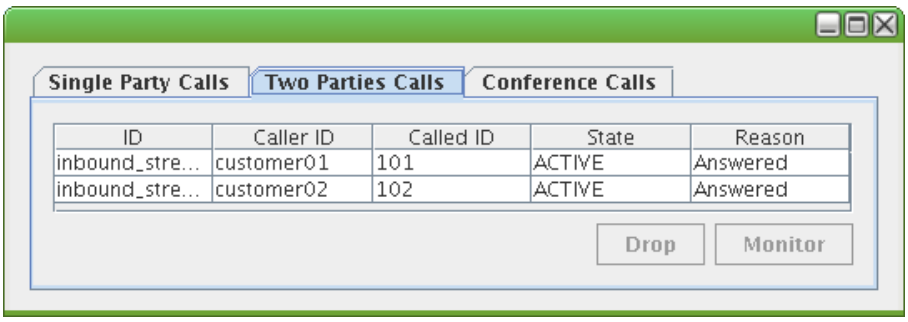


***Fig. 2**. CallWatcher's main window – two parties calls*

- *Conference Calls*:
  Conference calls will show up in this panel (*Fig. 3*). The number of participants in the conference is shown in the *Participants* column. For example: we have configured that those who dials the extension 5400 will be joined to conference room #400.

  A conference room with that number will be automatically created in case no such room currently exists. As a result, we should see a new entry (row) is added to the *Conference Calls*

1 Something that originates calls / where a call is terminated. A (SIP) phone is an example of user agent.

panel. Otherwise (if the room exists), the value of *Participant* column in the corresponding entry in the panel should be incremented by one. Similarly, whenever someone leaves a conference the the value of Participant column should be decremented by one.

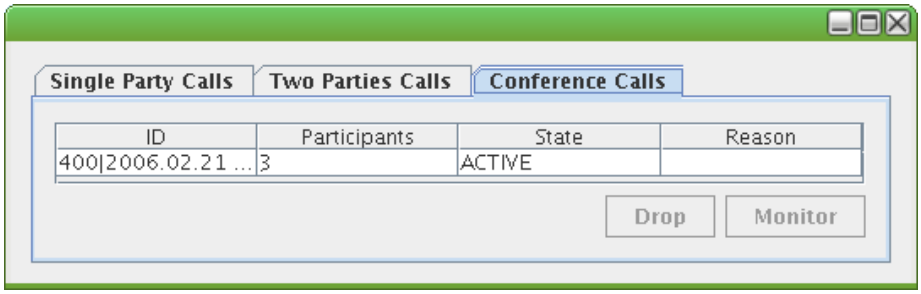


***Fig. 3**. CallWatcher's main window – conference calls*

On all the panels there is a column named *State*. That columns shows the state of the call. A call can be in any of this state:

- IDLE : initial state
- CONNECTING : when Asterisk is able to locate the other party – to which the call should be connected, in the case of *Two Parties Call* – and is in the process of connecting both parties.
- ACTIVE : when the call is active (involved parties are talking to each other).
- INVALID : when the call has been torn-down. In the case of *Conference Call*, it is invalid when everyone (including the dummy participant) has left the conference room. The conference room is also automatically disposed.

Additionaly, user of *CallWatcher* can perform an action on an active call. She does that by selecting an active call, and click the button that represents the action she wants to execute. As you can see in *Fig. 1* through *Fig. 3*, there are two possible actions:

- *Drop* : this will terminate the selected call. In the case of conference call, everyone in the room will be kicked out first.
- *Monitor* : this will instruct Asterisk to record the selected call.
  In the case of *Single Party* call, it's clear that we're interested in what the caller said and heard. So we'll tap in the channel between the caller and the Asterisk box.

  In the case of *Two Parties* call, we're interested in what either the caller or the callee said *and* heard. There's no difference, but you have to pick one (I pick the caller).

  In the case of *Conference* call, we're interested in what the dummy participant “heard”. Why? Because the dummy participant was there during the whole conference session. Other participants might left and rejoined, thus missed some parts of the discussion. That's not the case with the dummy participant. For that reason we'll tap in the channel between the dummy participant and the Asterisk box.

Well, that's it. Quite rough a requirement. We are going to refine our requirements as we move on with the design and – further – with the implementation.

The Design..., a Static View

The first step in designing a system, particularly system developed in an *object-oriented* way, is understanding the domain. This involves identifying the concepts within the domain, how they're related to each other, and how they work together to accomplish a set of objectives. The outcome of that process is something called *domain model*.

A good place to start is an existing standard, like JTAPI (*Java Telephony API*). I adopted several concepts that I learned from JTAPI [*Res. 6*]. Overview of the design is shown in *Fig. 4*. Please keep in mind, we're not implementing JTAPI here (*it's not easy*). The model that I present here works in our specific situation, but might not work in other / more general situations. It is something simple enough to understand, implement, and explain..., but still conceptually correct (or at least makes enough sense).

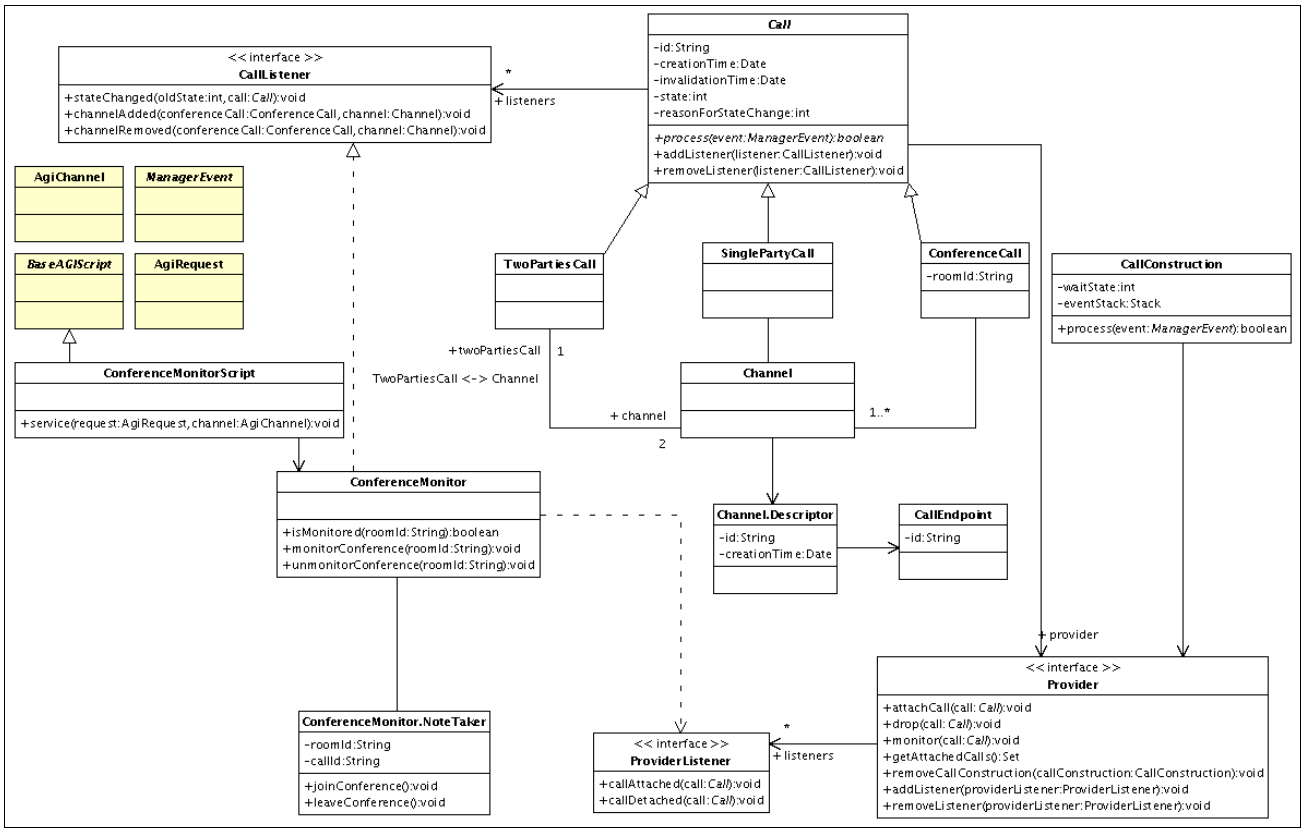


***Fig. 4**. Class diagram*

Let's start with the central concept: ✪ *Call.* It represents a telephone call, the information flowing between the service provider and the call participants[2]. The information flow between a call participant and the provider through an logical conduit that we call ✪ *Channel*. Each participant is represented by its address; we call it ✪ *CallEndpoint* in our model.

Another very important concept in our model is the ✪ *Provider*. It's an abstraction of telephony service provider software. Let's take a look at some of the things that we can do with a *Provider*:

2 Definition is borrowed from one of JTAPI Whitepapers (*The Java Telephony API – An Overview*).

- drop(Call call);
- monitor(Call call);

Abstraction effectively means that the one who uses it doesn't really know (nor care) *how* those operations are carried out. All she needs to know is the effect of the operation, which in the case of dropping a call (for example) is: all the participants will be disconnected from the call and the call will become invalid. We simply delegate that dirty job to our underlying telephony system..., Asterisk.

Sidenote. Please notice that in our model *Provider* is an *interface*, not a *class*. There's actually a concrete class – not shown in the diagram – that implements *Provider*. Its name is ✪ *AsteriskProvider*. It contains the logics of communicating with Asterisk.

Eventhough we have no other implementation of *Provider*, and neither do we have any plan to create another implementation of *Provider* (that talks with PBX other than Asterisk), still it's a good idea to keep that *Provider* interface. I resist the tempation (or a harrasment from a friend of mine) to convert *Provider* into a *class* and move all the logics from *AsteriskProvider* into *Provider* (and consequently purge *AsteriskProvider*).

The general rule that I apply is: an abstraction of an external system – such as Asterisk – should be *interface-ified*. It eases the testing because then we can do somekind of scenario based testings, even without the existence of that external system. This *interface based programming* gets along very well with *test driven development* (particularly using *mock object* approach).

For a comparison, take a look at the following diagram that I stole from JTAPI Whitepaper [*Ref. 1*]. The *Channel* in our model is more or less equivalent to ✪ *Connection* in JTAPI's model. The *CallEndpoint* in our model is a ripped off version of ✪ *Address* in JTAPI's model.

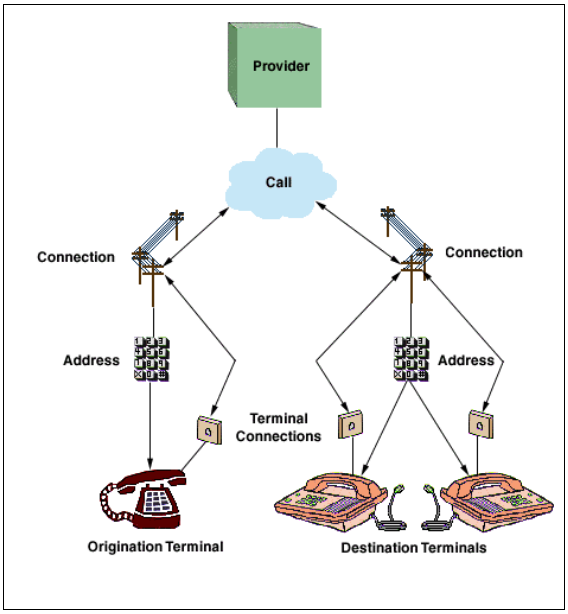


***Fig. 5**. JTAPI's call model*

Let's go back to the telephone *Call*. We have 3 types of call[3]:

- Single Party Call
- Two Parties Call
- Conference Call

Single Party Call

Single Party Call is a call with only one participant (thus one channel). It happens when someone calls an extension in Asterisk which is configured to switch the call to an Asterisk's built-in IVR. There is a flow of information *only* between the caller and the Asterisk. The class ✪ *SinglePartyCall* in our model represents this concept.

```
    -- Executing Answer("SIP/customer01-3ee8", "") in new stack
    -- Executing Echo("SIP/customer01-3ee8", "") in new stack

*CLI> show channels
Channel              Location             State   Application(Data)
SIP/customer01-3ee8  600@inbound_stress:2 Up      Echo()
1 active channel
1 active call
```

Fig. 6. Asterisk CLI – single party call

When I called the extension 600 – mapped to *Echo* application – the log that I observed on Asterisk's Command Line Interface (CLI) was like the one shown in *Fig. 6*. That log confirmed me that there are indeed cases when a call has only one channel.

Two Parties Call

Two Parties Call is a call with two participants – *two channels* – just like a *normal* phone call that most of us would expect :). The class ✪ *TwoPartiesCall* in our model stands for this concept.

In our *asterisk dialplan*, a call to certain numbers will be switched to another party outside of Asterisk. For example if someone calls the extension 101, she will be connected to someone named “Agent 1”. In that case the participants of the call are: Mrs. Jane Doe (the caller) and Agent 1. Take a look at *Fig. 7* (so that you're convinced :):

```
    -- Executing Dial("SIP/customer01-4e53", "SIP/agent01") in new stack
    -- Called agent01
    -- SIP/agent01-3aeb is ringing
    -- SIP/agent01-3aeb answered SIP/customer01-4e53
    -- Attempting native bridge of SIP/customer01-4e53 and SIP/agent01-3aeb

*CLI> show channels
Channel              Location             State   Application(Data)
SIP/agent01-3aeb     (None)               Up      Bridged Call(SIP/customer01-4e
SIP/customer01-4e53  101@inbound_stress:1 Up      Dial(SIP/agent01)
2 active channels
1 active call
```

Fig. 7. Asterisk CLI – two parties call

3 Warning: they are purely product of my imagination. You will not find them in JTAPI specification.

Conference Call

Conference Call is a call that can have more than two participants (therefore more than two channels). The class ✪ *ConferenceCall* in our model corresponds to this concept.

Asterisk has a built-in application for conferencing. Its name is *MeetMe*. There's a slight difference between Asterisk's model and ours (regarding conference). Take a look at *Fig. 8*.

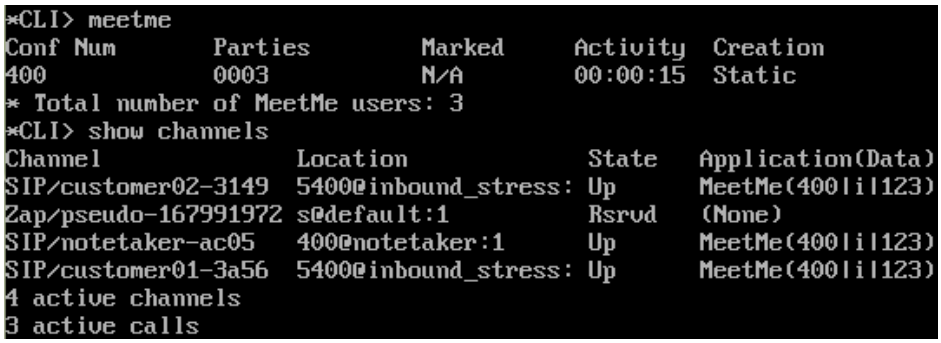


Fig. 8. Asterisk CLI – conference call

Looks like in Asterisk's model a conference is not a really call. It's more like a collection of single party calls which are destined to an extension which is configured to switch the call to a particular conference room, where the information from the callers are mixed and then distributed to every participants.

I decided not to follow Asterisk's model. I wanted to be as consistent as possible with the model that I learned in JTAPI. None of the subtypes of *javax.telephony.Call* in JTAPI has a set of calls as its property. Instead, an instance of *javax.telephony.Call* has a set of instances of *javax.telephony.Connection*. Therefore, in our model ✪ *ConferenceCall* is modeled to have a set of instances of *Channel*.

The Design..., the Dynamics

Let's start with a simple question: *"Who creates instances of Calls?"*. Let's see... a call is actually a concept within the telephony system. I mean, a call is established in the telephony system. Take a look again at *Fig. 6* to *Fig. 8*. It's understandable if someone says *"Ah, currently there are 5 calls* ***going on in*** *the Asterisk box"* after seeing the report in the Asterisk CLI that resulted from an execution of *"show channels"* command.

In our model the telephony system is represented by an instance of *Provider*. Therefore it makes sense to assign the responsibility of instantiating *Call*s to *Provider*. Currently there's no mechanism that lets users of our program to instruct *Provider* to create a call (call creation is not *on-demand*). Instead, calls are created automatically within the *Provider* on notifications received from the underlying telephony system.

Instance of *Provider* is a stateful object – it holds instances of *Call*. Because of memory usage consideration, we decided to make the *Provider* holds only the *active* calls. We refer to those calls as *attached calls*[4]. To get the list of calls attached to a provider simply invoke the method ✪ *getAttachedCalls()* on it.

> A boy cried: *"but..., I also want to be notified when a Call is created, and – at the same time – obtain the reference to that instance of Call"*.

So we invented an interface named ✪ *ProviderListener* to calm that boy. It has 2 methods:

- callAttached(Call call)
- callDetached(Call call)

Accordingly, we add methods into *Provider* to register and unregister a listener to it. Those methods are:

- addListener(ProviderListener listener)
- removeListener(ProviderListener listener)

In our program the class that controls the user interfaces – *net.raka.agiexp.gui.Main* – implements *ProviderListener*. Whenever a new instance of *Call* is attached to the provider that it listens to, a new row – that shows information about the call – will be added to the table in the appropriate panel (depending on the concrete type of the call).

Call construction is a multi-steps process. It's not like the Asterisk simply notifies our Java application *once* whenever a new call is created within the Asterisk, and passes along all the information needed to create an instance of *Call* in our Java program. The library that we're using to communicate with Asterisk – *Asterisk-Java* – works at the lower level. What the library passes to our Java application are notifications about the events that take place in the Asterisk, in the form of instances of ✪ *ManagerEvent*.

4 When a call becomes invalid it will be *detached* from the provider to which it was previously attached.

Even something as simple as calling the extension 600 – that is mapped to the *Echo* application – will trigger a sequence of manager events as shown in *Fig. 8*. Loosely speaking, that sequence of events builds up the single party call.

```
class: net.sf.asterisk.manager.event.NewChannelEvent
dateReceived: Sat Feb 25 02:19:42 CST 2006
getPrivilege: call,all
callerId: customer01
callerIdName: customer01
channel: SIP/customer01-5afa
state: Ring
uniqueId: 1140855582.4                                  1
--------------------------------------
class: net.sf.asterisk.manager.event.NewExtenEvent
dateReceived: Sat Feb 25 02:19:42 CST 2006
getPrivilege: call,all
appData: null
application: Answer
channel: SIP/customer01-5afa
context: inbound_stress
extension: 600
priority: 1
uniqueId: 1140855582.4                                  2
--------------------------------------
class: net.sf.asterisk.manager.event.NewStateEvent
dateReceived: Sat Feb 25 02:19:42 CST 2006
getPrivilege: call,all
callerId: customer01
callerIdName: customer01
channel: SIP/customer01-5afa
state: Up
uniqueId: 1140855582.4                                  3
--------------------------------------
class: net.sf.asterisk.manager.event.NewExtenEvent
dateReceived: Sat Feb 25 02:19:42 CST 2006
getPrivilege: call,all
appData: null
application: Echo
channel: SIP/customer01-5afa
context: inbound_stress
extension: 600
priority: 2
uniqueId: 1140855582.4                                  4
```

Fig. 8. Manager events generated when 600 is dialed

A *NewChannelEvent* – like the first event in *Fig. 8* – gives an indication to our application that someone / something is connecting to the *Asterisk*. But at that point we can not figure out if it is a single party call, two parties call, or conference call. More information is needed, that is the value of the next events in the sequence. Only when the sequence of events turn out to be like the one in *Fig. 8*, for example, we can conclude that a new single party call has been created.

Analysis of sequence of manager events will be discussed thoroughly in the *implementation* section. The point here is: we need something that keeps track of the events, and instantiates a *Call* at an appropriate point in time. In our model that "something" is ✪ *CallConstruction*. You can think of an instance of *CallConstruction* as a *finite state machine*. You feed manager events to it, sequentially. If it accepts the event (for processing) then it will switch its state, depending on the type and the properties

of the event. When it arrives to a specific state – lets call it *"call instantiation state"* – it will instantiate the correct type *Call*.

So, we have a manager event being passed around. First off, a manager event is raised by Asterisk. Then it travels through a TCP socket to *AsteriskProvider* that acts as ✪ *ManagerEventHandler*[5] that has only one method, namely ✪ *handleEvent(ManagerEvent event)*. See *arrow #1* in *Fig. 9*.

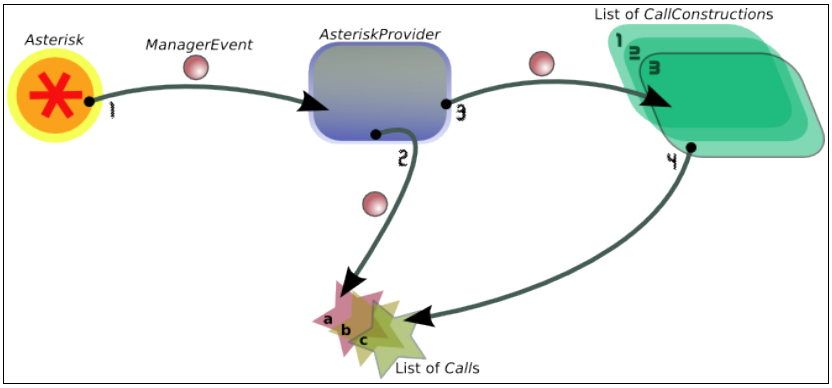


Fig. 9. Call construction

AsteriskProvider will first iterate through the attached calls, to see if any of them would "accept" the manager event. It does that by calling the ✪ *process(ManagerEvent event)* on each instance of *Call* (see *arrow #2*). The method will return *true* if the call accepts the event, or *false* otherwise.

An instance of *Call* also keeps track of *ManagerEvent*s in order to be able to decide if it should update its state. Of course not all events passed in to it will be accepted by the *Call*. A call determines acceptability of an event based on its current state and the previous events that it has accepted.

If none of the calls accept the event then AsteriskProvider will iterate through the living *CallConstruction*s. Again, it calls a method named ✪ *process(ManagerEvent event)* on each instance of *CallConstruction* (see *arrow #3*). If none of them accepts the event then *AsteriskProvider* will inspect the type of the event. If it is a ✪ *NewChannelEvent* and the state of the channel is "Ring", then a new instance of *CallConstruction* will be created and added to the list of living *CallConstruction*s.

Otherwise – if there's a *CallConstruction* that accepts the event – *and* the event causes the *CallConstruction* to move on to the "call instantiation state", a new instance of *Call* will be created and attached to the *AsteriskProvider* (see *arrow #4*). Then, the *CallConstruction* will be removed from the list of living *CallConstruction*s.

Another concept needs explaining is the java interface ✪ *CallListener*. A call will notify its listeners of interesting events that happen in it. *CallListener* has the following method, among others:

- stateChanged(int oldState, Call call)

The class that controls the user interfaces in our program – *net.raka.agiexp.gui.Main* – also implements *CallListener*, so that it can update the display on the table of calls (each time any of the calls that it keeps track of changes its state).

5 An interface defined in the Asterisk-Java library that handles events received from an Asterisk server.

Setting Up the Environment

Let's set up our playground before jumping in to the fun part that is coding...., starting with the installation of Asterisk. Oh, did I tell you that we need to work on Linux OS? I'm using OpenSuse 10 at home, but Asterisk can run in (m)any flavor of Linux.

Asterisk

The installation is easy. Once you have it downloaded from http://www.asterisk.org just do the following steps:

1. Unpack that file (i.e.: *tar xzvf asterisk-1.2.4.tar.gz*)
2. Go to the directory where you unpacked the file
3. Type in the command line: *make*
4. Then type: *make install*

Now we can move on to the configuration stage. Asterisk will look for its configuration files under the directory */etc/asterisk/*. There are more than a dozen of configuration files (*mucho*), each of them controls particular aspect of an Asterisk server. If you're lazy (like me), you wouldn't want to write those files from the scratch. Rather, you'd simply type *make sample* in the command line. That will copy sample configuration files to */etc/asterisk*.

Among those configuration files, we will be concerned with only four of them:

- sip.conf
- extensions.conf
- manager.conf
- meetme.conf

And let's start with *sip.conf*....

sip.conf

Among other things, here we define the locally connected SIP phones, which can be of any of these types: *user*, *peer*, and *friend*.

- For a UA that can only place calls to Asterisk we set the type to *user*. The UA will be asked authenticate itself everytime it places a call. The authentication user name and the password sent by the UA (in response to the challenge) will be matched against the name of the entry and the password specified for that entry, respectively.
- For the UA that Asterisk can place calls to – in addition to receiving calls from – we set the type to *peer*. That UA is kind of trusted by Asterisk, meaning that it will not be asked for authentication (password) when *placing* a call.

  However, for the UA to be able to receive calls, Asterisk needs to know its location. Registration – by the UA – is a way of notifying Asterisk about its location. Only then Asterisk will ask for authentication user name and password.

- The last type, *friend*, is *user* + *peer* in one.

Most of the time I set the locally connected phones to *friend*, though. However, to let people in the office to place or receive a call to outside parties through another SIP server (e.g.: *FreeWorldDialup*), we define an entry of type *peer*.

Let's experiment a little. First, copy and paste the following lines to the end of your *sip.conf*.

```
[johndoe]
type=user
secret=p@55w0rD
host=dynamic
canreinvite=no

[goldenboy]
type=peer
secret=p@55w0rD
host=dynamic
canreinvite=no
```

(*And...?*) Let's read about *extensions.conf* before running anything :).

extensions.conf

This file contains the *dialplan* of an Asterisk box. It's a text file where we – in a simplest scenario – specifies something like “if the caller dials 103 then switch the call to Jeniffer the CEO's assistant”. The book [*Ref. 2*] succintly describes it as:

> *... the heart of any Asterisk system, as it defines how Asterisk handles inbound and outbound calls. In a nutshell, it consists of a list of instructions or steps that Asterisk will follow.*

Dialplan are broken into sections called *context*, which is a named group of extensions. Some of the contexts defined in the sample extensions.conf are: international, longdistance, local, default, and demo. Now let's take a look again at *sip.conf*. You should find, in the *general* section, something like in *Fig. 10*:

```
[general]
context=default                 ; Default context for incoming calls
;allowguest=no                  ; Allow or reject guest calls (default
is yes, this can also be set to 'osp'
```

Fig. 10. A portion of sip.conf

The highlited line says that: the calls which come through the SIP channel, from a caller / user whose context is unknown / undefined, will be put in the context named *default*. The *default* context in the sample *extension.conf* is defined the following way:

```
[default]
;
; By default we include the demo.  In a production system, you
; probably don't want to have the demo there.
;
include => demo
```

Fig. 11. The default context in the sample extensions.conf

Ok, it leads us to another context named *demo* (grr...). There are several extensions defined in the demo

context, with the extension 600 being one of them (see *Fig. 12*).

```
; Create an extension, 600, for evaulating echo latency.
;
exten => 600,1,Playback(demo-echotest)  ; Let them know what's going on
exten => 600,n,Echo                     ; Do the echo test
exten => 600,n,Playback(demo-echodone)  ; Let them know it's over
exten => 600,n,Goto(s,6)                ; Start over
```

Fig. 12. The extension 600 defined in the demo context

The user *johndoe* doesn't have its context defined (in *sip.conf*). Therefore the call from *johndoe* will be placed in *default* context. Furthermore if the extension called by *johndoe* is 600, then it will be (1) greeted with the playback of an audio file named *demo-echotest*, then (2) handled by an application named *Echo* (that simply echoes back whatever the caller says), and (3) greeted again with the playback of an audio file named *demo-echodone*. Let's prove it, by making a call.

Your first call

First, start the Asterisk by typing *asterisk* in the command line, followed by typing *asterisk -vvvvvvvvvr* to get into the Asterisk CLI (so that we can see what's going on). To exit from the CLI – without killing Asterisk's process – type *exit* in the CLI. To stop the Asterisk type *stop now*.

Next, let's configure your softphone. There are at least 2 free SIP softphones that runs on Linux: *KPhone* (open source) and *Xten-Xlite*. On this occasion we'll use Kphone. With the Kphone, first, you need to configure the identity that it will use when placing a call. Follow the steps displayed in *Fig. 13*. You have to set the “Host Part of SIP URL” to the IP address of your Asterisk box.

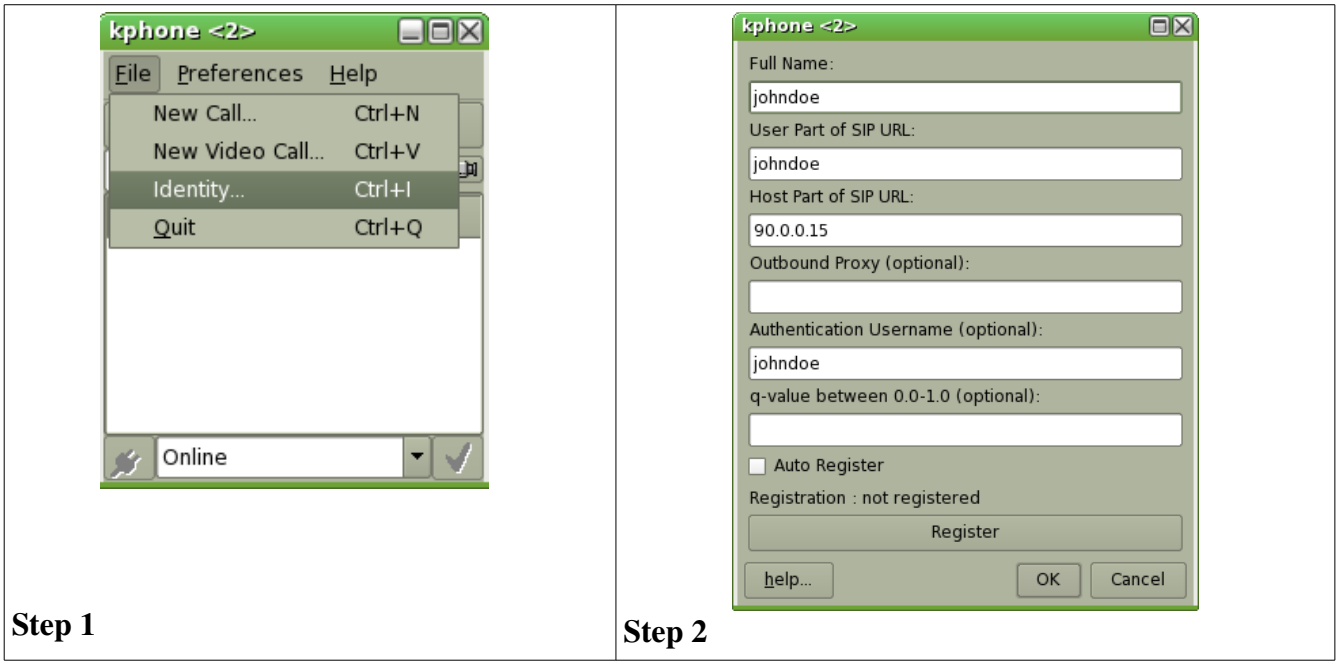


Fig. 13. KPhone identity configuration

Once you're done with *step 2*, click the *OK* button, then you will have to restart your Kphone (exit and rerun). To place a call, type the SIP URL of the destination – which in this case *600@90.0.0.15* – in the Kphone address bar (see *Fig. 14*), followed by hitting the *Enter* key[6]. When you're prompted for password type *p@55w0rD*. That's it..., *are you connected now*?

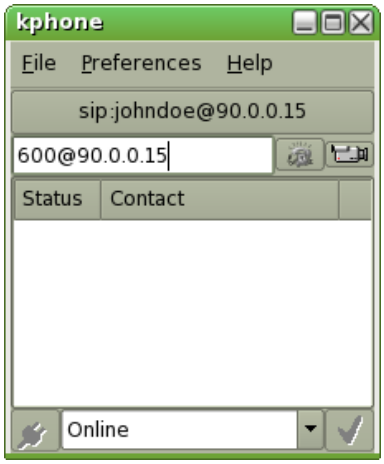


Fig. 14. KPhone

Your second call

Now let's try the following scenario: "*if someone (such as johndoe) calls the extension 666, he will be transferred to goldenboy...*". The first thing you'll have to do is registering the extension 666 to the *default* context. Copy the following lines...

```
exten => 666,1,Dial(SIP/goldenboy)
exten => 666,n,Hangup
```

... and paste them into *extensions.conf*, inside the definition of *default* context, such that your *extensions.conf* looks like *Fig. 15*.

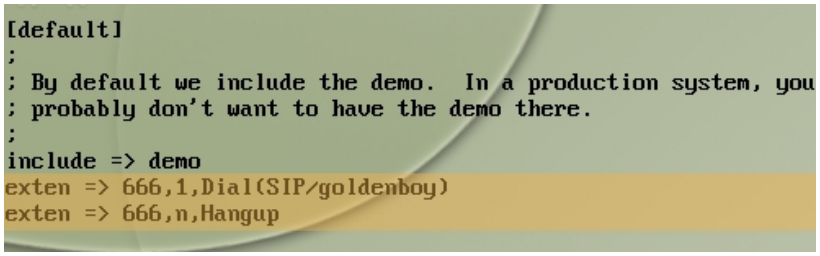


Fig. 15. Definition of default context, after the modification

Now you have to run another instance of Kphone for the peer *goldenboy*. If you have more than one machine, you're lucky, you just have to repeat the identity configuration steps explained above on the other machine, specifying *goldenboy* as the authentication user name. Otherwise you'll have to do two things:

1. Create another linux user on your machine, open a new instance of *Konsole*, login as that new user from that console, and run Kphone from the console. Follow the steps exemplified in *Fig. 16*.

6 Change the host part of the SIP URL to the IP address your Asterisk box.

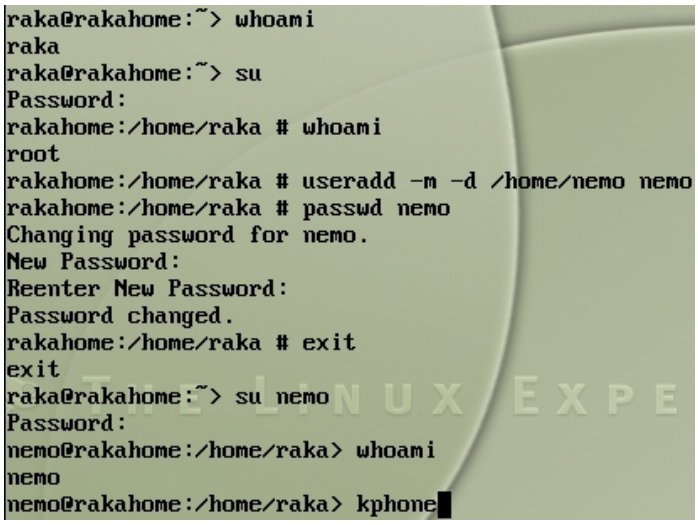


Fig. 16. Running another instance of Kphone (as user **nemo***)*

2. Do a little trick with the Kphone because by default KPhone will try to have an exclusive access to the soundcard during a phone call. The trick basically is configuring at least one of the instances of KPhone to use *null* device (*/dev/null*). Follow the steps exemplified in *Fig. 17*.

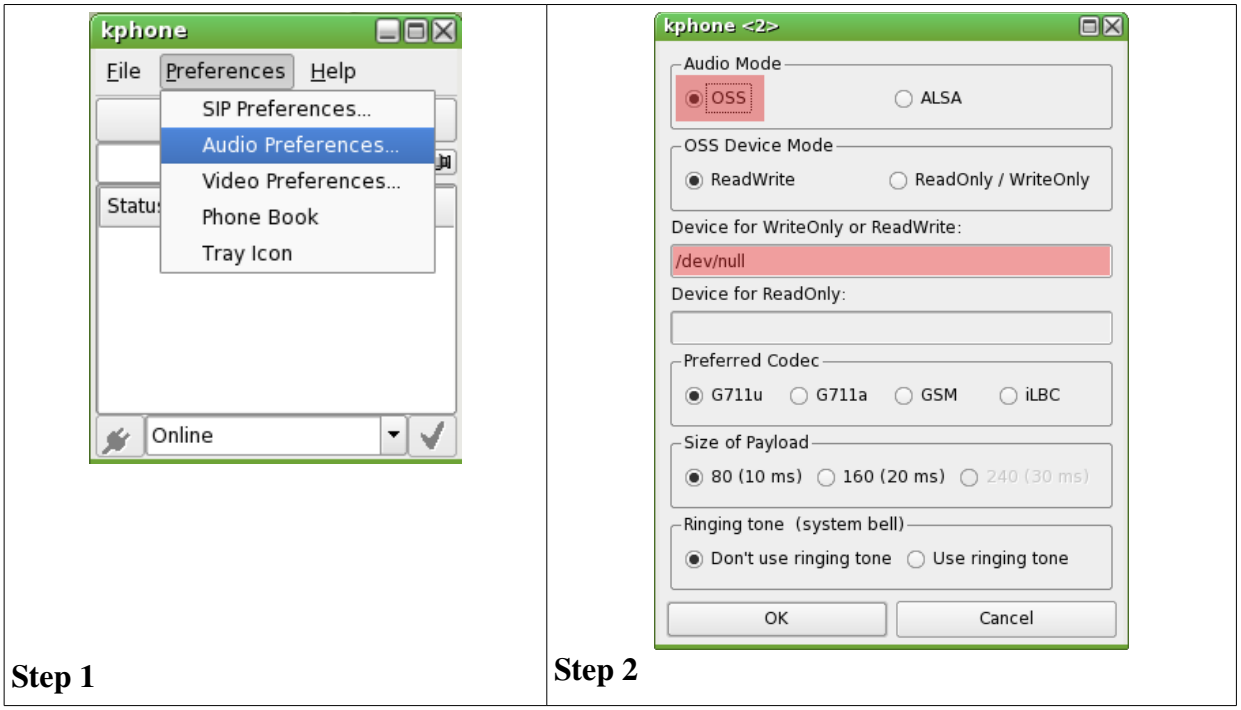
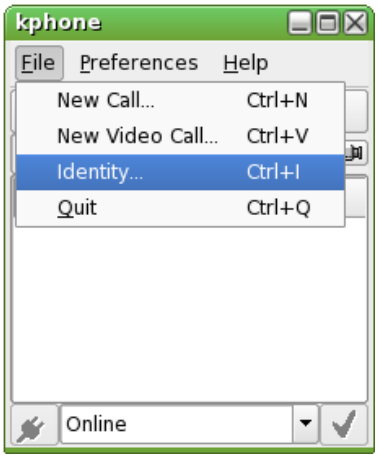
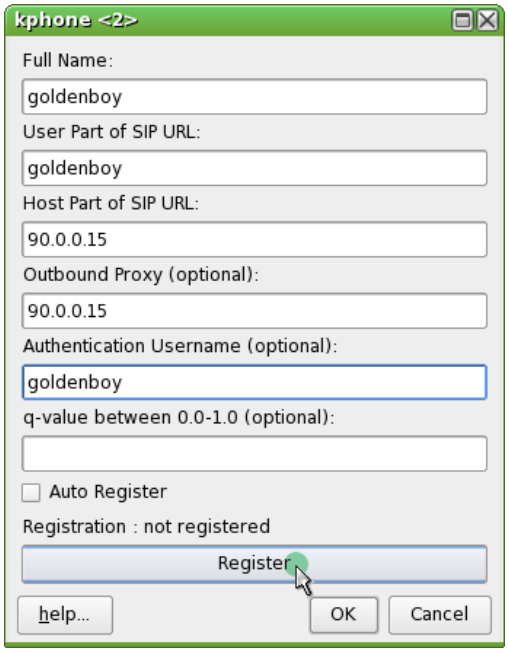


Fig. 17. Configuring KPhone to use /dev/null

In our scenario, this instance of KPhone is going to receive a call. Therefore it has to register itself to the Asterisk. Let's register the phone under the name *goldenboy*. Follow the steps shown in *Fig. 18*.



Step 1



Step 2

Fig. 18. Registering goldenboy

To verify if (location of) the phone has been registered in the Asterisk, go to Asterisk CLI and type *sip show peers*. From the example output of the command – see Fig. 19 – we can tell that the phone is located on the machine *90.0.0.15* and is bound to port *5062*.

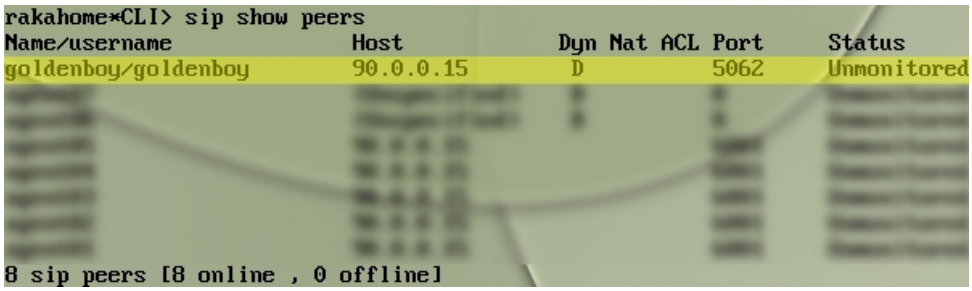


Fig. 19. Verifying registration of sip peer

At last, you can make your second call now. From the first instance of Kphone (*johndoe*), dial *666@90.0.0.15* (change the IP address). That's it..., *are you connected now?* That's all about the installation and configuration of Asterisk. I hope you didn't run into any troubles. Next, let's get dirty with the codes!

Implementation

Before anything else, I suggest that you download the source code of your application from [*Res. 1*]. Then you import the project into your Eclipse workspace and try to resolve the library dependency issues by yourself. Here's how the project structure looks on my Eclipse:

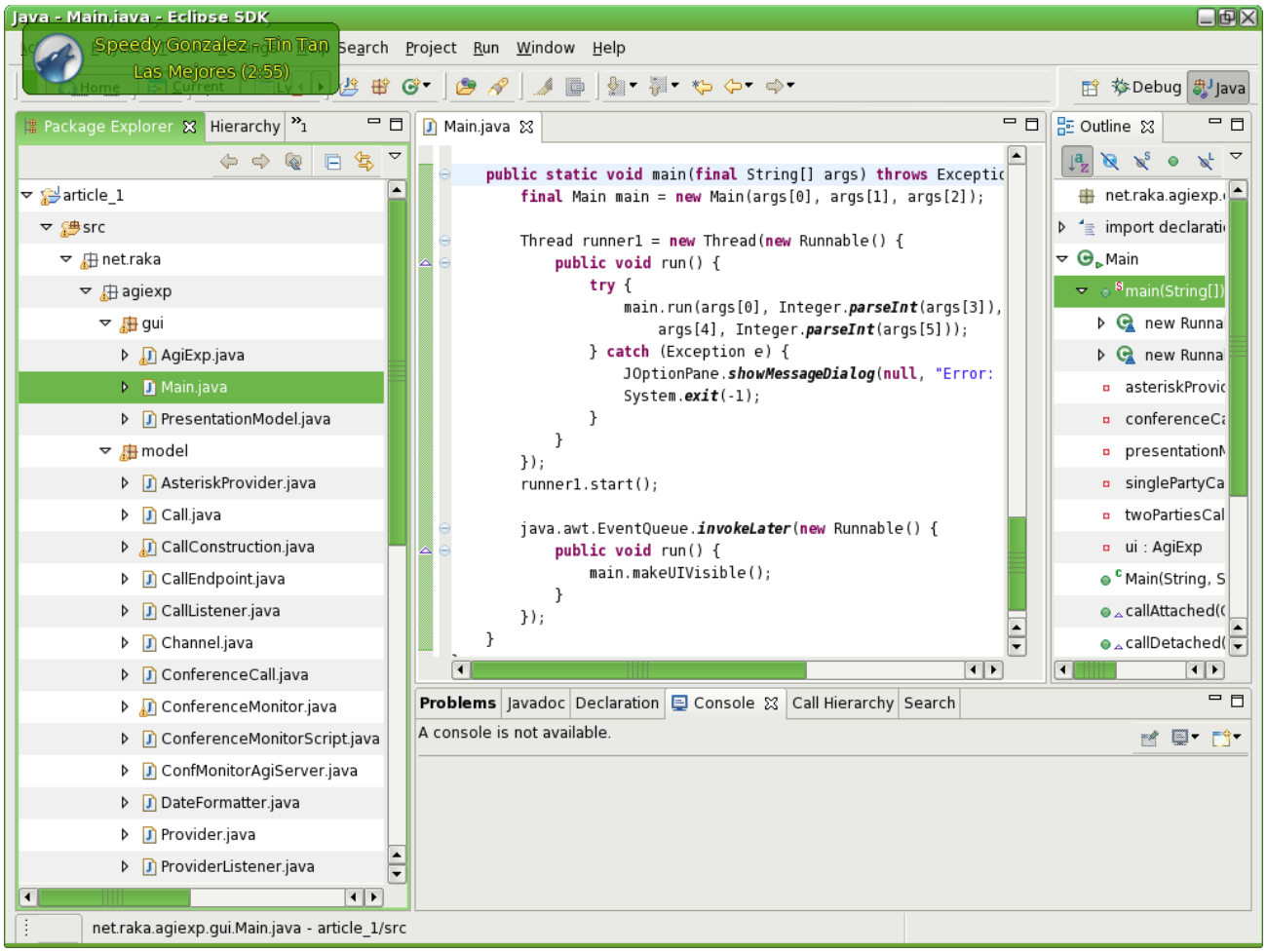


Fig. 20. Eclipse screenshot (... a teaser ;)

Let's begin our journey from the constructor of the *Main* class. There are three things that *Main* does when its being instantiated, they are: ❶ opening a connection to the Asterisk box, ❷ registering itself to the *AsteriskProvider* as one of its listeners, and ❸ registering *AsteriskProvider* as the handler of manager events received from the Asterisk box (through the connection established in step. 1). Please take a look at *Fig. 21* to see the corresponding lines of code.

```
ManagerConnectionFactory mgrConnFactory = new ManagerConnectionFactory();
ManagerConnection mgrConn = mgrConnFactory.getManagerConnection(asteriskIP, user, password);
asteriskProvider = new AsteriskProvider(mgrConn);
asteriskProvider.addListener(this); (2)
mgrConn.addEventHandler(asteriskProvider); (3)
mgrConn.login();
```

Fig. 21. Snippet from constructor of Main class

Now let's see how, roughly, the display (user interface) is updated to show the information about the new call everytime a new call is established in the Asterisk box. You must have understood that when a call is established in the Asterisk box, a corresponding instance of *Call* will be attached to the *AsteriskProvider*[7]. Being one of the listeners of the provider, *Main* will receive a notification of that event (❶).

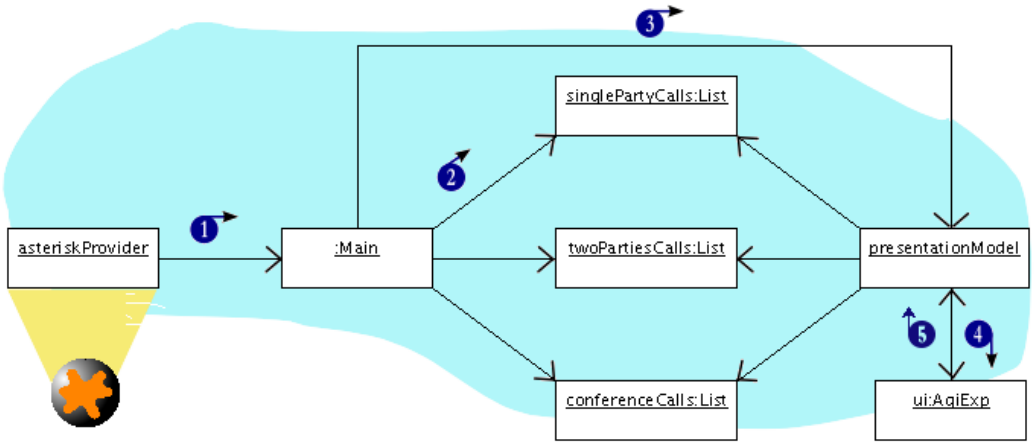


Fig. 22. Snippet from constructor of Main class

The provider pass in the call to the listeners (when doing the notification). *Main*, in this case, will inspect the concrete type of the instance of *Call* that it receives in the notification. Depending on the concrete type, *Main* will add the call to any of the following lists that it maintains: *singlePartyCalls*, *twoPartiesCalls*, or *conferenceCalls* (❷). The lines of code in *Main* that corresponds to this step is shown in *Fig. 23*.

```
public void callAttached(Call call) {
    if(call instanceof SinglePartyCall) {
        singlePartyCalls.add(call);
    } else if(call instanceof TwoPartiesCall) {
        twoPartiesCalls.add(call);
    } else if(call instanceof ConferenceCall) {
        conferenceCalls.add(call);
    }

    call.addListener(this);
    presentationModel.callAttached(call);
}
```

Fig. 23. Implementation of callAttached inMain class

The *Main* class pushes the notification further to an instance of ✪ *PresentationModel* by calling the ✪ *callAttached(...)* method on that presentation model (❸). *PresentationModel* represents the state and behavior of the presentation independently of the GUI controls used in the user interface. For example, instead of having the logics that decide whether or not the monitor button in the panel which shows single party calls should be enabled, mixed with the codes that define things like dimension and laying

7 Which is the reason why I draw a projection light from the Asterisk box to the *AsteriskProvider* – to indicate that the *AsteriskProvider* is representation of the Asterisk box in the world of our program.

out of the GUI controls, we keep them in two separate classes:

- Definitions of GUI controls go to a class named ✪ *AgiExp*.
- The logics mentioned previously go to *PresentationModel*.

The idea is that the presentation model can be used with various presentation technologies (be it *Swing*, web, or others). Of course, then we have to provide a *binding* between the presentation model and the user interface that lies on top of it, somehow. For more about *presentation model*, go to [*Ref. 3*].

Our presentation model pushes the notification even further, to its listeners (that are instances of ✪ *PresentationModel.Listener*), with *AgiExp* being one of them (❹). In response, *AgiExp* will ask the appropriate table to refresh itself to reflect the latest content of the corresponding list of calls pulled out from the presentation model (❺). The following figure shows the section of codes in *AgiExp* from where the request for a refresh is sent.

```
public void callAttached(PresentationModel model, Call call) {
    if(call instanceof SinglePartyCall) {
        fireTableDataChanged(singlePartyCalls_tblModel, dropSinglePartyCall_btn, monitorSinglePartyCall_btn,
            PresentationModel.SINGLEPARTY_CALLTYPE);
    } else if(call instanceof TwoPartiesCall) {
        fireTableDataChanged(twoPartiesCalls_tblModel, dropTwoPartiesCall_btn, monitorTwoPartiesCall_btn,
            PresentationModel.TWOPARTIES_CALLTYPE);
    } else if(call instanceof ConferenceCall) {
        fireTableDataChanged(conferenceCalls_tblModel, dropConferenceCall_btn, monitorConferenceCall_btn,
            PresentationModel.CONFERENCE_CALLTYPE);
    }
}
```

Fig. 24. The definition of method callAttached in AgiExp.

Responding to the request, the table will first ask its model for the number of rows that should be displayed (see point ❶ in *Fig. 25*). Then, for each column in the row it will again ask its model for the value to be displayed (❷)[8].

8 This is a typical way of implementing a *TableModel*. I assume that you're quite familiar with Swing programming.

```
public int getRowCount() { ❶
   return presentationModel.getCalls(PresentationModel.SINGLEPARTY_CALLTYPE).size();
}

public Object getValueAt(int rowIndex, int columnIndex) { ❷
    SinglePartyCall call = (SinglePartyCall) presentationModel
       .getCalls(PresentationModel.SINGLEPARTY_CALLTYPE).get(rowIndex);

    switch(columnIndex) {
        case 0:
            return call.getId();
        case 1:
            return call.getChannel().getDescriptor().getEndpoint().getId();
        case 2:
            switch (call.getState()) {
                case Call.IDLE_STATE: return "IDLE";
                case Call.CONNECTING_STATE: return "CONNECTING";
                case Call.ACTIVE_STATE: return "ACTIVE";
                case Call.INVALID_STATE: return "INVALID";
            }
        case 3:
            return call.getReasonForStateChange();
    }

    return null;
}
```

Fig. 25. A section of codes in AgiExp.SinglePartyCallsTableModel

So that was how “a new row is added to the table”. Now I'm going to explain the “how an entry – in a table – is updated”. When you run the program (later), you will observe that the state of the call changes over the time, until finally it ends up in *INVALID* state. The following is a scientific explanation for that phenomena.

Take a look again at *Fig. 23*, there the *Main* registers itself as a listener to an (new) instance of *Call* passed to it. Being a ✪ *CallListener*, the main class implements the following methods:

- void stateChanged(int oldState, Call call)
- void channelAdded(ConferenceCall conferenceCall, Channel channel)
- void channelRemoved(ConferenceCall conferenceCall, Channel channel)

Everytime the state of the call changes, for example, the call will notify its listeners by invoking the method ✪ *stateChanged(...)* on each one of them. The following figure shows how *Main* reacts to the changes of state of a call:

```
public void stateChanged(int oldState, Call call) {
    presentationModel.callStateChanged(oldState, call);
    if(call.getState() == Call.INVALID_STATE) call.removeListener(this);
}
```

Fig. 26. The definition of method stateChanged in Main

[*Finally...*] We're about to go through another interesting part of our journey, interacting with Asterisk that is. I'm going to show you, first, how to receive and handle manager events that come through a *manager connection*. I'll put it in its own section, for it's quite extensive. So, please, bear with me.

Receiving and handling manager events

Any object that wants to receive manager events from Asterisk has to satisfy the following requirements:

- Implements a java interface named ✪ *ManagerEventHandle*r, which is part of the *Asterisk-Java* library that we're using. The *AsteriskProvider* class does it.
- Be registered to the *manager connection* as one of its handlers. A manager connection is an instance of ✪ *ManagerConnection* – also part of *Asterisk-Java* library – that represents a (network) connection to the Asterisk box. The *Main* class does it for the *AsteriskProvider*, by invoking the method ✪ *addEventHandler(...)* on the manager connection (see *Fig. 21* point ❸).

The only method declared in *ManagerEventHandler* is ✪ *handleEvent(ManagerEvent event)*. The following figure shows how it is implemented in *AsteriskProvider*. I suggest that you also take a look back at *Fig. 9*, you'll notice that the lines of code in *Fig. 27* is the other form of the diagram in *Fig. 9*. You can compare them *point-by-arrow,* e.g.: point ❶ in *Fig. 27* corresponds to arrow #1 in *Fig. 9*.

```
public void handleEvent(ManagerEvent event) { ❶
    boolean processed = false;
    for(Iterator iter = attachedCalls.iterator(); iter.hasNext();) {
        Call call = (Call) iter.next();
        processed = call.process(event); ❷
        if(processed) break;
    }

    if(!processed) processEvent(event);
}

private void processEvent(ManagerEvent event) {
    boolean processed = false;

    for(Iterator iter = callConstructions.iterator(); iter.hasNext();) {
        CallConstruction callConstruction = (CallConstruction) iter.next();
        processed = callConstruction.process(event); ❸.❶
        if(processed) break;
    }

    if(!processed) {
        if(event instanceof NewChannelEvent) {
            NewChannelEvent nce = (NewChannelEvent) event;
            if("Ring".equals(nce.getState())) {
                CallConstruction callConstruction = new CallConstruction(this, nce);
                callConstructions.add(callConstruction); ❸.❷
            }
        }
    }
}
```

Fig. 27. The definition of method stateChanged in Main

By now you might be wondering how do I know that a *NewChannelEvent* in a “Ring” state indicates a new call. My answer: “from analysis of sequence of manager events”. Here, I'll show you....

Analysis of sequence of manager events

The way for an external program to know what's going on in Asterisk is by monitoring it through *manager interface*. Well, actually, manager interface serves two purposes: (1) monitoring and (2) controlling Asterisk from external programs. We'll focus on the monitoring part in this section.

There's nothing fancy here; Asterisk simply sends manager events to every external program that has a *manager connection* to it. The manager events received are ordered in time. However, an event is not necessarily related to the other event that came before it. Therefore, the program needs somekind of a state machine to sort out and make sense of the events.

For every distinct event in Asterisk there's a unique type of manager event (*sorry*). The following figure lists all the types of manager event. In our program we're going to deal only with a small subset of it, namely ✪ *ChannelEvent* (and all its subtypes), ✪ *LinkageEvent*, and ✪ *MeetMeEvent*.

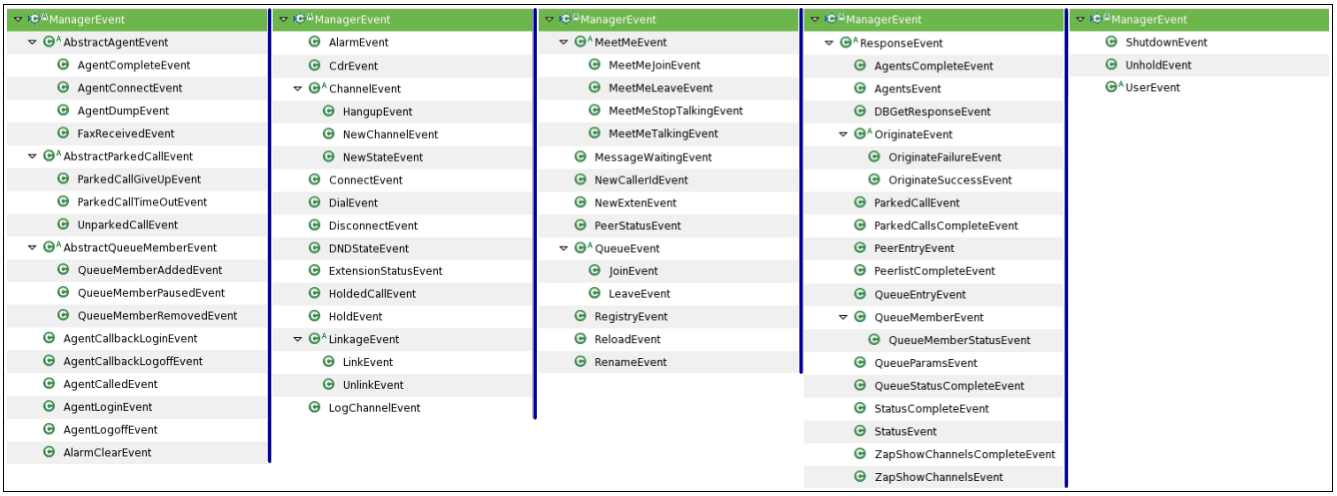


Fig. 28. The type hierarchy of class ManagerEvent

I have created a simple program that connects to the Asterisk manager interface, and dumps all the events it receives to the screen. The name of the program is ✪ *EventDumper*. Before you can run it we need to make sure that manager interface is enabled in Asterisk and create a manager account in Asterisk.

We do that by modifying the file */etc/asterisk/manager.conf*. Go to the *general* section and set the *enabled* property to *yes*. It is also a good idea to uncomment the *displayconnect* properties (so that you can confirm if your program really is connected). You should come up with something like in *Fig. 29*.

```
[general]
enabled = yes
port = 5038
bindaddr = 0.0.0.0
displayconnects = yes
```

Fig. 29. General section in manager.conf

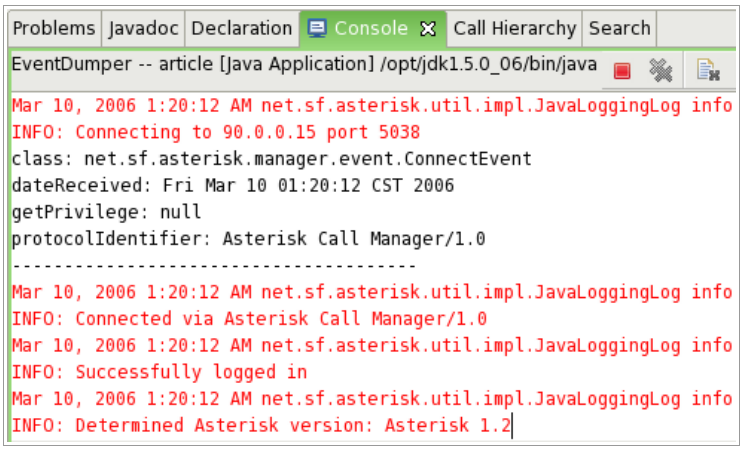
To add a manager account, add the following lines to the end of the file. Remember to change the IP

address – *90.0.0.15* in the example below – to the IP address of the machine where you're going to connect *from* (i.e.: the IP address of the machine where our program will be ran on).

```
[agiexp]
secret=password
deny=0.0.0.0/0.0.0.0
permit=90.0.0.15/255.255.255.0
read = system,call,log,verbose,command,agent,user,action
write = system,call,log,verbose,command,agent,user,action
```

Here are the steps for obtaining sequence of manager events generated by a call:

- Start or *restart* Asterisk (it wouldn't pick up the change you've made to the *manager.conf* unless you restart it).
- Run *EventDumper* from your Eclipse as java application. The program requires that you pass three arguments to it: (1) the IP address of the Asterisk box, (2) the user name of the *manager account* for the connection, and (3) the password of the manager account. You should see something like below in the output console of the *EventDumper* application:



- Then start the kphone, this time let's authenticate as user *johndoe*. First, let's find out manager events generated by a *single party call*. Dial *600@90.0.0.15*. All the events during the call are shown in *Fig. 30*.

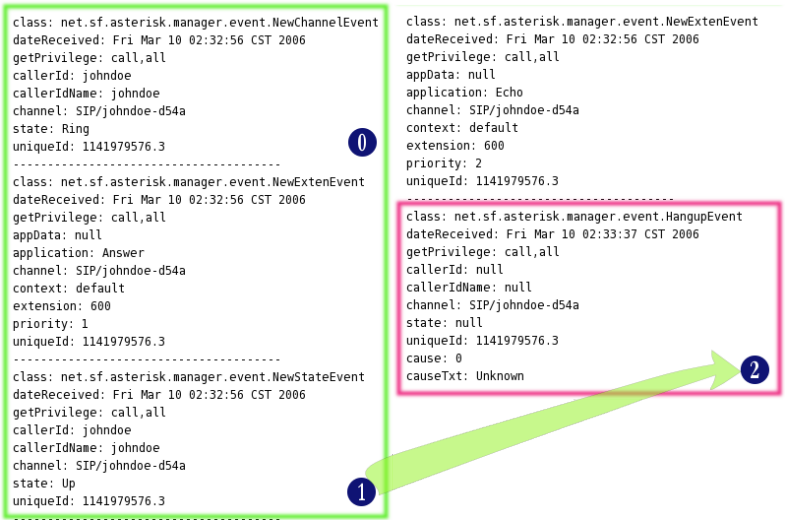


Fig. 30. Manager events generated for a single party call

- Let's not jump in to any conclusion so quickly. We yet have to see the events generated by a *two parties call*. Try making a call from *johndoe* to *goldenboy*. All the events generated are shown in *Fig. 31*.

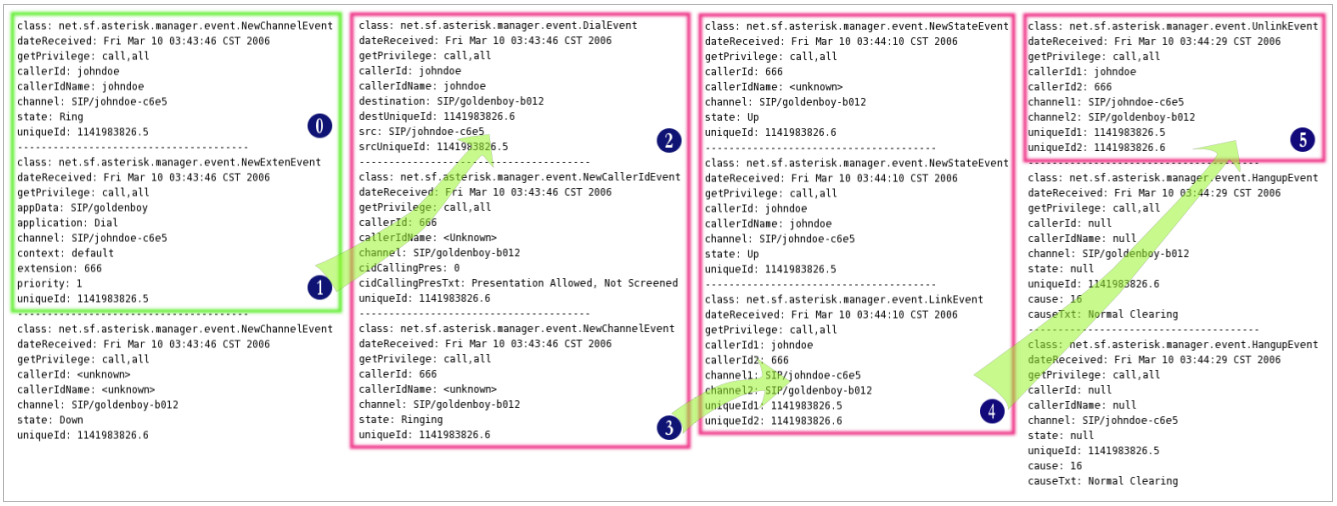


Fig. 31. Manager events generated for a single party call

Try both experiments – making a single party and two parties call – several times, just to be sure. From those logs one thing you can quickly spot is: they all start with *NewChannelEvent* whose *state* is RING. That's how came to the conclusion that the *AsteriskProvider* should know that when that event came in, a new call should be constructed (i.e. a new *call construction* should be commenced)[9]. Therefore, we have the lines of code as is shown in *Fig. 32* in *AsteriskProvider*.

9 I'm not showing you the events from a conference call. Too noisy. But, believe me, it's the same. Crescent my heart.

```
public void handleEvent(ManagerEvent event) {
    boolean processed = false;
    for(Iterator iter = attachedCalls.iterator(); iter.hasNext();) {
        Call call = (Call) iter.next();
        processed = call.process(event);
        if(processed) break;
    }

    if(!processed) processEvent(event);
}
```

```
private void processEvent(ManagerEvent event) {
    boolean processed = false;

    for(Iterator iter = callConstructions.iterator(); iter.hasNext();) {
        CallConstruction callConstruction = (CallConstruction) iter.next();
        processed = callConstruction.process(event);
        if(processed) break;
    }

    if(!processed) {
        if(event instanceof NewChannelEvent) {
            NewChannelEvent nce = (NewChannelEvent) event;
            if("Ring".equals(nce.getState())) {
                CallConstruction callConstruction = new CallConstruction(this, nce);
                callConstructions.add(callConstruction);
            }
        }
    }
}
```

Fig. 32. Manager events generated for a single party call

When instantiated the *CallConstruction* will go directly to its first state, that is waiting for *NewExtenEvent* to come in. The *Fig. 33* shows the state diagram of a call construction. To be more exact, it's limited to the area within the green border. The events are related to each other by their unique id, with the unique id of the *NewChannelEvent* as the basis. In other words, a particular instance of *CallConstruction* will only process events whose unique id matches the basis.

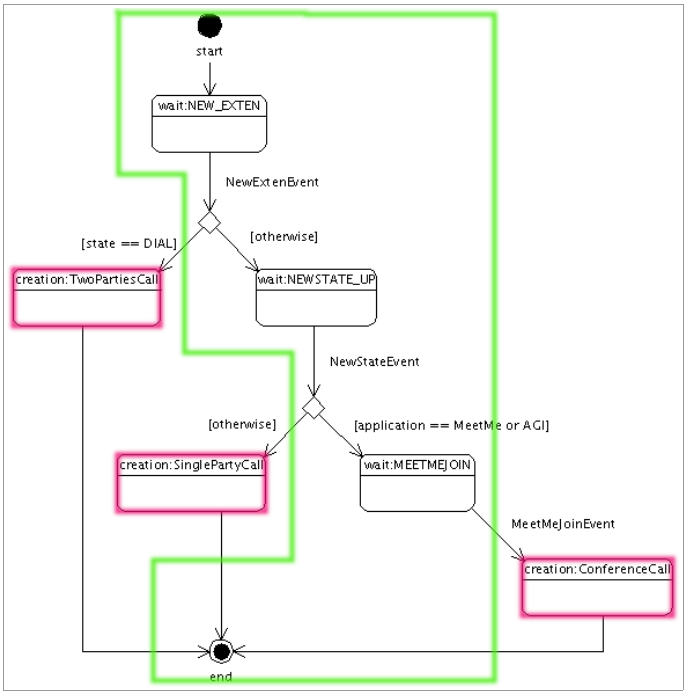


Fig. 33. State diagram of a call construction

Right after it reaches any of the “creation” states (the ones with red border), the call construction will go directly to its final state. From that point on it no longer expects for events (because subsequent related events will be handled by the call it just created). Therefore it will remove itself from the list of living call constructions maintained by the *AsteriskProvider*. The lines of code that correspond to *Fig. 33* is shown in *Fig. 34* (they're within the *processEvent(...)* method defined in class *CallConstruction*).

```
case NEWEXTEN_STATE:
    if(event instanceof NewExtenEvent) {
        NewExtenEvent nee = (NewExtenEvent) event;
        NewChannelEvent nce = (NewChannelEvent) eventStack.peek();
        if(nee.getUniqueId().equals(nce.getUniqueId())) {
            if("Dial".equals(nee.getApplication())) {
                nce = (NewChannelEvent) eventStack.pop();
                String callId = nee.getContext() + "|" + nce.getCallerId() + "|"
                    + nee.getExtension() + "|" + DateFormatter.format(nce.getDateReceived());

                Channel.Descriptor dialingChannelDesc = new Channel.Descriptor(nce.getChannel(),
                    nce.getDateReceived(), new CallEndpoint(nce.getCallerId()));
                Channel.Descriptor dialedChannelDesc = new Channel.Descriptor(
                    new CallEndpoint(nee.getExtension()));

                TwoPartiesCall twoPartiesCall = new TwoPartiesCall(callId, nce.getDateReceived(),
                    dialingChannelDesc, dialedChannelDesc);

                provider.removeCallConstruction(this);
                provider.attachCall(twoPartiesCall);
                return true;
            } else {
                eventStack.push(nee);
                waitState = NEWSTATE_UP_STATE;
                return true;
            }
        }
    }
    break;
```

```
case MEETMEJOIN_STATE:
    if(event instanceof MeetMeJoinEvent) {
        MeetMeJoinEvent mmje = (MeetMeJoinEvent) event;
        NewStateEvent nse = (NewStateEvent) eventStack.pop();
        NewExtenEvent nee = (NewExtenEvent) eventStack.pop();
        NewChannelEvent nce = (NewChannelEvent) eventStack.pop();
        String roomId = mmje.getMeetMe();
        boolean confCallExist = false;
        for(Iterator iter = provider.getAttachedCalls().iterator(); iter.hasNext();) {
            Call attachedCall = (Call) iter.next();
            if(attachedCall instanceof ConferenceCall) {
                ConferenceCall confCall = (ConferenceCall) attachedCall;
                if(confCall.getRoomId().equals(roomId)) {
                    confCallExist = true;
                    Channel.Descriptor newChannelDesc = new Channel.Descriptor(nce.getChannel(),
                        nce.getDateReceived(), new CallEndpoint(nee.getExtension()));
                    confCall.addChannel(newChannelDesc);
                    provider.removeCallConstruction(this);
                    return true;
                }
            }
        }
        if(!confCallExist) {
            Channel.Descriptor channelDesc = new Channel.Descriptor(nce.getChannel(),
                nce.getDateReceived(), new CallEndpoint(nee.getExtension()));
            ConferenceCall confCall = new ConferenceCall(roomId, mmje.getDateReceived(), channelDesc);
            provider.removeCallConstruction(this);
            provider.attachCall(confCall);
            return true;
        }
    }
    break;
```

```
case NEWSTATE_UP_STATE:
    if(event instanceof NewStateEvent) {
        NewStateEvent nse = (NewStateEvent) event;
        NewExtenEvent nee = (NewExtenEvent) eventStack.peek();
        if(nse.getUniqueId().equals(nee.getUniqueId())) {
            if("MeetMe".equals(nee.getApplication()) || "AGI".equals(nee.getApplication())) {
                eventStack.push(nse);
                waitState = MEETMEJOIN_STATE;
                return true;
            } else {
                nee = (NewExtenEvent) eventStack.pop();
                NewChannelEvent nce = (NewChannelEvent) eventStack.pop();
                String callId = nee.getContext() + "|" + nce.getCallerId() + "|"
                    + nee.getExtension() + "|" + DateFormatter.format(nee.getDateReceived());

                Channel.Descriptor channelDesc = new Channel.Descriptor(
                    nce.getChannel(), nce.getDateReceived(), new CallEndpoint(nee.getExtension()));
                SinglePartyCall singlePartyCall = new SinglePartyCall(callId, nee.getDateReceived(),
                    Call.ACTIVE_STATE, channelDesc);
                provider.removeCallConstruction(this);
                provider.attachCall(singlePartyCall);
                return true;
            }
        }
    } else if(event instanceof HangupEvent) {
        HangupEvent he = (HangupEvent) event;
        NewExtenEvent nee = (NewExtenEvent) eventStack.peek();
        if(he.getUniqueId().equals(nee.getUniqueId())) {
            provider.removeCallConstruction(this);
        }
    }
    break;
```

Fig. 34. Implementation of processEvent(...) in CallConstruction

You might be asking right now how do I decided, for example, to instantiate a *SinglePartyCall* when an instance of *NewStateEvent* whose *state* is UP and application is not *MeetMe* is received. Well, firstly, I've made comparisons between various call scenarios. So I *just* know (believe it like you believe in god[10] – ha!). Second: it's because at that moment the program will already have all the information it needs to instantiate a *SinglePartyCall*.

10 If you don't believe in god, you're lost dude. I have no time to convince you and do a salvation for you. Hehehe....

Ok, no more questions. Let's see how a call handles manager events passed to it. We'll start with single party call. The following figure shows the state diagram of a single party call.

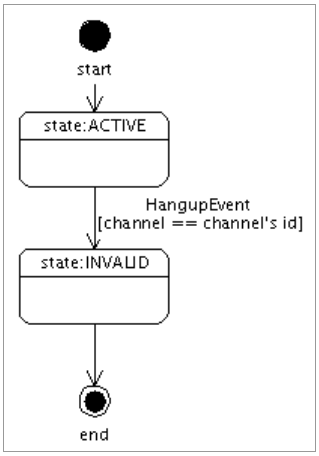


Fig. 35. State Diagram of single party call

And... *Fig. 36* below shows how a two parties call handles manager events passed to it.

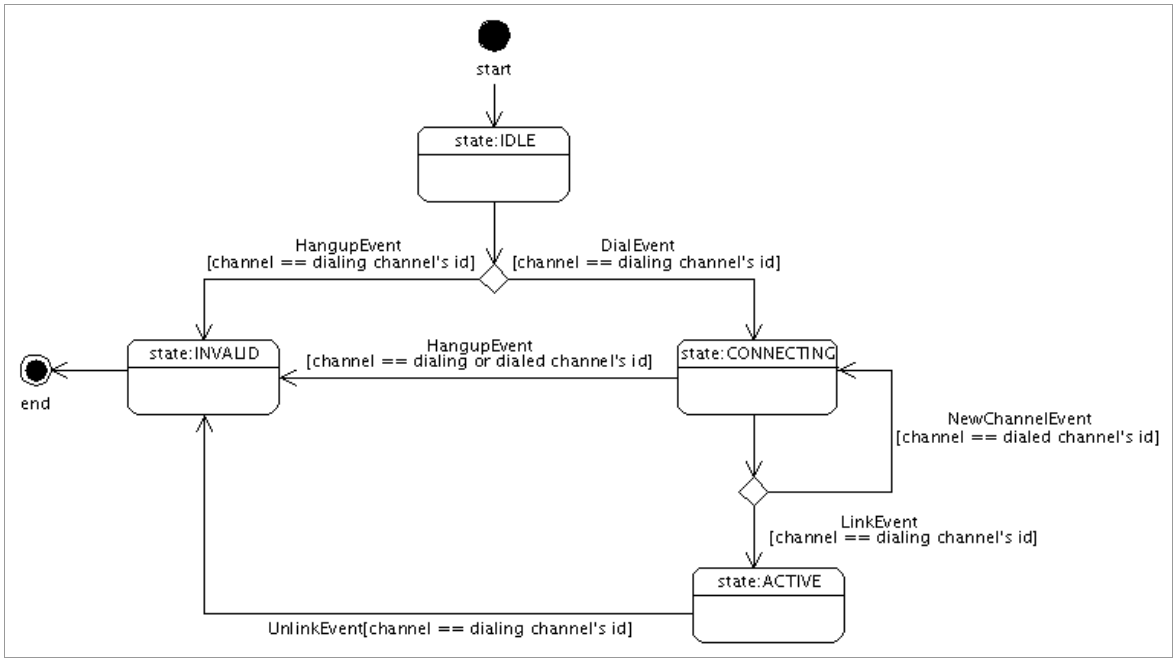
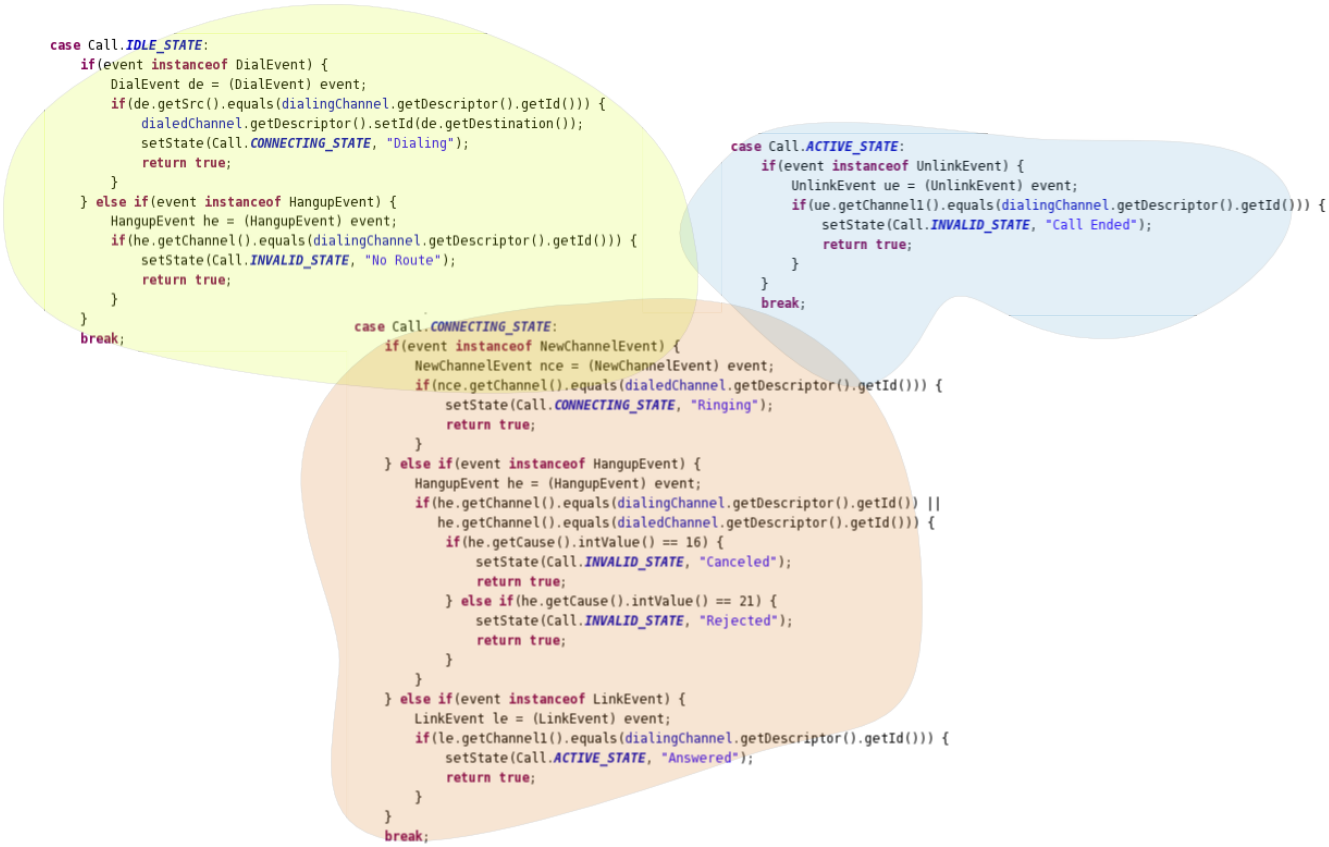


Fig. 36. State Diagram of two parties call

One thing in common between the way a single party and two parties call filter the events (such that only “relevant” events could affect its state) is that they check the name of the channel of the event – by calling ✪ *getChannel()* on the event – and compare it with the id of the descriptor of the channel of the call (*phew!*).

The following figure shows the java code representation of *Fig. 36*. It's the code within the method *processEvent(...)* in the class *TwoPartiesCall*. To get a very good grasp of the concepts and mechanics, I

encourage you to play with the debugger and experiment with various scenarios such as where (1) *goldenboy* doesn't register to Asterisk, (2) *johndoe* hangs up the call before *goldenboy* picks up the call, and (3) *goldenboy* rejects the call.



```
case Call.IDLE_STATE:
    if(event instanceof DialEvent) {
        DialEvent de = (DialEvent) event;
        if(de.getSrc().equals(dialingChannel.getDescriptor().getId())) {
            dialedChannel.getDescriptor().setId(de.getDestination());
            setState(Call.CONNECTING_STATE, "Dialing");
            return true;
        }
    } else if(event instanceof HangupEvent) {
        HangupEvent he = (HangupEvent) event;
        if(he.getChannel().equals(dialingChannel.getDescriptor().getId())) {
            setState(Call.INVALID_STATE, "No Route");
            return true;
        }
    }
    break;
```

```
case Call.ACTIVE_STATE:
    if(event instanceof UnlinkEvent) {
        UnlinkEvent ue = (UnlinkEvent) event;
        if(ue.getChannel1().equals(dialingChannel.getDescriptor().getId())) {
            setState(Call.INVALID_STATE, "Call Ended");
            return true;
        }
    }
    break;
```

```
case Call.CONNECTING_STATE:
    if(event instanceof NewChannelEvent) {
        NewChannelEvent nce = (NewChannelEvent) event;
        if(nce.getChannel().equals(dialedChannel.getDescriptor().getId())) {
            setState(Call.CONNECTING_STATE, "Ringing");
            return true;
        }
    } else if(event instanceof HangupEvent) {
        HangupEvent he = (HangupEvent) event;
        if(he.getChannel().equals(dialingChannel.getDescriptor().getId()) ||
           he.getChannel().equals(dialedChannel.getDescriptor().getId())) {
            if(he.getCause().intValue() == 16) {
                setState(Call.INVALID_STATE, "Canceled");
                return true;
            } else if(he.getCause().intValue() == 21) {
                setState(Call.INVALID_STATE, "Rejected");
                return true;
            }
        }
    } else if(event instanceof LinkEvent) {
        LinkEvent le = (LinkEvent) event;
        if(le.getChannel1().equals(dialingChannel.getDescriptor().getId())) {
            setState(Call.ACTIVE_STATE, "Answered");
            return true;
        }
    }
    break;
```

Fig. 37. Source code representation of Fig. 36

Testing (#1)

Now let's try running our program (*yay!*). We'll first do functional testing manually. Automatic testing will be explained later. Here are the steps you should follow:

1. Make sure that Asterisk is running..., with the manager interface enabled.
2. Run the main class of our program, namely *net.raka.agiexp.gui.Main*. You can do that directly from Eclipse. Don't forget to pass in the following program arguments:
    1. The IP address of the Asterisk box.
    2. The account name in manager interface (e.g.: *agiexp*).
    3. The password of the account in manager interface (e.g.: *password*)
    4. The port number where Asterisk's SIP service is bound to (by default it's 5060. Check the value of property named *bindport* in the general section in */etc/asterisk/sip.conf*).
    5. The IP address of the machine where you're running this program. Don't use *127.0.0.1* or *localhost*, use actual IP number. This value is used by conference call monitor, which I'll explain later.
    6. The port number where the conference call monitor will be bound too. Use any port which is not occupied (e.g.: 7000). This value is also used by conference call monitor.
3. Make the calls:
    1. Try calling 600 from *johndoe*, and watch the table in the tab named “Single Party Calls” in our program.
    2. Try calling *goldenboy* from *johndoe* (dial 666). Watch the table in the tab “Two Parties Calls”. Experiment with the scenarios I mentioned previously.

How was it? I hope you were satisfied. Did you notice that when you clicked on the row that displays an *active* call, the two buttons at the bottom right corner of the panel became active?

- Clicking the *Drop* button, the call will be, well, dropped (all the parties connected to the call will be kicked out, and at the end the call will become invalid).
- You can click the *Monitor* button to record the conversation within the call. I specified in the code that the file will be saved in */var/recordings*.

Umm..., you've guessed it. We're now getting to the other use of Asterisk manager interface, for controlling. So let's get back to code (we have much to learn).

Implementation Again (Controlling Asterisk)

The action handler of those buttons are defined in the class *AgiExp*. The clicking on the drop button, for example, will end up in the ✪ *drop(Call call)* method declared in *Provider*. The following diagram shows the path that the clicking will go through to reach the *drop(...)* method in *Provider*.

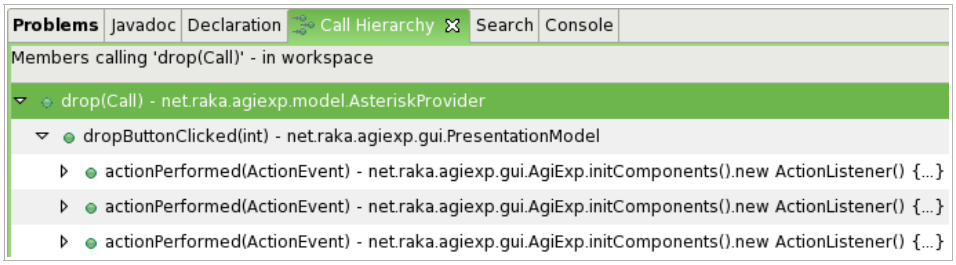


Fig. 38. Members calling 'drop(Call)' – in workspace

So now let's take a look at the implementation of the *drop(...)* method in *AsteriskProvider*.

```
public void drop(Call call) {
    if(call instanceof SinglePartyCall) {
        SinglePartyCall spc = (SinglePartyCall) call;
        HangupAction hangupAction = new HangupAction(spc.getChannel().getDescriptor().getId());
        try {
            managerConnection.sendAction(hangupAction);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    } else if(call instanceof TwoPartiesCall) {
        TwoPartiesCall tpc = (TwoPartiesCall) call;
        HangupAction hangupAction = new HangupAction(tpc.getDialingChannel().getDescriptor().getId());
        try {
            managerConnection.sendAction(hangupAction);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    } else if(call instanceof ConferenceCall) {
        ConferenceCall cc = (ConferenceCall) call;
        for(Iterator iter = cc.getChannels().iterator(); iter.hasNext();) {
            Channel channel = (Channel) iter.next();
            HangupAction hangupAction = new HangupAction(channel.getDescriptor().getId());
            try {
                managerConnection.sendAction(hangupAction);
            } catch (Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
}
```

Fig. 39. Members calling 'drop(Call)' – in workspace

So... controlling Asterisk from external program is as simple as sending out an instance of ✪ *ManagerAction* through the manager connection. In the case of dropping a call, we send an instance of *HangupAction*. For recording a call, we send an instance of ✪ *MonitorAction*. The following figure displays all the standard manager actions. But it doesn't mean all of them can be used. The number of actions available are determined by the modules presently loaded in the Asterisk.

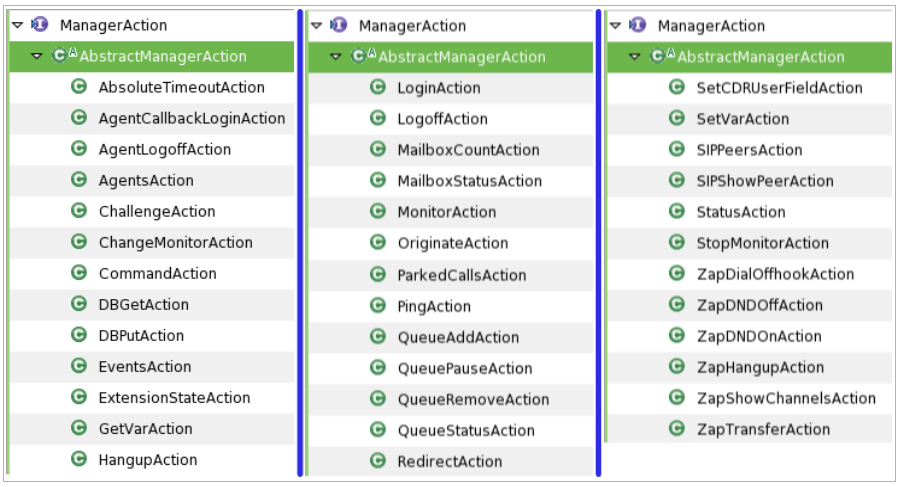


Fig. 40. Members calling 'drop(Call)' – in workspace

In most cases, maybe, the standard actions provided by Asterisk-Java suffice to get your job done. But, in case your fancy requirement requires (*ugh*) you to send an action that's not in the list above, you can have your custom action by making a class that extends ✪ *AbstractManagerAction*.Of course, on the Asterisk side, there must also be a module loaded that can handle your custom action.

(*What's the guideline for implementing a custom action – other than extending AbstractManagerAction?*)

Not much. I see that an instance of manager action that we throw into the manager connection is sent through the wire as a plain text, in a very simple format..., key-value pairs. The following is what I got from Ethereal when the program sent a *MonitorAction*:

```
action: Monitor
actionid: 421988_3#
file: /var/recordings/-1059416815
channel: SIP/johndoe-d54a
mix: true
format: wav
```

That text is constructed out of the properties of the action. You can verify this information by taking a look at the source code of *MonitorAction*. The class has the following methods: *getAction()*, *getActionId()*, *getFile()*, *getChannel()*, *getMix()*, and *getFormat()*. If you're still unconvinced – what an infidel – then check out the implementation of the method ✪ *buildAction(...)* in class ✪ *ActionBuilderImpl* (part of Asterisk-Java).

Conclusion: if you have a third party module that provides externally accessible extra functionalities installed in your Asterisk, and you want to be able to access them from your Java program, then ask for the documentation that explains the key-value pairs accepted / required by those functionalities. Then, on the Java side, you only have to make sure that your custom manager action(s) has the appropriate getters (for all the keys specified in the documentation)[11].

11 Just follow the JavaBean convention for the naming of the getters.

Implementation Returns (Conference Call)

Conference call is the final piece in our program. I put it the last, separated from the explanation about the two other calls, because (1) there are other configuration things to do in order to make it work (*a tiny hassle!*), and (2) there are other new concepts and techniques that I'd like to introduce.

Zaptel installation

To make the conference feature – using MeetMe – available on your Asterisk, you have to do several things. First of all, *MeetMe* requires a timing device in order to operate (Asterisk won't even compile it if no timing device is found). All Digium PCI hardware provides a 1-kHz timing interface. If you lack the PCI hardware required to provide timing, the *ztdummy* driver can be used as a timing device[12]. So..., you need to install zaptel drivers on your Linux. Here's a quick instruction:

1. Download *zaptel-1.2.x.tar.gz* from Asterisk's website [*Res. 2*], and unpack it.
2. I assume that you don't have Digium PCI hardware. So, you need to have ztdummy compiled too. Edit the the *Makefile* (of zaptel), and look for the following section (and remove the hash that precedes *ztdummy*):

```
MODULES:=zaptel tor2 torisa wcusb wcfxo wctdm wctdm24xxp \
         ztdynamic ztd-eth wct1xxp wct4xxp wcte11xp pciradio \
         ztd-loc # ztdummy
```

   **) If your Linux uses kernel 2.6.x, you don't have to do this.*
3. Finally do the: (1) *make clean*, (2) *make*, (3) *make install*, and (4) *make config*.
4. Re-compile and re-install Asterisk.

That's all there is to it. Well, not really. At least not in my case. I don't know what is wrong with my Linux (or zaptel installation), I have to do the following ritual before starting Asterisk:

```
rakahome:~ # whoami
root
rakahome:~ # modprobe zaptel
rakahome:~ # mkdir /dev/zap
rakahome:~ # mv /dev/zapchannel /dev/zap/channel
rakahome:~ # mv /dev/zapctl /dev/zap/ctl
rakahome:~ # mv /dev/zappseudo /dev/zap/pseudo
rakahome:~ # mv /dev/zaptimer /dev/zap/timer
rakahome:~ # modprobe ztdummy
```

Fig. 41. My silly ritual

Meetme configuration

The configuration options for the MeetMe conferencing system are found in */etc/asterisk/meetme.conf*. Inside the configuration file, you define conference rooms and optional numeric passwords (if a password is defined here, caller will be required to enter a password for entering the conference room).

To define a conference room (with number 400, and without password), simply add the following line to the end of *meetme.conf*:

12 Again, I copied-and-pasted several sentences from [*Ref. 2*] – Chapter 3, on Compiling Zaptel.

```
conf => 400
```

Now, to support such "requirement" as: "if the caller dials the number 5XXX, then she should be joined to conference room #XXX", we have to add the following lines in the *extension.conf* (put them in the section for *default* context):

```
exten => _5XXX,1,MeetMe(${EXTEN:1},i)
exten => _5XXX,n,Hangup
```

There you have your conferencing system set up. Re-start Asterisk, followed by making a call (from *johndoe*) to the number 5400 and once you're connected to the conference room, just follow the instruction on the phone (*after the tone bla bla bla...*)[13]. Well, I guess a conference wouldn't be very fun with only one person in the room. So, try making a call from other UA (e.g.: *goldenboy*) to 5400[14]. Observe! You can start with checking Asterisk's status from the CLI (see *Fig. 42*).

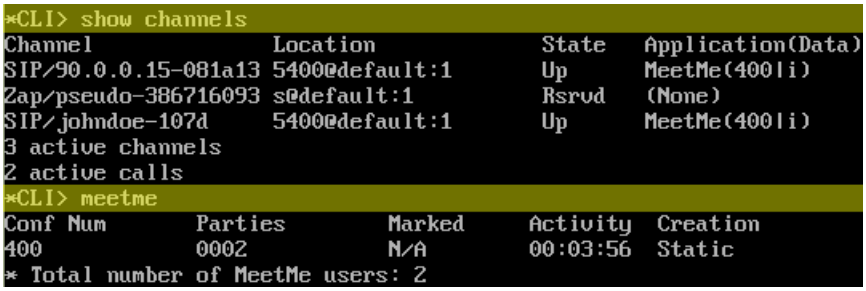


Fig. 42. Checking channels & meetme status

Back to our program

The logic of creation of a conference call goes like this (this is to clarify the diagram in *Fig. 33*):

- Incoming calls to a conference room go through the same state transitions, that is: *waiting for NewExtenEvent* ➔ *waiting for NewStateEvent* ➔ *waiting for MeetMeJoinEvent* ➔ *creation of conference call*.
- However, only the first call should cause the creation of an instance of *ConferenceCall* (❶). As long as the first call hasn't left the conference room, the subsequent calls should be added – as a new channel – to the instance of *ConferenceCall* created previously (❷).

The logic above translates to the following java codes in the class *CallConstruction*:

13 You need to install a package named *asterisk-sound* to be able to hear the instruction. Asterisk-sound is simply a collection of audio files.

14 It's better to carry on this experiment in a local area network with several computers (so that you can really play with sound – remember? the sound card exclusive access problem). But if you're only interested in the signalling, it's ok to run all the phones in a single computer.

```
case MEETMEJOIN_STATE:
    if(event instanceof MeetMeJoinEvent) {
        MeetMeJoinEvent mmje = (MeetMeJoinEvent) event;
        NewStateEvent nse = (NewStateEvent) eventStack.pop();
        NewExtenEvent nee = (NewExtenEvent) eventStack.pop();
        NewChannelEvent nce = (NewChannelEvent) eventStack.pop();
        String roomId = mmje.getMeetMe();
        boolean confCallExist = false;
        for(Iterator iter = provider.getAttachedCalls().iterator(); iter.hasNext();) {
            Call attachedCall = (Call) iter.next();
            if(attachedCall instanceof ConferenceCall) {
                ConferenceCall confCall = (ConferenceCall) attachedCall;
                if(confCall.getRoomId().equals(roomId)) { (2)
                    confCallExist = true;
                    Channel.Descriptor newChannelDesc = new Channel.Descriptor(nce.getChannel(),
                        nce.getDateReceived(), new CallEndpoint(nee.getExtension()));
                    confCall.addChannel(newChannelDesc);
                    provider.removeCallConstruction(this);
                    return true;
                }
            }
        }
        if(!confCallExist) { (1)
            Channel.Descriptor channelDesc = new Channel.Descriptor(nce.getChannel(),
                nce.getDateReceived(), new CallEndpoint(nee.getExtension()));
            ConferenceCall confCall = new ConferenceCall(roomId, mmje.getDateReceived(), channelDesc);
            provider.removeCallConstruction(this);
            provider.attachCall(confCall);
            return true;
        }
    }
    break;
```

Fig. 43. Conference call creation logic

Well, there was nothing new in the above explanation (just a reiteration of processing of manager events). The two ***new*** interesting things that I promised are integration using *Asterisk Gateway Interface* (AGI) and programming with SIP stack library. Let's begin with AGI.

Le Mayeur – Portrait of Ni Polok

Sorry for the interruption, but I have to give some words of caution to you. It's about the state machine. I don't want to (mis)lead you to think that the state machine in this program is reusable (can be used in all kinds of situation involving detection of creation and destruction of a call in Asterisk).

I would say that a state machine is highly related (coupled?) to the dialplan. Even a little modification to the dialplan might render the state machine useless. For example, if you used a function other than *Dial* to make an outbound call from your Asterisk, then you would not see the two parties call show up in the user interface. It's because our state machine assumes that the lifecyle of a two parties call begins when a *NewExtenEvent* – whose *application* property is set to *Dial* – is received (see again *Fig. 34*, blue shaded area).

So, I think, in developing a telephony application (using the approach similar to the one we use here) first you have to understand the requirements for the dialplan. Then you have its structure well defined, so that you know all the possible sequence of events. Finally you design your state machine based on that.

Asterisk Gateway Interface (AGI)

The Asterisk Gateway Interface, or AGI, provides a standard interface by which external programs may control the Asterisk dialplan.

You must have realized that the “language” we use in the */etc/asterisk/extensions.conf* to craft the dialplan is very simple (and limited). It supports branching in a very limited way, through the use of *dialplan functions* named *GotoIf*. It doesn't even seem to support looping. We can do simple string pattern matching, though, like we did with the extension to access a conference (_5XXX).

Clearly other – additional – ways of controlling a dialplan is required..., with other languages which are more capable (and more popular). That's the case for AGI; it's what CGI is to webserver to Asterisk. You can program an AGI in languages such as C, python, Java, and maybe some others.

The AGI script can be deployed in the same machine as the Asterisk (as is the common case with C or python-based AGI script), or on a remote machine (in which case, the communication is conducted between Asterisk and the AGI server that hosts the script through a socket, using *FastAGI* protocol).

In our program we take the later approach, and the programming language that we use to create the script is Java. The library *Asterisk-Java* comes with classes that allow us to run an AGI server in embeded mode (in addition to a standalone AGI server program). Now let's take a look how we call our remotely deployed AGI script from the Asterisk. Replace the following lines of code (that you put earlier) from *extensions.conf*:

```
exten => _5XXX,1,MeetMe(${EXTEN:1},i)
exten => _5XXX,n,Hangup
```

...with...

```
exten => _5XXX,1,Agi(agi://90.0.0.15/conferenceMonitor?roomId=${EXTEN:1})
exten => _5XXX,n,MeetMe(${EXTEN:1},i)
exten => _5XXX,n,Hangup
```

Based on the configuration above, when a call enters the extension 5XXX, a request will be sent to an AGI server running on the machine 90.0.0.15 to execute a script which is mapped to an identifier *conferenceMonitor*. A parameter named *roomId,* whose value is set to XXX, will be passed along the request. On successful return from the execution of the script, the next line in the dialplan will be executed.

To have a custom AGI server, you have to create a class that extends ✪ *DefaultAGIServer*. Furthermore, the instance of that server must have its *mapping strategy* set. A mapping strategy determines which instance of ✪ *AGIScript* has to be called to service a given ✪ *AGIRequest*.

We set the mapping strategy by calling ✪ *setMappingStrategy(...)* on that instance of AGI server[15]. In our case, we assign an instance of ✪ *ConfMonitorMappingStrategy* to our server, that returns an instance of ✪ *ConferenceMonitorScript* for the requests made to *conferenceMonitor*.

15 The mapping for standalone AGI server is specified in a properties file named *agiserver.conf*.

```
public class ConfMonitorAgiServer extends DefaultAGIServer {
    private ConferenceMonitorScript confMonitorScript;

    public ConfMonitorAgiServer(ConferenceMonitorScript confMonitorScript) {
        if(confMonitorScript == null) {
            throw new IllegalArgumentException("confMonitorScript can not be null");
        }
        this.confMonitorScript = confMonitorScript;
        setMappingStrategy(new ConfMonitorMappingStrategy());
    }

    class ConfMonitorMappingStrategy implements MappingStrategy {
        public AGIScript determineScript(AGIRequest request) {
            if("conferenceMonitor".equals(request.getScript())) {
                return confMonitorScript;
            }
            return null;
        }
    }
}
```

Fig. 44. Source code of ConfMonitorAgiServer

Our server is launched during the initialization of our program. We do that by starting a new thread – from the *run()* method in our *Main* class – that instantiates and run the server (see *Fig. 45*).

```
public void run(final String asteriskIP, final int asteriskSIPPort,
    final String confMonitorIP, final int confMonitorPort) {
    try {
        Thread agiServerThread = new Thread(new Runnable() {
            public void run() {
                ConferenceMonitor confMonitor = new ConferenceMonitor(confMonitorIP,
                    confMonitorPort, asteriskIP, asteriskSIPPort);
                asteriskProvider.addListener(confMonitor);
                ConferenceMonitorScript confMonitorScript =
                    new ConferenceMonitorScript(confMonitor);
                ConfMonitorAgiServer confMonitorAgiServer =
                    new ConfMonitorAgiServer(confMonitorScript);
                confMonitorAgiServer.run();
            }
        });
        agiServerThread.start();
    } catch (Exception e) {
        JOptionPane.showMessageDialog(ui, "Error: " + e.getLocalizedMessage());
        System.exit(-1);
    }
}
```

Fig. 45. Starting embeded AGI server

An AGI script, in our java program, is an instance of a class that extends ✪ *BaseAGIScript*. The class has to implements the *service(...)* method. The following figure shows the implementation of the *service(...)* method in *ConferenceMonitorScript*.

```
public void service(AGIRequest request, AGIChannel channel)
    throws AGIException {
    String roomId = request.getParameter("roomId");
    System.out.println("service.roomId : " + roomId);
    if(!confMonitor.isMonitored(roomId)) {
        System.out.println(roomId + " is not monitored");
        confMonitor.monitorConference(roomId);
    }
}
```

Fig. 46. The service(...) method of ConferenceMonitorScript

Our script simply delegates the call to an instance of ✪ *ConferenceMonitor*, asking it to monitor a conference with the given *roomId*.

Why all these intricacies (with the script, conference monitoring, and all)?

Well, with the script, basically we want to intercept calls to a conference room, such that we can sneak in a dummy (silent) participant to the room (if we haven't already done so). Sneaking in that dummy participant is the task that we assign to conference monitor. How does it accomplish the task? It's the topic for the next section....

A Sip of SIP

First of all..., what is SIP? It's an end-to-end, client-server signalling protocol. Signalling in telephone system is the key mechanism by which telephone calls are set up and terminated. Many value added services in a telephone system are made possible by having programs that mingle with the signalling process. Examples of value added services are: PBX features, group calling, click to connect, and many kinds of *intelligent network* (IN)[16] services.

Signalling on SIP is carried through IP network (therefore it's a VoIP protocol). It is specified in an RFC numbered 3261. The kinds of entity in the network that can deal with signalling messages are[17]:

- User Agent (user application), that can be further classified into 3 types:
  - UA Client (originates calls)
  - UA Server (listen for incoming calls)
  - B2B (back-to-back) UA
- SIP Proxy Server: relays call signalling, i.e. acts as both client and server.
- SIP Redirect Server: redirects callers to other servers.
- SIP Registrar: accepts registration requests from users.

One thing from SIP that attracts me most is the fact that it uses plain text for the messages used in the signalling. The format is also simple, *attribute-value-pair* (that's looks very much like HTTP messages). So, as long as you understand the protocol specification, and know how to parse and construct those messages in your program, you're good to go[18]. The SIP entities mentioned above are basically (SIP) message processors, that behave in such way that they conform to a specification. The following figure shows an example of SIP message.

Fig. 47. An example of SIP message

16 Intelligent network is a collecion of servers and other resources used to control call setup and to provide media services such as announcements, voicemail, etc.

17 Here “deal with” means sending, accepting, modifying, or passing / redirecting.

18 Well, maybe in the harsh reality it's not that simple.

In our program, we implement a very simple UA client. To help us with the construction and parsing of SIP messages (among other things[19]) we use NIST-SIP that is an open source implementation of a specification named JAIN SIP issued by Java Community Process.

This UA client represents a dummy participant that we put in a conference room. For each active conference we have a distinct instance of UA client. As a client, the only things it does are: (1) making a call to the conference room, and (2) disconnecting itself from the conference room when the conference ends. The details of those operations are encapsulated in a class named ✪ *NoteTaker*, that has the following methods:

- joinConference(String roomId)
- leaveConference()

Instantiation of *NoteTaker* and invocation of its *joinConference(...)* method take place inside the execution of the ✪ *monitorConference(...)* method of *ConferenceMonitor* (see *Fig. 47*).

```
public void monitorConference(String roomId) {
    if(roomId == null) {
        throw new IllegalArgumentException("roomId can not be null");
    }
    synchronized (noteTakers) {
        NoteTaker noteTaker = new NoteTaker(roomId);
        noteTaker.joinConference();
        noteTakers.put(roomId, noteTaker);
    }
}
```

Fig. 48. Implementation of monitorConference(...) in ConferenceMonitor

You can say that *ConferenceMonitor* is all ears, because it implements so many listener interfaces: ✪ *SipListener*[20]*, ProviderListener*, and *CallListener*.

- As a SIP listener, it processes response events it receives from the SIP peer[21], inside the ✪ *processResponse(...)* method (where it has the chance to send the SIP ACK message, that marks the completion of a call set up, back to the peer).
- As a provider listener, it registers itself as listener to a conference call whenever it receives a notification that a (conference) call has been attached to the provider.
- As a call listener, it listens to the dynamic of a conference call. Inside the ✪ *channelRemoved(...)* method, it has the chance to check the number of remaining participants in the room, and ask the *NoteTaker* to disconnect itself from the room if the number is 1 (see *Fig. 49*).

19 If you're a *regexp* guru, for example, you might think you don't need library just for text processing. However, there are some more high level concepts above SIP messages, such as *dialog* and *transaction*. If you'd like to work with them, easily, you'd want to use specialized library like JAIN-SIP.

20 An interface defined in JAIN-SIP.

21 The other SIP entity that it communicates with, which in our case is the Asterisk box.

```
public void channelRemoved(
    ConferenceCall conferenceCall, Channel channel) {
    //assume the only one left is the notetaker
    if(conferenceCall.getChannels().size() == 1) {
        unmonitorConference(conferenceCall.getRoomId());
    }
}
public void unmonitorConference(String roomId) {
    if(roomId == null) {
        throw new IllegalArgumentException("roomId can not be null");
    }
    synchronized (noteTakers) {
        NoteTaker noteTaker = (NoteTaker) noteTakers.get(roomId);
        if(noteTaker != null) {
            noteTaker.leaveConference();
            noteTakers.remove(roomId);
        }
    }
}
```

Fig. 48. Implementation of monitorConference(...) in ConferenceMonitor

We get to the important questions now: (1) what messages are exchanged during call set up and tear down, (2) how do we construct / parse them, and (3) how do we send them. Let's begin with the messages. The following figure shows the signalling of a normal case of session setup and tear down.

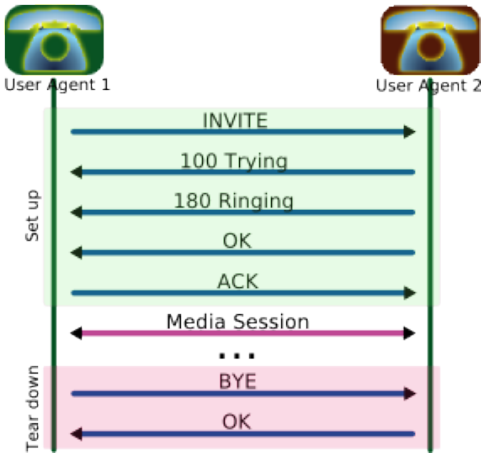


Fig. 49. Setup and tear down of a call session

In SIP parlance, making a call is “sending an invitation” (to the callee). The caller does that by sending a SIP request whose method name is set to INVITE (point 1 in *Fig. 50*). The answerer will send the response to the location destination specified in the first *Via* field in the request, which in our example it is port *5062* on machine with IP *90.0.0.15* (point 2).

The location specified in the *Via* field is not necesarilly the same as the location of the caller. It's possible that the request has travelled through proxy(s) before it reached the destination (where each proxy in the signalling path inserted a *Via* field before the first *Via* field found in the request). However, we can always tell the ultimate destination for the response from the value specified in the *Contact* field in the request (point 3).

The INVITE request contains a *session description* that indicates the desired communication means (audio, video, etc), parameters of those means (such as codec types), and address addresses for receiving media media from the party who will answer the call. In our example, the sender specifies that it supports codecs PCMA, iLBC, GSM, and PCMU. It also specifies that it's waiting for the media on port 1032 (point 4).

The OK response from the answerer also contains a session description. in our example it says that it supports GSM, PCMU, and PCMA, and is receiving media on port 16454 (point 5).

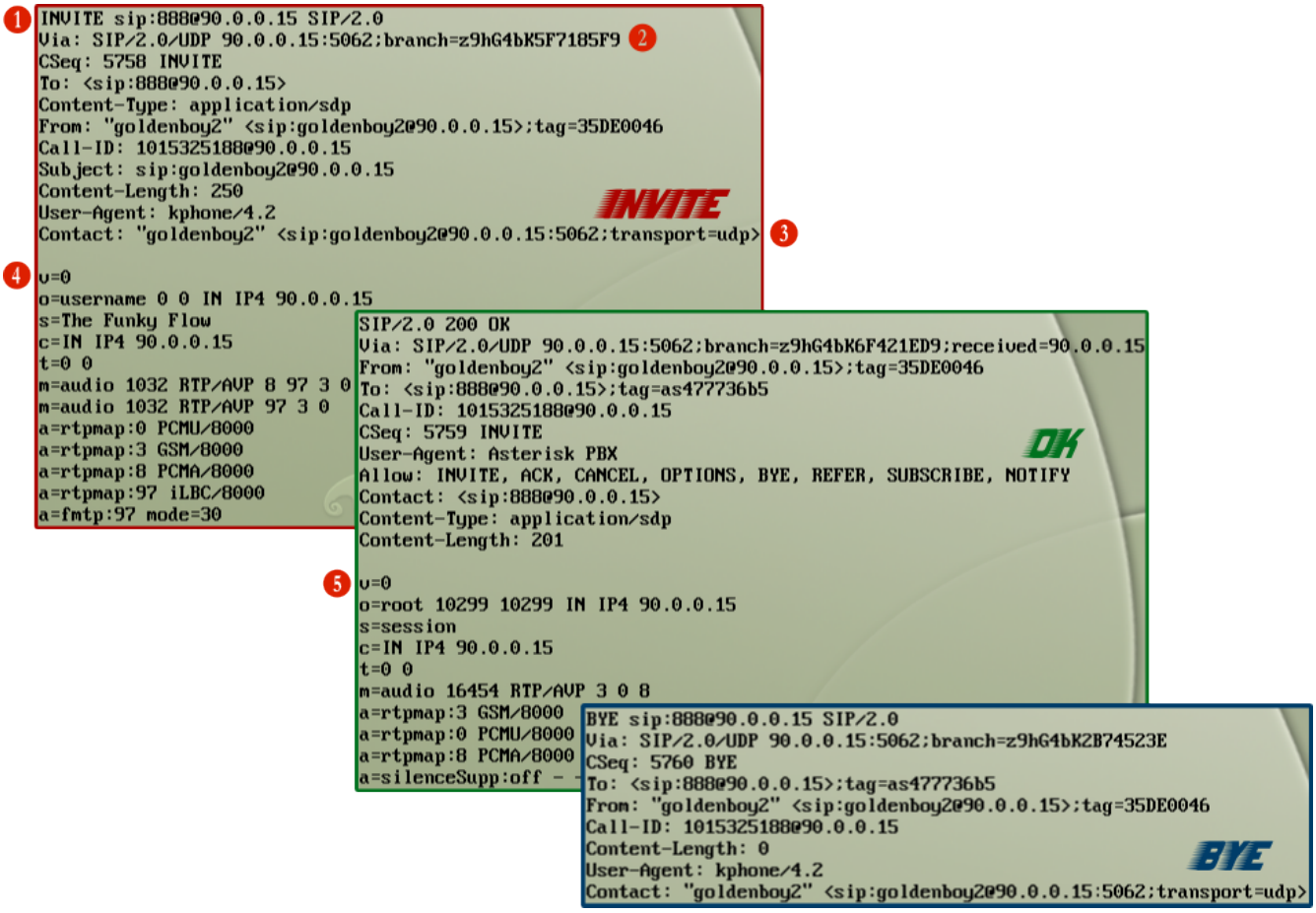


Fig. 50. Example SIP messages

What's the role of Asterisk in a SIP call? The answer is: it can be either a back-to-back user agent (B2B UA) or a SIP proxy. As a B2B UA it stays in the media path of the call (which means, there will be 2 media connections in the call: (1) between caller and Asterisk, and (2) between Asterisk and the answerer). As a SIP proxy, there will be direct media connections between the caller and the answerer. (see *Fig. 51*).

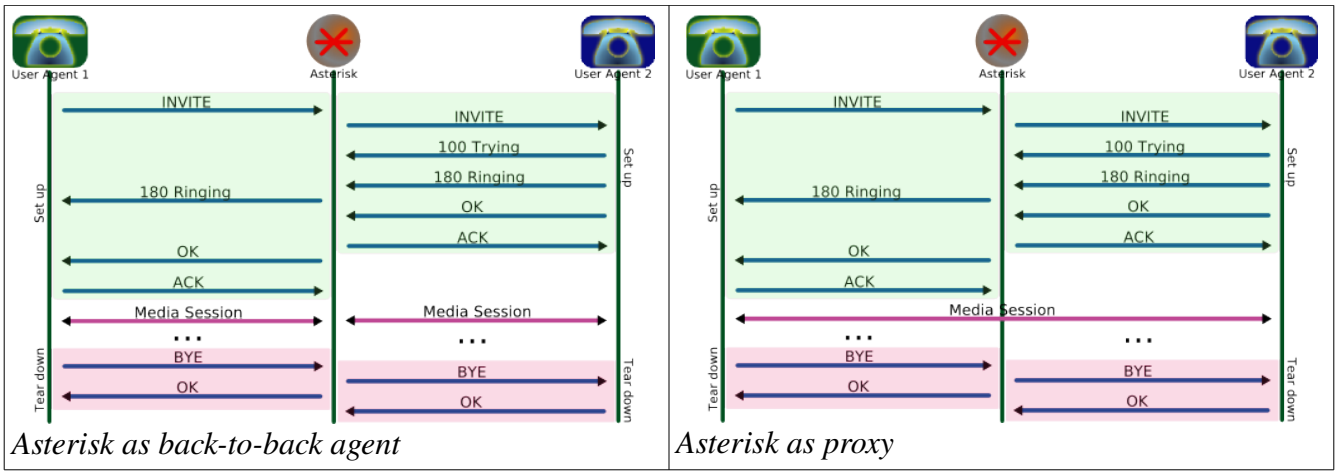


Fig. 51. Asterisk in a a SIP call

Asterisk will act as a proxy if *both* the caller and the answerer supports Re-INVITE, which is a mechanism of modifying the session description during the call. For UA that we register in *sip.conf*, we can set its re-invitability by setting the *canreinvite* property to *no*. In our program we have them all set to *no*. The reason is because we use Asterisk's built-in *Monitor* application for recording the call. That application requires that Asterisk be in the media path.

Another valid reason for operating Asterisk as B2B UA is to free us from worrying too much about compatibility issues (esp. regarding codec supports) between the UAs involved in the call, because Asterisk will iron out the incompatibility (if any) by performing *transcoding*.

I'm not going to explain how to use JAIN-SIP API in detail. I suggest that you take a look at the implementation of *joinConference()* and *leaveConference()* method in *NoteTaker* and decipher the code by yourself. Hints:

- From inside the *joinConference()* we send the INVITE
- From inside the *processResponse()* we send the ACK (when appropriate).
- From inside the *leaveConference()* we send the BYE.

A bit more configurations for the conference...

First, we need to register a UA to the Asterisk, that can only makes calls to it. This is for the notetaker. So, first, copy and paste the following lines to the end of your *sip.conf*.

```
[notetaker]
type=friend
host=90.0.0.15
port=7000
canreinvite=no
context=notetaker
```

We set the *host* to a specific IP address instead of *dynamic*. The *port*, too, is set to a specific number. The reason is because we want to keep the implementation of *NoteTaker* very simple (we don't want to implement registration step in our code). Therefore, it's important that we run our program on a

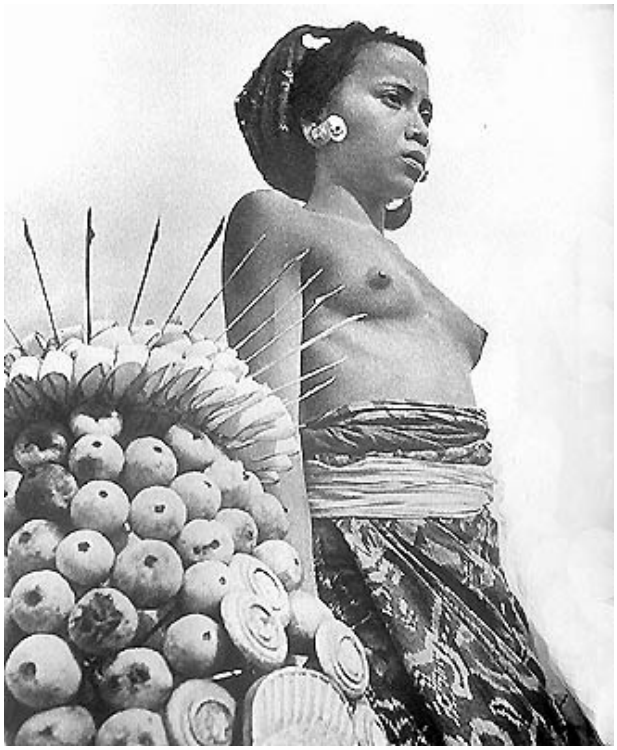
machine whose IP address matches the *host* we specify here, and we must also tell our program – through execution parameters – to bind the *notetaker* to the port we specify here (7000).

We also specify that calls from *notetaker* be put in a separate context named *notetaker*. This is to make sure that calls from *notetaker* – that should be directed toward a conference room – will not be confused with calls from other UAs (that might go to somewhere else). I'll explain, but first copy and paste the following lines to the end of *extensions.conf*.

```
[notetaker]
exten => _XXX,1,MeetMe(${EXTEN},i)
exten => _XXX,n,Hangup
```

Remember that in the *extensions.conf*, for calls that enter the extension 5XXX, we drop the first number from the extension (5) and pass the rest it (XXX) to the *conferenceMonitor* script as a parameter named *roomId*? ... Right, you do remember it :). Well, the *notetaker* will send an INVITE request to XXX@your.asterisk.host.

Now, let's suppose we didn't put the two lines above – anything except *[notetaker]* – in a separate context that is exclusive for calls from UA *notetaker*. And also, suppose that we had a conference room with number 666 (registerd in *meetme.conf*). Then (*bummer!*) the call from *notetaker* would directed to *goldenboy* (assuming that we also have defined extension 666 for *goldenboy* in the same context).



That's all for the SIP.
Let's move on to the last section (*yay!*)..., stress testing with SIPp.

Stress Testing with SIPp

SIPp is an (open source) performance testing tool for SIP protocol. We use it to simulate multiple UAs making a call at the same time. You must have realized that a call session (1) starts with the sending of an invitation (or accepting invitation), (2) is terminated with tear down (sending / receving a BYE), and (3) anything that happen in between. Together, they make a *call scenario*.

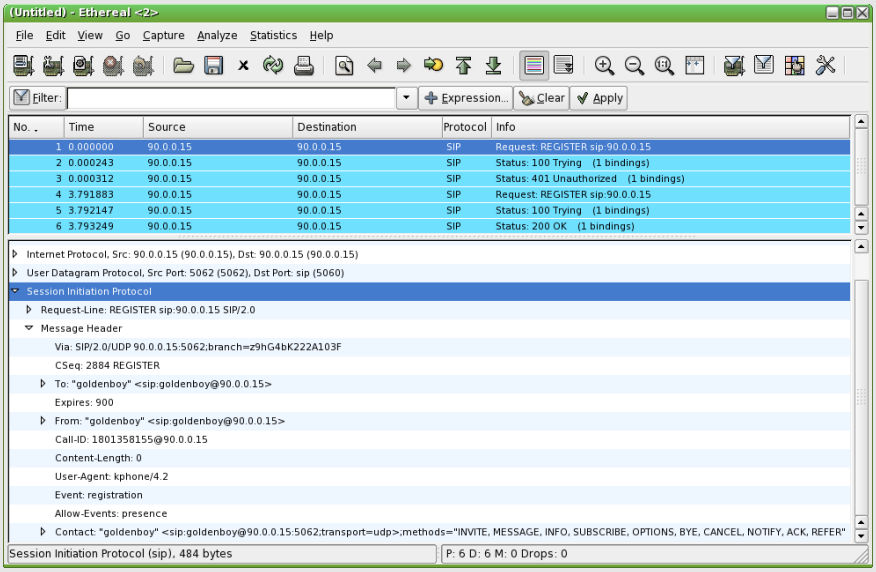
With SIPp, we specify the scenario in an XML file that contains the SIP requests to be sent, and the expected responses. They're processed in document order – from top to the bottom of the document – when the scenario is being played. In the file, a SIP request is enclosed within a <*recv*> tag, while a <*resp*> tag contains information about an expected response. For more information about the content of a scenario file, please consult SIPp documentation [*Ref. 4*].

In our experiment we have two scenario files. One of them represents the scenario from the standpoint of the caller[22], and the other one represents the scenario of the same call from the standpoint of the answerer[23].

Sidenote. How do I know what to write in the scenario files? The answer: copy-and-paste (and a bit of editing) from SIP traffic of a real call, made from SIP phone that I know works (e.g.: *Kphone* and *Xlite*), captured using Ethereal.

For example: I know that Asterisk SIP service was bound to port 5060, and KPhone was bound to another port (let's say 5062). What I did was a copy-and-paste of SIP messages originating from port 5062 and going port 5060 into *caller.xml*, and vice versa for the *answerer.xml*.

Easy, eh? Believe me, Ethereal is a (network) programmer best friend :). Here's a nice screenshot of it:



22 The file named *caller.xml* under *conf/* in the project directory.
23 The file named *answerer.xml*.

In order to make things simpler for the testing, we need to lift the requirement for the caller to authenticate. We do that by deleting / commenting out the *secret* line in the user agent configuration parameters in *sip.conf* (see *Fig. 52*).

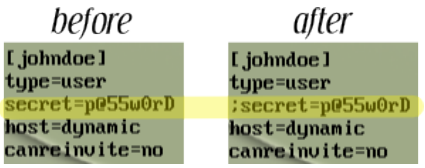


Fig. 52. Comment out the secret line

The other thing we need to do is dropping the requirement for the user agents which are assigned to receive calls to register[24]. This is done by changing the *host* property of the user agent from *dynamic* to an IP number, and by specifying a *port* number in the *sip.conf*. And, just like the caller, we also drop the requirement for the answerer to authenticate. See *Fig. 53*.

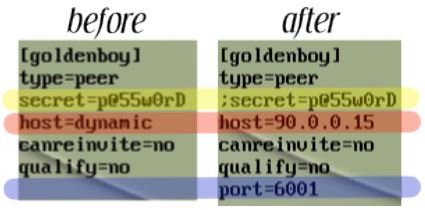


Fig. 53. Change to static registration

Now you're set! I have prepared two script files: *caller.sh* and *answerer.sh*. The following figure shows the content of *caller.sh*.

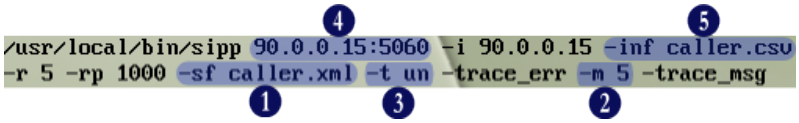


Fig. 54. The content of caller.sh

According to the script: the instance of SIPp will play the scenario specified in *caller.xml* (❶). The maximum number of calls that the SIPp instance will generate is 5 (❷), and each user agent associated with a call will be bound to a randomly selected distinct port (❸). The SIP request will be sent to SIP service running on port 5060 in machine 90.0.0.15, which in our case is the Asterisk's SIP service (❹). Finally, the values to replace the “fields” in the scenario file are read from a file named *caller.csv* (❺), see *Fig. 55*.

24 This is due to current limitation in SIPp, which I'll explain at the end of this section.

caller.csv

```
SEQUENTIAL
johndoe;600;[authentication username=johndoe password=p@55w0rD]
johndoe;666;[authentication username=johndoe password=p@55w0rD]
johndoe;5400;[authentication username=johndoe password=p@55w0rD]
janedoe;5500;[authentication username=janedoe password=p@55w0rD]
jilldoe;5500;[authentication username=jilldoe password=p@55w0rD]
```

(portion of) caller.xml

```
<send>
  <![CDATA[

    INVITE sip:[field1]@[remote_ip] SIP/2.0
    Via: SIP/2.0/[transport] [local_ip]:[local_port];branch=[branch]
    CSeq: [cseq] INVITE
    To: <sip:[field1]@[remote_ip]>
    Content-Type: application/sdp
    From: [field0] <sip:[field0]@[remote_ip]>;tag=[call_number]
    Call-ID: [call_id]
    Subject: sip:[field0]@[remote_ip]
    Content-Length: [len]
    User-Agent: sipp/1.1
    Contact: <sip:[field0]@[local_ip]:[local_port];transport=[transport]>

    v=0
    o=username 0 0 IN IP[local_ip_type] [local_ip]
    s=The Funky Flow
    c=IN IP[media_ip_type] [media_ip]
    t=0 0
    m=audio [media_port] RTP/AVP 0 8 97 3
    a=rtpmap:0 PCMU/8000
    a=rtpmap:3 GSM/8000
    a=rtpmap:8 PCMA/8000
    a=rtpmap:97 iLBC/8000

  ]]>
</send>
```

Fig. 55. Values for fields in caller.xml are read from caller.csv

The other side – *answerer.sh* – is configured to run in *UDP mono socket mode* (*-t u1* command line parameter) that means all the user agents it starts will be bound to the same port number (specified through *-p* command line parameter), which in our case is 6001[25].

Now we're ready to (stress) test the application. Follow these steps:

1. Run our application from Eclipse (or from the command line by execution the *runapp.sh* script)
2. Run the *answerer.sh*
3. Run the *caller.sh*

Good luck, and enjoy. For your reference, here's a screenshot of execution of *answerer.sh* & *caller.sh*.

```
raka@rakahome:~/eclipse-workspace/article_1/sipp> ./answerer.sh
------------------------------ Scenario Screen -------- [1-4]: Change Screen --
  Port   Total-time  Total-calls  Transport
  6001     17.06 s            1  UDP

  0 new calls during 1.007 s period      8 ms scheduler resolution
  1 concurrent calls                     Peak was 1 calls, after 4 s
  1 open sockets

                                 Messages  Retrans   Timeout   Unexpected-Msg
  ----------> INVITE              1         0                   0

  <---------- 100                 1         0
  <---------- 180                 1         0
  <---------- 200                 1         0
  ----------> ACK         E-RTD   1         0                   0

  ----------> BYE                 0         0                   0
  <---------- 200                 0         0
------------------------------ Sipp Server Mode -------------------------------
```

answerer

```
raka@rakahome:~/eclipse-workspace/article_1/sipp> ./caller.sh
------------------------------ Scenario Screen -------- [1-4]: Change Screen -
  Call-rate(length)     Port   Total-time  Total-calls  Remote-host
   5.0(0 ms)/1.000s   5061       5.03 s            5  90.0.0.15:5060(UDP)

  Call limit reached (-m 5), 1.002 s period  8 ms scheduler resolution
  5 concurrent calls (limit 450)         Peak was 5 calls, after 1 s
  0 out-of-call msg (discarded)
  6 open sockets

                                 Messages  Retrans   Timeout   Unexpected-Msg
      INVITE ---------->         5         0
        100 <----------          5         0                   0
        180 <----------          1         0                   0
        200 <---------- E-RTD    5         0                   0
        ACK ---------->          5         0
     Pause [  30000ms]           5                             0
        BYE ---------->          0         0
        200 <---------- E-RTD    0         0                   0
------- Waiting for active calls to end. Press [Ctrl-c] to force exit. -------
```

caller

25 It must match the port number we specify in *sip.conf* for user agents that we assign to answer the calls (e.g.: *goldenboy*).

Analysis of problem with registering from SIPp.
There are at least 2 possible modes of execution of SIPp: server mode and client mode. In the testing scenario for our software, specifically for the case of inbound call, we would like the SIPp instance that simulates the UA (user agent) of a contact center agent to be running in server mode. On the other hand, the SIPp instance that simulates the UA of a customer who makes a call to the contact center should be running in client mode.

When running in server mode, SIPp maintains map of active calls that it is handling. Each call in the map is identified by its Call-ID. Whenever SIPp receives SIP message whose Call-ID doesn't have corresponding entry in the map, it will create a new instance of call (in the memory) and store that call in the map (see *sipp.cpp* line 2060). I use SIPp version 1.1 (unstable) snapshot 2006-01-20 (... and this problem continues to exist in the latest snapshot, 2006-03-15).

When running in client mode, that message will be considered as out-of-call message, and will be treated as such (discarded); see *sipp.cpp* line 2091.

Unfortunately, we can't set the mode through somekind of execution option(s)..., SIPp will figure it out by itself, in an "interesting" way. It will set its mode of execution to server mode only if the first instruction in the scenario file is a <*recv*> (see *scenario.cpp* line 521). Previously we had a problem related to this, because in our old *answerer.xml* the first message is a <*send*> that sends a SIP message for registering the location of the UA of the agent to the Asterisk (such that Asterisk knows where to forward the SIP message to whenever it receives an incoming INVITE from the UA of the customer).

We need to adapt to this situation by removing the need for the UA of the agent to register itself, so we can comment out all the instructions that precedes the <*recv*> for INVITE in the *answerer.xml*.

This can be achieved by modifying */etc/asterisk/sip.conf*. Previously, for each agent in that file, we set the host property to dynamic. Now we need to switch to static mode, by specifying an IP address to that property. Additionaly, we need to specify the port (to which the UA of the agent is expected to listen for SIP messages). In our case, we specify that all agent UAs will be listening to the same (UDP) port: 6001. Correspondingly, we need to set the transport mode of the instance of SIPp for the agent's UA to UDP Mono Socket (in *answerer.sh* we now have the *-t u1 -p 6001* switch).

Analysis of problem with authenticating from SIPp.
The *auth_uri* included by SIPp in the response to the proxy authentication challenge doesn't match the one which is expected by Asterisk. The one generate by SIPp is composed of *remote_ip:remote_port* (e.g.: 90.0.0.15:5060), while Asterisk expects it to be *service@remote_ip:remote_port*.

The *auth_uri* sent by SIPp can be forced to a value specified through *-auth_uri* command line parameter when running SIPp. However, with this trick all the user agents generated by SIPp can only make a call to one SIP URI, that is the value of *auth_uri* parameter.

That's not appropriate in our situation because we want to have multiple agents making calls to different destinations: some of them call 600, the other ones call 666, and the rest call 5400 & 5500.

References

1. JTAPI Whitepapers: http://java.sun.com/products/jtapi/reference/whitepapers
2. Jim Van Meggelen, Jared Smith, and Leif Madsen. *"Asterisk™. The Future of Telephony"*. O'Reilly, September 2005.
3. Presentation Model: http://www.martinfowler.com/eaaDev/PresentationModel.html
4. SIPp 1.1 Reference: http://sipp.sourceforge.net/doc1.1/reference.html
5. RFC 3261: http://www.ietf.org/rfc/rfc3261.txt
6. Asterisk's Wiki: http://www.voip.org/wiki-Asterisk

Resources

1. Source code of the program: http://www.simitel.com/resources/booklet1/article_1.tar.gz
2. Asterisk website: http://www.asterisk.org
3. Asterisk-Java library: http://sourceforge.net/projects/asterisk-java
4. SIPp: http://sipp.sourceforge.net
5. Ethereal: http://www.ethereal.com
6. JTAPI: http://java.sun.com/products/jtapi/index.jsp
7. NIST-SIP library: http://jain-sip.dev.java.net

About the author...

Cokorda Raka Angga Jananuraga, a balinese born in 1978. He moved from Bali to Mexico in 2005, and currently is working for a telecommunication software company named Simitel. He loves to eat the greasy suckling pig (*babi guling*), a traditional balinese food.... What else? Hmm... I guess that's all.

Has been programming with Java since 6 years ago, and still can stand it until now (let's see...).

He posts his rants to his blog at http://www.jroller.com/page/donraka and can be contacted at rakabali78@yahoo.com

Troubleshooting

1. **Q:** My softphone freezes when I tried making call from it.
   A: Is your computer disconnected from the network?
   If yes, make sure that in the *routing configuration* of your network device (ethernet) the *default gateway* is set to none. I haven't got a chance to find out why. It happened to me once, and setting default gateway to none solved my problem.

Table of Contents

I hope you enjoyed reading the document as much as I enjoyed writing it. Also, I hope that you find the document useful, for you, or for member of your team. Please send your feedback (critics, suggestions, etc.) to rakabali78@yahoo.com. Because I need to know whether or not I've made myself clear enough, in an efficient way. I'm looking forward to your feedback! In the meantime, I'll be with these guys preparing *satay*. Cya.