

SOFA SO GOOD

By Anthony Minessale II

I started playing with VOIP for real in late 2000 I began with a quicknet card and all the openh323 stuff on both Linux and windows and had some things going for a while. When it was time to get more serious and use a PBX my Google instinct kicked in and I, of course, found Asterisk all over the place. From day 1 despite it's many options I found it lacking in what I wanted it to do and I immediately rolled up my sleeves and started reading the documentation.... 5 minutes later when I finished reading it all I began digging into the code which at the time, lucky for me, I didn't do C too much so I was not nearly as shocked as I should have been. Since then I have grown fairly proficient in C and have memorized most of the architecture of Asterisk and I have contributed a decent sized list of code and features <http://www.cluecon.com/anthm.html>

So now that I can say "I am properly shocked", the reason I decided to start thinking outside the Asterisk box came from all of the experience I gained trying to fix it. My first question was "Does an admin ***really*** need to learn ***that much*** about C and UNIX to make this thing work decently?" The answer is... "Yes, and then some". This is the most unacceptable thing of all but it's just the tip of the iceberg. I am not so familiar with openh323 and pplib code for 1 simple reason, it worked right away and has volumes of documentation so I never even needed to look at the code =D.

My second question was "Does the feature set need to be co-mingled with the core functionality so tightly?" The answer... "With Asterisk Yes". What happens with this approach is that a hello world module has the chance to crash the box. Do you want a guy who needs to write hello world to have access to the core of your system? Would you let this guy make you kernel modules and happily load them or would you have him make hello.c and run it safely in user space where it is welcome to die without taking the box with it?

Next Question, Who should you trust to solve a problem, People who work on that problem day and night or people who make their own version of it so they can own it? Considering Asterisk is "Open Source" why in the heck doesn't it use any other open source projects for various components? I think we all know why, because they have their finger on the panic button where if all else fails they can sell the rights to the code as-is for a hefty sum and/or license it privately all they want. Hypnotized zealots will surely tell you "so it's easier to manage but that is, of course, as you western hemisphere inhabitants would say Complete Rubbish!

Despite all my efforts to save Asterisk I believe it's time to stop compressions and call it...."Time of death, the minute they drove the last of the developers away." Maybe it can live on in a fork but I think only in a drastic overhaul of the entire architecture and after careful consideration of this idea I

chose “start something else” instead. I want to make a system that can be used on most operating systems if not all, is capable of bringing any media over IP protocol to a common ground where they can be switched and routed together, and uses the smallest possible rule set to make it possible to maintain full control over every component at a core level. This means I want to make the core contain ONLY the functionality and not the features. Once the core can provide pure functionality at a basic level I want to extend control of this functionality to an external source that will have absolute command of every component in an abstract fashion. This means a controlling entity may connect to the system and happily operate the core from a safe distance where any mistakes will not result in catastrophic failure to that core. Because this interaction is done via network communication the system can scale indefinitely since every channel in a call and that call’s controlling entity can exist on its own separate machine if so required. Also Important, the method that the components use to interact with each other and with the controlling entity must remain identical and each component must implement this common functionality above all else before any special considerations are made. This means that a SIP endpoint must be able to interact within the specifications of the abstraction layer first before considering how it may interact with other instances of itself. Once the common layer has been faithfully implemented it is guaranteed that every component can interact with each other at a common level and they will always have something to fall back to in a pinch.

That being said, an endpoint has the option to negotiate communication 2 levels above this basic layer. First there is an instance where the endpoints can agree on a common media or signaling protocol and decide to use it instead of the guaranteed default. Second is the situation where the endpoints determine they are not only the same, they are in the same process or hardware and can cross connect at an even higher level than we can anticipate. When these special communication levels are accessed the controlling entity will continue to receive the same information it always would this procedure is purely media orientated.

The vehicle we have chosen for signaling is Jabber. Jabber has been around for many years and has matured into a very robust messaging protocol. It is often overlooked for uses beyond simple human chat and it is my intention to exploit as much of it’s functionality into telephony as possible while trying to avoid any direct dependency. The design calls for a jabber account for every “master component” or “server process” involved in the system. There is 1 account for SIP another for conferencing and another for the controlling entity. Because Jabber provides the concept of presence and resource every “sub component” meaning individual threads in the master component has their own jabber id derived from the id of the master component it belongs to. The first benefit of this is that an outsider who happens to have a subscription to the master components id will receive presence events for every sub component in that system. So in other words you can watch the channels come up and down as they happen without lifting a finger.

The next benefit is that every thread of execution can avoid having common memory and locking interaction with each other because using jabber they can communicate with each other and it abstracts it to the point that it makes no difference where they are if it's the same process or even the same machine they can interact the exact same way. The other benefit of this approach is that the controlling entity also has a jabber id which you can use to route resources such as calls to a specific controller to decide the final outcome. For example, a call reaches a SIP endpoint which is registered on our system. As soon as that call is received the endpoint contacts the controller it has been designated to report all incoming calls to. That controller (which can be a Perl script for all we know since it's just another jabber id) looks at an encapsulated packet of information that is as simple as an everyday http request. Based on this event it decides that the call should be forwarded to the PSTN via the TDM gateway. It then sends a jabber message to the master id of the TDM gateway and requests a channel bound for the SIP channels destination. If/When the channel is accepted the controller sends a message to the SIP channel telling it to prepare a media bridge and notifies it of the TDM channel's jabber id. The SIP channel then sends a message to the TDM channel and since they have little in common they agree to use the default abstraction layer discussed above and arrange a simple network media loop to each other. If the call was busy the information is passed to the controller who can end the call at that time or choose to try another destination or whatever else it decides.

A second scenario with the same inbound SIP call to say, an IVR: The main controller decides that the DNIS on that call means the call should be forwarded to an IVR so it sends a jabber message to the SIP call telling it to report to that specific IVR's jabber id. When the IVR receives this message it decides it must fork a dedicated process for this interaction so it does and advises the call to now report to the new sub id. This IVR (almost a bot) now has full control over the channel and also has the capability to request more outbound channels to interact with. For instance it can command the call to hear a certain file played then all the while it accepts DTMF events and decides the correct combination of digits has been dialed to warrant connection to an external source. Once the new channel is up it can decide to maintain control of this call monitoring events all the while or instruct both legs of the call to report back to the master controller and it can safely exit. This passing of control can happen as many times as needed by any leg of the call without disturbing the call in any way.

To summarize, the main goals of my efforts involve:

- Implementing a scalable solid core where nobody ever looks at the code because it works right away, "thanks again Craig".

- Making sure the functionality is available from a safe distance to make sure an admin has full control but is not able to crash the system and has artistic freedom to concentrate on his features without that minimum requirement of a Masters Degree in computer science to configure his application.
- Using the best components available to solve each individual problem. There are enough problems to work on we need to eliminate as many as possible with existing technology so we can focus mostly on the unsolved or poorly solved problems.