How does FreeSWITCH compare to Asterisk?

How does FreeSWITCH compare to Asterisk? Why did you start over with a new application? These are questions I've been hearing a lot lately so I decided to explain it for all of the telephony professionals and enthusiasts alike who are interested to know how the two applications compare and contrast to each other. I have a vast amount of experience with both applications with about 3 years of doing asterisk development under my belt and well, being the author of FreeSWITCH. First I will provide a little history and my experience with Asterisk, then I will try to explain the motivations and the different approach I took with FreeSWITCH.

I first tried Asterisk in 2003. It was still pre 1.0 and VoIP was still very new to me. I downloaded and installed it and in a few minutes I was tickled pink over the dial tone emitting from my phone plugged into the back of my computer. I spent then next few days playing with my dial plan and racking my brain to think of cool stuff I could do with a phone that was hooked up to a Linux PC. Since I had done an extensive amount of web development in my past life I had all sorts of nifty ideas like matching the caller id to the customer's account number and trying to guess why they were calling etc. I also wanted to move on in my dial plan based on pattern matching and started hacking my first module. Before I knew it I had made the first cut of app_perl, now res_perl where I had embedded a Perl5 interpreter in Asterisk.

Now that I had that out of my system, I started developing an Asterisk-driven infrastructure to use for our inbound call Queues. I prototyped it using app_queue and the Manager Interface now proudly dubbed "AMI" (initials always make things sound cooler). It was indeed magnificent! You could call in from a PSTN number over a T1 and join a call queue where our agents who also called in could service the calls. "This rocks!" I thought to myself as I watched from my fancy web page showing all the queues and who was logged in. It even refreshed periodically by itself which was why I was surprised when the little icon in the corner of my browser was still spinning for quite some time. That's when I first head it. That word. The one I can never forget, deadlock.

That was the first time, but it wasn't the last. I learned all about the GNU debugger that day and it was just the first of many incidents. Deadlock in the queue app. Deadlock in the manager, Avoiding Deadlock on my console. It was starting to get to me a little but I kept going. By this time I was also quite familiar with the term Segmentation Fault another foe to the computer developer. After about a year's time wrestling with bugs I found myself a lot more well-versed in the C programming language than I even imagined and near Jedi caliber debugging skills. I had a working platform running several services on a DS3 worth of TDM channels spread over 7 asterisk boxes and I had given tons of code to the project including some entire files on which I hold the copyright. http://www.cluecon.com/anthm.html

By 2005, I had quite a reputation as an asterisk developer. They even thanked me in both the CREDITS file and in the book, Asterisk, The Future of Telephony. I not only had tons of applications for asterisk in tree, I had my own collection of code they did not need or want on my own site. (Still available today at http://www.freeswitch.org/node/50) Despite all of this I could not completely escape the deadlocks and crashes. I hid the problem well with restart scripts and 7 machine clusters but I could not see a way to scale my platform much more. I had to abandon some features because they just would not work right based on the way Asterisk was designed.

Asterisk uses a modular design where a central core loads shared objects to extend the functionality with bits of code known as "modules". Modules are used to implement specific protocols such as SIP, add applications such as custom IVRs and tie in other external interfaces such as the Manager Interface. The core of Asterisk is a threading model but a very conservative one. Only origination channels and channels executing an application have threads. The B leg of any call operate only within the same thread as the A leg and when something happens like a call transfer the channel must first be transferred to a threaded mode which often times includes a practice called channel masquerade, a process where all the internals of a channel are torn from one dynamic memory object and placed into another. A practice that was once described in the code comments as being "nasty". The same went for the opposite operation the thread was discarded by cloning the channel and letting the original hang-up which also required hacking the cdr structure to avoid seeing it as a new call, you will often see 3 or 4 channels up for a single call during a call transfer because of this.

```
/* XXX This is a seriously wacked out operation.  We're essentially putting the guts of
      the clone channel into the original channel.  Start by killing off the original
      channel's backend.   I'm not sure we're going to keep this function, because
      while the features are nice, the cost is very high in terms of pure nastiness. XXX */
```

This became the de facto way to pull a channel out of the grips of another thread and the source of many headaches for application developers. This uncertain threading scheme was one of the motivating factors for a rewrite.

Asterisk uses linked-lists to manage its open channels. A linked-list is a series of dynamic memory chained together by using a structure that has a pointer to its own type as one of the members allowing you to endlessly chain objects and keep track of them. They are indeed a useful programming practice but when used in a threaded application become very difficult to manage. One must use mutexes, a kind of traffic light for threads to make sure only 1 thread ever has write access to the list or you risk one thread tearing a link out of a list while another is traversing it. This also leads to horrible situations where one thread may be destroying or masquerading a channel while another is accessing it which will result in a Segmentation Fault which is a fatal error in the

program and causes it to instantly halt which, of course means in most cases all your calls will be lost. We've all seen the infamous "Avoiding initial deadlock" message which essentially is an attempt to lock a channel 10 times and if still won't lock, just go ahead and forget about the lock.

The manager interface or AMI has a concept where the socket used to connect the client is passed down into the applications letting your module have direct access to it and essentially write any data you want to that socket in the form of Manager Events which are not very structured and thus the protocol is very difficult to parse.

Asterisk's core has linking dependencies on some of it's modules which means that the application will not start if a certain module is not present because the core is actually using some of the binary code from the module shared object directly. To make a call in asterisk in at least version 1.2 you have no choice but to use app_dial and res_features because the code actually lives in those modules. The logic to establish a call and to do things like a forked dial actually reside in app_dial not the core, and res_features actually contains the top level function that bridges the audio.

Asterisk has no protection of its API. The majority of the functions and data structures are public and can easily be misused or bypassed. The core is anarchy with assumptions about channels having a file descriptor, which is not always necessary in reality but is mandatory for any asterisk channel. Many algorithms are repeated throughout the code in completely different ways with every application doing something different on seemingly identical operations.

This is only a brief summary of the leading issues I had with Asterisk. I donated my time as a coder, my servers to host the CVS repository and served as a bug marshal and maintainer. I organized a weekly conference call to plan for the future and address some of the issues I have described above. The problem was, when one looks at this long list of fundamental changes then thinks about how much work it would take and how much code may have to be erased or rewritten, the motivation to address the issues begins to fade. I could tell not many people would be on board with my proposal to start a 2.0 branch and rewrite the code. That is why in the summer of 2005 I decided I would do it myself.

My primary focus on FreeSWITCH was to start from the core and trap all the common functionality under the hood and expose it in a pyramid to the higher levels of the application. Like Asterisk, the Apache Web Server heavily inspired me and I chose to use a modular design. From the first day the basic fundamentals I chose to adhere to were that every channel has it's own thread no matter what it was doing and that thread would use a state machine function to navigate its way through the core. This would ensure that every channel would follow the same predictable path and state hooks and overrides could be placed into the machine to add important functionality very similar to how methods and class inheritance works in an object oriented programming language.

It hasn't been easy. Let me tell you. I've had my fair share of Segmentation Faults and Deadlocks while coding FreeSWITCH , (a lot more of the former than the latter I must say). But I built the code from the core and went from there. Since all of the channels operate in their own thread and there are occasions where you need to interact with them, I use read/write locking so the channels can be located from a hashing algorithm rather than a linked list and there is an absolute guarantee that the channel cannot be accessed or go away while an outside thread has reference to it. This alone makes it much easier to sleep at night and obsoletes the need for "Channel Masquerades" and other such voodoo.

The majority of functions and objects supplied by the FreeSWITCH core are protected from the caller by forcing them to be used the way they were designed. Any concept that is extensible or provided by a module has a specific interface which is used to front end that functionality therefore the core has no linking dependency on any of its modules. There is a clear cut layered API with the core functions being on the bottom and the amount of functions on each subsequent layer decreasing as the functionality increases. For instance it's possible to write a large function that uses an arbitrary file format module to open and play audio to a channel. But in the next layer of API there is simply a single function that will play a file to a channel that is then extended to the dial plan tools module as a tiny application interface function. So you can execute the playback from your dial plan, from your custom C application using the same function or you can write your own module that manually opens the file and plays it all using the services of the file format class of modules without ever divulging it's code.

FreeSWITCH is broken into several module interfaces. Here is a list of them:

Dialplan:
: Implement the ring state of a call, take the call data and make a routing decision.

Endpoint:
: Protocol specific interface SIP, TDM etc.

ASR/TTS:
: Speech recognition and synthesis.

Directory:
: LDAP type database lookups.

Events:
: Modules can fire existing core events as well as register their own custom events Which can be parsed from an event consumer at a later time.

Event Handlers:
Remote access to events and CDR.

Formats:
File formats such as wav.

Loggers:
Console or file logging.

Languages:
Embedded languages such as Python and JavaScript.

Say:
Language specific modules to construct utterances from sound files.

Timers:
Reliable timers for packet interval timing.

Applications:
Applications you can execute on the call such as Voicemail.

FSAPI (FreeSWITCH API interface [see I use initials too!])
Command line functions, XMLRPC functions, CGI type functions, Dialplan function variables exposed with a string in, string out prototype.

XML
There are hooks to the core XML registry that make it possible to do realtime lookups and create XML based CDRs

All of the FreeSWITCH modules work together and communicate with each other only via the core API and the internal event system. Great care was taken to ensure this and avoid any unwanted behavior from outside modules.

The event system in FreeSWITCH was designed to keep track of as much as possible. I designed it under the assumption that most users of the software would be connecting to FreeSWITCH remotely or using a custom module to gather call data. Thus, every important thing that happens in FreeSWITCH results in an event firing. The events are very similar to an email format having headers and a body. Events can be serialized into either a standard text format or an XML representation. Any number of modules may be written to connect to the event subsystem and receive events about presence, call state and failures. The in-tree mod_event_socket provides a TCP connection on which events can be consumed as well as log data. In addition call control commands may be sent over

this interface as well as bi-directional audio flow. The socket can be established by either an in-progress call as an outbound connection or from a remote machine as an inbound connection.

Another important concept in FreeSWITCH is the centralized XML registry. When FreeSWITCH loads it opens a top-level XML file which is fed into a pre-processor that parses special directives to include other smaller xml files and to set global variables which can be referenced from that point forward to template the configuration.
For instance you can set the preprocessor directive to set a global variable like this:

<X-PRE-PROCESS cmd="set" data="moh_uri=local_stream://moh"/>

now even on the next line in the file you can use $${moh_uri} and it will be replaced by local_stream://moh in the post processed output. The final post processed registry is loaded into memory and accessed by the modules and the core to provide several vital sections to the application:

Configuration
: Configuration data to control the behaviour of the application.

Dialplan
: An XML representation of a dialplan that can be used by mod_dialplan_xml to route calls and execute applications.

Phrases
: A markup of IVR phrase macros to use from IVRs and to speak multiple languages.

Directory
: A collection of domains and users for registration and account management.

Using XML hook modules, you can bind your module to lookups in the XML registry and, in real time, gather the required information and return it to the caller in place of the static data in the file. This makes it possible to do purely dynamic SIP registrations and dynamic voice mailboxes and dynamic configuration of a cluster using the same model as a web browser and a CGI application.

With embedded languages such as JavaScript, Java, Python and Perl, it's possible to write scripted application that can control the underlying power with a simple high-level interface.

The first phase of the FreeSWITCH project was to create a stable core on which to build scalable applications. I am happy to report that it will be completed on May 26th 2008 with the release of FreeSWITCH 1.0 "phoenix". We have been able to out perform

Asterisk by a factor of 10 in similar situations according to the accounts of two separate early adopters brave enough to go into production pre-1.0.

I hope this explanation is sufficient to outline the difference between FreeSWITCH and Asterisk and will shed some light on my decision to start the FreeSWITCH project. I will forever remain an Asterisk developer due to my vast involvement in the project and I wish them all the luck in the world with the future design of the application. I may even dig up some more of my long lost Asterisk code in my personal archives and release it to the public as a gesture of good will towards the project that gave me my start in telephony.

Asterisk is an open source PBX and FreeSWITCH is an open source soft switch. There is plenty of room for both applications among the other great open source Telephony applications such as Call Weaver, Bayonne, sipX, OpenSER and many many more. I look forward every year to presenting with and talking to all the developers of these projects at ClueCon in Chicago this summer. http://www.cluecon.com

We can all inspire each other to push the envelope on Telephony even farther. The most important question you can ask is. "Is it the right tool for the job?"