Soft-phones and hard security

Tim Panton
Westhawk Ltd

http://www.westhawk.co.uk

2006-03-29

Abstract

We have designed and implemented a Java based web applet that acts as a soft-phone for the Asterisk Open Source PBX.

This paper describes the difficulties (many) and compromises (few) that were encountered in the development process.

Most of these problems related to the security, networking or threading requirements of hosting the application in a browser. Our aim in presenting this paper at SANE is to assist systems and network professionals in their discussions with developers about what is possible vs what is acceptable in a secure, portable, low maintenance but compelling web based application.

1 Background

Westhawk Ltd have replaced our old ISDN PBX with a PC running Linux and Asterisk [1] Voice Over IP (VOIP) software and some VOIP handsets. Asterisk offered us the possibility to work at home or on site and still be reachable on our normal office numbers. To do this we needed a program we could run on our laptops and home PCs that would act as a VOIP handset - a soft-phone. We had the following criteria :

- Usable. It had to work well enough that the other end of the call would not know we were using a soft-phone
- Portable. We had a variety of systems we needed to support, including Windows 98, 2000, XP, SuSE 10 and Mac OS X
- Maintainable. We wanted to be able to control and configure the soft-phones centrally
- Asterisk compatible. We needed it to work with Asterisk
- Deployable. We wanted to be able to use it wherever we had an internet connection

We tried a few soft-phones and were not satisfied with any of them, they all exhibited one or more of the following problems:

- Only supports a single platform (most often Windows only)
- Requires installation of a program or ActiveX control
- Requires extensive changes to local firewalls. One of the better products requires the following ports to be permitted/forwarded through the firewall:
  - SIP : UDP 5060
  - RTP: UDP 8000-8010 (even numbers)
  - RTCP: UDP 8001-8011 (odd numbers)
  - STUN: UDP 3478

- Keeps configuration locally on the user's disk

As people with a network security and systems administration background we were horrified by our lack of acceptable options.

In true programmer fashion we decided to try and write a better soft-phone.

2 Human ears and the Internet

The human ear is quite a sensitive instrument. Most users can detect pitch shifts, delays and missing or miss-ordered data in an audio stream. When talking to someone they know well, these things are even more noticeable to a user and can prove very distracting.

Most Internet services are layered on top of TCP/IP, since this gives a guarantee of data delivery and ordering. TCP does this by detecting any missing or out of order IP packets and sends a retry request to the sender. The sender re-sends the missing data and all is well. However, whilst this is happening TCP is unable to pass any more data up to the application layer until the missing data has arrived. This hiatus will be (at least) equal to the round-trip time of the connection. On a LAN this would be fractions of a millisecond, which is inaudible. On a WAN (even on broadband) this would be at least 30ms, which is audible. For this reasons all VOIP services use UDP to carry their audio traffic and employ 'packet loss concealment algorithms' to mask any missing data.

UDP packets may arrive early, late, in the wrong order, or not at all, it is up to the application to cope with this. Most of these problems could be worked around by using deep buffers on the receive side. Each packet contains 20ms of audio data, buffering more than a very few packets results in an audible delay. We found that 5 packets (100ms) was an acceptable compromise. Any packets that are miss-ordered or delayed can therefore be inserted in the stream provided they arrive within the 100ms. The 20ms is a typical value used in many VOIP protocols. It represents a compromise between audio quality and bandwidth efficiency. Shorter times would result in better audio, since a single missing packet would be less audible, longer times would be more efficient as there would be fewer packets per second and hence lower overheads. Even at 20ms the overhead is significant, a single 20ms GSM [2] encoded sample takes 65 bytes of which the combined IP+UDP+VOIP headers are 32 bytes - almost half!

We found that the audio hardware in most PCs is of variable quality. One issue that caused us some considerable difficulty is the issue of clock skew. Imagine that the clock chip on the sending PC is running 2 percent faster than that on the receiving end. It would send 1 packet more per second than the receiver was expecting. If the receiver were to buffer up this excess, by the end of a 10 minute conversation there would be 12 seconds of conversation in the buffers! Clearly this would make the conversation awkward! There are two ways to avoid this, the first is to play the data faster by resampling the sound and in effect transposing the pitch. This is computationally expensive and might lead to pitch changes during a conversation. Instead we took the option of removing single datapoints from the sample steam. If we remove a maximum of two samples per buffer (160 samples) we can correct for up to 1.6 percent clock skew, which is adequate for the majority of hardware. Deviations beyond this cause us to drop a whole packet once all our buffers are full, which causes an audible click, fortunately most hardware is within this tolerance.

3 Write once, test everywhere

Westhawk have been writing portable software for 20 years, initially in C but in the last 10 years we have been working in Java, including open source projects in the Computer Telephony field [3] and low level network protocols [4] so it was a natural decision for us to try writing a soft-phone in Java. We have generally had a very positive experience of multiplatform development in Java, our SNMP stack [5] works well on at least 5 platforms, but there are always differences between platforms which need to be tested and worked around. We found that the closer to the hardware we get, the more the differences emerge. The sound system on Mac OS X for example only makes one sampling rate available to Java 44.1Khz, where as PCs all offer multiple rates, including the 8Khz rate we require. The JavaSound infrastructure does offer plugins that will resample data to change the sample rate, but these plugins increase the size of the application, complicate the licensing position and add to the audio latency, so we have written a 'quick and dirty' re-sampler that is only invoked on platforms that don't offer a 8Khz sample rate. We also elected to write new classes to implement audio codecs for similar reasons.

Where there is a common method that works across all the platforms we use it, especially if it is the defined standard. An example of this would be the use of `document.getElementById('name')` in Javascript rather than the Internet Explorer specific `document.all.name`.

Where no common method existed, or using it would compromise the 'better' platforms we wrote code to detect the platform and select a code path based on the platform. An example of this would be that the JavaSound implementation in Java 1.4.2 requires bigger input buffers than that in Java 1.5. We could have set the larger buffer size for both platforms, but this would have increased the latency for all platforms, so we elected to adjust the size based on the Java version.

Java has a reasonably complete threading implementation, allowing us to run separate loosely coupled threads for network and audio IO, along with a few housekeeping threads. There were a couple of pitfalls we had to avoid. The first was that all the threading operations (`wait()` `Timer` etc) are based on the system clock, not on the audio clock. As noted above the audio clock is often significantly different from the system clock. We therefore had to implement some scheduling that corrected for those differences. Java's threads can run at multiple priorities, but the highest of these corresponds to the default priority that an unthreaded C program would run at on the same machine. Thus anything less than `MAX_PRIORITY` will result in a Java thread being pre-empted by other programs on the system, possibly causing breaks in the audio. We have set all the critical threads to run at `MAX_PRIORITY`.

In order to achieve a reasonably smooth flow of data, as required by the VOIP protocol, we had to ensure that the JavaSound infrastructure never had to block waiting for data. If it does block, JavaSound blocks until its buffers are full. We found that on most platforms we had to allow 60ms of buffers in order to get clear sound. Blocking any of the main threads for 60ms would have been unacceptable.

4 Web 2.0 and Java 1.0

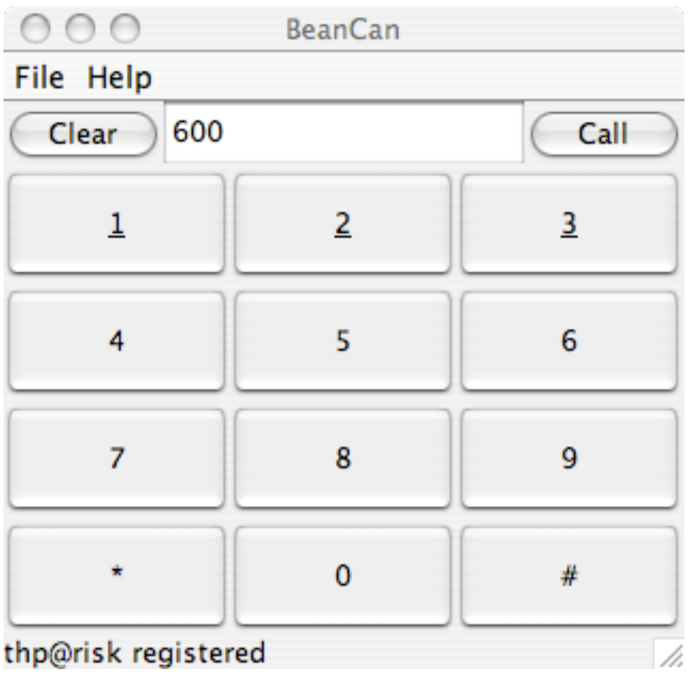
Having decided to use Java to implement the phone, this left us with some interesting options as to how to deploy, configure and maintain the soft-phone.

We could have simply shipped the soft-phone as an application, to be installed and run like any other application. This would have entailed writing an installer (per platform) and saving configuration (user credentials, server settings etc) to the local disk. This would have cost us much more time to develop and have been harder to support and maintain.

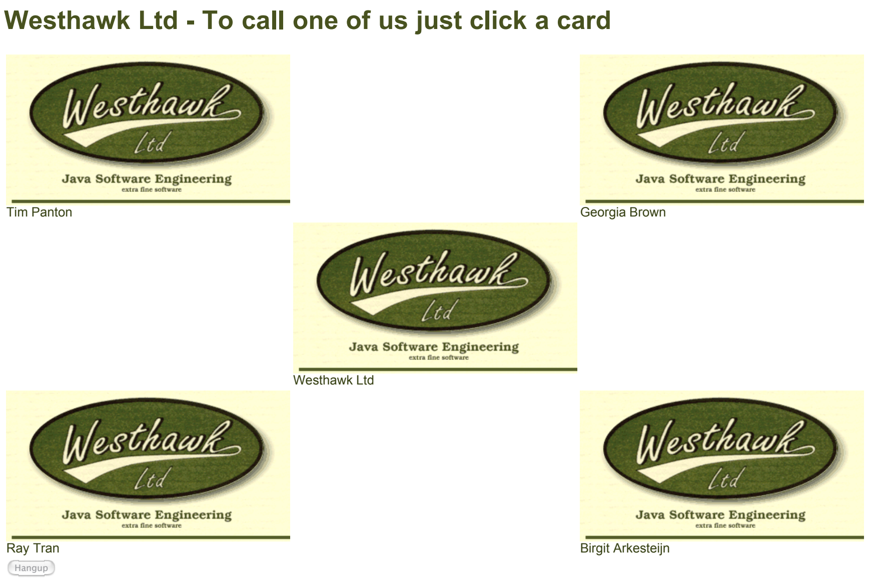
Java also supports the concept of a "Web start" where an application is downloaded, run and updated via a web browser, this somewhat reduces the work needed to write installers etc. but still leaves the configuration problems.

From its early days Java (version 1.0) has been able to run in most web browsers, in the form of Applets. Applets have some restrictions (of which more later), but they are very easy to maintain centrally. They are deployed from a central web server and generally leave no configuration files on the user's system. They don't have to be installed. The most recent version is automatically fetched from the web server whenever the enclosing web page is visited. More recently Applets have been able to interact directly with the enclosing page, triggering (and being triggered by) JavaScript methods, which in turn can alter the look and behavior of the HTML on the page. This style, when combined with the ability of JavaScript to fetch data updates in the form of XML fragments from a web server, produces much more interactive web pages that don't have to be completely re-drawn. It is often referred to as AJAX or Web 2.0.

Initially we wrote the soft-phone as a normal Java applet, with a full 'traditional' GUI written in Swing:



But this didn't really take advantage of all the possibilities in modern web browsers and clashed visually with almost any web page it was in, so we removed the swing GUI and added JavaScript methods to replace the needed functionality.


Westhawk Ltd - To call one of us just click a card
Westhawk Ltd Java Software Engineering extra fine software
Tim Panton
Georgia Brown
Westhawk Ltd
Ray Tran
Hangup
Birgit Arkesteijn

The look and feel is only limited by the skills of the web designer (me in the above case - it shows).

5 Theory meets the modern internet

In order to use our soft-phone with Asterisk, it has to talk one of the protocols that Asterisk supports.

5.1 MGCP

Wikipedia says: "MGCP is interesting because, although the Call Agent acts as a software switch for a VoIP network, it stays conspicuously uninvolved with the actual dirty work of encoding, decoding, and transferring the sound data." We decided not even to look at implementing an MGCP soft-phone.

5.2 SCCP

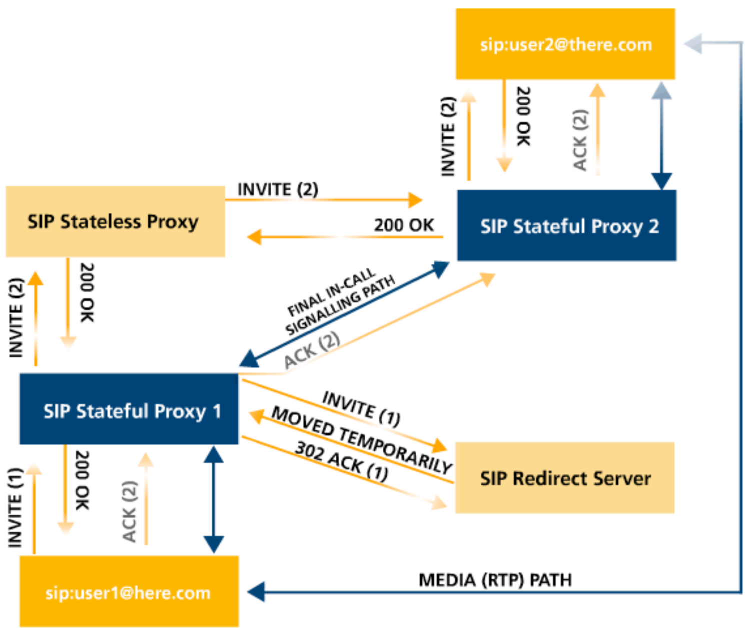
SCCP is a Cisco VOIP protocol with 2 implementations in Asterisk. Implementing SCCP is only possible if you have never signed a Cisco license, since such licenses forbid reverse engineering. Unfortunately there isn't enough information in the public domain to implement SCCP without reverse engineering.

5.3 H.323

H.323 is interesting protocol, but it is losing ground to the newer protocols. It still dominates in the video-conferencing area, and is well documented. It suffers from many of the same problems that SIP does in relation to firewalls and NAT.

5.4 SIP

Session Initiation Protocol.



SIP is a successful standard which defines how to set up and take down realtime sessions. The sessions may be chat, video, audio or of other types of communication. SIP delegates the actual transport of the session itself to a second protocol - RTP.

As can be seen in the diagram, the path taken by the RTP streams may be different from the path taken by the SIP packets that initiated the call. Indeed the endpoints may never have directly exchanged packets since both ends may have negotiated the connection via a proxy. On an unsegmented, unfirewalled LAN, this works very well, but modern WAN connections (especially home DSL connections) involve NAT and firewalls.

When SIP creates a bidirectional audio connection, packets are sent which specify the source and destination addresses and ports for the RTP streams in each direction. It is this exchange that causes the biggest problems. In an environment with a NAT router, the address that a host thinks it has will be different from the address that the far end would need to use to access it. NAT friendly SIP devices work around this by using an additional protocol - STUN - which notifies the sender of the external address that they are using. The NAT friendly SIP device then uses that value when referring to its own address. Smarter devices allow an exception for 'local' addresses (i.e. those also inside of the NAT router) for which they use their 'normal' address.

The ports that SIP can specify for the RTP streams can vary widely, and the streams are bidirectional, one is outgoing and the other is incoming. This makes SIP/RTP almost impossible to firewall well without inspecting the SIP packets and dynamically opening holes in the firewall according to the packet specification.

We wanted to have a simple set-up in a range of environments. We doubted that we could achieve that with SIP/RTP

5.5 IAX

IAX [6] is the 'native' protocol for Asterisk. It was designed to connect multiple Asterisk systems. It is closely related to the frame format that Asterisk uses internally, which in turn is based on ISDN protocols. IAX was designed after the widespread deployment of NAT and firewalls, and it copes with them much better than SIP.

All IAX traffic between 2 hosts is multiplexed over a single UDP session. The server has a single 'well known' port (4569). Sessions are initiated by a client sending a packet to the server. In the case of a client phone registering to receive calls it has to re-register often (or qualify the link), this preserves the port mapping in any NAT router or firewall which is keeping state.

IAX does not need a separate STUN server as the reply to an initial packet includes the apparent address of the sender.

IAX is not well supported outside the Asterisk community, but since we had already committed to using Asterisk, we decided to use IAX in our soft-phone.

When we started work on the protocol the RFC draft was somewhat incomplete so we had to use some reverse engineering and ask questions of the developers to fill in the gaps. Subsequent revisions of the RFC are more complete and we now believe that it is possible to implement IAX just from the RFC.

IAX also supports encryption of the datastreams, but not the control channel. We have started work on implementing this. Hopefully we will be complete by the time you read this.

In order to minimize the bandwidth required by multiple streams of data between a pair of hosts IAX supports the concept of 'trunking', which in effect is sharing the packet headers over multiple datastreams. Our use case is single connections from multiple 'client' systems to a single server, so we don't see any benefit in trunking in our case.

6 Security and Usability

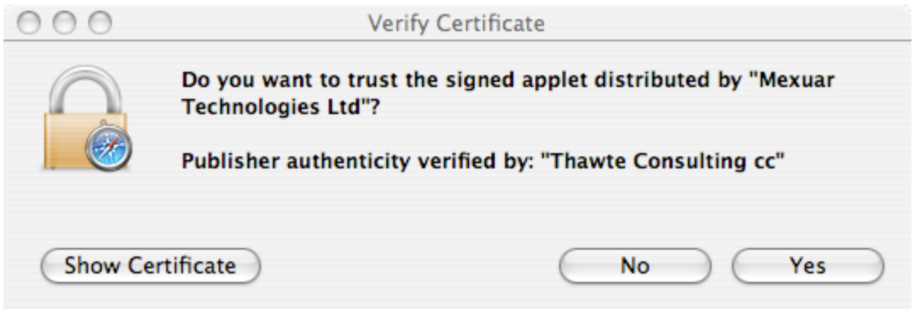
We wanted to ensure that using our soft-phone was as quick and easy as possible, so that people would accept it, ideally without needing to think too much about it.

We had decided to write the phone in Java and deploy it as an applet, to be run within the user's web browser. Java's (in)famous sandbox model severely restricts what a Java applet can do. It specifically prevents an applet from doing two things we need to be able to do in our soft-phone.

- Applets cannot read audio data from the microphone or line-in on the user's system
- Applets cannot communicate with systems other than the one which served the applet

We could (in principle) have worked within the second of these restrictions by ensuring that the Asterisk PBX was also the web server serving the applet. The ability to read audio data however is an absolute necessity in a phone!

Applets can 'escape' the sandbox if they are digitally signed and the user elects to accept that signature and to grant the applet additional permissions.


Verify Certificate

Do you want to trust the signed applet distributed by "Mexuar Technologies Ltd"?

Publisher authenticity verified by: "Thawte Consulting cc"

Digitally signing an applet isn't quite as simple as it sounds. First you have to obtain a digital certificate that is recognized by the common browsers. These certificates are not the same as the certificates used to sign SSL protected web sites. Certificates suitable for code signing are more expensive and the checking of the applicant is a bit more rigorous.

It is important to understand that a signed applet still operates within the Java security framework, it just has more permissions than a normal applet. This has practical consequences when it comes to the interaction between the applet and JavaScript on the web page containing it. The book [7] on Java security describes the implementation of the Java security model in the following pseudo-code:

```
for each caller in the current execution context {
     if (the caller does not have the requested permission)
        throw an AccessControlException
}
return normally;
```

When the user presses a button (say Dial) on the web page, this invokes a JavaScript function, which in turn calls a Java method within the applet. This method ends up trying to send a packet to Asterisk. Whilst the applet has been signed and has permission to send the packet, the JavaScript isn't signed and has no such permission. The Java security framework applies the above rule to JavaScript too, and refuses to permit the action.

We work around this problem by taking all external calls from JavaScript checking them and then creating tasks that are added to a queue of tasks. On initialization the applet starts a (privileged) thread that takes tasks off the queue and executes them within its own context.

Unlike Ruby (and Perl) Java does not have the concept of 'tainted' objects. In Ruby the task we create in the above would be considered 'tainted' as it is based on data from an untrusted source.

We have (for commercial reasons) chosen to hard code the IP address of the Asterisk server into the applet before we sign it. This restricts the demo version of the applet to talk to our demo Asterisk server only. For maintainability we decided to put the IP address in a simple bean class. The main applet would then query the `MyHost` class to obtain the address of the host to connect to. We were surprised to find that adding an additional (unsigned) class of the same name to the applet's classpath allowed a user to set the host to any value they wished but didn't break the signing. We fixed this problem by ensuring that the `MyHost` object remained in the call stack and its unsigned nature was therefore considered by the Java security framework when assessing permissions.

7 Licenses

The intellectual property involved in this development is now owned by Mexuar Technologies Ltd.

Our intention is to make as much as possible of the applet code Open Source, probably under the GNU Public License. At the moment however we are unable to do so for a number of commercial and legal reasons.

Since our code is not GPL we have been obliged to remove all GPL software from the applet. We had been using a codec that was GPL licensed, but we have now re-implemented it by going back to the standards description and starting afresh in Java. This turned out to be to our advantage as the applet is now smaller, and no longer needs to use the JavaSound codec infrastructure, which added some latency to the sound due to the way that it did buffering.

8 Summary

We started with what seemed like a reasonable set of requirements, to produce a mobile, zero install, easy to use soft-phone for our Asterisk PBX. We rapidly found that all the available products were unsatisfactory in one or more ways. We then decided that we could do better and set out so to do. We discovered that there were some difficult issues, especially in the area of security and deployment, but we managed to work around them. Our message to systems administrators and network security professionals is not to accept the unacceptable, encourage your developers and suppliers to work harder to produce maintainable, secure products.

References

[1] Van Maggelen, Smith and Madsen. *Asterisk - The Future of Telephony*, ISBN: 0-596-00962-3.

[2] ETSI. *GSM 06.10 version 8.1.1 Release 1999* , ETSI EN 300 961 V8.1.1 (2000-11).

[3] Generic JTAPI. Deadman, Panton et al , http://gjtapi.sourceforge.net/

[4] William Stallings. *SNMP, SNMPv2, SNMPv3 and RMON 1 and 2*, ISBN: 0-210-48534-6.

[5] Westhawk Ltd. *Westhawk's SNMP stack in Java*, http://snmp.westhawk.co.uk/

[6] Spencer, Capouch, Guy, Miller and Shumard. *IAX: Inter-Asterisk eXchange Version 2*, http://mirror.switch.ch/ftp/mirror/internet-drafts/draft-guy-iax-01.txt

[7] Gong. *Inside Java 2 Platform Security*, ISBN 0-201-31000-7