Anatomy of an APRS-IS Server

(The Evolution of javAPRSSrvr and Its Adjuncts)

In 2002, APRS-IS (APRS on the Internet) was in disarray and bordering on collapse. There were three core servers through which every packet in APRS-IS was supposed to pass. first.aprs.net was running aprsd on Linux, second.aprs.net was running APRServe on Mac OS 9, and third.aprs.net was running aprsd on FreeBSD Unix. There were over one hundred non-core servers, most running various versions of aprsd and some running AHub. So, what were the critical problems?

Many problems were interrelated. The core server reliability was less than 1 day of uptime. Packets were looping extensively. Many servers and IGates were changing data causing dupe checking algorithms to break. IGates were gating mangled packets to the Internet. The data flow had grown to a point which was causing many clients to crash or bog down after short connection periods.

I proposed a new server to attempt to address many of these problems and to try to bring stability to APRS-IS. The server had some basic requirements which had to be met if it was going to be of any value. It had to support many diverse operating systems since there was little commonality of OS's, even at the core. It had to be backwards compatible with existing servers and clients. It could clean headers, but the information field of the packet must not be altered in any way. It had to be easily modifiable, upgradeable, and expandable.

Beyond those base requirements, it also had to programmatically promote good network design within the constraints of the existing APRS-IS framework. To this end, multiple bidirectional outbound connections could not be allowed. Inter-hub connections could be restricted to only allow specified servers to interconnect using port 1313. All basic available port types would also need to be supported. Even with these restrictions, there would be no mandatory configuration requirements; everything would be at the control of the sysop.

To meet these goals I chose Java as the programming language because it was the only language available at the time which could be used across platforms without modification of the application code. It also provided certain constructs that would simplify the development and, more importantly, the maintenance of the software. The downside was hidden bugs in the various virtual machines, my relative inexperience with Java, threading requirements for network operations, and the requirement to have the Microsoft JVM and the Apple MRJ limit the highest level of constructs to 1.1.8. As it turned out, these limitations actually turned to be an advantage as the evolution of javAPRSSrvr occurred.

The first version of javAPRSSrvr was designed to emulate the other servers' port types, support APRServe's validation algorithm, clean packet headers, and provide CRC-32 dupe checking. It was designed with no fixed configuration parameters so the sysop could emulate any server they wanted. It was designed as a console application to ensure cross-platform operation. It also supported a basic text status page and full packet logging capability.

The original design was developed with significant input from Bill Diaz, KC9XG, in the first month and more complete discussions with Steve Dimse, K4HG, throughout the first year of development. The first version had some bugs. But, it was stable enough for Jon Anhold, N8USK, to take a chance and implement it on third.aprs.net. Soon after, Dick Stanich, KB7ZVA, began testing with his second tier server, ahubwest.net. One thing that became apparent early on: the Java application was stable, even if it was not bug-free. But then, none of the other servers were bug-free either. It also became apparent that the basic design was sound but there were a number of rough edges that needed to be addressed.

The original version supported text status ports (normally set to port 14501), no-echo ports, echo ports, history ports, and hub ports. The no-echo port type emulates the aprsd port 10152 where packets from the client would not be sent back to the client. The echo port does not filter out the client's packets from the data stream going to the client, similar to the APRServe port 23. The hub port is an unidirectional port designed for server interconnects. It is unidirectional to reduce problems seen with looped packets and unorganized massive interconnects. The history port is similar to APRServe and aprsd port 10151. However, the javAPRSSrvr history port is unidirectional and only sends the last posit and last object seen for each station and object within the last X minutes. It is unidirectional to help prevent stale information from being reintroduced into the APRS-IS.

javAPRSSrvr disables the Nagel algorithm on all network data ports. While this caused many servers with limited bandwidths problems (each packet used one IP packet), it soon became apparent that many delays in the network were starting to disappear. Because of this, dupe checking started working again. This also identified one key component of network delays: bandwidth shaping. We were able to isolate numerous instances where bandwidth shaping at the ISP could cause significant (seconds) delays.

By year end, javAPRSSrvr was very stable and requests had started to flow for different types of ports. Greg Noneman, WB6ZSU, sysop for second.aprs.net had migrated to javAPRSSrvr. The sysops for ahubca.net and ahubeast.net had migrated to javAPRSSrvr. Steve Dimse had migrated to javAPRSSrvr for use on findu.com. And there were others. The big allure, if you could call it that, of javAPRSSrvr is its stability, flexibility, and the ability to run it on almost any operating system.

At this point, a short review of the architecture of javAPRSSrvr is in order. First, javAPRSSrvr is not open source, but it is free to any ham who wishes to use it for amateur related activities. I decided at the start to not make it open source primarily for support issues. As it has turned out, I think this has been a very beneficial decision since I can keep a close eye on the architecture and can rapidly respond to bug reports, etc.

javAPRSSrvr has a thread for each port it listens on, a thread for each outbound connect procedure, and a thread for packet processing. This last thread runs between all of the read threads and the write threads for each connection. As with any multithreaded application, synchronization has always been a concern. Also, certain JVM's have restrictions on the number of threads that any one application can have.

Why not have each receive thread put the packet on each transmit thread's queue? This turned out to be a decision forced upon us by certain OS/JVM combinations. Specifically, we saw significant stack and synchronization problems with the Linux JVM's. Those problems went away with the introduction of a "go between" thread to do the packet decoding, dupe checking, and placing on the transmit queues. As it turned out, having this go-between thread has some other benefits such as enabling server adjunct and IGate adjunct interfaces.

During the later part of 2002, Dale Heatherington, WA4DWY, and I got together via email and created the q algorithm. Dale was becoming active in aprsd development again and we were looking for a generic way to track loops as well as to identify different network components, such as servers and

IGates. What we came up with added about 5% overhead (if you take into account IP overhead) on each packet that gave us rudimentary loop detection (this is how I discovered the flaw in how hubAndMsgPorts works) as well as network device identification. It has proven its worth over and over again in simplifying troubleshooting on APRS-IS.

At the end of 2002, I was contacted by Roger Bille, SM5NRK, concerning the possibility of adding server-side filtering. Bandwidth was at a premium, clients were running slow or crashing because of the quantity of data being sent, and sysops were looking for ways to cut down their visibility to the backbone providers. I was reluctant, but since Roger was volunteering to write the code, I worked with him to create a generic interface into javAPRSSrvr.

Roger came up with an additive filter (instead of taking things out of the data stream, it would say what to pass) that works very well. The generic interface (ServerAdjunctInterface) started as a simple 2 prong approach: one where all non-dupe packets get pre-processed and the other prong where each individual port processed the packet. While this worked, there was unnecessary overhead (most packets getting parsed twice). So, we reworked ServerAdjunctInterface so the adjunct would receive a packet with the header already parsed by javAPRSSrvr and then the adjunct could return an Object for use later on with the individual send threads. This greatly reduced overhead, both in CPU and in memory.

Since the filter is additive, filtering is only applied to restricted ports. At the time, that was the msgOnlyPorts and the clientOnlyPorts. Demand quickly added filteredHistoryPorts and readOnlyPorts (unidirectional filtered ports). We started to see problems with message acks not getting through. After some research, I added IGatePorts which was derived from clientOnlyPorts, but added a recently heard list to help ensure proper messaging.

I was also asked to add IPv6 and multicast capability. Fortunately, IPv6 only required a minor parameter format change as Sun implemented it within the standard socket architecture. I added multicast send-only capability for people to experiment with. So far, I am only aware of Gerry Creager, N5JXS, making use of either of those capabilities.

The year of 2003 was mostly involved with bug fixes and adding minor features such as keep-alive comment lines, login comment lines, etc. That in addition to adding the filteredHistoryPorts, readOnlyPorts, IGatePorts, and multicastPorts. There was a lot of work done on improving javAPRSSrvr and working around various OS/JVM problems.

With the year 2004, a request came from the FireNet sysops to see about implementing something so they would not need to run two servers. They wanted the filtering capability of javAPRSFilter, but they needed a way to take packets in and send them back out to local ports only. So I developed the localOnlyPorts. With that port type, there became a need for a modified hubPorts which would also pass local data. This was added as localHubPorts.

There continue to be many bug fixes and minor feature enhancements (sometimes the latter cause the former to be needed). We determined earlier this year (2004) that the hubAndMsgPorts were a source for loops and would possibly explain why aprsd was showing so many loops early on since the hubAndMsgPorts type was implemented to mirror aprsd's port 1313. Because of this discovery, hubAndMsgPorts have been removed from javAPRSSrvr.

As I write this paper, I have introduced three new adjuncts to javAPRSSrvr. The most recent is javAPRSDB. This is a server adjunct which allows support for other server adjuncts. It populates an SQL database with posits, tracking information, and weather information. It is used on www.jfindu.net to maintain that database. It is also used in conjunction with javAPRSFilter so a single server application provides both the database filling and the live APRS-IS feed.

The other two adjuncts are javAPRSIGate and javAPRSDigi. javAPRSDigi is an adjunct to javAPRSIGate and I will discuss that later in this paper. javAPRSIGate required the addition of a new interface, IGateAdjunctInterface, to javAPRSSrvr. This was added (like ServerAdjunctInterface) in two basic passes, release 1 and release 2. Release 2 was designed to simplify the interface and make it more reliable, which it has done. I avoided making an IGate until now because I wanted to maintain OS independence in javAPRSSrvr. I achieved that, to a degree, with the design of javAPRSIGate as TNCInterface is open source, just like ServerAdjunctInterface and IGateAdjunctInterface.

To get OS independence, I created a TNCInterface which allowed me to create interfaces for three primary TNC interfaces: AGWPE, KipSS, and Linux ax25 support. Roger Bille added direct KISS support for Linux and Windows. There may be other OS's in the future.

javAPRSIGate is a full-featured IGate allowing the sysop excellent flexibility in configuration. It supports smart maximum hop determination (including UIFlood and UITrace algorithms). It supports unidirectional (to APRS-IS) as well as bidirectional gating.

javAPRSDigi goes between the javAPRSIGate TNCInterface and the actual TNC interface. It provides full alias substitution, full UIFlood, and full UITrace operation. It also enforces RELAY in the first position of a path. It supports a maximum hop "turn-off switch" which allows the sysop to dictate what packets get repeated and which ones don't.

This brings us to where javAPRSSrvr is today. Has it met its goals? I propose that it has and continues to do so. It is running on multiple OS's and multiple JVM's, in over 75 locations throughout the world. The number of sysops continues to grow because the program "works as advertised". It does not modify packet data; it computes dupe checks not only on what may be normal, but also on what may be mangled packets (have you noticed the decline in Mic-E conversion packets); it supports every port type that has been presented; it continues to have new functionality added as well as any bugs fixed as they are reported; most importantly, APRS-IS has stabilized with its introduction to the network.

To help prevent the loss of support for javAPRSSrvr and its adjuncts caused by a sudden catastrophe to either Roger Bille or me, we keep the other author's code in escrow. Since Roger is in Sweden and I am in the United States, this makes for very good insurance which we both hope will never be needed.

73,

Pete Loveall AE5PL