javAPRSSrvr 4.3
Overview

Table of Contents

Section 1 – Introduction

javAPRSSrvr was written to provide the amateur radio APRS community a simple, but effective core server for interconnecting Internet/RF APRS Gateways (IGates) and other APRS clients via the Internet. It provides a multi-platform application tuned to present cleanly formatted packets to the APRS-IS network. It does this with effective duplicate checking to reduce bandwidth requirements and reduce loop occurrences present in a loose ad hoc network such as APRS-IS.

The current APRS-IS network consists of multiple core servers with multiple lower tier servers which may also act as IGates. javAPRSSrvr is designed to act solely as an APRS-IS server. However, javAPRSSrvr may be extended by both server adjunct and client software to provide services such as server-side filtering, email messaging, etc.

History

The javAPRSSrvr project was begun in early 2002 at a time when APRS-IS was in its infancy. At that time, there were 3 software packages available to provide core server functionality: APRServ (Mac), aprsD (Linux/FreeBSD), and AHub (Windows). APRServ was written by Steve Dimse K4HG and was the first APRS-IS server. aprsD was initially written to provide the first IGate functionality to APRS-IS and was expanded to full server capability. AHub was written to provide server capability to Windows machines. At the time of introduction of javAPRSSrvr, servers were seldom able to run for more than a 24 hour period, packet loops were rampant, and clients were failing just because of the large data stream being presented.

By the end of 2002, 2 of the 3 core servers were running javAPRSSrvr (first was running aprsD) and the aprs2.net tier 2 server group was started around the javAPRSSrvr software. javAPRSSrvr was stable and providing reliable connectivity for all of APRS-IS. During this time, Dale Heatherington WA4DSYand I created the “q algorithm” to help reduce loops. It was implemented on both javAPRSSrvr and aprsD. The reliability of APRS-IS was improved exponentially. A side-effect of the “q algorithm” is the identification of the “port of entry” of a packet which helps diagnostics immeasurably.

By the end of 2002, it became obvious that the weak link in APRS-IS was rapidly becoming the GUI clients. There was just too much data for them to process, even on a fast machine. Roger Bille SM5NRK offered to investigate the idea of a “dynamic filter” which became the basis for his javAPRSSrvr server adjunct called javAPRSFilter. If javAPRSSrvr and the “q algorithm” made APRS-IS stable, Roger’s javAPRSFilter made it usable by reducing the feed to GUI clients to a manageable flow.

Current

javAPRSSrvr 4.0 is the next generation javAPRSSrvr completely rewritten around Java 1.5 and above constructs and uses those constructs to implement a more flexible server while decreasing overall operational overhead. Focus was on making modular, reusable code while maintaining the high standard of reliability and usability of javAPRSSrvr 3.15. To accomplish this, however, meant that configuration compatibility with prior versions could not be maintained if the software was to be configurable at a modular level. The users guides go into the various configuration properties in detail but suffice it to say that while flexibility has increased, complexity has decreased as each module is configured independently.

It is the author's hope that this software will help to improve the reliability and the utilization of the APRS-IS. Please contact post your feedback on the javAPRSSrvr Yahoo group. It is only through this kind of feedback that improvements can be made.

Section 2 – Operational Overview

javAPRSSrvr is designed to run on any OS with any Java Virtual Machine 1.5.0 and later. Due to this newer JVM requirement, OS-specific versions such as Microsoft Java and J# .NET are no longer supported.

javAPRSSrvr is comprised of a number of classes are basically programming components. The main class is located in javAPRSSrvr.jar. This class is called at startup when using the –jar switch on the command line, loads the parameter files and respective classes, and begins execution of the different threads. javAPRSSrvr can be run without any knowledge of Java programming, just like any other command-line application you may use. The primary difference is that you run Java (the Java Virtual Machine or JVM) and tell Java to run javAPRSSrvr. Your operating system thinks it is running Java while Java is running javAPRSSrvr internally.

What's New in 4.0

javAPRSSrvr loads all necessary class files and libraries internally based on the various module properties files (ClassPath= and LibraryPath=). These are no longer defined on the command line, in the OS environment, or by placing files in a certain place. The core javAPRSSrvr jar files are javAPRSSrvr.jar, APRS.jar, APRSSrvr.jar, SerialIntf.jar, and Util.jar. These jar files must be present in the javAPRSSrvr startup directory for javAPRSSrvr to run.

Clients/Connections

javAPRSSrvr considers all connections, upstream, downstream and internal, as client connections. In the case of upstream ("dialing out" to a remote server) connections, all non-duplicate packets not marked as local-only will be passed to the remote server. Any packets received via an upstream connection are considered "local-only" to prevent looping of packets. Upstream connections may be bidirectional or receive-only. **If receive-only connections are specified, bidirectional connections are not allowed.** This prevents a single server putting unnecessary loads on remote servers and reduces the possibility of loops. Only one bidirectional upstream connection may exist in any server.

A second type of "upstream connection" is the server-to-server "connection". "Connection" is in quotes because a connection in the traditional sense is not created between peer servers. Instead, the peer servers send and receive UDP packets in the same fashion as normal upstream connections with the addition that packets injected via local-only ports are also passed (but not packets received from other peer servers or upstream connections). Each UDP packet contains one and only one APRS packet. The server-to-server connections are for core server use only and do not serve any useful function for lower tier servers. For server-to-server connections to be properly implemented, all servers in the core group must have a server-to-server connection defined for every other server in the core group.

There are multiple types of connections supported from remote clients. Older APRS server software had predefined port numbers for the various kinds of client ports; javAPRSSrvr eliminated this restriction to better conform to individual system requirements and preferences. javAPRSSrvr also modified or removed certain types of client ports to improve APRS-IS integrity (such as echo ports). There are also new types of client ports only offered by javAPRSSrvr to better serve the amateur radio community (client specified filter ports, for instance).

Remote Clients

Remote clients can connect to javAPRSSrvr via TCP or UDP. A TCP connection can also support a UDP feed from javAPRSSrvr to reduce loading on the server (no TCP overhead). UDP ports are receive-only and can either receive from predefined IP addresses or from any remote client if the UDP packet contains a login line in addition to the APRS packet.

TCP ports are three basic types. The first type is an HTTP receive-only port which accepts PUT and POST requests where the data is a login line followed by the APRS packet.

The second type of TCP port is the "full feed" port. This port sends all non-duplicate packets to the client after the client has logged in. This port does not support server commands other than the UDP command to route those server-to-client packets via UDP instead of TCP.

The third type of TCP port is the "restricted feed" port. At a minimum, this port will only send to the client message packets addressed to the client and associated posit (posit from the station sending the message) if the client uses a valid passcode at login. The "restricted feed" port can also maintain a "recently heard" list to support IGates which gate packets heard on RF the server. This "IGate port" type will also send to the client message packets addressed to "recently heard" stations and associated posits, and the server will send to the client any packets from the "recently heard" stations that were injected directly into APRS-IS.

All non-HTTP TCP “feed” ports may be “send-only” or bidirectional. If the TCP port provides historical packets, it will always be “send-only” to reduce the possibility of looping.

The “restricted feed” ports may also support server commands for adjuncts such as APRSFilter. APRSFilter is the 4.0 javAPRSFilter and provides server-side “filters” to add packets sent to the client which meet either predefined or user-defined criteria. Since this is an additive feature (adds packets to the stream), the full feed ports do not support this feature nor any other server adjunct that might be created. No filtering occurs on packets sent to the server from the client.

All operational clients such as email, whois, database, and IGate adjuncts appear as any other client in javAPRSSrvr. This facilitates multiple internal clients and port types to run concurrently in a single javAPRSSrvr instance. It also allows each client to specify what type of “feed” it wants to receive and if it wants the “connection” to be bidirectional or not. javAPRSSrvr prevents someone from logging into the server with the same callsign-SSID as an internal client.

javAPRSSrvr has built-in header filtering (clean-up) and certain types of packet filtering. Header filtering analyzes the header and puts all received headers into standard TNC-2 format. All TNC header information such as port number, timestamp, <>, [], and spaces are eliminated from the packet leaving only a clean header for retransmission to clients. The header filtering is used by javAPRSSrvr to accurately do duplicate packet checking based on the Source callsign-SSID, Destination callsign, and the packet payload. This is a basis of “station of origin and data” for duplicate checking. Special duplicate processing for packets containing trailing white space, embedded DEL characters or characters with the MSB set, and Mic-E translation is also included.

Callsigns must be no more than 9 ASCII alphanumeric characters and no less than 3 ASCII alphanumeric characters. A SSID, if present, must be only 1 or 2 ASCII alphanumeric characters separated from the callsign by a single hyphen and the total length of the callsign, hyphen, and SSID may not exceed 9 characters. Zero (0) is not a valid SSID as it is implied by no SSID being present.

javAPRSSrvr implements the q-construct which is used by javAPRSSrvr, aprsD, and some other servers and IGates to perform 2 major functions: reduce/eliminate loops and identify point-of-entry (POE) for packets. The q-construct is always a 3 character sequence in the packet path beginning with a lower-case “q”. The q-construct is NEVER to be seen on RF.

When a station connects to javAPRSSrvr and the connection is accepted, they receive the following comment line:

javAPRSSrvr 4.0.1b01

If the station's connection is rejected, it will receive one of the following comment lines depending on the reason for the rejection:

javAPRSSrvr 4.0.1b01 Port full.
javAPRSSrvr 4.0.1b01 Port unavailable.

The rejection comment line will be sent before disconnection occurs.

When that station logs in, they will receive the following line:

logresp callssid unverified, server userCall

callssid is the logged in call. If the passcode supplied by the client is valid and the port supports receiving packets from the client, unverified is replaced with verified. userCall is the "callssid" for the server.

If there is a server adjunct installed for that port and the login contains a command for the adjunct, the following will be sent by javAPRSSrvr:

logresp callssid unverified, server userCall, adjunct "command param" sa_response

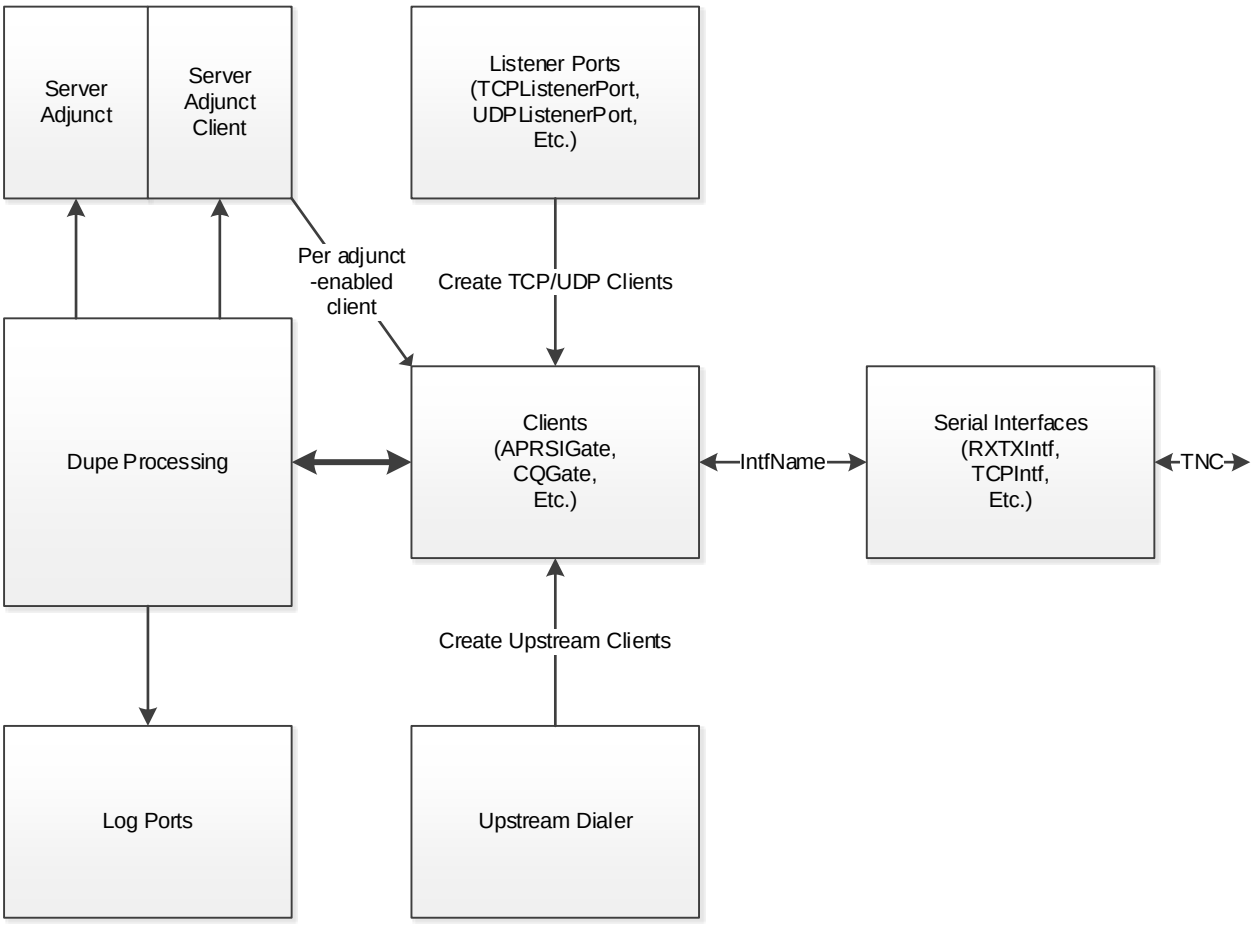
This is the same as before, with the adjunct info appended to the end of the line. "command param" is the command/parameter combination received by javAPRSSrvr. It is enclosed in double quotes to simplify readability. sa_response is the response issued by the adjunct.

Section 3 – Module Definitions and Interactions

javAPRSSrvr is made up of Java program files, called “classes” which are in various groups called “packages”. This method of isolating various aspects of a program is called Object Oriented Programming (OOP) and provides for extreme flexibility in configuration and implementation by minimalizing inter-module dependencies. One aspect of OOP is the concept of “inheritance” where you have a base definition like “football” and then have derived (sub) classes that inherit and refine the base definition. In the case of “football”, you would have “American”, “Canadian”, “Australian”, “Soccer”, etc. While some may be loosely related (“American” and “Canadian”, for instance), others may only be related by name, they all use a ball, and they all are 2 team sports with played on large fields with goals at each end of the field. In the case of javAPRSSrvr, you will see inheritance plays a big part in its architecture (all clients are derived from a single base class, all APRS packets are derived from a single base class, etc.).

Architecture

The basic architecture of javAPRSSrvr is shown below:



Dupe Processing Class

The key base class is the Dupe Processing class which all APRS packets pass through whether received from a remote client, generated locally, or received from another server. It is here that things like duplicate checking, packet parsing for APRS content, sending to log ports, etc. are done. It is also here that server adjuncts like APRSFilter see a packet for global processing. The Dupe Processing module also offers each non-duplicate packet to each client that wants to see the packet.

The previous line is significant. In 3.x, each client transmit thread processed every non-duplicate packet. In 4.0, each client transmit thread only processes non-duplicate packets the client has requested, either by filter or standard port processing. The reduction in overhead is significant as only the threads that need to actually send a specific packet will be woken up; the other threads remain dormant until a packet is available for them. This should dramatically increase the maximum allowable number of client connections per server on lower tier servers where restricted feeds are the norm.

Client Classes

The next base class that is significant is the Client class. All packets either originate or pass through the Client class. Currently implemented Clients include TCP, HTTP, UDP, Upstream, server-to-server, IGate, CQ Gate, Email Gate, DB Gate, Object Generator, Whois Gate, and WXNow Generator. Because these are all derivatives of the Client class, they all can be configured independently and make use of the various Client features such as server-side filtering (some can, others can't because server-side filtering, for instance, doesn't work with full feeds or receive-only feeds).

Because remote clients connect at random, a method for creating their local client class needed to be implemented. That job goes to the Port Listener classes. Each port listener is individually configured so that it generates the properly configured client for each connection (or packet in the case of UDP).

Serial Interfaces

The internal IGate is a special kind of client because it needs to talk to RF (either via a TNC, sound card, or other medium such as D-STAR). To accomplish this, another base class exists in javAPRSSrvr called the "Serial Interface" class. This class implements a buffering/sharing mechanism to allow multiple IGates to share the same TNC, for instance. All serial interface classes derive from this class. For an IGate to know which serial interface to use (there can be multiple just like there can be multiple IGate clients), a property exists in both the IGate properties file and the Serial Interface properties file called "IntfName". IntfName can be anything the sysop wants but it must be exactly the same in both properties files. The sequence that starts up javAPRSSrvr starts the serial interfaces before starting the clients. This way, the serial interfaces say "I exist and am called X" so the IGates say in their startup "I need to talk to the serial interface via X". The Serial Interface logic lookups "X" and gives the requesting IGate the proper Serial Interface object to talk to. This may sound convoluted but basically it lets any number of internal clients (currently APRSIGate and NSRDigi) to talk to one or more serial interfaces.

Another "special" internal client is NSRDigi. This client is not an APRS client so it neither sends nor receives data with the Dupe Processing module. It solely exists within javAPRSSrvr as a client to use a serial interface (usually shared with an internal IGate client).

The rest of the clients perform specific functions such as sending email, populating database tables, etc. They are configured like any other client port and their configuration is described in their respective User Guides.

The Upstream Dialer has already been described but it is sort of like a reverse Port Listener. Instead of waiting for a connection, it initiates a connection to a remote server. This feature was abused in the past on servers such as aprsD causing significant loops, wasted bandwidth, and wasted resources on the upstream servers. javAPRSSrvr 4.0 carries on the tradition introduced by javAPRSSrvr 1.0 where only one upstream connection may be open at any one time if that connection is bidirectional. Multiple concurrent upstream connections may be implemented for something like a data collector but those connections are not bidirectional and should be avoided as much as possible.

Server-to-server "ports" are actually individual clients which behave similar to a bidirectional upstream connection but use UDP between servers and multiple "connections" are permitted. They cannot be randomly created; each server-server definition must exist in both servers for the "connection" to be made.

There are other special use clients but those clients are best described in their respective Users Guides.

Log Ports

The log ports is the final base class to be discussed. A log port is a port where stamped packets are sent to attached clients. These are log ports and are send-only. They cannot be used for standard APRS client use and should only be connected to with a text client such as telnet or a text logging program.

Server Adjuncts/Server-side Filtering

But what about the Server Adjunct? Isn't that a base class? Yes, it is but it is not a base part of javAPRSSrvr. In other words, javAPRSSrvr works "as advertised" without a server adjunct installed. However, APRSFilter derives from the Server Adjunct class and provides server-side "filtering" for properly configured clients if it is included in the installation. APRSFilter uses the same command structure as javAPRSFilter did but is not based on the javAPRSFilter code. As javAPRSSrvr 4.0 is a complete rewrite from prior versions, APRSFilter is completely new code from javAPRSFilter. This was done to allow use of newer constructs which allow clients to share range and area filters for instance. It also allowed for those shared filters to "go away" after the last user was done with a shared filter.

Status Ports

Where are status ports? A status port is a Port Listener that doesn't create clients. A significant change from prior server status port implementations, the 4.0 status port generates XML data only, not HTML or text status pages. By implementing the status reporting in this way, the code only generates data and doesn't try to format any of it. HTML pages can still be generated but this is entirely under the control of the sysop. That transformation (XML to HTML) is done by the sysop defining an XSLT file which tells Java how to convert the data to HTML format. This is extremely flexible and complex sample XSLT files are found in the group file area.

Section 4 – Summary

javAPRSSrvr 4.0 is a leap ahead in performance, capability, and flexibility from prior versions. By fully using OOP techniques along with modern Java constructs, the sysop is able to fully configure a server by concentrating on individual pieces and not being hindered with the perennial “how does a change here affect this other module?” question.

When implementing, certain constraints remain which must always be considered:

All callsign-SSIDs must be unique across the entire network including the server’s callsign-SSID.

Log ports are for diagnostics only and should not be made available outside of the LAN.

4.0 configuration is done with individual properties files for each port or internal client.

javAPRSSrvr.jar no longer contains all classes. Be sure to include any jar files in your installation you will be using including the base jars (javAPRSSrvr.jar, APRS.jar, APRSSrvr.jar, SerialIntf.jar, and Util.jar) and APRSFilter.jar (if you use the Filter functionality).

APRSIGate, NSRDigi, and SerialIntf derivatives use the IntfName property to identify which serial interface to use. IntfName is completely sysop definable. All other clients do not communicate with serial interfaces.

KISS: **never** include default properties in your properties files. This only invites mistakes if you either mistype or misinterpret what the default value is.

Read the manuals! There is a lot of information there and it is there for a reason.

All diagnostics start with the javAPRSSrvr error log file(s).

When a question or problem does arise, present it to the group. That way, everyone can have the opportunity to learn what might be causing their unasked questions.