YAAC: The Development of "Yet Another APRS Client", an Open-Source Cross-Platform Application

Andrew Pavlin, KA2DDO

ABSTRACT

Of the many APRS clients currently available, all of them have some limitations or constraints. YAAC (Yet Another APRS Client) was designed and implemented to meet the author's needs and hopefully some unfulfilled desires of the Amateur Radio community, while also being a personal research project in the grand old Amateur Radio tradition of home-brewing.

Keywords: APRS client, open source, packet radio

1 Introduction

Although many APRS client programs and embedded systems have been created, none of them meets all possible needs; each is specialized towards what their implementers intended them to do (more or less). This paper discusses the process this author went through to create a customized yet flexible, extendable, and portable APRS client.

1.1 Why Write YAAC?

There are numerous APRS clients available, the best-known of which include (but are not limited to):

- the original APRS-DOS (by Bob Bruninga WB4APR)
- WinAPRS and MacAPRS (by Keith Sproul WU2Z and Mark Sproul KB2ICI)
- Xastir (by Frank Giannandrea and the Xastir Group)
- UI-View (by Roger Barker G4IDE)
- APRSIS32 and APRSISCE (by Lynn Deffenbaugh KJ4ERJ)
- APRSdroid (by Georg Lukas DO1GL)
- the built-in TNC's in some Kenwood and Yaesu amateur radio transceivers, such as the TM-D710

However, none of these completely met my needs for an APRS client that would simultaneously:

- operate completely disconnected from the commercial wireline Internet (as in simulated or real emergencies such as ARES or RACES would support);
- have up-to-date, readily available, highly detailed and customizable maps that were zoomable and pannable;
- operate consistently on multiple platforms, including (but not limited to) both Microsoft Windows® and Linux™;
- be extendable with arbitrary new features as the need for said features became apparent when I wanted to add the extensions (as opposed to waiting for someone else to get around to making the extensions for me);

- have support available for the software when I needed it;
- support modern platforms and be easily ported to other platforms as the need arose;
- provide the concise yet complete real-time tactical information that Bob Bruninga keeps reminding us about. [APRSTAC]

Additionally, as a amateur radio operator, my inclination is to tinker with the technology to see if I can do it better, or at least better understand how the technology (in this case APRS) works.

1.2 Goals for YAAC

Given the above stated needs, my goals for a suitable APRS client are:

1. Should run on my chosen platforms (Linux™ and Microsoft Windows®) without being restricted to only those operating systems.
2. Should have source code I can access and change as I wish, without violating any licenses or laws.
3. Should be up-to-date with current specifications and standards for APRS and the supported platforms, but able to grow when those standards evolve.
4. Should be backwards-compatible and interoperable with existing deployed APRS clients (such as those itemized above) and the APRS-IS backbone.
5. Should be fully functional without any commercial Internet connectivity whatsoever, but capable of taking advantage of the Internet when it is accessible.
6. Should work with standard and common APRS and AX.25 hardware, such as TNC2-compatible TNCs, Kenwood radios with embedded TNCs, NMEA-0183-compatible GPS receivers, and consumer-priced weather stations (i.e., equipment I already own).
7. Should be user-friendly to non-programmers, and generally quick, easy, and obvious to use.
8. Should perform reasonably well (response-time wise) on non-high-end computers (such as my entry-level laptop).

2 The Process of Creating YAAC

2.1 Preparation and Software Engineering

Several up-front decisions were made that would guide the rest of the project.

First, for software portability, the Java language was chosen for its "write once, run everywhere" capability (or, as Java developers joke, "write once, test everywhere"). And, both because it is a good software engineering practice, and because the Java language is strongly oriented towards it, the Object-Oriented Programming design philosophy was adopted. Additional philosophies adopted were the Model-View-Controller division of the software "objects", and the Internet robustness principle of "be conservative in what you send, liberal in what you accept" regarding standards compliance and interoperability. [RFC1122] Because it was such an excellent extendibility feature of both UIView and many other non-APRS applications, YAAC would have a "plug-in" means of adding extensions without having to issue a new release of the core software, using the Provider design pattern seen in the Java runtime library.

Because I wanted other users to be able to have the same benefits I would have with YAAC, open-source was the only way to release the software (so that other users could also extend YAAC as they wanted without having to wait for someone else to do it for them), and the Free Software Foundation's General Public License (or something similar to it) was the only way to guarantee that right couldn't be taken away. This decision restricted what other libraries and data sources could be used in implementing YAAC, due to incompatibility with the GPL [GPL], and the legal inability to give those components away as open-source. However, to avoid having to support any bad initial design decisions, YAAC would remain closed-source until it completed alpha-testing and was formally released; that way, significant architectural changes could be made to correct early design mistakes and limiting assumptions rather than having to be backwards-compatible with the mistake forever (a frequent issue seen in software).

2.2 The Design of YAAC

To meet the recommended data presentation needs, YAAC needed to be able to display incoming APRS traffic in several different ways (as a minimum):

- the expected geographical map of station/object locations, with complete control over how and which features are displayed on the map.
- tabular reporting of raw incoming messages, with user-selectable sorting.
- tabular listing of stations and APRS Objects, with user-selectable sorting.
- tabular reporting of incoming APRS text Messages, with user-selectable sorting.
- filtered tabular reporting and logging of incoming APRS text Messages, to support Bob Bruninga's field data entry proposal. [FIELD]
- bulletin display, per the APRS 1.0.1 specification in chapter 14. [APRS101]
- highlighting selected stations, APRS Objects, and map landmarks (to help the user locate any of these on a cluttered map).
- 1-to-1 station-to-station chat sessions, modeled on Internet instant messaging clients.

All of these view modes should use windowing-system-native display styles.

YAAC was designed from the beginning to use the Java I18N (internationalization) support to allow plugging in alternate-language display strings for the GUI and the online help.

Given the platform portability and disconnected-from-the-Internet goals, Google Maps® and its competitors were immediately rejected (they needed Internet connectivity), as was the Precision Mapping® program (used by UIView), DeLorme Street Atlas® (used by some Windows-based APRS clients), and their competitors (due to lack of platform portability). The OpenStreetMap Foundation's open-source map data was the most obvious choice for world-wide open-source map data [OSM], but it required a renderer to use it. Rather than use the OSM tile server (requires Internet connectivity) or adapt or copy one of the existing open-source renderers for OpenStreetMap data, I decided to "roll my own" renderer in a moment of hubris and excessive optimism (more about this later).

To support the functions that YAAC would have to perform, the following third-party libraries were selected:

- the open-source but non-GPL'd OpenMap library from BBN Technologies, [OPENMAP]

- the Lesser General Public License controlled RXTX library for serial-port I/O in Java, [RXTX]
- the old but still GPL'd JavaHelp 2.0 library from Sun Microsystems (now Oracle), [HELP]
- the Apache Commons' commons-compress library. [COMPRESS]

Because OpenMap's open-source license was not fully compatible with the GPL, the planned license for YAAC was changed to the FSF's Lesser General Public License. [LGPL] I felt that the additional license restrictions on OpenMap were acceptable for my project, versus trying to find a fully GPL'd/LGPL'd alternative, or re-creating the complicated mapping algorithms without making errors or infringing on someone else's copyrights or patents.

Knowing that YAAC would grow very quickly into a rather large program, Java's packaging design pattern was used to segment the different functional parts of YAAC from the very beginning. To uniquely identify the software, I chose a package hierarchy based on my callsign and the Internet domain name (ka2ddo.org) I would use for distributing YAAC alpha-test builds. The packages in YAAC were identified as follows:

- org.ka2ddo.yaac.aprs - classes related to specific APRS packet types.
- org.ka2ddo.yaac.ax25 - classes related to AX.25 frames (regardless of whether they contained APRS packets or not).
- org.ka2ddo.yaac.bootstrap - classes related to starting up YAAC.
- org.ka2ddo.yaac.core - classes related to the YAAC core functionality.
- org.ka2ddo.yaac.core.provider - classes for the default or "root" plugin, defining the core functionality of YAAC.
- org.ka2ddo.yaac.core.queries - handler classes for the APRS standard queries (as defined in the APRS 1.0.1 specification in chapter 15) [APRS101], and a few custom extension queries defined by the APRSIS32 client or unique to YAAC itself.
- org.ka2ddo.yaac.docs - HTML files and images for online help for the application.
- org.ka2ddo.yaac.filter - classes related to filtering AX.25 and APRS frames, so a user could look at a dynamically-defined subset of the data being processed.
- org.ka2ddo.yaac.gps - classes related to processing NMEA-0183 sentences.
- org.ka2ddo.yaac.gui - classes related to the Graphical User Interface of YAAC.
- org.ka2ddo.yaac.gui.configwizard - GUI classes for configuring YAAC.
- org.ka2ddo.yaac.gui.drawlayer - GUI classes for drawing custom features onto the map.
- org.ka2ddo.yaac.gui.genericwizard - infrastructure classes for implementing "wizard" panels and controlling the sequencing of wizard panels.
- org.ka2ddo.yaac.gui.table - classes for custom-controlling Java Swing JTables.
- org.ka2ddo.yaac.io - classes related to YAAC's connections to outside devices, such as radios and Terminal-Node Controllers, GPS receivers, and the Internet.
- org.ka2ddo.yaac.os - classes related to operating-system-specific features that YAAC would have to adapt to, that Java didn't already handle in a portable manner.

- org.ka2ddo.yaac.osm - classes related to processing and rendering OpenStreetMap data.
- org.ka2ddo.yaac.pluginapi - classes related to defining the standard plug-in Application Programming Interface (API) for YAAC.
- org.ka2ddo.yaac.util - utility classes that didn't fit in anywhere else.
- org.ka2ddo.yaac.weather - classes related to processing weather station data.

The main class for YAAC, the default-language localization resource bundle, and the JavaHelp indexing files were put in the top-level org.ka2ddo.yaac package.

Because YAAC would need to be uniquely configured for each installation (at a minimum, a unique callsign and I/O port to the Internet or a TNC), I chose the Java Preferences feature as a platform-portable place to store configuration data.

2.3 The Implementation of YAAC

YAAC's implementation used conventional software development tools on my home computer, very similar to what I use in my professional life as a software engineer. A commercial Java integrated development environment (IDE) from JetBrains called Intelli-J IDEA was used for Java development; although Eclipse was available for free, my personal preference was to use this easier-to-use commercial development tool despite its cost in real money (noting that my usage would not force any future developers to use this particular IDE). The open-source free software product Subversion was used as the version control system, and Apache Ant was used as the build tool. The appropriate operating-system variants of the Sun Java® Development Kit V1.6.0_31 (which includes the Java Runtime Environment of the same version) were used to compile and execute YAAC. The following sections discuss some of the major implementation details of YAAC over the past year's development.

2.3.1 Initial Development of YAAC

YAAC was implemented in a step-wise manner, such that older features could be used to help debug YAAC's own newer features. Initial features were developed in the following order:

1. The basic GUI framework, using the Java Swing toolkit to create the initial JFrame with a menu bar and OpenMap map panel.
2. The basic plugin API.
3. The startup code with means of reading saved port configuration data, opening previously configured ports, and loading the default plugin to define the initial set of port drivers and GUI menu choices. This startup code also provided common helper functions to support text message localization.
4. The default plugin with the initial set of menu choices; the first menu command implemented was the File->Exit command.
5. The AX.25 frame classes and AX.25 protocol stack (assuming a KISS TNC).
6. The Serial_TNC port driver (using the RXTX library) and the generic infrastructure for interface ports.
7. The "sniffer" window with its tabular display of raw incoming AX.25 messages (adding the View->Sniffer menu choice).

8. The configuration GUI for creating and modifying the settings of serial TNC ports (adding the File->Configure menu choice).

At this point, I had an initial working program which I could start unit-testing, and immediately discovered some "undocumented features" (as commercial software vendors are wont to call unexpected system behaviors).

First, the RXTX library had an interesting interaction with the Java virtual machine; it required that a Java property (java.library.path) identifying the directory containing native JNI library files be defined ***prior*** to starting the virtual machine (through a command-line option on the java command), because RXTX could only find its platform-specific native I/O libraries in directories listed in that property. This made it difficult for YAAC to run-time determine its host operating system and define the property to point at the correct set of native library files of the several sets provided in the RXTX binary distribution, and made it impossible to start YAAC just by clicking on its jar file in an operating system file browser such as Windows Explorer. I couldn't depend on the operating system itself to provide RXTX; only some Linux distros even included RXTX as an optional library component which was not guaranteed to be installed, and the most likely platforms (Microsoft Windows and Mac OS X) explicitly did not include RXTX (since it was not Microsoft or Apple proprietary software, respectively).

I considered modifying RXTX (since it is an open-source LGPL library) to accept the library directory name as a method parameter. However, I did not own copies of every platform RXTX already supported in its binary distribution (so I could recompile all the native libraries to account for any changes I made), and I did not know how long it would take my changes to become part of the official RXTX distro and how long those changes would then take to trickle into the Linux distros. I also looked for alternatives to RXTX (including some branched variations of the RXTX library that included automatic native library variant selection), but none of them were routinely provided by major Linux distros.

So, I instead wrote a "bootstrap" class to perform the sole function of finding out which operating system YAAC was running on, locate the correct platform's version of the RXTX native libraries, compute the correct value for the java.library.path property, and then launch the real YAAC program as a sub-process using the java.lang.Runtime class's exec() method, specifying the java.library.path property on the subprocess's invocation command line. This was inefficient, ugly, and wasteful of system resources (running two Java virtual machine instances), but it worked. The bootstrap class also turned out to be useful later when alpha-testers tried running YAAC on an older version of the Java runtime environment, and YAAC failed with a not-very-visible and certainly user-hostile error message:

> Exception in thread "main" java.lang.UnsupportedClassVersionError: Bad version number in .class file

Unfortunately, YAAC was using Java runtime features introduced in Java Standard Edition Release 6, so I couldn't successfully compile YAAC with a previous version of the Java compiler and have it work, without going to significant effort to re-implement these new features myself in the older release. However, the YAACBootstrap class was not using any of those new features, so this class alone could be compiled using an older compiler version (more strictly, telling the Java 6 javac compiler to produce classfiles for an older runtime with the "-target 1.5" command-line option), and YAACBootstrap's operating system test could also test the Java runtime's version for the minimum acceptable Java release, and display a more user-friendly error message explaining why YAAC wouldn't work (complete with directions for obtaining an upgraded version of the Java runtime) if the Java release was too old.

2.3.2 Implementation of the YAAC Plugin API

The plugin API was implemented as an abstract class, org.ka2ddo.yaac.pluginapi.Provider, defining a set of methods that would tell the plugin loading code what features each plugin was defining. Each plugin would be provided as a standard Java jar file, with the jar manifest file defining a global attribute, YAAC-Provider. This attribute's value would be the fully-package-qualified class name of a class in the jar file that extended the Provider class. YAAC's main class, org.ka2ddo.yaac.YAAC, would search the YAAC installation's plugins sub-directory for jar files (in modification date order), test if each jar file had a manifest with the YAAC-Provider attribute, and test if the named class in the jar file was a subclass of the abstract Provider class with a no-arguments constructor. If so, an instance of the class would be created using the Java reflection facility, and the methods defined by the abstract superclass would be called to determine what features (if any) the plugin would add to YAAC. Before the plugins directory search, the default provider implementation, org.ka2ddo.yaac.core.provider.CoreProvider, would be instantiated and called first to set up the core functionality. Plugins are able to replace core functionality by redefining the functions in the plugin's Provider; the specific ordering of plugin jars by modification date is to ensure older plugins don't override newer plugins.

The Provider's defined public methods that can be overridden by subclasses are:

public boolean runInitializersBefore(int providerApiVersion)	Execute this function before calling any of the other functions of this Provider. This allows any Provider-specific initialization to run before menus and drivers are loaded, and also permits the Provider to block usage of the plugin (for example, if the plugin provides services only available on Microsoft Windows®, but YAAC is being executed on Mac OS X®).
public ImageIcon getImageIcon()	Return an icon image associated with this Provider (displayed in the Help->About dialog as part of this plugin's identification).
public Provider.PortEntry[] getPortConnectorTypes()	Get PortConnector driver classes provided by this Provider.
public Map<String,javax.swing.JComponent> getConfigurationPanels()	Get any panels needed by this Provider to provision or configure the services offered by the Provider. Note that it expects already-localized strings for the tab names; this is safe because this method is not called until the File->Configure->Expert Mode menu command (installed by the default Provider subclass, CoreProvider) is invoked, by which time all Providers and their associated localized ResourceBundles will be loaded.
public org.ka2ddo.yaac.filter.Filter[] getFilters()	Get any filters to add to the main CumulativeBooleanAndFilter in the main class of YAAC.

public javax.swing.Action[] getMenuItems()	Get Actions to define new menu items from this Provider. Actions must define the following properties: • PRE_LOCALIZE_MENU_TAG_NAME - the pre-ResourceBundle.getString() lookup of NAME that potential overriding menu entries can be identified (i.e., "menu.View.Weather.Wind") • PRE_LOCALIZED_MENU_HIERARCHY - localization tag names for the menu names in hierarchical order to contain this menu entry (i.e., { "menu.View", "menu.View.Weather" }) • SMALL_ICON (optional) - to label the menu entry Other properties defined by Action or this class may also be specified. In particular, "selected" triggers use of JCheckBoxMenuItem or JRadioButtonMenuItem, and "BUTTON_GROUP_NAME" differentiates between them (JRadioButtonMenuItems are always associated with ButtonGroups).
public String[] getAboutAttributions()	Specify attributions, credits/acknowledgements, and license references to be displayed in the About dialog box. This method is called when the Help->About menu choice is selected.
public HelpSet getHelpSet()	Provide an additional JavaHelp HelpSet to merge into the base HelpSet for complete online documentation.
public void runInitializersAfter()	Execute this function after calling all of the other functions of all the Providers. This allows any Provider-specific initialization to run after all menus and drivers are loaded.

All of these methods were implemented in the abstract superclass as do-nothing, return success, return zero-length array, or return null functions (as appropriate), so that subclasses would only need to override the methods needed by that plugin.

2.3.3 Implementation of I/O, AX.25 and APRS Protocol Stacks

Because of the way the RXTX library implements serial port reading, all the serial port drivers (for KISS TNCs, Kenwood radios in APRS mode, GPS receivers, and weather stations) are implemented much the same way. The general abstract superclass org.ka2ddo.yaac.io.PortConnector declares the basic methods that all port subclasses have to support. The class PortConfig defines a generic set of fields for declaring the parameters of an open port, such as the operating-system-specific device file name, the baud rate of the RS232 serial port, a flag indicating whether transmitting is allowed through the port, etc. PortConfig also provides common code for saving and restoring these configuration

parameters to/from the Java Preferences backing store.

The subclass's implementation of the abstract configure() method opens a port of the driver's type using the PortConfig object's parameters. For serial-port-based drivers. this means obtaining the gnu.io.CommPortIdentifier for the device port, opening the port and its associated InputStream and OutputStream, and registering the PortConnector subclass instance as a gnu.io.SerialPortEventListener (which the subclass implements). The serialEvent() method then indicates whenever serial-port data is received.

When a SerialEvent of type DATA_AVAILABLE is received, the serial port is read, one byte at a time, as long as the InputStream's available() method indicates there is more data. The bytes are collected into a pre-allocated byte array until an end-of-frame is reached (carriage return for ASCII-line-based interfaces like the Kenwood, GPS, and weather drivers; 0xC0 (KISS Frame END) for serial TNC). Note also that the KISS escape codes have to be processed in the unlikely but possible case of a frame containing one of the KISS-protocol-reserved framing or escape codes. [KISS] For the GPS and weather drivers, the resulting byte array is converted into an ASCII-valued Java String object and passed to the GPSDistributor or WeatherDistributor singleton classes (as appropriate). The serial TNC driver passes the byte array to the constructor for the org.ka2ddo.yaac.ax25.AX25Frame class, then sends the constructed frame to the AX25Stack for processing.

The AprsIsConnector opens a socket-based network connection rather than a local serial port device. and reads the serial input data just like the Kenwood driver does. Both of these drivers, once they reach the end of the line of a TNC2-format APRS frame, convert the text string into an AX25Frame object for a UI frame with PID of "no layer 3 protocol", and send it to the AX25Stack for processing.

The serial input ports don't have much buffering at the kernel and RXTX native code levels, causing a high risk of losing data due to serial port overruns if processing time for any individual byte (such as the end-of-frame code) increases to only 1 millisecond (assuming 9600 baud link speeds). To protect against this, the AX25Stack uses a java.util.concurrent.ArrayBlockingQueue to buffer the received AX25Frames, allowing the serialEvent() method to quickly continue reading bytes from the serial port (or, for AprsIsConnector, from the TCP/IP socket). A separate thread drains the queue of frames, one frame at a time, and analyzes each frame for appropriate processing, according to the AX.25 protocol specification. [AX25]

UI frames with PID = 0xF0 (no layer 3 protocol) are assumed to be APRS packets and are passed to the APRSStack class for decoding into the appropriate APRS packet type. All other (non-APRS) frame types are delivered to all registered AX25FrameListener implementations.

The APRSStack class decodes each frame based on the first byte of the frame body, according to the message type codes defined by the APRS protocol specification. If a frame body starts with the "}" character, indicating a third-party relayed packet, the third-party header is stripped off and converted to an ASCII String, and the remainder of the frame body is analyzed again by the APRSStack parse() method to determine the true packet type.

In any case, the decoded APRS packets are stored in per-type subclasses of the org.ka2ddo.yaac.aprs.Message class, typed as follows:

Leading character(s)	org.ka2ddo.yaac.aprs class
!!	UltimeterRawMessage
! / = @	PositionReport
<	StationCapabilities
>	StatusMessage
' ` \u001c (single non-printable character) \u001d (single non-printable character)	MicE
;)	ObjectReport (also holds item reports)
_	PositionlessWeatherReport
:	MessageMessage
?	Query
$GP	GpsRawMessage
$UL	UltimeterRawMessage
anything else	DefaultMessage

All registered AprsMessageListener implementing classes will be informed of each decoded APRS packet.

2.3.4 Implementation of the Tactical Message Database

The received APRS messages are catalogued in several different collections, supporting different views of the received data. All of these collections of data implement the AprsMessageListener, DuplicateCheckedAX25Listener, and/or AX25FrameListener interfaces, so they can receive new packets as they asynchronously arrive from (possibly multiple) interface ports. All of these collections have at least one view mode (most accessible from the GUI's View menu). For memory efficiency, all of the collections hold the same packets; duplicates are not made per-collection.

The SnifferTableModel is a subclass of javax.swing.table.AbstractTableModel which implements both AprsMessageListener and AX25FrameListener, and records all incoming frames until their receive time is older than the configured maximum retention time. The Sniffer view (encapsulated in nested instances of the SnifferFrame, SnifferPane, and javax.swing.JTable classes) displays the current sorted and filtered contents of the SnifferTableModel. JavaSE 6's RowSorter and RowFilter interfaces are used on all tabular displays to filter the packets based on current filter settings and efficiently sort the packets in the user-desired order. The SnifferTableModel also supports optional logging of incoming frames to disk files, and restoring of log files for historical playback.

The MessageModel is also an AbstractTableModel subclass which implements AprsMessageListener, but only collects MessageMessage packets; other types of APRS packets are ignored. A window containing a JTable displaying this model will be opened and pushed to the foreground if a message is received that is addressed to this station's callsign or to an enabled broadcast address such as NWS_WARN. The broadcast address list can be changed from the configuration GUI.

The MessageFilterPane is another implementer of AprsMessageListener which only collects specific MessageMessage packets matching the message filters (which are independent of the normal message filters), and sequentially logs the matching messages in a javax.swing.JTextArea and/or a disk file in a user-specified format, supporting the Field Data Entry use case. [FIELD]

The OutgoingMessageTableModel is an AbstractTableModel subclass which records APRS packets originated by the local YAAC instance (not received or relayed by the local instance) for transmission. The JTable display of this data allows you to selectively cancel or immediately retransmit any outgoing message.

The StationTracker catalogs packets in reverse chronological order on a per-sending-station basis, and extracts state information to maintain a current consolidated state of each station for convenient display. Similarly, the ObjectTracker catalogs Object and Item report packets on a per-Object/Item name basis. StationTracker and ObjectTracker share common code in the GenericTracker superclass and StationState data class. Updates to the trackers can be reported to classes implementing the TrackerListener interface and registering with the appropriate tracker singleton.

To more easily display current real-time packet traffic, a frame styled after the Kenwood TM-D710 control head's APRS-mode display was implemented, showing the last packet received via the AprsMessageListener interface.

The AlohaTracker computes the current aloha circle radius for each open RF port. Each time a station is added to the StationTracker or a station's position is updated, the AlohaTracker's computation thread is scheduled for re-execution. It sorts all the known stations in radial distance away from the local station, and then adds up estimated message traffic from each recently transmitting station (per interface port) in order until the theoretical channel saturation point is reached; the distance to the station exceeding the expected channel capacity is taken as the Aloha radius. Since different RF ports on one station should theoretically not interfere with each other, as they should be on different RF channels, each port gets its own Aloha radius. If there are not enough heard stations on a port to theoretically saturate the channel, the distance to the furthest station is taken as the Aloha radius but the value is flagged as incomplete, so the plotted Aloha circle can indicate the incompleteness.

Dialogs were also created to display the current cached states of the singleton WeatherDistributor and GPSDistributor objects. Using a javax.swing.Timer to periodically force a repaint, the GPSStatusDialog and WeatherStatusDialogs display the last data reported by the local GPS receiver and weather station. Similarly, the BandwidthMonitor reports periodically-updated data throughput statistics for each open interface port, sampling the current PortStats objects associated with each open PortConnector.

2.3.5 Implementation of the StationRenderer

The com.bbn.openmap.gui.BasicMapPanel contains a series of layers (specifically, subclasses of the com.bbn.openmap.Layer abstract class), registered in back-to-front order. When the map panel needs to be repainted, the layers' paint() methods are called in registration order to add their contributions to the map. The class org.ka2ddo.yaac.gui.StationRenderer is one of the last layers registered, so the symbols for stations and objects/items are drawn on top of the background layers.

StationRenderer has several properties (using the JavaBeans terminology) to enable/disable various optional parts of the drawing:

- age - write the time since last data update for this station/object below the station's identification.
- aliasInsteadOfCallsign - display a user-entered string for this station or object instead of its official identification.
- allTrackStripes - draw a translucent stripe of the path taken by each moving station or object. Other methods are provided to allow drawing a track stripe for a single station or object instead of all of them.
- alohaCircle - draw an orange circle around the local station's position showing the maximum range for channel saturation (assuming 1200baud transmission rate, as is common for the national APRS 2-meter calling frequency).
- ambiguityCircle - fill a translucent circle the size of the position ambiguity of each station behind the station's icon.
- deadObjects - draw Objects and Items on the map, even if they have been killed by an appropriate Object or Item report message (per APRS 1.0.1 specification, chapter 11). [APRS101]
- deadReckoning - draw a dashed stripe of the predicted motion of a moving station based on its last known position and course/speed. If not enabled, always draw the station/object at its last known position instead of dead-reckoning a position.
- DF - draw a direction-finder cone for stations reporting bearing and NRQ parameters (APRS 1.0.1 spec, chapter 8, page 34). [APRS101]
- itemStatus - fill a translucent circle behind the station or object's icon, indicating state information about the station/object, consistent with the original APRSdos application. Color codes include white (fixed), cyan (moving), bluer cyan (dead-reckoned), gray (not updated for at least 80 minutes), red (station reporting an emergency), yellow (object owned by the local station), or purple (object owned by some other station).
- NWSMultiLine - draw NWS multilines and polygons on the map if they are received in a National Weather Service bulletin message. [NWS]
- rangeCircles - draw a circle around each station (but not objects) showing the estimated range of the station, according to the PHGphgd or RNGrrrr information in the station's last position report. [APRS101] The circle color is the same as the most prevalent color in the icon.
- self - draw the local station on the map at its current position. If the local station has a working GPS, the last reported GPS position is used, otherwise the configured fixed beacon position is used.
- selfLikeOther - if true, draw the local station using the standard station type icons, otherwise draw a purple crosshair (for a fixed station) or course pointer (for a moving station).
- weatherAsWeather - draws weather stations as old-style weather map symbols indicating the reported wind direction and speed for each station.
- maxAgeDRObject - the maximum time in milliseconds that dead-reckoning should be predicted for an Object or Item since its last position report, defaulting to 10 minutes.

- maxAgeDRSpecialObject - the maximum time in milliseconds that dead-reckoning should be predicted for an Object or Item considered "special" since its last position report, defaulting to 1 hour. Currently not used.
- maxAgeDRStation - the maximum time in milliseconds that dead-reckoning should be predicted for a station since its last position report, defaulting to 5 minutes.

When StationRenderer is told to repaint its content, it draws the following content (skipping anything that is disabled according to the above properties):

1. Aloha circle(s) if any RF ports exist and have received enough data to compute Aloha circles.
2. all of the stations currently listed in the StationTracker that pass the current filter settings, with the callsign or alias for each station to the right of the station's icon.
3. all of the objects currently listed in the ObjectTracker that pass the current filter settings, with the object name or alias for each object to the right of the object's icon.
4. the local station's symbol and callsign.
5. any AttentionAlerts. These are flashing arrows pointing at a location on the map from the four cardinal directions. Green arrows indicate a station or object that was located by the Locate->Station or Locate->Object menu commands; yellow arrows indicate a station that has just transmitted a priority message; red arrows indicate a station that has just transmitted an emergency message. AttentionAlerts are created when the condition to display them occurs, and last for a short period of time (10 to 30 seconds, depending on the triggering condition) before expiring and deleting themselves from the active list of alerts.

The StationRenderer also accepts mouse click input. If the mouse is left-clicked upon the map and one or more station and/or object icons are under the click, all clicked-upon stations and objects will have their current state (from their StationState objects) displayed in a popup dialog box (the PopupStationDialog). Clicking elsewhere on the map closes the dialog and potentially opens another if there was anything under the new click point. If the mouse is right-clicked upon the map, the defined list of popup menu Actions is scanned; each Action is given the current map position and an array of StationState objects describing the set of stations and objects at the click-point, so it can report if the Action is relevant for the coordinates. If at least one relevant Action is found, a JPopupMenu is displayed with the relevant Actions on it, and the user is allowed to select an Action or dismiss the menu.

2.3.6 Implementation of the OpenStreetMap Renderer

There were several reasons why I chose to implement my own renderer for the street map data. Firstly, I had had professional experience with commercial closed-source vector-to-raster map renderer libraries which took an extremely long time to render even sparsely featured areas. Secondly, estimating the size of a colored map tile raster image for a typical usable PC screen resolution, the amount of disk storage needed to cache pre-rendered tiles at multiple zoom levels for any reasonably large geographical area would be enormous, exceeding the available disk space of most portable laptop or netbook computers. So I chose to render on-the-fly on demand for the screen area, but sub-divide the vector data into geographical blocks, so as to waste as little time as possible reading vectors that aren't visible on the screen by only reading vectors from blocks overlapping the map panel.

OpenStreetMap data is provided as a bzip2-compressed XML file. Compressed, the planet.osm.bz2 file is approximately 22 gigabytes of data (5 times the capacity of a DVD-ROM optical disk), and

uncompressed over 10 times that size. Some mirror websites provide subparts of the planetary data set, segmented into continent, country, and/or state/province, but the data is several months out of date (not surprising, given the amount of time it would take to read the planetary data set into a database and then read back and XML-encode each bounded subset). Even optimized bzip2 decompressors running as fast as possible on a 3GHz server took 132 minutes (over 2 hours) to decompress the entire planet file without doing anything useful with the contents. Furthermore, the map data is structured as a series of Node objects with an identifying number and a point position in latitude and longitude plus optional Tag data, followed by a series of Way objects each specifying an ordered list of Node identifiers making up the polyline or polygon plus optional Tag data. Such data is horribly inefficient to assemble into renderable polylines/polygons without a relational database, whose metadata and index overhead would consume as much disk space as the uncompressed XML file, aside from the processing and disk I/O overhead needed to look up and assemble a Way object.

So I implemented an OpenStreetMap importer that would:

1. scan through the gigantic input file;
2. build an efficient index of Node identifier to latitude/longitude coordinates;
3. build Way objects by converting each Way's ordered list of Node identifiers into lat/lon pairs using the index,
4. extract only associated Tags that provide information useful for rendering maps (for example, the name of the individual OpenStreetMap contributor that added the Way to the OSM central database would not help draw the Way any better), and discard the rest;
5. write the Nodes and Ways in a compact efficient binary format to files in 1-degree by 1-degree "tiles" of vector data, intelligently segmenting Ways that spanned multiple tiles into partial Way records in each tile.

This importer still takes a very long time to run, even in its latest incarnation with a 3-thread architecture with pipes and queues between them to keep the CPU working while blocked on disk I/O; with the planet-120704.osm.bz2 file from http://planet.openstreetmap.org on the same 3 GHz 8 core CPU server with 12 GB of RAM and a 64GB solid-state disk (SSD) SATA drive, the latest version of the importer took 11 hours, although this was a vast imporivement over previous versions running on a rotating conventional disk with rotational latency accessing the sectors on the disk. On the other hand, the resulting size of the complete set of binary tile files is only approximately 9GB (less than half the size of the compressed XML file), and each Way and Node record is organized efficiently for the renderer to use, not requiring parsing and interpretation like the XML file would..

The OpenStreetMap renderer itself was implemented as an OpenMap Layer subclass called OSMLayer, registered to the BasicMapPanel prior to the StationRenderer and OpenMap's GraticuleLayer, so as to be drawn behind those layers. Because the vector-to-raster rendering might still take a long time, the renderer was implemented to render to a Java BufferedImage object the same pixel dimensions as the BasicMapPanel, and OSMLayer's paint() method merely draws the contents of the pre-rendered image (if it is valid) to the screen on demand. The actual rendering is triggered if any of these conditions happen:

- the BasicMapPanel is resized (reported through the java.awt.ComponentListener interface);
- OSMLayer's paint() method is called and there is no valid image to draw and there isn't already a renderer running;

- the map is panned or zoomed, changing the Projection of the map (reported through the projectionChanged() method required of all Layer subclasses);
- any of the controls affecting the rendering of the individual Ways and Nodes are changed.

The controls affecting how the map is drawn include:

- enabling or disabling the background map as a whole.
- enabling or disabling drawing highway signs on road-type Ways that have the "ref" tag associated with them.
- enabling or disabling drawing Nodes as points of interest.
- changing the zoom level above which a category of Ways (identified by its OSM "highway", "railway", "waterway", "aeroway", or "power" tag value, which are combined in YAAC into a composite WayType, an enumeration type used to efficiently identify the Way category).
- changing the color or stroke style for a category of Ways. The default colors and stroke styles were chosen to match the maps in a hard-copy road atlas.

These controls are accessed through menu choices defined by the CoreProvider.

Two actual BufferedImages are used to support the common use case of map panning without zooming; this allows the old image to be drawn with an offset (leaving some edges of the map panel blank) whuile the updated projection is rendered to the other BufferedImage.

The actual rendering code is executed in a background thread; the OSMLayer class implements the java.lang.Runnable interface so it contains the actual code to be executed by the thread. The renderer gets the current Projection of the map panel and saves it with the BufferedImage into which the rendering will be drawn. The image contents are cleared (or the BufferedImage is created from scratch if the renderer was invoked for a map panel size change). The lat/lon coordinates of the four corners of the map panel are obtained to define the lat/lon bounding box of the map (as opposed to the pixel coordinate bounding box). The rendering thread then iterates through each 1-degree by 1-degree tile that intersects the bounding box, and reads the file of Way records for that tile (if the file exists). Each Way is checked to see if its bounding box intersects the map panel, its type is enabled for rendering, and the type's maximum zoom level is >= the saved zoom level of the projection. If all of these conditions are met, the Way's vertices are translated from lat/lon to pixel x-y coordinates. If the translated Way is larger than 2 pixels in either X or Y dimension, the paint color and stroke style are obtained. If the Way has a color, it will either be drawn as a polyline or filled as a polygon (as defined by the Way). If the Way has a name and the Way is larger than the length of pixels it will take to draw the name string, the name will be drawn along the Way. If the map zoom level is less than a hard-coded threshold for Node rendering and point-of-interest rendering is enabled, the corresponding Node file in the tile will be read and all Nodes inside the map panel bounding box will be rendered with their names and appropriate icons.

At every iteration of the nested loops in the renderer, the renderer's working Projection is compared to the current Projection of the map panel; if the Projection has changed (due to a pan or zoom of the map) or the map panel size has changed, the renderer is aborted so it can be restarted with the new projection.

2.3.7 Implementation of the CoreProvider

The CoreProvider defines the basic capabilities that YAAC will provide before plugin extensions are added. The version number of YAAC, my name as author, and a description and desktop icon are all

specified in the constructor.

The getPortConnectorTypes() method is overridden to define 5 basic interface classes. Four of them (SerialTncConnector, SerialGpsConnector, SerialWeatherConnector, and KenwoodConnector) will only be defined if the RXTX library can be successfully linked to its native code library. The fifth, AprsIsConnector uses only the built-in network socket I/O classes in the Java runtime library and can therefore always work if YAAC can be successfully started.

The getConfigurationPanels() method is overridden to define 5 panels of configuration controls. These panels are somewhat arbitrary divisions of the total set of configuration controls for YAAC, trying to keep each panel small enough to fit on a netbook's limited-size screen (in fact, that was one issue raised during alpha-testing, because the testing user couldn't reach the Save button at the bottom of a configuration panel because the panel was taller than the screen). The core configuration panels are:

- General Parameters - time to preserve received messages, logging options, message addresses to accept as locally of interest.
- Transmit - parameters affecting the transmission of messages, including digipeating aliases and smart beaconing timing.
- Ports - listing all the configured interface ports, with controls to add, modify, and delete port definitions.
- Beacon - definition of the beacon message the user wants YAAC to send for the local station, including format (conventional, compressed, or MicE), symbol code, whether GPS position should be used, whether altitude and speed should be included, what the PHG parameters are, whether weather station data should be included, the proportional pathing digipeater choices to use, and whether to indicate when the local station is actively manned or not.
- Preferences - user preferences regarding fonts and display units (statute miles vs. kilometers, etc.).

The getFilters() method is overridden to provide a large collection of filter types:

- filter by age of the message (excessively old messages are not shown or are even discarded from YAAC's data pools);
- filter by whether the remote station indicates it is actively staffed (an APRS 1.1 feature);
- filter by whether the messages have priority and/or emergency precedence;
- filter by whether the stations are directly heard or are within 1 RF digipeat of the local station;
- filter by radial distance from the local station;
- filter by the remote station type (as specified by the symbol table ID and symbol code in the station's packets);
- filter by the sending stations' callsigns;
- filter by the last digipeater to relay packets;
- filter by the to-call (destination callsign) (which can indicate the software type of the sending station);
- filter by the category of station (as also implemented by the Kenwood TM-D710);

- filter by the interface port which brought the packets into YAAC.

The getMenuItems() method is overridden to define several dozen menu commands that were incrementally added to YAAC as the need for these commands arose and/or alpha-testers made feature requests for the commands. Each is implemented by subclassing one of the convenience abstract superclasses AbstractMenuAction (for frame window menu bars) or AbstractPopupMenuAction (for popup menus on the maps and table views); the actionPerformed() method of each Action is overridden with code to actually carry out the requested command. Many commands just call "setter" methods on other classes in YAAC to change their state; for example, the View -> View Map Layers -> Show Range Circles command just toggles the rangeCircles property on the StationRenderer to alternately enable and disable rendering range circles on the map. Other menu actions have more complex code as needed. The actual complete list of implemented commands is visible in the GUI and in the online help.

The getAboutAttributions() method is overridden to specify a multi-line description of core YAAC, its author, and the 3rd-party software included with YAAC.

The getHelpSet() method is overridden to locate the YAACHelp.hs file and load it into a HelpSet object.

The runInitializersAfter() method is overridden to register the StationTracker and ObjectTracker to listen for incoming APRS packets, and to register all the core Query message response handlers.

Along the course of adding all the core features, the various abstract superclasses in the pluginapi package were extended multiple times to add more capabilities needed by the newer features.

2.4 The Legalities of YAAC

Because I am a professional software engineer, employed for the specific purpose of designing and writing software for my employer, there was initially a problem with my releasing YAAC to the general public, which was that I didn't have the right to do so because, strictly speaking, I didn't own YAAC, despite having written it on my own time on my own computers (not company computers); legally, my employer owned YAAC, per my employment contract. Using the directions provided by the Free Software Foundation, I requested my employer to release any claims of ownership in YAAC to me personally. After several weeks, including evaluation of my software by the company's Chief Scientist and the Legal department's Senior Counsel to confirm I was not developing anything of interest to the company, I was finally given written notice that my employer released all ownership interest in YAAC to me, and I could do with it as I willed.

It wasn't surprising that it took a long time to resolve this issue, as the company and its relevant officials had more urgent and important (to the company) tasks to carry out than freeing my software; it was highly appreciated that they did take the time and gave me ownership of my software.

2.5 The Testing of YAAC

YAAC was originally tested on my own personal computers, which included a Fedora Core 13 Linux™ development system and a dual-boot laptop running Microsoft Windows XP® Service Pack 3 and Fedora Core 15. Besides (and even before) Internet connectivity, YAAC was initially tested with a MFJ-1278 multi-mode TNC (TNC2-compatible) connected to a Radio Shack scanner or an Alinco DR-1200 transceiver, a Kenwood TM-D710 mobile transceiver and TH-D72A handheld, a DeLorme TripMate GPS receiver, a Byonics GPS2 GPS receiver, and a Peet Bros Ultimeter 500 weather station. Some of the testing on the TM-D710 or TH-D72A and laptop was done in mobile and portable operation to try different areas' "flavors" of APRS and test the smart beaconing feature. Other "flavors" were tested as a

side-effect of load-testing YAAC with a APRS-IS connection using a large (8000km) radius filter; this load-testing was to ensure YAAC would behave reasonably (which it didn't always do) under extreme load.

Once I had obtained legal ownership of YAAC, a Yahoo group, yaac-alpha-testers, was created, a webpage was established describing YAAC and containing a current ZIP file of the executable YAAC distribution (updated regularly as new builds are prepared) and directions for installing YAAC, and an e-mail was sent to the aprssig@tapr.org mailing list inviting the list members to become alpha-testers. A total of 46 hams from around the world volunteered to try out YAAC, and most sent at least one bug report, feature request, or new platform support request; some sent several reports during the alpha test cycle. Testing was ad-hoc: each alpha-tester would try YAAC on their computer(s), and report back with any issues they had. The issues generally fell into one of five categories (in increasing order of occurrence):

1. lack of support for their preferred operating system or radio/TNC hardware;
2. complaints of sluggish response (mostly OpenStreetMap map rendering and importing);
3. user error due to unclear usage directions (requiring a GUI change and/or more online help);
4. feature requests for new capabilities in YAAC;
5. actual defects in YAAC requiring fixes.

My own personal testing was systematic unit and feature testing, verifying the resolution for whatever issue a user (including myself) found was working as I had designed it. I also carried out various performance (throughput) tests for both incoming packet rates (the above-mentioned 8000km radius APRS-IS feed) and map rendering. The Hprof feature in the Java virtual machine was used to collect CPU samples to identify "hot spots" in the software that would most benefit from performance tuning efforts. This became significant when large numbers of mangled APRS packets were received, generally two packets jammed together into a single line. This was eventually determined to be buffer overflow because of the extra processing required at the last byte of each incoming frame. A more heavily multi-threaded architecture with queues between the I/O port reading threads and the frame processing thread alleviated most but not all of this problem.

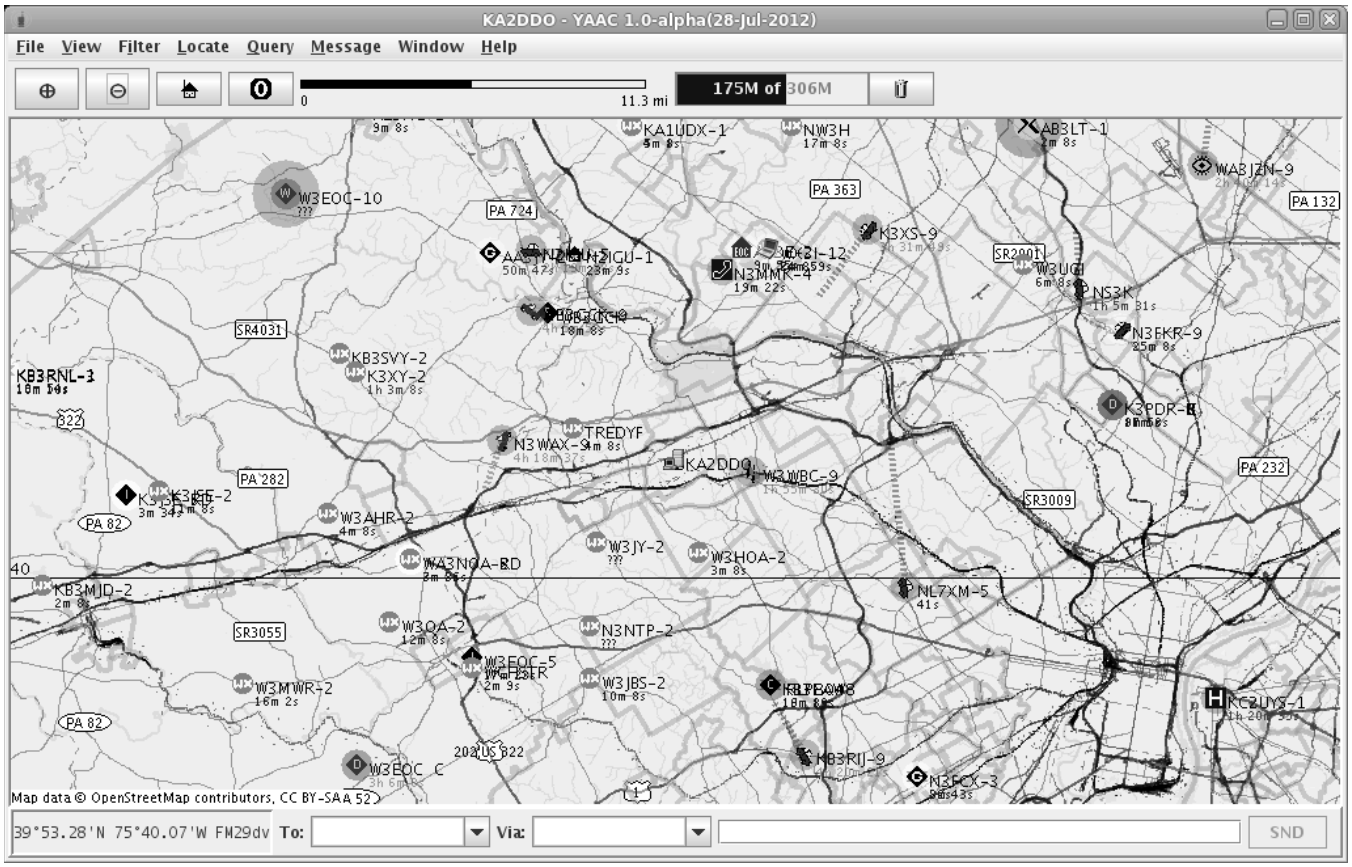
OSM Map rendering and data file import time were both taking far too long, so considerable effort was spent trying to reduce the processing time of these functions. Because even the latest version of the OSM importer took half a day to process the planet.osm.bz2 file, a new feature was added (by alpha-tester request) to allow users to download pre-imported map data from my website, obviating the need to do the import-and-tiling processing themselves.

Two "features" were found in the standard binary distro of RXTX. Firstly, Fedora Core 15 and later changed the directory for device locking files and RXTX did not know about this change, causing an inability to access serial ports on these dialects of Linux. A work-around was discovered for this problem. Secondly, RXTX by default only would recognize device files with the names /dev/ttyS* or /dev/ttyUSB*, and would not recognize an alpha-tester's Argent Data Systems OpenTracker USB, because Linux identified the device as /dev/ttyACM0 (an Arduino board) and this device name was not on the RXTX search list. A request to fix this was sent to the RXTX development team, but in the meantime, the Linux native libraries in the YAAC-bundled RXTX distro were replaced with recompiles using the alternate Linux device filename list (recognizing Arduino-based devices and other Unix-type serial port names).

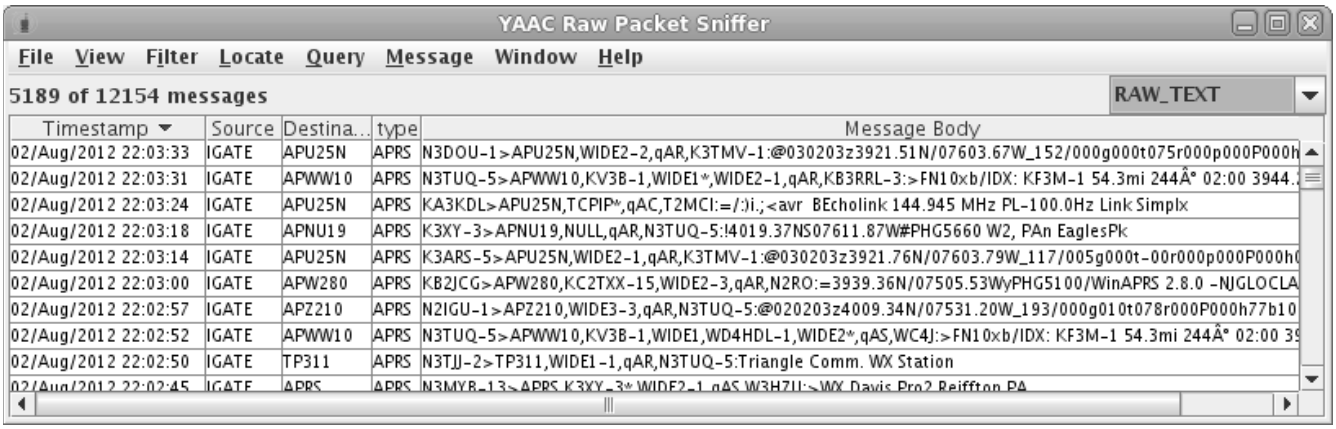
3 The Delivered Product

As of August 2nd, 2012, YAAC is a reasonably reliable product working on 4 different operating systems: Microsoft Windows®, Linux (tested on Fedora Core and Ubuntu distros), Apple Macintosh OS X®, and FreeBSD®, using either the Sun Java runtime or the OpenJDK runtime. It supports TNC2-compatible and Kamtronics TNC's, and the embedded TNC's in Kenwood VHF/UHF radios (there is still an unresolved reported issue in communicating with the Kenwood TS-2000's built-in TNC, but I don't have a TS-2000 to test with). Several different NMEA-0183 compatible GPS receivers were verified to work with YAAC, although some had problems with sentence garbling (possibly due to sending sentences at too high a rate for the build being tested).

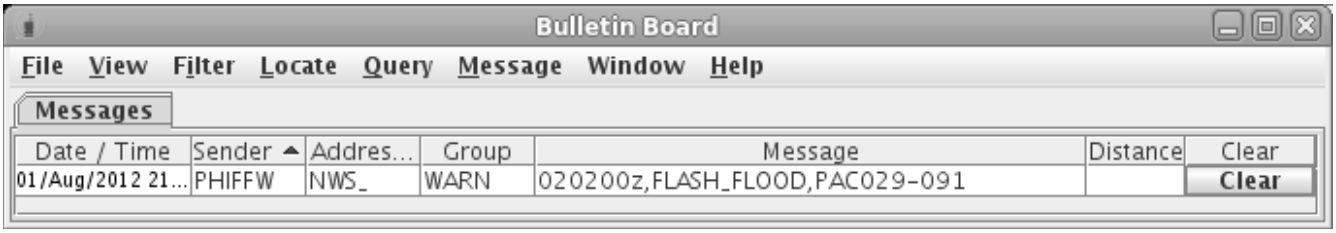
Here's a sample view of tactical state in my neighborhood:



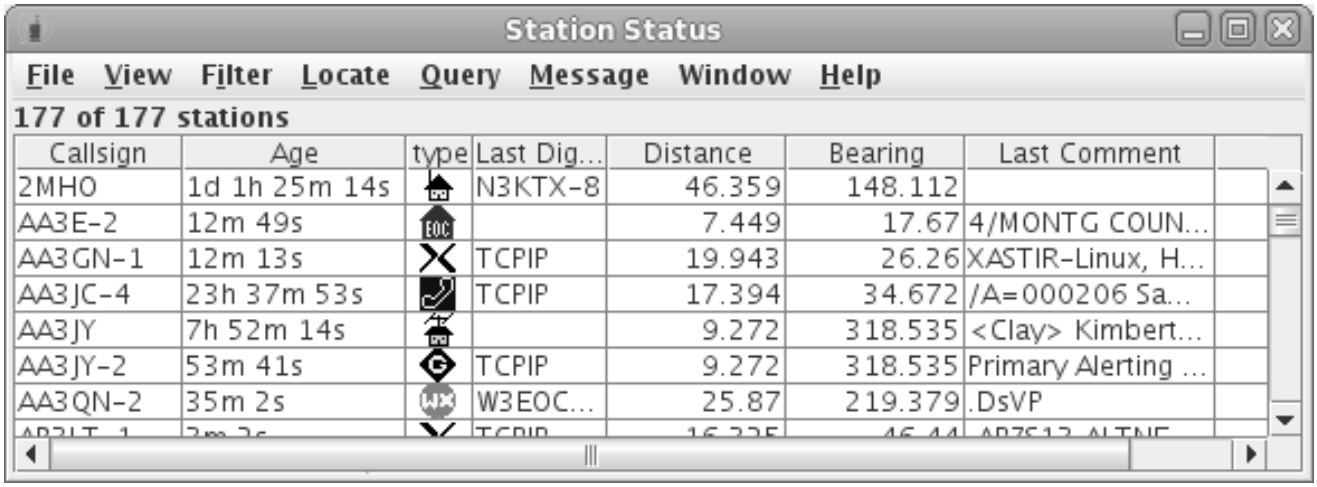
Here's raw messages received by YAAC:



A current bulletin board:



And here's part of the current station status:



4 Future Plans for YAAC

As of the date this paper was completed (2 August 2012), YAAC has not yet been released in open-source, and is still in alpha-test. Upcoming plans include (but are not limited to):

1. Complete alpha-testing, so that all implemented functionality is working for the alpha-test team.
2. Finish writing the first release of the online help and a plugin SDK (to guide other developers in writing extensions to YAAC).
3. Making a few more corrections to the architecture of YAAC to further separate the "back-end"

communications and message history database code from the GUI, so that it will be easier to write an alternate GUI based on, for example, the Android operating system (with its non-Java-standard UI library).

4. Finish properly adding LGPL-compatible copyright notices to all the source files (this was not done during initial development, before the licensing policy was finalized).
5. Bring YAAC up to full (instead of only partial) compliance with the APRS 1.1 and APRS 1.2 specifications; it is currently only fully compliant with APRS 1.0.1.
6. Create a SourceForge project for delivering the pre-built YAAC binary distribution and source distribution, replacing the personal website of the author as the distribution point.
7. Set up a formal bug-tracking system (not just an e-mail mailing list).

All of the above are prerequisites for officially releasing YAAC as an open-source product.

Additional plans (once the above tasks are complete) include:

8. Add support for AGWPE, soundmodem, and Sivan Toledo's javAX25 software modem/TNC's.
9. Add support for the US Geological Survey's National Elevation Database data format and elevation contour plotting.
10. Port YAAC to Android.

5 Conclusions

YAAC is a viable software product, though its popularity relative to other existing APRS clients is yet to be determined. Writing a full-functioned APRS client from scratch is a daunting challenge because of the wide range of capabilities needed. Trying to describe a year's software development in a single article is challenging because of the amount of condensation required; true knowledge of YAAC's internals will require access to the source code.

Open-source development would be faster than doing closed-source development as a solo developer; however, the risk of doing so in a brand-new product (before the design is settled down) would probably create a failure, because there were many incorrect initial decisions made from incrementally building the requirements and feature list. On the other hand, I have great sympathy for the alpha-testers that had to wait for me to get around to resolving their bug reports with a closed-source development cycle run by a volunteer in his spare time.

Incidentally, the process of writing this paper actually forced me to create a better software product. Reviewing the software in enough detail to explain it to others help me find mistakes I had made in implementation and repair them.

6 Acknowledgments

I would like to thank Bob Bruninga, WB4APR, for coming up with the idea of APRS in the first place.

I would also like to thank the team of YAAC alpha-testers for their efforts in testing YAAC and producing bug reports and improvement suggestions, including (in no particular order and a non-complete list) Lee Bengston K5DAT, Lynn Deffenbaugh KJ4ERJ (author of APRSIS32 and APRSISCE), James Ewen VE6SRV, Max Wheatley ZL2MAX, Carl Makin VK1KCM, Kurt Savegnago KC9LDH, Gerhard F5VAG, John K2ZA, Tom Hayward KD7LXL, Jan Peterson KD7ZWV, Ron Werthman VA3ACZ, John Zaruba Jr., and more.

All trademarks mentioned in this document are the property of the respective trademark holders.

7 References

[APRS101] Ian Wade G3NRW, ed., "Automatic Position Reporting System: APRS Protocol Reference, Protocol Version 1.0", http://www.aprs.org/doc/APRS101.PDF

[APRSTAC] Bob Bruninga WB4APR, "APRS Tactical Real-Time Operations", http://www.aprs.org/APRS-tactical.html

[AX25] William Beech NJ7P, Douglas Nielsen N7LEM, Jack Taylor N7OO, "AX.25 Link Access Protocol for Amateur Packet Radio, Version 2.2", July 1998.

[COMPRESS] Apache Commons compress, http://commons.apache.org/compress/

[FIELD] Bob Bruninga WB4APR, "APRS Touchtone: Field Data Entry", http://www.aprs.org/aprstt.html

[GPL] GNU General Public License, http://www.gnu.org/licenses/gpl.html

[HELP] JavaHelp 2.0, http://javahelp.java.net/

[KISS] Mike Chepponis K3MC, Phil Karn KA9Q, "The KISS TNC: A simple Host-to-TNC communications protocol", ARRL 6th Computer Networking Conference Proceedings, pp. 38-43, 1987, http://www.ax25.net/kiss.aspx

[LGPL] GNU Lesser General Public License, http://www.gnu.org/licenses/lgpl.html

[NWS] Peter Loveall AE5PL, "Multiline Object Format", http://www.aprs-is.net/WX/MultilineProtocol.aspx

[OPENMAP] BBN Technologies, "OpenMap Open Systems Mapping Technology", http://openmap.bbn.com/.

[OSM] The OpenStreetMap Foundation, http://www.openstreetmap.org.

[RFC1122] Internet Engineering Task Force, "Requirements for Internet Hosts -- Communications Layers", pg. 12, http://tools.ietf.org/html/rfc1122

[RXTX] RXTX Wiki, http://rxtx.qbang.org/wiki/index.php/Main_Page