aprsc, an open-source high-performance APRS-IS core server

2012-08-17

Heikki Hannikainen, OH7LZB

http://aprs.fi/

hessu@hes.iki.fi

Keywords: APRS, APRS-IS

Abstract

aprsc is a high-performance, open-source APRS-IS server software for Unix-like systems such as Linux, FreeBSD, Solaris and Mac OS X. This paper documents its main features, design, some implementation details and results of performance measurements.

Introduction

During the past years the APRS-IS network has been using javAPRSSrvr as its sole server software, without serious alternatives being available. Some APRS-IS server operators have expressed interest in an open-source C implementation of a server. In early 2008, when the core APRS-IS servers were having some capacity issues, Matti Aarnio, OH2MQK and myself discussed the scope and feasibility of reimplementing the wheel, and decided it would be an interesting exercise in high-performance server programming. The capacity issues were soon mitigated by moving CWOP clients to their own set of servers, but we had already kickstarted our own development project. The development has continued slowly, and a final sprint towards feature completeness has been under way during the summer of 2012.

In this paper we introduce aprsc, a high-performance, open-source APRS-IS server.

Scope of features and target environment

aprsc, pronounced *a-purrs-c*, is a dedicated APRS-IS server software intended to be used on the core and Tier2 APRS-IS servers. It is relatively small and compact, and only contains features needed by these APRS-IS servers. It intentionally lacks features such as igating, digipeating and interfacing to radios, leaving those functions to the numerous existing and specialized igate applications.

aprsc is written in the C language, and runs on Linux and Unix-flavoured systems. Porting it to run on Windows, VMS or other platforms should be possible, but the principal authors are professional Unix programmers and wish to leave porting efforts to contributors who are more eager to work on those platforms.

Functional requirements

A list of necessary features was collected before the development started. As is usual with software development projects, several features were overlooked and needed to be added in later. Some items took more time than anticipated. The list was cut down to a minimum according to what is actually used by the servers. Any features normally implemented by client software, such as object beaconing, are

generally not needed in the server.

Basic functional requirements implemented by aprsc:

- Duplicate packet filtering within a 30-second sliding window
- Q construct algorithm processing
- Client-defined filters and filters configured per port on the server
- APRS packet parsing only as necessary to support filtering
- I-gate client support
- Messaging support
- UDP client support
- UDP core peer links
- Uplink server support
- Passcode validation
- Web-based status view
- Machine-readable status dump over HTTP
- HTTP position upload using POST
- Full IPv4 and IPv6 support
- Configurable access lists on client ports
- Logging to either syslog, file or stderr
- Built-in log rotation when logging to a file

Architectural requirements

aprsc's basic design was drawn out in a pizza session in early 2008. Some requirements were put in just to increase the technical challenge, and to make the server match Internet-scale requirements, instead of simply meeting real-world APRS requirements.

High throughput

As of August 2012 the full APRS-IS feed is about 50 packets per second, 40 kbits/second of APRS data (excluding TCP/UDP/IP overhead). A busy server with 300 clients, many of them full feeds, can transmit about 2 Mbit/s of APRS data. aprsc should be able to transmit at least 100 Mbit/s of data on common server hardware.

Small enough latency: 10s of milliseconds

Latency in the millisecond range on the APRS-IS network is not very critical, as long as it is not significant when compared to normal Internet latencies and especially when compared to the 30 second duplicate checking window of APRS-IS. Latency caused by the 1200 bit/s APRS radio channel is the

dominating factor in any common case.

However, if a server would become overloaded for some unforeseen reason, there is a risk that it would delay packets long enough for them to escape the 30-second duplicate checking window. This could cause the load to increase even more.

The server needs to ensure that client connections are closed if queues become too large, and packets delayed within the server must be dropped before passing them on.

Support for thousands of clients per server

Most APRS-IS servers are configured to only accept 200 to 400 parallel client connections. Modern Internet servers (such as nginx and lighthttpd) are commonly designed to handle over 10000 connections. aprsc should handle a large amount of connections without consuming an awful lot of memory.

Support for heavy bursts of new clients: CWOP hits every 5 or 10 minutes

The problem at hand on the APRS-IS core in early 2008 was a large number of Citizen's Weather Observer Program weather stations connecting the servers at regular, synchronized times (00:10:00, 00:15:00, 00:20:00) due to an unfortunate design mistake in the client programs. The stations should really call in their reports at a randomized offsets (one client reporting 00:13:15, 00:18:15, 00:23:15..., another at 00:14:34, 00:19:34, 00:24:34). Many client programs have been fixed to distribute the load by now, but older versions are still in use.

Scalability over multiple CPUs

A single CPU core could easily handle current traffic load on the APRS-IS. However, the speed of individual processors is not increasing as before, and low-cost chips with multiple slower CPU cores in a single socket are becoming the norm. To make the solution future-proof, and to make things a bit more interesting, we chose to start working on an application which could distribute its workload to multiple CPUs.

> *Some people, when confronted with a problem, think,*
>
> *"I know, I'll use threads," and then two they hav erpoblesms.*
>
> \- Ned Batchelder

When a lot of data needs to be shared between threads of parallel execution, using threads (as opposed to multiple parallel processes) is usually more effective, since threads within a process share the same memory address space and can directly access common variables and buffers. Threads seem like the right choice for aprsc.

Low lock contention

To keep threads from stepping on each other's toes by writing on the same data structures at the same time synchronization and locking tools such as mutexes are used.

If there are multiple threads accessing the same variable very often in a way that requires locking, all of

the threads might end up spending most of their time waiting for the lock to become available. Lock contention makes some multi-threaded software effectively single-threaded, being unable to utilize more than a single CPU core. In less serious cases it can simply lead to increased CPU usage due to numerous locking operations.

aprsc uses four main methods to reduce lock contention:

- Read-write locks to allow concurrent reading of some data structures
- Reducing the amount of required locks by keeping data local to a thread (i.e. making it unnecessary for one thread to access another thread's data)
- Transferring data between threads in blocks of multiple data units (packets or clients) with a single locking and unlocking event
- Limited use of GCC built-in functions for atomic memory access [GCC-BUILTINS]

Further reduction of locking can be reached later with increased use of atomic memory access primitives and non-blocking algorithms [WIKI-NONBLOCK].

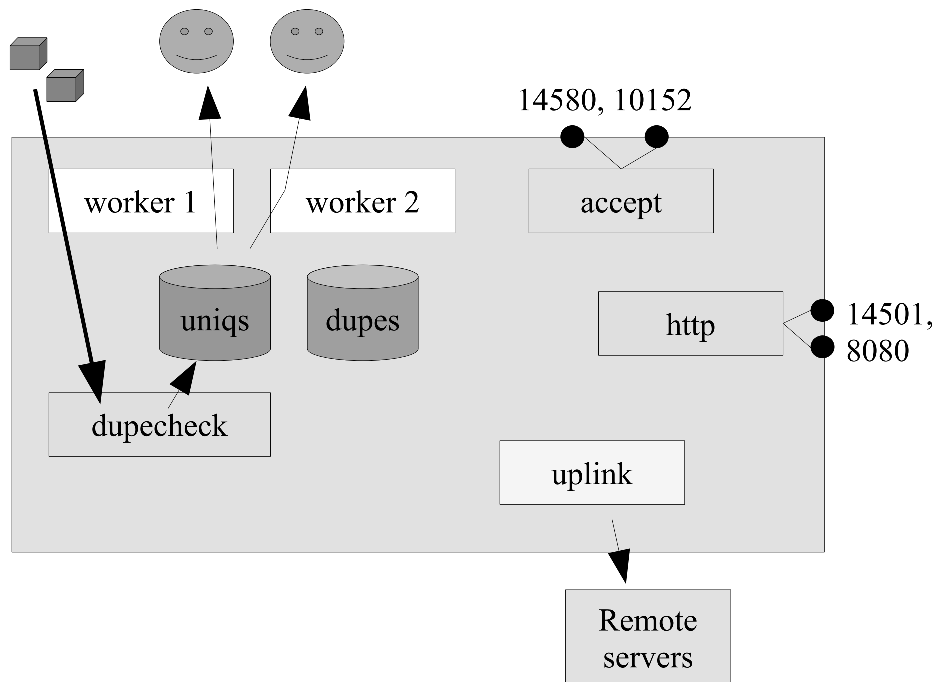
Low context switch overhead

A traditional design pattern for servers has been to create a new thread for each new client. In addition to taking up additional memory (per-thread stack space), having a large number of threads increases lock contention and context switching overhead. If a single packet needs to be sent to 200 clients, and each client is managed by its own thread, each thread needs to be woken up from sleep and scheduled to be executed on a CPU. It is a relatively slow operation for a CPU to switch from a thread to another, and the number of context switches per second is a closely watched overhead indicator on busy servers.

Most of recently developed high-performance Internet server software are **event-driven**, meaning that a single process or thread does all processing for a large number of clients, which reduces the need for context switches and other threading-related overhead. nginx and lighttpd are prime examples from the world of HTTP servers – both run only a few processes.

For aprsc, a hybrid threaded / event-driven approach was selected. aprsc runs a small, fixed number of threads, close to the number of CPU cores on the server, so that multiple CPU cores can be utilized, but the context switches between a high number of threads can be avoided.

Threads of aprsc



Drawing 1: Threads within aprsc

Most of the work is done by **worker threads**. In real-world APRS-IS of 2012, 1 worker thread is enough, but if a server was heavily loaded with thousands of clients, 1 less than the number of available CPU cores would probably be optimal. Each connection is assigned to one of the workers after the connection is opened. All traffic passing to or from a client or a remote server are processed by the worker.

A **dupecheck thread** maintains a cache of packets heard during the past 30 seconds. There is a dedicated thread for this cache, so that worker threads do not need to compete for access to the cache's data structure. The thread receives incoming packets from the worker threads, does dupe checking, and puts unique and duplicate packets in two ordered buffer queues. The workers then walk through those buffers and do filtering to decide which packets should be sent to which clients.

An **uplink threads** initiates connections to upstream servers and reconnects them as needed. After a successful connection the socket will be passed on to a worker thread which will proceed to exchange traffic with the remote server for the duration of the connection.

An **accept thread** listens on the TCP ports for new incoming connections, does access list checks, and distributes allowed connections evenly across worker threads. Login command processing is done within the worker thread.

An **HTTP thread** runs an event-driven, **libevent2**-based HTTP server to support a status page and HTTP position uploads. Since C is not the most convenient language for implementing friendly web user interfaces, the status page is produced using modern Web 2.0 methods. The HTTP server is only

able to generate a dynamic JSON-encoded status file and serve static files. A static index.html file sets up the basic structure of an empty status page and loads a static JavaScript file, which then periodically loads the JSON status data and formats the contents of the status page within the client's browser. This approach allowed clean separation of server code (C) and web presentation (HTML5/JavaScript/jQuery). A 48-hour statistic graph was added as a finishing touch with the help of the excellent Flot library.

Other optimizations

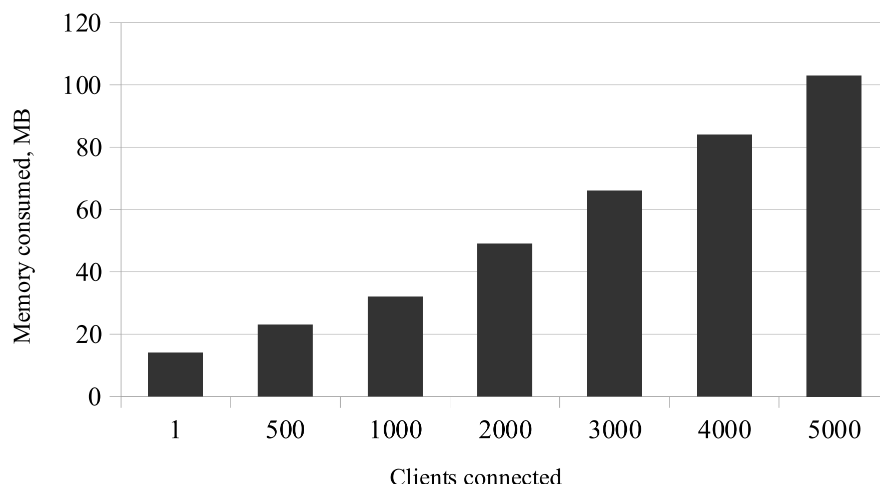
Copying of packet data around in memory is kept to a minimum. Incoming packets are parsed in place, and a single set of copy operations is done to reassemble the packet with a rewritten Q construct. A single, shared instance of the outgoing packet is then used by in the outgoing packet path by the worker threads.

Matti provided a cellmalloc implementation to reduce memory fragmentation and other memory allocation overhead which often becomes noticeable when continuously allocating and freeing large amounts of small memory records such as APRS-IS packets. Instead of calling the operating system's memory allocation separately for each and every packet, memory is preallocated in larger blocks of up to 2 megabytes. Separate packet buffer (pbuf) blocks are used for small, medium-sized and large packets.

Performance

The following performance figures were obtained on a Linux 2.6 system, Ubuntu 10.04 LTS, x86_64, Quad-core Xeon E5450 @ 3.00GHz (processor launched by Intel in 2007, sold in 2008, server found in the dumpster and "recycled" to amateur use in 2012). On Linux, 100% CPU usage indicates that a single CPU core is fully utilized.

The first diagram illustrates amount of memory allocated per client socket, measured by generating idle, logged-in clients simulated by a testing program.

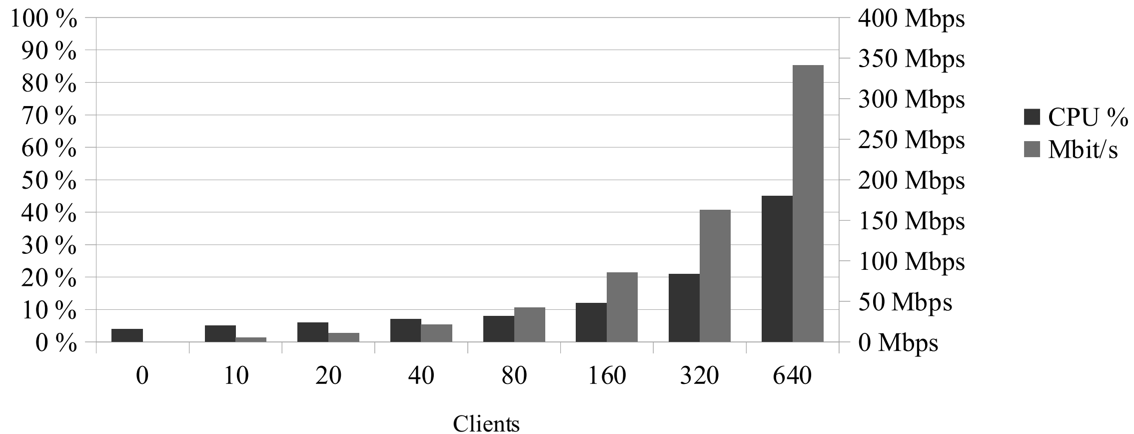


Core server load with full-feed clients

Let us imagine an APRS-IS load of the future with 20 times the traffic of today, 1000 packets per second fed in to the server (2012-08 nominal rate is about 50 packets/s), and see how aprsc performs as a core server with a large number of full-feed clients on port 10152. The test was performed by feeding a day's worth of recorded real APRS-IS packets at a controlled rate of 1000 packets/s. At the nominal 50 packet/s rate CPU usage was too low to be measured accurately.

CPU utilization with full-feed clients

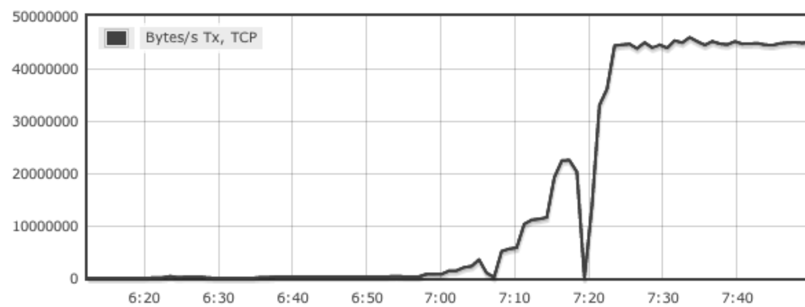
1000 packets/s input, ~700 non-duplicate packet/s output to each client
CPU % of single core of Xeon E5450 @ 3.00GHz from 2007



The following screen shot displays a part of aprsc's status page while executing the 640-client 1000 packets/s test.

Totals

Clients	643	0/s
Connects	4184	0/s
Bytes Rx TCP	583651420	92261/s
Bytes Tx TCP	93610818215	45819041/s
Bytes Rx UDP	0	0/s
Bytes Tx UDP	0	0/s
Packets Rx TCP	6431545	1024/s
Packets Tx TCP	1029937677	507803/s
Packets Rx UDP	0	0/s
Packets Tx UDP	0	0/s

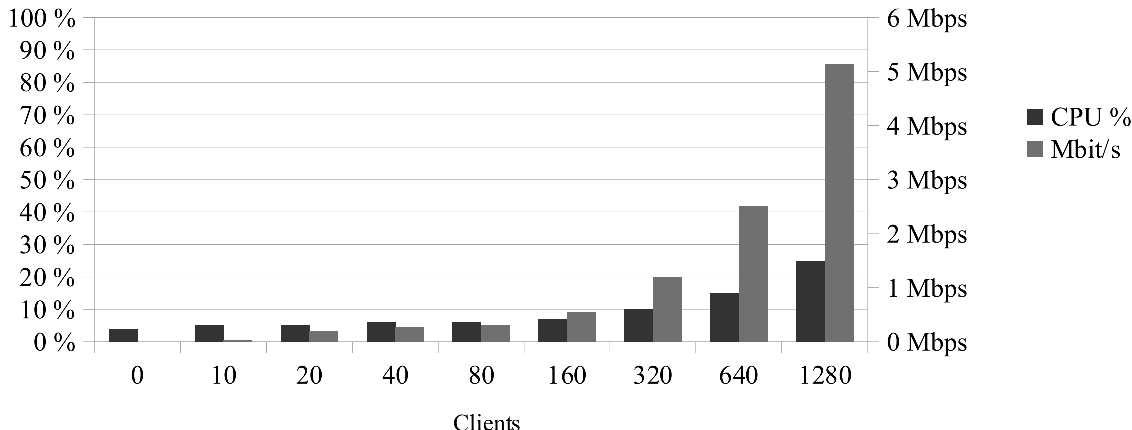


Duplicate filter

Duplicate packets dropped	1677258	252/s
Unique packets seen	4754043	794/s

Filtered-feed client-facing server load (Tier2 leaf)

The following chart displays CPU usage as a function of number of clients, when client-defined filtering is used on port 14580. The set of filters used by the clients was collected from a real-world APRS-IS server and contains a mix of range, area, packet type and callsign filters.



Real-world measurements

Real-world testing with 150 (+/- 2) real clients on T2FINLAND (finland.aprs2.net), a Solaris 11 server running on a HP blade server with four 3 GHz Xeon CPUs, shows reduced memory and CPU usage when compared to javAPRSSrvr 3.15. Roughly the same clients were connected to both, and switching back and forth between the two applications every 10 minutes and waiting for the clients to connect again displayed repeatable results. The CPU usage samples are 60-second averages obtained using the **top** utility. On Solaris the CPU percentage is calculated over all of the CPUs in the system – 100% on a 4-CPU system means that the process is fully utilizing all 4 CPUs, while 25% indicates that a single CPU's worth of processing power is used. NLWP is the Number of Light-Weight Processes (threads) running within the process.

```
  PID USERNAME  SIZE   RSS STATE  PRI NICE   CPU PROCESS/NLWP
18074 javaprs   164M  125M sleep   59    0  4.2% java/367

  PID USERNAME  SIZE   RSS STATE  PRI NICE   CPU PROCESS/NLWP
 9999 aprsc      30M   29M sleep    1    0  0.3% aprsc/7
```

120 megabytes, or 4.2% of 4 CPUs (12% of a single CPU core), is not a lot. If the CPU numbers were 42% vs. 3%, the optimization results would be more interesting.

Quality control

aprsc comes with an APRS-IS server test suite implemented using the Perl Test framework. A "make test" command executed in the tests/ subdirectory will execute automated tests for all of the basic functions of the server in about 2 minutes. Individual test scripts run fake APRS-IS servers, clients and peers around the tested server, and pass various valid and invalid packets through the server, checking for expected output.

Test-driven development methods have been used during development. For each APRS-IS protocol feature, a testing script has been implemented first, based on existing documentation and wisdom collected from the mailing lists and communication with other developers. The test case has been validated to match the functionality of javAPRSSrvr (by running javAPRSSrvr within the test suite), and only then the actual feature has been implemented in aprsc. This approach should ensure a good level of compatibility between the components and prevent old bugs from creeping back in.

Final notes

In hindsight, aprsc is slightly over-engineered and over-optimized for current real-word requirements. Sufficient performance could have been obtained with much less work. Reached performance, however, appears to match our initial goals.

aprsc has been in use on T2FINLAND and some of K4JH's T2 servers since early August 2012. A few bugs in client compatibility have been found and fixed, and it is likely that more will be found as more clients are exposed to aprsc in the wild. Some client-side bugs have also been identified and reported.

Some threading-related bugs have been introduced and subsequently fixed. A single-threaded event-driven server could provide sufficient performance while being easier to debug.

Links

aprsc home page: http://he.fi/aprsc/

Latest source code in version control: https://github.com/hessu/aprsc

Discussion group: https://groups.google.com/forum/#!forum/aprsc

References

[GCC-BUILTINS] Built-in functions for atomic memory access,

http://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html

[WIKI-NONBLOCK] http://en.wikipedia.org/wiki/Non-blocking_algorithm

Thanks

Pete Loveall, AE5PL, for javAPRSSrvr and all the answers to my questions about APRS-IS

Javier Henderson, K4JH, for patient beta testing and proof reading

Tier2 network folks, for allowing real-world testing on T2 servers