Tracking flights with Tcl and SQLite (and a bit of C++)

Peter da Silva

Tcl 2018

September 20, 2018

Summary

FlightAware needs a way to keep track of flights in the air, for web pages and various services we provide. We receive input from thousands of sources, including national air traffic control systems, airlines, satellites, and an extensive network of hobbyists passing on reports from aircraft ADS-B transponders. This data is in a variety of formats and of varying levels of reliability. This is converted (though software that Zach Conn discussed in `https://www.tcl.tk/community/tcl2016/assets/talk37/hyperfeed-paper.pdf`) into a stream of "flight events" - landings, takeoffs, positions, weather information, scheduled arrivals and departures and actual arrivals and departures.

This data is ever-growing, not only are there more aircraft in the skies, but new technology provides more frequent and more complex streams of data to deal with.

This stream of events needs to be turned into a coherent view of what aircraft are actually in the air, where they have been, and where they are going so we can produce those pretty blue images on our website.

For the past decade this data has been read, aggregated, and stored in a cluster of servers running a package called "Birdseye". The core of Birdseye has been Tcl and the "Speedtables" Tcl extension. We have recently created a replacement for Birdseye using Tcl, SQLite, and quite a bit of C++.

Birdseye

Birdseye can be thought of as a system that accepts flight events on input, turns them into flights, and provides a way for clients (such as webservers) to ask questions about flights.

The input is in a format called "daystream" which I touched on in `https://www.tcl.tk/community/tcl2017/assets/talk95/Paper.pdf`. A daystream file is a series of timestamped events stored as a set of tab-separated key-value pairs, with the first two tagged columns being seconds in the UNIX epoch, and a sequence number within that second. This may be read directly from a local file, or over a network connection from a remote server. The last record in each file is the time the file was completed and the path to the next file to be read.

This format is easy to read, easy to synchronize, and easy to distribute over the network.

Flightaware uses a number of daystream instances, the one that Birdseye reads is "controlstream". It reads it in Tcl and updates a set of speedtables containing the current state of every flight in the air, all over the world. This data is periodically snapshotted to disk, and when Birdseye starts up it reads the snapshot and starts reading from controlstream starting when the snapshot was taken and continuing to the current time. Once it's caught up, it registers itself and starts accepting queries...

The output side is the query interface. This interface is over a network socket. The client connects, write a query in the format of a Tcl command, and receives a single line containing the result of the query as a Tcl list. The queries themselves look a lot like Speedtables search queries, and this is not coincidence... the first version of what became Speedtables grew out of the earliest ancestor of Birdseye.

Birdseye problems

First of all, startup of a new Birdseye instance, or restart of a running Birdseye, is pretty slow. It has to read a significant snapshot of its previous state, and then read controlstream as fast as it can until it's caught up.

Second, reading controlstream in an interpreted language that does a lot of string manipulation has become a bottleneck. When there's a lot of air traffic, Birdseye instances have been known to be unable to keep up with real time.

Eagle Eye

The first candidate for replacing Birdseye was Eagle Eye. Instead of having a cluster of identical Birdseye servers, Eagle Eye had a single controlstream reader that used a large number of threads to write the flight status to a Cassandra cluster, and instead of using the trackstream interface, clients would connect to Cassandra and make queries in CQL, a SQL-like query language used by the Cassandra database.

Eagle Eye Problems

First of all, the latency of Cassandra was high enough that it wasn't practical for the readers to pull the information they needed to track flights out of its own database, the way Birdseye did withe Speedtables. This meant the master node had to maintain its own subset of the in-flight status, and reload it from Cassandra on startup. This wasn't as bad as loading the whole in-flight table but it meant that restarting the Eagle Eye service took some time.

Secondly, Cassandra can only distribute queries across the cluster if they are on the partition key, and the queries that can be made on the partition key are limited, so Eagle Eye had to write many many copies of the in-flight table using different keys for efficient queries to be possible.

Finally, the similarity of CQL to SQL is deceptive. CQL is an extremely limited query language, less capable in many ways than the simple filter model of Speedtables. Some queries, for example to locate nearby flights, needed to be sharded into many CQL queries fired off in parallel. So performance was never good.

Eagle Eye was never brought into production.

Popeye

Popeye is the second reimplementation of Birdseye. Popeye replaces Speedtables with SQLite, and reads controlstream using a C++ extension instead of parsing and manipulating the data in Tcl.

This is much more practical: SQL is a much richer query language than Speedtables, so implementing queries designed for Speedtables in SQL is a simple matter of programming. Also, SQLite is non-volatile, and SQLite's read latency is significantly higher than Speedtables, but not so high that simply caching data locally in C++ maps is practical and there's no need to completely reload state at startup.

The C++ library for reading controlstream that FlightAware uses is extremely efficient: it reads a block at a time and, splits the buffer into lines and fields without any further copying, storing the data as string_views. This allows the reader to run three to eight times faster than real-time even at busy times of the day.

The package we use for writing C++ extensions is CPPTCL. This was originally written by Maciej Sobczak but apparently abandoned in 2006. FlightAware has updated this project and is

maintaining a fork at https://github.com/flightaware/cpptcl. CPPTCL uses the C++ type system to map Tcl arguments and return values into C++ arguments and return values, provide access to Tcl objects and other structures from C++, and makes it easy to implement Tcl extensions in C++. For example, consider a C++ function:

```
int sum(int a, int b)
{
     return a + b;
}
```

This can be turned into a Tcl command with nothing more than:

```
i.def("sum", sum);
```

Popeye is event driven. It uses the Tcl file event interface to feed controlstream data to the controlstream parser, while running background maintenance tasks (flushing old flights, archiving track data, etc) on timers. When data is ready for reading from controlstream, the Tcl file handle is passed to "popeyedb filevent" - which reads events from controlstream and stores it in SQLite - until it runs out of new data.

Rather than modify the webservers and other client code, Popeye retains the trackstream interface, and maps the speedtables-like trackstream queries into SQL.

This process is similar to the Speedtables "STAPI" interface described in https://www.tcl.tk/community/tcl2016/assets/talk47/speedtables-paper.pdf.

Popeye doesn't use STAPI itself, since that's far more general than Popeye requires, and the trackstream query and response format isn't as regular as speedtables due to the way Birdseye evolved, but the experience made it much easier to generate efficient SQL.

Request latency can be a little higher than Birdseye but is fast enough. Once we have more experience with Popeye, we may upgrade the Trackstream interface to use SQL (directly or slightly modified) rather than speedtables-style queries.