Tcl at FlightAware

I've read that when first creating Facebook, Mark Zuckerberg used PHP, and that Facebook is still heavily PHP (although they built a fancy compiler for it.)

I reckon his motivations where expediency and familiarity. He was in a hurry, PHP was "there", he was familiar with it, and he was able to get up quickly.

I'm not trying to compare us to Zuckerberg, just to say I understand being in a hurry.

When we started FlightAware in 2005, we wanted to get going as quickly and with as little work as possible. We knew we wanted to use a scripting language, that a lot of the work we would be doing would be web-centric, and that we wanted to heavily leverage SQL.

Daniel Baker and I had already been using FreeBSD, Apache, Tcl and Postgres at our prior company, for server-side web/SQL development, and we were pretty happy with it, so it was a simple decision to take all that technology and experience and use it as our starting point.

Everyone's heard of the LAMP stack consisting of Linux, Apache, MySQL and PHP (or Perl or Python)

We called what we were doing the FATP stack of FreeBSD, Apache, Tcl and PostgreSQL.

For us, it was expedient. It was familiar. Wicked familiar. Deeply familiar.

We already had user management. We could make HTML tables from SQL query results. We were pretty confident that it could scale.

And we knew it was solid.

You know… correct programs that produce incorrect results due to bugs in lower layers of the system (such as the implementation of the interpreter) are schedule (and morale) killers. This is one of the hidden costs of trying to jump on the latest technology.

We were sold on SQL, and specifically PostgreSQL. We were not fans of MySQL. And buying an Oracle license was (and still is) financially out of the question.

We were sold, at that time, anyway, on FreeBSD.

We were firmly in the UNIX-style open source camp.

We wanted to license as little software as possible.

We wanted to have the source code to everything.

We wanted a flexible architecture where besides making webpages we could write event driven or declarative programs.

We weren't fans of Perl nor of PHP. In particular I didn't like the look of server-side PHP programs.

So Tcl it was.

Once we got a feed from the FAA, we used Tcl to parse it and began trying to make sense of it and writing stuff about flights into the database.

And it worked.

Let's go back to 2005

One of the first things we got going was this Tcl program that would connect and read the feed from the FAA and store it to disk in its native format.

Another Tcl program would read those files and decode and interpret the messages in them, and update the database and whatnot.

Within weeks, working on nights and weekends, webpage code written in Rivet, which is just HTML with embedded Tcl, accessed the database and produced webpages about flights, historical flights, and airport activity. While the handful of incumbents could show you only the last position of an aircraft, we could show the entire track. While they could tell you about the last flight, we could tell you about every flight we had seen. It was better than anything else out there.

We were doing a secret open beta, where we told people hey here's this thing, flightaware.com, check it out and let us know what you think, but don't tell anyone about it.

Well someone forgot to tell someone not to tell anyone and one morning we got up to discover that we had had several hundred users register overnight — someone had posted about FlightAware to a pilot's mailing list at Microsoft. This was fortuitous. Since the site had suddenly become public and was getting a lot of attention, it gave us a sense of urgency to improve it as quickly as possible.

The response was amazing. We went to the big annual aviation show and we were mobbed. It energized us even more to build out FlightAware as rapidly as we could.

Also we knew our stuff kind of sucked, so we wanted to fix that. (SLIDES)

While all this was going on we were still trying to understand flight tracking.

There were a lot of mistakes and errors in the feed. After all, a lot of it just comes from air traffic controllers typing stuff. There also wasn't any kind of unique ID for a flight. This was problematic, it turned out, because there were flights flying around with the same ident such as AIR1 and EGF104 and even UAL5.

The original program was naive and not stateful enough, so we wrote a new one.

The feed interpreter would read the feed and keep track of flights and match flights to flightplans, assign a unique ID to each flight, and normalize them, making every flight have a departure message, even if one wasn't received, and eventually arrive, at least in some manner.

We created a format for storing data called daystream. It stored one file per day with a directory structure of years and months above it. An Itcl class wrote daystream files, putting "switch" lines at the end of the day to direct the reader to the next file, all handled by the writer object without the writer's awareness or intervention.

Another Itcl class could read the daystream files, making them available to the program through callbacks, finding the files locally or streaming them remotely and automatically switching days without the reader having to know about it or do anything.

This has been and continues to be the backbone of how we read and write and chain the processing of flight tracking data.

We began looking at integrating additional datafeeds. Some use ICAO standard formats. Some use IATA. The first thing was always to write the data to nonvolatile storage in as close to the native format as possible. By "as close to" I mean that we might in some cases have to write framing information around the data, but we always endeavor to keep the data in

native format, as sometimes we discover that after importing and translating the data, bugs or misunderstandings on our part of the format would have led to a loss of fidelity had we not kept the original.

Next we make a translation pass to conform the data to standard formats of key-value pairs.

Most feeds are received by Tcl programs, although with a few of the newer feeds we are using Java for JMS compatibility with the sender.

Next a Tcl program, the combiner, subscribed to the feeds and ordered them roughly into clock order, and produced a single combined feed.

The feed interpreter read that feed and, guess what, interpreted it.

As message rates, number of flights, and number of data sources grew, and our own ADS-B network as well, the single-threaded feed interpreter could no longer keep up.

We not only needed to make it faster, we needed to make it a lot faster, to provide headroom for reasonably quick catchups in the event of an outage and to provide additional capacity for more messages and heavier computation per message.

We thought pretty hard about how to partition the data, like by geography is a natural one, but it had problems.

We experimented with cassandra and found out it wasn't flexible enough.

We settled on using shared storage between multiple child processes through Postgres, and called it hyperfeed. This was created and maintained by our flight tracking team, all of whom are here at the conference. Zach Conn presented on hyperfeed at last year's conference.

This is all Tcl.

When message rates began to approach the capacity of the combiner, we looked at various ways to make it go faster. In the end the flight tracking team created hcombiner, the feed combiner implemented in the Haskell functional programming language.

The output of hyperfeed is the controlstream feed, which is the normalized source of data that all the other applications read.

These include updating the database and track database, making "streaming" firehose feeds available to customers, realtime queries available through popeye, processing the data to generate alerts on things like aircraft departure and arrival, a certain number of minutes out, etc.

On the web side, it's almost all Tcl. Tcl code is the interface between the general database servers, the realtime popeye database servers, the tracks database servers, and the webservers.

We have a FlightXML service to allow querying of realtime and historical data and we have Firehose feeds which are realtime (and historical) streaming feeds.

Even most of our stored procedures in the database are written in PL/Tcl.

It's been handy to have commit for Pgtcl, by the way. We've also managed to get improvements to modernize PL/Tcl and extend its capabilities pushed back into PostgreSQL.

I think Tcl had stood up well to these challenges. And only a handful of times have bugs in the Tcl implementation caused problems.

A recurrent pattern, as the data rates and contractural playback rates have gotten up to these very high levels, is the need to heroically speed up Tcl

programs or rewrite them or parts of them in faster languages such as C++.

In some cases algorithmic improvements and recoding some critical paths as Tcl extensions written in C++ has been very successful. Our "multicom" alerts system went from slamming a dozen high-end 24-core Xeon machines to lightly loading just two.

In other cases there was no way algorithmic improvements alone could solve the problem. Due to performance demands from an important customer, we need to retrieve, examine and forward subsets of message sets at an input rate of hundreds of thousands of messages/sec. This was evolved from pure Tcl, to Tcl with C++, to Tcl with a lot of C++, and currently to where Tcl is used to set things up and orchestrate the high level activities of the program, but the fetching, decompressing, filtering and forwarding is all C++. And the C++ is multithreaded and uses somewhat exotic classes and makes serious efforts to avoid allocating and copying memory, etc. In other words, even in C++ considerable work was done on performance.

On the webservers, we have 710 packages loaded into Rivet, defining over 4500 procs.

When we needed to do XML, we tried some Tcl packages, found one we liked, and started using it. That's kind of been the approach. If there isn't anything or we're not happy with what's available, we'll write something ourselves, and usually open source it.

Since so often we're needing to speed up programs, we started looking at ways of extending Tcl that are high performance but less difficult to work with than C, even though we're good at that.

We wanted to be able to write in a high-performance, object-oriented systems programming language and be able to fluidly call our huge body of Tcl code and be called by it.

Swift is a really cool-looking language, very modern and expressive. Colloquially, it feels like a scripting language.

We did a good bit of work on SwiftTcl, to bridge between Swift and Tcl, but we ran into problems: Swift libraries for interacting with the operating system server-side were minimal, and there weren't a lot of class libraries for even relatively simple data structures and whatnot. Most of the attention that swift is receiving, including from its developers, is to create apps for iPhones and iPads.

Also Swift was a rapidly moving target, and FreeBSD support was sorely lacking.

It had some tantalizing aspects. Calling Tcl from Swift and calling Swift from Tcl was quite transparent, like Swift programs didn't really know they were calling Tcl and vice versa.

Even though the aforementioned problems basically killed it for us, it was promising enough that we kept looking. Then Shannon Noe found cpptcl, a C++ library for interoperability between C++ and Tcl, that had been written by Maciej Sobczak in the mid-2000s, and revived it. This is super important technology for us and Shannon's going to be presenting about it here at the conference.

Suffice for me to say that Tcl developers can use C++ to create high performance code that leverages all of the C++ ecosystem while providing dead simple ways to bring their C++ functions into Tcl, with expected Tcl semantics, and no funny business. Writing C++ extensions for Tcl using cpptcl is vastly easier and requires far less code than extending Tcl in C.

cpptcl is everything I'd hoped SwiftTcl would be, except C++ is a little bit clunkier than Swift.

Nonetheless, we have a good bit of in-house C++ experience and we're putting it to good use.

we push software out to servers with Tcl, among other things

we monitor applications with Tcl, among other things

flight feeders and piaware

Something big impacting our industry is the widespread adoption of ADS-B, Airborne Dependent Surveillance, Broadcast. A new kind of transponder that doesn't require radar, where aircraft know where they are using GPS and broadcast their position digitally by radio at 1090 MHz.

It's used for both air-to-air communication, such as the traffic collision avoidance system, TCAS, and air-to-ground and ground-to-air for air traffic controllers to be able to see and monitor and control aircraft.

We've created a succession of ADS-B receivers, and have shrunk them down to something really tiny, Raspberry Pi based, and fairly inexpensive. FlightFeeders we make ourselves and have made for us, and Piaware people build their own receivers. At this time we have nearly 20,000 active receivers world-wide.

We distribute the software as packages and as a complete bootable disk image that includes the Raspbian port of the Debian Linux operating system.

A program called dump1090, written by Salvatore Sanfilippo, a Tcl aficionado and author of Redis and, since then, extensively worked on by

Oliver Jowett and now, Eric Tran, both of FlightAware, decodes the ADS-B messages and makes them available on various TCP ports.

Tcl code backhauls the data to FlightAware, handling logging in, encrypting using TLS, supporting automatic and manual updates of the application, and even the operating system, providing a webserver to view aircraft activity called Skyview, and more.

On the FlightFeeders we have touchscreens for configuration management, diagnostics and status, and that's all done using Tk.

(SLIDE)

Today FlightAware is still principally Tcl, with about thirty developers, plus QA, plus data analysts, and others, using it and other languages to develop, maintain and extend our products.

But we are not a single language shop. We have code in C, C++, Swift, Python, Java, Scala, and Haskell, and I'm sure I'm leaving something out.

By the numbers we have…

About 165,000 lines of code in our big set of FlightAware-specific packages.

123,000 lines of Rivet code.

145,000 lines of flight tracking

probably a few tens of thousands of lines in other packages

5000 in popeye

4000 in global beacon

9000 in multicom

The site has over 10M monthly users; it's a top 400 site by users, a top 100 site by page views

My colleague Anne-Leslie Dean will be presenting on web application development using Rivet, Tcl and JavaScript at this conference. Garret McGrath on building a fault-tolerant distributed system with zookeepertcl, Jonathan Cone on debugging Tcl using visual studio code, Peter da Silva just presented on Popeye, and Shannon Noe will present on cpptcl. I hope I haven't missed anyone.

I hope this gives you a sense of the depth and breadth of our use of Tcl at FlightAware. Thank you.

Questions?