`Hyperfeed`: FlightAware's parallel flight tracking engine

Zach Conn

Lead Software Developer, FlightAware

Abstract

This paper discusses `hyperfeed`, FlightAware's core flight data processing engine written in Tcl. Developed incrementally over the course of a decade, `hyperfeed` is responsible for ingesting all of FlightAware's 40+ data feeds, aggregating the data together, resolving inconsistencies, filling in gaps, detecting and filtering out unreliable or bad data, and producing a single data feed that represents an all-encompassing coherent view of worldwide flight traffic as understood by FlightAware. These results are visible to millions of users through the website, mobile apps, and various APIs.

`Hyperfeed` is a high-performance, highly parallel and concurrent system that can easily saturate 32+ cores of a modern high-end machine, and it's all written in Tcl. It can run for weeks or months at a time and is only restarted to pick up software updates. It consists of a dispatcher process which routes incoming messages to child interpreters, which do work in parallel while sharing data and communicating state when necessary through a centralized PostgreSQL data store. Read-committed and linearizable/serializable transactional semantics from PostgreSQL are relied upon to guarantee correctness under contention.

1. Introduction

Founded in 2005, FlightAware was the world's first major flight tracking company, and it remains the largest and most successful to this day. The company now processes 40+ data feeds, ingesting 150 GB of data per day consisting of 50-100 million messages. FlightAware is particularly interesting to the Tcl community because virtually all of FlightAware's software is written in Tcl, from the website to the most demanding distributed backend applications.

This paper explores the history of one particular piece of backend infrastructure at FlightAware. We call it `hyperfeed`, but it's gone by a few different names in the past, including `feed interpreter`. Beyond being interesting in and of itself, `hyperfeed` has had a unique history of interaction with the Tcl programming language, and it's one of the most mission-critical pieces of production infrastructure anywhere using `speedtables`, a high-speed in-memory database exclusively for Tcl.

The present parallel implementation still makes several interesting uses of Tcl:

Email address: `zachery.conn@flightaware.com` (Zach Conn)

- Data that is not potentially in contention from multiple processes, doesn't need transactional semantics, and which wouldn't benefit from post-hoc analysis is stored in speedtables.

- Tcl's event loop allows us to easily asynchronously defer processing of messages until we've learned more information over time. We've built a virtual sequencer on top of this to facilitate the use of virtual clocks in scheduling and sequencing commands; this allows us to replay historical scenarios easily.

- During a major architectural rewrite in 2015, when we redesigned `hyperfeed` from a single-threaded program to a parallelized system, we needed to introduce the use of PostgreSQL as a replacement for some of the speedtables used by the older architecture. Tcl's dynamic and introspective nature allowed us to do this with minimal rewriting of code by dynamically redefining procs and introducing a query translation layer.

- We have often performed software updates with zero downtime and no restarts thanks to hot code reloading, again made possible by Tcl's highly dynamic nature.

2. The problem space

`hyperfeed` receives data from over 40 different feeds, including data from air traffic control systems in over 55 countries, from FlightAware's network of ADS-B ground stations in over 140 countries, from Aireon space-based global ADS-B, and using global datalink (satellite/VHF) via every major provider, including ARINC, SITA, Satcom Direct, Garmin, Honeywell GDC, and UVdatalink.

From a very high level perspective, we can place messages from all these feeds into a few categories:

- **Flightplan messages** at a bare minimum tell us the origin, destination, scheduled departure time, and scheduled arrival time of a particular flight. These messages come in a huge variety of formats from different sources, and many of them include far more information than just this, but they all fundamentally just tell us that a particular flight is planning on leaving one airport at a certain time and landing at another airport at another time.

  Note that for a given flight we might receive multiple flightplan messages which revise information about the flight as updates become known to our various sources.

- **Position messages** tell us that a plane was at a particular location at a specified timestamp.

- **Event messages** tell us that a special event has occurred for a flight.

  - **Departure messages** tell us that a plane has taken off from its origin airport.

  - **Arrival messages** tell us that a plane has landed at its destination airport.
  - **Offblock messages** tell us that a plane has pushed back from its gate.
  - **Onblock messages** tell us that a plane has arrived at its gate at its destination airport after landing.

From this, it's easy to conceptualize how we try to track a flight. Ideally we receive a flightplan message in advance of the flight (the sooner the better, so that we can display the flight on the website). Between the initial flightplan message and departure, we'll receive revised flightplan messages that provide updated departure and arrival times (amongst other data). On departure, we'll receive a departure message. Then we'll receive a sequence of positions. And finally we'll receive an arrival message.

Unfortunately, it doesn't always happen like this, and that's why `hyperfeed` exists. Here are just a few ways in which the above picture can break down:

- Nobody sends us the flightplan until the flight is already in the air.
- Nobody ever sends us the flightplan at all, but the flight does fly.
- We receive a flightplan, but the information it contains is inaccurate. For instance:
  - The departure time might be off by an hour or more.
  - The flight might be delayed.
  - The callsign for the flight might change.
  - The flight may never fly at all despite us having received a flightplan message for it.
  - The flightplan data might just be flat-out wrong (e.g., wrong origin or destination). It might have even been filed on the wrong day entirely.
- A flight departs, but we don't receive a departure message.
- We receive a departure message, but the plane didn't actually depart.
- A flight lands, but we don't receive an arrival message.
- We receive an arrival message, but the plane is still in the air.
- The flight diverts to a different destination airport and nobody tell us this.
- We're told that a flight has diverted, but it didn't.
- We receive positions with the wrong callsign, making it appear that these positions belong to one flight when they actually belong to another.
- We receive positions that indicate the plane is in the air, but it's actually on the ground.

- We receive positions that indicate the plane is on the ground, but it's actually in the air.

In short, Murphy's law applies: if something can go wrong, it will go wrong. `Hyperfeed` exists to make sure the flight data that we stitch together actually presents a coherent and sensible picture of air traffic activity around the world. It synthetically detects arrivals, departures, and diversions even if nobody tells us about these events. It can create adhoc flightplans for flights that we receive positions for but never received a flightplan for. It can remember flightplans for days so that when it sees a departure three days later it knows which flight it's for.

In general it's skeptical of every message it sees. It might see a cancellation message but determine that the cancellation is most likely false and ignore it. It might see a departure message but determine that the departure is most likely premature and hold onto the information in-memory for several minutes waiting for a second source to corroborate the departure.

In addition to the above, there's the problem of conflicting data privileges. Not all of our customers have access to all of our data feeds. A particular customer should only be able to see a version of a flight which could be deduced and reconstructed from the data sources that they are allowed to see. `hyperfeed` thus maintains multiple versions of every flight it tracks, specifically one version for every unordered combination of distinct privilege classes seen among the sources that contributed data to the flight. It's clear that in the worst case this leads to exponential growth in the number of flights maintained in-memory as the number of privilege classes grows.

In summary, `hyperfeed` is a hand-crafted AI that uses fuzzy logic instead of machine learning in order to make intelligent decisions about input data. It's the PageRank of FlightAware.

3. A bit of history

Because `hyperfeed` has to keep flightplan information in-memory for days at a time, it relies heavily on having a robust and queryable data store. The current implementation uses PostgreSQL for this purpose, which provides exceptional support for concurrency as well as durability and the extremely flexible query semantics of SQL. However, for most of its history `hyperfeed` was single-threaded and known as the `feed interpreter (FI)` instead. `FI` used speedtables exclusively for its in-memory data store.

Speedtabless provides a high-performance memory-resident database implemented as a C extension to Tcl. Unlike, say, redis, speedtables has full support for secondary indexes and more complex queries beyond just simple key-value lookups. This made it extremely well-suited to the problem of storing and querying hundreds of thousands of flightplans and millions of positions in memory from Tcl.

Here's an example of how a flightplan search might have been performed using speedtables:

```
set compList [list [list = ident $data(ident)] [list = orig $orig] \
                   [list = dest $dest] [list null child]]
flightplans search -compare $compList -key id -array flightplan -code {...}
```

This is very similar to a SELECT query in SQL; it's searching for all known flightplans between a given origin and destination and which have a specified callsign. (The "`null child`" condition is very FA-specific.)

`FI` was one of the biggest production uses of speedtables and fully utilized its performance and support for secondary indexes and flexible queries. Unfortunately, due to issues that will be discussed in more detail below, speedtables didn't have adequate support for concurrency, so when we parallelized `FI` we ended up replacing it with PostgreSQL. However, speedtables are still used in `hyperfeed` to store static data sets (e.g., airport elevation data), since speedtable queries are still faster than PostgreSQL queries.

As an interesting aside, the speedtables API has a query translation layer. This layer can translate speedtable queries into SQL queries, which has several uses:

- If a speedtable is being used as a high-speed cache for a PostgreSQL table, the queries can fall back to SQL if needed for some reason.
- In the case of the transition from `FI` to `hyperfeed`, we used this translation layer to avoid the need to rewrite all of the speedtable queries in `FI`. This was a huge win in terms of minimizing the amount of code that needed to be rewritten.

This translation layer is open source and available in the public distribution of speedtables; it's a core part of `STAPI`, the speedtables API.

As an example, consider the following speedtable query written in pure Tcl:

```
flightplans search -compare [list [list = ident UAL5]] \
                   -array matchedPlan \
                   -code { ... }
```

The speedtables API (`STAPI`) will optionally convert this into a simple SELECT statement:

```
SELECT * FROM production.flightplans WHERE ident = 'UAL5';
```

There are advantages to writing the query in pure Tcl rather than SQL. For instance, Tcl data structures such as lists are easier and cleaner to dynamically generate than strings, and the underlying translation layer can also handle argument quoting for preventing injection attacks.

4. The structure of `hyperfeed`

`hyperfeed` consists of a variety of components. Each of these will be examined individually:

- The **dispatcher** is responsible for routing incoming messages to appropriate child interpreters.
- There are around 100 **child interpreters** which receive messages to process from the dispatcher. These child interpreters connect to the `hyperfeed` database and attempt to make sense of each message.
- The **projector** is a separate process that concurrently looks for flights that are airborne but haven't received real positions recently. For these flights it estimates or "projects" where they should be at the current time based on their filed route.
- The **housekeeper** is a separate process that concurrently looks for flight data that we no longer need to maintain and deletes it from the relevant database tables.
- The **controlstream writer** is a separate process that each child interpreter connects to and sends output messages to. `controlstream` is our name for the output data feed that `hyperfeed` produces. The writer process is responsible for synchronizing writes to the feed from all the child interpreters.
- The **virtual sequencer** is an internal event loop, inspired by the Tcl event loop, which allows us to schedule events to occur at points in *virtual time*. This makes historical replays very easy.
- The **database** acts as a centralized transactional data store of flightplans, positions, and other data that must potentially be operated on concurrently by child interpreters.

4.1. The dispatcher

The design we eventually settled on for the dispatcher is very simple, but in the process of getting there we considered several much more complex algorithms. Ultimately it was the use of PostgreSQL that allowed us to simplify the dispatcher so drastically. What follows is an examination of how the problem of dispatching messages can potentially be quite difficult.

A special case of the problem can be isolated and understood. Flights often have *two* possible identifiers: the callsign and the tail number. Sources do not consistently identify a flight by one or both of these; some will use the callsign, some will use the tail number, and some will use both. If we wish to dispatch messages to children in such a way that messages for the same flight always go to the same child, we already run into difficulties.

Define the **signature** of an input message to be the tuple (c, r) where c is the callsign and r is the registration, where either or both may be the empty string. Consider the following sequence of events:

- A message with signature (c_1, r_1) is received. c_1 and r_1 are both assigned to child h_1.

- A message with signature $(c_2, \text{“”})$ is received. c_2 is assigned to a different child, say h_2.
- A message with signature (c_2, r_1) is received. A conflict is encountered between children h_1 and h_2. It's unclear which child this message should be sent to.

Suppose a mechanism exists which allows us to completely copy a flight and all its associated data (positions, etc.) from one child to another. Even if we ignore the fact that such an operation is likely to be prohibitively expensive if it must be frequently performed for many flights per second, we will still run into issues:

- We move all the flight data for any flights with tail number r_1 from child h_1 to child h_2.
- It's now clear that we should send the message with signature (c_2, r_1) to child h_2.
- A message with signature (c_1, r_1) is received. We move back any flight with tail number r_1 from h_2 to h_1.
- In the worst case, an oscillatory pattern results.

The problem is that transferring flights associated just with registration r_1 from one child to another is not enough. We have to account for a network effect, whereby any flight with a callsign that has been paired with r_1 must also be transferred, and then any flight that has a tail number that has been paired with a callsign that has been paired with r_1 must be moved, and so on. It's clear that this amounts to a standard breadth-first traversal of a certain abstract graph.

The graph G can be constructed as follows:

- For each distinct callsign c, create a new node with the label $(c, 0)$.
- For each distinct tail number/registration r, create a new node with the label $(r, 1)$.
- Introduce edges into G as follows: if there exists a message with signature (c, r), add an edge between node $(c, 0)$ and node $(r, 1)$.

If we decide that every flight associated with a particular registration r_1 needs to be moved to child h_1, then we also need to relocate all flights discovered via the following recursive procedure:

- Transfer all flights with a callsign c that occurs in a node of the form $(c, 0)$ with an edge connecting it to node $(r_1, 1)$.
- For each such callsign c, transfer all flights with a registration r' that occurs in a node of the form $(r', 1)$ with an edge connecting it to the node $(c, 0)$.
- For each such registration r', transfer all flights with a callsign c' that occurs in a node of the form $(c', 0)$ with an edge connecting it to the node $(r', 1)$.

- Rinse and repeat.

This is evidently nothing other than a modified breadth-first search. In fact, it can be visualized quite nicely as assigning child IDs to connected components of G.

We were immediately skeptical of introducing this level of complexity into the dispatcher. In particular, this would mean that reconstructing historical replays would be very delicate indeed, but this is critical to debugging flight tracking errors. All of this could be avoided if only different children could concurrently operate on the same flight data, since this would mean we wouldn't need to be so rigorous about cleanly separating flights across children. Among other considerations, this is ultimately what led us to use PostgreSQL as the data store.

With the transactional semantics of PostgreSQL, the dispatcher becomes much simpler:

- In general, messages are sent to children in simple round-robin order.
- If we receive a message with signature (c, r) and we've recently sent a message with the same signature to a particular child, then we violate the round-robin order and send the current message to the same child. We call this "**forgetful affinity**": each signature has an affinity for a certain child, but the affinity expires after a short period of time.
- This round-robin forgetful affinity dispatcher can be implemented easily and efficiently using a simple time-to-live hash map. For extra efficiency, we implemented the TTL map using a speedtable.

4.1.1. Child recycling

The dispatcher is also responsible for periodically recycling the child interpreter processes (which, as discussed below, are created as copies of the dispatcher process via the `fork` system call, exposed through TclX).

Periodically recycling child interperters has a few advantages, including refreshing the prepared statements which each interpreter uses. The problem of recycling forked copies of the dispatcher is an interesting case study of how to delicately use the Tcl event loop.

A `recycle_next_child` proc is scheduled to execute periodically in time. The following is a simplified version of how this is done:

```
proc periodically_recycle_children {} {
    set dt [expr {1000 * $::hyperfeed::childRestartInterval}]
    set ::hyperfeed::nextRestartEventPID \
        [after $dt [list ::hyperfeed::periodically_recycle_children]]
    ::hyperfeed::recycle_next_child
}
```

`recycle_next_bucket` determines the ID of the child interpreter to recycle next and then invokes `recycle_bucket`, which handles the dirty work of cleanly shutting down the child

interpreter and creating a new one via TclX's `fork` proc. The delicacy arises from the fact that the newly forked child process will inherit the parent's event loop, including the self-rescheduling proc `periodically_recycle_children`. Tcl's `after cancel` command makes this situation easy to remedy.

4.2. The child interpreters

Each child interpreter is created initially as a copy of the dispatcher process, using the `fork` system call provided by TclX.

We chose to create children via `fork` because of the copy-on-write semantics of Unix process creation. Each child interpreter loads a fair amount of static data on startup into speedtables. This quickly becomes a bottleneck during startup if performed by 100+ children simultaneously. Staggering the children in time makes startup even slower. But simply doing it once in a parent process and then creating children via `fork` makes startup very fast and clean.

In particular, the difference in startup time between `hyperfeed` with 100 children and the previous single-threaded speedtables-backed `FI` is staggering. It takes a few seconds at most to launch the dispatcher and all 100 children and for each process to establish a connection to the `hyperfeed` database. `FI`, on the other hand, would have to load all its speedtables in-memory from a checkpoint written to disk, which in the worst case could take up to a few minutes. Of course, we could have used client-server speedtables, but this still didn't offer the concurrency properties we needed nor the rock-solid durability of PostgreSQL.

4.2.1. The structure of a child interpreter

Each child interpreter passes incoming messages through a simple processing pipeline:

- **Parsing and normalization.** The interpreter tries to normalize different messages so that they can assume only one of a few forms. For instance, it tries to convert any position messages into its internal format for positions, and likewise for flightplan updates, departures, arrivals, diversions, and more.

- **Flightplan matching.** Once normalized, an input message must be matched against the correct flightplan held in the database by `hyperfeed`. There may be no valid match, in which case a new flightplan will be created.

- **Flightplan forking.** Inspired by the system call of the same name, `hyperfeed` creates a **fork** of a flightplan if the current message has a privilege class that has not yet contributed data to the flightplan. A flightplan fork can be thought of as a version of the flight which contains only data from a particular subset of the set of all privilege classes that have contributed data to the flight.

- **Status updates.** `hyperfeed` must deduce how the state of the flight changes based on information from the message and the flightplan it matched against.

- **Output message generation.** Finally, of course `hyperfeed` must prepare messages to be sent downstream to describe the new state of the flight and any updates that might have occurred, including the creation of new forks of the flight.

Flightplan matching and forking are particularly interesting and deserve more explanation.

Flightplan matching. Flightplan matching consists of two steps:

- Given an input message, query `hyperfeed`'s internal state (stored in the database) to determine a set of potential candidate flightplan matches.
- For each candidate flightplan, judge the likelihood that the input message corresponds to the candidate flightplan.

The first step was previously done using speedtables queries, and now it's done using SQL queries.

The second step uses hand-crafted fuzzy logic to compute the probability that an input message matches against a candidate flightplan. This logic has been tweaked repeatedly over the years and contains within it a lot of the domain-specific knowledge that FlightAware has accumulated over its lifetime.

If no match probability is sufficiently high, the match will be declared a failure, and a new flightplan will be created and stored in the database.

Flightplan fork creation. After a message has successfully matched against a flightplan, the child interpreter compares the privilege class of the new message against all forks of the flightplan. To each existing fork there corresponds a *subset* of the set of all possible privilege classes. If the privilege class of the new message does not appear among any of these subsets, then the child interpreter knows that new forks must be created.

The number of forks that must be created is such that at the end of the process `hyperfeed` should have one fork of this flight for every possible unordered combination of privilege classes that have contributed data to the flight. Note that there is not a one-to-one correspondence between privilege classes and data sources; many data sources have the same privilege class.

4.2.2. Transactional message processing

One of the more interesting features offered to us by PostgreSQL over alternatives like speedtables is the ability wrap multiple queries inside a transaction. In the case of `hyperfeed`, each child interpreter has an internal **transacion retry loop**.

This loop functions as follows. When the interperter first begins processing a message, it opens a new transaction. When it's done with the message, it attempts to commit the transaction. If the commit fails, it can retry the entire message, creating a new transaction and attempting to commit it. This continues until either the commit succeeds or the retry limit is exhausted.

There are various delicate details involved in making this work correctly. For instance, output messages must be queued instead of emitted directly, and the queue must be managed correctly on message failure or retry.

`hyperfeed` also supports nested transactions, or subtransactions, by using PostgreSQL savepoints. A nested transaction is such that it can be rolled back without rolling back the enclosing transaction. As an example, the process of flightplan fork creation is wrapped in a nested transaction, so that failure to create a fork does not necessarily cause the entire message transaction to fail.

This transaction retry loop means that `hyperfeed` can naturally make use of a serializable or linearizable isolation level for its transactions. When two concurrent transactions are run with a serializable isolation level, PostgreSQL guarantees that only results will be obtained which could be obtained by choosing some ordering of the concurrent transactions and processing them serially. If results are obtained that could not be obtained serially, PostgreSQL will force one of the transactions to fail with a serialization error, and `hyperfeed` will be able to retry it.

4.3. The projector

In the modern incarnation of `hyperfeed`, the projector is a separate process which periodically runs and connects to the `hyperfeed` database.

The projector has a fairly simple but very important role. It queries the shared database to find flights that are known to be en route currently, but which haven't received any real positions recently. For each such flight, it attempts to estimate or "project" the plane's current location based on information about the flight (e.g., origin, departure, scheduled departure and arrival times, airspeed, and/or the filed flight route).

The projector is itself parallelized. After determining the set of flights that need to be projected, it creates various worker threads to project batches of these flights in parallel. This means that the projector can complete a run successfully in only one or two seconds.

4.4. The housekeeper

Like the projector, the housekeeper is a separate process that runs periodically and connects to the `hyperfeed` database. Its purpose is to identify flights that no longer need to be stored by `hyperfeed` and to delete them and all their associated data.

Here are some examples of criteria the housekeeper uses to decide if a flight qualifies for deletion:

- If a scheduled flightplan has received no activity for at least two hours after its filed departure time, the housekeeper will issue a synthetic cancellation.
- If a flight was seen to depart and has been projected (estimated) all the way to its destination, the housekeeper may issue a synthetic "flight result unknown" arrival for the flight.
- If a flight has a recent real position near its destination, the housekeeper may issue a synthetic arrival for the flight.

- If a flight has been arrived and in a state of completion for at least several hours, the housekeeper may simply delete it from the database. The flight data will live on downstream in other FlightAware services; `hyperfeed` in particular will not need to make use of the flightplan information anymore.

There are many other criteria that the housekeeper looks for, but these examples capture the spirit of how it works and what it does.

4.5. The virtual sequencer

One of the most unique features of Tcl is its built-in event loop. `hyperfeed` has its own event loop called the **virtual sequencer** which is heavily inspired by Tcl's own event loop.

The primary motivation of the virtual sequencer was the need to schedule events in *virtual time*. For instance, for debugging purposes we may want to run `hyperfeed` over a subset of historical data for just a particular callsign or tail number. Not only does this mean that this `hyperfeed` instance will be reading data from potentially months or years ago; it also means it will be progressing much faster than normal due to ingesting a much smaller and more restricted input data set. For instance, we might want to schedule an event to happen five minutes in the virtual future, but this might correspond to only a few milliseconds in the real future if `hyperfeed` is progressing very quickly.

The *virtual clock* is determined by the timestamps of the messages that a `hyperfeed` instance is processing. By being able to schedule events in virtual time instead of real time, high-speed historical replays follow as a trivial byproduct.

In fact, each child interpreter has its own virtual sequencer and virtual clock. However, they all share the same event schedule, which is stored in the `hyperfeed` database. This allows us to maintain the invariant that each child interpreter can pretend that it's the only one, even if it's just one among one hundred. For instance, child interpreter h_1 might schedule event A to happen at a particular point in the virtual future. Event A will be assigned specifically to child h_1, but another child, say child h_2, could cancel A or re-schedule it without ever knowing that A was meant to be executed by h_1, not h_2.

4.6. The database

We chose PostgreSQL for the database. At this point we've already touched on many of the reasons behind choosing a SQL database instead of an in-memory database like speedtables or redis:

- `hyperfeed` makes extensive use of secondary indexes and complex queries. This makes simpler in-memory databases like redis or memcached much less ideal candidates. Speedtables has fantastic support in this category.

- We need world-class support for concurrency. While a single-threaded redis server can in some cases be so fast as to offer a convincing illusion of parallelism, redis transactions are just atomic batches of commands. The same is true for speedtables. More precisely, there is no notion of an "open" transaction in these systems. With

PostgreSQL, a child interpreter can open a transaction, execute queries, make decisions based on the results of those queries, and still decide at the end whether to commit or rollback the transaction. Accomplishing something similar with redis or speedtables would have required a considerable rewriting of the entire codebase.

- PostgreSQL offers fantastic durability, and the performance here is quite tunable.

As a note on performance tuning, we found we got the best results by keeping `fsync` on in PostgreSQL, but turning `synchronous_commit` off and setting the WAL writer delay to 10 seconds. In fact, this is the key tweak that made it viable for us to use PostgreSQL. We also make heavy use of prepared statements.

5. The role of Tcl in dynamic query translation

As mentioned, in 2015 we migrated from our old single-threaded system (known as the `feed interpreter`) to the parallelized version (known as `hyperfeed`), which from a high level simply runs 100+ instances of the old `FI` side by side. In making this transition, we wanted to minimize the amount of code that needed to be rewritten. Tcl turned out to aid in this endeavor quite a bit.

As an example, we used the query translation layer in the speedtables API to avoid having to rewrite all the speedtables queries. We achieved this by dynamically creating procs named after the old speedtables. For instance, we previously had a speedtable named `flighplans`. Now `flightplans` is a table in a SQL database and has no existence as a Tcl object. So we created it dynamically as a proc:

```
set ::pgInstances($instanceName) \
    [::stapi::connect sql:///${tableName}?_key=${keyName}]

# This rewrites a table name, like 'flightplans', into a proc
# which references the Postgres object ID, allowing existing
# speedtable queries to work automatically.
proc $instanceName {args} [format {
    return [uplevel $::pgInstances(%s) $args]
} $instanceName]
```

Thanks to the dynamic nature of Tcl, this is all it took to avoid rewriting all the queries in `hyperfeed`, which was incredibly powerful in bootstrapping the project and obtaining a working prototype.

6. Asyncrhonous console interfaces and hot code reloading

FlightAware has an open source `fa_console` package, which when embedded in a Tcl program allows one to create a console port which one can connect to and send commands

to the running program for execution. Both the older `feed interpreter` and the modern `hyperfeed` expose console interfaces via this mechanism.

The live console interfaces enabled in this way can be incredibly powerful, even dangerously so. For years we lived on the edge a bit. We routinely deployed small code updates to `FI` by connecting to the program's console port and issuing Tcl `source` commands. We've done this countless times and it never failed or malfunctioned once. This allowed us to roll out minor tweaks and updates with zero downtime and no outages. Keep in mind that a `hyperfeed` outage is an outage for basically the entire company.

We can still do this in the modern `hyperfeed`, except we can't just use the built-in `source` command due to the existence of multiple children. Instead we use a custom `inject` command which propagates the command to all the children.

In practice, we don't do this anymore because the architectural changes in going from `FI` to `hyperfeed` made startup and shutdown so fast that it's practical now to just restart `hyperfeed` completely even for very minor and small code changes.