Multicom – FlightAware's Alert Delivery System

Mary Ryan Gilmore
TCL Conference 2019

Introduction

Multicom is a high-performance alert delivery application written in TCL. The system reads in a stream of 35 million flight event messages per day (that's 400 updates per second on average), matches the event against more than 440,000 alert triggers, and sends more than 300,000 alerts each day.

As FlightAware gains access to more data sources, the number of events processed per day is only increasing. Sometimes the change is gradual and sometimes it is very sudden. Recently, FlightAware gained access to a new data source that gave us position messages all over the globe. This was exciting for the company, but detrimental to Multicom. With this new data feed, Multicom could no longer keep up the input of data in real-time. We had to quickly come up with a way to address the immediate shortfall, and we had to examine the system's weaknesses in order to make it more performant. In order to address the performance issue, we used a number of TCL libraries such as sqlite and sqlbird. With this upgrade, we saw a significant performance boost, and our code became much more readable. Since this catastrophe, we are constantly looking for more ways to increase reliability and performance in order to easily accommodate more events. As a result, we have successfully introduced even more TCL technology to Multicom such as tclrmq and zookeepertcl. This paper will review the basic design of the system, and how we have used many different TCL packages to improve performance and durability.

Initial System Design Summary

The main system that consumes event messages and matches them to users' alert triggers was run on 3 separate servers with 16 children on each server (for 48 children total). The flight event messages were hashed and equally distributed among all of the children. In the initial design, we used speedtables (an internal technology that we use to store data tables) for caching, and all children wrote to and read from the same cache on the server. We also used speedtables to replicate important database tables, such as the table that holds all of the alert triggers.

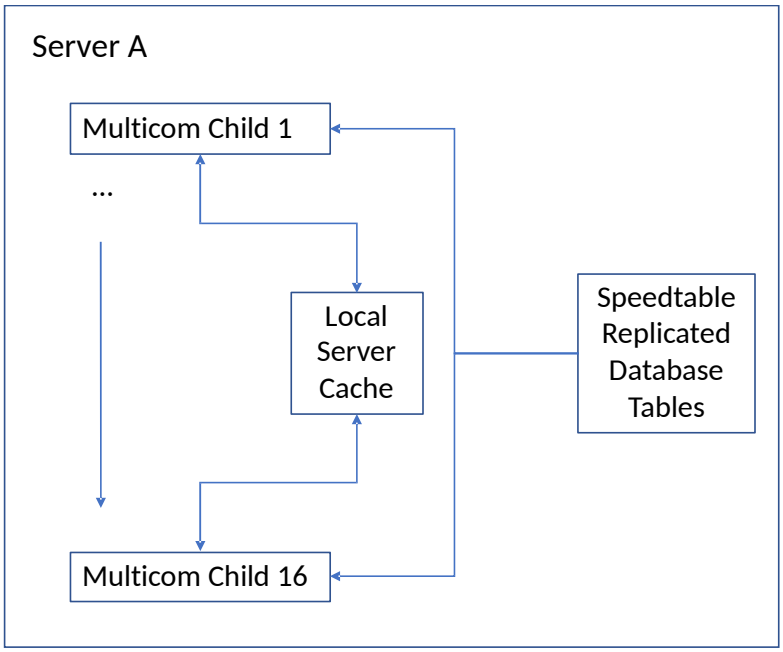

Figure A: old Multicom design with speedtable caches and database table replication

This design posed two major issues for us: 1) speedtables do not have an “OR” functionality like postgres, and 2) there was a lot of write contention on the speedtable cache because all 16 of the children on one server where attempting to write to it.

Response Following Sudden Event Data Increase

After we realized that introducing a new data source would debilitate the existing Mutlicom system, we needed a way to quickly resolve the issue so that we could have more time to address performance issues. We did this by throwing more servers at the problem. So instead of running 48 child processes across 3 servers, we started running 96 child processes across 6 servers. While this solved the original issue, deploying new code across 6 servers was cumbersome. In addition, we knew we had major issues that we needed to address to make the system faster and more reliable.

Introducing Sqlite and Sqlbird

The main issue that we encountered using speedtables was that there was no basic “OR” functionality. In order to get around this issue, we would have to run multiple speedtable queries. This created ugly code and the total processing time was very long.

```
foreach field {base_id ident reg origin destination aircrafttype} {
        set search_list [list [list match $field $data($field)] {true enabled}]
        $::st(mc_trigger_tracking) search -compare $search_list -array trigger -code {
        ...
        }
}
```
Code Snippet A: example of speedtable loop query to replicate “OR”

This is the reason why we started to explore using sqlite. Sqlite syntax is almost exactly like postgres, so it had the basic “OR” functionality that we were looking for, among many other useful syntax features. The language was also already widely used at FlightAware, because we have a postgres database server. We immediately started using sqlite for caching, and we reconfigured each child to write to and read from its own cache in order to eliminate the write contention.

We also wanted to use sqlite for database replication as well, so we turned to sqlbird, which was brand new at the time. Sqlbird is a sqlite database table and view replicator that allows us to store database information locally for quick access. Any process running on a server can access the replicated sqlbird tables in read-only mode, and sqlbird updates all of the tables/views in a configurable time interval.

Overall this design change and move to using sqlite was extremely successful for us. Along with some other small design changes, this enabled us to handle the new event data on only 2 servers with 12 children each (for 24 children total). This gave us a lot of room for future growth and allowed us to spend time addressing durability issues. It also significantly improved readability of our code.

```
set sql “SELECT * FROM mc_trigger_tracking WHERE ident = :data(ident) OR reg = :data(reg) OR origin
= :data(origin) OR destination = :data(destination) OR aircrafttpye = :data(aircrafttype)”

sqlbird::select $sql trigger {
        ...
}
```

Code Snippet B: example of sqlbird query (same operation as previous speedtable query loop)

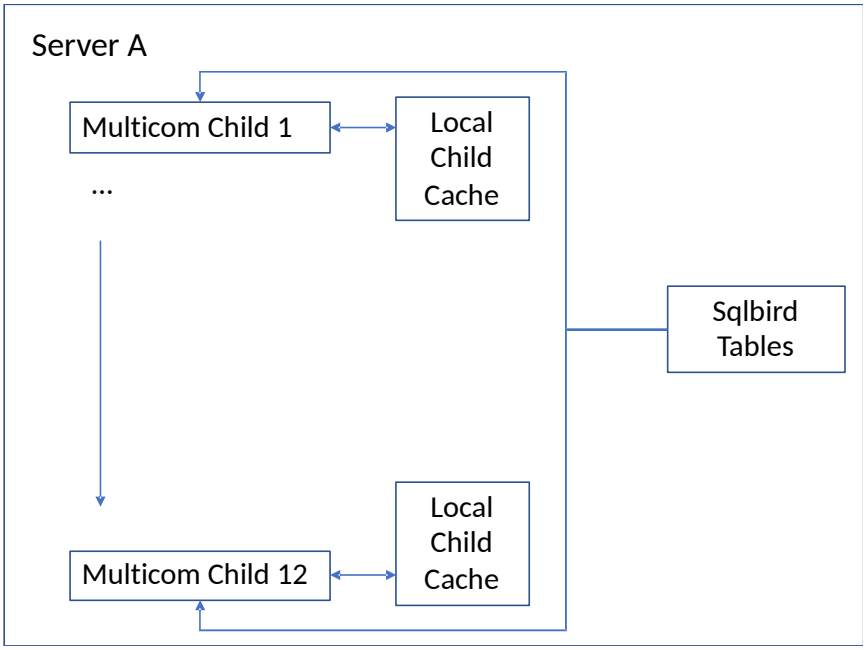

Figure B: new Multicom design with sqlite caches and sqlbird database table replication

The New Problem: Database Dependence

After sqlbird had such a great rollout in Multicom, we rolled it out to almost all of the systems at FlightAware. This means that whenever we had database problems, every application that accessed sqlbird exclusively instead of the database, was almost completely unaffected. This prompted a new expectation for applications at FlightAware – if the database goes down, your application does not.

This new expectation challenged Multicom yet again, because whenever Multicom would queue an alert to be sent, it used a postgres database table for a queue. The main queuing system would insert a line into this table for every alert that we wanted to queue for delivery. Then we had delivery channel daemons (5 total – 1 for every media type) that would continuously query this table and deliver any unprocessed alerts that had been queued. As a result, these additional reads and writes significantly increased the load on the main database. This was a result of legacy code that had been around for years, and we knew that we needed to make this better. It was time to stop using the database table as a queue and start using a real queue.

We decided to use tclrmq in order to help us with this transition. Using this package, we configured our own Mutlicom RabbitMQ virtual host with one queue for each delivery channel. The main system would write to each queue, depending on what channel the user configures for each of their alerts. Then each delivery channel daemon would receive these messages directly through RMQ without any database access.

Another thing that we used the database for was storing point in time references for each child process. This enabled us to restart Multicom and start at the same event messages that we just left off on. We found that zookeeper would be a better place to store this information, so we used zookeepertcl to make that change.

Overall I found the tclrmq package and zookeepertcl package to be very well documented and easy to use. These particular packages have a lot of value at FlightAware because they are used in critical technology that will have a very large effect in case of any failure. Whenever I found any issues with either package, I got serious attention and timely fixes. Although these packages are not part of the native language, the support that I received for them through the FlightAware organization was very good.

Conclusion

Multicom has grown significantly as a system within the past 2 years. In this time, it has faced significant hurdles from being completely overrun by a sudden influx of data, to meeting the company’s expectations to continue to run, even when the database is unavailable. Although the trials have been significant, we attribute a lot of Multicom’s continued growth and success to new TCL libraries such as sqlite, sqlbird, tclrmq, and zookeepertcl. These libraries have been instrumental in allowing this system to grow, and we look forward to learning about new ways that we can use TCL to improve this system.