Scalable and Fault Tolerant Multi-Receiver ADS-B Data Aggregation System with Kafka and Enriched GraphQL API

Copeland Royall

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2023

Abstract

At any given time, thousands of aircraft populate the skies above us. Unlike cars, which can be governed with simple signalling and common rules, aircraft operate at such speeds that pilots are unable to fly safely through visual contact with each other, particularly in congested airspaces where tens of airplanes might be queuing to land at one runway. Radio communication between pilots can be a substitute in very remote airports, but can never scale due to the limited speed of human speech. This is the reason for the air traffic control system — individuals on the ground coordinate the movements of aircraft by analyzing a representation of the real-world state of their airspace.

Automatic Dependent Surveillance-Broadcast (ADS-B) is the latest in air traffic control technology. It provides a means for aircraft to be located by controllers passively (unlike the existing solution, radar) and exposes more information than possible with traditional systems. It can be received and decoded with inexpensive software-defined radios, which has given rise to large networks of receivers achieving global tracking coverage.

This project aims to produce a system which can aggregate ADS-B data from multiple receivers and make it available over a GraphQL API enriched with Ordnance Survey geographic information. Such an interface is unique in the ADS-B data platform landscape, and provides far greater flexibility in querying than existing solutions — for instance, saving processing time and network bandwidth on mobile devices by only requesting certain data to be included in the response. This project also aims to utilize modern technologies — from the Go programming language, to the buffering and low-latency data processing system Kafka — to ensure it can be deployed and operated according to current best practices in large-scale software systems.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Copeland Royall*)

Acknowledgements

Michael Glienecke — Thank you for the many iterations of detailed feedback and the tandoori pickles.

Table of Contents

1 Introduction **1**
1.1 Achievements . . . 1
1.2 Report overview . . . 2

2 Background **3**
2.1 Aviation . . . 3
2.1.1 Air Traffic Control . . . 3
2.1.2 Primary Surveillance Radar . . . 4
2.1.3 Mode A/C . . . 4
2.1.4 Mode S . . . 4
2.1.5 ADS-B . . . 5
2.2 Large software systems . . . 6
2.2.1 Scaling . . . 6
2.2.2 Fault tolerance . . . 7
2.3 Multi-receiver networks . . . 7
2.3.1 ADS-B Exchange . . . 8
2.3.2 OpenSky Network . . . 10
2.3.3 AIRPORTS DL . . . 11
2.4 Impact of publicly-accessible ADS-B data . . . 11
2.5 Internet APIs . . . 12
2.5.1 GraphQL . . . 13
2.5.2 Alternatives . . . 14
2.6 Persistent data stores . . . 14
2.6.1 Relational and non-relational databases . . . 15
2.6.2 MongoDB . . . 15
2.7 ADS-B message processing . . . 16
2.7.1 Apache Kafka . . . 16
2.8 Go . . . 18

3 System design and development **19**
3.1 High-level introduction . . . 19
3.2 Architecture . . . 19
3.3 Implementation . . . 20
3.3.1 Feeder . . . 20
3.3.2 Ingest . . . 24

3.3.3 Kafka and MongoDB bridge 24
3.3.4 GraphQL API 25
3.4 Production deployment analysis 28

4 Evaluation **32**
4.1 Performance testing 32
4.1.1 Throughput 32
4.1.2 Latency 33
4.2 Malicious actors 33
4.2.1 Feeder de-anonymization 33
4.2.2 Invalid data 33

5 Conclusion **37**
5.1 Objectives met 37
5.2 Contributions to aircraft tracking domain 37
5.3 Future work 38

Chapter 1

Introduction

The ability to track the whereabouts of aircraft has previously been inaccessible to the general public both for financial and legal reasons — the radar technology used by the aviation industry requires sophisticated radio installations and high power transmission on extremely regulated frequencies. However, the introduction of a new aircraft location reporting standard, ADS-B, along with cheap software-defined radios[1] has allowed individuals to accurately monitor the skies above. By aggregating the data of many receivers located around the world, it is now possible to extend this view to a global scale.

Access to this data is useful not only for the aviation industry, but also the public as a means of monitoring the opaque activity of powerful individuals and governments (as will be discussed in section 2.4) — an example of the insights open aircraft data can provide when combined with other sources. It is therefore important that there are many solutions for aggregating this data and accessing it so that any use case can be explored unimpeded. However, currently all mechanisms for obtaining global ADS-B data with low latency are made available using the same REST interface design methodology (as will be discussed in section 2.5.1). As a result, the project aims to provide data access through a new type of programming interface — a GraphQL API — along with producing an alternative and modern implementation of a scalable, fault-tolerant ADS-B data aggregation system.

1.1 Achievements

This project has accomplished the following:

- Send decoded ADS-B data from a receiver to a central aggregation system
- Process and store ADS-B data with a scalable and fault tolerant architecture
- Make stored ADS-B data available for consumption via novel means — a GraphQL API, enriched with information about populated places and the ability

[1]Radio hardware which can receive signals from a wide range of frequencies without hardware modification.

to add new data sources with minimal effort

- Achieve low (10ms) end–to-end latency

1.2 Report overview

The report contains the following sections:

Chapter 2 will provide information on the aviation concepts relevant to this project, a description and architecture analysis of existing aggregation platforms and the software systems and tools used in the project with justifications for their selection.

Chapter 3 will describe the project's implementation — the basic components of the system, information about their subcomponents and detailed discussions on their source code.

Chapter 4 will evaluate the system's ability to be used in the real world.

Chapter 5 will conclude the report, providing final thoughts on the success of the system and future work.

Chapter 2

Background

This chapter will introduce the concept of ADS-B data through its context within past and current air traffic control technologies, the challenges associated with large-scale computer systems, the advent of global ADS-B data aggregators and the software used to implement this project — including GraphQL, the alternative solution to existing techniques of querying data from ADS-B platforms.

2.1 Aviation

From the perspective of a passenger, the operation of a flight is a simple procedure — taxi, takeoff, climb, cruise, descend, land and a final taxi to the gate. However, behind the scenes, each one of these phases of flight is an intricate process with constant communication between pilots and specialized traffic controllers. With the number of lives at stake, controllers need access to accurate visualizations monitoring the state of their designated airspace.

2.1.1 Air Traffic Control

The rise in air traffic in the last century has required increasingly sophisticated monitoring systems. The initial implementation of the Air Traffic Control (ATC) system we use today was introduced in the 1920s, and relied on controllers using verbal pilot reports and a scale map on which aircraft, represented by flags, were placed [24]. Given the last known position of aircraft, time elapsed and airspeed, controllers could predict future collisions and maintain safe separation between aircraft. In the 1950s, Radio Detection And Ranging (radar) was introduced to civil aviation [20]. The far greater precision and update frequency afforded by radar allowed for closer physical (and therefore temporal) separation between flights even as airplanes shed propellers for turbines and aviation entered the current Jet Age.

The following subsections will introduce the techniques used to obtain positional and additional information about aircraft — primary and secondary surveillance radar (including Mode A/C), Mode S and ADS-B.

2.1.2 Primary Surveillance Radar

The radar technology introduced after the second world war is now known as Primary Surveillance Radar (PSR) [36]. PSR emits a radio pulse and times how long a reflection of the pulse takes to return. Because radio travels at the speed of light, the distance of the reflecting aircraft to the radar can be calculated given the round-trip time. Radar systems utilize the Doppler effect — the shift in frequency of a wave when there is relative motion of the reflection surface, in this case, an aircraft, to the receiver — to determine the aircraft's movement away or towards the radar [16]. To achieve coverage in all directions, the radio transceiver and antenna are mounted on a rotating platform.

2.1.3 Mode A/C

Also in use today is Secondary Surveillance Radar (SSR) [36]. SSR differs from PSR in that aircraft take an active role — upon receiving a secondary radar ”interrogation”, aircraft transmit messages with their squawk code (ATC-assigned temporary identifier) and pressure altitude[1] using the Mode A and Mode C protocols respectively. This contrasts with PSR, where aircraft are passive reflectors of radio waves. Mode A/C SSR makes it easier for ATC to identify aircraft as this task is handled by the system, and it means they can observe all three dimensions of their airspace, even if only via a 2D representation.

SSR uses a different radio frequency to PSR — 1030 MHz for interrogation and 1090 MHz for replies.

2.1.4 Mode S

Mode S (Select) SSR improves upon the Mode A/C SSR protocols by assigning each aircraft a globally-unique 24-bit hexadecimal address unlike the locally-unique 12 bit ID afforded by Mode A's squawk code. This registration is performed by the International Civil Aviation Organization (ICAO). Mode S also facilitates ”selective interrogation” so aircraft can be addressed individually [36]. This increases the capacity of the 1090 MHz radio channel used for SSR and allows for higher-density airspace because interrogation replies do not interfere.

Mode S messages are not limited to the information provided by Mode A/C [36]. There are 11 message types in use, identified by the first five bits. The two ”extended squitter” message types are used for Automatic Dependent Surveillance-Broadcast (ADS-B) — ”automatic” because broadcasts are not requested by an observer and ”dependent” because the given information relies on onboard systems. This means ADS-B data can always be received even in areas where no SSR interrogations are made.

[1]The altitude calculated when the aircraft compares the outside air pressure to the observed pressure at the airport it is operating from. Above a certain ”transition” altitude the standard value of 1013 hPa is used by all aircraft.

2.1.5 ADS-B

Since 2020, all civil aviation aircraft operated in Europe and the United States are required to use ADS-B [36], with other countries following suit to varying degrees [10]. The 24-bit ICAO address is included in all messages. Each broadcast contains a single piece of information about the aircraft.

Examples of the message types are:

- Callsign (human-intelligble combination of airline code and flight number used by ATC and pilots, eg. VIR7B for a Virgin Atlantic flight from London to Los Angeles)
- Position (latitude/longitude)
- Position accuracy
- Altitude
- Vertical rate (change in altitude)
- Velocity (track angle and ground speed)

ADS-B transmissions can be rebroadcast by ground systems so aircraft can receive data beyond their receiver range.

ADS-B data can also be detected by satellite (see figure 2.1) — this means oceanic flights can be precisely tracked without radar coverage, succeeding the procedural control method (see figure 2.2) where aircraft are separated by 10 minute intervals (rather than spatial distance) on a shared route, with ATC relying on verbal position reports from aircraft for tracking [20].

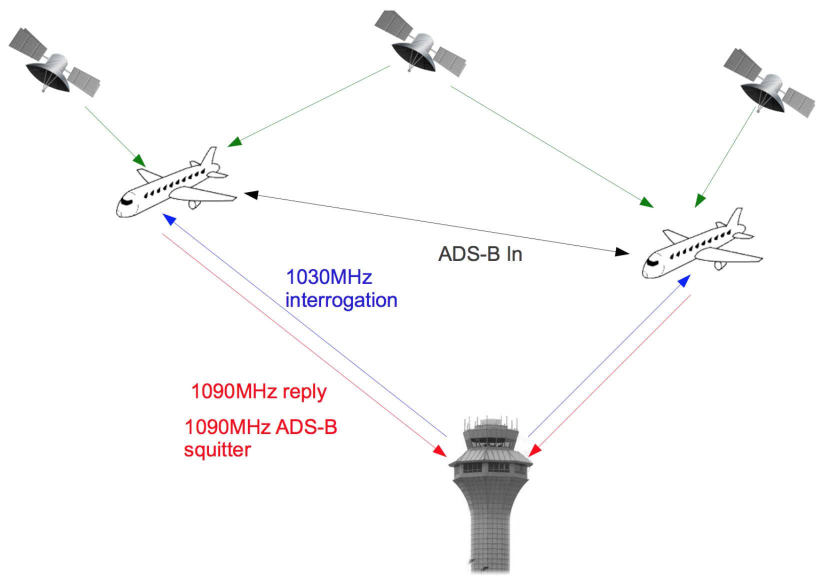
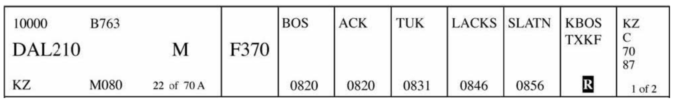


Figure 2.1: Data transfer of Mode S and ADS-B messages [32]



10000	B763			BOS	ACK	TUK	LACKS	SLATN	KBOS TXKF	KZ C 70 87
DAL210		M	F370							
KZ	M080	22 of 70 A		0820	0820	0831	0846	0856	R	1 of 2

Figure 2.2: Example FAA (U.S. aviation regulator) oceanic flight strip used to track an aircraft while using procedural control, with estimated times of arrival at reporting positions BOS, ACK etc. [18]

2.2 Large software systems

Large online platforms, like the ADS-B data aggregators mentioned in section 2.3, cannot be run in the same way an application is run as one copy on a personal computer — the volume of ingested data would overwhelm any single machine, and the reliability of the platform would be too dependent on one system. This section will introduce some of the considerations which are made when creating any service for wide-scale use.

2.2.1 Scaling

Computer systems operating on the internet need to be capable of handling many clients. If the underlying hardware is operating at maximum load, the service will need to be given access to more computing resources. This scaling can occur in two directions — horizontally or vertically.

Vertical scaling increases service capacity by increasing the computing capabilities of individual machines, through any combination of higher performance processors, more memory capacity, faster storage devices or the addition of hardware accelerators, like graphics cards. In public cloud computing, this is as easy as changing the processor count or memory capacity of the deployment's virtual machine. In most cases, this is a very simple approach to scaling because no changes need to be made to the software installation. However, there is a real-world limit to the performance of a single machine which will inevitably be surpassed with enough growth in usage.

Horizontal scaling increases the number of parallel service instances responsible for running the platform. If the existing software implementation is not designed to be run with multiple parallel copies, this could require extensive rewriting because of the challenges associated with parallel programming — namely assigning work, minimizing communication between workers and accessing shared data safely. If a service is well designed and written, horizontal scaling can provide effectively limitless computing resources.

As will be explored in section 3.3, the data aggregation problem associated with this project is embarrassingly parallel — given a single data storage system (which itself can be a parallel system) any number of writing and reading applications can be deployed because no interaction is needed between them. This makes it straightforward to implement within the horizontal scaling paradigm — for example, using a server orchestration system like Kubernetes[2], which could dynamically scale the number of

[2]Multi-machine service instance configuration and execution system.

service instances used by the platform depending on the amount of active flights and the resulting volume of ingested data.

2.2.2 Fault tolerance

Failures in the operation of any complex system are guaranteed. With computers, this can occur in the form of software or hardware faults leading to incorrect output or preventing proper execution of the system's code. Beyond an individual machine's state, large-scale issues like extreme weather events or infrastructure failures can impact online services. Lastly, operator intervention can cause downtime, whether intentionally through maintenance or unintentionally through misconfiguration.

The primary way to architect systems to withstand these challenges is through redundancy, which is a useful side-effect of building a horizontally-scalable system. This minimizes the number of single point of failures, where the integrity of the overall system depends on a single component remaining operational. For many online services, the critical tool to facilitate this has become viable in the last decade — cloud computing. Cloud providers offer organizations a method of provisioning resources at will in all continents but Antarctica. The responsibility for maintaining the hardware infrastructure then falls upon the cloud services provider. Similarly, public cloud platforms offer managed software systems, like databases, where they are also responsible for the software deployment, with users only needing to be concerned with their application code. If an instance of the database were to fail, another would be automatically provisioned and configured to exactly the same specifications. As a result, a large amount of modern systems are built to target these platforms. Section 3.4 will describe the ways in which this project's implementation approaches this.

2.3 Multi-receiver networks

The introduction of ADS-B and the rise in low-cost software-defined radios have made it accessible to track aircraft even for hobbyist individuals (see figure 2.3). However, terrestrial receivers have ranges in the hundreds of nautical miles — a small fraction of global air traffic. This led to the creation of multi-receiver networks, aggregating data to provide global coverage, visualized on maps (see figure 2.4). Some networks are commercial, selling access to their data and providing perks to ”feeders” (enthusiasts supplying data from their receiver). Others are community driven, with open access to air traffic data as the aim of the platform. Networks will often obtain data from satellite ADS-B receivers and perform Multilateration (MLAT, see figure 2.5) between ground receivers synced with the time from a shared ADS-B signal to estimate the position of aircraft without ADS-B capabilities (only Mode S) — this is impossible for a single receiver on its own. As of March 2023 the largest network, Flightradar24, is comprised of over 35,000 receivers [3].

The next sections introduce academic and commercial ADS-B networks and evaluates their features and architectures.

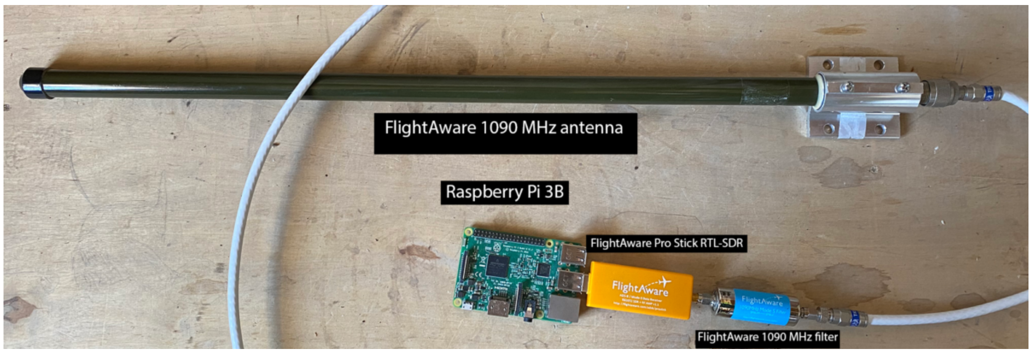

FlightAware 1090 MHz antenna
Raspberry Pi 3B
FlightAware Pro Stick RTL-SDR
FlightAware 1090 MHz filter

Figure 2.3: Components of a high-performance hobbyist ADS-B receiver [9]

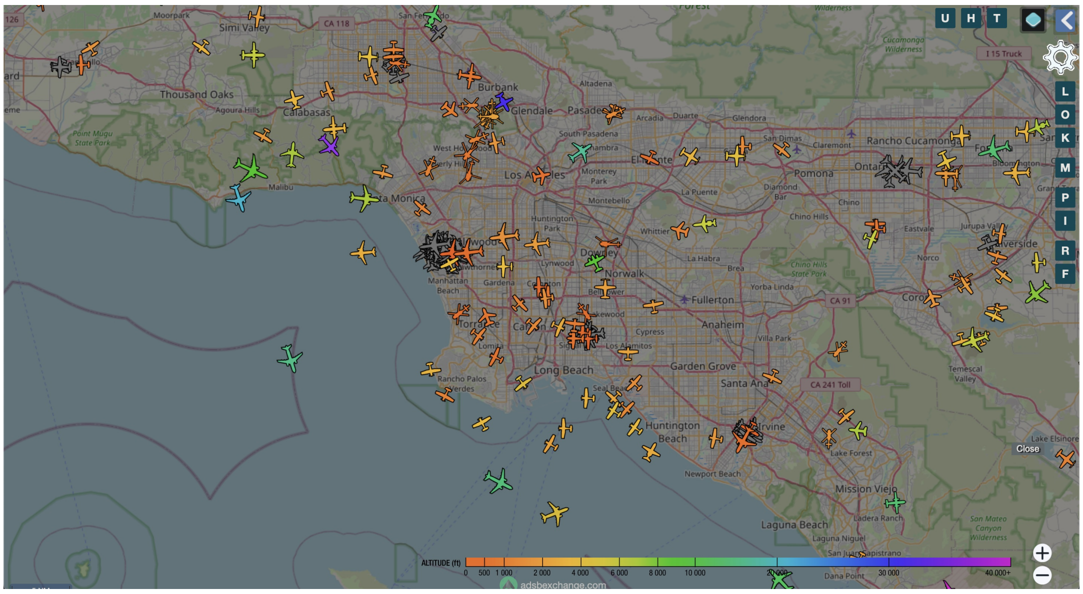


Figure 2.4: ADS-B Exchange visualization of evening traffic in the Greater Los Angeles area

2.3.1 ADS-B Exchange

ADS-B Exchange is self-described as a cooperative network. No data is filtered and its target demographic is primarily aviation enthusiasts (including those that feed the platform). The feeder ingest is horizontally scalable as a result of the parallel feed mergers and ADS-B decoders (see figure 2.6) — the number of these can be increased to cope with more feeders.

The site offers a historical data service [4], suggesting a persistent data store is part of the final layer. However, the main focus is live tracking, highlighted by the fact that this storage is not even mentioned in the architecture diagram.

The live REST API has very limited options for querying, only allowing the user to fetch the position of an individual aircraft based on ICAO address, registration or call sign; all aircraft in a preset nautical mile radius from a point and aircraft with a certain squawk code. A more sophisticated API would enable developers to write new types of applications.

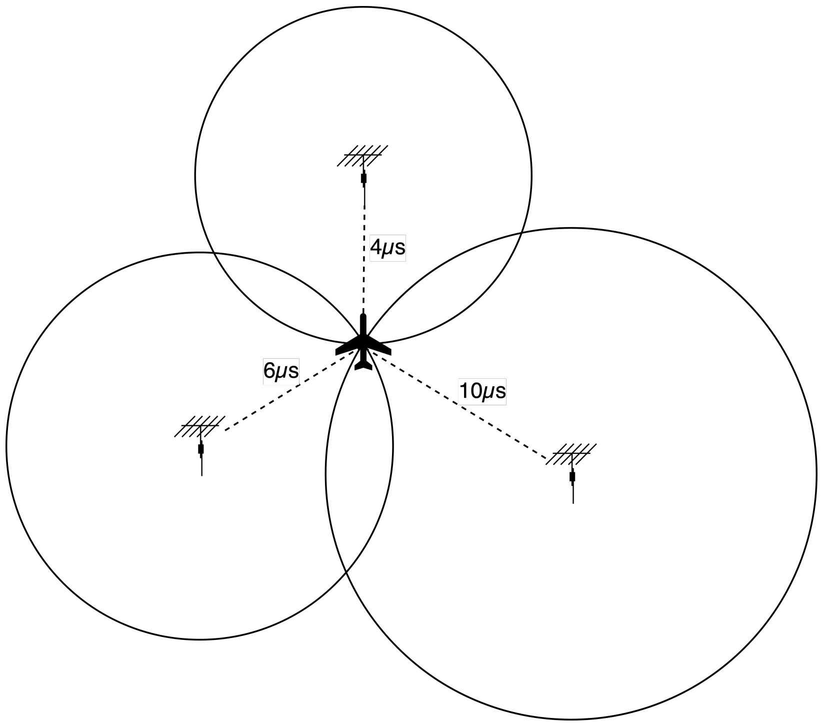


Figure 2.5: Multilateration of an aircraft's position based on the time delay of a message to each receiver and their positions

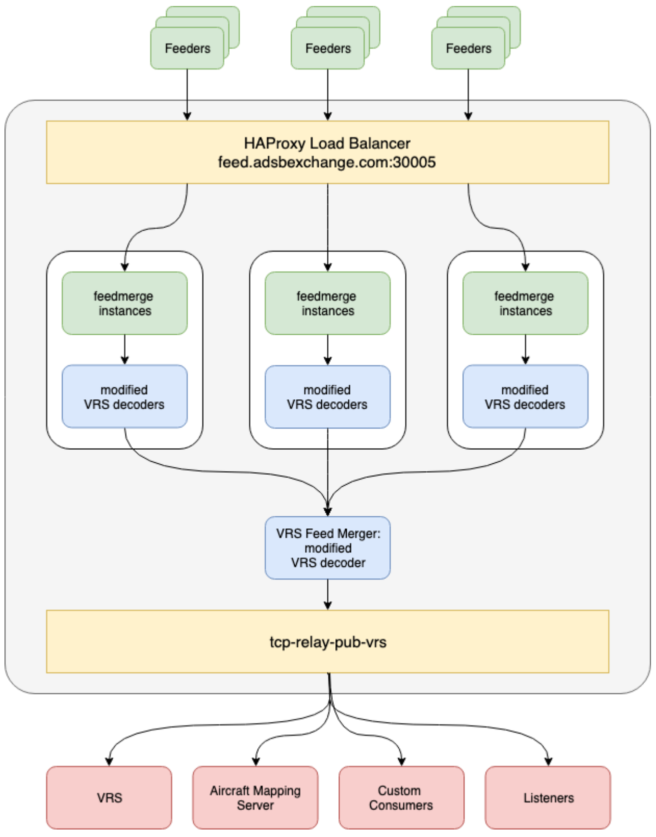


Figure 2.6: ADS-B Exchange system architecture [5]

A potential area for improvement is fault-tolerance as the root feed merger is a single point of failure — if it were to break, no incoming messages would be saved beyond what would be retained in the decoder buffers. A distributed approach to the merging and publishing of feeder data would improve the system in this sense since the only change would be a reduction in the maximum ingest rate. This would also make the system more scalable because the number of feed mergers could be increased to deal with higher load.

Another problem with the architecture is the lack of ingest buffering. As will be discussed in section 2.7, there is a risk of lost messages if enough of the feedmerge and VRS decoder nodes are down and the ingest load balancer routes an unmanageable volume of data to a few nodes. With a buffer, while the system is in a degraded state it could still accept all messages but process them slower than the ingest rate, catching up in a future time of lower load (assuming the buffer never becomes full).

2.3.2 OpenSky Network

OpenSky Network was created by academics from the UK, Germany and Switzerland as a freely-available source of aircraft tracking data for research. The platform initially struggled to scale as it used a single MySQL database [35]. It was later re-implemented using a lambda architecture[3] (see figure 2.7) — firstly an Apache Kafka ingest cluster to buffer incoming data, arriving in the order of 25 billion Mode S messages per day (averaging nearly 300,000 per second) [13]; a speed layer for realtime streaming with Apache Storm; a batch layer utilizing the Hadoop framework for storage and large-scale processing and a serving layer to distribute the ADS-B data [22]. All layers of the platform can be horizontally scaled.

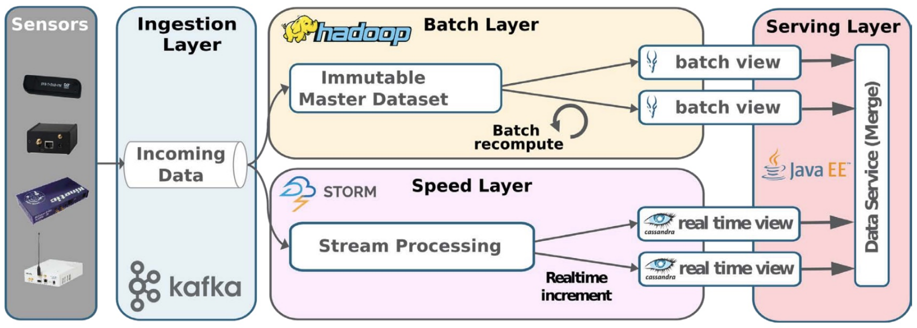


Figure 2.7: OpenSky Network system architecture [22]

The aim of using the platform for research is reflected in the integration of big data processing technologies in the system's architecture — historical data stored using the Hadoop big data framework can be accessed via an interactive Hadoop-native Apache Impala shell [21], unlike the extremely limited filtering abilities offered by ADS-B Exchange's historical JSON data service which requires manual downloading of single query responses.

[3]A system architecture that allows for realtime and aggregate data processing by using stream and batch processing in parallel.

Such a complex architecture has higher potential for issues when compared to a single-purpose system, however. Initial configuration and deployment of all necessary nodes will be time consuming. Whether because of upfront cost for self-hosted hardware or cumulative cost of renting cloud resources, operating the system will be very expensive. Ongoing maintenance and developer onboarding will be challenging. The use of multiple software systems creates a larger attack surface, both due to bugs in the applications' code and the increased likelihood of misconfiguration. While the clusters are internally fault-tolerant, overall there are many breaking points which could impact the health of the service.

2.3.3 AIRPORTS DL

AIRPORTS DL (see figure 2.8) is an academic proposal for an architecture of an aviation big data store [19]. While its function is not the same as other platforms — it does not directly handle ADS-B feeder data — it is relevant because it aggregates data from multiple sources. The use of Apache Flume and HBase (using the Hadoop framework) makes it unsuitable for low latency streaming of ingested ADS-B data due to batch processing in Hadoop and the Hadoop filesystem.

It is very similar to the batch layer of OpenSky Network but is able to make use of multiple data sources.

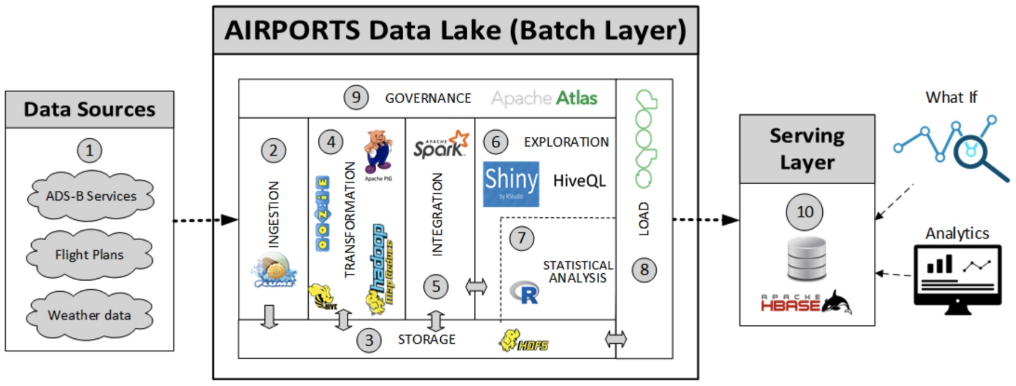


Figure 2.8: AIRPORTS DL system architecture [19]

2.4 Impact of publicly-accessible ADS-B data

For the aviation industry, improved aircraft tracking will result in greater efficiency and safety. There is significant interest in ADS-B/Mode S data for academic usage — Opensky lists 370 publications from research which made use of Opensky data [33]. However, there are uses for the general public too.

ADS-B data can be used to monitor adherence to flight rules by residents living near airports — for instance, to report aircraft that fly too low.

An aviation enthusiast developed Twitter bots to report on the movements of billionaires' private jets, prompting Elon Musk to try to take the account down [30]. Similarly, the environmental impact of billionaires' travel has been put under scrutiny based on ADS-B flight logs of their aircraft [25].

The 2020 George Floyd protests in the U.S. were revealed to have been monitored by government surveillance aircraft, including a Predator drone, likely carrying thermal imaging cameras and mobile phone trackers [39].

Dictator Alert [15] monitors the use of aircraft owned by dictatorships as defined by the Economist Intelligence Unit's Democracy Index.

It is worth nothing that every one of the investigations mentioned above used data from ADS-B Exchange — the platform does not use any FAA flight information, meaning it is not legally required to obey Limiting Aircraft Data Displayed requests [26]. However, the company is an LLC, a for-profit business, and was acquired in 2023 by an aviation industry intelligence provider, JETNET, which is itself supported by growth equity firm Silversmith Capital Partners [23]. When those with the most to gain from hiding their air travel from these platforms are also the most financially and politically powerful people in the world (for example, Elon Musk) it is hard to imagine that ADS-B Exchange data will remain unfiltered, especially given JETNET's industry customers will likely only be concerned with the aircraft in their own fleet or the industry as a whole, preventing there from being any monetary incentive to collect all data. As was discussed in section 2.3.1, ADS-B Exchange uses entirely open source software, so equivalent platforms have been created by the project's volunteers, but none with the same feeder coverage. As a non-profit, OpenSky is the largest aggregator which is relatively immune to these external pressures, but having multiple such platforms is important for academic progress and redundancy's sake, both in the separation of organizations running the platforms and the composition and complexity of their software implementations.

2.5 Internet APIs

When a program wants to fetch information over a network, it uses an Application Programming Interface (API). For end-user clients communicating with a server over the internet, this usually involves a Representational State Transfer (REST) API.

Often a client will make multiple requests to get all the information it needs when using a REST API. For example, when a browser views the tracking map on an ADS-B data exchange mentioned in section 2.3, it first requests all the ADS-B data for aircraft in a given area. If the user would also like to overlay aircraft-specific information on the map, the browser would need to make separate requests to fetch that data based on the ICAO address of the selected flight. This increases network usage and response time for the user because the requests must be performed serially and with a new connection for each query — REST APIs are stateless, meaning servers do not store session information.

As one goal of this project is to provide a novel means of querying ADS-B data, and all low latency APIs from existing platforms utilize REST, a different solution must be used.

2.5.1 GraphQL

GraphQL is an alternative to REST and other API approaches. A client can request all the information it needs in one query, with the server performing the task of combining data together, potentially making multiple external requests of its own (see figure 2.9) before returning a single response. This saves network bandwidth and reduces client latency because of the reduction in queries and the minimized response. This can also be used to prevent the client from needing to communicate with multiple APIs since it can request any data it needs, with the responsibility of sourcing the data on the server.

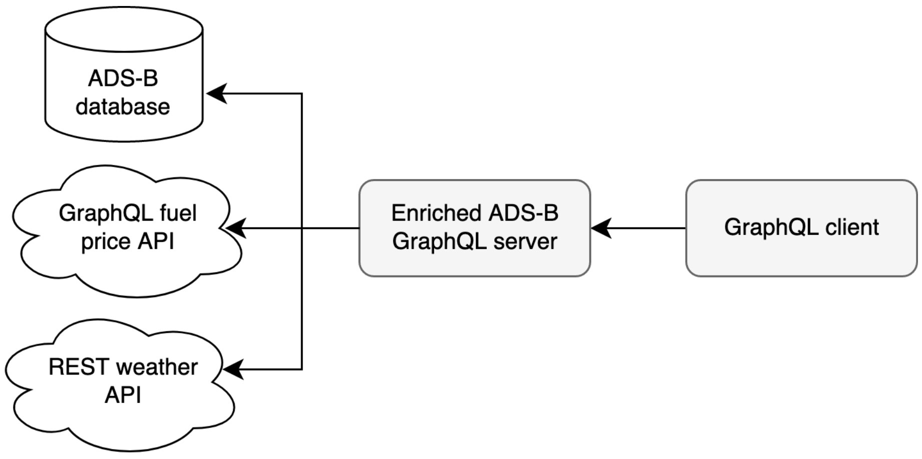


Figure 2.9: Aggregation of multiple data sources into one GraphQL API. The client can query data which may come from any combination of the upstream sources

A GraphQL API is defined by a schema. This is a document that describes the types of data that can be queried and their relations to each other (which could also be represented as edges on a graph). Each type is a structure of fields potentially containing references to more custom types, as well as primitive types like numbers and strings. A GraphQL query has to request data using a custom type — often one defined with the name Query — with subfields for different queries, each with parameters to filter its responses. Within the constraints of the requested type's structure, a query can be written to include fields in unforeseen and complicated ways that would be difficult to do RESTfully. The fundamental requirement for the API to have a schema benefits users because the API is self-documenting and it is impossible for it to be outdated. The schema also provides type safety because clients know exactly what can be returned for all queries.

Another advantage of GraphQL is that it is not limited to stateless requests if GraphQL Subscriptions are used. These can push updates to clients when previously-requested data changes in realtime. Under the hood, the WebSocket protocol establishes a TCP connection between client and server, eliminating the need for the client to continuously poll for new data by making the same query repeatedly and comparing to previous results.

There are two types of GraphQL server implementations — schema-first and code-first. Schema-first means the data types that can be returned are defined by hand and used to determine the query functions that need to be implemented. Compared to code-first approaches, this is easier to understand conceptually as the end goal is defined first rather than writing code that happens to create the desired schema. This also means it can produce more stable APIs as an accidental change in code cannot change the API's schema — it will be caught during the server program generation stage. Schema-first implementations can have better integration with the implementation language's type system because new types can be created to match those from the schema, giving the benefits of static typing (detecting bugs before the program is executed). However, this rigidity also makes it more time-consuming to modify the API because any changes must first be finalized in the schema before the implementation can be updated, potentially having a knock-on effect in limiting the evolution of the system's architecture. In large projects, writing a schema before any implementation could take a significant amount of effort; this project is small enough that this is not the case.

2.5.2 Alternatives

Other than GraphQL, there are two main alternatives to REST — gRPC (gRPC Remote Procedure Call) and Simple Object Access Protocol (SOAP), which is used almost exclusively by legacy systems.

SOAP uses Extensible Markup Language (XML) messages as opposed to the JavaScript Object Notation (JSON) used by GraphQL and many REST APIs (though some use XML and other formats). XML is notoriously verbose, meaning large messages will have an unnecessarily high bandwidth cost and take more processing time to deserialize on the client — potentially a problem if accessing realtime global air traffic data on mobile devices. SOAP is rigid, providing stronger guarantees about the usage of the API but also being complex for end users — the opposite of the goal of making ADS-B data accessible in new ways.

gRPC, while being very performant due to its use of HTTP/2[4] and predetermined serialization and deserialization formats instead of the ability to handle generic JSON types, is unsuitable as it is challenging to use with a web browser [38]. The fixed formats also require synchronization between client and server versions, making modifying the API much harder. These factors make it even less applicable to creating an open interactive API than SOAP.

Overall, it is clear that GraphQL is the only API approach that would provide value beyond the REST APIs of existing ADS-B aggregation platforms.

2.6 Persistent data stores

Storing data can be accomplished as simply as writing plain text files to a hard drive, but for a reliable and scalable system a more sophisticated solution is needed. A database is a tool that organizes, stores and queries data.

[4]A modern version of HTTP with many improvements over the existing HTTP/1.1 protocol from the late 1990s, including making multiple parallel requests over the same connection.

2.6.1 Relational and non-relational databases

The traditional approach to database management systems (DBMS) is the relational database, in which data is stored within related structured tables, similar to spreadsheets. Each row is an entry described by its columns. Because their structure is so well-defined, data can have specific constraints and queries, often made using Structured Query Language (SQL), can be highly complex.

Non-relational databases are a different type of DBMS that do not use the tabular approach, of which document databases are a subset. Document-oriented databases are well suited for storing semi-structured (less rigidly defined than table rows) ADS-B messages — ADS-B data has a very large number of optional values, each of which would need to have its own column that would often be empty, wasting resources. Other data used to enrich the API, like weather or mapping, can also have many optional fields. For instance, the Ordnance Survery Open Names dataset used to enrich the data with populated places information has 12 optional fields out of 22. To store a received ADS-B JSON message (as will be explored in section 3.3.1) the system would need to map each input field to a specific column. If the JSON message format changed, this mapping would also need to be changed. With a document database, the JSON message can be stored directly, though any queries would need to account for a change in the message format. There are multiple databases that are capable of being hosted on local machines (and are open source) that fit this criteria — MongoDB, Apache Cassandra, RavenDB and Couchbase.

2.6.2 MongoDB

MongoDB was selected as the persistent data storage system for multiple reasons beyond its document-oriented design.

For the mapping-enriched feature of the API, MongoDB is a good choice because it has native support for storage and operations on GeoJSON data.

MongoDB can be sharded into many instances. This allows the system to scale if the receiver output or client requests exceeded the limits of a single server.

There are other document-oriented databases with similar functionality to MongoDB, such as Couchbase and Apache Cassandra, but MongoDB has the largest user base [11] — even Amazon DocumentDB has a MongoDB-compatible API [37]. This makes it suitable for self-hosted deployments and use in the cloud with managed solutions. Overall, though, the most significant reason for the choice to use MongoDB was its developer community and ease of use, and not for technical reasons — all the major document databases are of high quality.

The database used in this project has two document collections — one for ADS-B messages, as described previously, and one for the GeoJSON Ordnance Survery Open Names dataset used for the API enrichment. This data was converted from the British National Grid reference system to WSG84 (latitude and longitude) using the Python package bng-latlon [28].

2.7 ADS-B message processing

This section will introduce a major part of the system's architecture that is used as part of the ADS-B message ingest pipeline before permanent storage in MongoDB. In the current implementation, it is only equipped to improve write performance, but with further work it could carry out extra processing.

By itself, a MongoDB cluster could handle the maximum load of the largest ADS-B aggregation networks — each message is $\sim$1KB, with a rate of messages in the order of (at most) hundreds of thousands per second (tens of GB per day). However, a temporarily degraded cluster (for example, the period between a node failing and a new one being provisioned) during peak times could cause the cluster's write capacity to be lower than the ingest rate. This would result in lost messages. The solution to this is a buffer, a separate system that can handle writes at a higher rate. The database can read from the buffer at its maximum write rate, and catch up on the extra messages when the overall system ingest rate is lower.

There are other problems that can be solved before the data reaches the persistent storage. As mentioned previously, ADS-B aggregation platforms perform MLAT to determine the location of aircraft broadcasting only on Mode S. The networks also reject data that appears invalid or is duplicated. These steps could be performed before the data is written to the database to prevent excess database operations. This type of workload is called stream processing — data is constantly analysed and processed while it is ingested.

There are many options for software to perform either or both of these tasks, like RabbitMQ, Apache Pulsar, Apache Storm and Apache Spark. However, the following section will introduce the system that was chosen for the project — Apache Kafka.

2.7.1 Apache Kafka

This project uses Apache Kafka for data ingest buffering. Kafka is the de facto standard for stream processing workloads, used by 80% of all Fortune 100 companies [7]. This makes it the easiest solution to get support for because of the size of the developer community (as with MongoDB). Kafka can also perform both buffering and stream processing workloads. As mentioned before, OpenSky initially used a single database and had to be reimplemented to use Kafka for buffering to allow the platform to scale.

From a high level, data producers write messages into Kafka topics — abstracted streams of related messages. Consumers can then read from topics in the same order that messages were initially written.

Kafka is ideal for ingest buffering because it can be horizontally scaled. This is achieved through topic partitioning (see figure 2.10). Each message is associated with a key when it is produced. The key is hashed to select the partition for it to be written to. These partitions can be created on separate machines, so that each is responsible for a fraction of the overall amount of messages being ingested. Messages with the same key are sent to the same partition, meaning the order of the messages in the topic is guaranteed.

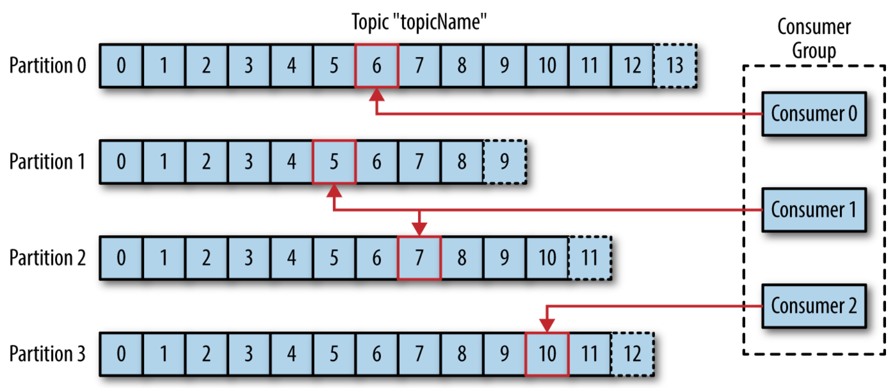


Figure 2.10: A topic with multiple messages and partitions being read by multiple consumers [1]

In this project, Kafka is used between the receivers sending ADS-B messages to the system and the permanent data store where the messages are stored and queried, MongoDB (see section 3.2 for more information). In Kafka, the persistent ID of receivers is used as a key for its submitted messages (see 3.3.1). The order of messages is not important because each is timestamped by the receiver and can be retrieved in order from the database. However, this consistency of mapping receivers to partitions ensures the load is evenly distributed across all the machines in the Kafka cluster without using the trivial round-robin approach.

An important potential usecase of Kafka for the project would be to use its stream processing capabilities to perform computation on the ingested data before it is stored in MongoDB. For example, MLAT to determine the location of aircraft without ADS-B capabilities (see figure 2.5), or rejection of invalid messages through comparison with other feeders' messages or predictive algorithms. Instead of passing messages straight from the ingest program through Kafka to MongoDB, the stream processing programs would create intermediary Kafka topics, with the processed data finally ending up in a single topic for MongoDB to read from. A stream processor in Kafka could also fetch enrichment data from external sources before feeding it into MongoDB — this would allow the data to be modified before permanent storage and use.

Kafka writes all its messages to persistent storage as well as MongoDB. This is useful as it means streams can be reprocessed (for example, if there is an bug with the MLAT implementation). A traditional database is still necessary though, because Kafka cannot be queried efficiently.

There are many alternatives to Kafka, including Apache Pulsar and RabbitMQ. At the small scale of this project, the majority of these tools would work as effectively as each other. However, Kafka is relatively featureful and has the most developer support due to its widespread use.

2.8 Go

All custom artifacts of the system are written in Go. Go is a suitable choice for its technical merits and the developer experience.

Go is a compiled language, meaning the conversion from high-level human-written code to hardware-specific executable machine code occurs before the application can be run, whereas interpreted languages (like Python and JavaScript) are translated as the program runs. This increases execution time for interpreted languages due to the constant conversion steps and memory overhead for the translation program. Go also offers concurrency, so programs can execute in multiple threads.

For developers, Go is easy to work with. Its high level of abstraction means developers do not need to handle memory management — a challenging programming problem and a cause of many errors and security vulnerabilities. As mentioned before, Go offers concurrency. This is done through Goroutines, a simple and lightweight method of running multiple tasks at once. Because Go is statically typed, any related errors can be caught during compilation and developers can infer more information about variables than just the name. Being a compiled language, Go programs with few dependencies can be containerized[5] with minimal effort as they require no runtime environment. Its minimal syntax makes it easy to read and write.

Lastly, because of all the above reasons, Go is already a popular language for systems and infrastructure. This means it has strong tooling and a large developer community who would be able to work on the project if it were open source.

As with the the decision to use Kafka, at this small scale the choice of using Go against other high level languages is partly for the developer experience and not only technical reasons.

[5]A new approach to server software packaging that enables applications to run in isolated environments.

Chapter 3

System design and development

This chapter will present the project's final implementation, through its structure, detailed descriptions of its subcomponents and GraphQL API.

3.1 High-level introduction

The project's software implementation is comprised of two discrete parts (see figure 3.1).

The first is a feeder application. This program sends the data being decoded from the ADS-B receiver to the platform.

The second is the data aggregation system. This is comprised of many parts and is the majority of the project's work.

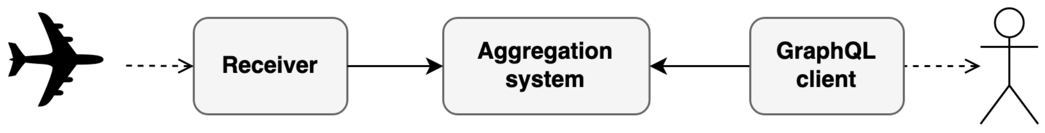


Figure 3.1: High-level system architecture

3.2 Architecture

This section will describe the internal structure of both the receiver and aggregation system.

The feeder makes use of an open-source program, readsb [34], to decode the raw receiver signal (see figure 3.2). This is because decoding from a software-defined radio receiver requires a sophisticated driver implementation. This project's feeder application is written to use readsb for this purpose because it provides a simple JSON feed of received data. Other ADS-B aggregation platforms use server-side decoders working with raw hexadecimal-encoded data sent from the feeders because it allows them to perform MLAT. However, this introduces additional complexity and server-side processing.

The feeders submit their JSON data to an HTTP ingest endpoint in the aggregation system (see figure 3.3). Each feeder has a persistent identifier which is added to each submission. This means basic HTTP authentication can be performed to limit malicious messages, and data from individual receivers can be isolated in GraphQL queries.

The ingest application pushes the received data into an Apache Kafka cluster.

Another service reads the ingested data from Kafka and writes it into MongoDB.

Lastly, a GraphQL server program handles requests from clients, fetching relevant data from MongoDB. This includes non-ADS-B data, which is used to ”enrich” the API and is a differentiator between this system and preexisting APIs from public platforms.

3.3 Implementation

This section will explain the implementation of each subcomponent of the final system.

3.3.1 Feeder

The feeder application forwards the ADS-B data decoded by readsb to the aggregation system.

Among other formats, readsb can output data as JSON (see code 3.1). It exposes this through the filesystem or via a TCP port. The filesystem approach is suboptimal because of the limited write endurance of the disks used by single-board computers that are dominant in ADS-B receivers — for instance, the Raspberry Pi uses a microSD card [14]. A way to solve this problem would be to make use of a Unix temporary file system (tmpfs) which would store the file in memory. However, this solution requires manual configuration to create the tmpfs mount so is less end-user-friendly. In contrast, using the TCP option requires no setup other than enabling the relevant flags for readsb. It has the additional benefit of allowing the feeder program to be run on a separate machine (for instance, if compute resources are extremely constrained on the receiver).

As mentioned in the architecture overview, each feeder application needs a unique identifier for authentication and data querying purposes. This could be generated by the feeder program randomly, but persisting the ID between sessions would require writing to a file, introducing code complexity and configuration overhead. Instead, the program makes use of the fact that readsb targets the Linux platform. Each Linux

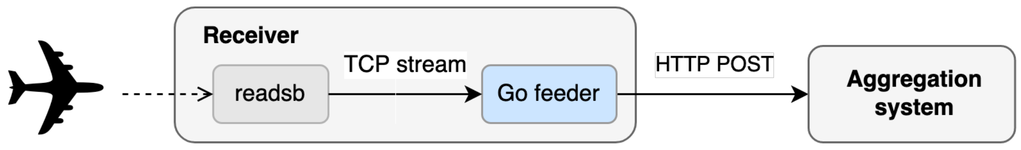


Figure 3.2: Internal structure of receiver

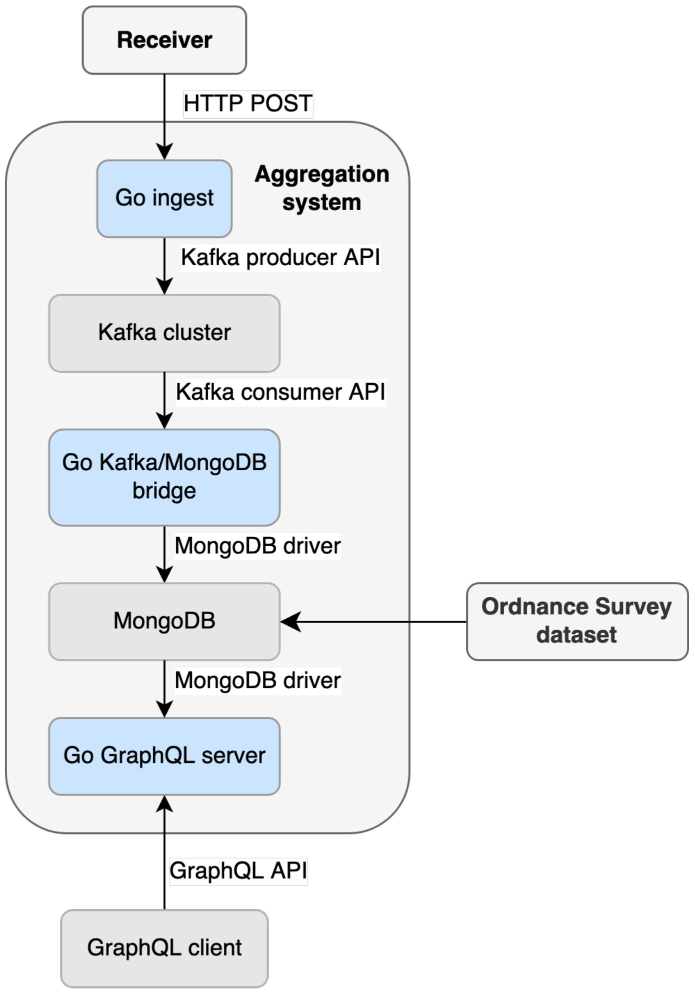


Figure 3.3: Internal structure of aggregation system

```
{
    "hex": "4CA2A7",
    "flight": "RYR4",
    "alt_geom": 3200,
    "tas": 192,
    "messages": 1253,
    "rssi": -4.4,
    "now": 1668193393.204,
    "geom_rate": -288,
    "nav_qnh": 1015.2,
    "lat": 56.027061,
    "alt_baro": 2950,
    "gs": 159.7,
    "true_heading": 279.01,
    "alert": 0,
    "ias": 182,
    "baro_rate": -448,
    "roll": -15.47,
    "squawk": "6012",
    "category": "A3",
    "nav_heading": 274.92,
    "mag_heading": 280.02,
    "lon": -3.086901
}
```

Code 3.1: Example readsb message JSON output (some fields removed) installation has a 128-bit Universally Unique ID (UUID) stored as 16 hexadecimal characters in the /etc/machine-id file. Using the full OS UUID as the feeder ID in each ADS-B message would result in wasted network bandwidth and data storage — even the largest receiver networks only have numbers in the range of tens of thousands. The feeder program truncates the UUID to 10 digits (see code 3.2) providing a balance between collision likelihood reduction and a shorter length. For instance, the machine-id ”e8d39f4e4b4c4e5d9c5919bafedf6b7d” would be shortened to a feeder ID of ”e8d39f4e4b”.

Assuming Linux UUIDs are uniformly random and using a generalized birthday problem[1] approximation:

$$p(n,d) \approx 1 - \left(\frac{d-1}{d}\right)^{\frac{n(n-1)}{2}}$$

where $d = 2^{10\times4}$ (number of possible IDs with 10 digit hexadecimal string) and $n = 30,000$ (number of feeders in the largest ADS-B receiver networks) the probability of an ID collision is 0.04%. Given the consequences of a two feeders sharing an ID are not significant in the current implementation, this is an acceptable risk of error. However, if the system were to incorporate MLAT, it could use the other feeder’s location as its own when attempting to triangulate the aircraft position, which would make the results unusable. The collision would also be a problem if the system began performing invalid data checking because the receiver’s location could be impossibly far from the ADS-B messages it receives. The system would then reject all messages from that feeder. If these features are introduced, the feeder program could be updated to increase the number of machine-id characters it uses for its ID to reduce the probability of collision to a low enough level. This would have no effect on existing receivers because IDs of different lengths cannot collide and the system has no assumptions about ID length.

```
machineFile, err := os.ReadFile("/etc/machine-id")
if err != nil {
    log.Fatal(err)
}
uuid := string(machineFile[:10])
```

Code 3.2: Go source code for feeder ID selection

As was introduced in section 2.7.1, Kafka, the software used for feeder data ingest buffering, can be configured to use multiple partitions per topic by assigning a key to each related message. In this proof-of-concept implementation, the feeder ID is used to ensure messages from the same feeder all use the same partition.

[1]The probability of at least two people sharing a birthday given a specified number. In this case, the probability of a collision between two feeder IDs depends on the number of feeders and the number of possible IDs (which would usually be fixed to 365 in the case of birthdays).

3.3.2 Ingest

The ingest application receives new ADS-B data from deployed feeders. It exposes an HTTP endpoint which accepts POST requests. The request body is written to the Kafka cluster by a producer using the Confluent Kafka Go client module. For simplicity's sake, the Kafka cluster used for the project is configured to run on the same machine as the ingest application. This means the client connects via the ”localhost” address without authentication and firewall configuration. The client has a fixed ID: ”adsb_ingest”. This would need to be set dynamically if multiple instances of the ingest application were to be run simultaneously.

HTTP is used for ingest as opposed to raw TCP connections. This allows the ingest to perform feeder ID authorization checks using the Authorization HTTP header field (see code 3.3). A major advantage of this approach would be relevant at a larger scale — because HTTP is so widespread, adding encryption (via HTTPS, to prevent feeder location information and feeder ID authentication from being revealed over the internet) load balancing and web application firewalls would be straightforward as all tools are designed to work with HTTP. In this project's small scale, HTTP is simpler to implement features with compared to writing a custom protocol — connections do not need to be created and kept track of with the application's code, and HTTP headers can be used as opposed to manually defining a standard for metadata to be included before ADS-B message data (which would need to be kept consistent between all feeders and ingest instances, making it very difficult to update either).

```go
// Go does not have a type for sets, so use a map
var authenticatedFeeders = map[string]bool{"e8d39f4e4b": true}

...

feeder := request.Header.Get("Authorization")
if !authenticatedFeeders[feeder] {
    http.Error(w, "Unauthorized feeder", http.StatusForbidden)
    log.Println("Unauthorized access: " + feeder)
    return
}
```

Code 3.3: Go source code for ingest feeder ID authorization check

3.3.3 Kafka and MongoDB bridge

Once data is ingested by the program mentioned above and published to Kafka, an intermediary application is needed to consume the data and write it to MongoDB. This could be accomplished by the MongoDB Kafka Connector [29], though it is complicated to setup for development purposes. Instead a custom Go bridge program performs this task using the Confluent Kafka Go client and MongoDB Go driver modules so that all parts of the system (other than Kafka and MongoDB) can use the same development environment and deployment practices as it is all written in Go. In the future this program could also perform the task of fetching static (not GraphQL query-specific) enrichment data — something that would be challenging to do with the premade connector as it would involve modifying its source code and merging any upstream changes to the connector with each update.

This part of the architecture is also where any stream processors (for example, invalid data rejection) would function — taking ingested data from one topic, performing computations and outputting the modified data on another topic, which the bridge program would read from.

3.3.4 GraphQL API

This project uses gqlgen to generate a GraphQL API server. Of Go GraphQL server implementations, gqlgen is the most feature-rich and uses Go types [12] — the code-first library graphql-go uses generic types which eliminates one of the advantages of statically typed languages like Go. Once the schema is defined (see code 3.4) a server generation command is used which automatically writes the source code for a Go GraphQL server using the given schema. This includes a file in which the resolvers for each GraphQL data type are written — in this case, using the MongoDB API to make queries to the ADS-B and OS place enrichment datasets. The server code can then be built and run like any other Go application (again, allowing consistency in deployment with the other custom Go software).

The GraphQL server supports the following queries:

1. messages: Fetches all messages for a given ICAO address. This can be used to fetch the full history of received data for an aircraft.

2. flightpathHistory: Gets the flightpath of an ICAO address within a time period defined by the current time and a given length of time in the past. This could be used by a frontend client to display the recent flightpath of an aircraft when it's selected. The return value contains the flight's location history coordinates in the period and a list of names of populated places (defined by Ordnance Survery) the flight passed within a given radius of.

3. flightsOverPlace: Gets flights that have passed within a radius of a longitude/latitude pair within a time period (see figure 3.4). This could be used to check compliance with airspace restrictions. The return value is a list of flights, each with identifying information and the coordinate history that matched the requirements.

4. flightsInConstraints: Gets messages filtered to match the given altitude and speed constraints. In the U.S., this could be used to check that aircraft adhere to regulation FAR 91.117(a), requiring aircraft to fly at less than 250 knots below 10,000 feet [2].

5. flightsAtRisk: Gets messages that indicate an aircraft has an ADS-B alert active, or is descending at a greater rate than a given value, or is travelling at a speed greater than a given value while below a given altitude. This could be used to monitor aircraft that may need assistance.

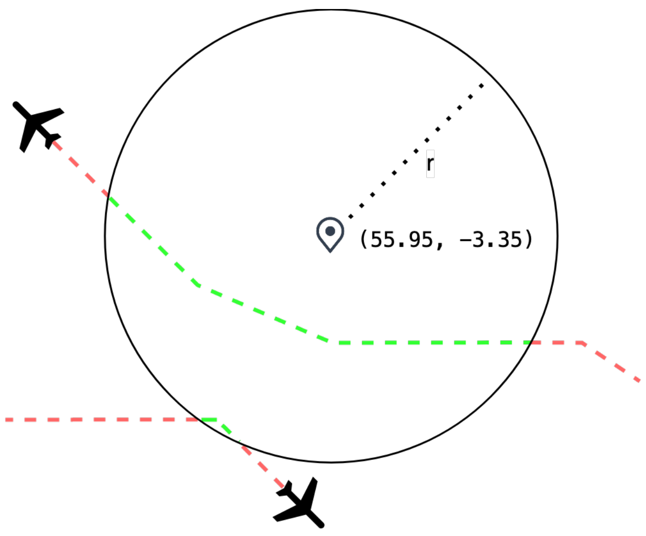


Figure 3.4: Subsections of flightpaths that would be returned for flightsOverPlace query

3.3.4.1 Polylines

Flightpaths are encoded using Google's polyline algorithm [17], taking a list of co-ordinates and generating an ASCII string, compressing the path in the process. This approach is lossy, so the accuracy of the path is limited, but it greatly reduces the data needed to be transferred in the query result. readsb decodes degrees to a precision of six decimal places, which the polyline algorithm lowers to five. At the equator, where the geographical difference between longitude lines is greatest, $1e{-}6^\circ \approx 0.1$m, meaning the encoded position will be off by at most ~1m. Since a common use of ADS-B platforms is to view the tracking map by eye, this difference is irrelevant, especially when the wingspans of even narrow-body commercial airlines are ~36m. If more precise location history data is required, messages could be exported directly from MongoDB without GraphQL, which would also give more flexibility in possible queries.

Figure 3.5 shows the final 16 points of a flight landing at Edinburgh Airport, 314 characters in comma-separated value (CSV) form. The same information is encoded in 25% the size as a 78 character polyline:

rt{RkdvtIjj@jMpl@jNpk@zMxh@ L‘l@jN‘l@lNje@—Kdj@—Mbj@pM‘g@dLfk
@bNha@lJdk@dNhSzE

The full 580 point flightpath from Paris Charles de Gaulle Airport to Edinburgh Aiport is 11,214 characters in CSV form and 2,976 as a polyline, 27% of the size.

3.3.4.2 Populated place data enrichment

The flightpathHistory query type returns a flightpath containing both the route travelled in coordinates (encoded as a polyline) and a list of populated places the flight travelled near (range specified by the query parameters). This data is taken from the Ordnance Survery Open Names dataset. It could be considered to be a human-readable encoding of the flightpath.

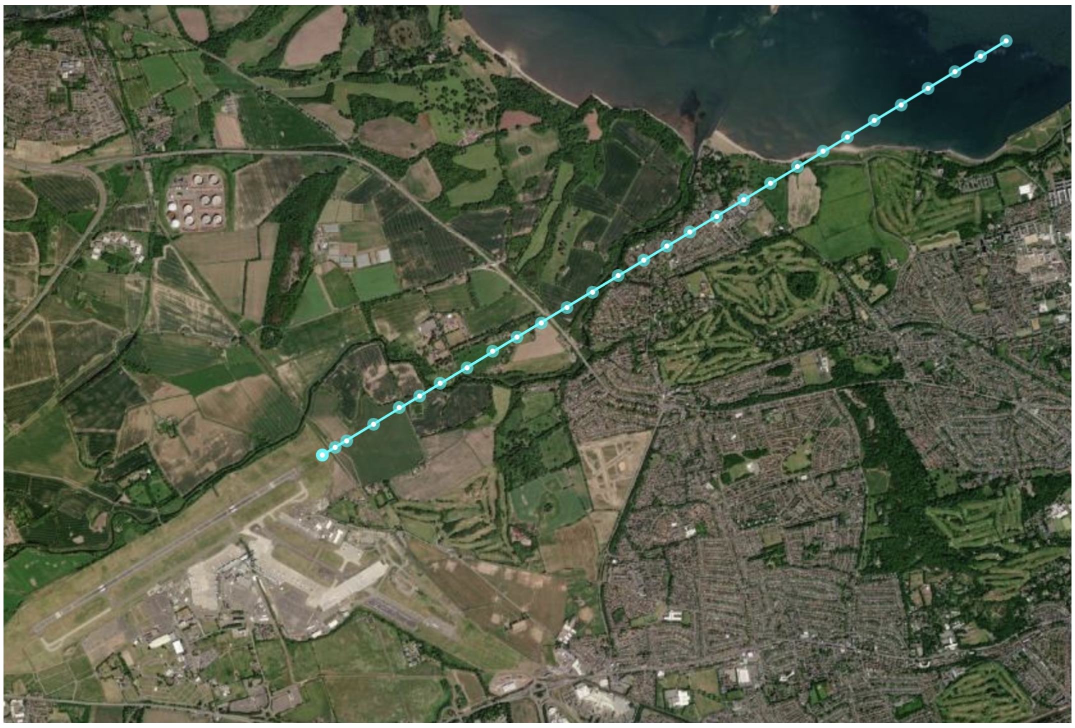


Figure 3.5: ADS-B message locations for airplane on short final for Edinburgh Airport runway 24 (satellite image © 2023 Esri)

3.3.4.3 GeoJSON

GeoJSON, an open standard format for encoding geographical data, is used to more easily operate on location information. For the flightsOverPlace query, message locations are checked for intersection with a circle defined by the query parameters.

For the flightpathHistory query, temporally-adjacent pairs of messages are formed into rectangles. These rectangles are combined into a GeoJSON MultiPolygon object, which places are checked for intersection against. Because polygons are used, the size of the search area is slightly too large — the corners of the rectangles should not be included (shown red in figure 3.6). This is a limitation of GeoJSON, as circles are not included in the specification.

An alternative solution would be to create approximations of circles at each flightpath point (many-sided polygons) and reduce the length of the rectangles. However, this would add unnecessary computation time and implementation complexity for no practical gain.

Code extract 3.5 is used to produce a GeoJSON multipolygon from a list of coordinates. As mentioned above, the coordinates of temporally-adjacent ADS-B messages are combined into rectangular polygons. Five points are used to define each polygon to close it — GeoJSON creates lines between adjacent points, but will not wrap around from the last to the first.

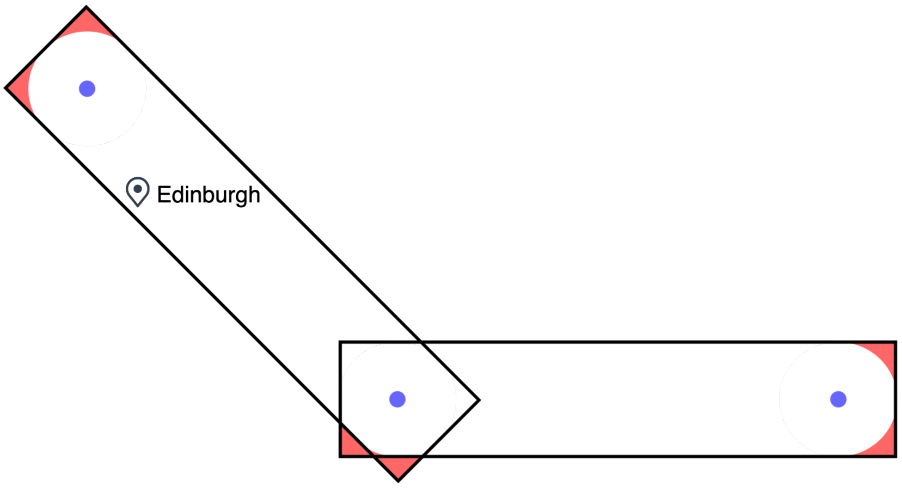


Figure 3.6: Search area for places near to flightpath. Red sections highlight superfluous search areas

3.4 Production deployment analysis

This section will illustrate how the system is capable of being deployed on a large scale, addressing the issues highlighted in section 2.2.

For the sake of simplicity, the project was implemented and tested on a single node. However, this deployment strategy cannot serve thousands of feeders and API clients. As revealed in section 2.3, the largest platforms utilize many machines running dedicated tasks in parallel. Deploying this system using modern production standards would entail only a couple of extra implementation steps.

As is the case with many workloads, the platform would benefit from public cloud deployment. The custom Go code — HTTP ingest, Kafka-MongoDB bridge and GraphQL server — is easily containerized as it has so few dependencies. These containers could run inside a managed Kubernetes cluster, with multiple replicas of each because of the statelessness of the code. They would then make use of the desirable features of Kubernetes, like fault tolerance and load-based autoscaling. Load balancers would need to be put in place in front of the ingest and GraphQL API endpoints to spread requests over the replicas (see figure 3.7). A couple of minor code changes that would need to occur would be assigning the Kafka-MongoDB bridge instances into the same Kafka consumer group so that every ADS-B message is only consumed once, even with multiple consumers for the same topic; and giving each bridge its own ID.

Both the Kafka and MongoDB clusters could be implemented using the cloud platform's managed version for ease-of-use and features. As discussed already, the feeder ID would be the most appropriate key to use for Kafka partitioning and MongoDB sharding, as it would ensure all messages from the same feeder would be processed together.

Combined, these deployment techniques would address the issues with large software platforms discussed in section 2.2.

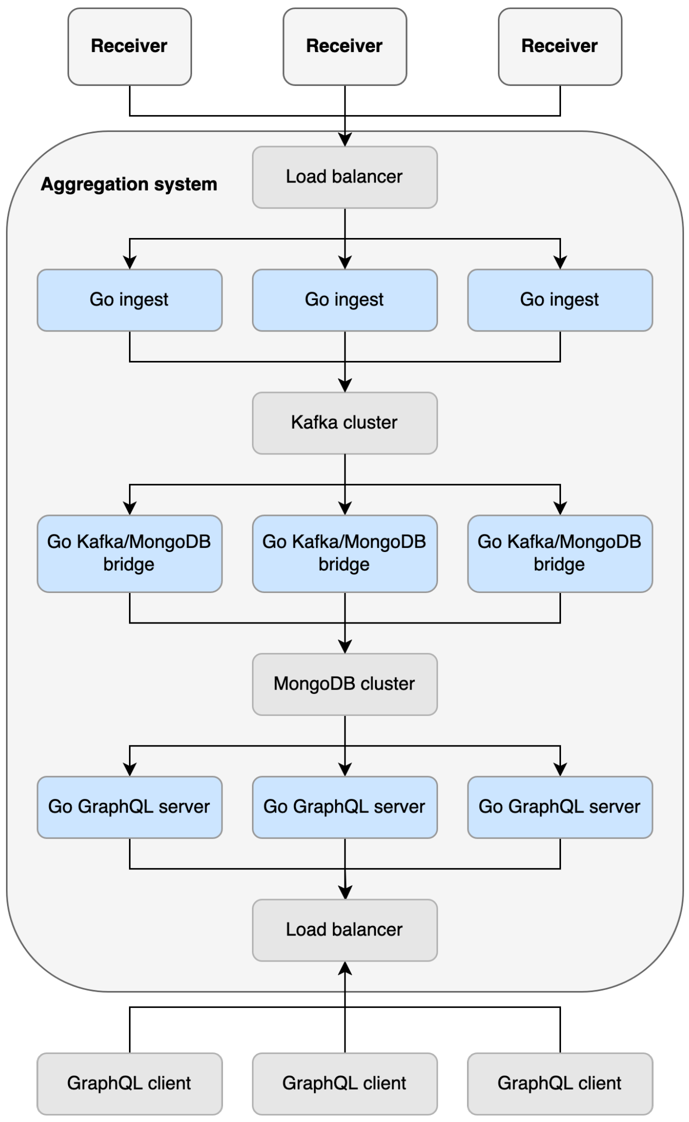


Figure 3.7: Example high performance and availability architecture

```
type Message {
    hex: String!
    now: Float!
    mach: Float!
    flight: String!
    feederUUID: String!
    ias: Int!
    category: String!
    altGeom: Int!
    altBaro: Int!
    gs: Float!
    tas: Int!
    track: Float!
    lat: Float!
    lon: Float!
}

type Flightpath {
    polyline: String
    places: [Place]
}

type Flight {
    hex: String
    flight: String
    polyline: String
}

type Place {
    name: String!
    type: String
    county: String
    country: String
}

type Query {
    messages(hex: String): [Message]!

    flightpathHistory(hex: String!, timeWindow: Int,
    radius: Float!): Flightpath

    flightsOverPlace(latitude: Float!, longitude: Float!,
    timeWindow: Int, radius: Float!): [Flight]

    flightsInConstraints(minAltitude: Int, maxAltitude: Int!,
    minIAS: Int, maxIAS: Int!): [Message]

    flightsAtRisk(minBaroRate: Int, minIAS: Int, maxAltitude: Int):
    [Message]
}
```

Code 3.4: GraphQL API schema

```
var multiPolygon [][][][]float64

for i := 0; i < len(coords)-1; i++ {
   polygon := [][][]float64{{
        {coords[i][0] - radius, coords[i][1] - radius},
        {coords[i][0] - radius, coords[i][1] + radius},
        {coords[i+1][0] + radius, coords[i+1][1] + radius},
        {coords[i+1][0] + radius, coords[i+1][1] - radius},
        {coords[i][0] - radius, coords[i][1] - radius},
    }}
    multiPolygon = append(multiPolygon, polygon)
}
```

Code 3.5: Go source code for multipolygon construction from a flightpath

Chapter 4

Evaluation

This chapter will assess the viability of the completed system as use as a large-scale ADS-B aggregation platform.

4.1 Performance testing

Tests were performed on an Apple M1 using MacOS 12.6.2, Apache Kafka 3.3.2, OpenJDK 19.0.1, MongoDB Community 6.0.3 and Go 1.19.5. All software was kept at its default, untuned configuration, so there is scope for improvement without any changes to the implementation.

A simulated feeder program is used for testing (see code 4.1). The program generates a given number of feeders by utilizing Goroutines — the lightweight concurrency feature of Go — each started at a random time once the program has begun running (since it would be unrealistic for all feeders to submit data at the same time). Feeders submit a JSON ADS-B message that is representative of the data output by readsb.

4.1.1 Throughput

The throughput testing approximated the feeder configuration of OpenSky [13]. In 2023, OpenSky has stored messages at a rate of $\sim$10bn/day, or $\sim$115K/s. With $\sim$5.5K receivers, this equates to 21 messages per receiver per second, or 10 per ADS-B broadcast cycle, which runs at 2Hz at most. With each simulated feeder Goroutine submitting at this rate, over five runs the system achieved a mean message throughput rate of 4873 messages per second. While nowhere near the levels required for the full OpenSky network, with the use of server-grade hardware, the introduction of Goroutines in the ingest and bridge programs (see section 5.3) and the use of multiple nodes, the system could likely scale to such a level. However, this does not account for any of the processing that would be required by large public ADS-B networks, like MLAT and invalid data rejection, which would add significant computational cost.

4.1.2 Latency

Using a modified version of the feeder test program which injects the epoch timestamp in nanoseconds on fake ADS-B messages and uses a fixed aircraft hex string ”testing”, and a Python program (see code 4.2) querying the GraphQL endpoint, over five runs the system achieved a mean unloaded (single feeder) end-to-end latency of 10.5ms with a standard deviation of 0.7ms. Since ADS-B position data only updates with 500ms period (2Hz) when aircraft are airborne, such a low latency makes the single-node deployment more than suitable for low latency flight monitoring of a small number of aircraft — the aircraft updates are less frequent than the system's update rate. According to Confluent's AWS Kafka performance benchmark [8] and Lindvall and Sturesson (2021) [27], it is likely that even in a large-scale production environment this latency will remain well below 500ms.

4.2 Malicious actors

This section will explore potential problems with open-access ADS-B data platforms, any mechanisms the system uses to combat these exploits and more sophisticated solutions that could be implemented as future work.

4.2.1 Feeder de-anonymization

Feeder location privacy could be compromised by a simple attack. With the 550,000 hex ICAO strings found in the OpenSky aircraft database [31], a malicious actor could obtain a copy of all messages in the database using the aircraft messages query. They could then geolocate feeders by analyzing the location of all messages each feeder has received — essentially an inverse of the MLAT performed by existing platforms to determine the location of aircraft based on the received signal strength from multiple receivers with known locations.

A web application firewall could be configured to rate limit these queries but this would be easy to bypass by changing the IP address of the GraphQL client machine. To prevent this from being a viable strategy, users could be rate limited by API key. However, even with a draconian rate limit of one query per 10 seconds, it would take less than a day and a half to query all $\sim$19000 registered aircraft in the UK [6] from a single API key. The only way to completely prevent this would be to omit the feeder ID from all publicly accessible GraphQL query results.

4.2.2 Invalid data

There are two causes for invalid data. Receivers could be incorrectly configured or defective. An attacker might want to submit fake messages to hide the true whereabouts of an aircraft — for example, the movements of an individual's private jet (see section 2.4).

The current implementation attempts to solve the problem of deliberately invalid data by making the system closed-access, requiring authentication by feeder ID. However, this is insecure as a malicious actor could gain access simply by learning an approved feeder's ID. A far more sophisticated approach would be to use asymmetric encryption and store each feeder's ID and public key on the ingest server (or sign the public key using a certificate authority). Messages would only be valid if the public key associated with the source feeder ID can be used to successfully decode them. If feeders are able to prevent the leakage of their private key, an attacker could only successfully submit a fake message by cracking the underlying encryption algorithm.

If the authentication mechanism is broken, without additional processing invalid data would still be accepted as the truth (deliberate or not). By utilising Kafka's stream processing as mentioned in section 2.7, the system could reject messages through comparison against predictive algorithms or feeder quorum, though this would require accepted messages to be received by at least three feeders, effectively reducing the system's coverage to high feeder density areas. These methods would help protect the system from attacks designed to hide the whereabouts of specific aircraft. As mentioned already, because Kafka retains its own persistent storage of the data, if the prediction algorithms were to improve then all rejected data could be reassessed.

```
// Generate random plane identifiers
var planes []plane
for i := 0; i < planeCount; i++ {
        planes = append(planes, plane{hex: randSeq(6),
                                    flight: randSeq(6)})
}

// Delay start of feeding for realistic workload
time.Sleep(time.Duration(rand.Float64() * float64(time.Second)))

client := &http.Client{}

for {
        for _, planeFields := range planes {
            // Fill in a few details in template ASD-B message
            body := fmt.Sprintf(emptyMessage, planeFields.hex,
                        planeFields.flight, feederUUID)

            // Create POST request
            req, err := http.NewRequest("POST", ingestHTTPAddress,
                            strings.NewReader(body))

            ...

            // Sets feeder ID as auth field in HTTP header
            req.Header.Set("Authorization", feederUUID)

            // Send request
            _, err = client.Do(req)

            ...
        }

        // Wait for next broadcast cycle (2Hz)
        time.Sleep(time.Duration(messageRate * float64(time.Second)))
}
```

Code 4.1: Source code of simulated feeder Goroutine (error handling omitted)

```
import requests
import time

# GraphQL server query endpoint
url = "http://localhost:1337/query"

# Query for ADS-B messages with special hex ID "testing"
data = {"query": "{ messages(hex:\"testing\") { now } }"}

while True:
    x = requests.post(url, json=data)

    # Once query response is not empty (a message has arrived)
    if x.text != '{"data":{"messages":[]}}':
        print(time.time())
        break
```

Code 4.2: Source code of Python GraphQL client for latency testing

Chapter 5

Conclusion

This chapter will highlight the achievements of the system and future work that should be conducted to make it viable for large-scale use.

5.1 Objectives met

This project has met its primary goals of creating an alternative ADS-B data aggregation platform which can be fed using real-world receiver installations and queried over a low latency GraphQL API. However, it is not at the stage where it could be used as a replacement for existing platforms — the throughput performance is unlikely to be sufficient due to the single-threaded nature of the implementation (see section 5.3).

5.2 Contributions to aircraft tracking domain

A goal of the project set out in chapter 1 was to create an alternative aggregation platform independent of current solutions. This is important as highlighted in section 2.4. The choices of modern tools and technologies justified in chapter 2 ensures the system will be supported by upstream software for the foreseeable future — there is no requirement on software developed by existing ADS-B platforms.

In comparison to other aggregation platforms discussed in section 2.3, the system presented in this report is suitable for both low-latency and historical queries while maintaining a simple architecture. OpenSky is similar in its ability to handle both workloads, unlike ADS-B Exchange and AIRPORTS DL, but is much more complex. This project could be suitable for smaller teams (fewer systems to maintain) but with a need for more throughput and fault tolerance than a single machine can provide.

The system also provides place data enrichment in its ADS-B data API, with the ability to easily add more sources of data from both the MongoDB database and external systems and APIs.

5.3 Future work

There is a lot of scope for future improvement of the project. The major points are outlined below.

- Implement multithreading: Use Goroutines in the ingest, bridge and GraphQL Go programs. This was not a priority for the proof of concept but would provide a performance uplift by allowing multiple I/O operations from the same instance of the program. The considerations for horizontal scaling discussed in section 3.4 (Kafka consumer groups) would be relevant, except applicable to multiple processor cores and not separate nodes. A possible alternative would be to use a server orchestration technology, like Kubernetes, to run multiple instances of the programs on the same node, though this would have higher overhead because each instance would be its own process and have its own copy of the libraries.
- Implement stream processing: Write programs to conduct stream processing in Kafka for MLAT and invalid data rejection as mentioned in section 2.7. Additionally, support could be added for decoder programs other than readsb by decoding from raw hexadecimal with a stream in the aggregation system rather than on the receiver system. This would remove the reliance on readsb's JSON decoding and allow other programs to feed the platform, as well as reduce processing load on the feeders.
- Implement live fetching of enrichment data: Add functionality to a stream processing instance in Kafka, the Kafka/MongoDB connector or the GraphQL server itself to fetch data additional enrichment data (see figure 5.1). The position of this functionality would depend on the data — if it needed to be bespoke for each query, the fetching would take place in a gqlgen resolver function. If many queries could make use of it, it could be done before or after Kafka, depending on if there was a potential for the data to be reprocessed by using Kafka's persistent storage.
- Implement direct MongoDB queries: Allow for sanitized querying of the MongoDB database for complex historical queries not supported by the GraphQL API. This would perform a similar function to the Hadoop querying shell in OpenSky (see section 2.3.2).
- Implement security mechanisms: Require public-key cryptography for feeder authentication and key-based GraphQL API access as discussed in section 4.2.
- Containerize: As discussed in section 3.4, the optimal deployment strategy would involve the use of containers. This requires the writing of Containerfiles to build images from the source code.
- Performance tuning: Once the system is fully implemented and deployed, load test to determine bottlenecks and resolve with code changes and system configuration (e.g. MongoDB sharding, Kafka buffer size, Linux kernel parameters).

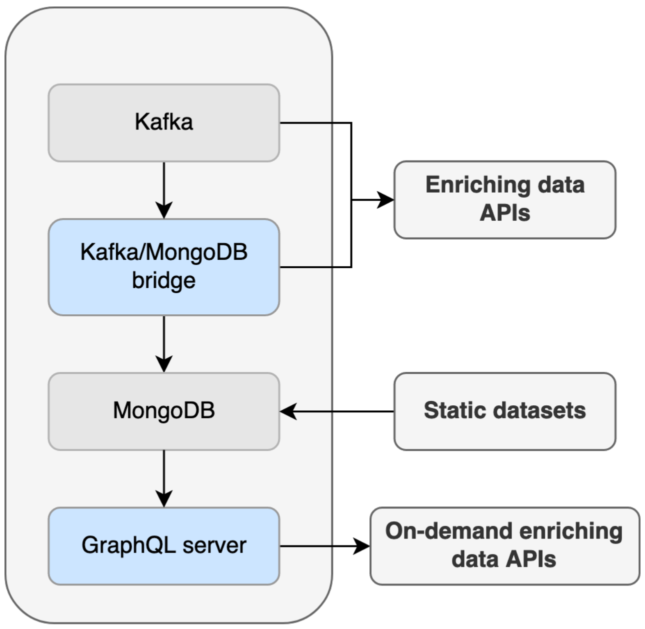


Figure 5.1: Points for integration of enrichment data for GraphQL (see also figure 2.9)

Bibliography

[1] *1. Meet Kafka - Kafka: The Definitive Guide [Book]*. URL: https://www.oreilly.com/library/view/kafka-the-definitive/9781491936153/ch01.html (visited on 03/14/2023).

[2] *14 CFR 91.117 – Aircraft speed.* URL: https://www.ecfr.gov/current/title-14/chapter-I/subchapter-F/part-91/subpart-B/subject-group-ECFRe4c59b5f5506932/section-91.117 (visited on 01/18/2023).

[3] *About Flightradar24*. en-US. URL: https://www.flightradar24.com/about.

[4] *Accessing Data Collected by ADS-B Exchange*. en-US. URL: https://www.adsbexchange.com/data/ (visited on 10/11/2022).

[5] ADSBexchange.com. *ADS-B Exchange Developer Documentation*. original-date: 2019-09-08T02:13:20Z. Dec. 2021. URL: https://github.com/adsbxchange/adsbexchange-documentation (visited on 09/30/2022).

[6] *Aircraft register statistics — Civil Aviation Authority*. URL: https://www.caa.co.uk/data-and-analysis/aircraft-and-airworthiness/aircraft-register-statistics/ (visited on 02/14/2023).

[7] *Apache Kafka*. Apache Kafka. URL: https://kafka.apache.org/ (visited on 02/13/2023).

[8] *Apache Kafka® Performance, Latency, Throughout, and Test Results*. Confluent. URL: https://developer.confluent.io/learn/kafka-performance/ (visited on 01/24/2023).

[9] Austin. *Receiving aircraft ADS-B (position) signals - part 2*. Austin's Nerdy Things. Mar. 10, 2021. URL: https://austinsnerdythings.com/2021/03/10/receiving-aircraft-ads-b-position-signals-part-2/ (visited on 03/17/2023).

[10] *AUTOMATIC DEPENDENT SURVEILLANCE - BROADCAST (ADS-B) OUT EQUIPAGE MANDATE IN INDIA*. Oct. 25, 2018. URL: https://aim-india.aai.aero/sites/default/files/aip_supplements/AIPS_2018_148.pdf.

[11] *Compare MongoDB Vs. Couchbase*. MongoDB. URL: https://www.mongodb.com/mongodb-vs-couchbase (visited on 02/10/2023).

[12] *Comparing Features of Other Go GraphQL Implementations — gqlgen*. URL: https://gqlgen.com/feature-comparison/ (visited on 03/16/2023).

[13] *Coverage & Facts*. URL: https://opensky-network.org/network/facts (visited on 01/27/2023).

[14] Stephanie Crawford. *How Secure Digital Memory Cards Work*. howstuffworks.com. URL: `https://computer.howstuffworks.com/secure-digital-memory-cards.htm`.

[15] *Dictator Alert — Be a citizen of the world. Track dictators.* URL: `https://dictatoralert.org/` (visited on 02/09/2023).

[16] *Doppler Radar — SKYbrary Aviation Safety*. URL: `https://skybrary.aero/articles/doppler-radar` (visited on 03/10/2023).

[17] *Encoded Polyline Algorithm Format — Google Maps Platform*. Google Developers. URL: `https://developers.google.com/maps/documentation/utilities/polylinealgorithm` (visited on 01/19/2023).

[18] *Flight Progress Strips*. URL: `https://www.faa.gov/air_traffic/publications/atpubs/atc_html/chap2_section_3.html` (visited on 01/30/2023).

[19] Iván García et al. "Towards a Scalable Architecture for Flight Data Management". In: *Proceedings of the 6th International Conference on Data Science, Technology and Applications*. DATA 2017. Setubal, PRT: SCITEPRESS - Science and Technology Publications, Lda, July 2017, pp. 263–268. ISBN: 9789897582554. DOI: `10.5220/0006473402630268`. URL: `https://doi.org/10.5220/0006473402630268` (visited on 10/12/2022).

[20] *History of Air Traffic Control — USCA*. en-US. Oct. 2015. URL: `https://www.usca.es/en/profession/history-of-air-traffic-control/` (visited on 10/04/2022).

[21] *Impala Shell Quick Guide*. URL: `https://opensky-network.org/impala-guide` (visited on 10/19/2022).

[22] Institute of Electrical and Electronics Engineers and American Institute of Aeronautics and Astronautics, eds. *OPENSKY: A SWISS ARMY KNIFE FOR AIR TRAFFIC SECURITY RESEARCH — 2015 IEEE/AIAA 34th Digital Avionics Systems Conference (DASC 2015): Prague, Czech Republic, 13 - 17 September 2015*. eng. Piscataway, NJ: IEEE, 2015. ISBN: 9781479989409 9781479989416. URL: `https://lenders.ch/publications/conferences/DASC15.pdf`.

[23] *JETNET Acquires ADS-B Exchange Extending JETNET's Product Offerings*. Jan. 25, 2023. URL: `https://finance.yahoo.com/news/jetnet-acquires-ads-b-exchange-160000127.html` (visited on 02/17/2023).

[24] *Jimmy Jeffs – UK — 100 Years Air Traffic Control*. en-GB. URL: `https://www.atc100years.org/jimmy-jeffs-uk/` (visited on 10/04/2022).

[25] *Just Plane Wrong: Celebs with the Worst Private Jet Co2 Emissions — Insights*. en. URL: `https://weareyard.com/insights/worst-celebrity-private-jet-co2-emission-offenders` (visited on 10/17/2022).

[26] *Limiting Aircraft Data Displayed (LADD) — Federal Aviation Administration*. URL: `https://www.faa.gov/pilots/ladd` (visited on 02/17/2023).

[27] Josefin Lindvall and Adam Sturesson. *A comparison of latency for MongoDB and PostgreSQL with a focus on analysis of source code*. 2021. URL: `http://urn.kb.se/resolve?urn=urn:nbn:se:hj:diva-54652` (visited on 01/24/2023).

[28] Hannah Fry Malina F. *bng-latlon: Converts british national grid (OSBG36) to lat lon (WGS84) and vice versa.* Version 1.1. URL: `https://github.com/fmalina/blocl-bnglatlon` (visited on 11/21/2022).

[29] *MongoDB Kafka Connector — MongoDB Kafka Connector*. URL: `https://www.mongodb.com/docs/kafka-connector/current/` (visited on 02/13/2023).

[30] Rupert Neate and Rupert Neate Wealth correspondent. “Teen monitoring Elon Musk’s jet ‘tracking Gates, Bezos and Drake too’”. en-GB. In: *The Guardian* (Feb. 2022). ISSN: 0261-3077. URL: `https://www.theguardian.com/technology/2022/feb/02/teen-tracking-elon-musk-jet-bill-gates-jeff-bezos-drake-jack-sweeney-tesla-flight-tracker-bot` (visited on 10/17/2022).

[31] *OpenSky Aircraft Database*. URL: `https://opensky-network.org/datasets/metadata/` (visited on 01/23/2023).

[32] John Pauly. *Assignment 4: Automatic Dependent Surveillance-Broadcast (ADS-B)*. URL: `https://web.stanford.edu/class/ee26n/Assignments/Assignment4.html` (visited on 02/07/2023).

[33] *Publications using the OpenSky Network*. URL: `https://opensky-network.org/community/publications`.

[34] *readsb/README.md at dev · wiedehopf/readsb*. GitHub. URL: `https://github.com/wiedehopf/readsb` (visited on 03/27/2023).

[35] Matthias Schäfer et al. “Bringing up OpenSky: A large-scale ADS-B sensor network for research”. In: *IPSN-14 Proceedings of the 13th International Symposium on Information Processing in Sensor Networks*. Apr. 2014, pp. 83–94. DOI: `10.1109/IPSN.2014.6846743`.

[36] Junzi Sun. “The 1090 Megahertz Riddle: A Guide to Decoding Mode S and ADS-B Signals”. en. In: (2021). DOI: `10.34641/MG.11`. URL: `https://books.open.tudelft.nl/home/catalog/book/11` (visited on 09/30/2022).

[37] *Supported MongoDB APIs, Operations, and Data Types - Amazon DocumentDB*. URL: `https://docs.aws.amazon.com/documentdb/latest/developerguide/mongo-apis.html` (visited on 02/10/2023).

[38] *The state of gRPC in the browser*. gRPC. Jan. 8, 2019. URL: `https://grpc.io/blog/state-of-grpc-web/` (visited on 02/10/2023).

[39] Pete Muntean Wallace Gregory. *US government spy planes monitored George Floyd protests — CNN Politics*. en. June 2020. URL: `https://www.cnn.com/2020/06/11/politics/spy-planes-george-floyd-protests/index.html` (visited on 10/17/2022).