FleetFox
Real-Time Tracking for the MBTA

May 8, 2017
Casey Hong¹, Jisoo Min², Hyeyoung Shin³
R06 / 2pm / Melva James

¹ jhong47@mit.edu  
² jisoomin@mit.edu  
³ hyshin@mit.edu
# Table of Contents

1 Introduction 2

2 System Modules 2
   2.1 High-level Objectives 2
   2.2 Bus Control 3
   2.3 Warehouse Servers 3

3 System Design 4
   3.1 Fleet Monitoring 4
      3.1.1 Client-side Bus Monitoring 4
      3.1.2 Server-side Fleet Monitoring 5
   3.2 Failure Detection and Recovery 6
      3.2.1 High Demand 7
      3.2.2 Route Unavailability 7
      3.2.3 Unexpected Route 7
   3.3 Passenger Feedback 8
   3.4 Security 8
   3.5 Communication Protocol 8

4 Evaluation 9
   4.1 Normal Operation 9
   4.2 Scalability 9
   4.3 Communication Overhead 10
   4.4 Frequency Assignment 10
   4.5 Data Transmission 11
   4.6 Server Storage 11
   4.7 Data Accuracy 12
   4.8 Failure Recovery Use Cases 12
   4.9 MBTA Target 12

5 Conclusion 13

6 Acknowledgements 13
1 Introduction

As a modern-day public transit system in a busy city, the MBTA seeks to provide the best possible service to its clients. To this end, the MBTA has optimized for availability by establishing a comprehensive span of service so that its fleet of buses can operate at a reasonable service frequency while covering the entire Greater Boston area. It has also installed ADA-compliant devices in all its buses to improve accessibility. Despite these efforts, however, the MBTA frequently fails to meet its goals regarding reliability and comfort of service.

We propose FleetFox, a lean yet robust real-time tracking system that augments the existing infrastructure to more effectively monitor reliability and comfort. FleetFox emphasizes simplicity and scalability in its design, and pursues an aggressive prevention-based strategy to control congestion and failure during the MBTA’s day-to-day operations. Specifically, FleetFox interfaces with the bus controls and bus sensors to communicate critical information regarding the buses’ punctuality and occupancy levels to the servers, and implements a distributed server management scheme in the data warehouse so that the servers can efficiently infer the health of the bus network at any given time.

2 System Modules

2.1 High-level Objectives

FleetFox’s overarching philosophy is to minimize regret in the long run. This means that its main goals are to collect high-quality, real-time data and analyze trends in this data over time in order to recommend smarter changes to the MBTA’s routes and schedules.

---

---

Figure 1. FleetFox’s modules, interactions, and messages. The bus control collects and stores data, and pushes reports to the server when soft failures are detected. Updated values can be sent from the server to the bus control.

---

---

Monitoring the status of a thousand buses servicing more than a hundred different routes accumulates a massive body of data. To manage this data effectively, FleetFox is committed to the following core principles:

1. Deferring communication and computation until soft failures\(^5\) are reported by the bus control
2. Organizing the servers hierarchically, offloading heavy computation to data-agnostic computers and storing datasets optimally across the servers
3. Prioritizing simplicity and scalability in the design

FleetFox prefers to make data-driven improvements over the long term rather than patching over glitches in the short term, and therefore has a relatively conservative failure recovery model.

### 2.2 Bus Control

The bus control is a 128MB computer that is responsible for three critical tasks: communicating route information to the bus-operator tasks, collecting relevant data from the sensors on the bus, and identifying and communicating events of soft failure to the servers.

The bus-operator interface requires the bus control to supply it with the next three stops on the bus’s route at any given time. To do this efficiently, the bus control loads its route timetable at the beginning of the day (see Section 3.1.2 for further details) and then pushes the information for the very first three stops to the bus-operator interface. Then at each subsequent stop, the bus control pushes the information for only one additional stop.

The bus control follows a fine-grained sensor data collection policy in order to get the best possible estimates for various metrics of interest without having to offload computation to the servers. Some of this data is read in continuously; for example, the bus control keeps a running account of passenger count and transaction data from the payment interface. This passenger count data is only for incoming passengers, so FleetFox accounts for the outgoing passengers by reading in data from the beam sensor. Other sensors are only used as data sources in the event of a soft failure. Video feed from the security camera and lat/lon coordinates from the Global Positioning System (GPS) are only requested when the bus control is preparing a message to the servers.

This data is persisted in temporary storage. The storage scheme is simple: each entry is a key-value pair with all the sensor data collected at a given segment keyed in by the corresponding segment ID. We define a segment as a unique section between two consecutive bus stops on a particular route. Note that two distinct routes may have the same endpoints of a segment, but they will have different segment IDs. Due to the storage constraints of the device, FleetFox flushes out any key-value pairs that are more than an hour old.

The bus control also performs some simple computations to monitor passenger load and to identify soft failures in real-time. Section 3.1.2 describes some of the lightweight algorithms that are used to accomplish this in greater detail.

### 2.3 Warehouse Servers

The server-side of the system architecture is composed of nine machines in the MBTA central warehouse, which are collectively responsible for aggregated data storage, video feed analysis, and any extraneous computations. We define four types of servers in the system: the master, the librarian, the historian, and the dynamic servers.

---

\(^5\) A soft failure is effectively a signal that a bus is under-performing with respect to one of the MBTA’s target goals. It is defined in the system as an event that is emitted by a bus control under one or more of the following situations:

1. The bus is three or more minutes late to a scheduled arrival time.
2. The passenger-to-seat ratio on the bus is greater than or equal to 1.1.
3. Over two thirds of passenger feedback from an integrated smartphone app is marked as “dissatisfied.”
The **master** is in charge of monitoring the fleet throughout the day. It is the only server in the warehouse that can communicate directly with the bus controls via the trunked radio network, and it is the only reliable machine that is immune to the two-minute server failure model. Therefore, it is uniquely positioned to act as a critical intermediary between the bus controls and the other servers. The master delegates computation to the dynamic servers; when the master receives notice of a soft failure from a bus control, it sends the job to an available dynamic server to be processed and stores a copy of the output before sending back a message to the bus control. This communication protocol is described in greater detail in Section 3.5.

The **librarian** stores datasets that have already been curated by the MBTA. This includes census data, bus meta-data, route and schedule data, alternate stop data, and operator data. FleetFox also stores a fast lookup data structure mapping all of the stops from the route and schedule dataset and the alternate stop dataset to a tuple of type (isTransitDependent, nearbyCitizens). The tuple is prepared using the census dataset; the first value is a boolean and the second is an integer. Once the hash table is created, it does not need to be created again. New entries can easily be appended to the hash table when new stops are added, and entries can be efficiently updated with the most recent census data using fast lookups. The master performs read operations from the librarian at the beginning of the day, but we assume that this is done far enough in advance that a two-minute failure won’t affect the overall performance of the system.

The **historian** is an archival database server. At the end of the day, the master writes a summary data packet to the historian containing information about the total load, the load of transit-dependent passengers, and the soft failure events for each bus in operation that day. This data can be read at any time of day to handle any MBTA customer service queries. Because the nature of the reads and writes to the historian are not time-sensitive, the two-minute failure model does not affect its performance. Data is not lost during the failures, so the master or customer can safely wait for two minutes before proceeding with the write or read operation without compromising the integrity of the data.

The **dynamic servers** are data-agnostic machines that are used as computational workhorses in the system. They accept jobs as delegated by the master server, and then return the output of their computations back to the master server for aggregation. The implications of this hierarchical design are twofold. First, it removes dependencies between the buses and servers; since the output is returned to the master, which is fail-safe, FleetFox does not need to spend extra resources for data replication. Second, it gives the system a greater degree of fault-tolerance, because if a server undergoes a two-minute failure and cannot immediately perform a job assigned by the master, the job can simply be reassigned to a different available dynamic server.

### 3 System Design

#### 3.1 Fleet Monitoring

FleetFox keeps state on both the bus controls and the servers in order to perform its fleet monitoring responsibilities. The bus control monitors bus-specific reliability, comfort, and load metrics without offloading any computation to the servers, and only sends data to the servers in the event of soft failure. FleetFox then *infers* the health of the bus and routes on the server-side based on the transmission (or lack of transmission) of soft failure events. In this way FleetFox is able to enforce modularity and simplicity in design, while also ensuring that the MBTA is able to monitor the fleet as a whole at any time on the server-side.

##### 3.1.1 Client-side Bus Monitoring

For comfort and load monitoring, the bus control uses a simple counting algorithm to monitor the number of passengers on the bus in real-time with at least 85% accuracy. The algorithm sums the transactions registered on the
payment interface to derive the number of people entering the bus at each stop. When the bus reaches a stop that is flagged as transit-dependent, a separate counter is also updated. In the pseudocode in Figure 2, the transit-dependent flag is detected by looking up the stop ID in the stop-census hash table, and checking if the boolean stored in the first value in the tuple evaluates to true. If so, the delta value is computed and then added to the transit-dependent counter stored on the bus control.

FleetFox also utilizes both the beam sensors and security cameras in order to efficiently and effectively monitor occupancy levels on the bus. The beam sensor is used to detect the number of passengers that are exiting the bus at each stop, since this information is not captured by the payment interface. Because beam events indicate whether the passenger was entering or exiting, we simply need to check this indicator before decrementing the counter variable accordingly.

```
def count(counter, stop, payment_data, beam_events):
    delta = 0  # transit dependent
    for payment in payment_data:
        counter += 1
        if STOP_CENSUS[stop][0]:
            delta -= 1
    for event in beam_events:
        # beam events indicate whether the passenger was entering or exiting
        if event.direction == 'exit':
            counter -= 1
    return counter, delta
```

*Figure 2.* Pseudocode for the counting algorithm in the bus control.

When the `counter` variable exceeds 1.1 times the bus capacity, the bus control emits a soft failure event. The bus control then reads 5 frames of video footage from the security camera and adds it to the data field of the soft failure message. The message is transmitted to the master server via the system-wide trunked radio network, and then a suite of computer vision algorithms is applied to the frames in order to compute a more accurate value for the passenger load. This value is guaranteed to be at least 95% accurate, and is returned to the bus control so that the counter can be properly updated. Sending video footage through the radio network is costly but is only done when the bus is at risk of failing to meet its comfort requirement. Using this conservative counting procedure, FleetFox is able to facilitate smart data collection that simplifies communications between the bus control and the master server.

In order to help the MBTA measure the value of its bus network, the bus control also keeps an 800 byte transfer buffer data structure in which to persist data about Charlie Card users that have transferred from one bus to another. When the buffer is full, the bus control sends a packet of 100 64-bit Charlie Card IDs to the servers for processing in order to update that day’s count of transferring passengers.

Because both the bus control and the master server have knowledge of what each stop’s arrival times are scheduled to be, FleetFox employs a no news is good news policy for monitoring the reliability of a bus and frequency of service of a route in operation. At each stop, the bus control checks that the bus has arrived at the stop within three minutes of its scheduled arrival time. If it does not arrive within three minutes of the scheduled time, the bus control emits a soft failure event.

### 3.1.2 Server-side Fleet Monitoring

The master keeps several data structures to monitor the fleet in real-time.

For each bus in operation, it stores the bus ID, the identity of the route it has been assigned, and a stop table. The stop table has an entry for each stop in the route, with three fields that correspond to the soft failure constraints and
two fields indicating if the stop is located in a high-density or transit-dependent area. Figure 3 displays a sample entry in the stop table.

FleetFox prepares this nested data structure at the beginning of each day, well before operating hours, by reading in the route and schedule dataset from the librarian and the fast lookup stop-census hash table from the historian. This data is transferred onto the master via the in-house wireless connection. After the data structure is prepared and stored on the master, the master iterates through the entries and transmits the appropriate route timetable onto each bus control using the in-house wireless network.

![Figure 3](image)

**Figure 3.** An entry in the stop table for a bus that is servicing a transit-dependent, high-density stop that has experienced a soft failure in the load and timeliness dimensions, but not the passenger feedback dimension.

As soft failure events are reported to the master throughout the course of the day, the master immediately pushes the data packet to an available dynamic server for processing. For each dynamic server, it stores its identity and its state (available, working, completed).

There are two types of tasks that can be delegated to a dynamic server. The first is passenger count estimation, which applies computer vision algorithms on 5 frames of security camera footage. The second is value to network calculation, which accepts a stream of 64-bit Charlie Card IDs and outputs the number of passengers that had transferred from another bus. This computation is done by reading in the transfer table from the master server, which is persisted on a daily basis and contains the unique identities of passengers that have transferred from one bus to the other, and appending the identities from the stream to the transfer table if they are not already present.

### 3.2 Failure Detection and Recovery

FleetFox defines a soft failure to be an event that is emitted by a bus control under one or more of the following situations:

1. The bus is three or more minutes late to a scheduled arrival time.
2. The passenger-to-seat ratio on the bus is greater than or equal to 1.1.
3. Over two-thirds of passenger feedback from the integrated smartphone application is negative.

This mechanism prevents superfluous information from flooding the servers and also mitigates risk of congesting the system-wide radio network.

Hard failures are identified on the server-side. A hard failure occurs when

1. A bus fails to meet its comfort requirement for more than 5 days within a 2 week timeframe.
2. A bus has failed to meet its scheduled arrival time for one or more stops on more than 5 days within a 2 week timeframe.
(3) A bus has failed to meet its rider satisfaction goals for more than 5 days within a 2 week timeframe.

FleetFox favors a very conservative failure recovery model. In the following three sections we describe several common use cases that require failure recovery, and discuss the tradeoffs that determine how failure recovery is executed in each case.

3.2.1 High Demand

FleetFox’s policy for handling high demand is one of aggressive preventative action. This means that FleetFox focuses on failure detection in order to optimize the span of service and routes in the long term, but does not prioritize failure recovery for fluctuating demand levels in the short term.

When the system is inundated with an unexpectedly high number of passengers. FleetFox reacts by sending more soft failure packets to the servers detailing the passenger count as discussed in Section 3.1.1, and then saves the load metrics for long-term analysis. Aside from collecting more precise data more frequently, FleetFox explicitly opts to take no further action for failure recovery. This design decision can best be understood using a simple cost-benefit analysis; “recovery” can be very expensive, as the number of idle buses required to properly respond is unbounded in the worst case, and with very weak guarantees for rider satisfaction since it would take time for the idle buses to reach the congested points in the city. It is more worthwhile, then, to conduct failure detection as precisely as possible, and then revise the route schedule directly if a trend in congestion is detected on the server-side.

3.2.2 Route Unavailability

FleetFox pursues different strategies for handling route unavailability, depending on the immediacy of the root cause.

In the event that the cause of route unavailability is known ahead of time, such as construction, FleetFox chooses to perform the rerouting through human rather than algorithmic judgment. During the construction period, a system administrator determines a reroute path, using the alternate stop dataset. Rather than attempt to reconfigure the entire network of routes, the system administrator is only responsible for selecting an alternate bus stop for each bus stop that is rendered unavailable by the construction activities. This selection process might be guided by the system administrator’s past experience of traffic patterns, but is mainly driven by the proximity of the alternate stop to the original stop. This has the additional benefit of increasing passenger comfort, as they likely do not have to walk very far to access the reroute path. While the result of this rerouting process may not necessarily result in the most efficient or harmonious use of resources, FleetFox trades efficiency in this case for simplicity because the problem at hand is inherently a short-term issue.

When routes become unavailable due to flash emergencies such as fire or serious accidents, FleetFox sends an idle bus from the warehouse to the affected route(s). We assume that MBTA system administrators are aware of relevant emergencies via other channels (i.e., direct notifications from city police or firefighters) and are thus capable of putting FleetFox in emergency recovery mode. This permits the warehouse to send route data to an idle bus-control. In this data packet, the first stop of the unavailable section of the route is flagged and the idle bus must start its route at that stop. If the fire is spreading and other routes are in danger of being affected, the task of selecting alternate routes or dispatching more idle buses is at the MBTA system administrator’s discretion.

In either case, any outstanding updates are posted on Twitter and the MBTA mobile application to alert users. For long-term periods of unavailability, physical signs are placed at the rerouted stops and their corresponding original stops to remind riders of the change.

3.2.3 Unexpected Route

FleetFox prescribes the dispatch of an idle bus when the MBTA needs to serve an unexpected route. For example, if the Red Line breaks down and requires backup bus service between Kendall/MIT and Park Street, FleetFox will
assign $n$ idle buses a new route that only cover the stops in between Kendall/MIT and Park Street. FleetFox defers to the system administrators when determining the exact value of $n$, as they will have the most pertinent information regarding the traffic levels at the particular time and place of the unexpected route.

3.3 Passenger Feedback

Passenger feedback is collected using an integrated application\(^5\). This is a secure mechanism for passengers to indicate satisfaction or dissatisfaction with each ride. When a Charlie Card user taps onto a bus, the bus control immediately pushes a small data packet containing the timestamp of the payment and the identities of the bus and route. This enables the user to access a simple feedback form. If at the end of the ride the passenger chooses to leave feedback,\(^7\) the bus control appends a new key-value pair with the most recent segment and the feedback value, but not any passenger-specific data to preserve anonymity. Note that passengers can only view information relevant to their transaction, and do not have access to the video feed, operator ID, or other security sensitive data.

FleetFox pursues this method of feedback collection with the assumption that most people are smartphone users and that *long-term clients* of the MBTA are likely have Charlie Cards. Even with these constraints, however, the integrated application proves to be a worthwhile module in the system because it allows FleetFox to collect a considerable amount of user input without adding any serious complexity on the bus control or server-side.

3.4 Security

FleetFox does not persist any information that is specific to a particular individual in the long-term, so it is impossible for an adversary to infer anything about a passenger’s whereabouts by intercepting the data in the system. Feedback is collected in such a way that anonymity is preserved (see Section 3.3 for details) and Charlie Card information is sent to the master server for value to the network computations without any relation to the passenger’s geographical location. In this way FleetFox is able to enforce security in the system.

3.5 Communication Protocol

FleetFox’s communication protocol is built on top of the MBTA’s reliable trunked radio network.

All data packets are formatted in the same way:

<table>
<thead>
<tr>
<th>source</th>
<th>target</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>48 bits</td>
<td>48 bits</td>
<td>$d$ bits</td>
</tr>
</tbody>
</table>

The source address (src addr) is a 48-bit identifier that contains either the bus’s ID or the master server’s network address, which is 0x00...0.

The destination address (dst addr) is a 48-bit identifier that contains either the bus’s ID, the master server’s network address, or the broadcast address.

The data field carries different values, depending on the directionality of the communication. Bus-to-server communication occurs in the event of a soft failure, so the body of the data field will contain a failure code specifying which constraint was violated, as well as the identity of the stop at which the soft failure was incurred. A failure code of 1 signifies a soft failure with respect to comfort, so the data field will be augmented with the current passenger count on the bus control, primarily based on the beam sensor measurements, as well as 5 frames of security camera

---

\(^5\)Recall that an integrated application is one that is downloaded onto a user’s smartphone, and is capable of direct communication with the bus control.

\(^7\)Simplicity is obviously a key design priority here, because a passenger is unlikely to use the application if it becomes an inconvenience. To this end, FleetFox chooses to make the feedback mechanism in the integrated application a *binary decision* -- either positive or negative.
footage. A failure code of 2 signifies a soft failure with respect to timeliness, so the data field will be augmented with the GPS location (latitude and longitude) of the bus control as well as the timestamp of when it actually arrived at the stop. A failure code of 3 signifies a soft failure with respect to passenger satisfaction, so the data field will be augmented with counts of the positive and negative feedback left at that stop.

A failure code of 2 signifies a soft failure with respect to timeliness, so the data field will be augmented with the GPS location (latitude and longitude) of the bus control as well as the timestamp of when it actually arrived at the stop. A failure code of 3 signifies a soft failure with respect to passenger satisfaction, so the data field will be augmented with counts of the positive and negative feedback left at that stop.

Figure 4. An example of what the data field of a soft failure packet might contain if soft failures were detected across all three dimensions.

Another case of bus-to-server communication occurs when the bus control’s transfer buffer is full and the stream of Charlie Card IDs are pushed to the master server. In this case, the data field contains just the 800 byte stream of IDs.

Server-to-bus communication occurs when the servers want to return a more accurate passenger load count to a bus control that has previously reported a soft failure with respect to comfort. In this case the data field will only contain the stop ID and the value of the updated passenger count.

4 Evaluation

4.1 Normal Operation

Under normal operation FleetFox trades a small degree of data accuracy for simplicity and scalability. First, we observe that 100% data accuracy is never possible. Even if we ignored the constraints of the system-wide radio network and chose to constantly monitor all 1036 buses’ security camera frames to measure occupancy levels on the buses, we would only be able to attain 95-98% accuracy. Instead, by using the no news is good news policy and a simple yet effective counting algorithm, FleetFox is able to spend its resources on collecting the most accurate data when the reliability and comfort constraints are in danger of being broken, and then relaxes when things are running smoothly. This allows the system to scale up powerfully, which is likely to appeal to the MBTA.

4.2 Scalability

Scalability is at the heart of FleetFox’s design, and the system would easily be able to accommodate more buses and routes due to its conservative use of the radio network and its compact method of data storage on the server-side.
It is important to note that FleetFox is scalable, but not necessarily flexible. In other words, FleetFox scales well only when the new requirements do not contradict the core strategy of the system. For example, if the MBTA wanted to service each stop every ten minutes instead of every twenty and added several hundred more buses to achieve this goal, FleetFox could manage the influx of new agents without any major issues. However, if the MBTA wanted FleetFox to collect data every 20 seconds from every bus, FleetFox would falter because this fundamentally contradicts its conservative communication policy and the network would be ill-prepared for that level of traffic. Scalability is also limited to some degree due to the MBTA’s stringent budget and hardware constraints. For example, if the MBTA decided to make each route contain 20 times as many stops, FleetFox would struggle with client-side bus monitoring due to the space constraints of the bus controls.

4.3 Communication Overhead

FleetFox’s efficient communication protocol allows the system to collect enough data to analyze trends in failures over time while minimizing the communication overhead as much as possible.

Given that the MBTA bus fleet is currently struggling to meet their reliability standards (according to the data in the MBTA’s 2015 System Report, only 63% of buses met the 75% On-Time Performance standard)\(^6\), FleetFox must inevitably incur some significant amount of communication overhead in order to monitor the fleet in real-time. However, because FleetFox only sends data from the bus controls to the servers when the buses are not undergoing normal operation levels for some reason, FleetFox is able to minimize the overhead considerably.

The overhead can be quantified by estimating that we push the data from the bus controls to the MBTA server around every three bus stops. This means that the master server is constantly accepting data from 991 bus controls with approximately 1.5 data pushes per second.

\[
b = \text{number of active buses} = 991 \text{ buses} \\
s = \text{average number of MBTA bus stops per route} = 8500 \text{ bus stops} / 177 \text{ routes} = 50 \text{ bus stops} \\
p = \text{average percentage of buses with soft failures} = \frac{1}{5} \\
t = \text{average time for a bus to complete a cycle from origin to destination} = 1 \text{ hour} = 60 \text{ minutes}
\]

amount of traffic sent between buses to the MBTA server is \( (b \cdot s) \cdot p^2 / t = 1.5 \text{ data push / 1 second} \).

Thus, while FleetFox may have constant communication overhead in the amount of traffic sent between buses and the MBTA servers, it is the minimal amount of cost necessary for preventing more severe failures. We trade-off this edge-case cost for performance.

4.4 Frequency Assignment

The trunked radio network is capable of transmitting 16 Mbits per second on each of its 10 frequencies. Therefore, frequency assignment is only ever not instantaneous when more than 10 buses try to communicate to the master server at once. In the worst case, all 10 frequencies would be occupied with transmissions that started at the exact same time, and all would contain the largest possible data packet size, which is 9.6 Mbits even including the uncompressed security camera frames. In this highly unlikely scenario, the eleventh bus would have to wait 0.6 seconds before the trunk controller would be able to assign it to an available frequency.

\(^6\)According to real MBTA data, only 52% of the 23 Express routes, 70% of the 15 Key Bus Route routes, 63% of the 127 Local routes, and 83% of the 5 Silver Line routes meet the 75% On-Time Performance (OTP) standard.
4.5 Data Transmission

The real-time data transmission rate in FleetFox can be evaluated by analyzing the size of the data packets that are passed between the bus controls and the master server.

There are 2 general classes of data packets that are sent from the bus control to the master server.

A soft failure packet is in the format of [src,dst,data], where src and dst are both 48 bits.

<table>
<thead>
<tr>
<th>1. Data for comfort failure</th>
<th>2. Data for schedule failure</th>
<th>3. Data for passenger satisfaction failure</th>
</tr>
</thead>
<tbody>
<tr>
<td>a. 1 digit failure code = 1 bit</td>
<td>a. 1 digit failure code = 1 bit</td>
<td>a. 1 digit failure code = 1 bit</td>
</tr>
<tr>
<td>b. Bus stop id = 64 bits</td>
<td>b. Bus stop id = 64 bits</td>
<td>b. Bus stop id = 64 bits</td>
</tr>
<tr>
<td>c. Beam count data = 2 bits</td>
<td>c. GPS location = 64 bits</td>
<td>c. Pos/neg feedback count = 4 bits</td>
</tr>
<tr>
<td>d. Video data = 250 kbytes per frame (5 frames sent total)</td>
<td>d. Timestamp = 32 bits</td>
<td></td>
</tr>
</tbody>
</table>

One data packet: [src, dst, data] = 48 bit + 48 bit + (data)
- Data for comfort failure : 48 bit + 48 bit + (1 bit + 64 bit + 2 bit + 5 * 250 KBytes) = 96 bit + 67 bit + 1200 KBytes = 1.51e-4 Mbit + 9.6 Mbit
- Data for schedule failure : 48 bit + 48 bit + (1 bit + 64 bit + 64 bit + 32 bit) = 96 bit + 161 bit = 2.57e-4 Mbit
- Data for passenger satisfaction failure : 48 bit + 48 bit + (1 bit + 64 bit + 4 bit) = 96 bit + 69 bit = 1.65e-4 Mbit

Time taken for data transmission for each type of data packet:
1. Total comfort failure data packet = 9.6 Mbit
   - 9.6 Mbit * (1 sec / 16 Mbit) = 0.6 seconds for comfort packet to be sent through radio network
2. Total schedule failure data packet = 0.000257 Mbit
   - 0.000257 Mbit * (1 sec / 16 Mbit) = 0.000165 seconds for packet to be sent through radio network
3. Total satisfaction failure data packet = 0.000165 Mbit
   - 0.000165 Mbit * (1 sec / 16 Mbit) = 0.000165 seconds for packet to be sent through radio network

Thus, on average the data transmission rate for real-time data can range from as little as 1.0e-5 seconds to as much as 0.6 seconds per packet.

A transfer buffer packet contains 48 bit + 48 bit + 6400 bits = 812 bytes of data. However, if the buffer is not full at the end of the day, the data that it holds must be transmitted to the master server at the end of the day via the radio connection, in order to get an accurate value to the network estimate for the historian server. In this case, data transfer from the bus control to the servers can take as long as a full working day.

4.6 Server Storage

Server storage is not a limiting factor for the system. The historian is the only real contender to be the bottleneck in terms of server storage because it is the only server in which we continually persist data over time, but since FleetFox only writes summarizing data to the historian, the archival database is incredibly compact.

We illustrate the unlikeliness of running out of space in the servers using a worst-case analysis, with unknown quantities filled in with pessimistic estimates. First, recall that a summary packet contains the 48-bit ID of a given bus,
the 64-bit ID of its route, a field for the total load for the day, a field for the transit-dependent load for the day, and soft failure data for each stop on the route. A soft failure entry includes the failure code, lat/lon coordinates, timestamp, stop ID, and perhaps either a passenger count or feedback value. In the worst case, all 1036 buses would be deployed and each bus would be servicing a route with 24 stops, and every single stop would contain a 512-bit soft failure entry. A single summary packet would take up 1036*(48+64+24*512)=1605.8 kilobytes of space. This means that if we stored this data daily for 5 years, we would only use 0.003 TB of storage. Because this is only 0.03% of the total space on the machine, FleetFox is able to offer an extremely robust guarantee on server storage availability.

4.7 Data Accuracy

FleetFox does not attempt to achieve 100% data accuracy. Rather, it focuses on collecting data that is good enough to detect soft failures with high probability. Passenger count data hovers at 85% accuracy until soft failures with respect to comfort are detected, at which point the computer vision algorithms are applied to the security camera frames to reach 95% accuracy.

The reliability data is guaranteed to be more stable because of the margins that are built into the requirement itself. Because reliability is measured by checking if a bus arrives at its origin, midpoint, or destination timepoint within three minutes of schedule, FleetFox is able to make very accurate estimates due to the very nature of the soft failure system, since it is assumed that the bus is on time unless soft failure events are explicitly reported. The accuracy of this data could be compromised through a security breach, but we note that this is an unlikely edge case.

4.8 Failure Recovery Use Cases

For most cases in the failure model, FleetFox chooses to do nothing in the short-term, and reroute in the long-term. Hard failures are assigned on a biweekly basis, so it could take the system as long as a month to respond to a failure. While this may seem like a bad guarantee at first, we justify this decision by considering the costs of the alternatives. Pulling a bus off its original route and onto a new one jeopardizes the reliability of service for that bus’s route. It also introduces a large amount of unnecessary complexity, because we would have to establish and maintain a hierarchy of preferences for all the buses and routes in order to choose a bus to reroute. Meanwhile, this would only reap marginal benefits since the patrons of the route would have to wait for the rerouting algorithm to pick a bus as well as a path for the bus to migrate to the new route. The riders would already be dissatisfied at this point, so we choose not to spend our resources in a relatively fruitless endeavor. Putting a currently idle bus in operation introduces less complexity and is thus a more viable solution, but we only reserve this for specific cases as detailed in section 4.1. This is because we do not want FleetFox to develop too much of a dependency on the quantity of idle buses that the MBTA has on reserve, as this bodes poorly for scalability.

4.9 MBTA Target

FleetFox ensures that the MBTA will meet all its service targets under normal operation conditions. However, for certain edge cases FleetFox may not be able to ensure that all of the targets are met. For example, the reliability standard will not be met if a large number of buses suddenly break down, since FleetFox systematically chooses to do nothing when buses are in short supply. For coverage, FleetFox could fail if a large construction site causes a bus stop to be replaced with the nearest existing bus stop (as discussed in Section 4.1.3) which could feasibly be out of the 0.5 mile range. FleetFox could also fail to meet the comfort target if there are no patterns to spikes in the passenger count data.

---

9 This is a pessimistic estimate from actual MBTA route data. We are given that routes can range from 4 to 38 miles; MBTA route data suggest that this roughly translates to an average of 12 stops per route. The most densely serviced routes may have up to 24 stops per route, so we use this value in our calculations.
5 Conclusion

Monitoring a large fleet of buses in a busy city generates a massive body of data. FleetFox provides a simple yet scalable solution for real-time tracking that works under the MBTA’s hardware and budget constraints by enforcing modularity in the design and optimally distributing data collection and processing tasks to infer the performance of the fleet on the server-side. While the system is designed to self-correct and optimize itself over the long term, a series of trial runs are recommended to measure general rider satisfaction at the outset. User tests can also help to measure the usability (and therefore the usefulness) of the integrated application before deployment.

6 Acknowledgements

We would like to thank our instructors Melva James, Michael Trice, Rebecca Thorndike-Breeze, and Cathie Yun for their guidance and advice.