Imperial College London

Department of Computing

Make Drones Not Do A Crash: A SpatialOS Story

Author:
Paul Balaji

Supervisor:
Prof. William Knottenbelt

Second Marker:
Prof. Kin Leung

June 18, 2018

Abstract

test post pls ignore

Acknowledgements

ack

Contents

List of Figures

List of Tables

Part I

Foundational Knowledge

Chapter 1

Introduction

Drone technology is becoming increasingly popular. Their agility and ability to be used remotely makes them ideal for a number of use cases in several industries such as film, law enforcement, emergency services, agriculture and commercial delivery[40].

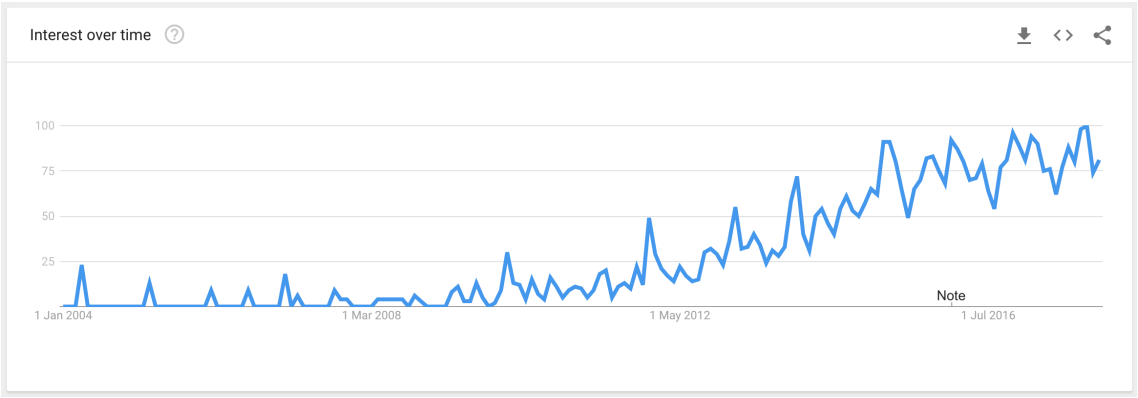


Figure 1.1: Google Trends for "Drone Technology". [24]

Due to numerous advances in technology, drones are quickly advancing to the point where human input is no longer a necessity. This has led to many companies showing interest in integrating drones with their work in the coming years.

Although it may be an engineer's dream for a fully automated world, drones in particular are a harrowing reminder that there are real risks associated with them. There are already several incidents of drones crashing into planes and flying into areas they shouldn't, most notably near Heathrow airport[13].

All of this provides motivation to introduce some form of autonomous air traffic control system to navigate these drones to their respective destinations in a safe manner.

However, prior work has been done on the routing and navigation aspect of such a system[10]. In order for drones to truly take over more aspects of our lives, we must look at how they can provide a tangible benefit to specific use-cases. There is no doubt that simply removing the human element can save costs drastically, and there is none more exciting an application than with physical delivery networks.

The main factors that could prompt higher adoption of drones for delivery networks are cost, delivery speeds, and convenience. As figure 1.2 shows, drones can be both faster and cheaper than existing deliver options that consumers have access to.

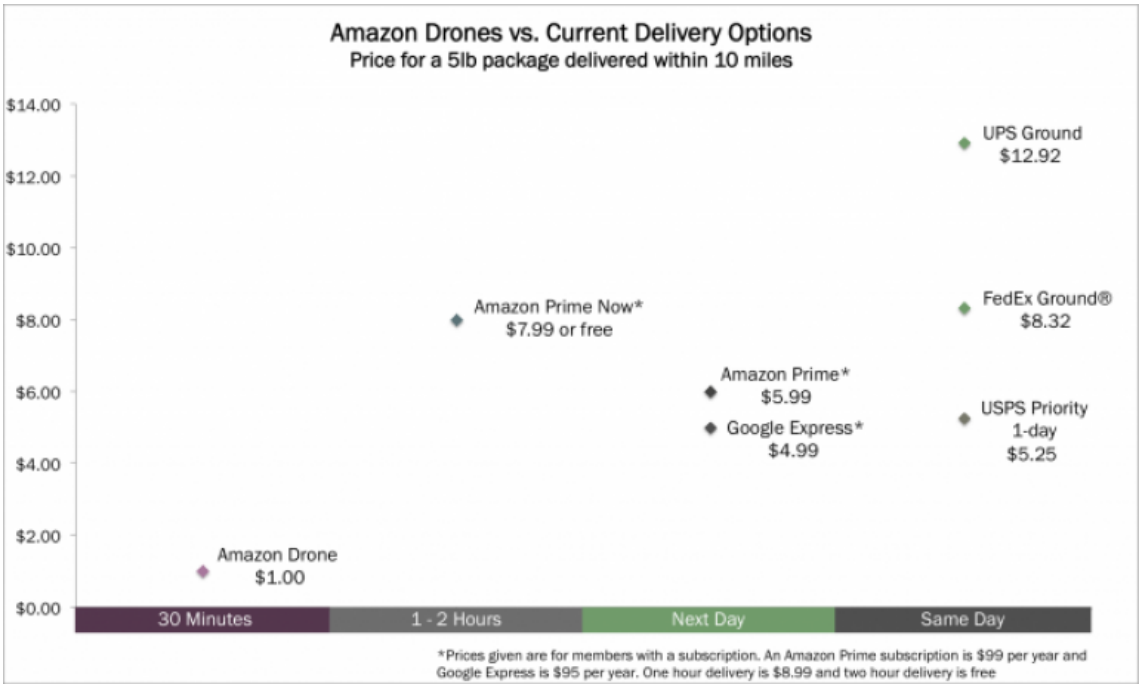


Figure 1.2: How Amazon Air fits into the wider delivery market. [57]

It appears then that Amazon stands to increase their margins considerably if they can successfully pull off their Prime Air initiative[58], but to do this they need to be able to handle the additional problems of scheduling and load balancing massive quantities of drones, on top of the existing routing problems.

In a more general sense, goods delivery networks have a lot to gain by optimising for profit, and to do this better we consider a variable delivery pricing mechanism that factors in the quality of service provided to the user. A high-priority product may warrant a higher delivery fee for quicker delivery but a similarly high penalty for being late.

At the same time, a low-priority product may not have any such penalty, instead opting for a low delivery fee. Being able to factor this new fee model when scheduling tasks and allocating drones may be the next step in the ongoing automation of last-mile delivery[37].

Regardless of how such a model is implemented, it needs to be tested thoroughly before being deployed into production with actual drones. Considering a typical delivery drone could cost around $100 to $500[44], and that there is likely be a lot of failure before a working implementation, there could be several hundreds of thousand dollars worth of losses.

Therefore, our best port of call is to simulate scenarios that are as close to the real world as possible. One tech firm investigating how to produce richer, meaningful and more realistic simulations is *Improbable*[29] - a London-based tech startup actively developing a distributed simulation platform called *SpatialOS*[32].

We aim to leverage the power of SpatialOS to produce novel simulations of a drone delivery network. The simulation will build upon existing routing solutions[10] and would aim to schedule deliveries based on a Quality of Service value curve. Furthermore, we wish to show how the profitability of a drone delivery network with this form of scheduling might compare to delivery networks that utilise a different scheduling mechanism.

Chapter 2

Background

We first provide an insight into new developments in the physical delivery network sector, and then specifically considerations to account for when integrating drone technology in our day to day lives. Additionally we summarise prior work completed by Imperial students on an autonomous air traffic control system for drones.

We then continue to discuss methods of prioritising a delivery network for financial gain. Finally, we give details about Improbable's SpatialOS and the reasoning for using this platform for the drone simulation.

2.1 Future of Delivery Networks

From the ancient days of Assyrian trade in 19 BCE [52], to the modern day online market - the ability to trade goods and services has had a profound impact on the way we live our lives. Thanks to globalisation and the explosion of the internet, we are now able to use a service such as Amazon Prime and get goods delivered within days, if not hours, of purchasing through a simple click. Through it all, physical delivery networks enable this online retail behemoth.

As technology evolves, so too will these delivery networks - and many companies are currently looking into how to leverage upcoming tech[43] in order to generate hype for their business and, more importantly, increase their profit margins. Some promising developments are delivery robots, autonomous cars and, of course, drones.

2.1.1 Delivery Robots

Delivery robots are unmanned devices that are designed to be operated on pavements and cycle lanes at low speeds of about 4 miles per hour[43]. Although the unmanned device market is in its infancy, there are already prototypes hitting the market such as Starship's robot[42], which can hold upto 20-25 pounds, and Dispatch's "Carry"[39], that holds up to 100 pounds.

Figure 2.1: Starship's robot confronting a senior citizen. Image from Engadget[51].

Due to their on-the-ground nature, delivery robots are primarily designed for low-footfall suburban neighbourhoods, closed residential areas or campuses. They would be scheduled through an Uber-like application that may allow users to track the location of the robot, for the customer's knowledge, and to also unlock the robot once it reaches its destination[27].

Thanks to their similarities to autonomous cars, they may have regulatory advantages compared to drones because much of the legislation coming through for autonomous vehicles can apply to delivery robots as well. However, there are profitability concerns because each robot can cost hundres of dollars and there is at this stage there is a high risk of failure, damage and even theft.

Despite some drawbacks, it is evident these robots are designed for short journeys in the so-called "first-mile delivery" sector - with exciting use cases being the delivery of flowers, groceries and possibly medication.

2.1.2 Autonomous Vehicles

Once the de-facto icon of science fictional futurism, autonomous cars are now a close reality. Companies such as Tesla are doing pioneering work in bringing self-driving cars[53] to the general consumer. Meanwhile, other businesses like Uber are experimenting with autonmous taxi services[23].

In fact, Google has obtained a patent for an autonomous delivery truck [26] with different compartments that could be unlocked by validated package recipients - much akin to an Amazon Locker-on-wheels. If not this, these vehicles could be used as an autonomous courier to pick-up locations that users could visit to retrieve their packages.

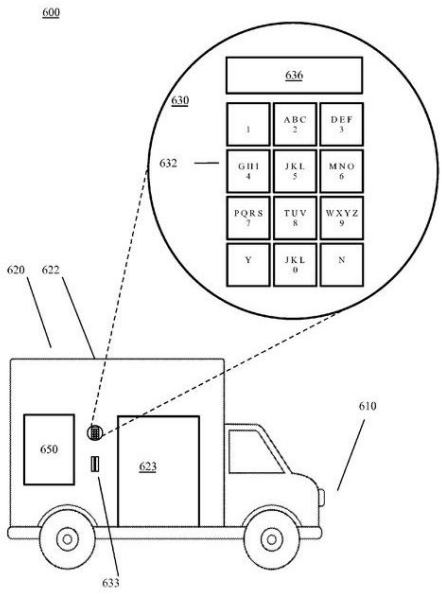


Figure 2.2: Illustration of Google's Delivery Truck. [47]

Although current regulations dictate that a driver must remain present in the vehicle in the event of software performing differently to expections, this is a necessary precaution whilst driverless systems are still being tested - with further amendments to these laws in sight once these systems can be deemed safe for public use. [16]

2.1.3 Drones

The concept of drones has been around for decades, particularly in the military, but only in recent times has the technology evolved to a point where anyone can walk into a consumer electronics store and purchase a drone for their own private use. Though these are manned, it is admirable that many witness the benefits of drones in film/television[56], agriculture[36], policing[12] and other exotic activities[50].

Being able to fly allows drones to not be held hostage to traffic jams or congestion - after all, the sky is a far greater expanse than even the most intricate of road networks. Not only that, they can literally travel "as the crow flies" - allowing them to access remote and previously hard-to-reach areas.

Despite several near-misses [18] and accidents[11] at just Heathrow airpot alone, engineering teams across the world are working on ways to improve drone visibility to its surroundings - one such technology being automatic dependent surveillance-broadcast (ADS-B) [19]. Drones with ADS-B work by determining their own geolocation and velocity in order to broadcast this back to the global network.

ADS-B is already widely used in aircraft, enabling flight-tracking services such as flightradar24 to exist[22], but recent developments have allowed the ADS-B units to be compact and cost-effective enough to be suitable for drones. As more drones adopt this technology, instead of having to get instant updates from a drone directly, autonomous drone traffic control systems could make use of the shared global network for their scheduling and routing calculations.

Over time, an increasing percentage of the world's drones will likely join these shared autonomous systems. This could enable different delivery networks to manage their own fleets as their own air traffic controllers, all the while making use of the shared pool of readable data to identify, intercept and avoid drone collisions at the earliest stage in their pathfinding as possible.

2.2 Amazon Prime

Amazon Prime is a paid subscription service that Amazon offers to users, giving unlimited access to their video and music streaming services as well as providing free same-day and next-day delivery on a large majority of goods sold on their platform. The Prime service applies to products that are either sold directly by Amazon or fulfilled on behalf of a third-party merchant.

2.2.1 Fulfilment by Amazon

One of the services Amazon provides to merchants is the *Fulfilment by Amazon* service, where goods are held at Amazon depots until a user purchases a product. The fee paid by the merchant for Amazon to take care of delivery depends on the duration of time the product is held by Amazon as well as the type of package and weight of the product.

1. FULFILMENT FEE *(flat fee per unit based on dimensions and weight)*

To use the chart, simply take **your product's dimensions (in centimeters)** and its **Outbound Shipping Weight* (in grams)** and compare them with the first row in the table below. If your product's package dimensions (any side) or Outbound Shipping Weight exceed any of the maximum values listed, move to the next largest tier (row).
***Outbound Shipping Weight (gm) = your unit's weight + our Packaging Weight**

Packaging Type *(Packaging Weight - grams)*	**Dimensions** *(cm)*	**Outbound Shipping Weight*** *(g)*	**Fee**
Small Letter[1] *(8 g)*	≤ 23 x 15.5 x 0.4 cm	0 - 100 g	£0.60
Large Letter[1] *(25 g)*	≤ 30 x 22 x 2.4 cm	0 - 250 g	£0.80
Small Envelope *(20 g)*	≤ 20 x 15 x 1 cm	0 - 100 g	£1.09
Standard Envelope *(40 g)*	≤ 33 x 23 x 2.5 cm	0 - 100 g	£1.21
		101 - 250 g	£1.34
		251 - 500 g	£1.54
Large Envelope *(40 g)*	≤ 33 x 23 x 5 cm	0 - 1,000 g	£1.77
Standard Parcel *(100 g)*	≤ 45 x 34 x 26 cm	0 - 250 g	£1.73
		251 - 500 g	£1.79
		501 - 1,000 g	£1.84
		1,001 - 1,500 g	£2.26
		1,501 - 2,000 g	£2.48
		2,001 - 3,000 g	£3.32
		3,001 - 4,000 g	£3.42
		4,001 - 5,000 g	£3.42
		5,001 - 6,000 g	£3.47
		6,001 - 7,000 g	£3.47
		7,001 - 8,000 g	£3.55
		8,001 - 9,000 g	£3.55
		9,001 - 10,000 g	£3.55
		10,001 - 11,000 g	£3.56
		11,001 - 12,000 g	£3.56

Note: Unit weights, dimensions, and other measures used to calculate fees are as determined by Amazon and are subject to variations based on packaging. Dimensions include item packaging and may change based on packaging type. **[1]Small and Large Letter:** If your products are priced under £9, weigh less than or equal to 250 g (including packaging weight), have dimensions less than or equal to 30 x 21 x 2 cm and are listed on Amazon.co.uk, we have a new FBA Small and Light programme for such products.

Figure 2.3: Fulfilment fees for standard-size items on Amazon UK. [6]

An incentive for merchants to join this scheme is to attract customers with the same-day and next-day delivery that Amazon Prime offers. This allows them to piggy-back off the existing delivery network that Amazon has developed for significantly lower fees (Figure 2.3) than if they were to store the package and source a distributor for themselves.

2.2.2 Amazon Prime Air

In December of 2013, Amazon announced their intentions to research and develop their own drone delivery service dubbed *Amazon Prime Air* [1]. Amazon intends this service to deliver packages within 30 minutes of an order being placed, with a maximum package weight of 5 pounds (approximately 2.27 kg).

Following this, in November of 2015 the company enlisted the help of Jeremy Clarkson to give a closer look at a more developed drone prototype [2], describing how Prime Air would fit into the lives of modern families. Just over a year later, Amazon released a video demonstrating a fully autonomous delivery using Prime Air [4].

Figure 2.4: One of Amazon's Prime Air prototypes. [3]

One of the many roadblocks to scale this technology out across the country is regulatory approval. Since drones are a new technology that improve in leaps and bounds each year, authorities and enterprises are careful to make hasty judgements that either hamper drone adoption or cause civilian harm. To this end, Amazon are also suggesting proposals to how airspace can be best utilised by drones. [5] [7]

2.3 Drone Considerations

Drones stand to be a revolutionary part of our lives as we welcome the new, incoming era of automation. However, to be practical there are a few key concepts one must understand to ensure that they remain a help and not a hinderence to mankind.

2.3.1 No Fly Zones

No Fly Zones (which we will abbreviate as NFZs) are geographical areas where a drone is not allowed to enter or fly at any altitude. Examples of these may include Hyde Park, Buckingham Palace, airports and military locations. Typically these are static obstacles that will always remain a NFZ, however we could also consider cases when they could be created dynamically.

Suppose there was an issue of national security, it would be beneficial for the security and intelligence services to set up a temporary NFZ around areas where they deem a risk so that they can conduct their own operations without worry of external parties interfering with the situation.

2.3.2 Other Drones

Naturally we would want to make sure that we avoid colliding into other drones as well. In a perfect system, a single air traffic controller would have totalitarian dominance over all drones that take to the skies - however this is not *Black Mirror* and we have to assume that there will always be drones that this system will not be able to control or predict.

Nonetheless, a system can ensure that it navigates drones under its control as best as it can, avoiding these external drones and other rogue drones that may be flying with the sole intention of causing problems to others.

2.3.3 Manned Aviation

Manned Aviation is any form of airborne transport with humans on board. This would include passenger jets, private jets, helicopters and other miscallaneous vechicles. For simulation purposes, we can consider these to have straight paths from a start to an end because relative to the zig-zagging of drones - they are effectively straight.

It is paramount to avoid collisions with manned aviation because there is a high risk of human calamity, in addition to the bad press and financial costs associated with such an air traffic incident.

2.3.4 Toll Zones

Toll Zones are geographical areas where a drone has to pay an extra fee to pass through at any altitude. It is a very similar idea to the one that led to Congestion Charges being applied to much of central London. By restricting certain areas only to those who are willing to pay the fee to use the airspace, it reduces the density of air traffic in a specific area.

These charges could also be used as an incentive to reduce pollution in the toll zone, and the additional revenue generated by the toll fees could be used towards the drone-related systems in place within the zone.

2.4 Autonomous Air Traffic Control (AATC)

2.4.1 What is AATC?

During the Autumn Term of the 2016-17 academic year, several students undertook a group project in association with Microsoft and Altitude Angel to produce an autonomous air traffic control system. The goal of the project was to safely navigate drones from their start to their end goals, whilst avoiding obstacles and taking the shortest possible path to minimise battery use.

2.4.2 Technical Overview

AATC has a simple client-server architecture, where drones connect to the REST service to send their telemetry information and goal waypoints every second. The server sends back direction recommendations to navigate the drone to its destination.

Because it would have been too expensive to trial the system on actual drones, a test bench was also implemented that would simulate the drones polling the server and responding to its recommendations. This test bench produced a set of simulation data, that is then available to view on the AATC visualiser. [9]

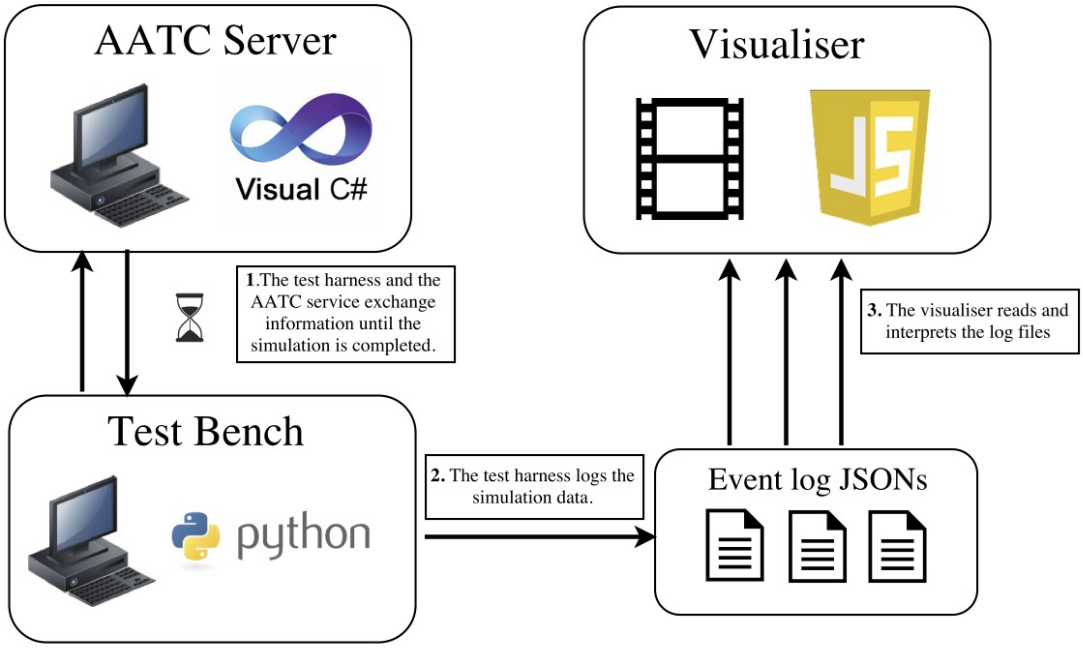


Figure 2.5: Technical Overview of AATC. [10]

The system was designed with three challenges in mind. The first challenge was to route drones from their origins to their destinations, and secondly, to ensure they avoided collisions with both static and dynamic obstacles - such as the ones mentioned in Section 2.3. The last challenge is to design the system in such a way that it would be able to scale to hundreds and thousands of drones.

To this end, a *Global Layer* was built to deal with static obstacles (such as NFZs) by calculating a path for the drone around NFZs before it begins its journey. The second problem was tackled with a *Reactive Layer* that handles non-static obstacles (such as manned aviation and other drones). This is the layer that would be providing the real-time updates to drones as they poll the server every second.

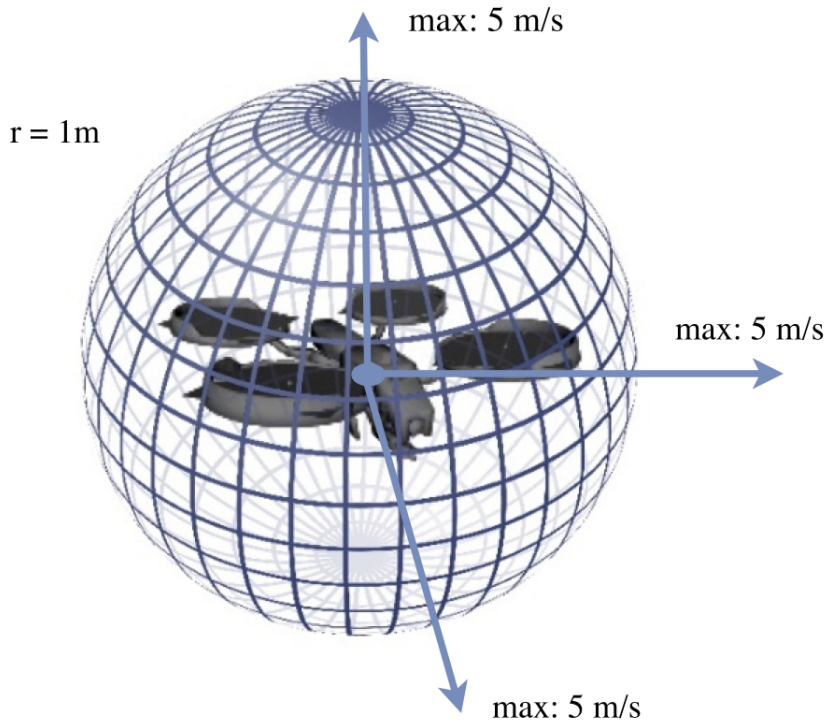


Figure 2.6: The default AATC drone model. [10]

Only one drone type was used in AATC, with its specification outlined in Figure 2.6. It has a radius of 1m and a maximum velocity in any direction of 5 metres per second.

2.5 The Global Layer

The Global Layer holds the static representation of the world so that given a start and end, it can compute an optimal set of waypoints that a drone should follow - thereby dealing with AATC's path-finding problem.

2.5.1 Dijkstra's Algorithm

Dijkstra's algorithm is a popular algorithm to find the shortest paths between nodes in a graph. When using a co-ordinate grid system, each coordinate could represent a node in a graph and thus the algorithm can also be used to find the shortest path between a source and destination.

However, in the real world, we also have to consider the cost of computation and potentially make use of heuristics in order to return the shortest path given a limited amount of time. Especially in the drone case, we want to compute a good path as quick as possible in order to let the drone proceed along its way.

2.5.2 A* Algorithm

Enter, the A* algorithm. This algorithm is a generalisation of Dijkstra's algorithm that reduces the number of nodes explored during the search process by use of a heuristic - typically a minimum distance to the destination. A benefit over Dijkstra in particular is that it considers the distance already traveled into account, which aids the heuristic mechanism.

Naturally, by searching less nodes on a graph, less computation is performed and therefore A* has better performance than just using a pure form of Dijkstra's algorithm.

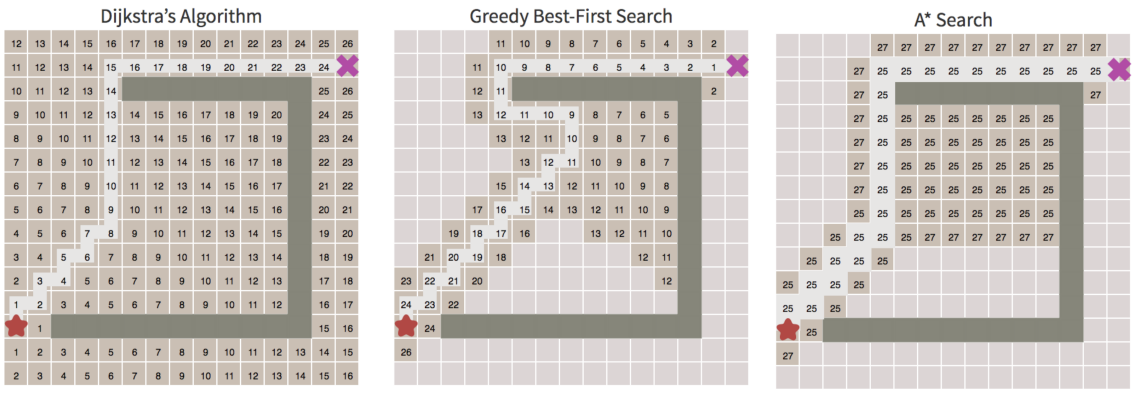


Figure 2.7: Comparing popular search algorithms. [10]

But for all these positive aspects, one must remember that our use for this algorithm is to compute paths in a real world, which is not necessarily split up into a nice, clean grid. So we turn our attention to a modification of the A* algorithm: Theta*.

2.5.3 Theta* Algorithm

The Theta* algorithm is an any-angle pathfinding algorithm based on the A* algorithm that introduces Line of Sight (LOS) checks, which means that each jump from node to node in the returned path can be at any angle and not just up, down, left or right. This neat addition allows Theta* to be capable of finding near-optimal paths with a runtime comparable to A* [55].

Being able to get as short a path as possible is absolutely vital in the drone use-case because they have limited flying time. After all, their batteries can only last so long before they need to be recharged. Therefore, it is important to ensure drones travel as little distance as possible in order to maximise the number of times they can be used between charges.

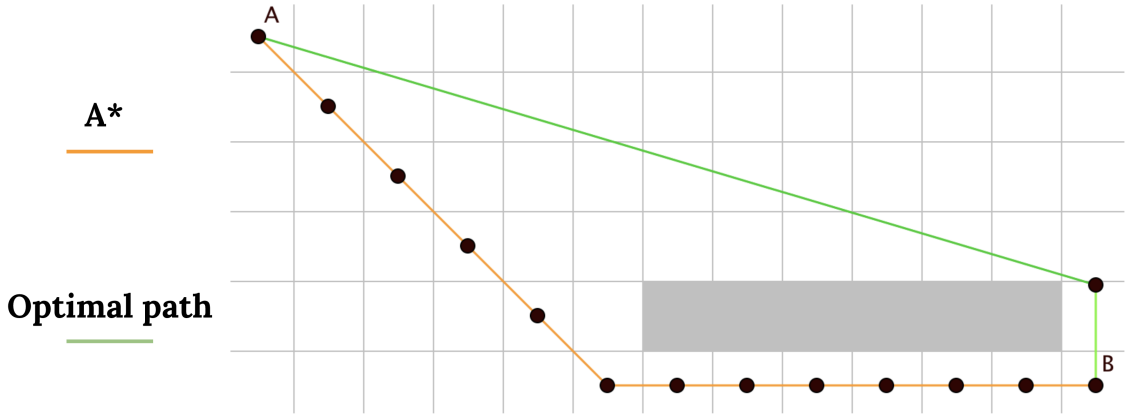


Figure 2.8: A* vs Optimal Path. [10]

2.5.4 Lazy Theta* Algorithm

A further optimisation is to reduce the number of LOS checks performed, as found in another paper by the original authors of the Theta* algorithm[49]. This algorithm assumes every node is in line of sight of its parent, and the LOS check is only performed once the child node is expanded on. If this turns out to be false, then the algorithm defaults to a typical A* approach. Reducing the number of LOS checks therefore improves the algorithm's overall performance.

2.5.5 Conclusions

As seen on Table 2.2, A* has far better execution times than both Theta* variants but when we get to Table 2.1 it is evident that Lazy/Theta* produces better paths because the total distance travelled by drones is fewer. Therefore, Lazy Theta* was chosen as the core algorithm for the Global Layer.

Test Case	A*	Theta* and Lazy Theta Star
Queen Threat	6.11	5.98
The Imperial Tunnel	10.83	10.22
The Great London Beehive	52.63	49.77
The Nightmare of Hyde Park	118.4	115.32
The Great Wall of Imperial College	16.37	15.46
Multi Drone collision	15.27	14.82

Table 2.1: Total path distances for all drones (in km). [10]

Test Case	A*	Theta*	Lazy Theta*
Imperial Tunnel	118	141	124
	120	138	132
	121	137	129
Imperial Tunnel Mean	119.7	138.7	128.3
The Great London Beehive (GLB)	859	1318	1166
	916	1486	1075
	894	1555	1201
The GLB Mean	889.7	1453	1147.4
The Nightmare of Hyde Park (NHP)	9771	28436	21330
	9687	27970	24786
	10118	28273	21609
The NHP Mean	9858.7	28226.4	22575
The Great Wall Of Imperial College (GWIC)	394	913	838
	447	910	840
	416	912	841
The GWIC Mean	419	911.7	839.7

Table 2.2: Execution times for pathfinding algorithms on AATC test cases. [10]

2.6 The Reactive Layer

Given the list of waypoints that the Global Layer generates, the Reactive Layer's task is to provide the right speed and direction to drones whilst taking into consideration any dynamic obstacles that the drone may face, such as manned aviation, other drones and potentially dynamic NFZs.

2.6.1 Artificial Potential Fields (APF)

A novel way to approach this layer is to create an Artificial Potential Field (APF) in the area the drone operates in, and give each point in this field a potential[59]. By having a low potential for the destination and large potentials at obstacles, the drone simply has to move in a way to get to point of lowest potential. It is analogous to a magnet in a magnetic field, repelled by obstacles and attracted to its destination.

Although one might argue that the Global Layer is not needed anymore because the Reactive Layer solves the pathfinding problem, there are a few issues that arise with this idea. As Figure 2.10 shows, a U-shaped object poses a problem because the drone may be attracted to the destination and then quickly repelled by the obstacle, and then back to being attracted - and this cycle is seemingly endless.

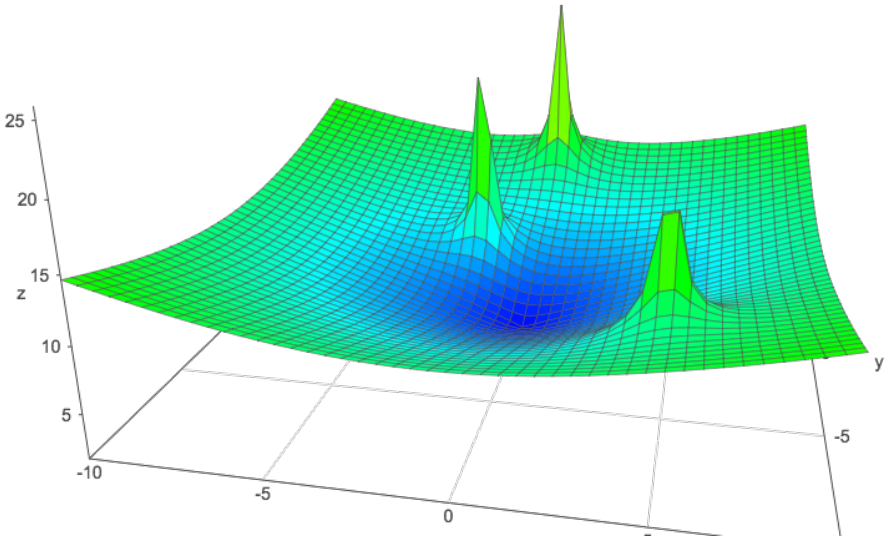


Figure 2.9: APF with obstacles (green peaks) and a destination (blue trough). [10]

By introducing a random element, as in Rotating APF, the object is repelled in a slightly different direction each time to make it out of the trap and eventually reach its destination. Even this method, however, does not actually guarantee that the drone makes it past the U-shaped obstacle because it depends on how the random element behaves and also whether the drone has enough battery to be loitering for long.

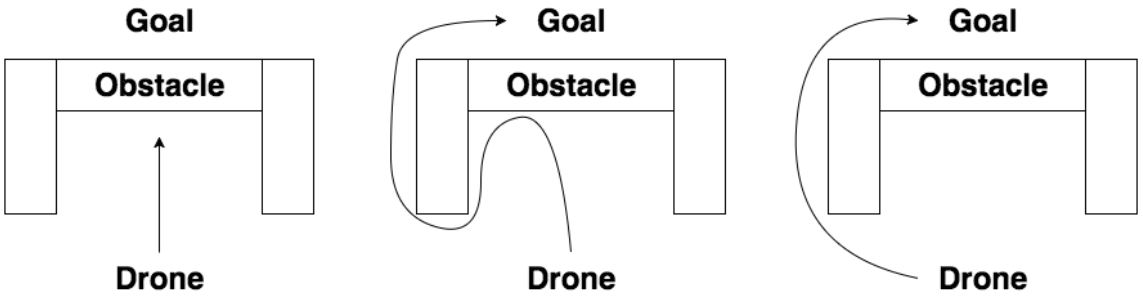


Figure 2.10: Pure APF vs Rotating APF vs AATC Implementation. [10]

Clearly, integrating both layers proves most fruitful - as the waypoints generated by the Global Layer are used to quite literally navigate around the problematic aspects of the Reactive Layer. This integration that was implemented in the AATC project.

2.6.2 Equations for APF

The full set of equations used to obtain the recommended unnormalized velocity v_{raw} of the drone is:

$$U_a = \rho_a d_{goal}$$

$$U_r = \begin{cases} \rho_r \frac{1}{d_{obst} - d_{safe}} & d_{obst} \leq d_{influence} \\ 0 & otherwise \end{cases}$$

$$U_{ret} = \rho_{ret} d_{last}$$

$$U = U_a + U_r + U_{ret}$$

$$\boldsymbol{v_{raw}} = \nabla U$$

where ∇U is the gradient of U, and U is the magnitude of potential at a point[10]. Since ∇U is the recommended velocity, it can be computed from first principles:

$$\nabla U(x,y,z) = \begin{pmatrix} \frac{\delta U(x,y,z)}{\delta x} \\ \frac{\delta U(x,y,z)}{\delta y} \\ \frac{\delta U(x,y,z)}{\delta z} \end{pmatrix} \approx \begin{pmatrix} U(x+1,y,z) - U(x,y,z) \\ U(x,y+1,z) - U(x,y,z) \\ U(x,y,z+1) - U(x,y,z) \end{pmatrix}$$

The recommended velocity can now be computed by first calculating the potential at 4 points. To calculate the potential at a given point, one must find the sum of:

- U_a - attraction potential using euclidean distance to next waypoint.
- U_r - repulsion potential using euclidean distance to nearest obstacle.
- U_{ret} - return potential using euclidean distance to point from last time step.

To allow drones to gently come to their goals instead of shooting past and potentially missing, the recommended speed is calculated by taking the minimum of the drone's max speed and the distance to the goal:

$$speed = min(max\_speed, d_{goal})$$

The gradient ∇U is then normalised to be a unit vector before being multiplied by the speed to compute the final velocity.

2.6.3 Genetic Algorithm

By paying close attention to Section 2.6.2, we can identify several undefined constants: ρ_r, ρ_a, ρ_{ret} and $d_{influence}$. The balance between them is key, because if ρ_a is too high relative to ρ_r then the drones ignore obstacles and die on impact, but if ρ_r is too high then they will oscillate between objects and never reach their goal. After much fiddling about with "magic numbers", a genetic algorithm was employed to provide the optimal values for the specific drone model.

Genetic algorithms are a metaheuristic inspired by the process of natural selection, and they are a way by which high-quality solutions can be generated for optimisation problems[46]. In the case of AATC, the aforementioned constants need to be fine-tuned in order to produce an APF model that, to its best ability, does not return absurd, erroneous velocities. See Figure 2.11 as an example.

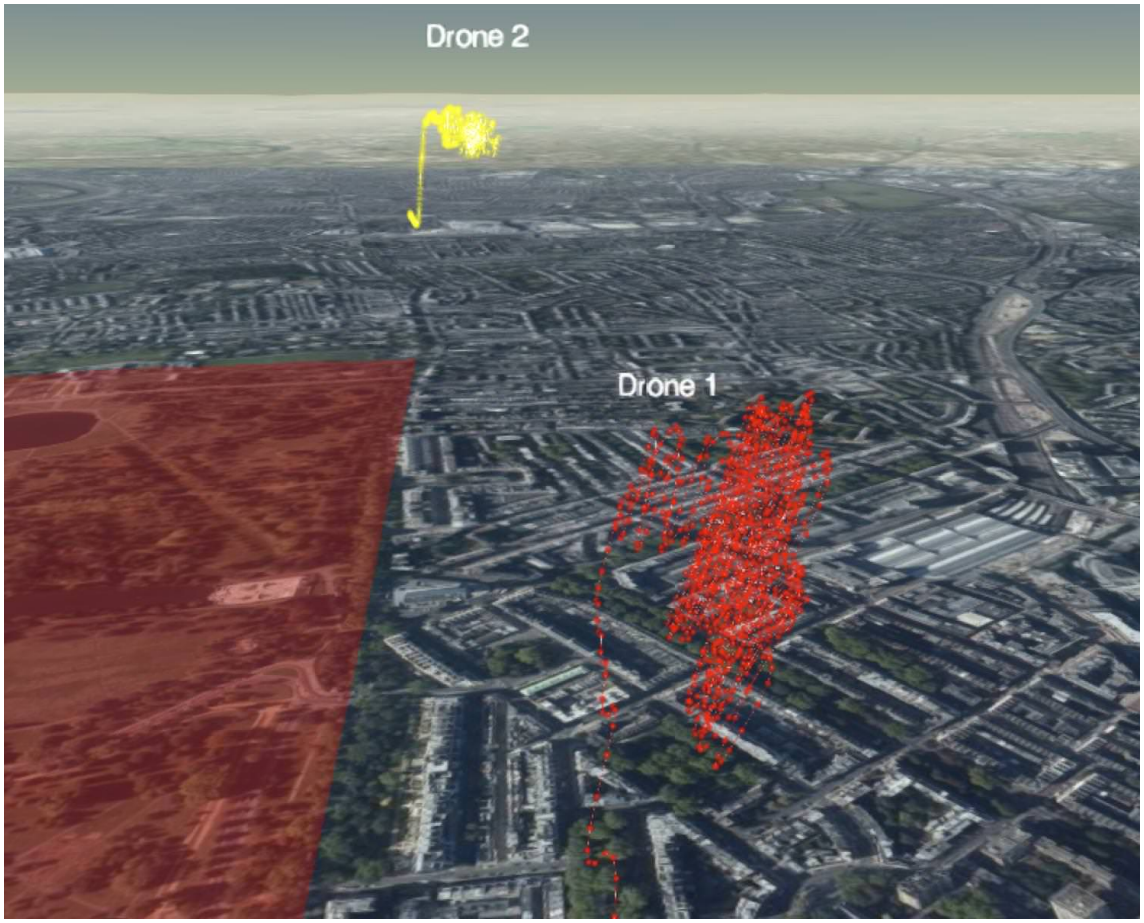


Figure 2.11: Drone paths after increasing repulsion constant by 3 orders of magnitude.

Effectively, an initial range of values (see Table 2.3) is provided and the genetic algorithm cycles through all of these, letting these constants compete against each other to see which set of constants produces the simulation with the lowest cost. This cost is determined from a cost function that returns the remaining battery life of a drone if it reaches its destination, or a relatively huge number otherwise.

$$cost = \sum_{testcase} \sum_{drone} \begin{cases} batteryUsed & if\ drone\ reached\ destination \\ 100000 & otherwise \end{cases}$$

Constant	Min	Max	Step	Total
Attraction	0.8	1.1	0.1	3
Repulsion	100	500	100	5
Return	0.1	0.7	0.2	4
Influence distance	300	600	100	4
Population				240

Table 2.3: Initial configuration of the genetic algorithm for AATC. [10]

2.6.4 Conclusions

In the end, the genetic algorithms only improved the simulations by about 2%[10] which could mean that the original range of constants was already a good set to work with. However, there is still scope for even greater improvements if the genetic algorithm was to be run with a greater number of test scenarios, bigger range of values to cycle through, and generally larger simulations.

2.7 Scheduling Algorithms

In this section we look at a few well known CPU scheduling algorithms[14]. Although CPU scheduling is not part of this project, being able to understand a few different approaches to the problem will give insight into how to schedule drones later on. However, there are two key difference between drone and CPU scheduling.

The first is that CPUs can time slice different tasks whereas once a drone is scheduled onto a task, it has to complete the whole task before being able to pick up a new one. Secondly, there are vastly more drones that could be running at a point in time compared to the number of cores in multi-core CPU, leading to far greater parallelism.

2.7.1 First Come First Serve (FCFS)

As the title says, this algorithm operates as a FIFO queue where tasks are scheduled in the same order that they arrive. Some might consider this the fairest form of scheduling, since it does not discriminate against any task, but simply prioritises the one that arrived soonest.

2.7.2 Shortest Job First (SJF)

This approach looks at the time each incoming task would take to complete, and puts the shortest task at the front of the schedule. While this means that small tasks get completed quickly, a constant stream of small tasks could mean that any heavier tasks never get scheduled or completed.

2.7.3 Priority Scheduling

Priority scheduling is a more general case of SJF, since each task is now given a priority - defined either internally or externally - and tasks with the highest priority are scheduled first. In SJF, the inverse of the estimated job time is used as the priority.

2.7.4 Multi-Level Feedback Queue (MLFQ)

MLFQ is an extension of an ordinary multi-level queue, where there are several queues of tasks to be completed and tasks are scheduled from non-empty queues in a round-robin fashion. A benefit of this approach is that each queue could be running different scheduling algorithms that better fits the priority of that queue.

The “feedback” element of MLFQ allows tasks to be moved from one queue to another, depending on changes of circumstances. For example, if a queue has been left in a low priority queue for long enough, it may be moved to a higher priority queue - thus improving the chance that all incoming tasks get scheduled at some point in time.

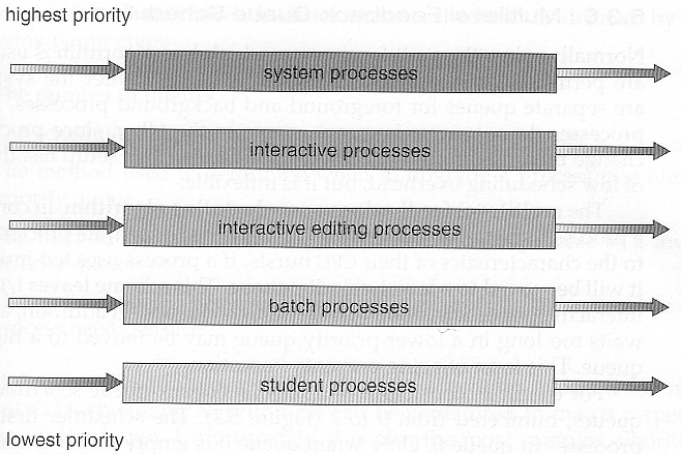


Figure 2.12: Multiple levels of queues, scheduled in a round-robin fashion. [14]

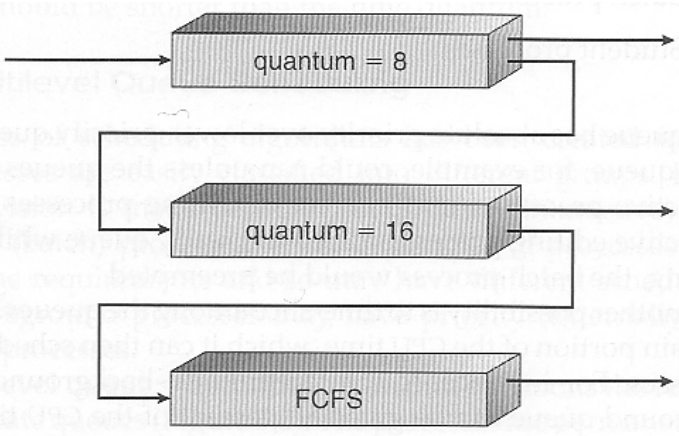


Figure 2.13: Arrows indicate tasks can be moved up one queue to another. [14]

2.8 Time Value

Thus far, we have shown that whilst there is great scope to utilise drones for delivery of physical goods, the existing implementation of AATC only touches upon the routing aspect - how to get from point A to point B. Even in all the test cases, the drones' start and end points were hand-picked by the developers[10] to give an insight into how AATC would operate in a variety of scenarios.

To gain greater understanding of how drones can positively impact industries, simulations need to be performed which take into account the financial aspect of their use case. For example when it comes to drone delivery networks, we want to ensure that the drones not only reach their destination quickly, but that they also get scheduled and utilised in the appropriate manner to maximise the profitability of the network.

2.8.1 Time-Value of Money

As described by Investopedia [35], the *Time-Value of Money* is "the concept that money available at the present time is worth more than the identical sum in the future due to its potential earning capacity". This draws from the idea that investors would prefer to receive the same amount of money sooner rather than later, in order to obtain interest from the sum or reinvest it for greater growth.

2.8.2 Time-Value of Data

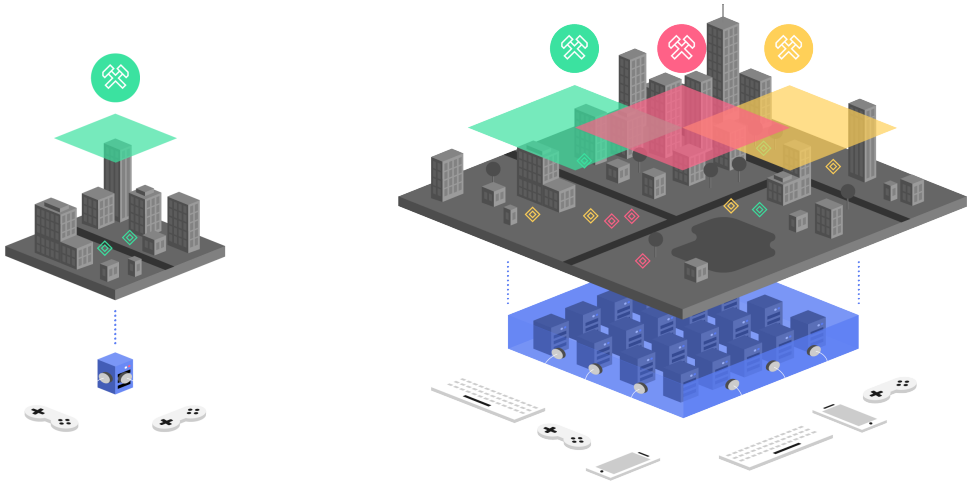
By a similar train of thought, there is the concept of the *Time-Value of Data* - whereby the value to a business of some data decays over time and it is best to gain insight from it as soon as possible. This idea is often used in business intelligence, where companies may have large sets of unstructured data that need to be processed and analysed to derive value for the business. For example, a retailer would rather know what its users are interested in now rather than a month ago, so that they can tailor their sales and offers to leverage customer interest.

2.9 SpatialOS

SpatialOS is a platform, created by *Improbable*[29], for running massive-scale simulated worlds. In their own words[32]:

> SpatialOS is a cloud-based computational platform that lets you use many servers and engines to power a single world. The platform coordinates a swarm of micro-services called workers, which overlap and dynamically reorganize to power a huge, seamless world. The platform also lets you handle a huge number of concurrent players across different devices in one world.

Although most commonly used for large distributed game worlds, SpatialOS has been used for simulating cities[48], the backbone of the Internet[34] and a model of the brain[33]. It is an ideal platform for simulating "independent entities that have a location in space"[33].



(a) The current approach. (b) How SpatialOS is different.

Figure 2.14: Comparing approaches to multiplayer. [32]

Anybody is able to sign up and use SpatialOS to build games using either the Unity3D or Unreal game engines. Alternatively, one can use the C#, C++ and Java SDKs to create custom workers to be ran on SpatialOS deployments.

A benefit of SpatialOS's Unity and Unreal SDKs is that creating a sole game client, without manually writing network code, is almost enough to get multiplayer built-in to the game for "free", i.e. taken as granted.

2.9.1 Unity SDK

Of the two game engine integrations, the Unity SDK is arguably the most stable because it has been in development for far longer than the Unreal SDK, and it is not in beta. From prior industrial experience, the Unity integration is also easier to implement quick prototypes with and iterate on.

2.9.2 Abstraction

SpatialOS adopts an **Entity-Component-Worker** model.

- **Entities** are anything in the world that have a position.
- **Components** define state and how other entities interact with them.
- **Workers** are micro-services that simulate components of entities in the world.

2.9.3 Developer Tools

There is extensive logging of performance metrics, which is a fantastic starting point for scaling up a simulation[17]. In addition to this, there is an active developer **Forum** - where SpatialOS developers and Improbable engineers meet to help other SDK users solve their problems[28].

Naturally as the public SDKs have matured, so too have the public-facing developer tools. Currently, there is an **Inspector** to view the location of entities and the values that their components have at a given time, in either a local or deployed simulation.

2.9.4 Layered Simulation

By virtue of games having several dynamic systems all interlinking with each other, it is possible to create layered simulations on SpatialOS. This works well with the concept of Global, Reactive and Zonal layers that have been mentioned already in Sections 2.4, 2.5 and 2.6. Furthermore, as SpatialOS is distributed by nature, it is the ideal platform to create and execute massive, layered, distributed simulations.

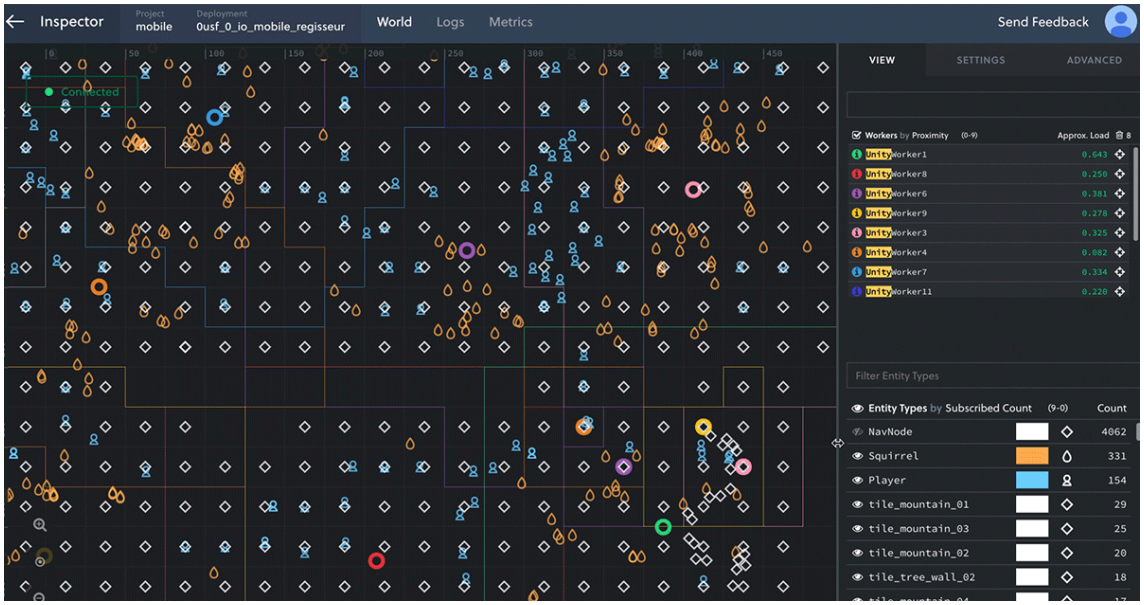


Figure 2.15: The Inspector demonstrating a deployment of the iOS Demo, *Quest*. [31]

Part II

Skeleton Implementation

Chapter 3

Translating AATC to SpatialOS

Before even thinking about simulating orders and scheduling deliveries, there must be a backbone in place that allows drones to get from a start to end point. To this end, the pathfinding and collision avoidance capabilities of AATC (Sections 2.4-2.6) will be the backbone of the simulation.

3.1 Basic Interaction Model

However, AATC was designed as a simple master-slave system such that the server assumed total control over each drone in the system. To produce a scalable delivery network, modifications have to made that allows entities to be decoupled and not limited to the constraints imposed by the existing AATC model.

3.1.1 Drones

As a testament to hardware and software improvements over time, modern consumer drones are now fully capable of avoiding obstacles [20] and navigating to GPS locations on their own [21]. By taking this into account we can have drones responsible for their own movement, leading to greater autonomy in the simulated delivery network, in turn enabling the system to scale up to bigger areas with vastly more drones than AATC.

In the AATC case, drones would have to ping the server every second to request a suggested velocity for the drone. This means that the controller would have to compute new velocities for every single drone it controls, every second.

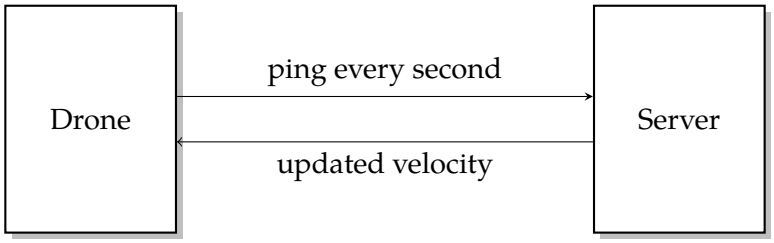


Figure 3.1: Drone-Server Interaction in AATC.

This is a lot of effort that can be offloaded to the drone such as in our proposed model, where we simply provide drones their next waypoint and let them navigate to it by themselves. The only pings to the server now would be to request a new waypoint or to retrieve an updated list of nearby static obstacles.

This new Drone-Controller dynamic means that drones need to be responsible for avoiding dynamic obstacles and other physical entities all by themselves.

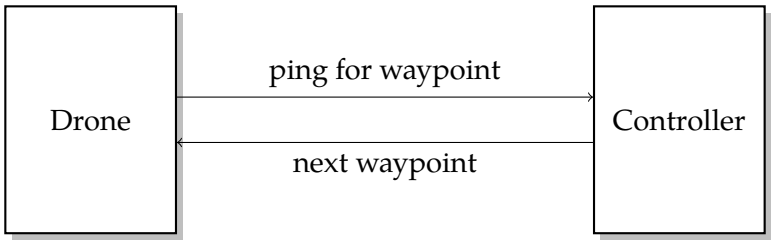


Figure 3.2: Updated Drone-Controller Interaction.

Essentially, the AATC Reactive Layer has been split from the server and will now reside on each drone. The drone must periodically ping a controller for details of the nearest static obstacle, and combine this information with data from its own sensors in order to effectively avoid both static and dynamic obstacles.

3.1.2 Controllers

Much like an air traffic control tower, controllers exist as entities responsible for guiding drones in a particular region of the world. As the collision avoidance Reactive Layer will now run on the drones, the controller will be running the pathfinding Global Layer and keep a bitmap representation of the world.

Since drones only know about their next waypoint, it is the responsiblity of the controller to know which waypoint a drone last visited and to return the correct next waypoint when requested, which could involve checking if the drone is actually at the waypoint it claims to be at. The controller should also return information on the closest static obstacle to the drone at a given point.

3.2 Representing Entities

3.2.1 Unity SDK

As we are working with the SpatialOS Unity SDK, we must become familiar with the concept of a *Prefab*, an asset type that can be used as a template to spawn new object instances. In addition to prefabs, *MonoBehaviours* are scripts added as a prefab component that are then attached to an object instance of the prefab.

For our simulation it is immediately clear that we need to create Drone and Controller prefabs, as they are two entity types that will be interacting with each other. Even if they do not possess any physical or graphical components, they require representation in the simulated world.

A Drone prefab would require MonoBehaviours to handle movement and collision avoidance, whereas a Controller prefab will contain scripts to deal with waypoint management, pathfinding and world representation.

3.2.2 SpatialOS Components

SpatialOS, like many game engines, works as an Entity-Component-System (ECS). The way to define these the components that entities in the world can have is through the *Schema*. The schema is used to generate code that can then be used by prefab scripts in order to update the SpatialOS state of objects.

Each component requires a unique, explicitly defined component ID to distinguish it from other components. Then optionally, we can define properties, events and commands. Properties hold the state of a component, events are used to trigger intra-entity actions, and commands are request-response operations that indicate how different entities communicate with each other.

Listing 1: Example Schema - Health

```
package improbable.example;

type DamageResponse {}
type DamageRequest {
  uint32 amount = 1;
}

component Health {
  id = 1234;
  uint32 health = 1;
  command DamageResponse damage(DamageRequest);
}
```

When spawning an entity, we create a *template* of a defined prefab and add the components that we wish to be on that entity, setting the initial state of component properties in the process. In the case of a Health component we may wish to set *health* of a player to 100, but 20 for little monsters and perhaps 200 for boss fights.

SpatialOS is then tasked with spawning an entity using the template, with the ability to set callbacks for both successful and failed creation attempts.

Listing 2: Creating a Cube Template

```
public static Entity CreateCubeTemplate()
{
  return EntityBuilder.Begin()
    .AddPositionComponent(...)
    .AddMetadataComponent(entityType: SimulationSettings.CubePrefabName)
    .SetPersistence(true)
    .SetReadAcl(...)
    .AddComponent(new Rotation.Data(...))
    .Build();
}
```

Listing 3: Spawning a Cube Object

```
var cubeTemplate = EntityTemplateFactory.CreateCubeTemplate();
SpatialOS.Commands.CreateEntity(PositionWriter, cubeTemplate)
  .OnSuccess((obj) =>
    Debug.LogFormat("Created␣Entity␣{0}", obj.createdEntityId))
  .OnFailure(() => Debug.LogError("Unable␣to␣spawn␣Cube."));
```

Chapter 4

Implementation of Core Entities

4.1 Controller

A Controller has three distinct components to it which all operate together, two of which are the Global Layer and corresponding Bitmap. A *GridGlobalLayer* script would be used to generate a simple path from start to destination when requested. On initialisation, it would need to populate and update the Bitmap component with details of all the No Fly Zones in the that controller's region of the world.

GridGlobalLayer itself will be called by *ControllerBehaviour*. This MonoBehaviour is responsible for overseeing the overall operation of a Controller. Spawning drones, calling the pathfinding component, and updating the drone-waypoint mappings are all examples of tasks that this "master" behaviour would have to handle.

Figure 4.1 describes how the three components slot next to each other, each performing its own set of tasks but maintaining separation of logic and state.

4.1.1 No Fly Zones

No Fly Zones are represented in a similar way to AATC, as a list of float vectors with two vectors to track opposing corners of the zone's bounding box. An initial list of No Fly Zones is passed into the Global Layer upon creation of the component, with scope to add and remove from the list if one so wishes.

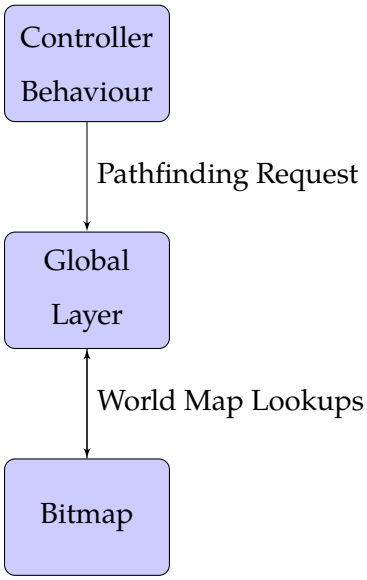


Figure 4.1: Example Component Flow.

Listing 4: NoFlyZone Schema Type

```
type NoFlyZone {
  improbable.Vector3f bounding_box_bottom_left = 1;
  improbable.Vector3f bounding_box_top_right = 2;
  list<improbable.Vector3f> vertices = 3;
}
```

To complement the C# class that SpatialOS generates from this, we have implemented a static class (Appendix A.1) containing methods that are necessary for integrating No Fly Zones with the Global Layer and Bitmap components. These are checks to see if:

- a given point is within a zone's bounding box
- a given point is within a zone's polygonal area
- a given point is within any zone from a given list

4.1.2 Pathfinding and Waypoints

Once the No Fly Zone data is initialised, it is the controller's job to generate a list of NFZ-avoiding waypoints that a drone will follow to get from start to destination. The controller needs to maintain a list of waypoints for each drone, as well as the next waypoint that each drone will need to go to.

This is achieved by keeping a mapping of a drone's ID to it's DroneInfo type, which contains a list of waypoints and the index of the last sent waypoint.

When a drone reaches a waypoint it calls the *request_new_target* command on its assigned controller, which then locates the DroneInfo of said drone. The *next_waypoint* field is incremented and the next waypoint is subsequently found and returned to the drone. If the field value is greater than or equal to the length of the list, the drone has clearly reached the final waypoint, its ultimate destination.

Listing 5: Basic Controller Schema

```
enum TargetResponseCode {
  SUCCESS = 0;
  WRONG_CONTROLLER = 1;
  JOURNEY_COMPLETE = 2;
}

type TargetRequest {
  EntityId drone_id = 1;
}

type TargetResponse {
  improbable.Vector3f new_target = 1;
  TargetResponseCode success = 2;
}

type DroneInfo {
  int32 next_waypoint = 1;
  list<improbable.Vector3f> waypoints = 2;
}

component Controller {
  id = 1200;
  uint32 max_drone_count = 2;
  map<EntityId, DroneInfo> deliveries_map = 3;
  bool initialised = 4;
  Vector3f top_left = 5;
  Vector3f bottom_right = 6;
  command TargetResponse request_new_target(TargetRequest);
}
```

4.1.3 World Bitmap

In the original AATC implementation, the bitmap was represented as a byte array with each cell representing a 25m x 25m area of the world. Since the only information stored in the bitmap is the presence and proximity to NFZs, the array is sparse and therefore better represented as a map from cell number to GridType.

Listing 6: Bitmap Component and GridType Enum

```
enum GridType {
  OUT = 0;
  IN = 1;
  NEAR = 2;
}

component BitmapComponent {
  id = 1204;
  improbable.Vector3f top_left = 1;
  improbable.Vector3f bottom_right = 2;
  int32 width = 3;
  int32 height = 4;
  int32 grid_width = 7;
  int32 grid_height = 8;
  map<int32, GridType> grid = 5;
  bool initialised = 6;
}
```

The benefit of this implementation is that only relevant data is stored in the map, with failure to find a cell in the map implying that the corresponding area of the world is not in or near a No Fly Zone. Since every controller will have a Bitmap component, this design improves memory efficiency and provides more options to the controller. It could simply reduce its memory footprint, deal with larger areas, or reduce the cell size to represent the world with a higher precision.

4.1.4 Assisting the Reactive Layer

Although collision avoidance runs on the drone, it can not do the job properly without knowing the closest point to it on a NFZ. This is where the controller comes in and asks the bitmap to return the closest NFZ point. Since NFZs extend across all altitudes, the drone's altitude is used as the y-coordinate of the closest NFZ point.

This means that there is now a fourth component to the Controller entity, existing just to deal with requests from drones for the nearest static obstacle. This is shown as the *get_nearest_obstacle* command, which returns the *APFObstacle* used by drone-side collision avoidance code.

Listing 7: Reactive Layer Request Handler

```
void GetNearestObstacle(RequestHandle handle)
{
  Vector3f nfz = bitmap.nearestNFZPoint(handle.Request.location);
  APFObstacleType type = APFObstacleType.NO_FLY_ZONE;
  if (nearestNoFlyZone.y < 0)
  {
    type = APFObstacleType.NONE;
  }
  handle.Respond(new ObstacleResponse(new APFObstacle(type, nfz)));
}
```

Listing 8: Reactive Layer Schema

```
package improbable.drone;

import "improbable/vector3.schema";

enum APFObstacleType {
  NONE = 0;
  MANNED_AVIATION = 1;
  DRONE = 2;
  NO_FLY_ZONE = 3;
  HIDDEN_OBSTACLE = 4;
}

type APFObstacle{
  APFObstacleType type = 1;
  improbable.Vector3f position = 2;
}

type ObstacleRequest {
  improbable.Vector3f location = 1;
}

type ObstacleResponse {
  APFObstacle obstacle = 1;
}

component ReactiveLayer {
  id = 1205;
  command ObstacleResponse get_nearest_obstacle(ObstacleRequest);
}
```

4.1.5 Drone Spawning

For simplicity at this early stage, controllers just generate a path between two random non-NFZ points in the world. They spawn a drone at the first random point and despawn each drone once it reaches its final waypoint. For testing purposes collisions were initially disabled as the drones just blindly moved from waypoint to waypoint without any collision avoidance.

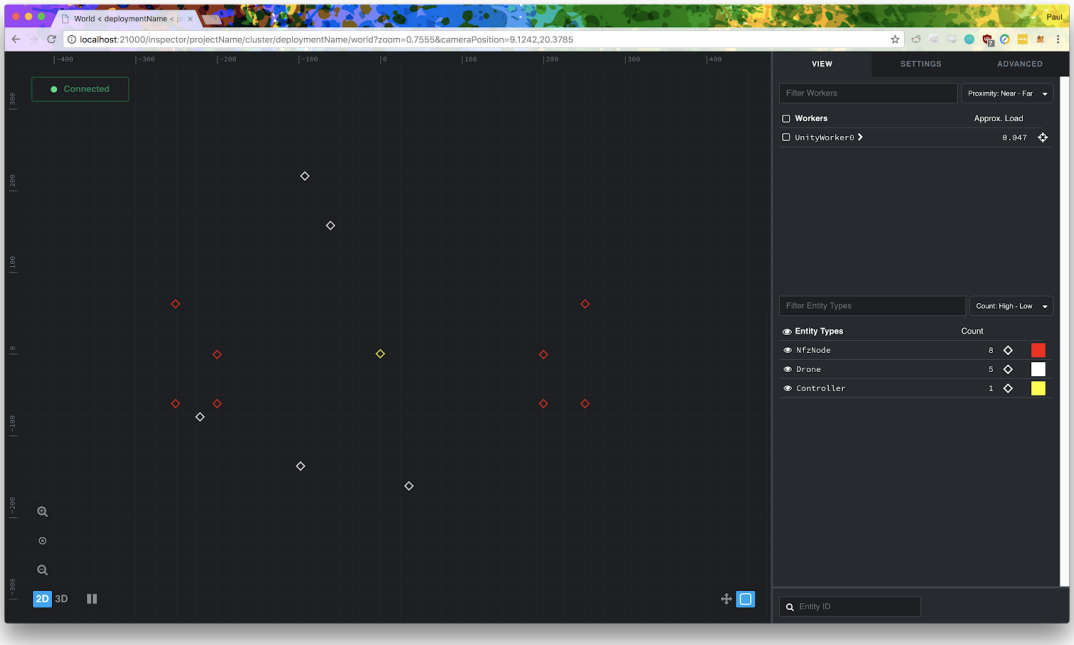


Figure 4.2: Testing basic controller functionality.

In Figure 4.2, we check if the controller returns correct waypoints in order by spawning drones either side of the NFZ and observing if the flight path avoids the obstacle.

4.2 Drone

Compared to a controller, a drone is a far less complex entity. It has one main component to maintain state of the drone's next waypoint, current direction, target request status, and maximum speed. The core loop of the *DroneBehaviour* script executes every second, updating SpatialOS with the drone position and then either recalculating direction or requesting a new target, depending on the drone's status.

Listing 9: Main Drone Loop (once a second)

```
void DroneTick()
{
  if (simulate)
  {
    SendPositionUpdate();
    if (DroneDataWriter.Data.droneStatus == DroneStatus.MOVE)
    {
      apf.Recalculate();
    }
    if (DroneDataWriter.Data.targetPending == TargetPending.WAITING)
    {
      requestNewTarget();
    }
    float distanceToTarget = Vector3.Distance(target, transform.position);
    if (DroneDataWriter.Data.targetPending == TargetPending.REQUEST
      || distanceToTarget < radius)
    {
      requestNewTarget();
    }
  }
}
```

Actual movement of the drone occurs in a separate method (Listing 10) *four* times a second. This is done so that the drones visually appear to move smoother and the separation exists to reduce how often we perform the costly tasks of detecting and avoiding nearby drones.

Adding this separation will also make it easier down the road when we introduce energy consumption to the drones, because we can then simply switch on the drone status to adjust how much energy has been consumed in the previous time frame. For example, a moving drone will consume more energy than a hovering drone.

Listing 10: Drone Movement (four times a second)

```
void MoveDrone()
{
  if (simulate)
  {
    if (DroneDataWriter.Data.droneStatus == DroneStatus.MOVE)
    {
      transform.position += direction * DroneDataWriter.Data.speed
        * SimulationSettings.DroneMoveInterval;
    }
  }
}
```

4.2.1 Drone Detection

In AATC the server knew the location of every drone in the world, which meant that searching for nearby drones would just be a case of iterating through the list of drones and finding the closest one. However, with SpatialOS we delegate this responsiblity to the drones.

The naive method would be for each drone to request all SpatialOS entities within a certain radius, and then filtering to find the closest one. The problem with this is that Spatial find queries are extremely expensive as they have to propagate up to the server instance, be executed, and then returned down to individual entities. In a world with 30 drones where each drone is recalculating its direction every second, that's 30 queries that have to be dealt with in tandem every second.

Due to the fact that each distributed "worker" in SpatialOS is an instance of the Unity game engine, a more practical approach is instead to utilise the engine and perform a simple physics collision check.

Although this is still resource-expensive, it is still quicker than a Spatial search because using native game engine functionality is more performant than manually searching for entities in a radius. By applying a Sphere Collider to each drone, checking for nearby colliders will only ever return instances of drones (Appendix A.2).

4.2.2 Collision Detection

Now that drones are detected easily, a collision is simply the case where a nearby collider is at a distance of less than 1m (2 drone radiuses). In the real world, a controller would detect that two drones have stopped responding and assume a crash occured. For our simulation, we instead add a command for drones to call so the controller knows a collision occurred.

Listing 11: Reporting a Collision

```
type CollisionResponse {}
type CollisionRequest {
  EntityId drone_id = 1;
  EntityId collider_id = 2;
}

command CollisionResponse collision(CollisionRequest);
```

Upon successful receipt of a *CollisionRequest*, a controller may then destroy the drone instances to signal that they are not part of the system anymore. Looking ahead, the controller may decide to use this as a trigger to dispatch drone recovery services or apply penalties for crashing drones.

4.3 Summary

Now that the Drone and Controller entities have been created and relevant components added, we are able to create drones, send them between random locations, and destroy them upon journey completion.

Chapter 5

Delivery Network Architecture

As SpatialOS abstracts away all server-to-server connections and raw networking, we can consider any size world as one singular simulation. As explored thus far, our work only needs to go as far as the definition entities and how their components interact with each other. Since pathfinding and drone-controller communication channels have been set up and tested, our efforts now turn to upgrading this system into a scalable, distributed, delivery network.

5.1 Delivery Destinations

As per the current design, controllers spawn drones, generate a path between two points and then sequentially provide waypoints the drone should follow to get to its destination. To modify this for a delivery network case, the points should correspond to the locations of the controller and the delivery end point. As controllers are both stationary and responsible for pathfinding, making this change is trivial. The question now: how do controllers find out where the delivery end points are?

To resolve this we introduce an Order Generation entity to the simulation, whose sole functionality is to simulate a stream of delivery requests and route these requests to the appropriate controller. The Order Generator may eventually have a load balancing component to it in the future, but for now we simply route each delivery request to the controller closest to the package destination.

Figure 5.1 shows how the Order Generator fits into the existing model. While controllers and drones represent physical things that will exist in the real world, order generation is a virtual entity that only exists for the purposes of the simulation. If such a system were to go live, the order generation entity would be replaced by the actual stream of orders that real customers would be placing.

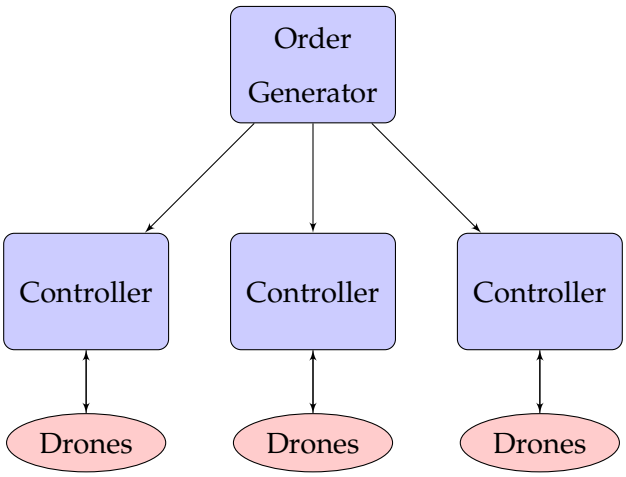


Figure 5.1: Delivery Network Flow.

5.2 Basic Order Generation

Before generating any orders, we must first define the components on the Order Generator and Controller that deal with the sending and receiving of delivery requests.

5.2.1 Delivery Handler / Scheduler

Controllers will need a *DeliveryHandler* component to enqueue incoming delivery requests. It follows that there are a finite amount of drones available to deliver goods at a particular point in time, therefore the Delivery Handler's function is to maintain a queue of requests and only provide the next delivery to serve when the *ControllerBehaviour* asks for it. For the time being, we implement a very basic FIFO queue to serve requests in the order that they were received by the Controller.

The delivery handler effectively acts as a scheduler for the controller, because it decides the order to return requests in a manner completely transparent to core controller operation. The main loop (Appendix A.3) now checks if the controller can deploy a drone, asks for the next request, handles the delivery request if the scheduler returns one back, and sends the updated queue state up to SpatialOS.

To match up with this functionality the schema defines four basic elements:

1. *DeliveryRequest* type
2. *QueueEntry* type
3. list of QueueEntry items
4. *request_delivery* command

Listing 12: Delivery Handler Schema

```
type DeliveryResponse {
  bool success = 1;
}

type DeliveryRequest {
  improbable.Vector3f destination = 1;
}

type QueueEntry {
  float timestamp = 1;
  DeliveryRequest request = 2;
}

component DeliveryHandler {
  id = 1201;
  list<QueueEntry> request_queue = 1;
  command DeliveryResponse request_delivery(DeliveryRequest);
}
```

We designed it this way so that *DeliveryRequest* data could be extended later if we so wish. Important information about a delivery that will extend to support in the future include package weights, package types and even the priority of said delivery. Being able to extend the *QueueEntry* also may open the door for us to play with more complex scheduling algorithms down the line.

5.2.2 Order Generator

Now that we have defined a basic *DeliveryRequest* type, it is the order generator's job to regularly create and send these requests to the appropriate controller. At the moment the request only requires a location vector, so our plan to do this is to generate a random point in the world not in a No Fly Zone, and then find the closest controller to said point.

We would like the orders to be generated at regular time intervals such that each Controller receives an order every 30 seconds on average. By distributing controllers evenly such that each one is responsible for similar sized regions of the world, we can infer that the Order Generator works generates requests every $\frac{30}{\#controllers}$ seconds.

Listing 13: Order Generator Schema

```
type ControllerInfo {
  EntityId controller_id = 1;
  improbable.Vector3f location = 2;
}

component OrderGeneratorComponent {
  id = 1300;
  list<improbable.controller.NoFlyZone> zones = 1;
  list<ControllerInfo> controllers = 2;
}
```

In order to generate valid requests, we need to know about all the No Fly Zones in the world as well as the location and EntityId of each Controller. Since each controller will be placed at pre-determined locations from a starting snapshot, we know both of these details at snapshot-creation and can subsequently use this to set the starting state of the Order Generator.

Part III

Incorporating Reality

Chapter 6

Achieving London-Scale Simulation

A snapshot represents the state of a simulated world at a point in time, storing each entity as well as each component's properties. Every deployment, both local and in the cloud, requires a starting snapshot to populate the simulation. To speed up this process, we create a *scene* for snapshot generation and utilise the power of Unity Editor scripts to programmatically generate and modify snapshots with ease.

Figure 6.1: Our selected region of London. [25]

6.1 Area Selection

London was chosen as the city to simulate partly because of our geographical understanding but also because it could be a prime candidate for rolling out drone delivery services to. Traffic jams, congestion charges and fuel consumption are all ever-present issues for existing delivery networks. As the technology develops, being able to avoid and mitigate these issues by taking to the skies is a leap of faith many operators may be seriously considering.

In the spirit of walking before running, we opt to limit ourselves to a 5km x 4km region of central London, of which 4.8km x 3.8km is actually simulated. We wish to include a 100m wide border to avoid invoking (literal) edge cases in SpatialOS that may arise by simulating entites right at the boundaries of the world.

This particular part of London was chosen due to the variety of institutions and types of land there are available. There are parks, hospitals and skyscrapers that may be potentially protected by No Fly Zones, but there are also educational institutions either side of the river that could serve as ideal locations for Controllers if a pilot was commissioned.

Another major benefit to choosing this area of London is extensibility. Should we decide to, we have the ability to increase the size of the world to test the scalability of the system, adding more of the city and surrounding suburbia to the simulation.

6.2 Snapshot Scene

Figure 6.2 shows how the London scene looks in Unity when visually editing a snapshot. The leftmost column is known as the *Hierarchy*, a list of all the objects currently in the scene. As we are able to nest objects, adding empty game objects allow one to group objects together. For example, No Fly Zones and Controllers could be nested in different groups to more easily distinguish the type of each object.

To the side of the Hierarchy is the *Inspector*, which shows all the Unity components on a selected prefab, in this case a Drone. You may notice an additional component that we have not previously covered residing on this prefab.

Entities that are active and likely to be moving around the world need a special component called a *Transform Receiver*, which simply updates the object's world position whenever its Spatial position is updated.
As each component is authoritative on exclusively the client or server, a completely server-side simulation (such as ours is at present) would require a *Transform Receiver* to ensure that objects are receiving and applying the appropriate Spatial updates if we decide to implement a client-side simulation viewer in the future.

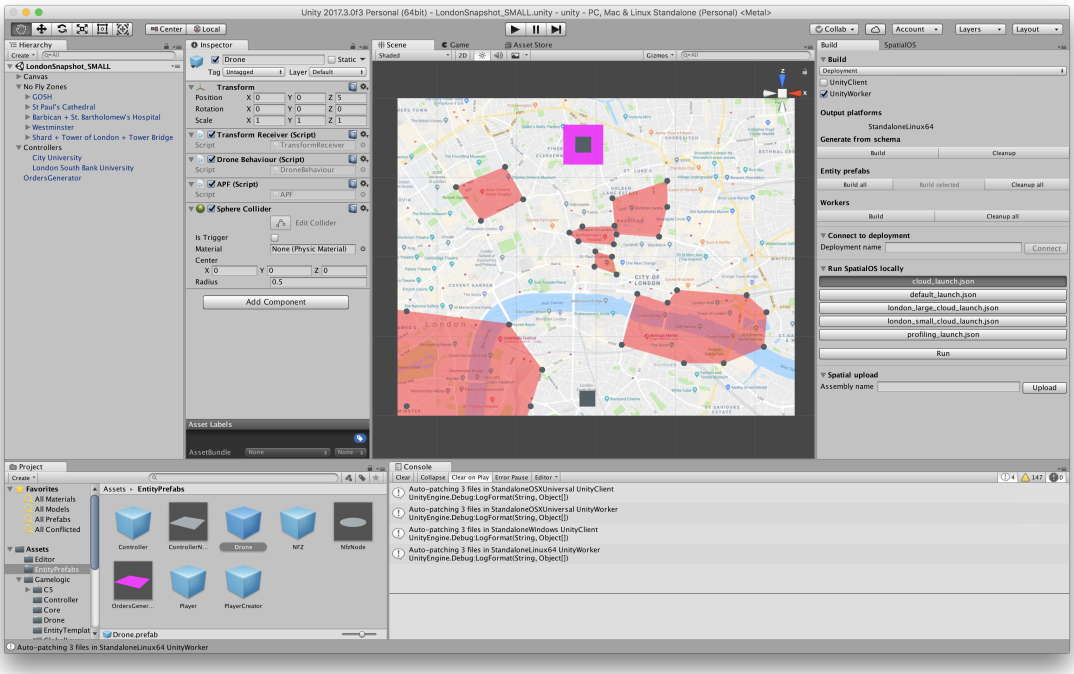


Figure 6.2: Viewing the London scene in the Unity Editor.

The world view in the middle is for visual aid when placing entities in the scene, with red highlights indicating parts of the world that are No Fly Zones. The column on the far right is a menu for SpatialOS build operations. This menu is particularly useful as being able to quickly rebuild the project within the Unity Editor greatly reduces iteration times.

6.2.1 Representing No Fly Zones

Earlier we defined No Fly Zones as a list of vectors to represent its vertices. To accurately place them in a scene, we create the *NFZ* and *NfzNode* prefabs in addition to a complementary *NFZScript* MonoBehaviour.

Listing 14: NFZScript MonoBehaviour

```
using Improbable;
using Improbable.Controller;
using UnityEngine;
using System.Collections;

public class NFZScript : MonoBehaviour
{
  public GameObject[] nfzNodes;
  public Improbable.Controller.NoFlyZone GetNoFlyZone()
  {
    List<Vector3f> positions = new List<Vector3f>();
    for (int i = 0; i < nfzNodes.Length; i++)
    {
      positions.Add(nfzNodes[i].transform.position.ToSpatialVector3f());
    }
    return NFZ_Templates.CreateCustomNoFlyZone(positions);
  }
}
```

The script exposes an array of GameObjects, which can then be populated via the Unity Editor. By placing NfzNode objects in the scene and then populating the script's objects with these, we hold information about each zone's vertices. During snapshot generation we simply iterate through these NfzNodes and extract their positional information to create a NoFlyZone struct.

Figure 6.3 shows an instance of a NoFlyZone prefab being populated with NfzNodes around Great Ormond Street Hospital. Note that renaming an object for our clarity does not change its underlying prefab, making this a really effective workflow for rapidly adding No Fly Zones to a world.

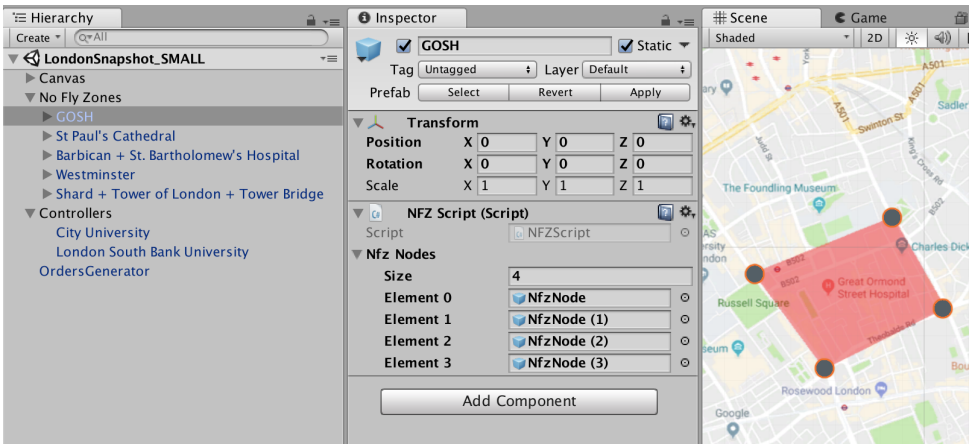


Figure 6.3: Inspecting GOSH's No Fly Zone.

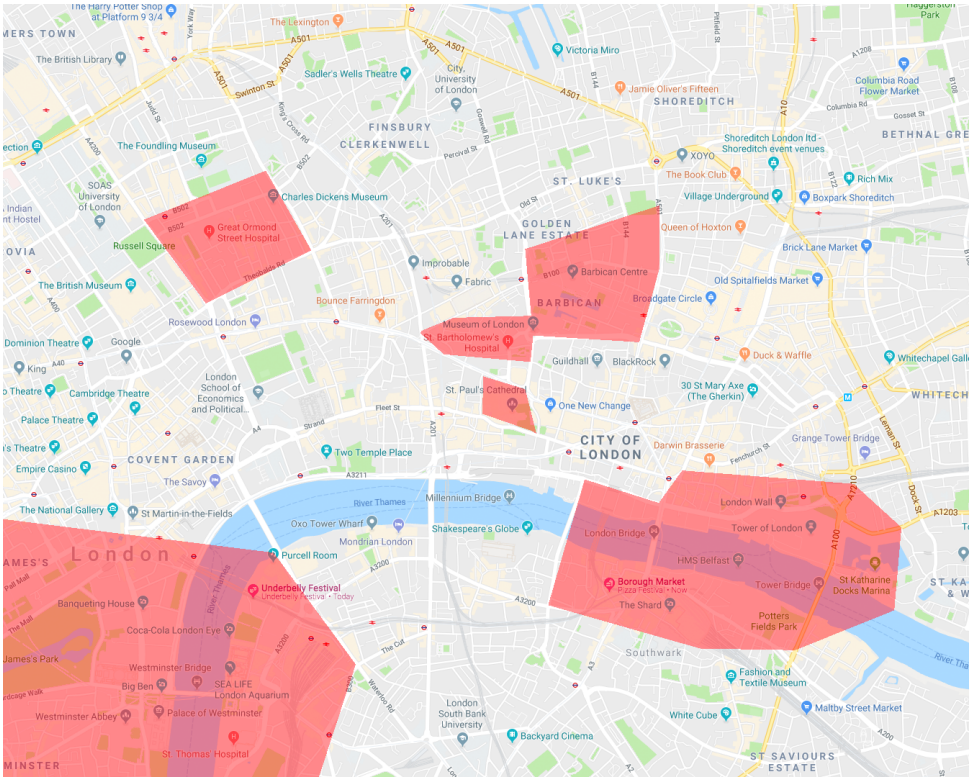


Figure 6.4: Updated version of Figure 6.1 with highlighted No Fly Zones.

6.3 Entity Placement

6.3.1 No Fly Zones

Making use of this new workflow, we added five major No Fly Zones to the world. Our intentions were to protect famous landmarks, public hospitals that may provide Air Ambulance support, the royal parks [15], vertically large constructions and national security hotspots. To this end we placed No Fly Zones over:

1. Famous Landmarks
    - Tower of London
    - Tower Bridge
    - St. Paul's Cathedral
    - Westminster Abbey
    - London Bridge
    - Barbican
2. Major Hospitals
    - St. Bartholomew's Hospital
    - St. Thomas' Hospital
    - Great Ormond Street Hospital
3. Royal Parks
    - St. James' Park
    - Hyde Park
4. Tall Constructions
    - The Shard
    - London Eye
5. National Security Interests
    - Buckingham Palace
    - Houses of Parliament
    - Downing Street

6.3.2 Controller

For a simulation of only 5km x 4km, we felt that there was no need for more than two controllers as we operated under the assumption that the simulation is only of an early-stage pilot delivery network. We scouted for two higher education establishments to act as Controller locations, coming up with a shortlist of University College London (UCL), SOAS University of London (SOAS), King's College London, London School of Economics (LSE), City University, and London South Bank University.

Not only do we have to consider the merits of each location, but how pairs of locations would work. For example, it makes little sense to have a "distributed" delivery network with two controllers in close proximity to each other. For this reason, we felt strongly that there should be one location below the river and one above, meaning that London South Bank University became the site of our first Controller.

After fixing one location, the flaws of others became more apparent. King's was rejected on the grounds that it was placed in an awkward position such that it would end up being responsible for far more of the map than South Bank would. LSE was in a prime position in the centre-left of the map, but with the location came the consideration that nearby hotels and fine dining establishments would not appreciate a major drone operation occuring above their property.

Lastly, UCL and SOAS were rejected for being too close to a major hospital zone. This left City University to become the host of our second Controller, giving us a nice bit of symmetry to the world layout.

6.3.3 Order Generator

Although its location does not have any bearing on the simulation outcome, the SpatialOS load balancer prefers to have this entity in the same place as any random Controller, so we stuck to our above-river bias and planted it under City University.

6.3.4 Drones

No drones exist in the world at startup because it is the job of a Controller to manage the life cycle of a drone. Therefore they do not need to be considered here.

6.4 Snapshot Generator

The actual process of collating all the objects in a scene and creating a snapshot file is triggered by the custom *Drone Sim ¿ Generate London Snapshot* menu option, shown below in Figure 6.5.

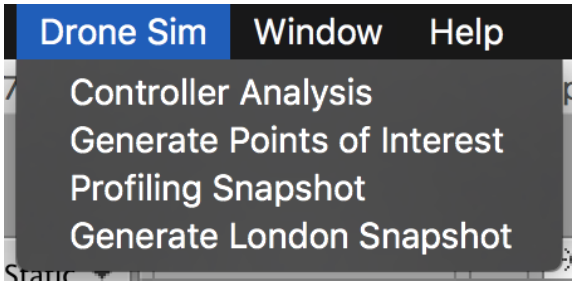


Figure 6.5: Custom drop-down options.

This button then triggers a function to construct a snapshot, entity by entity:

1. Reserve EntityId for Order Generator
2. Initialise a Dictionary mapping EntityId to entity template
3. Find NFZScript objects and extract each NoFlyZone
4. Find Controller objects, pass in NFZ data, create template, store EntityId and position in a list, then add to dictionary
5. Find OrderGenerator object, create template with NFZ data and controller details, then add to dictionary
6. Use a given function to save out the dictionary as a binary snapshot file

On successful creation, a log entry appears in the Unity Editor to say that a snapshot has been created at a particular filepath. This path is then used to point to a snapshot when launching local and cloud deployments.

6.5 Custom Load Balancing

Part of the reason so much thought has been put into Controller placement is the load balancer. Left to its own devices, SpatialOS moves its workers geographically around the simulation to best distribute load across workers.

Although this is brilliant for many use cases, our delivery network is operating under the assumption that each Controller is responsible for its own region with no real inter-region communication at this present point in time. Therefore, it makes sense to put each Controller onto a worker of its own, and minimise as best as possible the overlaps between each region and worker.

To do this, we update the worker launch configuration to make use of *Point of Interest* load balancing [30]. This strategy works by specifying key locations that must not be simulated by more than one worker in the world, trusting the load balancer to divide and distribute regions of the world across a specified number of controllers. Each region consists of an area of the world closest to a given point of interest, like a Voronoi diagram.

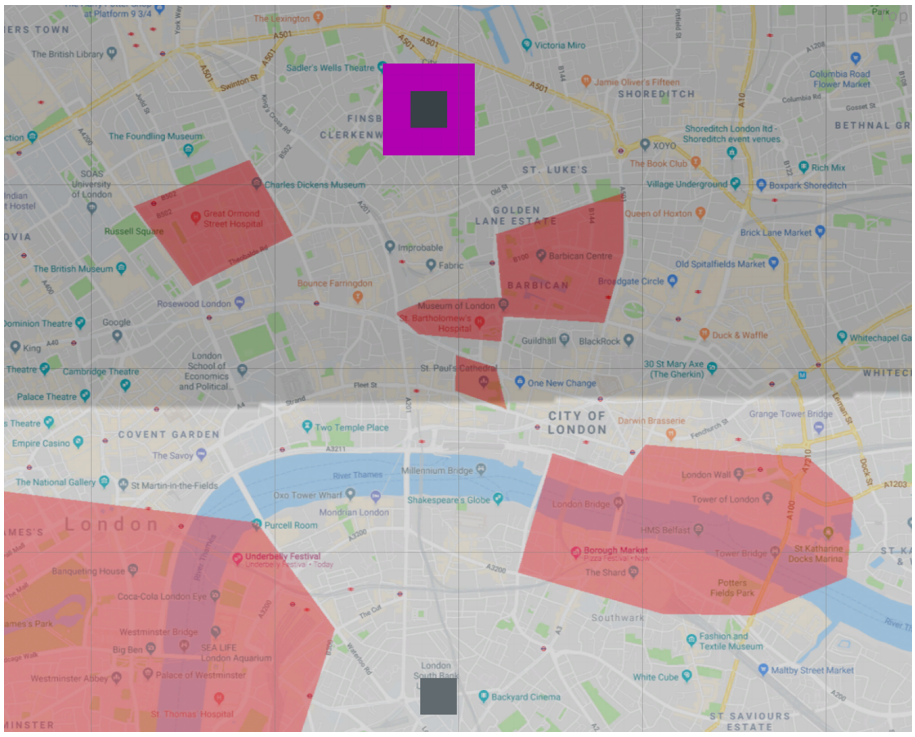


Figure 6.6: Voronoi diagram of the London snapshot.

To generate our own Voronoi diagram of the world, we assign each controller a colour, traverse our world's map and assign a colour to each point based on which controller was closer. After looping through the map, we are given a nice visual representation of the area split. One observes in Figure 6.6 that the two shaded regions of the world are fairly even in size, a positive sign that delivery requests are likely to be balanced well between controllers.

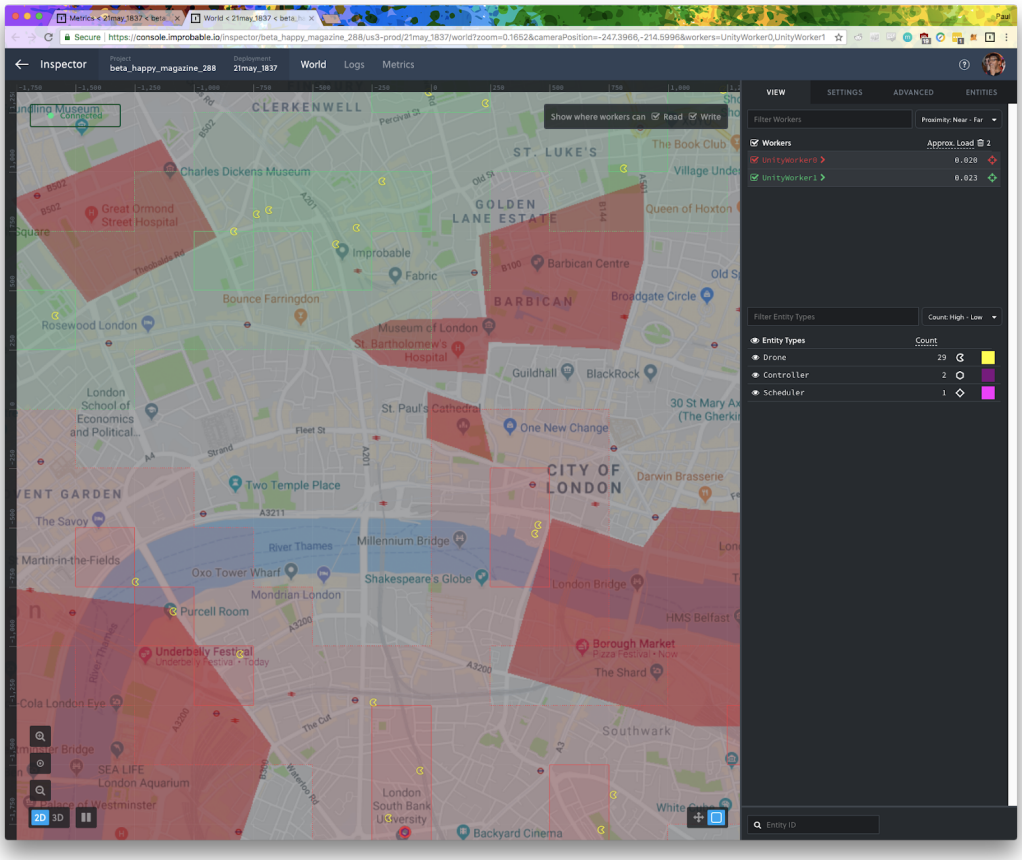


Figure 6.7: SpatialOS view of a running simulation. Note that *OrderGenerator* has been incorrectly labelled as *Scheduler* in the image.

When inspecting a live simulation through the SpatialOS Inspector such as in Figure 6.7, one can clearly see the worker boundary split the world through the middle. Though as mentioned before, SpatialOS constantly tweaks the load balancer so the boundaries may move around slightly as each region's load fluctuates with the ebb and flow of incoming orders and drone availability.

Chapter 7

Integrating Delivery Economics

With a basic London-scale simulation implemented to handle the logistical operations of a drone delivery network, we now look at how to extend this work to factor in, and later optimise for, the economic side.

7.1 Improved Order Generation

Before we can define a revenue model for our delivery network, we must improve the Order Generator to generate a greater variety of delivery requests. Packages being delivered by drones will vary in size, weight and priority, just like current the delivery market, and the DeliveryHandler schema needs to upgraded (Appendix A.4) to support these new properties of a delivery request.

7.1.1 Package Type & Weight

We take our definitions of package types from Amazon's own *Fulfilment by Amazon* service, as described by Figure 2.3 in Section 2.2. Subsequently we update the *DeliveryRequest* schema type to include a new *PackageInfo* property, to capture both the package type and weight. Each of the 6 package types accepts a different range of weights, hence generating a PackageInfo is (at the moment) a case of randomly picking a type, and then generating a random number within each type's accepted weight range.

7.1.2 Delivery Priority

In addition to the actual package information, each delivery request may have a priority corresponding to how urgently a customer wants their order to be delivered. Existing delivery companies each have their own range of services to customers, so to simplify this parameter we define four levels of priority: STANDARD, PRIORITY, URGENT and SUPER_PRIORITY. We randomly select a priority for now, with the option to adjust bias further down the line just like with package types.

7.2 Revenue Model

As we are adopting the package types described by the *Fulfilment by Amazon* programme, it makes sense to adopt their pricing model too as a starting point for standard priority deliveries.

Since we aim to provide a better service to orders with higher priorities, the price for each priority tier is a 50% mark up from the previous tier. For example, standard delivery of a large letter would generate 80p, but priority delivery would generate £1.20 (80p x 1.5).

In addition to the priority system, we introduce a mechanism whereby the cost to a consumer per delivery, ergo ther revenue generated, decreases with the time taken to perform said delivery. The cost is halved if it takes more than 30 minutes to deliver the package after order generation, and reduced to 0 if the delivery takes more than 60 minutes to complete.

By charging both bigger fees for higher priority requests we provide incentives for controllers to prioritise more important requests to generate more revenue. However by offering reductions for slower delivery of goods, we also ensure that controllers attempt to serve all delivery requests as soon as possible.

Table 7.1: Package Revenue Distribution.

Package Type	Weight (g)	Delivery Fee (£)			
		Standard	Priority	Urgent	Super Priority
Small Letter	0 - 100	0.60	0.90	1.35	2.03
Large Letter	0 - 250	0.80	1.20	1.80	2.70
Small Envelope	0 - 100	1.09	1.64	2.45	3.68
Standard Envelope	0 - 100	1.21	1.82	2.72	4.08
	101 - 250	1.34	2.01	3.02	4.52
	251 - 500	1.54	2.31	3.47	5.20
Large Envelope	0 - 1000	1.77	2.66	3.98	5.97
Parcel	0 - 250	1.73	2.60	3.89	5.84
	251 - 500	1.79	2.69	4.03	6.04
	501 - 1000	1.84	2.76	4.14	6.21
	1001 - 1500	2.26	3.39	5.09	7.63
	1501 - 2000	2.48	3.72	5.58	8.37
	2001 - 3000	3.32	4.98	7.47	11.21

7.3 Operating Costs

To calculate the profitability of the drone delivery network, we need to consider the costs of operating such a system. For our simulation we have identified the main sources of expenditure to be the energy consumption overhead of running the drones, along with any penalties that may be charged for failed deliveries and mid-flight collisions.

Although we have designed each Controller to act as a delivery dispatch point, like an Amazon Fulfilment Centre, we do not consider the costs of running such a warehouse. Instead we only focus on economic matters that are influenced by our newly introduced drone delivery dynamics.

7.3.1 Energy Consumption

As a baseline to model our simulated drones on, we look at information available about Amazon Prime Air prototypes. We can compute the expense of running the drones by first keeping track of a drone's energy usage per delivery, then multiplying it by the cost per unit of energy.

We know that the power rating of a moving drone is 8355.1W, but we must compute the power of a drone hovering ourselves using the current and voltage readings. Using a basic understanding of Physics, we define energy and power below.

$$energy = power * time$$

$$power = current * voltage$$

The prototype in question [38] has 8 motors, each rated at 36.59V and 2.29A. Giving us a power rating of 670.3W when the drone is hovering.

$$8\ motors * 2.29A * 36.59V = 670.3W$$

Because we move drones at fixed time intervals, we can pre-calculate the energy increments depending per drone state. When updating the position of the drone every 0.25 seconds, we can also increment the energy usage by these fixed amounts. Being able to calculate these in advance reduces the work we have to do on every call to *MoveDrone()*.

$$energy\ (motion) = 8355.1 * DroneMoveInterval\ (seconds) = 2088.775\ Ws$$

$$energy\ (hover) = 670.3 * DroneMoveInterval\ (seconds) = 167.575\ Ws$$

Once a drone completes its delivery and has returned to the Controller, we can compute the total cost of performing said delivery. Taking into account the different units of time used so far:

$$cost\ of\ delivery\ (pence) = energy\ usage * 0.0055/3600$$

We believe delivery network operators would be able to source electricity at wholesale rates, which we found to be around £55 per MWh [8], or 0.0055p per Wh.

7.3.2 Penalties

Understandably, nothing will ever work perfectly forever. To mitigate bugs in the simulation, we periodically "prune" drones that have been unresponsive for too long. We therefore consider the penalties applied for actions the drone has failed to complete at the time of pruning. We therefore define four main infractions:

1. Failed Launches
2. Failed Deliveries
3. Failed Returns
4. Collisions

When a drone is unable to be launched, we consider that a delivery request was accepted but then unable to be completed. From a consumer point of view, they may have received a notification indicating that their delivery is on their way, only to be quickly told that their order has been cancelled because of some error in launching a drone. In this instance we do not charge the customer any delivery fees, instead giving them a fixed £5 compensation.

If a drone is launched but still fails to deliver a package for whatever reason, we compensate the consumer as before and also calculate the cost of sending a truck to manually retrieve the drone that was sent out. When a drone fails to return for some reason *after* making a delivery there is no reason to compensate the customer, but we still have to calculate the cost of retrieving the drone.

When calculating the cost of retrieving a drone, we model our lorry on a standard UPS delivery truck with a mileage of 13.1 miles per gallon [41]. We consider fuel prices to be 125.1p per litre [54], which translates to 568.7p per gallon.

$$retrieval\ cost = 2 * (distance * 0.621371) * mileage * fuel\ cost$$

In the event of a collision, we add a £400 ($500 [38]) penalty for every drone involved in the collision, assuming the worst case where each drone would need to be completely replaced. In addition to this, we add appropriate penalties for failed deliveries and add a unified cost of retrieving all the drones.

7.4 Case Study: Delivering a Milkshake

To demonstrate how cost effective drone delivery is, we create a hypothetical case where a customer would like to order a milkshake at late night from the comfort of her room. The round trip of picking up, delivering the milkshake and then returning back to the Controller is approximately 4.33km.

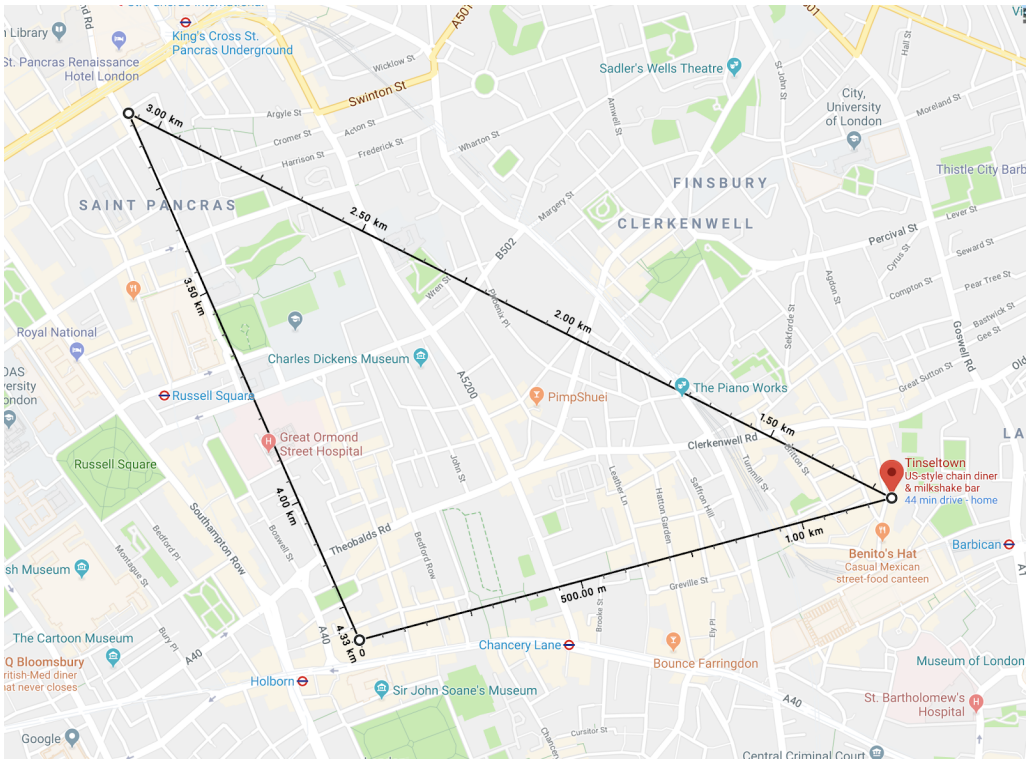


Figure 7.1: Measuring the total distance of the delivery round trip.

For estimation sake, we assume that the drone is always moving. Therefore at a maximum speed of 10 metres per second, the drone would be in flight for 433 seconds. This corresponds to 3617758.3 Ws of energy use. Substituing this into our equation gives us:

$$cost = 3617758.3 * 0.0055/3600 = 5.527130736\ pence$$

As per Table 7.1, the revenue generated for standard delivery of an 800g milkshake is £1.84 for a operating costs of just 5.53p to the delivery network operator. As this is a mostly-autonomous system, the lion's share of generated revenue is pure profit.

For comparison, a large portion of Deliveroo's standard delivery fee £2.50 may be used to pay the delivery person. Perhaps the trickling down of these monetary, time and efficiency savings to the customer becomes the catalyst for increased drone adoption across delivery networks.

Chapter 8

Time-Value of Delivery

Before considering how to maximise profitability, we introduce the concept of the *Time-Value of Deliveries* (TVD). This is based on the *Time-Value of Data* idea we described earlier in Section 2.8. Rather than deal with the fixed pricing model of delivery fees as we see in the current day, we propose a new mechanism where the cost of the delivery is a function of the time taken to do said delivery. This is known as a Time-Value Function (TVF).

For example, a user may pay £5 for a delivery that takes less than 30 minutes but be charged nothing if it takes longer than one hour. One may recognise that we are already adopting an informal TVF that halves the delivery fee after 30 minutes, making it 0 if delivery takes over 1 hour. Note that "delivery time" is the time taken from a controller receiving an order to a drone reaching the requested destination.

8.1 Function Representation

To reduce the computational effort required to calculate the cost of a delivery, we simplify each TVF to have 10 steps at equal time intervals. By keeping an array of booleans, we simply toggle each step true or false to indicate whether there is a price drop at that step.

Figure 8.1 shows early concept art of a TVF with price drops at 18, 42 and 60 minutes. In this example, our array would have the the 3rd, 7th and 10th elements of the TVF array set to true.

$$[0, 0, 1, 0, 0, 0, 1, 0, 0, 1]$$

For our model, we also state that the price decrease per step is constant per TVF. The maximum value of a delivery is therefore extracted from Table 7.1. The value of a step decrease is the maximum value divided by the number of steps in the TVF. This is to reduce the transmission size of a TVF, as each delivery request may provide its own function.

An array of 10 booleans takes 10 bits, but an array of 10 integers for the value at each step would be 40 bytes, 32 times the number of bits. Considering a TVF will be sent with every delivery request and each controller maintains a queue of requests, reducing the memory footprint of a TVF may allow the simulation to scale better down the line.

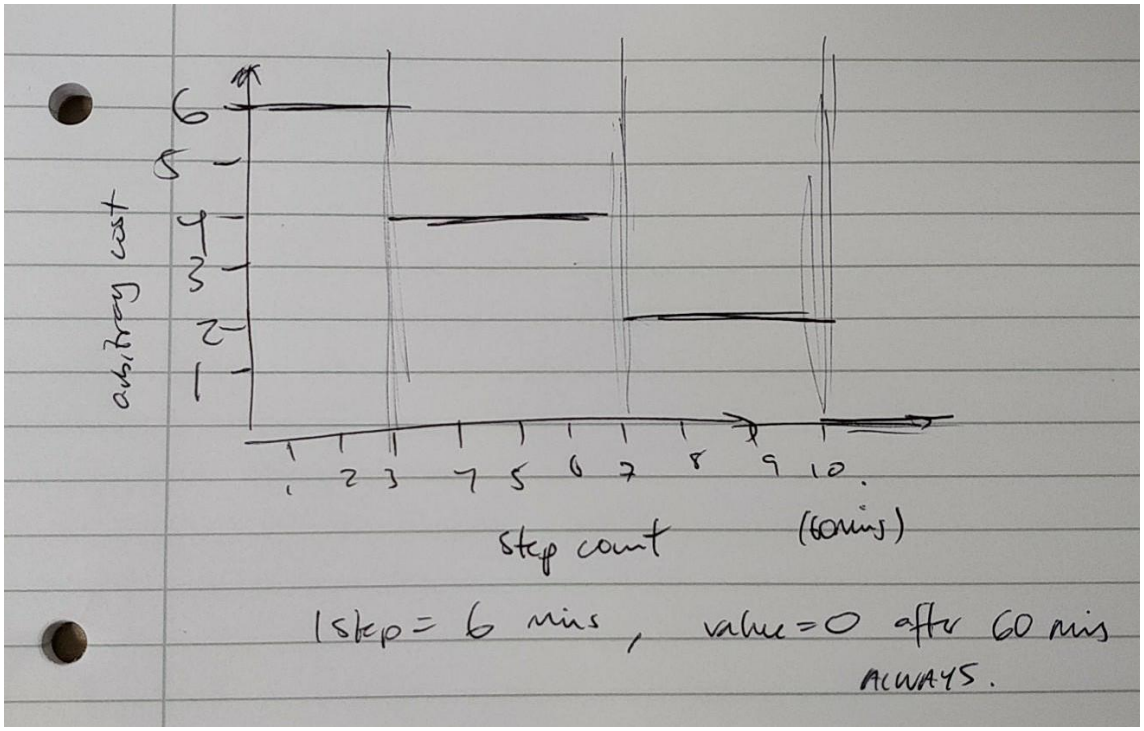


Figure 8.1: Conceptualising a Time-Value Function.

8.2 Utilised Functions

To integrate TVFs with our simulation, we update the Order Generator to support two models. For every delivery request generated, the Order Generator randomly chooses between *The Halvening* or *Stepwise Decrease* functions.

Although we could simply enumerate these two models and calculate the delivery fee using pre-defined value functions, our chosen implementation opens up the option to experiment with more complex TVFs in the future.

8.2.1 The Halvening

$$[0, 0, 0, 0, 1, 0, 0, 0, 0, 1]$$

The Halvening is a formalisation of the price-halving model we had already implemented before we introduced Time-Value Functions. By allowing longer periods of time before fee decreases, controllers are given more flexibility to, perhaps, schedule a higher priority request and still generate the same revenue on the other request. In general, this model might be best suited for a customer who is not *too* strict on the swiftness of a delivery.

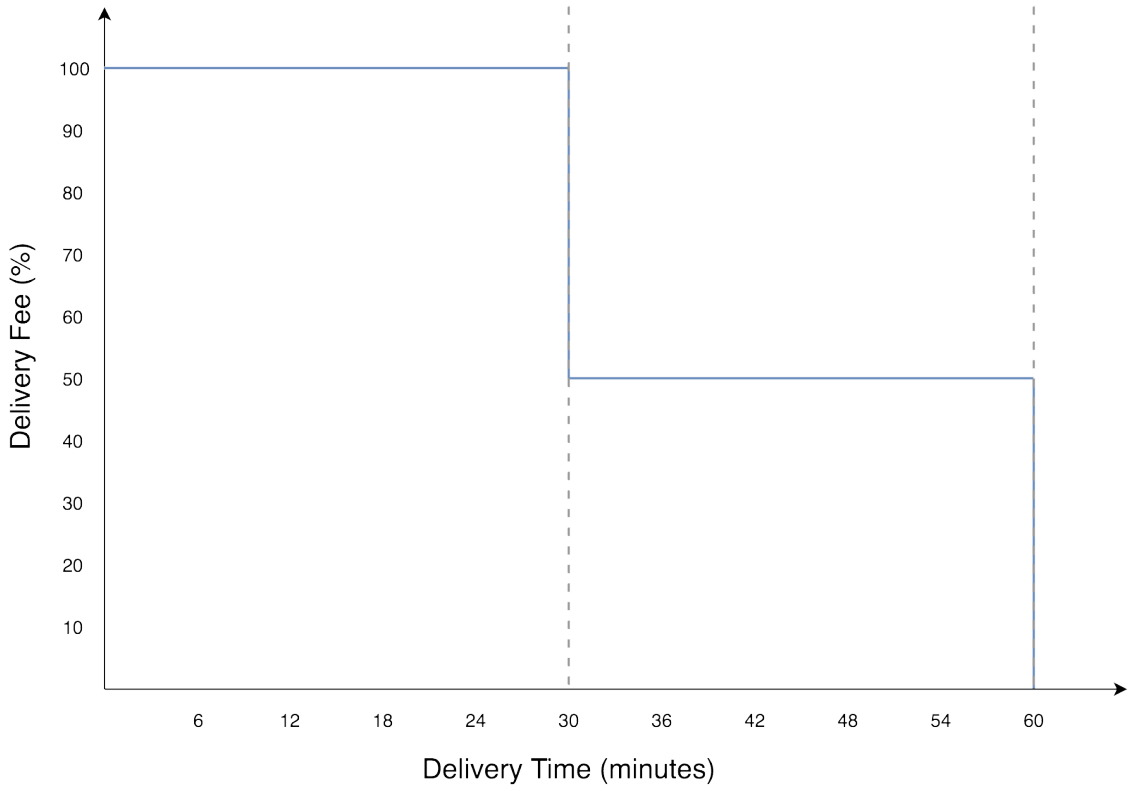


Figure 8.2: Plotting *The Halvening*.

8.2.2 Stepwise Decrease

$$[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]$$

More customers may consider opting for this model as it enforces a fee decrease every 6 minutes, which means Controllers are given greater incentive to deliver orders with this TVF as quick as possible.

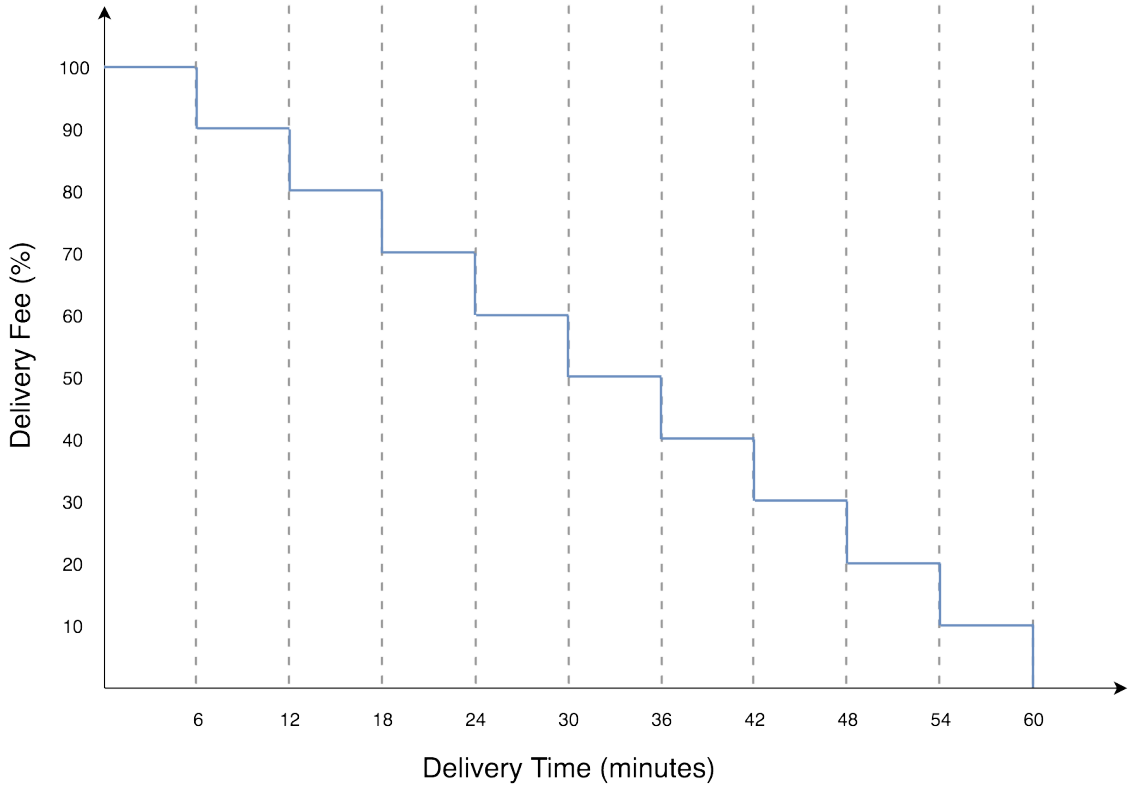


Figure 8.3: Plotting *Stepwise Decrease*.

Part IV

Optimisation & Scale

Chapter 9

Profit-Optimised Scheduling

Our simulation now consists of a distributed delivery network, variation in order generation and a novel revenue model, but each Controller is still using a very primitive form of scheduling - First Come First Serve (FCFS). As we have demonstrated in Section 7.4, drone delivery networks are already extremely profitable, therefore in our simulation controllers do not stand to make any sort of loss. Nonetheless, there is no shadow of doubt that there are modifications that can be made to the scheduler to prioritise higher value orders and generate a greater profit.

In our simulation, each controller only operates 15 drones at a time with a maximum request queue size of 40. After this point orders will still be coming in, but a FCFS scheduler would simply discard any new requests to serve the existing queue. One would then imagine that by re-evaluating the request queue on each new order received to weed out lower value orders, we would be able to maintain requests with higher potential values and thus increase our profitability.

In order to make use of such a priority queue to filter out the lowest value orders, we must first define a way of assigning a priority to each request. For our simulations we have chosen to implement the novel *Least Lost Value (LLV)* algorithm as our primary profit enhancing technique, as this approach takes advantage of the TVF-based revenue model.

In the interests of comparing LLV with a more popular but well-performing algorithm, we also take a look at and implement *Shortest Job First (SJF)* scheduling.

9.1 Least Lost Value

As described earlier, the concept of deliveries having a value that decreases with the time taken to complete it is inspired by the *Time-Value of Data (TVD)* model used in big data and business intelligence. Therefore when investigating methods of optimising our scheduler for profit, we looked at ways that big data jobs would be scheduled for optimal time-value of data.

Our quest for optimisations led to a meeting with a PhD student of our supervisor, Prof. Knottenbelt. The PhD student, Shireen Seakhoa-King, was working on a scheduling optimising tool for the Big Data use-case and was informed by our supervisor about the project described in this report. This opened the communication channels to discuss her tool and its potential applications further.

After multiple meetings it became clear that Shireen's tool was a TVD-based scheduling algorithm that claimed to process data with higher values compared to other TVD-based methods. Intrigued by this pitch, we agreed to implement her algorithm as our primary profit-optimising scheduling policy.

9.1.1 Algorithm Overview

As the name suggests, this algorithm aims to schedule items according to the *Net Lost Value* metric. The next request to be taken from the queue is always the element with the lowest net lost value. To compute this value, LLV considers both the *Potential Gain Value (PGV)* for performing a request as well the *Potential Loss Value (PLV)* by doing other jobs later.

To calculate the *Expected Value (EV)* of a job, its duration is modelled as a task duration distrubution in order to account for variance in execution times. This duration is then used to compute EV in a similar manner to what we have previously described with Time-Value Functions.

$$EV(j_i, t_c) = \int_{t_0}^{\infty} V_i(t).f_i(t - t_c)\, dt$$

The Potential Gain Value is therefore defined as the difference of value between processing a job now versus at a later time.

$$PLV(j_i) = \sum_{j_k \in J,\ k \neq i,\ k-1}^{n} (EV(j_k, t_c) - EV(j_k, t_c + p))$$

The Potential Loss Value is defined as the sum of the lost value of all the other jobs that have not been selected.

$$PGV(j_i) = EV(j_k, t_c) - EV(j_k, t_c + \bar{p}_i)$$

Once these have been computed, the Net Lost Value per delivery is defined as Potential Gain subtracted from the Potential Loss.

$$NLV(j_i) = PLV(j_i) - PGV(j_i)$$

The NLV is then computed as above for all jobs wanting to be scheduled, and the next job to be scheduled is the one with the lowest NLV.

9.2 Adapting LLV for Delivery Scheduling

To adapt the LLV algorithm for our specific drone delivery use case, we must consider how to model the duration distribution and also how to evict elements from the request queue when it becomes full.

9.2.1 Duration Distribution

To simplify the problem and reduce the computation complexity of the scheduler, we opt out of modelling the expected duration as a distribution. Instead, we assume the drone maintains a constant speed from start to end, and simply treat the duration as a fixed value derived from the distance between the controller and the delivery destination.

$$estimated\ duration = \frac{distance\ to\ destination}{maximum\ drone\ speed}$$

Although this means that we do not consider the variance of delivery times due to factors such as dynamic collision avoidance, deviations in start and return positions, and the time taken for the drone to reach cruising altitude, this method allows us to obtain a quick and dirty estimation that can be re-calculated with ease.

Being able to re-calculate this value with ease is of particular interest because the expected duration is something that the LLV algorithm attempts to compute several times during the course of its execution. However, since our way of calculating this variable generates the same result every time for a particular order, due to delivery destinations being fixed, we can add this field as a property to the *QueueEntry* schema type.

Calculating the value once, storing it and then re-using it further reduces the complexity of the LLV algorithm in our particular use case.

9.2.2 Queue Implementation

If the priority of a queue under LLV policy is the NLV, then for every item in the queue we would have to traverse all other elements in the queue to calculate that item's priority. Assuming queue size n, we find that the time complexity of sorting a queue under this mode of scheduling is $O(n^2)$.

As each element's priority depends on the values of all other elements, we have to re-sort the queue every time an element is added or removed. Since we are expecting a new order to arrive approximately every 30 seconds, we get a $O(n^2)$ operation running at least twice a minute for the entire uptime of a Controller!

Although a Controller makes an attempt to dispatch a drone every 10 seconds, we only remove elements from the queue at 10 second intervals when there is no contention among the requests for access to drones. By that, we mean that the priority of requests only really comes into play once we have more pending requests than available drones.

We attempt to keep our Controllers busy to give our schedulers (quite literally) a run for their money, but even in this case the time between dequeueing of requests may well be in the minutes range as we wait for drones to return from the deliveries they make. Therefore, it makes sense to only re-sort the request queue when *DeliveryHandler* is polled by *ControllerBehaviour* for the next request to serve.

Even if the number of elements in the queue grows beyond the maximum limit we set, we prune the queue after re-sorting to make sure that only the top 40 highest priority elements remain.

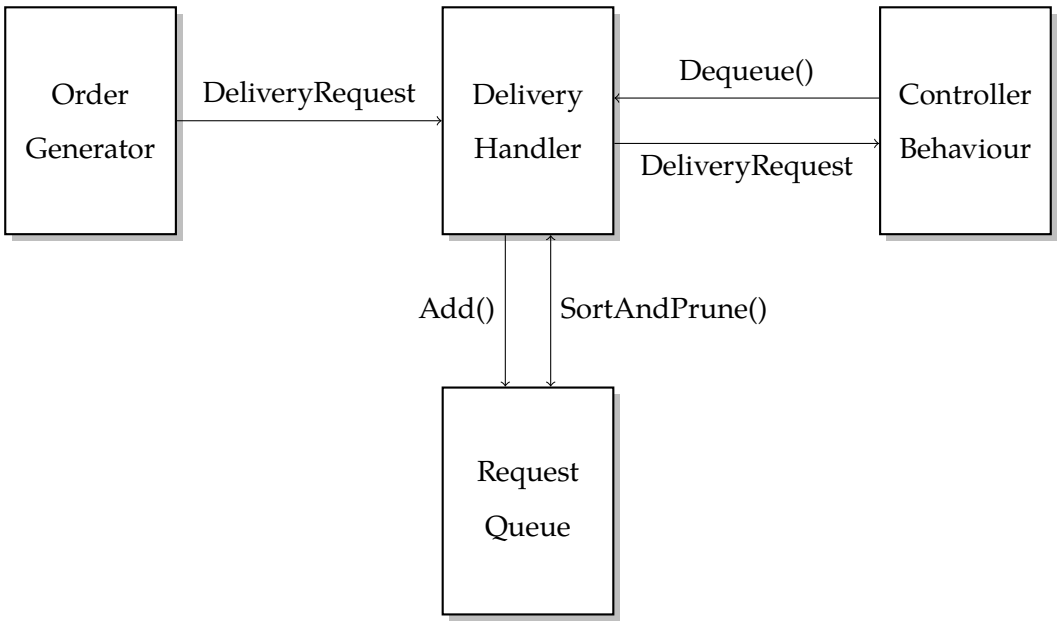


Figure 9.1: Request Queue Interaction.

9.2.3 Preliminary Results

Preliminary runs using the LLV scheduler appeared to demonstrate far better performance than our existing First Come First Serve (FCFS) scheduler, in terms of both average profit per delivery and the overall profit generated over the course of the simulation.

This proved to be an interesting development that piqued our curiosity into investigating how our Least Lost Value policy stacks up against a Shortest Job First (SJF) approach, which Shireen expressed was the only other scheduler that would beat LLV under some conditions in her own testing. There was especially great interest to see how the two schedulers would fare in our specific use case of a drone delivery network.

9.3 Shortest Job First

In this algorithm, the request queue would be ordered by the estimated time to complete each delivery. For the sake of estimation and to be consistent with our assumptions in LLV scheduling, we assume a constant drone speed. Hence, we can efficiently order the queue by the total distance (as the crow flies) from the controller to the request destination.

$$time = distance/speed$$

$$job\,length \propto distance$$

9.3.1 Pruning

For the LLV scheduler, we re-sort the queue on every poll for a new *DeliveryRequest*. On the other hand the priority of a *QueueEntry* in SJF scheduling does not change once it has been calculated, therefore we can add and remove requests to the queue using built in priority queue functions with ease.

When a new delivery request comes in and the queue is smaller than the maximum allowed size, we calculate its priority and add it to the queue. If the queue is already full we compare the new request with the lowest priority item in the queue, and remove which of these two requests has the lower priority.

9.3.2 Comparison to LLV

We hypothesise that the SJF scheduler will perform exceedingly well when it comes to the sheer volume of deliveries it completes because it prioritises the shortest request all the time. Naturally, over the same duration of time, shorter jobs on average will mean more completed deliveries.

However if the average profit generated by these jobs is too low for the higher volume to compensate, the LLV scheduler will perform better. At this point, it is hard to predict a winner without conducting a through evaluation of simulation data.

9.4 Advanced Scheduler Implementation

We implement our advanced schedulers as priority queues with custom comparators.

9.4.1 Priority Queue

To allow us to prune low priority items in addition to dequeueing those with the highest priority, we need to make use of a data structure that efficiently keeps track of both maximum and minimum entries. To this end, we make use of the C#-native `SortedSet<T>` class [45] due to its capability to easily return the maximum and minimum entries, and in doing so making use of a comparator that we can define ourselves.

9.4.2 Custom Comparator

Our priority queue will be of type `SortedSet<QueueEntry>`. As QueueEntry is a struct generated from the type we define in the Schema, we have to define our own method of comparing two different `QueueEntry` items and determining which is greater or if they are the same. The method that forms this comparison is known as a comparator. To aid the comparator, we introduce a `priority` component of type float to `QueueEntry`. We use this to determine if an element is greater or lesser than another one.

To check equality, we initially decided to check if two timestamps were equal. Despite it being a grave computer science sin to compare floating points, our rationale was that it would be almost impossible for two orders to arrive at a Controller at the same time. In fact, it would also be extremely unlikely that two orders arrived with a time delta smaller than floating point precision. However, we noticed on test simulations that the queue size would continually increase every minute until the worker finally crashed.

To remedy this we added an `id` field of type `int64`(long) to the `DeliveryRequest` schema. By ensuring that the Order Generator keeps incrementing the id of each outgoing request, we would never have two requests with the same id. Being an integer type, we are able to check equality of the request ids without having to worry about floating point precision errors as before.

Our custom comparator now checks for equality of `DeliveryRequest.id` and if inequal orders elements based on their `QueueEntry.priority`.

Listing 15: Custom Comparator Implementation

```
class CustomComparer : IComparer<QueueEntry>
{
  public int Compare(QueueEntry x, QueueEntry y)
  {
    if (x.request.id == y.request.id)
    {
      return 0;
    }
    if (x.priority > y.priority)
    {
      return 1;
    }
    return -1;
  }
}
```

Chapter 10

Snapshot Enlargement

In order to test the scalability of our current delivery network architecture, we need to try scaling up our simulation. This involves generating a larger snapshot with more Controllers, allocating bigger regions of the world to each Controller, as well as tweaking both the Simulation and SpatialOS configuration settings to support a simulation at the scale we desire.

10.1 Expanding London

Using our experience from building the smaller snapshot of London as a starting point, we define our key considerations for drone delivery network snapshot design to be:

- Simulation Settings
  - World Size & Location
  - Max Drone Count
  - Max Queue Size
  - Order Generator Interval
- No Fly Zones
- Controllers
  - Controller Count
  - Distribution

10.2 Simulation Settings

10.2.1 World Size & Location

As our first snapshot was very focused on central London, we want our expanded snapshot to extend outwards in all directions and encompass more of suburban London. Simulating more of the city provides us a larger canvas and greater ability to tinker about with Controller placement. Perhaps Controllers in suburban locations are able to deal with larger areas as less orders may be requested in the outer areas?

With these thoughts in mind we define the four corners of our world to be Richmond, Wembley, Goodmayes and Bexleyheath, as shown by the area in Figure 10.1. These bounds were selected as it encompasses a large variation in London land mass. Not only would it allow us to have a good amount of realistic No Fly Zones, we could also get decent diversity in the size and shape of zones.

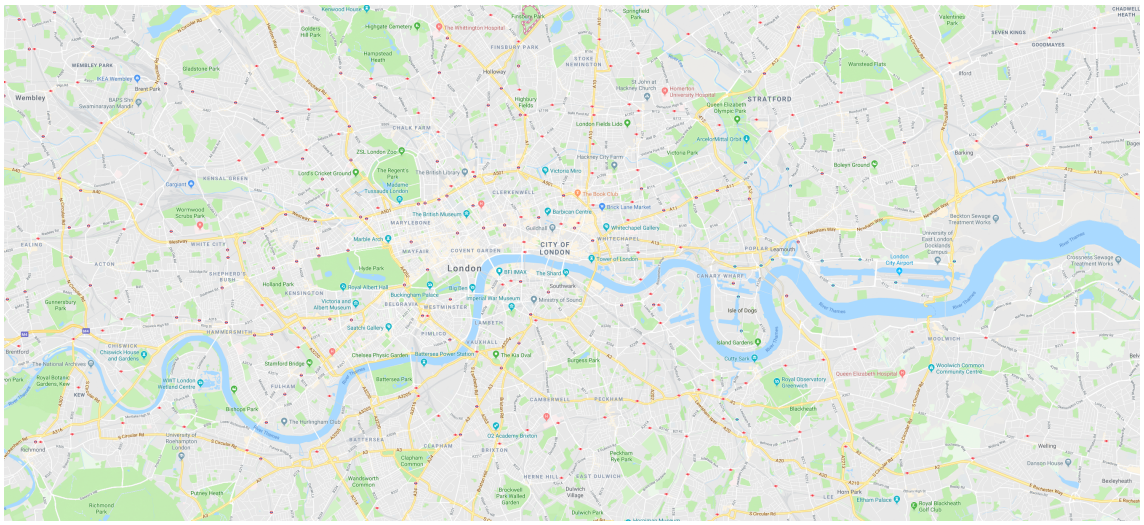


Figure 10.1: Our large London snapshot.

The new enlarged area we have chosen to simulate would generate a snapshot of London with dimensions of 32km x 14.5km, of which 31.5km x 14km is routable. For comparison our existing London snapshot has dimensions of 5km x 4km and a routable bounds of 4.8km x 3.8km, which means that the new snapshot represents over a 24x increase in routable area.

10.2.2 Order Generator Interval

Though the raw value of this variable is directly dependant on the number of Controllers we have to make use of in this world, we decide to double the expected incoming request rate per Controller so that we are able to observe how they deal with higher request volumes.

Suppose we deploy 20 Controllers, we would have to generate a new order every 1.5 seconds in order to average a new incoming `DeliveryRequest` every 30 seconds per Controller. In the new model doubling the rate means halving the interval time, meaning the OrderGenerator would now send out a new `DeliveryRequest` every 0.75 seconds.

10.2.3 Other Settings

As each Controller is expected to handle larger regions of land than before, we increase the number of drones per Controller from 15 to 20, and boost the maximum queue size from 40 to 50. If we doubled the request rate but also doubled the availability of all other resources, there is no net change in simulation behaviour except an increase in volume.

By scaling the various parameters differently, we aim to make life more difficult for the simulated delivery network. In this way, our scale test also can act as a stress test to pinpoint weakpoints in the system that need to be amended or mitigated before such a delivery network "takes off" in real life.

10.3 No Fly Zones

Though we lift the restrictions on adding No Fly Zones above major hospitals, we still maintain the rule regarding Royal Parks. Subsequently, we add 7 NFZs associated with Royal Parks to the snapshot. Until there is regulatory clarification over whether flying over parks could be a violation of several privacy rights, we also add Victoria Park, Olympic Park, Kew Gardens and Battersea Park to our list of parks protected by a No Fly Zone.

Note that the Regents Park NFZ is also the protector of both London Zoo and Lord's Cricket Ground.

As it is the financial and entrepreneurial hub of the capital, we protect the City of London under a No Fly Zone. London Bridge, Tower Bridge and the Tower of London can all be found under this NFZ. To avoid creating a massive blockade in the center of the city, we place the Barbican under a separate NFZ.

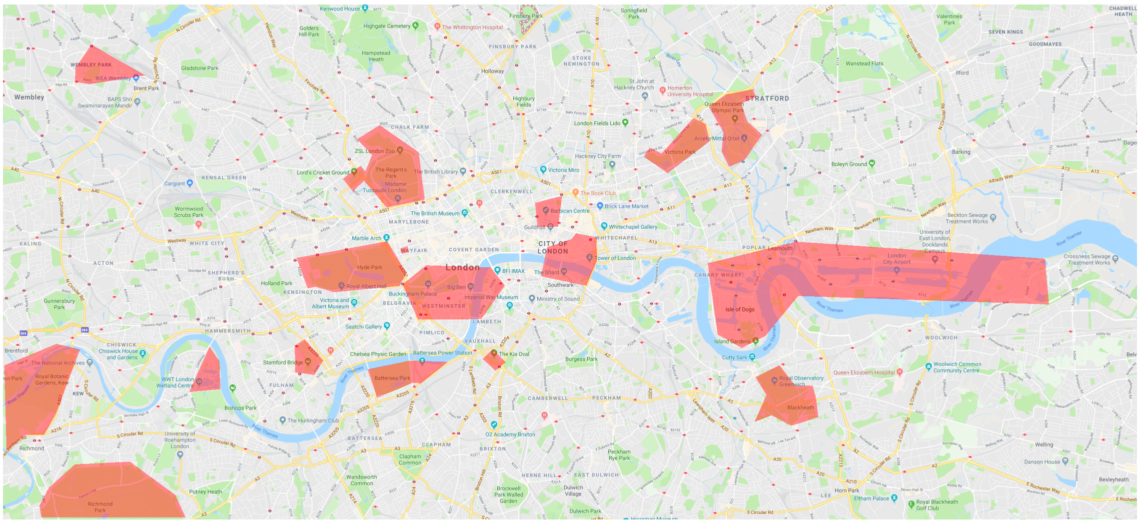


Figure 10.2: Our large London snapshot with NFZs highlighted.

We also apply No Fly Zones to cultural and sporting hotspots such as the Oval, Wembley Stadium and the O2. However, we also extend the O2 No Fly Zone across a large stretch of East London in order to cover both the skyscapes of Canary Wharf, and the flight paths of planes coming in and out of London City Airport.

The full list of No Fly Zones can be seen in Appendix B.2.

10.4 Controllers

Now that we have our world populated with No Fly Zones, we are able to consider how we might choose to configure our Controllers. Placing Controllers near a NFZ reduces their effectiveness because one of the directions around them is not really a feasible option to route drones via.

By placing plenty of distance between each Controller and its closest No Fly Zone, we are able to maximise the number of directions that we are able to send drones out to.

10.4.1 Initial Placement

Initially 18 Controllers were placed across the snapshot, with the intention of having a greater density of drones in the middle of London. However, it can be seen immediately that some Controllers are responsible for huge swathes of suburban London.

When compounded with the fact that some of these Controllers would cover regions were there were a lot of No Fly Zones, some could be incredibly underutilised and others could be pushed to their absolute limits by trying to serve all the requests in a large region. This erratic behaviour could be mitigated by analysing their geological distribution properly.

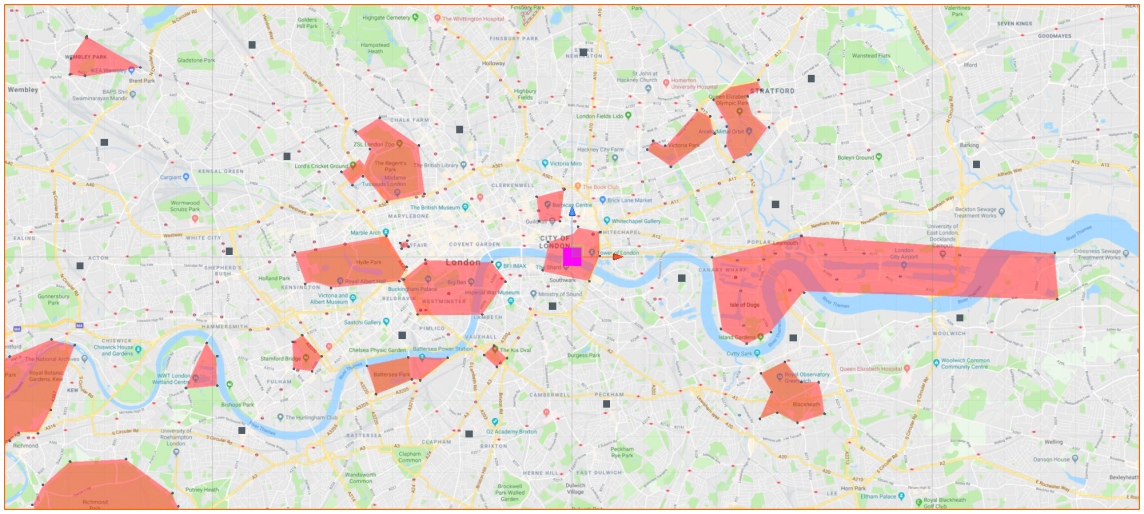


Figure 10.3: Intial Controller Placement.

10.4.2 Voronoi Analysis

As may be familiar from Section 6.5, we produce a Voronoi diagram for the initial placement of Controllers shown in Figure 10.3. We then notice that some cells are far larger than others and iteratively make the cell sizes more even by adding Controllers in between large cells.

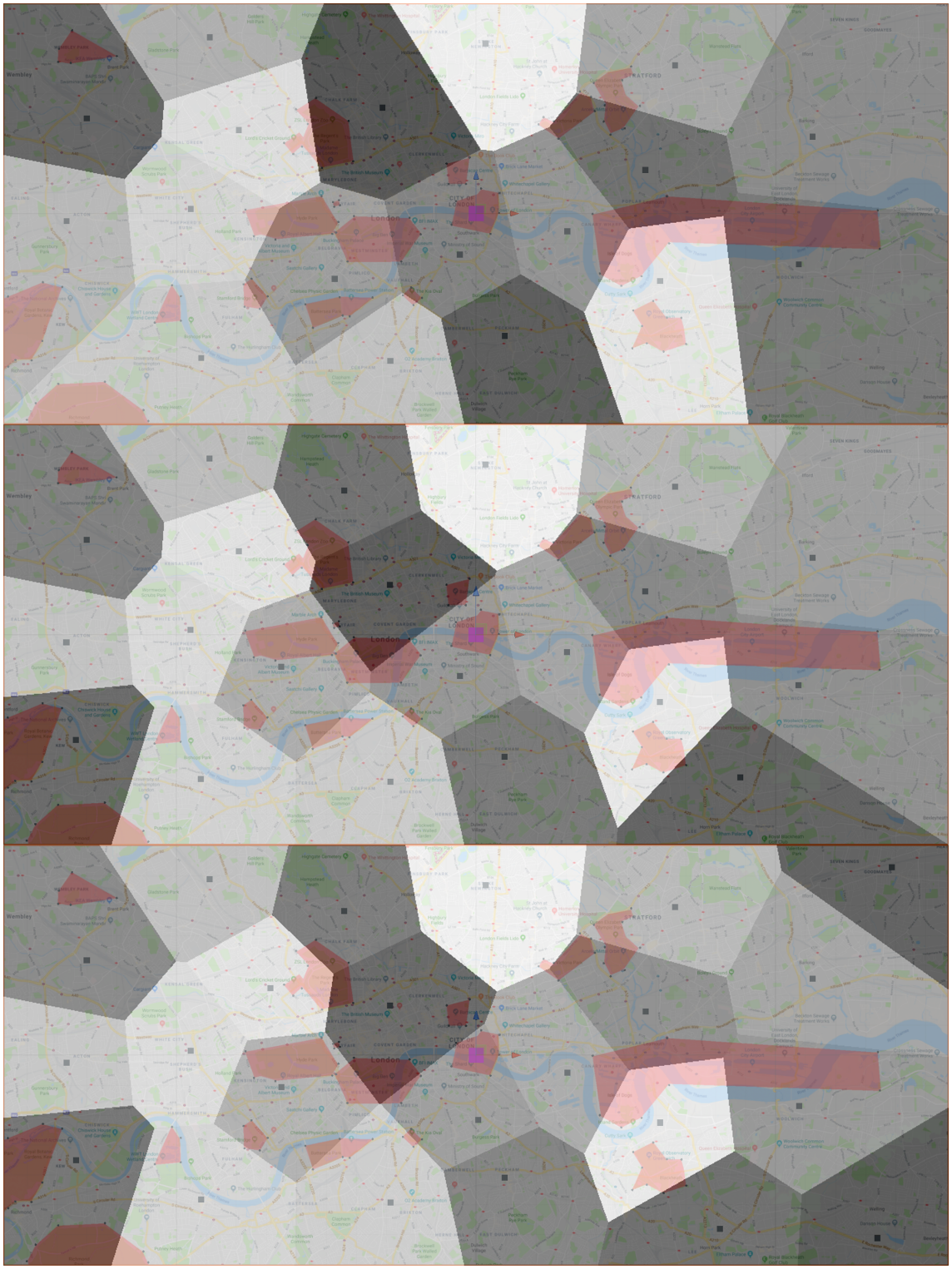


Figure 10.4: Controller Placement Iteration.

After performing a few rounds of this analysis, we arrive at a distribution of Controllers that we are happy with for now. Although we could continue to generate smaller cells by adding more Controllers to the world, we have to think practically in two ways.

Firstly, would a real distributed drone delivery network have over 60 depots or hubs in just one city? Even Amazon, master strategician to one of the world's largest delivery networks, works by having a few major warehouses instead of several minor safehouses scattered across the place.

Secondly, as per our earlier load balancing configuration, SpatialOS will attempt to place each Controller on its own worker. If we continue to add Controllers and reduce the average cell size, each Controller would be responsible for a smaller region of land. Below some minimum threshold of area covered, running a worker may be overkill and a complete waste of compute power.

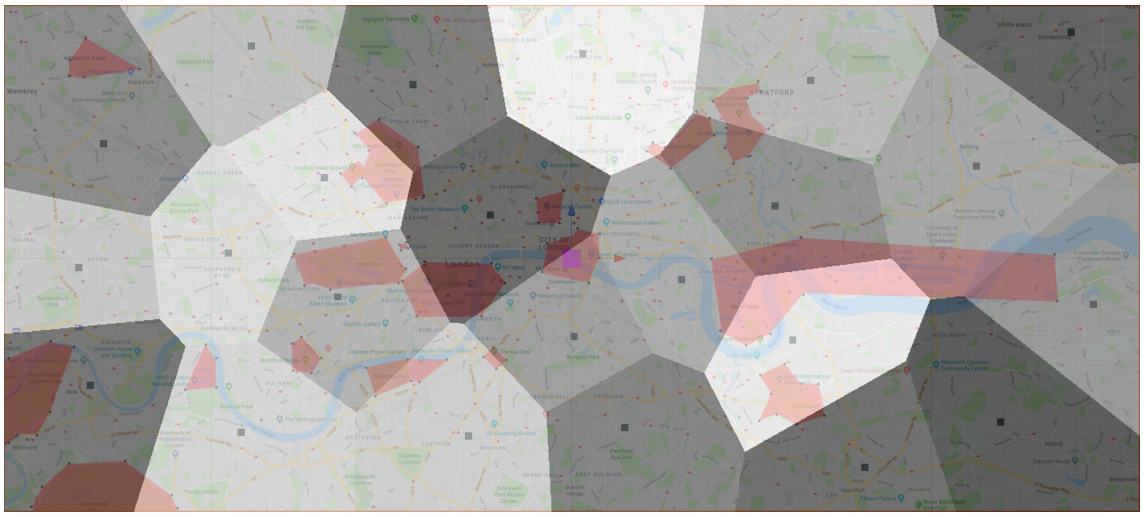


Figure 10.5: Final Controller Placement.

The full list of Controller locations can be seen in Appendix B.2.

10.4.3 Drone Launch & Landing Pads

To reduce the chances of drones colliding upon launch or return to a Controller, we introduce the concept of a launch pad. Drones begin their journeys from a launch pad at a specified offset from the Controller entity's location, and end their journeys at a return pad at an equal but opposite distance from the Controller location.

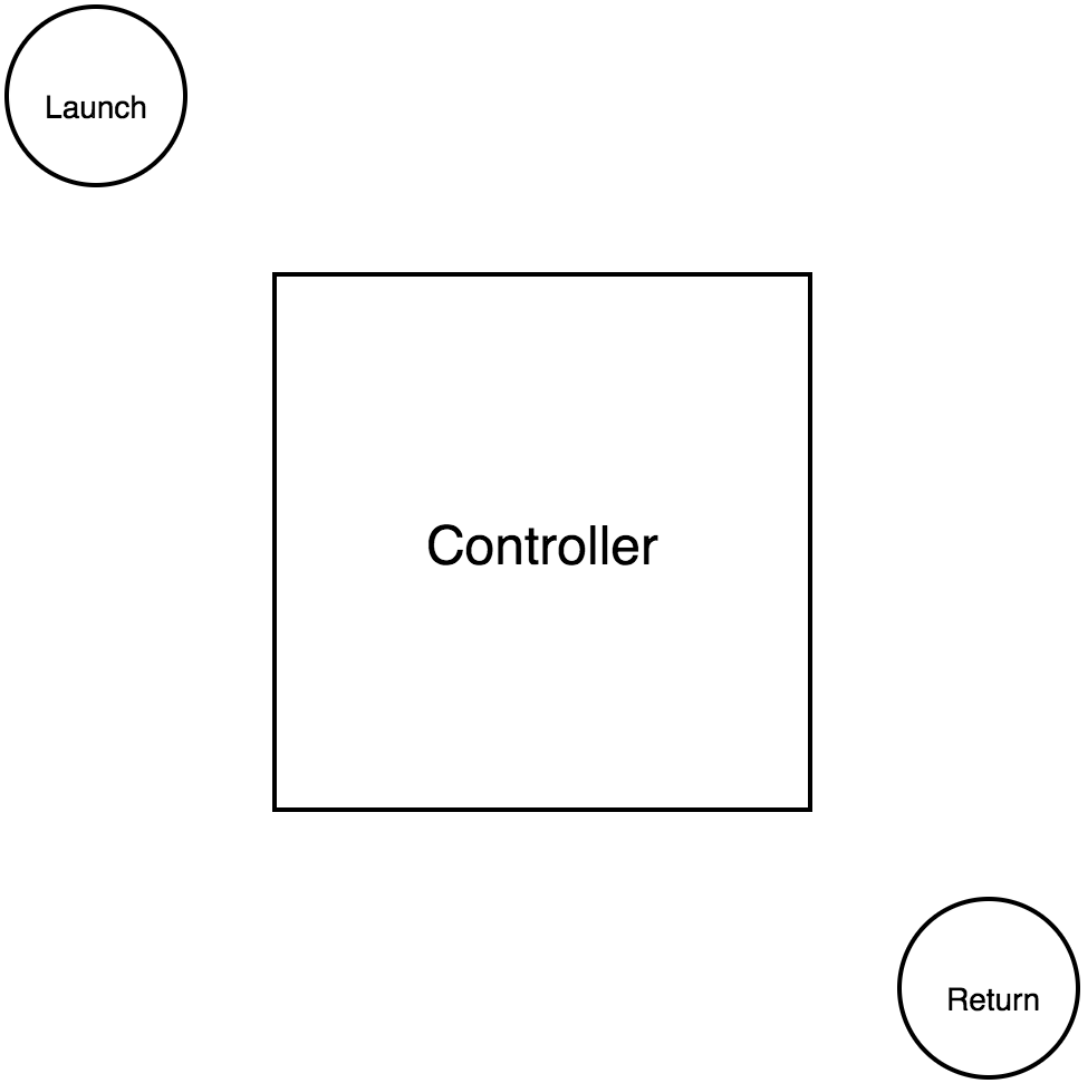


Figure 10.6: Demonstrating the offsets of the launch and return pads.

By treating each pad like a circle, every launch and return position is randomly assigned to somewhere on the pad. This means that drones are extra unlikely to crash both when launching off to perform their deliveries and also when landing back at base to refuel.

Chapter 11

Scaling with SpatialOS

No matter how much is done to make the logic of the simulation as efficient and scalable as possible, there always comes a time where we must look at modifying the SpatialOS worker and launch configurations in order to make better use of the underlying platform.

11.1 Load Balancer

One obvious configuration change we have made and kept since the beginning is to use the Point of Interest based load balancer. By defining Controllers as the central authority over the land closest to them, and telling SpatiaOS to split each Controller off onto its own worker using a similar land division strategy, we minimise the inter-worker communication overheads immensely.

In fact, the only intentional cross-worker communication is from the OrderGenerator to a target Controller when sending a new `DeliveryRequest` at its defined rate. Essentially, by distributing the raw workload in a similar way to the way we route delivery requests across Controllers, we obtain an efficient, parallel, distributed simulation.

To help with updating these parameters when a change was made to the distribution of Controllers, we added an option to the drop-down menu to return the waypoints of every single one of them in a format that could be directly copied into the load balancer configuration.

11.2 Chunk Size

In SpatialOS, a chunk is the smallest unit of area that a world can subdivided into. Every entity belongs to only one chunk, and each chunk has its own overheads because it performs operations to sync with the SpatialOS runtime.

By default, the chunk size was set to 50 in our configuration file. For a default world of size 2000 x 2000, that corresponds to 1600 chunks. On the other hand, for our world of size 32000 x 14500 we would be creating 185,600 chunks, which could correspond to over a 100-fold increase in memory, networking and CPU usage!

In a post on their forums, one of Improbable's engineers (Callum Brighting) describes a good approach to take when scaling SpatiaOS[17]. In his reply, Callum mentions that one of the company's clients operates a 40km game world with a chunk size of 250m. As our world is of a similar size, we opt to set our chunk size parameter to 250 too.

Now if were to do the calculations again with our updated chunk size, we find out that our world consists of 7424 chunks - a far more reasonable figure than 185,600.

11.3 Entity Interest Range

The second configuration change we make is to the entity interest range. This variable is used to define what entities a worker gets component updates for. Obviously workers authoritative over a specific area will get updates on entities in that area, but they also need to know about what is happening **nearby**.

Given a worker with authority over entity E. The worker gets updates for the entire chunk that E is in. However, if there is a neighbouring chunk with any point inside this radius from entity E, the worker gets updates on this chunk too. By reducing the entity interest range, we can therefore reduce the number of neighbouring chunks that a worker has to check out and receive updates for.

Remembering that the chunk size used to be 50, our initial entity interest range of 200 would check out the neighbouring 4 chunks for every entity the worker was authoritative over. Furthermore, was there any need for the worker to be worrying about the 200m surrounding every one of its entities?

For our drone simulation, we know that each drone gets information about static obstacles from the Controller. The only information that it needs about its surrounds is regarding what drones are nearby. Since our drone-side collision avoidance never checks beyond a radius of 25m, we can set our entity interest radius to be the same value.

11.4 Problems

11.4.1 Worker Genocide

11.4.2 OrderGenerator Downtime

11.4.3 Max Unacked Pings Rate

Part V

Findings

Chapter 12

Evaluation

To critically evaluate this project, we first define some important metrics that provide useful insight when comparing different simulations. We then use these metrics to study the standard-scale performance of the three schedulers implemented. Following on, we assess the impact of scaling up the simulation. A discussion on the successes and failures of the project then concludes our evaluation.

12.1 Metrics

In order to understand how our delivery network and different scheduling models have performed, there needs to be some way to extract useful metrics about each simulation. Although the online SpatialOS portal gives us the ability to view all sorts of statistics and runtime metrics about a deployment, we want to delve deep and understand our actual simulation better.

To find out more about the inner workings of our simulated delivery networks, we created a couple of schema components purely to maintain the latest state of important simulation metrics. In addition to this, we implemented a method in the Controller that prints a log of important data stored in these components every 60 seconds.

This data is then displayed in the SpatialOS logs view on a deployment. To transfer this into a format we can perform analysis on, we have also implemented a browser-based tool that iterates through the logger view and extracts the logged metrics to a local file.

As the focus of the project lies with the operation of a delivery network, the important metrics for us are those that directly relate to the state of the Controllers and not necessarily the scale of the simulation.

We preface each log string with the "METRICS" keyword, to make it easy to filter our logs from the runtime logs. Since each log is printed by a Controller, we keep track of the Controller's EntityId on each log so that we can investigate how Controllers in different regions of the world may behave relative to each other.

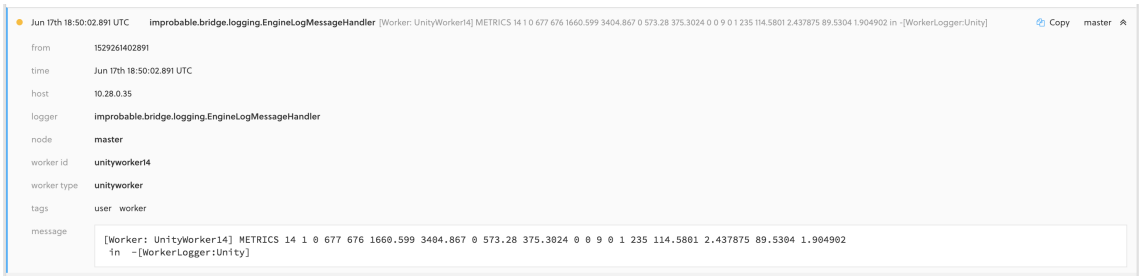


Figure 12.1: Expanded log entry with our selected metrics.

Knowing the number of drones in operation and length of the request queue is incredibly useful to determine the load on a Controller. For example, if the queue size is often a low number then we know that all incoming requests are going to be served as there would no eviction occuring.

In order to calculate the profitability of a Controller, we need to know the revenue, costs and any penalties that may have been paid. Keeping track of the number of deliveries and completed round trips also allows us to compute the average profit per delivery down the line. Another interesting financial figure is the potential lost by each scheduler. This is the sum of expected values for the deliveries that were evicted or rejected from a scheduler. We also track the average potential lost so we can determine which scheduler is better at evicting requests from the queue.

In addition to the metrics defined in this section, there are values that represent the number of failed actions by a Controller. Controllers do not fail often enough anymore for these values to provide insight, but they have been left in for legacy debugging purposes.

12.2 Comparing Schedulers

12.3 Large Scale London Simulation

12.4 section name

Chapter 13

Conclusion

Chapter 14

Future Work

Bibliography

[1] Amazon.com Inc. *Amazon Prime Air*. 2013. URL: https://youtu.be/98BIu9dpwHU (visited on 06/11/2018).

[2] Amazon.com Inc. *Amazon Prime Air*. 2015. URL: https://www.youtube.com/watch?v=MXo%7B%5C_%7Dd6tNWuY (visited on 06/11/2018).

[3] Amazon.com Inc. *Amazon Prime Air Homepage*. 2018. URL: https://www.amazon.com/b?node=8037720011%7B%5C&%7Dref=tsm%7B%5C_%7D1%7B%5C_%7Dyt%7B%5C_%7Ds%7B%5C_%7Damzn%7B%5C_%7D290791026 (visited on 06/11/2018).

[4] Amazon.com Inc. *Amazon Prime Air's First Customer Delivery*. 2016. URL: https://www.youtube.com/watch?v=vNySOrI2Ny8 (visited on).

[5] Amazon.com Inc. "Determining Safe Access with a Best- Equipped, Best-Served Model for Small Unmanned Aircraft Systems". In: *NASA UTM 2015: The Next Era of Aviation* (2015). URL: https://images-na.ssl-images-amazon.com/images/G/01/112715/download/Amazon%7B%5C_%7DDetermining%7B%5C_%7DSafe%7B%5C_%7DAccess%7B%5C_%7Dwith%7B%5C_%7Da%7B%5C_%7DBest-Equipped%7B%5C_%7DBest-Served%7B%5C_%7DModel%7B%5C_%7Dfor%7B%5C_%7DsUAS.pdf.

[6] Amazon.com Inc. *Fulfilment by Amazon (FBA) Programme Fees*. 2017.

[7] Amazon.com Inc. "Revising the Airspace Model for the Safe Integration of Small Unmanned Aircraft Systems". In: *NASA UTM 2015: The Next Era of Aviation* 1 (2015), pp. 2–5. URL: https://images-na.ssl-images-amazon.com/images/G/01/112715/download/Amazon%7B%5C_%7DRevising%7B%5C_%7Dthe%7B%5C_%7DAirspace%7B%5C_%7DModel%7B%5C_%7Dfor%7B%5C_%7Dthe%7B%5C_%7DSafe%7B%5C_%7DIntegration%7B%5C_%7Dof%7B%5C_%7DsUAS.pdf.

[8] Jillian Ambrose. *Electricity prices hit 10-year high as cheap wind power wanes*. 2018. URL: https://www.telegraph.co.uk/business/2018/03/02/electricity-prices-hit-10-year-high-cheap-wind-power-wanes/ (visited on 05/20/2018).

[9] Paul Balaji, David Cattle, Andrea Janoscikova, Galina Peycheva, Jan Matas, and Samuel Wood. *AATC Visualiser*. 2017. URL: https://samuknet.github.io/aatc-visualiser/ (visited on 01/24/2018).

[10] Paul Balaji, David Cattle, Andrea Janoscikova, Galina Peycheva, Jan Matas, and Samuel Wood. *Autonomous Air Traffic Control*. Tech. rep. 2017, p. 39.

[11] BBC News. *'Drone' hits BA plane: Police investigate Heathrow incident*. 2016. URL: http://www.bbc.co.uk/news/uk-36069002 (visited on 01/25/2018).

[12] BBC News. *First UK police drone unit launched in Devon, Cornwall and Dorset*. 2017. URL: http://www.bbc.co.uk/news/uk-england-devon-40595540 (visited on 01/25/2018).

[13] BBC News. *Passenger jet approaching Heathrow in drone 'near-miss'*. 2017. URL: http://www.bbc.co.uk/news/uk-england-london-39457371 (visited on 01/23/2018).

[14] John Bell. *CPU Scheduling*. 2018. URL: https://www.cs.uic.edu/%7B~%7Djbell/CourseNotes/OperatingSystems/5%7B%5C_%7DCPU%7B%5C_%7DScheduling.html (visited on 01/26/2018).

[15] Mark Blunden. *No-go drone zones announced in London amid fears over voyeurism and safety*. 2016. URL: https://www.standard.co.uk/news/techandgadgets/nogo-drone-zones-announced-in-london-amid-fears-over-voyeurism-and-safety-a3397811.html.

[16] Owen Bowcott. *Laws for safe use of driverless cars to be ready by 2021*. 2017. URL: https://www.theguardian.com/law/2017/dec/14/laws-safe-use-driverless-cars-ready-2021-law-commission (visited on 01/25/2018).

[17] Callum Brighting. *Crowd Sim In SpatialOS: How to scale a deployment?* 2017. URL: https://forums.improbable.io/t/crowd-sim-in-spatialos-how-to-scale-a-deployment/2547 (visited on 01/17/2018).

[18] Rob Davies. *Drone flew 'within wingspan' of plane approaching Heathrow*. 2017. URL: https://www.theguardian.com/technology/2017/mar/31/drone-wingspan-plane-approaching-heathrow-near-misses (visited on 01/25/2018).

[19] Clay Dillow. *This unsexy technology is set to revolutionize the drone industry*. 2015. URL: http://fortune.com/2015/05/05/drone-technology/ (visited on 01/25/2018).

[20] DJI. *DJI Guidance*. URL: https://www.dji.com/guidance (visited on).

[21] DJI. *DJI Intelligent Flight Modes*. URL: https://www.dji.com/intelligent-flight-modes (visited on 06/12/2018).

[22] Flightradar24. *About Flightradar24*. 2018. URL: https://www.flightradar24.com (visited on 01/25/2018).

[23] Samuel Gibbs. *Uber plans to buy 24,000 autonomous Volvo SUVs in race for driverless future*. 2017. URL: https://www.theguardian.com/technology/2017/nov/20/uber-volvo-suv-self-driving-future-business-ride-hailing-lyft-waymo (visited on 01/25/2018).

[24] Google. *"Drone Technology" Google Trends*. 2018. URL: https://trends.google.co.uk/trends/explore?date=all%7B%5C&%7Dq=drone%20technology (visited on 01/26/2018).

[25] Google. *London Snapshot Small*. 2018. URL: https://www.google.com/url?q=https://www.google.com/maps/@51.5130906,-0.1031115,14z%7B%5C&%7Dsa=D%7B%5C&%7Dust=1528914428694000 (visited on 06/15/2018).

[26] Kristen Hall-Geisler. *Google Gets a Patent for an Autonomous Delivery Truck*. 2016. URL: https://www.popsci.com/google-gets-patent-for-an-autonomous-delivery-truck (visited on 01/25/2018).

[27] Kristin Hohenadel. *Skype Co-Founders Want to Overhaul Local Delivery With Sidewalk Robots*. 2015. URL: http://www.slate.com/blogs/the%7B%5C_%7Deye/2015/11/02/starship%7B%5C_%7Dtechnologies%7B%5C_%7Dis%7B%5C_%7Dbuilding%7B%5C_%7Dself%7B%5C_%7Ddriving%7B%5C_%7Dsidewalk%7B%5C_%7Drobots%7B%5C_%7Dthat%7B%5C_%7Dwill.html (visited on 01/25/2018).

[28] Improbable Worlds Ltd. *Improbable Forums*. 2018. URL: https://forums.improbable.io/ (visited on 01/25/2018).

[29] Improbable Worlds Ltd. *Improbable Home Page*. 2018. URL: https://improbable.io/ (visited on 10/23/2017).

[30] Improbable Worlds Ltd. *Load Balancing: Points of Interest*. 2018. URL: https://docs.improbable.io/reference/13.0/shared/worker-configuration/loadbalancer-config%7B%5C#%7Dpoints-of-interest (visited on 06/15/2018).

[31] Improbable Worlds Ltd. *Quest- an iOS io game from Improbable*. 2017. URL: https://improbable.io/games/blog/quest-an-ios-io-game-from-improbable-io (visited on 11/12/2017).

[32] Improbable Worlds Ltd. *SpatialOS Technical Breakdown*. 2018. (Visited on 01/25/2018).

[33] Improbable Worlds Ltd. *What is SpatialOS?* 2018. URL: https://docs.improbable.io/reference/12.0/shared/concepts/spatialos%7B%5C#%7Ddeveloping-with-spatialos (visited on 01/23/2018).

[34] Improbable Worlds Ltd. *What we found when we simulated the backbone of the entire Internet on SpatialOS*. 2016. URL: https://improbable.io/company/news/2016/03/24/what-we-found-when-we-simulated-the-backbone-of-the-entire-internet-on-spatialos (visited on 11/18/2017).

[35] Investopedia. *Time Value of Money - TVM*. URL: https://www.investopedia.com/terms/t/timevalueofmoney.asp (visited on 06/11/2018).

[36] Mark Jarman, John Vesey, and Paul Febvre. "UAVs for UK Agriculture : Creating an Invisible Precision Farming Technology - White Paper". 2016. URL: https://sa.catapult.org.uk/wp-content/uploads/2016/07/White-paper-UAVs-and-agriculture%7B%5C_%7DFinal2.pdf.

[37] Martin Joerss, Florian Neuhaus, Christoph Klink, and Florian Mann. "Parcel delivery The future of last mile". In: September (2016). URL: https://www.mckinsey.com/%7B~%7D/media/mckinsey/industries/travel%20transport%20and%20logistics/our%20insights/how%20customer%20demands%20are%20reshaping%20last%20mile%20delivery/parcel%7B%5C_%7Ddelivery%7B%5C_%7Dthe%7B%5C_%7Dfuture%7B%5C_%7Dof%7B%5C_%7Dlast%7B%5C_%7Dmile.ashx.

[38] Sunghun Jung and Hyunsu Kim. "Analysis of Amazon Prime Air UAV Delivery Service Analysis of Amazon Prime Air UAV Delivery Service". In: June (2017).

[39] Kia Kokalitcheva. *This Cute Self-Driving Robot Wants to Deliver Your Food or Laundry*. 2016. URL: http://fortune.com/2016/04/06/dispatch-carry-delivery-robot/ (visited on 01/25/2018).

[40] Robin Koontz. *Drones to the Rescue*. 2014. URL: http://www.kidsdiscover.com/teacherresources/drones-uavs-rescue/ (visited on 01/24/2018).

[41] M Lammert. "Twelve-Month Evaluation of UPS Diesel Hybrid Electric Delivery Vans". In: *Contract* December (2009). DOI: 10.2172/969727.

[42] Mike Laris. *Driverless delivery robots could be hitting D.C. sidewalks soon*. 2016. URL: http://www.chicagotribune.com/bluesky/technology/ct-driverless-delivery-robots-20160328-story.html (visited on 01/25/2018).

[43] Hau L. Lee, Yiwen Chen, Barchi Gillai, and Sonali Rammohan. "Technological Disruption and Innovation in Last-Mile Delivery". In: *Stanford Business* June (2016), pp. 1–26. URL: https://www.gsb.stanford.edu/sites/gsb/files/publication-pdf/vcii-publication-technological-disruption-innovation-last-mile-delivery.pdf.

[44] Prashob Menon. *HOW MANY DRONES DOES AMAZON NEED?* 2013. URL: http://iveybusinessreview.ca/blogs/pmenonhba2010/2013/12/11/how-many-drones-does-amazon-need/ (visited on 01/26/2018).

[45] Microsoft. *SortedSet¡T¿ Class*. 2018. URL: https://msdn.microsoft.com/en-us/library/dd412070(v=vs.110).aspx (visited on 06/17/2018).

[46] Melanie Mitchell. "An introduction to genetic algorithms". In: *Computers & Mathematics with Applications* 32.6 (1996), p. 133. ISSN: 08981221. DOI: 10.1016/S0898-1221(96)90227-8. URL: https://books.google.ca/books/about/An%7B%5C_%7DIntroduction%7B%5C_%7Dto%7B%5C_%7DGenetic%7B%5C_%7DAlgorithms.html?id=0eznlz0TF-IC%7B%5C&%7Dpgis=1%7B%5C%%7D5Cnhttp://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20%7B%5C&%7Dpath=ASIN/0262631857%7B%5C%%7D5Cnhttp://linkinghub.elsevier.com/retrieve/pii/S0898122196902278.

[47] Jussi Myllymaki. *Autonomous Delivery Platform*. 2013. URL: http://pdfpiw.uspto.gov/.piw?PageNum=0%7B%5C&%7Ddocid=09256852%7B%5C&%7DIDKey=EA66D62BF248%7B%5C%%7D0D%7B%5C%%7D0A%7B%5C&%7DHomeUrl=http%7B%5C%%7D3A%7B%5C%%7D2F%7B%5C%%7D2Fpatft.uspto.gov%7B%5C%%7D2Fnetacgi%7B%5C%%7D2Fnph-Parser%7B%5C%%7D3FSect1%7B%5C%%7D3DPTO2%7B%5C%%7D2526Sect2%7B%5C%%7D3DHITOFF%7B%5C%%7D2526p%7B%5C%%7D3D1%7B%5C%%7D2526u%7B%5C%%7D3D%7B%

5C%%7D25252Fnetahtml%7B%5C%%7D25252FPTO%7B%5C%%7D25252Fsearch-bool.html%7B%5C%%7D2526r%7B%5C%%7D3D25%7B%5C%%7D2526.

[48] Herman Narula. *Google Cloud Next '17 — Herman Narula Day 3 Keynote*. 2017. URL: https://youtu.be/fTuijWHwAsk?t=4m46s (visited on 01/25/2018).

[49] A. Nash, S. Koenig, and C. Tovey. "Lazy Theta *: Any-Angle Path Planning and Path Length Analysis in 3D". In: *Third Annual Symposium on Combinatorial Search (SOCS-10) Lazy* (2010), pp. 153–154.

[50] Jonathan Roberts and Unlike Formula. *What is FPV drone racing ?* 2016. URL: https://phys.org/news/2016-02-fpv-drone.html.

[51] Aaron Souppouris. *Skype co-founders build delivery bot that rides on sidewalks*. 2015. URL: https://www.engadget.com/2015/11/02/starship-technologies-local-delivery-robot/ (visited on 01/25/2018).

[52] Peter Stearns. *The Encyclopedia of world history : ancient, medieval, and modern, chronologically arranged*. Boston: Houghton Mifflin, 2001. ISBN: 0395652375.

[53] Tesla Inc. *Tesla Autopilot*. 2018. URL: https://www.tesla.com/en%7B%5C_%7DGB/autopilot (visited on 01/25/2018).

[54] The AA. *April 2018 fuel prices*. 2018. URL: http://www.theaa.com/driving-advice/driving-costs/fuel-prices (visited on 05/25/2018).

[55] Tansel Uras and Sven Koenig. "An Empirical Comparison of Any-Angle Path-Planning Algorithms". In: *Aaai* (2015). DOI: 10.1002/bse.

[56] Richard Verrier. *Drones are providing film and TV viewers a new perspective on the action*. 2015. URL: http://www.latimes.com/entertainment/envelope/cotown/la-et-ct-drones-hollywood-20151008-story.html (visited on 01/25/2018).

[57] Dan Wang. *Drone Delivery Economics*. URL: https://www.flexport.com/blog/drone-delivery-economics (visited on 01/26/2018).

[58] Adrienne Welch. "A cost-benefit analysis of Amazon Prime Air A Cost-Benefit Analysis of Amazon Prime Air". In: (2015), p. 57.

[59] Lihua Zhu, Xianghong Cheng, and Fuh-Gwo Yuan. "A 3D collision avoidance strategy for UAV with physical constraints". In: *Measurement* 77.Complete (2016), pp. 40–49. DOI: 10.1016/j.measurement.2015.09.006.

Appendices

Appendix A

Code Listings

A.1 Static No Fly Zone Class

Listing 16: Static No Fly Zone Class

```
public static class NoFlyZone
{
    private static bool isInPolygon(
      Improbable.Controller.NoFlyZone nfz,
      Improbable.Vector3f point);

    public static bool hasCollidedWith(
      Improbable.Controller.NoFlyZone nfz,
      Improbable.Vector3f point);

    public static bool hasCollidedWithAny(
      List<Improbable.Controller.NoFlyZone> zones,
      Vector3f point);

    public static void setBoundingBoxCoordinates(
      ref Improbable.Controller.NoFlyZone nfz);

    public static bool isPointInTheBoundingBox(
      Improbable.Controller.NoFlyZone nfz,
      Improbable.Vector3f point);
}
```

A.2 Nearby Drone and Collision Detection

Listing 17: Nearby Drone and Collision Detection

```
private void CheckForNearbyDrones(Vector3 dronePos, bool collisionsOn)
{
  nearestDrone.type = APFObstacleType.NONE;
  nearestDrone.position = new Vector3f(0, -1, 0);
  nearestDroneDistance = radiusOfInfluence;

  Collider[] hitColliders = Physics.OverlapSphere(dronePos, apfRadius);
  foreach (Collider hitCollider in hitColliders)
  {
    if (hitCollider.gameObject.EntityId() != gameObject.EntityId())
    {
      float currentDroneDistance
        = Vector3.Distance(hitCollider.transform.position, dronePos);

      if (collisionsOn && currentDroneDistance < 1)
      {
        /* collision reported to controller */
      }

      if (currentDroneDistance < nearestDroneDistance)
      {
        nearestDrone.position
          = hitCollider.transform.position.ToSpatialVector3f();
        nearestDrone.type = APFObstacleType.DRONE;
        nearestDroneDistance = currentDroneDistance;
      }
    }
  }
}
```

A.3 Main Controller Loop

Listing 18: Main Controller Loop

```
void ControllerTick()
{
  if (!ControllerWriter.Data.initialised)
  {
    /* initialise global layer */
  }

  QueueEntry nextRequest;
  if (ReadyForDeployment())
  {
    if (scheduler.GetNextRequest(out nextRequest))
    {
      HandleDeliveryRequest(nextRequest);
    }
  }

  scheduler.UpdateDeliveryRequestQueue();
}
```

A.4 Updated DeliveryRequest Generation

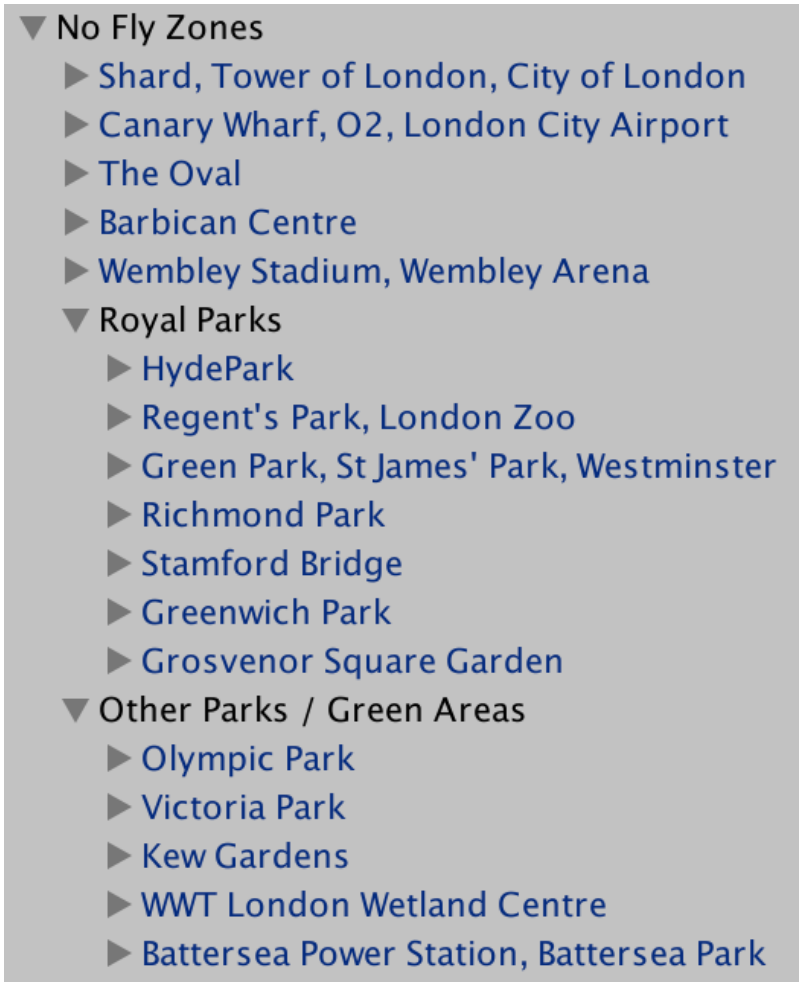
Listing 19: Additions to DeliveryHandler Schema

```
enum PackageType {
  LETTER_SMALL = 0;
  LETTER_LARGE = 1;
  ENVELOPE_SMALL = 2;
  ENVELOPE_STANDARD = 3;
  ENVELOPE_LARGE = 4;
  PARCEL = 5;
}

type PackageInfo {
  PackageType type = 1;
  float weight = 2;
}

enum DeliveryType {
  STANDARD = 0;
  PRIORITY = 1;
  URGENT = 2;
  SUPER_PRIORITY = 3;
}
```

Appendix B

Large London Snapshot

B.1 List of No Fly Zones



B.2 List of Controller Locations