Simulating Urban Air Mobility Supply

by

Lisa Y. Yoo

B.S. Computer Science and Engineering, Massachusetts Institute of Technology (2021)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2022

Author . . . .
Department of Electrical Engineering and Computer Science
January 21, 2022

Certified by . . . .
Ali Shamshiripour
Research Scientist
Thesis Supervisor

Certified by . . . .
Moshe Ben-Akiva
Professor of Civil Engineering
Thesis Supervisor

Accepted by . . . .
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Simulating Urban Air Mobility Supply

by

Lisa Y. Yoo

Submitted to the Department of Electrical Engineering and Computer Science
on January 21, 2022, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Urban air mobility (UAM) is a relatively new concept in the transportation industry. As on-demand services like Uber and Lyft have transformed our daily lives, our objective is to explore how on-demand UAM impacts mobility patterns by modeling the supply-side of such a service within a realistic, high-fidelity simulation. We present a design and implementation of UAM within SimMobility, a multi-scale, multi-modal activity- and agent-based simulation software, which was developed in the MIT Intelligent Transportation Systems (ITS) Lab. This includes a network of vertiports, fleet of UAM aircrafts, and controller logic to accommodate passenger requests and control the fleet. We also implement novel service features including priority landing, stand designation, and matching algorithm customization through parameterized buffer times. Explicit models to simulate key characteristics of UAM services, supported by a comprehensive review of the underlying literature, has enabled us to develop a uniquely realistic simulation consistent with state-of-the-art technological developments, as well as the current urban landscape. The contribution of this thesis is twofold: first in the realistic simulation of UAM supply as described, and second in providing a replicable architecture that can be emulated for future SimMobility mobility service controllers.

Thesis Supervisor: Ali Shamshiripour
Title: Research Scientist

Thesis Supervisor: Moshe Ben-Akiva
Title: Professor of Civil Engineering

Acknowledgments

This research was supported and funded by Ferrovial, to whom I would like to extend a special thanks. Additionally, I would like to thank Kexin Chen for all of her work done on the demand-side of the project and her help in designing the UAM service operation, as well as Dr. Ali Shamshiripour, Professor Ravi Seshadri, Professor Md Sami Hasnine, Dr. Andre Romano Alho, and Professor Moshe Ben-Akiva for their direction and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The recent development of advanced air mobility (AAM) offers short-haul air travel to a wider audience for a variety of applications. There are several potential applications of AAM that have been discussed in the transportation community, from package delivery services to passenger travel [9]. One flavor of AAM of growing interest to the transportation industry is urban air mobility (UAM), which refers to short-haul air travel for passengers in an urban context. For the scope of this thesis, we focus on UAM as an on-demand mobility service. Passengers request rides to various destinations within a city or to another city. Current similar on-demand services for ground transport, namely Uber and Lyft, have radically disrupted transportation in the past few years. Considering how the problem of commuting and travel within a congested urban area is a persistent question for transportation research, UAM offers a novel way to circumvent some city congestion by diverting some traffic to the air network, away from ground travel.

Some existing literature attempts to model demand [3, 7, 8, 9, 10], while other research characterizes the supply of potential UAM services. On the supply-side, different works have proposed specifications for UAM infrastructure, aircrafts, and operations – this includes details on aircraft attributes, vertiport locations, and flight logistics [1, 3, 7, 8, 9, 10, 12, 17, 18, 20]. Although there is growing literature on the subject, there are still gaps to be filled in order to realistically model both supply and demand. There is very little existing research that simulates UAM in a realistic, high-fidelity, agent-based and activity-based simulation, taking supply, demand, and their interactions all into consideration. Though there is some work on this as seen in MATSim [17], this area of study remains underdeveloped, missing key features like rebalancing, charging, and vertiport-level operations for UAM. We aim to bridge these gaps by designing, implementing, and testing a more realistic UAM service within SimMobility.

SimMobility is a state-of-the-art, multi-scale, integrated, activity- and agent-based mobility simulator developed by the MIT Intelligent Transportation Systems (ITS) Lab. The demand and supply components are fully integrated in SimMobility with proper feedback loops, allowing for a realistic simulation of how changes in supply characteristics influence all travel related choices [2]. The software enables modeling of the plans and actions of millions of agents – including agents from pedestrians to drivers, infrastructure from traffic lights to GPS, vehicles from cars to trains, and time periods from a second-level granularity to year-level granularity. As a result, the software allows us to simulate and observe the effects from different technology and policy scenarios [5, 15, 16].

This thesis is part of a higher-level research project at the ITS Lab, to comprehensively model both the supply and demand of UAM and observe its effects in a typical day of operations in various prototypical cities [15]. The overarching goal is to augment SimMobility to accommodate UAM scenarios. This entails prototype city generation and the development and calibration of different behavioral models on the demand-side. On the supply-side, the UAM service operations must be realistically designed and parameterized, and the codebase needs to be able to accommodate this conceptual design.

Within this larger project, the contribution of this thesis is the design and implementation of the supply-side of an on-demand UAM service integrated with SimMobility. More in detail, this includes the effective simulation of vertiport and aircraft operations as well as complex algorithms for matching and rebalancing among other roles of the hypothetical UAM controller mimicking a UAM service provider in the real world. This new code must also be integrated into SimMobility, feeding into and

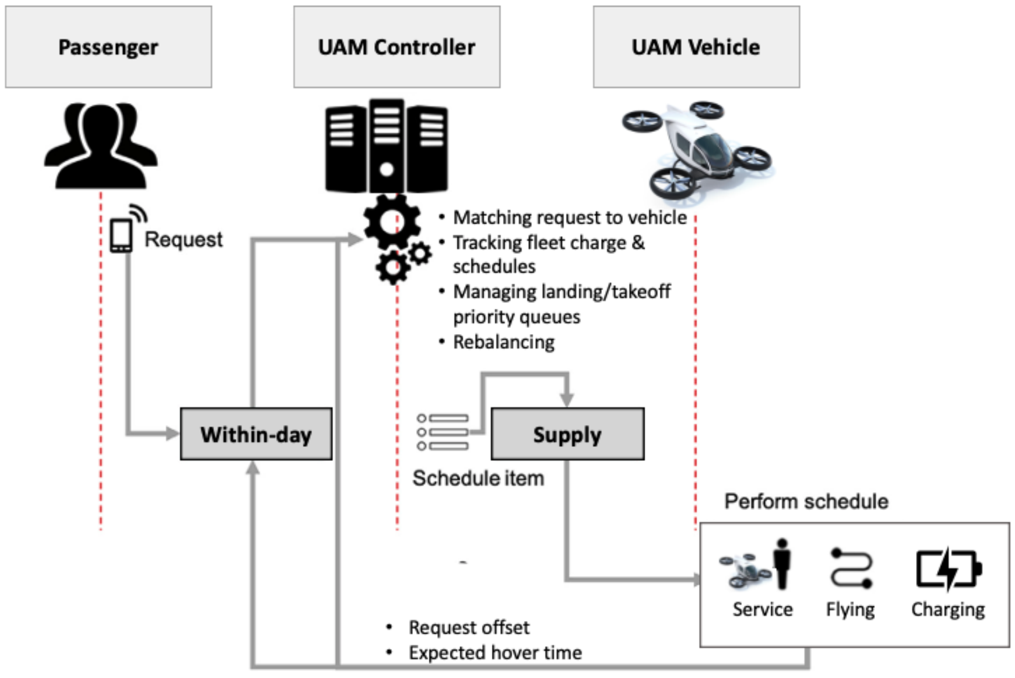


Figure 1-1: Desired components and behavior for a UAM service controller within SimMobility.

learning from results in tandem with SimMobility. The details of this are further discussed in the next section.

In order to augment SimMobility for the UAM scenario for the supply-side, we need effective simulation of all parts of a UAM service – from interfacing with passengers trying to use the service, all the way to maintaining a network of vertiports and fleet of aircrafts to accommodate these passengers. The high-level components for a UAM service within SimMobility are shown in Figure 1-1.

A passenger within the population will submit a request to take a trip to the UAM service, represented by the UAM controller in SimMobility. The UAM controller fulfills a variety of functions, including matching trip requests to aircrafts, tracking the status and schedules of all the aircrafts in its fleet, managing vertiport operations, and rebalancing the fleet. The controller takes the passenger request and attempts to assign this trip to an available aircraft. This UAM aircraft will perform the schedule by serving passengers (picking up and dropping off), flying through the air network, and charging.

The contributions of this thesis are two-fold: (1) to effectively simulate UAM supply as described above; (2) to provides a replicable software architecture that can be emulated for future external controllers and modules to work with SimMobility. The novel architecture we propose, instead of implementing the UAM controller directly within the C++ code of SimMobility, implements this in a separate module external to SimMobility. With this architecture, while there is no significant loss of information, enables a level of flexibility and customization which is favorable for projects with not much initial information and a need for several iterations to converge on an optimal service design. Leveraging this UAM controller, we are able to model the supply of UAM accurately. We use the newly designed and implemented controller to observe the impact of various UAM operations parameters including the fleet size on the performance of the simulated UAM service.

The rest of this manuscript will adhere to the following structure. In section 2, we evaluate literature on UAM to derive elements of our design and to identify gaps in existing research. In section 3, we provide a detailed look into the methodology, by describing the software needs based on our designed UAM scenario, then showing how our implementation meets these needs. In section 4, we analyze results from running the controller. In section 5, we conclude and reflect on our findings, while providing some possible directions for future work.

Chapter 2

Related Works

Although urban air mobility is a relatively new concept, the underlying literature has been growing at a fast pace, approaching the problem from many different angles. Existing research on UAM has focused on both on the demand side [3, 7, 8, 9, 10, 20] and the supply side as reviewed in this section.

On the supply-side, different studies have proposed UAM service, aircraft, and vertiport operational specifications, which we took into account for the design of the UAM supply for this thesis. The literature largely assumes the Electric Vertical Take-Off and Landing (eVTOL) technology, which as the name suggests, enables vertical take-off and landing [3, 7, 8, 9, 10, 18]. In an urban setting, vertical take-off and landing helps conserve valuable space. Passenger capacity ranges from 2-5 seats per aircraft [3, 8, 9, 10, 18]. Both the proposed aircraft specifications in these studies and actual eVTOLs under development claim that aircrafts will be able to fly at a cruising speed in the range of 125-175 km/h [1, 3, 7]. The ranges of trips that these aircrafts could serve varies widely – some research considered shorter distances up to 100 km [3, 8, 10], while other studies considered longer trip ranges of up to 480 km [7]. Having heterogeneous fleets composed of different aircrafts is explored in the literature, but most literature worked with homogeneous fleets [3, 8, 10]. Kim et. al delve into this deeper and find that fleet size is a more influential factor than the composition of the fleet itself [12].

Uber Elevate proposes a specific charging scheme to achieve higher operational efficiency given the limited aircraft ranges [11]. That is, while passengers are deboarding/boarding the aircraft, the aircraft is charged for a duration of about 7 minutes before taking off for the next flight, allowing the aircraft to optimize airtime before having to stop for a full-term charge. We heavily consider existing evidence on recent technology developments, particularly by Lilium and ABB, which claim rapid battery charge in their eVTOLs of approximately full charge of batteries in 30 minutes and up to 80% of battery in 15 minutes [1]. eVTOL battery design and behavior is a topic of interest for a lot of literature, which more go in depth to develop complex methodologies to model battery charge and discharge.

Alongside efforts to characterize UAM aircrafts, there is also some literature attempting to characterize UAM vertiports. The vertiport, as a central hub of UAM activity, must support takeoff/landing of aircrafts as well as picking up/dropping off of passengers. Since UAM is a mode of air travel, there are additional procedures that need to take place before and after flights, similarly to airplane travel. To accommodate this functionality, some existing literature separates takeoff/landing clearing slots and stands where passengers will board and de-board these aircrafts [18]. Aircraft behavior should accommodate occupying these different spaces and taxiing between them [18, 19]. A couple studies consider the times that aircrafts might spend in these different phases, for instance turnaround time (the time spent at the gate) from 1-6 minutes [4, 19] and clearing time for takeoff/landing of 45-60 seconds [4, 13].

Studies considered both UAM services with pre-determined routes and with non-deterministic routes [7, 10]. Similarly to existing on-demand services for ground transport, like Uber and Lyft, rideshare options can be available, where multiple passengers share an aircraft [10]. The route choice problem is not one that has been deeply explored for UAM, but Rothfeld et. al consider shortest path using Dijkstra's between all origin-destination pairs [17]. Rebalancing, or distributing fleet to predicted areas of high demand, is also a relatively new area of study for UAM, but Kim et. al propose particle swarm optimization or a genetic algorithm for this purpose [12].

Rothfeld et. al implement UAM services in an agent-based simulation environment called MATSim [17]. The study builds a centralized air network and sets up an infrastructure for vertiports. The UAM aircrafts in this research serve on-demand requests, and follow schedules. While their implementation addresses some important parts of UAM services, a few major details are missing. Though they account for static waiting, boarding, and deboarding times, the state dynamics of aircrafts inside the vertiports are not implemented beyond pick-up, drop-off, flying, and staying tasks. Modeling these at a granular level can potentially introduce new issues to explore, for instance with aircraft power consumption while hovering, which have not been addressed. Although there has been literature exploring rebalancing and charging/power consumption for UAM aircrafts, these features not been implemented within an agent-based simulator. These may be critical characteristics of a UAM service to simulate – predictively rebalancing can improve efficacy and performance of the service; charging can be an important constraint in the aircraft movements throughout the network and congestion at vertiports for charging resources.

Despite the sizable literature on UAM, the lack of a realistic, high-fidelity agent-based simulation software that accounts for the complex interaction of demand and supply prevents both operators and planning and regulations agencies from obtaining a comprehensive understanding of future scenarios. The objective of this thesis is to deliver a realistic design and implementation of the UAM service supply in SimMobility, while incorporating the ongoing technological developments in the field, as well as including important features such as vertiport operations, aircraft charging and power consumption while on the fly or in hovering phase, priority landings, details of aircraft state dynamics inside and in between the vertiports, and rebalancing.

Chapter 3

Methodology

We propose a design and implement a solution to effectively simulate UAM infrastructure and supply, integrated with the existing SimMobility software.

3.1 Background on SimMobility

SimMobility is a multi-scale, integrated, activity- and agent-based mobility simulator developed by the ITS Lab [2, 6, 14]. This C++-based, parallelized software has a hybrid time-based and event-based design, through which we can design and simulate various transportation scenarios in an urban context.

SimMobility has three main modules, which enable it to simulate mobility on a wide range of granularities, as seen in Figure 3-1 taken from [2]. The long-term module simulates high-level land use, economic activity, and agents' life cycles. The mid-term module simulates the agents' daily activity and transportation supply/demand. The short-term module simulates the movement and behavior of agents in the simulation at the finest granularity. The expansion to the codebase to enable UAM simulation will be added in the mid-term module.

There are two main components, pre-day and within-day, for running SimMobility in the mid-term module. In the pre-day, agents follow a Day Activity Schedule (DAS) based on a calibrated demand model. The DAS details the agent's general schedule for the day, including activities it needs to perform at different locations at specific

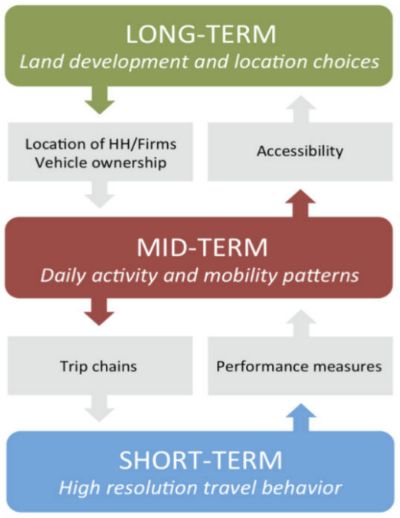


Figure 3-1: Overview of long-term, mid-term, and short-term SimMobility modules, from Adnan et. al.

times of day and preferred modes of transportation between these locations. After the pre-day runs, the agents run within-day models to transform the DAS into decisions and actions. Since actual message communications and movements are happening in the within-day, the movements and activities that the agent performs are subject to change from the initial DAS. For instance, if there is more congestion than expected in the first leg of a trip, an agent may miss the bus they planned to take in their DAS, and may need to wait for the next bus or find a different mode of transportation to complete this leg of the trip, resulting in re-routing. The supply simulator explicitly represents aspects of the supply involved in the simulation. At a high level, the mid-term module establishes consistency in the interaction between the within-day and supply components.

Lastly, there is a day-to-day learning module which enables a feedback loop from the results of the within-day and supply run to update agents' expectations about the transportation system performance indicators such as travel times, which will affect their choices in the next day. For instance, if in the first day, there is too much demand for MoD, when within-day runs, much of the agents' preferred mode of MoD will not be accommodated with the quality of service that they assumed in the pre-day – the day-to-day module will take this knowledge back to the pre-day, such that a portion of the agents will choose other alternatives and thereby the transport demand for this mode decreases.

A mobility service is a service provided by an organization satisfying trip requests from users. Within the context of SimMobility, each mobility service is associated with a controller, which is a piece of software that manages a certain mode of transportation, coordinating the behavior and movement for a fleet of aircrafts. Different mobility services may run in parallel, and users have the ability to use any of these services for their trips. There are already controllers within SimMobility for other modes of transportation, including bus, on-hail taxis, and mobility on demand (MoD). The UAM controller should fulfill similar functions as these existing controllers, managing a fleet of aircrafts and serving passenger requests. It qualifies as a mobility on demand service, but for air travel rather than ground transportation.

3.2 UAM Software Needs

To properly simulate UAM within SimMobility, there are software needs on both the demand-side and the supply-side. There have been expansions on the demand-side, namely in creating a new switching model for the pre-day component and building out the day-to-day learning script to learn parameters specific to UAM. Since the scope of my thesis is in the supply-side, I will focus on the high-level design of what needs to be done to effectively simulate UAM supply, in correspondence to the UAM scenarios and specifications that we've scoped out.

Aligned with how SimMobility works, the UAM implementation must operate as a hybrid time-based and event-based simulation. The overall simulation time period is broken up into shorter frame ticks, and during each tick, every element of the simulation performs their designated action for the duration of the tick. Additionally, components such as the controller and the vertiports and the aircrafts must be able to communicate with each other to trigger different actions to take place.

3.2.1 Network of vertiports

Our simulation will run in a simulated city, with its own population, infrastructure, and modes of transportation. The support for a UAM service will require a network of vertiports, where each vertiport is a multi-functional central station for UAM activity. In a network representation, these vertiports represent the nodes, while all possible flight routes between them are the links. We considered the network to be fully connected, meaning that a UAM aircraft can fly between any two vertiports in a city's UAM network. This simplifies the problem of route choice, since the shortest Euclidean distance between two vertiports will always be the direct route. We chose to make this decision since the air network is assumed to not be largely congested. Yet, this assumption could be relaxed in further studies, since different factors such as bad weather conditions and congestion of the airspace in certain areas could mean an indirect route between two vertiports could be better than the direct one [9].

Each vertiport will need to support a wide range of operations, such as: takeoff, landing, holding waiting passengers, and charging of UAM aircrafts. As opposed to the current ground-based mobility-on-demand services, UAM requires more complicated procedures before take-off and after landing due to maintenance and charging requirements. After landing, the aircrafts taxi to a **stand** where charging starts. Passengers board into the aircraft after charging finishes, and the aircraft starts taxiing to the **FATO** (Final Approach and Take-Off Area) area where they can take off from the vertiport into the air. On the other end of the flight when an aircraft is ready to land, there must be an available FATO to receive the aircraft, and then they taxi to a stand to de-board passengers and charge.

Queue maintenance: Since the number of stands and FATOs per vertiport are limited, the simulation needs to be able to have a way of maintaining queues for these spaces. In general, these queues are first-in-first-out, and serve both aircrafts that on their way to take off or coming in trying to land. As soon as an aircraft leaves the space (finishes charging at the stand, or completes landing, for instance), the first aircraft in the queue can move into the space. Queue maintenance is a key functionality that the simulation must support, since it's one metric to evaluate congestion and the UAM service as a whole. The one exception to this first-in-first-out behavior is in our priority queue feature, which is further discussed in section 3.2.4.

Stand designation: One design enhancement we make is the separation between arrival-designated stands and departure-designated stands. For each vertiport, some stands will prioritize arriving aircrafts while other stands will prioritize departing aircrafts, in order to increase throughput of incoming aircrafts. As an input to each vertiport, there are a number of stands which prioritize arriving aircrafts and a number of stands which prioritize departing aircrafts.

3.2.2 Vertiport-to-vertiport mobility-on-demand service

The UAM service that we're simulating is an on-demand travel mode. People will be able to make requests to the service, which will attempt to match them with an eligible aircraft that can pick them up and drop them off at a desired origin vertiport and destination vertiport. Similarly to on-demand ground transportation services like Uber and Lyft, flights can be pooled, serving multiple passengers going from the same origin to the same destination.

As a mode of travel, UAM must integrate with other modes of transport which may come before (access) or after (egress) a UAM trip. For instance, a passenger might take the bus from where they currently are to reach the origin vertiport. After their UAM trip, from the destination vertiport, they may need to walk a few minutes to reached their actual desired destination. With our design and implementation, we are able to model a variety of access/egress modes along with the UAM travel mode, including public transportation, private car, walking, and Mobility on Demand (MoD).

All of these different legs of an individual's overall journey will be represented as moving along a network made up of nodes and edges, as depicted in Figure 3-2. The squares represent vertiports, and the circles are potential origin and destination nodes. When an agent travels from an origin to a destination, they take an access leg which

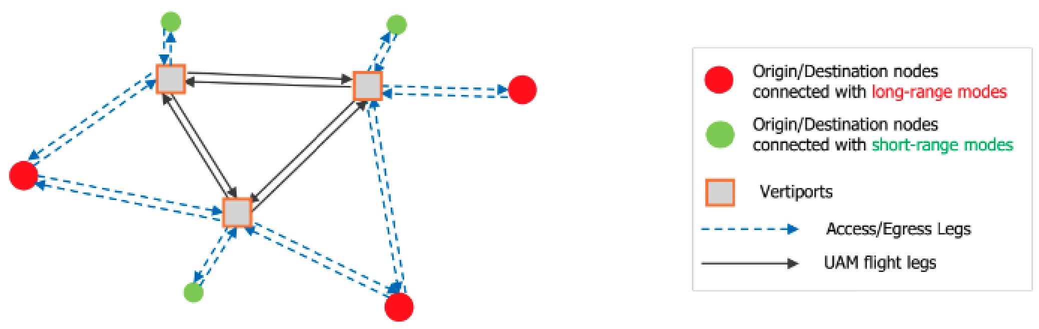

Origin/Destination nodes connected with long-range modes
Origin/Destination nodes connected with short-range modes
Vertiports
Access/Egress Legs
UAM flight legs

Figure 3-2: Network representation of the UAM air network.

is dotted to a vertiport, where they then take a UAM flight. From that vertiport, they travel along the egress leg to their final destination. The within-day component deals with the problem of route choice from origin to destination, including access and egress. This is not a simple deterministic model that always chooses the closest vertiport. Although for the UAM subset of the network, we always choose the direct route, we use a behavioral probabilistic choice model to determine an agent's route from origin to destination, including choosing the vertiports used for the UAM trip.

3.2.3 UAM aircrafts

UAM aircrafts also have complex operations that must be simulated, mainly dealing with moving through the network of vertiports, charging, and handling UAM passengers. The typical flow of a UAM aircraft serving a trip is shown in Figure 3-3, in parallel to the flow of actions that a passenger on that aircraft would experience. An aircraft starts idle at a vertiport, and when it has a trip to serve, moves to a vertiport stand to charge for the upcoming trip and board any passengers. Once that process is complete, the aircraft starts taxiing to FATOs and waits nearby if no FATOs are available at the time. Upon availability of an FATO, the aircraft uses it to take off, and flies to its destination vertiport. At the end of its flight, the aircraft hovers until an FATO becomes available for landing (see section 3.2.4 for information on priority queuing). After landing, the aircraft taxis to a stand, where it charges and de-boards its passengers. In order to progress through these states and serve passengers, air-

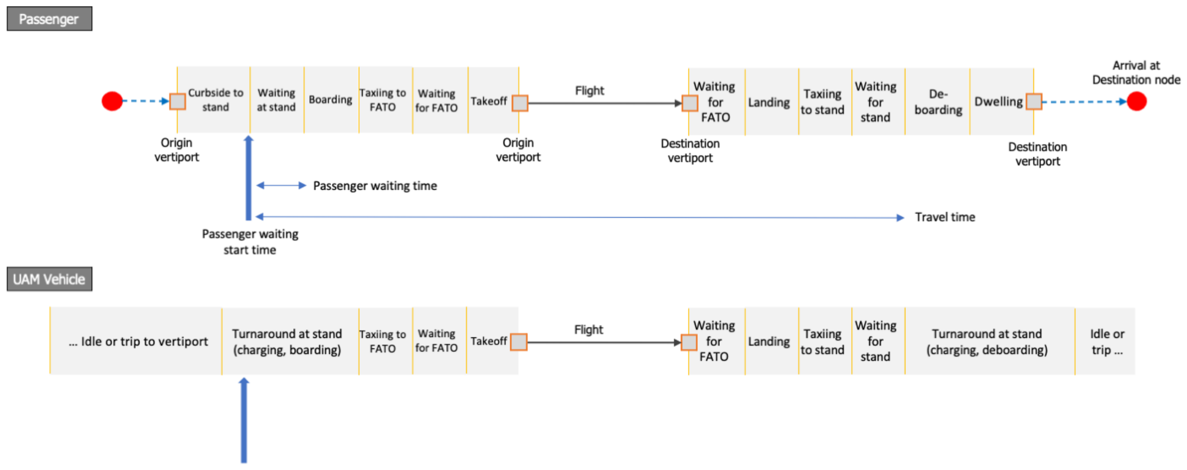


Figure 3-3: Overview of state flow for UAM aircrafts and passengers.

crafts should keep track of their current schedule (what trips are on their schedule to serve) and any passengers they're currently holding.

The passenger goes through the same flow of states while inside the aircraft, but may start waiting at a stand before the aircraft arrives. The time at which the passenger starts waiting is called the **waiting start time**, and the **passenger waiting time** is the time spent from then until when boarding starts. The **travel time** also starts at this waiting start time, and ends when the passenger deboards. The passenger boards and deboards during the corresponding turnaround time for the aircraft.

3.2.4 UAM controller

With the vertiport network infrastructure and fleet of aircrafts in place, the last component needed is a UAM controller.

Keeping track of aircraft charge and state: The controller should have full visibility into all aircrafts in the fleet, being able to see any given aircraft's current state and charge. This is crucial to keep track of aircrafts that are in danger of running out of charge while holding passengers, and for the logic of the matching and rebalancing algorithms.

One design enhancement to the vertiport queuing is **priority queue maintenance** by the controller. As detailed in section 3.2.1, vertiport spaces are limited and there must be queuing practices in place for the stands and FATOs. However, when aircrafts are waiting for an FATO for landing, they are still depleting charge while hovering. To combat this, the controller needs to keep track of vertiport queues and re-order them to prioritize aircrafts that may run out of charge. Whenever an aircraft dips below a minimum threshold of charge or has been hovering for a certain amount of time (**maximum hovering time**), it should be moved to the front of the queue to land as soon as possible.

Matching requests: The core logic of the matching algorithm is as follows: pick the aircraft that can take the least amount of time to serve a given passenger trip request. However, there are a variety of conditions that an aircraft must fulfill in order to be considered eligible for matching, as well as exceptions. In order for an aircraft to be eligible for matching, the following conditions and parameters are considered:

- Trip length
- Distance from the request's origin vertiport
- Amount of aircraft charge
- Capacity of the aircraft
- Whether aircraft currently has a trip assigned

Whenever the controller matches a trip request, the aircraft is assigned the trip in its schedule items and takes action to embark on and complete the trip. Any trips unable to be matched with an eligible aircraft for any reason, will remain in the queue of trip requests and the controller will attempt to match the passenger request in the next frame tick.

Pooling trips: Since our on-demand UAM service will support pooling trips, we also accommodate this in our matching algorithm. For all incoming trip requests, we should not only consider empty aircrafts, but also aircrafts that meet the following conditions:

- At least one seat available
- Assigned other passengers within a specified time period (**pool match buffer time**)

The matching algorithm should compute a time predicted to serve the trip request for these aircrafts as well. However, pooling should be given a slight preference over solo trips. The controller needs to keep track of the aircraft that can serve the fastest solo trip and aircraft that can serve the fastest pooled trip. If the best pooled aircraft can serve the trip faster or only **pool priority time** slower than the best solo aircraft, then the pooled aircraft should be chosen.

Request offset: The request offset is a learned parameter per vertiport that represents passengers learning how much time in advance they need to make a request.

Expected hover time: The expected hover time is another learned parameter per vertiport, representing the amount of time a UAM aircraft is expected to wait at each vertiport for an FATO for landing. This time is used in the matching algorithm when calculating the required aircraft charging time.

Rebalancing fleet: The controller must also support rebalancing. If aircrafts simply remain where they are after serving a trip, this could lead to an imbalance of aircrafts in the network and congestion. In order to learn from the past trip requests and in an attempt to better predict where demand might come from in upcoming requests, the controller rebalances its fleet. The controller will send aircrafts to with no current schedule items to "cruise" to the vertiports where the most recent requests originated, in expectation that by the time they get there, there will be some passenger requests that can conveniently use these aircrafts to get to their destination faster.

Rebalancing utilizes aircraft resources, and in order to ensure that enough aircrafts are available for matching, we designed a **cruise buffer time** parameter. When an aircraft is assigned to cruise to a certain vertiport, it waits for a certain amount of time before proceeding to fly – this gives the controller the opportunity to assign the aircraft a passenger before leaving.

In summary, the UAM supply simulation must be a time-based simulation that comprises a network of vertiports, fleet of aircrafts, and a controller that will control the matching of trip requests and rebalancing of the UAM fleet.

3.3 Implementation

3.3.1 High-level architecture

The bulk of the UAM supply implementation takes place in a module external to the main SimMobility software. In this section, we outline the high-level flow of the implementation, as detailed in Figure 3-4.

The process begins by running the pre-day, within-day, and supply of SimMobility. Taking the output of the within-day module, we extract trip-level information for UAM trip legs, including the origin/destination vertiports and the time at which the agent arrives to the origin vertiport. We pre-process this data by offsetting the origin vertiport arrival time by a fixed dwelling period to get the **waiting start time** for the UAM trip leg. For each trip leg, we store the metadata of person ID, origin, destination, and waiting start time to pass along to the main simulation. The UAM controller implemented in Python is analogous to the supply module of SimMobility – the simulation creates a fleet, network of vertiports, and queue of trip requests for a controller to manage, representing the supply. The simulation reads some additional learned inputs from the database when initializing the vertiports: the **request offset** and **expected hovering time** for each vertiport. These values are learned by the supply module via the within-day loop in the external module – based on outputs from the UAM controller simulation, the within-day component of the module updates these values.

The output of the supply simulation is the **total waiting time** and **total travel**

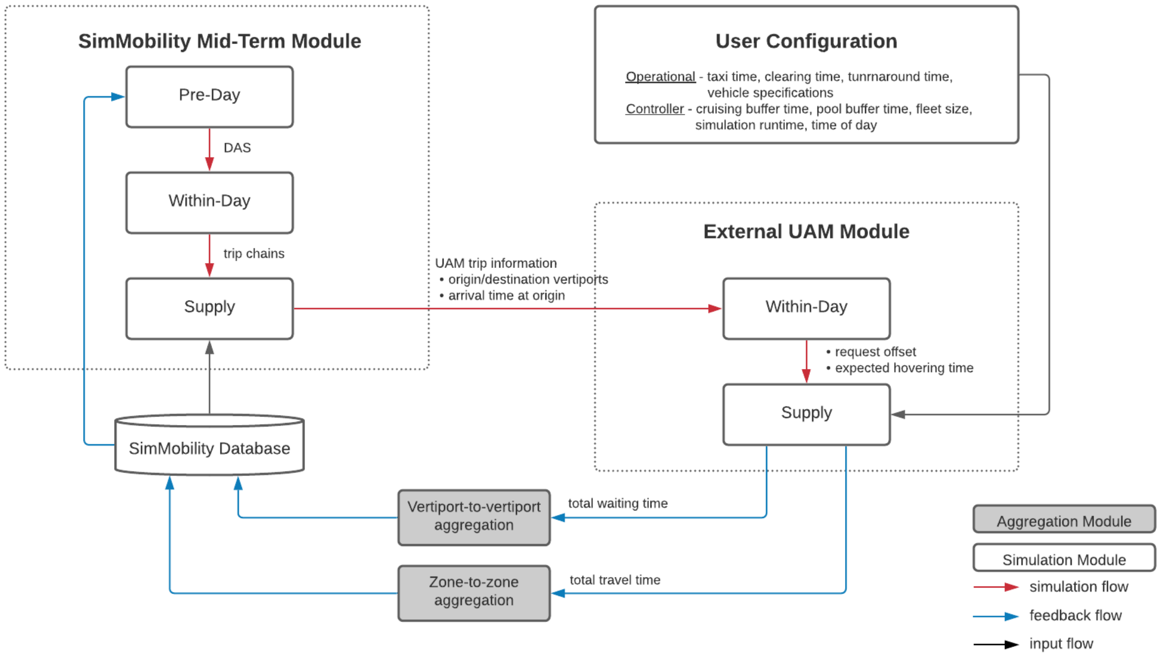


Figure 3-4: High-level architecture of UAM implementation as an external module & interactions with SimMobility.

time per UAM trip. The total travel time is aggregated by vertiport-to-vertiport pair, while the total waiting time is aggregated by zone-to-zone pair. As part of the day-to-day learning process, based on these outputs, the corresponding tables in the database are updated. These are then read by the supply component within the SimMobility mid-term module, by a simple teleporting controller. Whenever an individual is scheduled to take a UAM trip, we utilize the existing "Walk" mode in SimMobility and have them travel for the predicted travel time according to the vertiport origin-destination pair they're traveling in between.

This high-level architecture is different from the typical controller design and implementation within SimMobility. This alternative would have been to implement the UAM supply fully integrated into the mid-term module of SimMobility, building out the vertiport network components, UAM aircrafts, and fit into the existing infrastructure for controllers and movement within the simulation. Though this would have been more integrated with SimMobility at a fundamental level, there were some key reasons why we chose this flow in designing UAM supply to work with SimMobility.

Since UAM is not an existing service and has not actually been simulated in very many software settings, we wanted to be able to test out a variety of scenarios and controller designs in rapid iterations. By externalizing the UAM controller, this has enabled us to build a highly customizable piece of software with complex service operations and algorithms. While SimMobility is a massive piece of software, the external module is lightweight. Implementing UAM within SimMobility would require significant additional code to build out an air network infrastructure and accommodate agent movement within this network, then to pass it between the air and road networks. With the external module coded in Python, this representation can be much simpler without unnecessary complications. Additionally, we were able to add any parameters and outputs we needed, which is considerably more difficult in SimMobility.

Of course, there are trade-offs in deviating from the typical controller implementation. With the external module, we were unable to leverage built-in functionalities of messaging and the ground transportation infrastructure. There is also some loss of information since we are aggregating by vertiport-to-vertiport pair and time of day instead of the trip level. Despite not being able to use baked-in elements of SimMobility, we considered these to be elements that we were willing to give up in order to get more complexity for the UAM situation in particular.

3.3.2 Python module

In this section, we will discuss further the implementation of the supply component of the external UAM module.

Trip class

The first main class is the `Trip` class, which represents an individual UAM flight. Each Trip has a unique `id` attribute, and has metadata of `origin_id`, `destination_id`, and `waiting_start_time` passed in from the SimMobility DAS. The `waiting_start_time` corresponds to the time at which the person associated with the `Trip` starts their waiting period at the origin vertiport.

A `Trip` can represent a UAM flight in a aircraft taken by multiple passengers going from the same origin to the same destination. If two `Trips` get assigned to the same aircraft, only one `Trip` will be kept. The `person_ids` list is an array that holds the person associated with the `Trip`.

After SimMobility runs, the DAS passes all of the information about UAM trip legs to the controller module, initializing one `Trip` object per trip leg. All `Trips` are put into a queue which is sorted in increasing order of the trips' `waiting_start_time`. The UAM controller uses and maintains this queue in order to serve all passenger requests in time order. Whenever a request is matched to be served by an aircraft, this `Trip` object is directly used as a schedule item for the aircraft to serve. The usage of `Trips` by the controller and aircrafts are further discussed in the following sections.

Vertiport class

The UAM air network is composed of `Vertiport` objects which correspond to individual veritports. Each `Vertiport` is initialized with:

- a unique `id` attribute
- `x` and `y` coordinates corresponding to its location
- `num_fatos` and `num_stands_arriving` and `num_stands_departing` corresponding to the number of FATOs and stands it holds
- `request_offset`: an operational parameter representing the time after a trip's `waiting_start_time` that the controller should attempt to match it to an available aircraft.
- `expected_hovering`: an operational parameter representing the expected hover time

There are multiple data structures around the FATO and stand representations to effectively simulate occupying and waiting for these spaces. The `fatos`, `stands_arriving`, and `stands_departing` are data structures which hold the aircrafts currently occupying the FATOs, stands which prioritize arriving aircrafts, and stands which prioritize departing aircrafts respectively. However, the number of these spaces are limited, so the `waiting_for_fatos`, `waiting_for_stands_arriving`, and `waiting_for_stands_departing` queues hold the aircrafts currently waiting for available FATOs and stands. These are populated first-in-first-out by aircrafts trying to enter these spaces. The order of the `waiting_for_fatos queue` is modified with the priority queue enhancement, as described in section 3.2.4. Each data structure has an associated add and remove method to maintain their elements, which the `Vehicles` use to enter and queue for these spaces.

To initialize the `Vertiport` objects with correct metadata, the UAM module reads from the SimMobility database and loads the `Vertiports` into a map.

Vehicle class

The `Vehicle` class represents aircrafts within the service fleet. Each `Vehicle` object is initialized with:

- a unique `Vehicle id` attribute
- `capacity`: the number of passengers the aircraft can hold at once
- `move_speed`: the speed at which the aircraft can move
- `hover_speed`: the rate at which the aircraft depletes charge while hovering

The following key variables maintain the charging, occupancy, and behavioral state of the aircraft:

- `passengers`: a dictionary that holds the people currently aboard the aircraft
- `schedule_items`: a queue that holds the `Trip` objects assigned the aircraft, which dictate the vertiport that the aircraft will travel to next
- `dist_can_travel`: the distance that the aircraft can travel with the amount of time it has been charged for

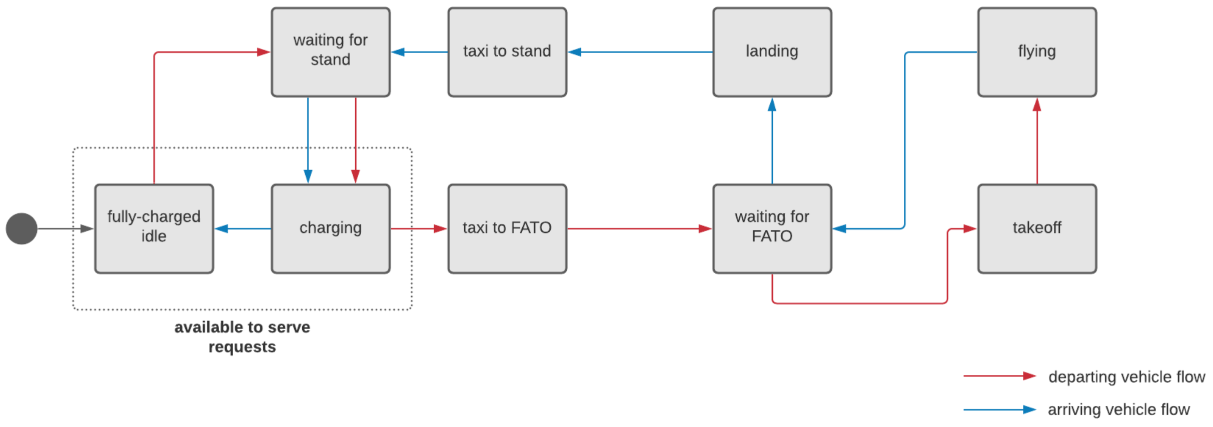


Figure 3-5: UAM aircraft state transitions.

- `state, prev_state`: the current state, the activity the aircraft is performing and the directly previous state. This is discussed in further detail in Figure 3-5.
- `state_duration_time`: the time that the aircraft needs to spend in its current state before attempting to move to the next one
- `hover_time`: the time that the aircraft has been hovering. Increments when an aircraft starts hovering and gets reset to zero when the aircraft is able to land.

At the beginning of simulation, all `Vehicles` start in the `FULLY_CHARGED_IDLE` state distributed evenly across the vertiports, with a full charge. First, we can step through the state flow when an aircraft is departing a vertiport. When a `Vehicle` receives a directive from the controller to serve a request, it transitions to `WAITING_FOR_STAND`, where it enters the waiting queue for a stand, or the vertiport's `waiting_for_stands_departing` queue. If there's an open stand, the aircraft can immediately transition to the `CHARGING` state and enter the vertiport's `stands` data structure. When all passengers are boarded and the aircraft is ready to take off, it transitions to `TAXI_TO_FATO`, then at the end of taxiing enters the `WAITING_FOR_FATO` state and gets put in the vertiport's `waiting_for_fatos` queue. Similarly to the `WAITING_FOR_STAND` state, as soon as there's space in the FATOs for the aircraft, the aircraft gets put in the vertiport's `fatos` and transitions to `TAKEOFF`, then transitions to the `FLYING` state when it clears the takeoff area. Next, we can step through the state flow when a aircraft is arriving to its destination vertiport. When the aircraft has reached the destination vertiport, it immediately transitions from the `FLYING` state to the `WAITING_FOR_FATO` state and gets put in the destination vertiport's `waiting_for_fatos` queue, since the FATOs accommodate both aircrafts that are taking off and landing. Once the aircraft reaches the front of the queue, it enters the LANDING state, then clears the FATO area and proceeds to `TAXI_TO_STAND`. Once it has finished taxiing, it enters the `WAITING_FOR_STAND` state, then when it reaches the front of this `waiting_for_stands_arriving` queue, transitions to the `CHARGING` state and gets put in the `stands` data structure. If the aircraft otherwise does not have any schedules to serve, it charges to 100% battery capacity, then transitions back to the `FULLY_CHARGED_IDLE` state. When in either the `CHARGING` or `FULLY_CHARGED_IDLE` state, the aircraft is considered to be eligible by the controller to be matched to a trip request.

The `state_duration_time` attribute is key to the state flow shown above. The times spent in some of the states are parameterized and set before the simulation is run (namely takeoff/landing times and taxi times), while others are computed during the simulation (charging times), but waiting times spent in queues cannot be precomputed.

All of the state changes and aircraft behavior is held in the `frame_tick` function of this class, which is essentially a state machine. This external supply module is run as a time-based simulation, which progresses in frame tick increments of 5 seconds over a total simulation runtime. Every tick, every aircraft's `frame_tick` function is called, and based on its current `state` and `state_duration_time` , the aircraft will complete some action and update its internal state. If the `state_duration_time` has not expired for the current `state`, for example, if the aircraft needs to charge for a while longer, then the `frame_tick` function will keep the aircraft in its current state and decrement pause time by the tick duration. However, if the `state_duration_time` reaches zero or some other event happens, the aircraft may change state.

Whenever an `Vehicle` transitions its state, `change_state` is called. This is a helper function which takes in the previous state and new state to correctly set the `state_duration_time`, remove or add the aircraft from any `Vertiport` data structures, and adjust the aircraft's `schedule` and `passengers` as needed.

There are also `move` and `charge` functions which are called within `frame_tick` whenever the aircraft is `FLYING` or `CHARGING`. They modify the aircraft's `dist_can_travel` attribute which corresponds to charge, and the `move` function also increments `hover_time` for use with the priority queue feature described in section 3.2.4.

UAM Controller logic

The UAM controller logic has full visibility into the network of `Vertiport` objects, fleet of `Vehicle` objects, and `Trip` request queue, which it leverages to perform matching and rebalancing procedures.

Every frame tick, the controller accesses each `Vertiport` object's `waiting_for_fato` queues and checks the queuing aircrafts' `dist_can_travel` (representing its charge) and `hover_time`, implementing the priority queue procedure described in section 3.2.4.

The matching algorithm is implemented in `match_trips`, which is called every frame tick of the simulation. Each time `match_trips` is called, the controller goes through the request queue of `Trips`. If the `Trip`'s `waiting_start_time` plus the associated `request_offset` is greater than the current simulation time, then the controller attempts to match it with an available `Vehicle`. The categories of available `Vehicles` are outlined below:

1. Empty aircrafts, can serve a solo trip
    - `FULLY_CHARGED_IDLE` or `CHARGING` state
    - Empty `schedule_items`, empty `passengers` (no current schedule or passengers to serve)
2. Partially empty aircrafts, can serve a pooled trip
    - `FULLY_CHARGED_IDLE` or `CHARGING` state

    - Length of `Vehicle passengers` list is less than `capacity` of the aircraft
    - Current schedule item's `assigned_time` is less than `pool_match_buffer_time` offset from the current simulation time

3. Rebalanced aircrafts, can serve a solo trip
    - `FULLY_CHARGED_IDLE` state
    - `Vehicle`'s `state_duration_time` is > 0, meaning the aircraft has not started cruising to its rebalanced location yet
    - Current schedule item is designated as a cruising rebalancing trip

For each `Vehicle`, the controller calculates how much time it would take to serve the request, by computing and summing the following:

- The time for the `Vehicle` to travel from its current `Vertiport` to the origin `Vertiport` (calculated by aircraft's `move_speed` and distance between vertiports)
- The time for the `Vehicle` to travel from the origin `Vertiport` to the destination `Vertiport` (calculated by aircraft's `move_speed` and distance between vertiports)
- The time for the `Vehicle` to charge for these flights (calculated by charging profile)
- The `expected_hover_time` for the destination `Vertiport` (set as parameter)

Throughout the function, for each individual `Trip` request, the best pooled aircraft and best solo aircraft, along with their associated times, are kept track of. Once all aircrafts are checked, the controller chooses the best pooled aircraft, if it has a lower time to serve the trip, or if it is slower by less than the `pool_priority_time`. Otherwise, the trip request is matched with the best solo aircraft. Once the best aircraft is picked for the given trip request, the controller calls `assign_trip` on the chosen `Vehicle` to alter its `schedule_items`, which should set it in to action the next time its `frame_tick` function is called. The controller iterates through all the `Trips` that need to be matched, removing from the request queue ones that have been matched, and leaving those that weren't able to be matched. This ensures that next frame tick, the `Trips` that have been request first will be attempted to be matched again over newer requests, since they come earliest in the queue.

Rebalancing is handled via the `rebalance` function. The controller maintains `latest_vertiports`, a dictionary of `Vertiport` objects mapped to how many matched `Trips` originated from those vertiports. There is no entry for a `Vertiport` in this dictionary if the number of matched requests is zero. Whenever a `Trip` request is matched in `match_trips` and assigned to a `Vehicle`, the entry in `latest_vertiports` corresponding to the origin vertiport is incremented. In `rebalance`, while there are entries in `latest_vertiports`, each available aircraft (qualified by being in the `FULLY_CHARGED_IDLE` state and having no schedule items) is assigned a vertiport to cruise to from `latest_vertiports`. The controller calls `assign_trip` on the chosen aircraft to add the cruising item to its schedule, but also adds the `cruise_buffer_time` as the aircraft's `state_duration_time` before the aircraft actually moves, per the enhancement described in section 3.2.4. Then, the chosen vertiport from `latest_vertiports` has its value decremented, or the entry is removed from the dictionary if the value is zero. When the length of `latest_vertiports` is zero or there are no available aircrafts to cruise, the controller ends its rebalancing for the frame tick.

Chapter 4

Results

4.1 Running the simulation

In order to run the simulation, there are a few additional procedures to handle inputs and outputs. The direct input from the SimMobility module to the external UAM module is a CSV file of the UAM trip legs from the DAS. This is pre-processed to get the trip metadata as described in the initialization process for `Trips` in section 3.3.2. There are several user-configured parameters and learned operational parameters from the SimMobility database as well. During the run, the controller writes to a log every time a UAM aircraft is assigned a trip or changes state. This log is post-processed to extract the time spent in each aircraft state per UAM trip, which allows us to glean key components of passenger waiting time and travel time. We can use these values to update the request offset and expected waiting time in the within-day loop as detailed in section 3.3.1 on high-level architecture. The passenger waiting time and travel time outputs are aggregated and fed to the within-day script for next day's run of SimMobility.

4.2 Inputs & Configuration

For our experiment, we assume an AI (auto-innovative) protoype city. The demand model is calibrated such that there are 18 million total trips in the city, with an air

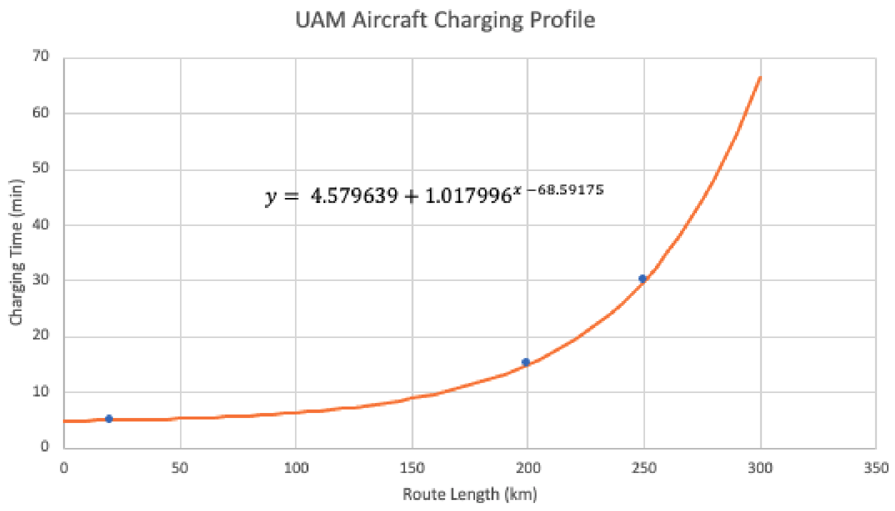


Figure 4-1: UAM aircraft charging profile.

network of 23 vertiports. The vertiport layouts were selected based on [18], which guide how the number of stands and number of FATOs per vertiport were chosen. The number of stands per vertiport ranges from 8 to 56 while the number of FATOs per vertiport ranges from 2 to 14. The UAM ticket price was set at \$4.52/km based on [9].

For our scenarios, we assume a homogeneous fleet of UAM aircrafts, meaning all aircrafts have the same charging profiles, flight speeds, and passenger capacities. These specifications are detailed in Table 4-1, based on what is within the range of existing literature: a flight speed of 200 km/h [3, 4, 7, 9], a maximum trip distance of 250 km [3, 7, 9], and 3 seats in the aircraft [7, 9, 10]. The aircrafts for our UAM service will be piloted, in context of user preference for piloted air travel in demand-side literature [7, 9]. Based on the Lilium/ABB eVTOLs [1] and the assumption of a maximum trip range of 250 km, we designed the charging profile in Figure 4-1, which is dependent on the trip length that the aircraft is expected to fly for an upcoming trip.

We set the take-off and landing clearing times to 60 seconds and the taxi time to 0 seconds, in alignment with existing literature [4, 13, 19]. The parameters used by the UAM controller are set as follows: pool match buffer time is set to 2.5 minutes;

Table 4.1: UAM aircraft specifications.

UAM Aircraft Specifications	
Flight speed	200 km/h
Max trip distance	250 km
Number of seats	3 seats

pool priority time is set to 3 minutes; maximum hover time is set to 12 seconds.

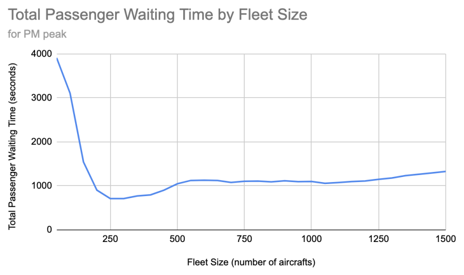
After sufficient iterations of day-to-day learning, the equilibrium demand for this setup was 33.7K UAM trips resulting in a penetration rate of 0.187%. The results in the following section are based off of this demand and these inputs.

4.3 Testing the UAM module

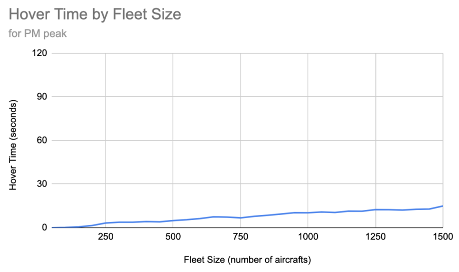
We test our UAM module by adjusting independent variables among our parameters, then observing how these changes affect the results. In the case of our UAM module, we found that different components of passenger waiting time and travel time are good indicators for the quality of service, and therefore, overall UAM service performance.

As existing literature indicated, fleet size is a potentially critical parameter for UAM scenarios [12], we focus on the effect that fleet size has on different components of passenger waiting time – in particular the total passenger waiting time and hover time. The total passenger waiting time is defined as the time it takes from when the passenger starts waiting at the stand for the aircraft until when they reach the destination vertiport stand. The hover time is defined as the time the passenger's aircraft spends hovering, waiting to land at the destination FATO.

We run the UAM scenario for the PM peak period (4PM-5PM) on a range of fleet sizes, from 50 to 1500 aircrafts, keeping all other parameters constant as described in the previous section. We then observe the total passenger waiting time and hover time. From Figure 4-2(a), we can see that there is an optimal fleet size at 250-300. Before this point, the waiting time is extremely high, and after this optimal size, the total waiting time increases. When the fleet size is too low, there are not enough vehicles to

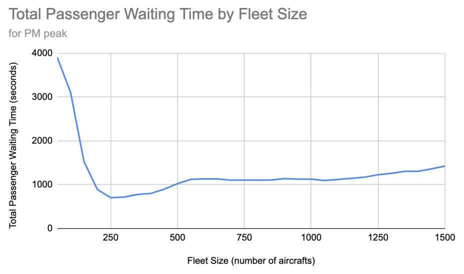


(a) Total Passenger Waiting Time by Fleet Size.

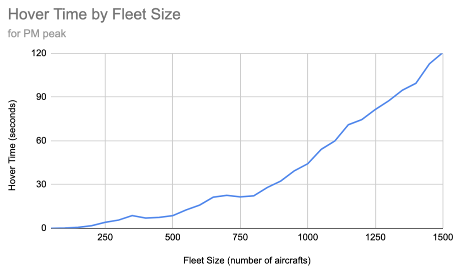


(b) Hover Time by Fleet Size.

Figure 4-2: Components of waiting time over varying fleet sizes.



(a) Total Passenger Waiting Time by Fleet Size.



(b) Hover Time by Fleet Size.

Figure 4-3: Omitting the priority landing queue enhancement, components of waiting time over varying fleet sizes.

accommodate all passenger requests, causing longer wait times. Meanwhile, when the fleet size is too high, there is more congestion in the air network resulting in longer waiting times. In Figure 4-2(b), we see that as the fleet size increases, the hover time increases as well. When the fleet gets larger, the network becomes more congested, making waiting queues for FATOs longer.

We also ran the UAM scenario on the same period with varying fleet sizes without the priority queue enhancement. From Figure 4-3(a), we observe a similar overall curve to the total passenger waiting time as it varies with fleet size. However, whereas the hover time with the priority landing remained under 30 seconds for 1500 aircrafts, without the priority landing the hover time grew way past this threshold to 120 seconds at 1500 aircrafts as seen in Figure 4-3(b). These results demonstrate the importance of this added feature to limit how much time aircrafts spend hovering and how much power they need to consume while waiting.

Chapter 5

Conclusion

With our novel design and implementation for the UAM supply module, we are able to effectively simulate supply while gaining insights on key metrics for UAM services. Simulating UAM supply is an underdeveloped area of study, let alone within a state-of-the-art, multi-scale, agent-based software like SimMobility. Our implementation and design of UAM takes into consideration complex operational details of potential UAM services such as queuing at vertiports, charging, and enhancements for rebalancing and optimizing matching requests. Particularly with the detailed vertiport operations, we were able to separate out components of passenger waiting and travel times, and analyze these granular waiting times to improve our UAM service design.

Additionally, with the implementation of this design as an external module that integrated with SimMobility, we created a replicable architecture for future mobility controllers to emulate. For mobility services like UAM with uncertainty and a need for multiple iterations of development and design, this external module flow offers a high degree of flexibility. For UAM, this architecture allowed for a development process which resulted in novel service features, including the priority queuing, cruise and pool match buffer times, request offset, and stand designation. Because the module is not directly embedded into SimMobility, this module is lightweight and customizable. Ultimately, the supply-side implementation architecture offers an alternative to how controllers were implemented in SimMobility, that prioritizes flexibility in design, exploration of new design possibilities, and rapid development.

There are many streams of work moving forward that could be followed. One is implementing the controller internally in C++. Since the UAM vertiport, aircraft, and service operations have been designed, we could implement this within the Sim-Mobility codebase to adhere to the typical pattern of implementation other controllers follow. Another is utilizing a heterogeneous fleet instead of homogeneous with the current Python UAM controller. Having different types of aircrafts that serve short and long trip ranges could lead to interesting patterns and differences in the UAM service performance.

Bibliography

[1] ABB and Lilium team up to revolutionize charging infrastructure for regional air travel. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/, Oct 2021.

[2] Muhammad Adnan, Francisco C Pereira, Carlos Miguel Lima Azevedo, Kakali Basak, Milan Lovric, Sebastián Raveau, Yi Zhu, Joseph Ferreira, Christopher Zegras, and Moshe Ben-Akiva. Simmobility: A multi-scale integrated agent-based simulation platform. In *95th Annual Meeting of the Transportation Research Board Forthcoming in Transportation Research Record*, 2016.

[3] Christelle Al Haddad, Emmanouil Chaniotakis, Anna Straubinger, Kay Plötner, and Constantinos Antoniou. Factors affecting the adoption and use of urban air mobility. *Transportation research part A: policy and practice*, 132:696–712, 2020.

[4] Haleh Ale-Ahmad, Hani Mahmassani, and Michael Hyland. Simulation framework for on-demand urban air mobility (uam). Technical report, 2020.

[5] Bilge Atasoy, Takuro Ikeda, Xiang Song, and Moshe E Ben-Akiva. The concept and impact analysis of a flexible mobility on demand system. *Transportation Research Part C: Emerging Technologies*, 56:373–392, 2015.

[6] Carlos Lima Azevedo, Katarzyna Marczuk, Sebastián Raveau, Harold Soh, Muhammad Adnan, Kakali Basak, Harish Loganathan, Neeraj Deshmunkh, Der-Horng Lee, Emilio Frazzoli, et al. Microsimulation of demand and supply of autonomous mobility on demand. *Transportation Research Record*, 2564(1):21–30, 2016.

[7] Robert Binder, Laurie A Garrow, Brian German, Patricia Mokhtarian, Matthew Daskilewicz, and Thomas H Douthat. If you fly it, will commuters come? a survey to model demand for eVTOL urban air trips. In *2018 Aviation Technology, Integration, and Operations Conference*, page 2882, 2018.

[8] Mengying Fu, Raoul Rothfeld, and Constantinos Antoniou. Exploring preferences for transportation modes in an urban air mobility environment: Munich case study. *Transportation Research Record*, 2673(10):427–442, 2019.

[9] Rohit Goyal, Colleen Reiche, Chris Fernando, JD Jacquie Serrao, Shawn Kimmel, Adam Cohen, and Susan Shaheen. Urban Air Mobility (UAM) Market Study-Booz Allen Hamilton Technical Briefing. Technical report, 2018.

[10] Shahab Hasan. Urban air mobility (UAM) market study. 2018.

[11] Jeff Holden and Nikhil Goel. Fast-forwarding to a future of on-demand urban air transportation. *San Francisco, CA*, 2016.

[12] Sang Hyun Kim. Receding horizon scheduling of on-demand urban air mobility with heterogeneous fleet. *IEEE Transactions on Aerospace and Electronic Systems*, 56(4):2751–2761, 2019.

[13] Lee W Kohlman, Michael D Patterson, and Brooke E Raabe. Urban air mobility network and vehicle type-modeling and assessment. 2019.

[14] Yang Lu, Muhammad Adnan, Kakali Basak, Francisco Câmara Pereira, Carlos Carrion, Vahid Hamishagi Saber, Harish Loganathan, and Moshe E Ben-Akiva. Simmobility mid-term simulator: A state of the art integrated agent based demand and supply model. In *94th Annual Meeting of the Transportation Research Board, Washington, DC*, 2015.

[15] Jimi B Oke, Youssef M Aboutaleb, Arun Akkinepally, Carlos Lima Azevedo, Yafei Han, P Christopher Zegras, Joseph Ferreira, and Moshe E Ben-Akiva. A novel global urban typology framework for sustainable mobility futures. *Environmental Research Letters*, 14(9):095006, 2019.

[16] Jimi B Oke, Arun Prakash Akkinepally, Siyu Chen, Yifei Xie, Youssef M Aboutaleb, Carlos Lima Azevedo, P Christopher Zegras, Joseph Ferreira, and Moshe Ben-Akiva. Evaluating the systemic effects of automated mobility-on-demand services via large-scale agent-based simulation of auto-dependent prototype cities. *Transportation Research Part A: Policy and Practice*, 140:98–126, 2020.

[17] Raoul Rothfeld, Milos Balac, Kay O Ploetner, and Constantinos Antoniou. Agent-based simulation of urban air mobility. In *2018 Modeling and Simulation Technologies Conference*, page 3891, 2018.

[18] Parker D Vascik. *Systems analysis of urban air mobility operational scaling*. PhD thesis, Massachusetts Institute of Technology, 2020.

[19] Parker D Vascik and R John Hansman. Scaling constraints for urban air mobility operations: air traffic control, ground infrastructure, and noise. In *2018 Aviation Technology, Integration, and Operations Conference*, page 3849, 2018.

[20] Pavan Yedavalli and Jessie Mooberry. An assessment of public perception of Urban Air Mobility (UAM). *Airbus UTM: Defining Future Skies*, pages 2046738072–1580045281, 2019.