CS237 Final Report (Group 3): IoT Data Generator

1 Introduction

Simulations have been widely used in the research community to aid in generating data when obtaining real-world data is difficult. In general, such simulations can be categorized into one of two types: event-based or time-based. In event-based simulations, the progression of the simulation is driven by the passing of events which have timestamps; it is important in such simulations to obey properties such as causality. In contrast, time-based simulations are driven by the passing of time, in which events are able to occur. In both cases however, spaces and events are heavily reliant upon each other for the simulation to run. In this skeleton for a framework, another entity is often used as the point of interest: agents. When considering that all three entities work almost in lockstep with each other, the scale of such simulations can grow incredibly large.

The main focus of our project is to take one such simulator: an IoT Data Trajectory Simulator built by us, and explore the manner in which we can distribute the workload of the simulator among a number of nodes. A few questions arise: (i) What is the most efficient method of distributing spaces to nodes?; and (ii) How can we measure such variability? What methods will be used to determine the quality of a distribution? In our project, we ignore the validation of the output, as this project is meant to explore the scalability of such a simulator. In the next section, we elaborate on how we will approach scalability of the simulator.

2 IoT Data Generator Overview

In this section, we will describe the simulator that we wish to make distributed: an IoT Data Simulator. We first describe the entities that play a role in our simulator: agents, events, and spaces. For each of these entities we will describe the data structures they need to store, and the messages they need to pass to one another. We will use a university case study to describe examples of the above.

The IoT Data Simulator is a event-based simulator that relies on three main entities: agents, events and spaces. At a high level, an agent will attend events, which are hosted in spaces. To describe each of these entities, we will use a typical university use case scenario.

Agents are the people of the simulator who attend different events throughout the day, moving from space to space. In our simulator, the events that an agent goes to depends on what affinity they have towards them. For example, if there is an agent whose classification is professor, then they would be more likely to attend seminar events and research meeting events, rather than study events. Agents themselves do not have tendencies to use one space over another; instead, this is handled through associating events and spaces together (discussed next).

Events, conceptually, are a list of things that the agents of the simulator can do; they are hosted in spaces. In our simulator, events are related to closely coupled with agents and spaces. Each event must store the capacity of each different type of agent to attend; this is used, for example, in lecture events where only one professor should attend, but many students should attend. We will refer to this as the capacity of an event. With respect to spaces, an event needs to store all the different spaces in which it could be hosted. We note that for most events in our university scenario, there will only be one space per event, but some examples of events that are not contained in only one space include discussions (multiple can be at the same time), or seminars that can overfill and be broadcasted to different rooms.

Spaces are a list of spaces in which events are to take place, as well as for agents to move around. The main concern of spaces include storing occupancy over time - we do not want spaces to be overoccupied. Internally, this is stored as an interval tree, where an interval denotes the occupancy of a person in a space. As the simulation runs on, such an interval tree is bound to grow, and it is for this reason that we wish to explore the effect of distributing the simulator. We conceptually think of this list of spaces as represented by a graph.

The high level algorithm for generating trajectories is as follows: we first loop from the simulation start date to the simulation end date; for each day that goes by, each agent is assigned a conceptual start / end time, and must continually attend events until the end of their day. We note that many details have been intentionally omitted here because this is not the focus of the paper.

3 Key Objective

In this project, we focus on the scalability of spaces to nodes. That is, we wish to explore how differing methods of distribution of spaces to nodes can change the efficiency / scalability of the simulator.

4 Use Case Scenario: University

To contextualize our work, we use the example of a university use case. In our university use case, the agents of our simulation come from one of several groups: professors, undergraduate students, graduate students, janitors, administrative staff, and visitors. The space in which our university use case scenario occurs in is the DBH building; in particular the first and second floors are simulated. Lastly, the events in the building can range from lectures and discussions, to study meetings, project meetings, and office hours.

5 Related Work

It is clear that there is a need for a more sophisticated understanding of real-world phenomena; this is accompanied by the need of scalability in simulations. In order to simulate an IoT environment for example, simulators must mimic the behavior of such devices. Thus, it is natural to attempt to exploit the parallelism of simulations in a parallel or distributed manner. We note a small difference between parallel and distributed systems: parallel simulations are typically performed on tightly coupled multiprocessor platforms, whereas distributed simulations adopt loosely coupled distributed computing platforms. In addition, while parallel simulations can generally communicate through the use of shared memory, distributed simulation must rely upon message passing. In this project, we focus on exploring distributed simulation; however we will survey both types of simulations.

The framework of parallel or distributed simulations typically has three layers: the simulation model, the simulation engine, and the top to bottom system [1].

The **Simulation model layer** includes codes and states of the simulation. The code is the abstraction of applications since different applications have different behavior; that is, different events would occur. Moreover, for each local process of the simulator, it maintains some states for tracking and triggering the events of the simulation.

The **Simulation engine layer** consists of the software that manages the simulation. For instance, the synchronization algorithm maintains the causalities of events, which ensures the execution order of events. Other examples are communication (messages) between processes, data distribution, and so on. The objective, workload balancer, belongs to this layer as well.

The **System layer** contains the low-level components related to system configuration and protocols, such as power mode management and communication protocols.

Other than proposing the above framework, Fujimoto et al. [1] empirically analyzed the energy efficiency of different layers with different components. For instance, they studied the energy consumption of different synchronization algorithms for events. Two types of synchronization algorithms are used: conservative and optimistic. The conservative synchronization algorithm proactively keeps track of the execution order of events, while the optimistic synchronization algorithm detects the synchronization errors during the simulation, and once an error happens, it rollbacks and recovers from the error. Varies experiments were conducted, and the authors conclude that a high volume of message exchange consumes much higher energy. Although the workload balancer lies in the simulation engine layer, the authors only scratch the surface of it and fail to propose a solution for it, which is the main objective of this project. Moreover, by balancing the workload among processes, one can imagine that the workloads of messages among processes are evened, and hence may help reduce the energy consumption.

Similar to the conclusion of [1], Wang et al. [2] concluded that the overhead of simulations is mostly from communication. Nonetheless, they focus on parallel simulations for multi-core systems, which is not the paradigm adopted by our project. The authors conducted empirical experiments to understand the performance of simulation under different communication methods through shared memory. In the multi-core system, two types of shared memory methods are exploited, which are Message Passing Interface (MPI) and multi-thread. The message exchange between processes through shared memory space. Besides, each

process maintains an output queue of events and sends the event to the shared memory when it is available. Hence, the number of messages (events) transmitting between processes becomes critical for the performance of the simulator. On the other hand, in a multi-thread system, threads have a shared address space, and thus there is no need for each thread to maintain its queue. However, the performance is bounded by the number of physical threads of a system. Moreover, memory allocation becomes an important issue for multi-thread simulation in multi-core systems. If the message (event) is allocated far from the thread, the read time dominates the execution time.

Apart from hosting the simulation on resourceful devices such as clusters and servers, Maqbool et al. [3] proposed to perform simulations on mobile devices due to a large number of growth in mobile devices. The authors performed empirical experiments to understand the energy consumption of three different synchronization algorithms, which are time-stepped (no need for synchronization), conservative, and optimistic. However, our project aims to simulate the IoT environment, where the number of simulated devices is large, and hence not applicable in simulation on mobile devices.

Grande et al. [4] proposed several workload prediction methods for distributed simulation systems and then utilize the prediction model to migrate jobs between processes. However, they apply simple migration policies, which are mainly switching the jobs from overloaded processes to underloaded processes. Lack of an overall objective, such as workload balance among processes, is the main reason for us to develop a workload imbalance sensitive algorithm for distributed simulation systems. Moreover, the adopted time-series oriented prediction model is a simple look-back method, which accounts for the historical data for a period of time. Hence, it inclines to be affected by the local bias of data, especially for non-deterministic simulations. The author dedicated themselves to eliminate the effect of local biases. However, the method may not be applied to all types of simulations. Hence, a more sophisticated learning tool, such as neuron networks, can be applied to solve the problem. For our project, accurate prediction can help us to maneuver the workload distribution among processes. Nonetheless, our primary focus now is offline workload distribution. We leave workload prediction for future work.

Aside from the discussion on whether simulations should be parallel or distributed, there has been varying work done in scaling up simulations using MapReduce. In [5], Wang et al. present a computation engine that relies on the MapReduce framework to parallelize work. Here, they present a programming langauge akin to database lanaguages, and drive simulations by "large iterated spatial joins". The authors mention that such simulations often have associated patterns (state-effect pattern), and the neighborhood property. To this end, they can exploit such assumptions to assign spaces to nodes. The authors discard the possibility that splits based on other properties (e.g., degree, type of space) can be efficient, while we aim to test the validity of such splits. With this in mind, [6] introduces a MapReduce framework to support spatial data. In this framework, we note that there are several key operations that are provided: range queries, kNN, and spatial joins; the authors cite that oftentimes, Hadoop is viewed as a black computation box, and that there is that can be exploited. Both authors implement a language to work with their framework. The authors go on to more specifically describe the operations and data structures used.

With regards to the IoT, there has been research done on the scalability of simulation platforms over the past few years. In 2012, Looga et al. [7] introduced their framework for a massive scale emulation platform for the IoT. In their work, called Mammoth, they aim to get linear scalability through the placement of virtual machines. However, our work differs from that of Mammoth in that we focus more on the simulation side, not the emulation of devices. It can be noted that the authors spend a great deal of time simulating network traffic to help emulate IoT devices better.

In another paper, Angelo et al [8] argues for the necessity of agent-based, parallel and distributed simulation approach to multi-level simulation, using a smart territory use case. The authors explore discrete event simulation in a parallel way along with dealing with synchronization issues. As they continue with their paper, they lay out a few different models that could be used, but ultimately do not propose a concrete solution to how scalability can be achieved. The same authors also discuss how such a simulation approach would be applied to vehicular movement in [9], making similar insights. In a more recent paper, Ferretti et al. [10] relate scalability and accuracy together through the use of a two-level simulator. The authors discuss the usage of the discrete event simulator OMNet++. This simulator is used to create network simulations. Unfortunately, the authors fail to successfully enable scalability with OMNet++ itself, but simply make a proposal for the viability of their approach. In the work, the authors resort to simply using multiple local machines to enable scalability; we wish to explore how deployment to the cloud could enable scalability.

6 System Architecture

We will introduce our system architecture in this section, including trajectory generator, space maintenance module, and the queries / results handlers, as shown in Figure 1. The detailed descriptions of our system are given as follows.

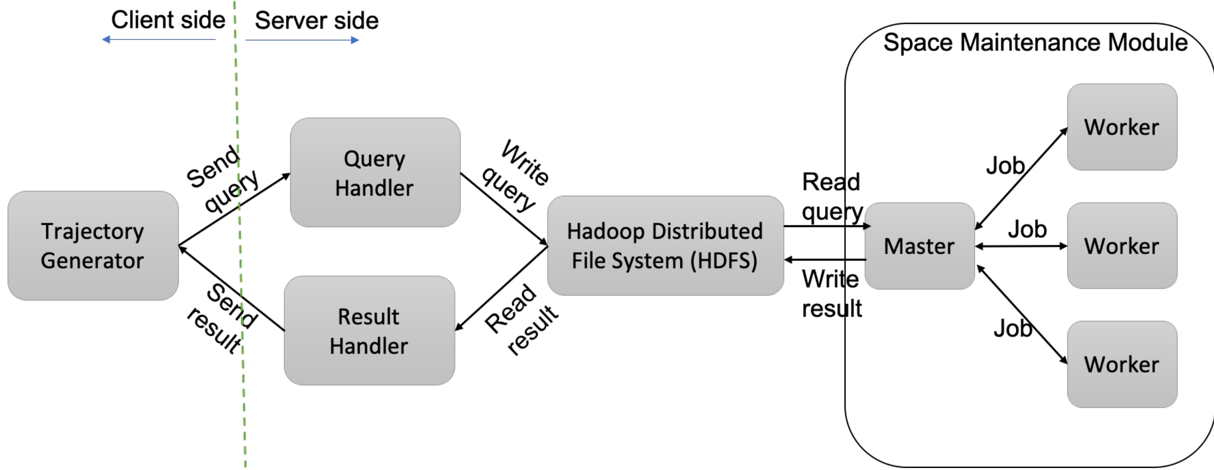

Figure 1: System Architecture

Distribution Method that we adopt is similar to the client / server paradigm. The trajectory generator can be seen as a client. On the other hand, the space maintenance module and the queries / results handlers are the components of the server. For the client-side, in order to decide whether a space has enough room for agents to attend the event, we have the trajectory generator keep querying the space maintenance module about the occupancy information of spaces. For the server-side, the space maintenance module maintains the occupancy of each space with the data structure, called *Interval Trees*. We then implement two handlers to handle the queries and the query results stored in the Hadoop Distributed File System (HDFS). Specifically, the trajectory generator has the following queries, such as getOccupancy, insertOccupancy, and getPath, which asking the occupancy per room over time, updating the occupancy per room, and asking the paths that an agent will traverse, respectively.

Trajectory Generator maintains when each agent attends which event during the simulation period. For example, on 2020/03/12, agent 1 attends one class starting from 10:30 am to 12:00 pm and one seminar starting from 1:30 pm to 3:00 pm. Besides, each event has a corresponding space where the event occurs. However, the information of spaces, such as occupancy, is maintained by the space maintenance module. Hence, the trajectory generator needs to query the space maintenance module about whether the spaces have enough room for agents. The queries are sent to the query handler via a socket, and then the query handler stores the queries into HDFS so that the space maintenance module can process the queries in a batch manner with Apache Spark (detailed later). Note that the current version adopts a linear query method, and we leave the batch query as the future work.

Space Maintenance Module resides in an Amazon EMR cluster and is run by Apache Spark, a batch processing engine. A batch of data to be processed is wrapped into a unique data representation called *Resilient Distributed Dataset (RDD)*, which can be operated on in parallel with fault-tolerance. The space maintenance module is composed of one master node and several worker nodes. The master node is responsible for distributing each job that is processing an RDD to a corresponding worker node. The queries are stored in HDFS so that the space maintenance module can fetch them distributedly.

Moreover, for each space, we exploit an interval tree to store its occupancy of corresponding periods. In our system, we encapsulate the interval trees into RDDs for further processing asked by input queries. Besides, in order to specify the exact worker node to be responsible for a query, we use the *makeRDD* function of Apache Spark as a workaround method. Apache Spark does not allow users to assign jobs to a particular worker node directly. As a workaround, we exploit the policy of Apache Spark, which lets the

worker nodes process the jobs on the RDDs stored in their memory. Hence, we use the function to specify the location of RDDs so that the master node would assign the corresponding worker node to process them.

Query / Result Handlers, which are written in Python, handles the input query from the trajectory generator and returned results from the space maintenance module. Each handler has its own server socket connected to the corresponding client socket in the trajectory generator. First, the query handler keeps receiving the input queries from the trajectory generator and write the queries into HDFS. On the other hand, the result handler keeps monitoring the HDFS. As soon as a result is written into the HDFS by the space maintenance module, the result handler sends it back to the trajectory generator.

6.1 Distribution Methods

In this section, we will describe the different methods by which we will distribute spaces to nodes. For the project, we use a static mapping of spaces to nodes, because we want to directly control where each node gets placed.

Degree-based Mapping. Spaces are modeled as a graph in the IoT Data Generator; we can therefore use properties of graph to partition. In this scheme, we use the degree of a space s: the number of neighboring spaces that s has. The table below describes the how the mapping of spaces to nodes was constructed.

Node	Spaces Assigned
Node 1	Spaces with degree 1
Node 2	Spaces with degree 2
Node 3	Spaces with degree 3
Node 4	Spaces with more than degree 4

Proximity-based Mapping. Conceptually, the space that we simulate is a university building; in particular, we simulate two floors of DBH. Thus, a natural partition to make is by floors. The table below shows the mapping:

Node	Spaces Assigned
Node 1	Spaces on the 1st floor
Node 2	Spaces on the 2nd floor

Type-based Mapping. Each space can be associated with its function in the building. In our university case study scenario, there are offices, lecture halls, classrooms, among more mundane spaces such as hallways, elevators, etc. In this type-based mapping, we assign spaces to nodes based on the type of space they are. The table below shows the mapping done:

Node	Spaces Assigned
Node 1	Outside, Exits, Stairs, Elevators
Node 2	Hallways
Node 3	Lobby, Restrooms, Classrooms, Lecture Halls, Conference Rooms
Node 4	Office Rooms, Kitchen, Other

6.2 Iterations: Implementation Challenges

In this section we will describe some of the difficulties and implementation challenges we faced during this project.

Sockets Only

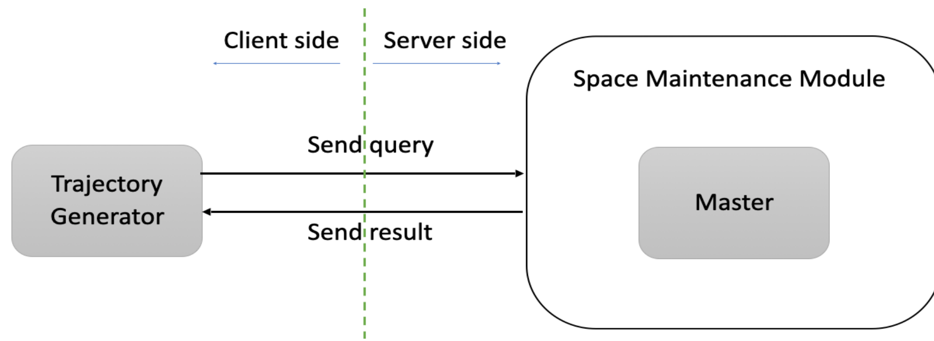

Figure 2: System Architecture

The simulation is consist of two main parts, the Space Maintenance Module, and a Trajectory Generator. Our first step is to isolate the Space Maintenance Module and the Trajectory Generator, and have the two ends communicate using sockets. The Space Maintenance Module is written in Python, and the Trajectory Generator is written in C. Both ends would be running on port 5457, and the packets are in string. Example given, "P0,1422!" and "G1422,1577865572!". At this stage of the development, there is no spark involved, and we were expecting to use PySpark for further development.

Spark with Sockets

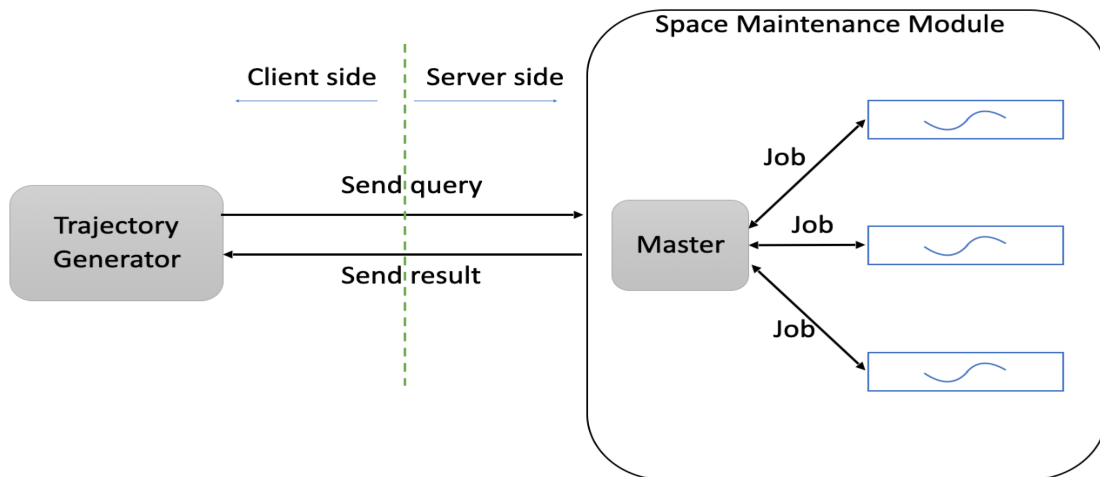

Figure 3: System Architecture

After some trial and failure on the previous step, we faced the difficulty that the Python package PySpark does not allow us to define our own distribution method. So we had to re-write the Space Maintenance Module in Scala. In this stage of the development, we were able to run the Space Maintenance Module on the master node, but not able to distributed the workload to the entire cluster when running on cluster mode as the sockets were not functioning. This is perhaps due to the nature of Spark. To execute jobs, Spark breaks up the processing of RDD operations into tasks, each of which is executed by an executor. Prior to execution, Spark computes the task's closure. The closure is those variables and methods which must be visible for the executor to perform its computations on the RDDs. This closure is serialized and sent to each executor. The variables within the closures sent to each executor are now copies and thus, when a variable, in this case the socket, is referenced, it's no longer the original one on the driver node. There is still a variable in the memory of the driver node but this is no longer visible to the executors. The executors only see the

copy from the serialized closure. A simple way to solve this is to write all the queries generated by the data generator into a file and read from it, and in the end, we decide to use the HDFS file system as mentioned in the previous sections.

Spark Streaming In Figure 4, we show the architecture of this iteration. In this iteration, we try to connect the trajectory generator and the space maintenance module via Apache Spark Streaming directly. However, we faced two issues: the data format and the one-way property of Apache Spark Streaming. First of all, instead of RDD, the data format of Apache Spark Streaming is Discretized Stream (DStream), which represents a continuous stream of data. Moreover, it is nontrivial to convert a DStream to an RDD. Second, Apache Spark Streaming provides only one-way streaming, which means we cannot directly send the result back with it. Instead, we need to send the result to another platform to handle it, such as Apache Kafka and HDFS.

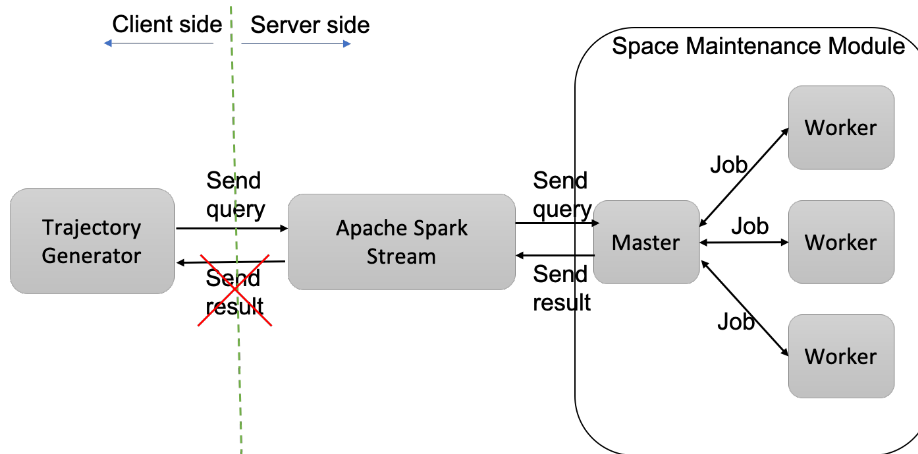

Figure 4: Iteration with Spark Streaming

7 Experiments: Evaluation Plan, Results and Analysis

For each of the partitions discussed earlier, we will run our IoT Data Generator with 5 agents and 20 events - all randomly generated. For each of the experiments, we will record the execution time (the total time taken), the task time (the amount of time that worker nodes are doing work), and the average number of tasks across nodes. The table below shows our results.

	Total Time	Task Time	Average nTasks
Degree	46 min	1 min	883.8 across 4 nodes
Proximity	1.4 hr	1.7 min	4452 across 2 nodes
Type	1.8 hr	1.9 min	1668.5 across 4 nodes

Total Time. We define this as the amount of time it takes for us to wait until obtaining a result. That is, we can think of this as the execution time. The first thing that we can note here is that all of these experiments take a very long time to run, and thus is not suitable for real-time execution. Looking at this time, we speculate that the degree distribution did the best at 46 min; proximity in second place, and type in third.

Task Time. The Spark History Server allows us view some statistics involving the jobs that we run on the nodes. One of this is the task time, which is defined as the amount of time that is taken for the actual processing of a task. We note that the task time is many orders of magnitude smaller than the total time, which implies to us that there was a bottleneck in the I/O and communications done between HDFS and the master node.

Average nTasks. Another piece of information that the Spark History Server provides is the number of tasks that were handled by the nodes. We can see that one of the possible reasons that the degree based mapping takes a shorter amount of time is that there are less tasks to do.

One thing that we notice is that when the average number of tasks is set to be roughly equal to each other, all total times / task times come to be within similar values with each other. There are two possible reasons for this. The first is that the partitioning schemes presented do not make a difference in the execution time of the simulator. Since our scheme only uses spark to act similar to a distributed database (in its current implementation), it may not matter where the data is stored. Another possible reason the small input size given to the simulator: 5 agents and 20 events. The small number of entities could possibly play a large role in the number of tasks run. To this end, we attempted to run the simulation multiple times per partitioning scheme, but were forced to lower the number of agents / events to minimize the amount of time taken per experiment.

8 Conclusion and Future Work

There are a number of things that we have learned from this project. The first is that the IoT Data Generator may need to be redesigned in order to accomodate scalability across multiple nodes. This is most evident from the data collected, which suggests that there is large bottleneck in the I/O and communications between our local and cloud nodes. Therefore, it is evident that we must split the work in a more reasonable way. One way this might be accomplished is once again by distributing the number of spaces across nodes, but to additionally split up the agents and events so that each node would be responsible for the trajectories of a number of agents.

In addition, we acknowledge that there is a need to study the degree to which each of the entities driving our trajectory simulator interacts with each other. This is important to characterize in order to be able to more effectively distribute the simulator to multiple nodes.

References

[1] R. M. Fujimoto, M. Hunter, A. Biswas, M. Jackson, and S. Neal, "Power Efficient Distributed Simulation," in *Proceedings of the 2017 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pp. 77–88, 2017.

[2] J. Wang, D. Jagtap, N. Abu-Ghazaleh, and D. Ponomarev, "Parallel Discrete Event Simulation for Multi-Core Systems: Analysis and Optimization," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 6, pp. 1574–1584, 2013.

[3] F. Maqbool, A. W. Malik, I. Mahmood, and G. D'Angelo, "SEECSSim - A Parallel and Distributed Simulation Framework for Mobile Devices," in *2018 IEEE/ACM 22nd International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pp. 1–7, IEEE, 2018.

[4] R. E. De Grande, A. Boukerche, and R. Alkharboush, "Time Series-Oriented Load Prediction Model and Migration Policies for Distributed Simulation Systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 215–229, 2016.

[5] G. Wang, M. V. Salles, B. Sowell, X. Wang, T. Cao, A. Demers, J. Gehrke, and W. White, "Behavioral simulations in mapreduce," *arXiv preprint arXiv:1005.3773*, 2010.

[6] A. Eldawy and M. F. Mokbel, "Spatialhadoop: A mapreduce framework for spatial data," in *2015 IEEE 31st international conference on Data Engineering*, pp. 1352–1363, IEEE, 2015.

[7] V. Looga, Z. Ou, Y. Deng, and A. Ylä-Jääski, "Mammoth: A massive-scale emulation platform for internet of things," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, vol. 3, pp. 1235–1239, IEEE, 2012.

[8] G. D'Angelo, S. Ferretti, and V. Ghini, "Simulation of the internet of things," in *2016 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 1–8, IEEE, 2016.

[9] G. D'Angelo, S. Ferretti, and V. Ghini, "Modeling the internet of things: a simulation perspective," in *2017 International Conference on High Performance Computing & Simulation (HPCS)*, pp. 18–27, IEEE, 2017.

[10] S. Ferretti, G. D'Angelo, V. Ghini, and M. Marzolla, "The quest for scalability and accuracy: Multi-level simulation of the internet of things," in *2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pp. 1–8, IEEE, 2017.