CIRRELT
1971 – 2021

CIRRELT-2021-33

Internship Report

Simulation of A Multi-Period Freight Transportation System for Shipping Corridors

Ali At-Tayeb

August 2021

Bureau de Montréal
Université de Montréal
C.P. 6128, succ. Centre-Ville
Montréal (Québec) H3C 3J7
Tél : 1-514-343-7575
Télécopie : 1-514-343-7121

Bureau de Québec
Université Laval,
2325, rue de la Terrasse
Pavillon Palasis-Prince, local 2415
Québec (Québec) G1V 0A6
Tél : 1-418-656-2073
Télécopie : 1-418-656-2624

Simulation of A Multi-Period Freight Transportation System for Shipping Corridors*

Ali At-Tayeb

Abstract. We introduce a simulator model and instrument for the operations of multi-stakeholder freight transportation systems. The systems are characterized by an integrated optimization of the selection of shipper transportation requests, the selection of carrier transportation and storage capacity offers, the time and space consolidation-based assignment of accepted requests to selected services and departure times, and the consequent movement of freight. The simulator integrates the connection to optimization modules.

This document describes the project, including 1) the functionality, structure, components, and working principles of the model; 2) the design and implementation of the software; and 3) the experimentation and performance analysis of the simulator. The publication also includes a User Guide and a Developer Guide.

Keywords: simulation model, simulation - optimization integration, many-to-one-to-many freight transportation systems

Acknowledgements. Funding for this work was provided by the Natural Sciences and Engineering Council of Canada (NSERC) through its Collaborative Research and Development Grand program. This funding is gratefully acknowledged.

* Internship period January-June 2021, under the supervision of Professors T.G. Crainic, W. Rei and M. Gendreau.

---

* Corresponding author: attayeb.a@gmail.com

Dépôt légal – Bibliothèque et Archives nationales du Québec
Bibliothèque et Archives Canada, 2021

Contents

1 Introduction

With items being sent across different locations around the world, networks to efficiently get these items from one location to the next are necessary. Whether a producer sending their goods to a warehouse or an individual, cost and time constraints are often variables considered in one's decision when choosing a carrier to complete the shipment of goods. More specifically, it is in the shippers's interest to have the shipment carried within their time constraints at the lowest possible cost. On the other end of the spectrum, carriers wish to travel a route with the most profitable load of items within their physical and temporal constraints.

The *Centre interuniversitaire de recherche sur les réseaux d'entreprise, la logistique et le transport* (CIRRELT), the organization with which I am doing my internship, focuses on developing knowledge on the design, management, operation, safety logistics, network technologies of network organization and infrastructures [1]. As part of the ongoing projects at the CIRRELT, they are working on a state-of-the-art transportation 4.0 automated decision-making system which optimizes operations and profitability for both freight shippers and carriers. In a brief overview, the system receives requests from shippers to have items carried across two locations and offers from carriers to fulfill these shipment requests. Based on temporal, physical and economic attributes of the shipper requests and carrier offers, the system optimizes shipment-to-carrier assignments which aims to reduce a shipper's delivery cost and increase a carrier's revenue all while respecting their imposed temporal and physical constraints.

While designing these complex systems, to replicate their performance in a real world setting, it is desirable to model them through discrete event simulation (DES). DES allows the decomposition and modeling of a real world system into a set of distinct processes. In DES, time progresses in discrete time steps and at each step, an event may occur at each of these processes. Between these events, the system is considered to be in a fixed state and unchanging. The resulting output of each event may be used to affect the behavior of other processes at the current or future time steps [2]. Of its benefits, a DES model may be used to experiment a system under different parameters without the financial implications of testing each considered variant of the system in a real world setting.

My six month internship with the CIRRELT revolved around implementing a DES model which replicates the behavior of the state-of-the-art transportation system being developed. In Section 2, we explore the problem statement being addressed in more detail. Section 3 covers how phases of the project were split, while Sections 4, 5, 6 and 7 respectively cover the chosen DES software, the model's design, the implementation and testing phases. Lastly, Section 8 covers the knowledge I gained and improved throughout the internship.

2 Problem Statement

In this project, the overall task revolves around sending goods from a shipper facility to a consignee location. These shipper facilities and consignee locations may be located across different geographical regions connected by various modes of transportation such as roads, railroads, sea, air travel, etc. To fulfill these shipment requests, carriers make offers to transport goods from one location in the network to the next. The problem at hand becomes efficiently assigning these shipment requests to carrier offers all while making sure the physical, temporal and economic constraints of both parties are satisfied.

To complete this task, we present the *Many-to-One-to-Many* (M1M) system. The M1M system is broken down into three distinct components; shippers, carriers and an *Intelligent Decision Support Platform* (IDSP).

The first *M* in the M1M system represents shippers. We define a shipper as an entity which requests that a good be transported from a facility to a consignee location. Shippers include producers, wholesalers, distributors, logistic-service providers, etc. A shipper request is characterized by its physical, temporal and economic parameters. The physical parameters of a request describes the volume and dimension of the items which need to be delivered. Temporal attributes cover information such as the time at which the items are available to depart the shipper facility and the desired time by which they should arrive at the consignee location. Economic parameters include the revenue of fulfilling the shipment as well as possible penalties if some temporal constraints are not met.

The second *M* in the M1M system represents carriers. We define a carrier as a company which offers services to transport goods across locations. A carrier's offer is characterized by geographical, physical, temporal and economic parameters. The geographical characteristic of an offer represents the route and travel time a carrier will take between its origin and destination locations. We note that these routes can be serviced by various modes of transportation such as trucks, trains, cargo ships, airplanes, etc. The physical aspect of an offer indicates the volume capacity the carrier can transport while the temporal aspect indicates the time window in which the carrier is available to perform the given offer. Lastly, the economic aspect of an offer covers the fixed cost of utilizing the proposed offer as well as the variable cost representing the cost of transporting an item per unit of volume and distance.

The last component of the M1M system, the IDSP represented by the *1* notation, takes as input the shipper requests and carrier offers. From their given geographical, physical, temporal and economic parameters, the IDSP assigns each shipper request to a carrier offer. For shippers, it looks to assign their items to the carrier with the lowest cost within the request's constraints. For carriers, the IDSP looks to use as much as their capacity as possible within their given route and temporal constraints to avoid travelling with some free space and losing possible revenue.

From the described problem setting , the main goal of this project revolved around building a DES model conceptualizing the presented M1M system. Of the functionalities

to design, the DES model has to be able to simulate the movement of the carriers across a hyper-corridor network described in more detail in Section 5.1. Additionally, for a given list of shippers and carriers, data representing requests and offers based on the presented parameters has to be generated. Moreover, an optimization model which performs the shipper-to-carrier assignments based on user provided or the randomly generated data is to be integrated into the simulation model. Lastly, the designed DES model is to be able to keep track of the movement of the carriers across the physical network while collecting logs and statistics on the system as a whole. In Section 5, we define in more detail each functionality of the DES model as well as the workflow of the designed DES model which allows these various functionalities to integrate with one another.

The goal behind the presented DES model is to be able to experiment the system under different network configurations, shippers, carriers, optimization models, economic penalties, etc. Collecting logs and statistics about the system under various parameter combinations gives the ability to examine the benefits or drawbacks of a parameter and allows for operational decisions to be determined based on the observed results.

3 Task & Timelines

The six months internship, running from January to the end of June, was split into the following stages. The first stage, lasting about one and half months, consisted of familiarizing myself with the goals and components of a DES model. Additionally, an analysis of the available DES software to build the simulation model on was performed. Lastly, this time period was also used to follow tutorials and learn the concepts behind the used DES software.

The second stage, running from mid February to the end of March, focused on defining the functionalities of the simulator as well as its design. The third stage, also lasting about a month and a half, was geared towards implementing and testing the components designed in the previous stage. Over the course of these 3 months, these two stages largely blended into one block as the design and implementation were done in an iterative process. When designing an individual component, they were implemented and tested before moving on to the next component.

The fourth stage, lasting about a month, consisted of revisiting some designed components and experimenting the built simulation model. As my understanding of the designed system and its concept improved over the months, some previously designed components were modified to better fit the operational plan of the M1M system [3]. Moreover, with all building blocks of the simulation model implemented, the model was tested under different scenarios to validate its proper functioning.

Lastly, the remaining weeks of the internship were spent writing a user guide (included in Appendices A, B and C) and finalizing this report. The user guide destined for future users and contributors of the designed simulation model, covers aspects such as installing the

chosen DES software, running the simulation, the DES model's parameters, how to modify some of the designed functionalities, etc. Additionally, the remaining weeks were used to train two master's student from Polytechnique Montreal who are doing their research with the CIRRELT on how to use and modify the designed DES model. Their feedback on the training and user guide were utilized to adjust the user guide in order to make it more comprehensible for future readers.

4 Stage 1: Software Analysis

To perform a large-scale discrete event simulation, a computer software is called upon. Using a software allows one to more easily model the simulation task at hand and collect a multitude of metrics aiding in evaluating its performance. As an initial step in building a DES model, a DES software had to be chosen.

There are a multitude of widely known simulation software readily available such as *AnyLogic, Arena, AutoMod, GPSS, Flexsim, etc.* However, a drawback of using these software is their lack of ability to interact with external components such as a custom optimization model. Thus, we decided to explore the use of an open source DES software.

An open source software (OSS) is a software where the source code is publicly available for the general public to utilize, inspect, modify or enhance [4]. In our use case, using an OSS allows us to use the preexisting foundation of a simulation software all while being able to integrate a custom built data generator and optimization models which are part of the simulated system. These pre-designed software functionalities reduce the need of investing effort in implementing multiple fundamental DES functionalities and gear our focus towards only implementing missing functions which are specific to our use case.

4.1 Software Criteria

To choose a DES OSS for our simulation task, we set some criteria we believe would be beneficial in the long term development of this project. The criteria can be split into two sub categories; required criteria and desired criteria. A required criterion is one where if an OSS lacks it, we remove it from our list of considered options. A desirable criterion is one that would be beneficial to have, but can be worked around depending on the other aspects of the software.

As a first step in filtering OSS to consider, I visited sources such as Wikipedia and widely used version control software sites, such as GitHub and SourceForge. Version control software sites are usually where the source codes and project architectures of OSS are shared. Within those sources, a simple criterion was whether the OSS's source code was easily accessible and could be installed on a Windows machine. Based on these criteria, the following list of

OSS were retained: *DESMO-J [5]*, *Facsimile [6]*, *JaamSim [7]*, *JSimpleSim [8]*, *ManPy [9]*, *NS-3 [10]*, *OMNeT++ [11]*, *OpenSIMPLY [12]*, *SimJulia [13]*, *SimPy [14]* and *Stochastic Simulation in Java (SJJ) [15]*. A complete representation of the considered software and the examined criteria can be found in Table 1.

OSS Name	Operating System	Language	Last Update	Community	GUI	Licensing
DESMO-J	All	Java	2014	No	No	Free
Facsimile	All	Scala	Within 12 months	Yes	No	Free
JaamSim	All	Java	Within 6 months	Yes	Yes	Free
JSimpleSim	All	Java	Within 6 months	No	No	Free
ManPy	All	Python	2016	Yes	Yes	Free
NS-3	All	C++	Within 6 months	Yes	Yes	Free
OMNeT++	All	C++	Within 6 months	Yes	Yes	Paid
OpenSIMPLY	Windows,Linux	Pascal	Within 6 months	No	No	Free
SimJulia	All	Julia	2019	Yes	No	Free
SimPy	All	Python	Within 6 months	Yes	No	Free
SJJ	All	Java	2018	No	No	Free

Table 1: DES OSS specifications

4.1.1 Required Criteria

In the OSS analysis, required criteria were listed as the operating systems which the software is compatible with, the programming language behind the software and the frequency at which they are updated.

There are three main computer operating systems generally used; Windows, Mac OS and Linux [16]. Having the OSS compatible with the three operating systems eases the task of making the designed DES model available to most users and future contributors. In Table 1, the annotation "All" represents an OSS compatible with these three operating systems.

Moreover, there is a large variety of programming languages available on which DES OSS are built. To facilitate multiple users being able to make changes to the developed model, we wanted to focus on programming languages which are widely used such as Python, Java or C++ [17].

Lastly, an important aspect considered is the frequency at which the OSS is updated. This represents whether the software is continuously maintained to address any underlying issue and also promotes the possibility of new useful features being integrated in the future. For this purpose, we considered software with a new release not more than a year ago.

With the above required criteria considered, the OSS remaining in our analysis were JaamSim, JSimpleSim, NS-3, OMNeT++ and SimPy. That being said, further looking into SimPy, it is a python framework rather than a stand alone software. Being a framework would facilitate the process of building DES components, however, it does not align with our goal of building components on top of an existing OSS. Thus, it was not considered in

the subsequent steps of our analysis.

4.1.2 Desired Criteria

As a desired criterion, we considered whether the OSS has an existing graphical user interface (GUI), an active community of users, licensing requirements and the OSS's industry of focus.

To begin with, in terms of GUI, the only remaining OSS lacking this aspect is JSimpleSim. On the other end, JaamSim and OMNeT++ present a strong user interface with a drag and drop functionality allowing to seamlessly build the networks of the simulated system. This aspect is specifically of interest if members contributing to the project have more limited programming knowledge or are not familiar with the Java and C++ programming languages on top of which these OSS are built. As for the remaining OSS, NS-3, it has a GUI to have a visual representation of the simulation model, but would still require some programming knowledge to build said models.

Moreover, having a community of active users opens up the ability to have insight from other users in the prospect of being stuck with a specific functionality of the software. An active community on an OSS is also an indication of the software being actively used which promotes the possibility of new features or improvements being implemented. Excluding JSimpleSim, the remaining three OSS have active communities to both answer questions and address bugs in the software [18][19][20] .

In terms of licensing, JaamSim, JSimpleSim and NS-3 are under licenses which allow their usage without charge under commercial purposes. OMNeT++ follows a different licensing scheme where the software's use is permissible without a purchased license for educational purposes, but for industrial purposes or research for industrial purposes, a license must be purchased. That being said, once the system is used in a commercial setting, the licensing may prove to be advantageous given the additional support and features geared towards commercial usage offered by OMNeT++ [21].

Lastly, as DES models can be applied for a wide range of industries, some of the analyzed software are geared towards more concise use cases. For instance, NS-3 and OMNeT++ were originally designed to simulate computer networking models. Although OMNeT++ has adapted the software to address a more general range of DES models, our research shows that NS-3 is still heavily targeted for simulations specific to computer networking.

4.1.3 Chosen Software

With the above criteria considered, given the availability of a GUI, active communities and their ability to be integrated in the transportation industry, we opted to narrow our analysis to either JaamSim or OMNeT++.

After deliberation with the professors overlooking the project, we opted to choose OMNeT++. With some of the project's team members having previous experience with OMNeT++, their prior knowledge of this software could prove to be beneficial in terms of best practice usage and better understanding of some OMNeT++ specific functionalities. Lastly, as the project expands, OMNeT++ dedicated support for licensed users could also prove to be beneficial rather than solely depending on the software's user community.

In terms of version, the latest stable release of OMNeT++, version 5.6.2, was used. Although OMNeT++ version 6 is released, it is marked as a pre-release with some non backward compatible features being added [22]. In other words, a simulation model built on version 6 would not function in OMNeT++ version 5.6.2. On the other hand, a simulation model built on version 5.6.2 only requires very few changes to be adjusted to version 6.

Additionally, the newer version of OMNeT++ is only compatible with the latest macOS and Windows versions. Version 5.6.2 thus proved to be more favorable in terms of being accessible to the largest audience of users and future contributors.

5 Stage 2: Simulation Functionalities & Design

With the DES OSS chosen, the next step consisted of defining the functionalities and design of the simulation model. In Section 5.1, we start by defining the physical network across which items are to be delivered. Moreover, we cover the time aspect of the simulation and introduce the operation planning horizon concept in Section 5.2. Next, Section 5.3 covers the random data generator entity and the constraints behind the generated data. Section 5.4 details the optimization entity of the simulation and the currently used optimization models. Lastly, Section 5.5 covers the workflow of the simulation indicating how the events presented in the previous sections connect to one another, how the carriers navigate the physical network as well as the collected logs and statistics.

5.1 Shipping Network & Terminals

To simulate goods being shipped from a shipper facility to a consignee location, we defined a physical network across which these facilities are located. In Figure 1, we display a sample physical network consisting of shipper facilities, zones, terminals and long haul segments. In the displayed figure, numbered disks enclosed in a larger disk represent the shipper facility while the outer disk holding them represents a zone. The lettered squares depict a terminal with the arrow between two squares representing a long haul segment connecting two terminals. For the remainder of the report, the aggregation of shipper facilities, zones and terminals will be referred to as nodes. Below we describe the physical network's components in more detail.

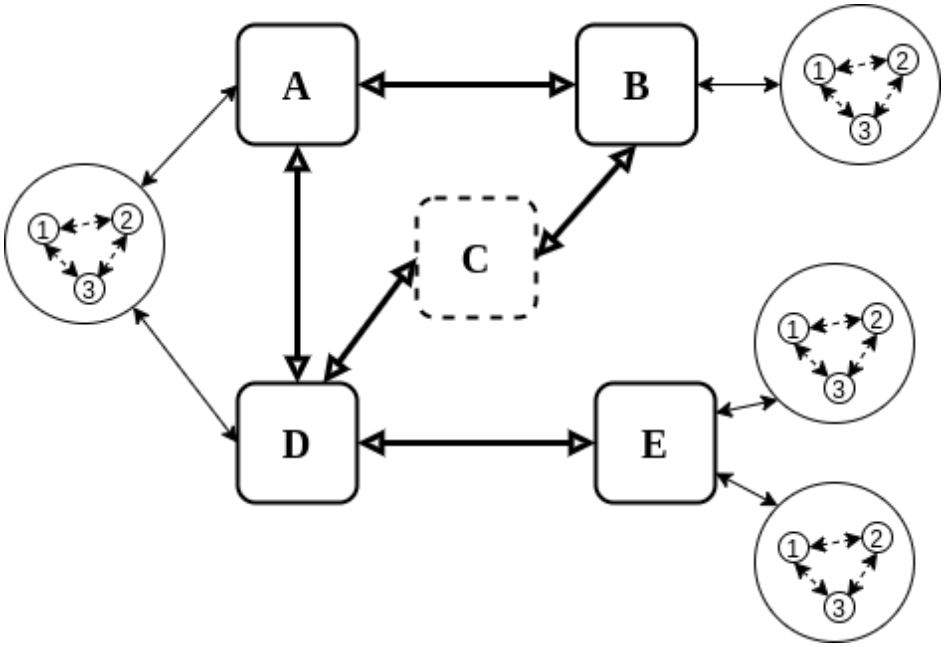


Figure 1: Sample Physical Network

5.1.1 Shipper Facilities & Zones

Zones represent any administratively or commercial relevant area such as an urban region. Within these zones are shipper facilities and consignee locations that represent the origin and destination location of items which need to be shipped. In the current design, a single numbered disk within a zone represents both a shipper facility and a consignee location.

5.1.2 Terminals

In Figure 1, terminals are depicted through lettered squares. Terminals are locations which can perform 5 possible services.

1. Receive shipments from nearby zones
2. Sort shipments received from other terminals or zones preparing them for the next leg of the journey
3. Transfer shipments from one vehicle to another without sorting
4. Storing shipments for them to be picked up by another carrier at a later date
5. Preparing shipments to be distributed to nearby zones for shipments to the consignees

As displayed in Figure 1, a terminal may service multiple zones and a zone may also be serviced by multiple terminals. However, not all terminals are connected to a zone. There are two types of terminals; squares with solid lines depicting consolidation terminals and squares with dotted lines depicting transfer terminals. A consolidation terminal, which is connected to zones, can perform all five operations listed above while transfer terminals can only perform operations 2 to 4. Transfer terminals are terminals not located near a zone and are used to transfer or sort shipments from one carrier vehicle to the next.

In the designed DES model, to simulate services which can be performed at a node, time delays are introduced. Each node is assigned a speed at which they perform a service. The speed is denoted as the volume of items which can be processed during a single period. For example, if a node could sort 5,000 cubic feet of items per period, it would take up to two periods to sort the load of a carrier transporting items amounting to a volume of 7,500 cubic feet. Consequently, if the volume of items transported by the carrier is greater than the volume of items which can be processed in a single period, delays may occur.

As opposed to the pick up, sorting, transfer and drop off services, storing items entails more than simply consuming time. For storage services, the physical capacity of a terminal is also considered. Each terminal is assigned a storage volume as part of its parameters. Consequently, when a storage request comes in, it is first validated whether there is enough space for it to be stored. As the full definition of the storage service coming in later versions of the M1M system, in the case the terminal is at its storage capacity, the current design merely emits a warning indicating the terminal's warehouse is full.

5.1.3 Long Haul Segments

In Figure 1, long haul segments are depicted with two ended arrows connecting terminals. Each long haul segment is defined by a mode specific distance connecting both terminals. Consequently, there could be parallel segments between two terminals representing each mode of transportation. This entails that the distance between terminal A and B could be different depending on whether the mode of transportation is a truck or a train. Additionally, the distance between two terminals is not symmetric with the distance between terminal A to B and B to A being possibly different. The notion of non-symmetric distances also applies to the distance between terminals and zones as well the distance between shipper facilities within a zone.

It is important to note that a carrier does not necessarily stop at each terminal within their route as further described in Section 5.3.2.1.

5.1.4 Physical Network Input Parameters

Once the structure of the physical network is defined, some input parameters are expected from the user before the simulation is launched.

For each segment between two nodes, the user is expected to indicate the mode specific distance in kilometers of the arc. This information is used to determine the possible travel time between two nodes.

Moreover, for each node, the user is expected to indicate the volume of items which can be processed within a single period. This value may be different for each service offered at the node. If a node does not offer a service, such as a transfer terminal not offering pick ups, the volume of items which can be picked per period defaults to 0. The user is also expected to indicate the storage capacity of a terminal. If not indicated, it is assumed the node does not offer storage services with the storage capacity set to 0.

5.2 Time

When designing a DES model, one of the main entities to consider is the time aspect. At each time steps of the simulation, it is expected that an event may occur. The time step at which an event occurs is defined as the event time. During the simulation, a possible design is to advance time without interruption until a trigger launches an event. Another alternative is performing a synchronous simulation where time advances in predefined time increments, such as one hour or 3 days, and all applicable events at the given time step are performed [23].

In the designed DES model, the time dimension used is synchronous as it is more favorable to optimization [3]. The time step by which the simulation progresses were set to be one-day. At a period t, we assume that all events scheduled to take place during said day occur at the beginning of the period. For the duration of the simulation, at each period t, the entities presented in the subsequent section occur in a specific order. Of these entities, at each period, data is generated for the Operations Planning Horizon (OPH) presented in the section below.

5.2.1 Operations Planning Horizon

We start by defining the two types of data (shipper requests and carrier offers) being considered; known and predicted data. At the current time t, known data includes requests and offers received at a previous period, received at the current period or confirmed requests and offers for future periods. Predicted data includes forecast shipper requests and carrier offers for future periods starting at period $t + 1$.

At each period t of the simulation, the IDSP performs a shipper-to-carrier assignment indicating which shipper requests are assigned to which carrier offers and the period at which the carrier departs. Additionally, during each period, the IDSP decides whether to accept or reject a shipper requests or a carrier offer. The acceptance and rejection process is discussed in more details in Section 5.5.2.1.

In the process of making the shipper-to-carrier assignment, the IDSP looks to base its decisions on the information currently known as well as what might happen in the future. Including predicted future requests and offers in the decision making process could lead to more profitable decisions by the IDSP. Consider the scenario where at period t, known shipper requests are assigned to a known carrier offer departing at period $t+\lambda$. Additionally, consider that by period $t+\lambda$, we receive more shipper requests and a carrier offer with a larger capacity having the same origin and destination as the previously retained carrier offer decided at period t. If at period t we were able to predict more requests and suitable offers will be received by time $t+\lambda$, the IDSP may have made a different potentially more profitable shipper-to-carrier assignment.

Given the above, at each period, the IDSP looks to make decisions from the current period t up to a future period T. Although both known and predicted requests and offers influence the decision making process of the IDSP, only known requests and offers are retained. Predicted requests and offers are only presents to influence the decisions taken for periods $t+1$ to T.

We denote the periods from t to T as the *Operations Planning Horizon* (OPH). Within the OPH, there are two components; the current implementation and look ahead periods. The current implementation periods, ranging from period t to $t+\delta$ (where $t+\delta < T$), include an interval of periods where the decisions are final. In other words, if the IDSP decides on a shipper-to-carrier assignment which falls within the time interval between periods t and $t+\delta$, the decision is honoured.

The look ahead periods, ranging from period $t+1$ to T, is defined as a window of periods where the decisions are not necessarily final. While decisions between the periods $t+1$ to $t+\delta$ are final, decisions between the periods $t+\delta+1$ to T are only used to influence the decisions of the current implementation periods and could be modified at future periods of the simulation.

As the simulation progress in discrete steps, the window of final decision periods and look ahead periods increase at the same step rate. Thus, at each new period of the simulation, the T OPH periods shift forward by 1 period in a rolling horizon fashion.

Moreover, the simulation model is also able to function in a myopic setting. In the myopic setting, the decisions are solely based on the information known at the current period t and does not take into consideration what may happen in future periods. Consequently, setting the look ahead periods to 0 results in the shipper-to-carrier assignment decisions being solely based on the information known at the given period.

5.2.2 Time Input Parameters

Prior to launching the simulation, the user must indicate a few input parameters regarding the time aspect.

The first time parameter specified by the user indicates the total number of periods for which the simulation runs. This parameter is essential in determining the number of periods after which the simulation model is to terminate.

Moreover, the user indicates the number of look ahead periods for which predicted data is considered. Assuming that as input the user indicates the look ahead to be θ periods, At each period t, periods $t+1$ to $t+\theta$ are considered the look ahead periods window.

Lastly, the user indicates the length of the current implementation periods for which the decisions are final. At a period t, all decisions between period t and $t+\delta$ are final, where δ is the input parameter specified by the user which indicates the number of periods for which a decision can not be modified.

5.3 Data Generator

When discussing the time aspect of the simulation, it was alluded to that decisions are made based on a set of shipper requests and carrier offers. In this section, we discuss each variable of the randomly generated shipper requests and carrier offers. Note that in the formulas below, $rand(x, y)$ refers to a random variable chosen between x and y from a uniform distribution.

Additionally, as further discussed in Section 5.4, the DES model uses optimization algorithms presented by Crainic et al. in their paper *Multi-period bin packing model and effective constructive heuristics for a corridor-based logistics capacity planning* [24]. Consequently, the range of values from which the random numbers below are generated resembles the data generating process used for the aforementioned paper [25].

5.3.1 Shipper Request

In our simulation task, a shipper request represents a company which has goods which need to be shipped from a shipper facility to a consignee location. A shipper request is characterized by geographical, physical, temporal and economic parameters.

Prior to the simulation being launched, the user indicates a list of shippers and their category. In the current version of the designed DES model, the category of the shipper has no significance. However, this attribute was added for future developments to the DES model in which the category of the shipper may be relevant. From said list of shippers, requests are generated according to the following set of attributes.

5.3.1.1 Geographical

Each generated shipper request holds an origin and destination. From the list of shipper facilities defined in the physical network, an origin location is chosen at random. Similarly,

for the destination, we choose a random shipper facility with the added rule that the origin and destination facilities can not be the same.

5.3.1.2 Physical

A shipper request may hold either one or multiple identical items which need to be shipped. When generating a request, we first randomly assign a value in the interval $[1, 8]$ indicating the number of items in a request.

Furthermore, each item within a request is defined by its volume. As items in a request are identical, the same size is applied to all items within the request. The volume of an item part of the shipper request k is determined according to the function below.

$$w(k) = rand(10, 160)$$

5.3.1.3 Temporal

A shipper request is defined by 4 pre-departure temporal attributes; the request's reception period, the earliest release indicating the period the items are available to be picked up by the carrier, the latest release indicating the latest period at which the items may be picked up by the carrier and the latest acceptance which indicates the last period by which the IDSP must accept or reject to fulfill the request. For a request k, these attributes are generated as follows.

- Request reception: $\alpha^A(k) = t$, where t is the period data is being generated for.
- Earliest release: $\alpha^R(k) = \alpha^A(k) + rand(0, 1)$
- Latest release: $\beta^R(k) = \alpha^R(k) + rand(1, 6)$
- Latest acceptance: $\beta^A(k) = min\{\alpha^A(k) + rand(0, 8), \beta^R(k)\}$

Moreover, a shipper request is defined by 4 arrival temporal attributes. These four attributes are used to define the delivery intervals indicating a delivery which arrives early, within the target interval desired by the shipper or a late delivery.

- Early delivery: $[\alpha^E(k), \beta^E(k)]$
- Target delivery: $[\beta^E(k), \alpha^L(k)]$
- Late delivery: $[\alpha^L(k), \beta^L(k)]$

We generate these four attributes according to the following functions.

- $\alpha^E(k) = \alpha^A(k) + rand(0, 1)$
- $\beta^E(k) = \alpha^E(k) + rand(0, 1)$
- $\alpha^L(k) = \beta^E(k) + rand(0, 5)$
- $\beta^L(k) = max\{\beta^R(k) + rand(0, 1), \alpha^L(k)\}$

From the described functions, we note that the latest acceptance period can not be after the latest release. Additionally, the variable $\beta^L(k)$ indicating the tail end of the late delivery interval may not be prior to the latest release.

Lastly, all generated temporal attributes cannot be assigned a period greater than the last simulation period. The period at which the simulation ends is an input parameter specified by the user when indicating the simulation's time parameters.

5.3.1.4 Cost

A shipper's request may be fulfilled through two carrier mediums. The first method is through a regular carrier bin representing carriers which can carry multiple items as part of their load. The second method is spot-market bins which represent an on demand carrier which transports items from a single request as part of their load. Additionally, for sport-market bins, the carrier's route goes directly from the origin to the destination facilities without stopping at intermediate terminals along the way.

Shipping requests sent through the spot-market incur a higher shipping cost. For each of the request's valid time period, periods between the earliest and latest release, a spot-market cost is assigned. We introduce the variable D_p which indicates the period at which the item leaves the terminal. Additionally, in Section 5.1.3, we present that each shipper facility is connected by a set of zones to terminal segments and long haul segments. We define the variable md as being the minimum distance between the origin and destination facilities. Using said information, we assign a spot-market cost based on the departure time, the minimum distance to travel and the item's size. The later an item departs in comparison to its earliest release, the higher the cost incurred. For shipping requests fulfilled through the spot-market, their cost is computed as below.

$$spotMarket_{cost} = (w(k) * 3) + \lfloor\sqrt{md}\rceil * 5 * (D_p - \alpha^R(k))$$

5.3.2 Carrier Offer

In the problem setting being addressed, a carrier offer indicates a company offering to transport goods from an origin to a destination location. Similar to shipper requests, carrier offers are characterized by their geographical, physical, temporal and cost attributes.

As input parameters, prior to the simulation being launched, the user must indicate the list of available carriers. Carriers are defined by their mode, category and serviced locations. A carrier's mode (truck, train, ship, airplane, etc) indicates the means by which their offers are to be fulfilled. Additionally, their category represents the type of vehicle offered such as a truck or train size. Lastly, the list of serviced locations indicates nodes that the carrier could deliver items to and from. Note that a carrier, represented by their carrier ID, could be listed more than once offering services through different modes, categories or serviced locations.

5.3.2.1 Geographical

In terms of determining the origin and destination locations of a carrier offer, a similar logic as a shipper request is applied. The main difference being that carrier offers may originate or end at a terminal and the list of possible nodes it may originate or end at is limited to the locations it services.

Once the origin and destination locations are assigned, from the specified physical network, we retrieve all possible paths between the two determined locations. From these sets of paths, a random route that the carrier offer will follow is chosen. Moreover, we define a leg as a path within the route connecting two consecutive nodes. For each leg in the offer's route, we assign a travel time in terms of periods indicating the time it would take to get from the origin to the target node.

The current design follows a deterministic rule to determine the minimum travel time, denoted $travelTime_{min}$, between two nodes. For offers fulfilled through trucks, we assume that it may travel up to 1200 kilometers per period equating to 12 hours of travel time per day at a speed of 100Km/h. For trains, it is assumed they could travel 2000 kilometers per period representing 20 hours of travel time per day at a speed of 100Km/h. The amount of kilometers which can be covered per period for each mode can be adjusted by the user within the data generator module as described in Appendix C.1.1.

Based on the minimum travel time between two nodes, a travel time is determined as follows.

$$travelTime = travelTime_{min} + rand(0, 1)$$

Other modes of transportation are not considered in the current version of the DES model.

However, the DES model is designed in a way where a new mode can be easily added. By adding mode specific distance information as part of the physical network definition and physical information about the mode in the data generator, the new mode will be supported by the DES model.

Note that for a given route, the carrier does not necessarily stop at each terminal between their origin and destination. It is possible that they only pass close by to a terminal, but not stop at it. For instance, consider a carrier travelling from terminal A to E within the network presented in Figure 1. A possible route between those two terminals being through terminals B, C and D, it is possible that no services are required from them at these terminals resulting in them only passing near them.

5.3.2.2 Physical

The physical attributes of a carrier offer depend on the carrier's category and the number of identical vehicles making up the offer.

For trucks, we first assign a random value in the interval $[1, 5]$ indicating the number of identical vehicles making up the offer. There are three accepted truck categories the user could indicate for a carrier as part of the input parameter; 24f, 40f and 53f. For an offer o, based on the category of the carrier, a single vehicle capacity $u(o)$ is determined as follows. These volumes are based on the actual capacity of the indicated truck types [26][27].

- 24f: A 24 feet truck having a maximum capacity of 1500 cubic feet
- 40f: A 40 feet truck having a maximum capacity of 2385 cubic feet
- 53f: A 53 feet truck having a maximum capacity of 3830 cubic feet

Moreover, for trains, as part of the input parameters specified by the user, they indicate the size and number of containers the train consists of. The capacity of a carrier offer for a train is determined as follows where C represents the number of containers on the train.

- 40f: A train of 40 feet containers having a maximum capacity of C*2385 cubic feet
- 53f: A train of 53 feet containers having a maximum capacity of C*3830 cubic feet

Lastly, each leg of the route has a random capacity assigned with its maximum value following the rules indicated above.

5.3.2.3 Temporal

A carrier offer is defined by four temporal aspects; the offer reception time, the latest time to accept the offer and the time window of periods in which the offer is available to start working. For an offer o, we compute these values as follows

- Request reception: $\alpha^A(o) = t$, where t is the period data is being generated for.
- Earliest availability: $\alpha^R(o) = \alpha^A(o) + rand(0, 2)$
- Latest availability: $\beta^R(o) = \alpha^R(o) + rand(0, 5)$
- Latest acceptance: $\beta^A(o) = min\{\alpha^A(o) + rand(0, 6), \beta^R(o)\}$

Similarly to shipper requests, all the offer's temporal attribute can hold a maximum value not greater than the last simulation period.

5.3.2.4 Cost

The fourth carrier offer specific set of attributes are the fixed and variable cost. The fixed cost represents the cost of accepting and using a carrier offer while the variable cost represents the unit cost activity per unit of volume and distance. The fixed and the variable cost, both dependent on the capacity and distance travelled, for an offer o are computed as follows.

- Fixed Cost: $f(o) = \lfloor\sqrt{u(o)} * \sqrt{tD} * 0.5\rceil$, where tD is the total distance travelled
- Variable Cost: $c(o) = \frac{tD}{u(o)*3}$

From the given formulas, we observe the fixed cost increases proportionally to the distance travelled and the vehicle's capacity. On the other hand, for a constant distance travelled, the variable cost decreases as the vehicle's capacity increases.

5.3.3 Requests to Offer Assignment Cost

With the attributes specific to both shipper requests and carrier offers defined, the last set of data to generate is the cost of assigning a shipper request k to a carrier offer o departing at a period p.

To compute the assignment cost, we first compute penalties for assigning a shipper request to a carrier which is departing at a period which will result in the request being delivered within its early or late delivery time intervals. We recall that requests delivered prior to the period $\beta^E(k)$ or after the period $\alpha^L(k)$ are respectively considered early and late deliveries.

- Early Delivery Penalty: $\psi^E(k) = max\{(\beta^E(k) - p), 0\}$
- Late Delivery Penalty: $\psi^L(k) = max\{(p - \alpha^L(k)), 0\}$

Given the item's size $w(k)$ and the offer's unit cost $c(o)$, the shipper-to-carrier assignment cost is computed as follows.

$$regularBin_{cost} = \lfloor w(k) * c(o) \rceil + 3 * (p - \alpha^R(k)) + 2 * \psi^E(k) + 3 * \psi^L(k)$$

Similar to the spot-market cost, the later an item leaves the shipper facility from the time it becomes available, the higher the assignment cost. This is represented in the function above by taking the difference from the departure time p and the earliest release $\alpha^R(k)$.

5.3.4 Data Generator Input Parameter

In addition to the list shippers and carriers the user specifies as input parameters, the user also indicates whether a request or offer may be split. As a shipper request may include multiple items, this parameter indicates whether the items in a single request may be split across multiple carriers or if they must travel as a bundle. Similarly, as a carrier offer may be composed of multiple vehicles, this parameter indicates whether all vehicles from an offer must travel together or if they may be split independently.

5.4 Optimization Models

With the shipper requests and carrier offers generated, we call on an optimization model to perform the shipper-to-carrier assignment. As the physical component of shipper request is characterized by the size of the items and carrier offers by the capacity of their bin, a possible approach to perform shipper-to-carrier assignment would be using the Bin Packing Problem (BPP). In essence, the BPP looks to assign a set of N items to the least amount of bins while making sure the total volume of items per bin does not exceed the bin's capacity [28]. Of the drawbacks of the BPP, it is assumed that all bins are of the same size and carry the same cost attribute. Consequently, the assignment decision is solely based on the item's volume.

However, as the presented shipper requests and carrier offers in our use case are defined by additional parameters, the use of a different shipper-to-carrier assignment approach is considered. Crainic et al. [24] proposed a multi-period bin packing model which addresses some of the drawbacks of the original BPP. In the proposed BPP variant, not only are the physical attributes of both requests and offers considered, the cost of assigning items from a shipper request to a bin from a carrier offer is accounted for. Furthermore, the cost

of assigning an item to a carrier's bin also takes into consideration the time at which the assignment is scheduled for, adding a temporal component to the decision process.

There are two optimization models presented in the *Multi-period bin packing model and effective constructive heuristics for corridor-based logistics capacity planning* paper [24]; a multi-period and myopic model. For each model, there are multiple heuristic algorithms which can solve the item-to-bin problem. In the designed DES model, there are four heuristic algorithms which are integrated. These four algorithms, *HM1* to *HM4*, are designed to solve a multi-period item-to-bin assignment problem while taking into consideration physical and cost attributes. This indicates that given items from a shipper request and bins from a carrier offer with temporal attributes over multiple periods, these models work towards minimizing the cost over all considered periods. These algorithms my also address the myopic version of the problem if the given shipper requests and carrier offers are for a single period.

In the designed simulation model, these optimization algorithms are treated as black box algorithms. In other words, the simulation model simply passes input data to the algorithms and expects an output without examining their inner working. As input, the optimization algorithms are given the physical, temporal and cost information for both shipper requests and carrier offers. The input data also includes both the spot-market bin and the regular bin assignment cost. As an output, the algorithms returns an item to regular bin assignment and the period at which the carrier departs a location. Additionally, for items not assigned to a regular bin, the period at which they are to be delivered through a spot-market bin is indicated.

As an input parameter to the simulation model, the user indicates which optimization algorithm between *HM1* to *HM4* is to be used throughout the simulation's run.

5.5 Simulation Workflow

With the task of each entity within the designed simulation DES model presented, we now cover the workflow of the simulation model indicating how the entities interact with one another. More specifically, as we follow a synchronous time model and a set of events occur at each period, we look at the workflow of the simulation during a single period. Figure 2 covers the periodic events as part of the simulation design.

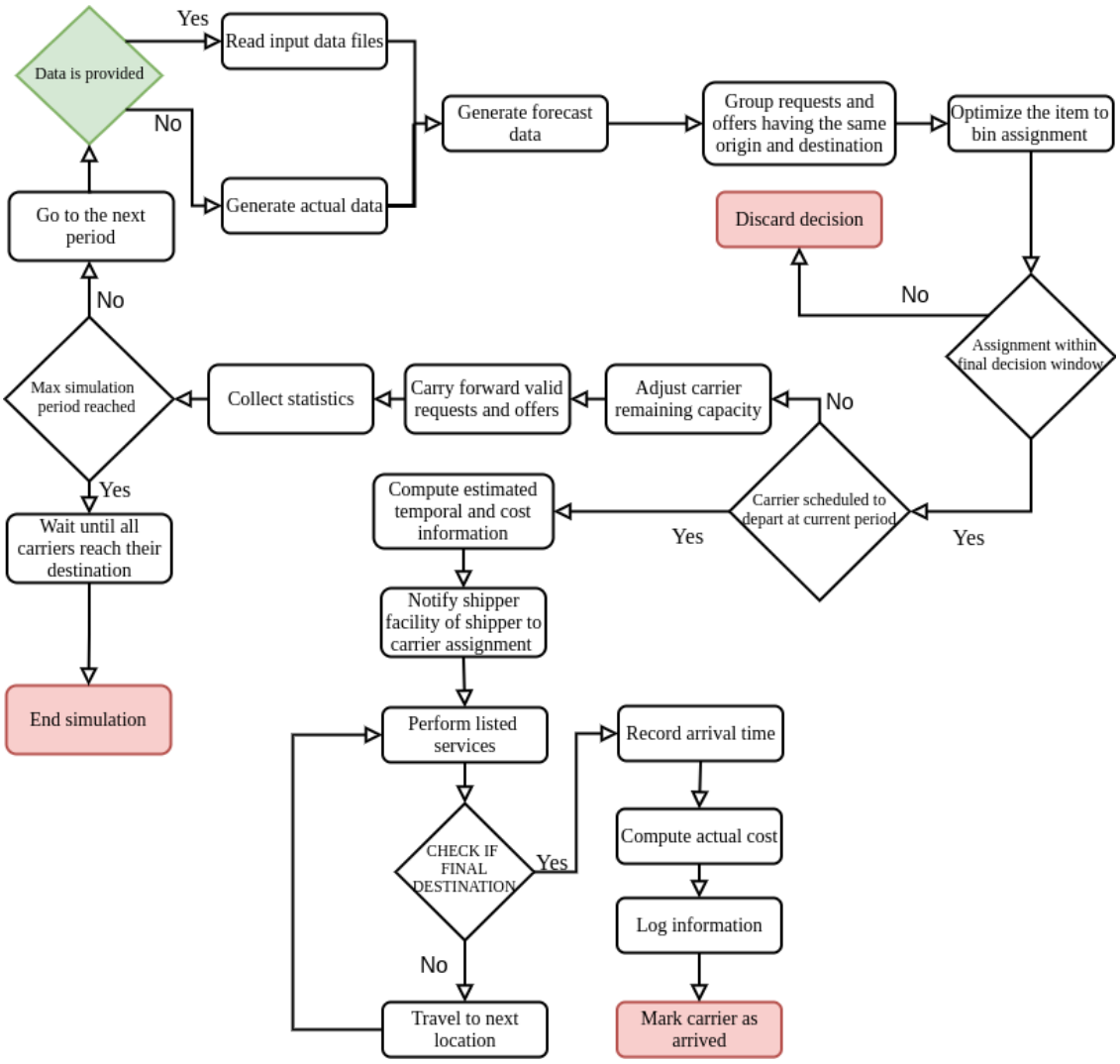


Figure 2: Simulation Workflow

5.5.1 Data Generator & Optimization

At the beginning of each period, we first look if input data representing known or actual shipper requests and carrier offers were specified by the user. If no data is specified, random data representing actual requests and offers is generated according to the rules indicated in Section 5.3.

Moreover, we generated predicted data which forecasts the requests and offers expected to arrive during the look ahead periods $t + 1$ to T. For each period within the look ahead periods interval, data is generated according to the specifications indicated in Section 5.3. Actual data and predicted data are differentiated with their requests and offer IDs. The ID of a predicted data is composed of a whole number preceded by the letter "F" while the ID of actual data is simply a whole number.

Additionally, predicted data is generated differently depending on if we are at the first period of the simulation or not. At period 1, predicted data is generated for periods 2 to T. At any other period t where $t \neq 1$, the first option is to generate predicted data for period $t+1$ to T, similarly to the process at period 1. The second option is using the predicted data from period $t-1$ and only generating new predicted data for period $T+1$. Note that for the second option when predicted data is carried forward, we first remove all predicted data for the current simulation period as decisions for the current period are solely based on known requests and offers.

Furthermore, the currently integrated optimization models are utilized to solve the item-to-bin assignment in a single-segment physical network. We define a single-segment network as a network consisting of only two terminals connected through a long-haul segment and each terminal is connected to a single zone. In this setting, as the origin and destination locations are not given as input to the optimization models, the DES model first groups all requests and offers which have matching origin and destination nodes. For each origin-destination pair, a shipper-to-carrier assignment is produced for the duration of the OPH. The optimization results are given to the shipping orchestrator.

5.5.2 Shipping Orchestrator

From the given optimization results, the shipping orchestrator performs actions ranging from accepting, rejecting and carrying forward requests and offers. Additionally, it is in charge of releasing carriers from a shipping facility and tracking their movement between their origin and destination locations. We cover each aspect in more details in the section below.

5.5.2.1 Accepting, Rejecting & Carrying Forward Requests & Offers

At a period t, when reading the shipper-to-carrier assignment results from the optimization model, the shipping orchestrator first collects shipper-to-carrier assignments scheduled to depart within the current implementation periods, meaning they are to depart between periods t and $t+\delta$.

For carriers scheduled to depart within the current implementation periods, the shipping orchestrator marks them as accepted offers. This indicates the offered carrier service will be utilized. Additionally, each shipper request assigned to one of the departing carriers, whether through regular or spot-market bins, are also marked as accepted.

Moreover, for requests not assigned to a carrier and carriers which haven't had any requests assigned to them, we verify their latest acceptance period. If the current period t corresponds to their latest acceptance period, they are marked as rejected. For shipper requests, this indicates we won't be able to fulfill their demand of shipping their items to the desired consignee location. On the carrier's end, it indicates their given offer will not be utilized.

For the remaining shipper requests and carrier offers that haven't yet been accepted and the current period is not their latest acceptance period, they are carried forward to the subsequent period $t + 1$. This simply means that at period $t + 1$, they will be included as part of the shipper requests and carrier offers for which the optimization model is to decide a shipper-to-carrier assignment.

Lastly, the shipping orchestrator splits the current implementation periods into two groups; carriers departing at period t and carrier departing between periods $t + 1$ and $t + \delta$. For carriers scheduled to depart between periods $t + 1$ and $t + \delta$, the shipping orchestrator carries forward their offer to the subsequent period. However, it takes into account that their departure period is now fixed and their capacity is reduced given the current items assigned to it. This allows for items not currently assigned or part of a new request generated in the future to be assigned within the remaining capacity of the accepted offer.

The second group, carriers scheduled to depart at period t, they go through the release process described in the next section.

5.5.2.2 Releasing Carriers

For carriers scheduled to depart at the current period t, the shipping orchestrator notifies the node at which they are to depart from about the item's assigned to it. For a departing carrier offer, the shipping orchestrator records the estimated departure and arrival times. Additionally, based on the carrier's fixed cost and the shipper-to-carrier assignment cost, the estimated cost of utilizing the offer is computed. When delays are experienced, actual departure and arrival times prove to be useful in computing the delay experienced and the information may be used to compute cost penalties.

Moreover, the services to be performed at the shipper facility are performed prior to departure. In the current design, this included picking up the items from their origin facility. As described in Section 5.1.2, this entails a time delay depending on the volume of items being loaded. Once the pick up action is complete, the carrier departs their origin destination.

5.5.2.3 Tracking Carriers

After departing their origin location, the shipping orchestrator tracks the carrier's movement up until they arrive at their destination. At each intermediary leg within their route, if the carrier has services to perform, the shipping orchestrator applies a time delay representative of the volume of items for which the service was performed.

At their final destination, a similar process is applied where in the current design, a drop off action is performed. Once the service is complete, the carrier is marked as arrived and the shipping orchestrator records the actual arrival time and computes the adjusted cost if delays were experienced.

5.5.3 Logs & Statistics

As a carrier arrives at their destination, we add their information and that of the items they transported to log files. The log files mainly include information recorded by the shipping orchestrator. This includes both estimated and actual departure time, arrival time and cost. The log file also includes whether the shipper request or carrier offer experienced any delays.

Moreover, for carrier offers specifically, we log the total volume of items transported in comparison to their available capacity. We also note the list items which were transported as part of the offer. On the shipper request's end, we log the the ID of the carrier in which it was transported.

Additionally, a log file containing all received requests and offers indicating their geographical, temporal, physical and economic parameters is generated. This file includes whether a request or offer was accepted or rejected.

To finish, at the end of each period, we also collect general statistics about each node in the physical network. These statistics include the number of departures and arrivals per period, the number of items serviced (picked up, sorted, stored, transferred and dropped off), number of departure and arrival delays and the average time spent by a carrier at the node.

5.5.4 Termination Criterion

We repeat the steps above for the number of simulation periods specified by the user as an input parameter. At the last simulation period P, the shipping orchestrator verifies if all departed carriers have reached their destination. If all carriers are at their destination, the simulation ends.

If some carriers are still within their route at period P, the simulation continues beyond the last simulation period P. However, for periods $P+1$ and above, there is no new data being generated or shipper-to-carrier assignment optimization being performed. These additional periods only serve to track the carrier's movement within the physical network, log their arrival information and collect terminal statistics. Once all carriers are marked as arrived, the simulation concludes.

5.5.5 Workflow Input Parameters

As general input parameters to the workflow, the user indicates the number of shipper requests and carrier offers to be generated at each period. Additionally, for the predicted data, the user specifies whether to generate new data for the look ahead periods or only generate new data for the additional look ahead period as discussed in Section 5.5.1

Lastly, the user indicates whether actual data is provided by them or it is to be randomly generated. If the data is provided, it must follow a specific format described in Appendix B.4.

6 Stage 3: Implementation

With the entities and workflow making up the DES model design established, the next step was to implement the simulation model. The implemented model followed the design described in section 5. In this section, we focus on implementation specific details which were not reflected as part of the design.

For the implementation, I opted to follow an iterative and incremental software design process [29]. This approach allows us to break down the project into small pieces. At each increment, the active task being implemented builds on the previous increment where eventually, the fully functioning system is built. This approach also allows to revisit previously built functionalities and reimplement them for either improvement purposes or to fit new requirements of the project without having major impacts on the components built on top of it.

In the remainder of this section, we will start by covering OMNeT++ specific concepts, the defined simulation network and the inclusion of external components used in the simulation.

6.1 OMNeT++ Concepts

The simulation environment was built on top of the OMNeT++ 5.6.2 API. Before covering the implemented classes, we start by covering concepts which are specific to OMNeT++

6.1.1 Modules

A simple module represents an entity able to perform a specific task [30]. For instance, in our use case, a simple module can be viewed as an entity which performs one of the 5 possible services at a terminal (pick up, sorting, transferring, storing, drop off). From an object oriented point (OOP) of view, a simple module is a class.

On the other hand, a compound module is a dummy module that merely englobes one or more simple modules. Compound modules in themselves can not perform any actions. Their functionalities are fully dependent on those of the simple modules they contain [31]. In our model, a compound module could be represented by a terminal holding 5 simple modules depicting the 5 possible services.

6.1.2 Messages

In OMNeT++, two modules have the capabilities of sending each other messages. Messages can be either simple words or data structures holding information populated according to the developers requirements [32].

6.1.3 Gates

In OMNeT++, a corridor between two locations is defined by gates [33]. Two modules, whether compound or simple, need to be connected through gates in order to send each other messages. OMNeT++ offers three types of gates; input gates capable of ingesting a message, output gates capable of sending a message and inout gates able to both ingest and send messages.

6.1.4 NED

In OMNeT++, the simulation's network is defined using the NED programming language [34]. The NED file is where modules, gates and how the overall components connect to one another is defined. The NED programming language loosely uses multiple OOP practices, in particular, inheritance.

6.1.5 Parameters

Each simple and compound module may be defined by some parameters. For instance, a sample parameter may be a value indicating the storage space of a module. In OMNeT++, module parameters allow for information specific to a module to be read or updated along the simulation run time [35].

6.2 Defined Simulation Network

In OMNeT++, a simulation first needs to be defined by a simulation network. The simulation network covers the design of the defined modules and how they interact with one another. Note that the simulation network is different from the physical network presented in Section 5.1. The simulation network is an OMNeT++ specific concept which describes the connection between entities of the simulation environment, while the physical network describes the network across which items are being sent.

6.2.1 Simple Modules

In Figure 3, we present the simple modules conceptualizing the simulation network. In the presented diagram, each square represents a simple module which is connected to a C++ class defining its functionalities. Additionally, each module holds gates in order to be able to communicate with other modules. Lastly, in Figure 3, the downwards arrow represents a module being an inherited child module of the module from which the arrow originates. Like in OOP, an inherited module could utilize the functionalities defined by its parent module's class. Moreover, an inherited module can also utilize the gates defined by its parent module.

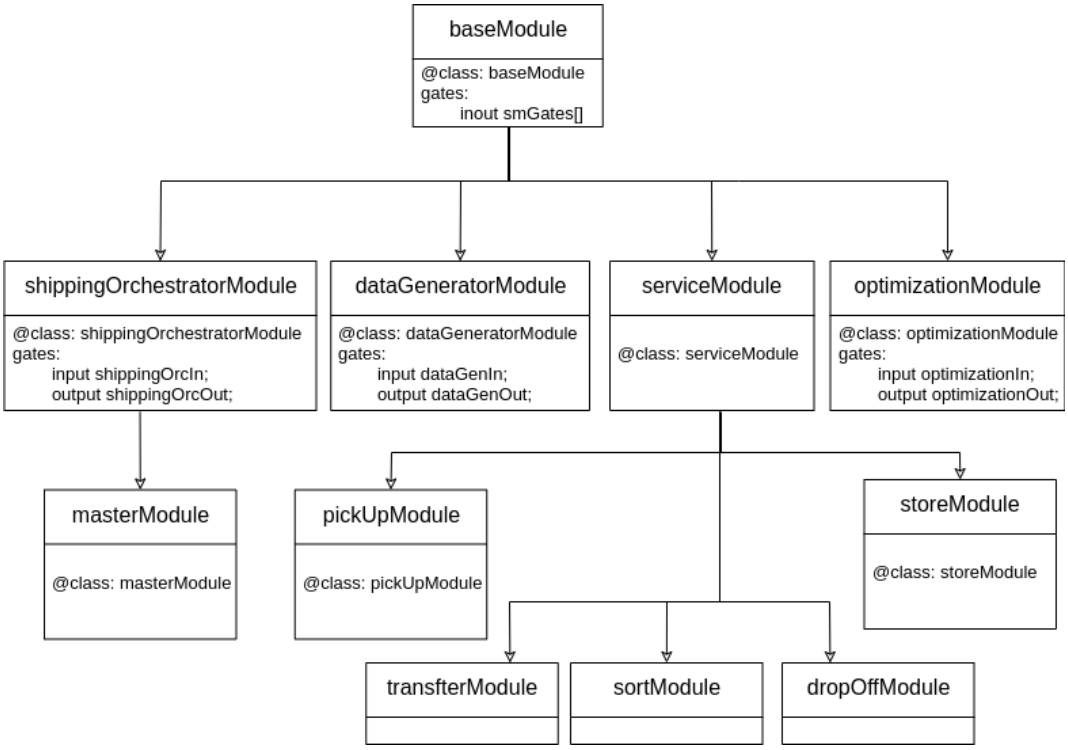


Figure 3: Simulation Network Simple Modules

The *shippingOrchestratorModule* is connected to C++ class of the same name. This class defines the functionalities described in Section 5.5.2. As the functionalities of this class were previously defined, we will omit further covering it in this section.

6.2.1.1 baseModule

The *baseModule* is the module that all other created simple modules inherit. Of its components, it includes a set of inout gates allowing the modules inherited from it to be able to communicate with one another.

Moreover, as the root parent class, when the simulation is launched, the entry point

of the simulation model is through the *baseModule* class. Consequently, this class includes a function used to validate the input parameters indicated by the user. The parameter validation is performed before the first simulation period to reduce unexpected behaviors which could be introduced from malformed input parameters. For details on the entire list of expected user parameters and their format, refer to Appendix B.3.

With the user parameter validated, the next step performed in the simulation's initialization process is finding all possible routes between nodes. To find these routes, we first find all shipper facilities, zones and terminals as part of the defined network. For each mode included in the physical network definition, we build a graph indicating which two nodes have a direct connection between them. On the built graph, a breadth-first search (BFS) algorithm is performed to find all possible routes between each pair of nodes. The BFS algorithm efficiently searches a tree or graph data structure to find a target node given a root node [36]. In our use case, the origin terminal was considered the root node and the target to be found was set to the destination node. If a route exists between the two nodes, the BFS returns all possible routes leading to the destination node from the origin.

The results from the BFS algorithm are stored in a nested map data structure. For each tuple of mode, origin and destination, the map stores a vector holding each possible route. This information becomes useful when generating random carrier offers and assigning the route the carrier will take. Additionally, with all possible routes found, we can also register the shortest path to get between two nodes. This information proves to be useful for requests shipped through the spot-market bins as they take the shortest possible route between two nodes.

Lastly, the *baseModule* class contains helper functions used by multiple inherited child modules. These include functions to search whether a value is found in a vector, create file directories and files, generate random variables from a uniform distribution, etc.

6.2.1.2 serviceModule

The *serviceModule* class defines the function to compute time delays incurred from performing a service as described in Section 5.1.2.

As transfer, sorting and drop off services only incur time delays, their modules simply utilized the functions defined in the *serviceModule* class. As these services may have more intricate functionalities in future versions, it was decided to define each service type as its own module to simplify the process of adding new functionalities.

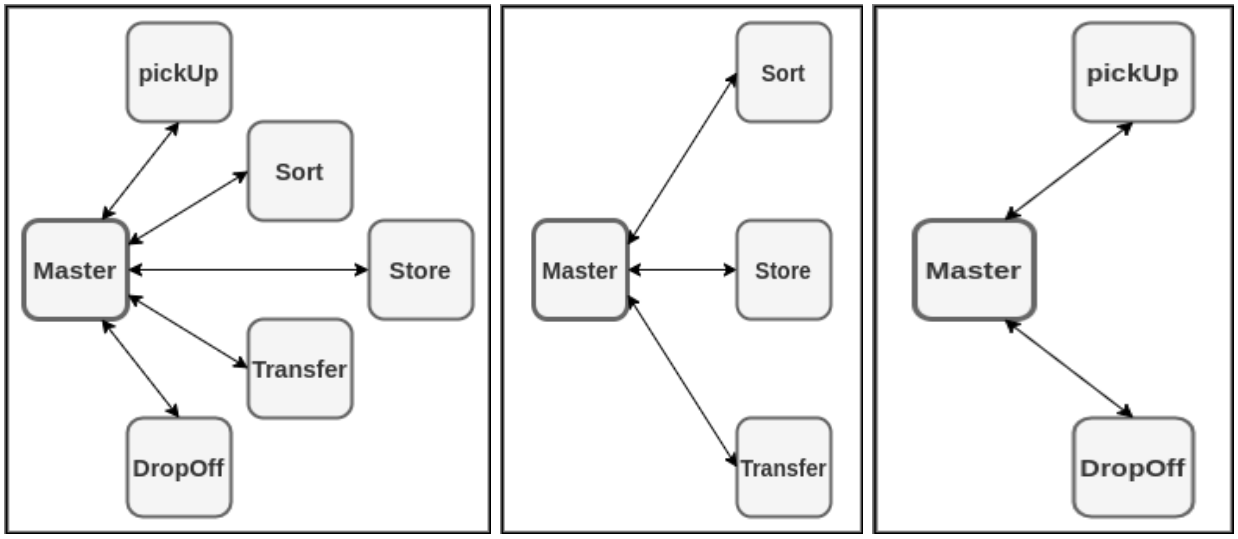
On the other hand, in addition to incurring time delays, the action of storing items includes an additional step. The *storageModule* class defines a function which adds stored items to a map holding the ID of stored items and computes the node's updated storage capacity. Additionally, we define that an item is removed from storage when it is picked up. Consequently, the *pickUpModule* class includes a function which updates the map of stored items to reflect the items which have been picked up and removed from storage. The

pickUpModule class also updates the storage capacity after items have been removed.

6.2.1.3 masterModule

The master module is considered the brain of each individual shipper facility and terminal. Let's first define how each terminal was designed to better understand the role of the master module in a node.

In our implementation, a compound module was used to represent a terminal. As modules follow the rules of inheritance defined in OOP, given we have two types of terminals in our network, at first, the transfer terminal was implemented. From an OOP point of view, the transfer terminal can be seen as the parent object since all the functionalities it can perform, the consolidation terminal can also perform. Consequently, the consolidation terminal was built as a derived child module from the transfer terminal with two services added. Figure 4 depicts a visual representation of the inner components of each terminal type. Additionally, we also define a shipper facility compound module from which items could be picked up or dropped off.



(a) Consolidation Terminal (b) Transfer Terminal (c) Shipper Facility

Figure 4: Inner Depiction Of Compound Modules

As vehicles don't necessarily stop at each node and may perform only a few services out of the possible 5, a master simple module was introduced. When a vehicle passes by a node, the master first verifies if there are any services to be performed at their node. If no services are required, the vehicle simply continues their route to the next node and considers this stop a pass by. If services are to be performed, the master routes the operation to the service at hand. After each service, the operation returns to the master to verify if any more services are required. Once there are no more services to be performed, the master routes the carrier towards the next node within their route.

6.2.2 Message

When the shipper-to-carrier assignment is complete, the shipping orchestrator sends the carrier along their route. In the OMNeT++ environment, the carrier was represented using a message. For our use case, a message in the form of a data structure holding all the information a carrier may need between their origin and destination was designed. We define a *jobDetails* message as the data structure below.

```
jobDetails
{
    int StopCount = 0;      //A counter for the number of terminals visited
    string CarrierIdentity; //An ID representing the carrier
    string CarrierType;     //Whether a regular or spot-market bin
    string CarrierMode;     //The mode of the carrier
    string VehicleType;     //The truck size or train container size
    int ScheduledDeparture; //Expected time the vehicle leaves the terminal
    int ScheduledArrival;   //Expected time of arrival
    int ActualDeparture;    //Actual time the vehicle left the terminal
    int EstimatedCost;      //Cost computed prior to departure
    int VechileFixedCost;   //The fixed cost associated to the offer
    int VehicleCapacity;    //The capacity of each individual vehicle
    int TransportedVolume;  //The aggregate volume of items transported
    string ItemList;        ///List of itemIDs in the carriers load
    string Origin;          //The origin terminal
    string Destination;     //The destination terminal
    Route RouteDetails[];   //An array of a holding the route details
    int DepartureDelays;    //Number of delayed periods on departure
    int ArrivalDelays;      //Number of delayed periods on arrival
}
```

Along the route, information which wasn't available when the message was created are added. For instance, the actual departure or delays are recorded by the master of the terminal when a carrier arrives. Once the carrier arrives at their final destination, the shipping orchestrator uses the information found in the message data structure to generate the logs described in Section 5.5.3.

Additionally, the route to be followed by a carrier, defined as *RouteDetails[]* in the *jobDetails* message, had its own data structure implemented. To determine the route, an array was built with each index of the array representing the terminal to stop at and the services to be performed. At each index, a list of item IDs for which the services are to be performed as well as the volume of each item were included. As the time a service takes depends on the volume of an item, having this information as part of the route planning allows to know the time a carrier will take to have the services complete. Lastly, a variable

is included to keep track of when a carrier arrived at a node and the time it took to perform all services. The design of the route data structure is as below.

```
Route
{
    string NodeAdress;   //The name of the target terminal to visit
    int travelTime;      //Time it will take to reach the next terminal
    deque Actions;       //Vector of services to perform at target terminal
                         //If Actions is empty, a pass by is performed
    vector ItemIDVect;   //Vector of ID's on which to perform the services
    vector ItemSize;     //Vector of the size of items in ItemIDVect
    int ArrivalPeriod    //Arrival period at the current node
    int ServiceTime      //Time to perform scheduled services at terminal
}
```

6.2.3 Routing

Furthermore, the *baseModule* holds a function in charge of routing a message from one node to the next. Each set of nodes having a route between them need to have gates to access these routes. As a first design, I created an inout gate specific to the connection between each node. Consequently, two nodes would be connected with gates following the naming convention gate_Origin_Destination. For instance, a connection between nodes A and B would be represented by the inout gate gate_A_B. Each node would read the route details found in the message and send the message to the gate name which is a compound of their name and the next target node.

The above implementation proved to be problematic from a scalability point of view. For instance, assume a hyper-corridor with more than 26 nodes. Not only would it be a tedious task to create a gate specific to each possible connection, the naming convention of gates may start to be confusing.

OMNeT++ allows for arrays of gates to be created. In other words, only one inout gate array defining the connection between each node would need to be created. For each node, a subset of the gate array indicating the name of connected neighbouring nodes is generated by OMNeT++. Thus, when sending a message from one node to another, the task becomes sending the message through the index in the array holding the target node's name. To perform the above, the below simple algorithm was implemented.

```
Input: neighbour_nodes_name_array, target_nodes_name
Output: index of the gate connecting both nodes in the gates array

N = neighbour_nodes_name_array.length()
FOR i=0,...,N do
      If neighbour_nodes_name_array[i] == target_nodes_name
        return i
      ENDIF
ENDFOR
return -1          //This means an error happened
```

With the logic above, the hyper-corridor would be able to route a message through the predefined route whether the network contains 2 nodes or a high number of nodes which would be complex to manage with a gate specific connection between each pair of two nodes.

6.2.4 Time

In Section 5.2, we introduced the used time definition of synchronous periods of one-day. On the implementation end, time progresses in steps of one-hour. With time steps of one-hour, this allowed for periodic events taking place within a period's duration to be scheduled at specific hours to assure they follow a desired order. For instance, new data is generated at the first hour of the period, optimization at the second hour of the period and so on.

That being said, for aspects depending on time (departures, arrivals, travel time, delays, logs, etc), the output doesn't differentiate between when an event occurred within a period. In other words, although operations within a period of one-day happen at specific hours, the output only reflects at which daily period they occurred.

6.3 External Components

Moreover, the *dataGeneratorModule* and *optimizationModule* classes call on external components to complete their tasks described respectively in Sections 5.3 and 5.4.

The decision to have these modules depending on external components was two fold. To begin with, as the project of building the M1M system has multiple contributors, there are team members working on the optimization and prediction components. Having these components not highly coupled with the OMNeT++ environment allows for their development to progress independently of the simulation environment.

Secondly, for the optimization algorithms presented in Section 5.4, they were already implemented by the authors of the *Multi-period bin packing model and effective constructive heuristics for corridor-based logistics capacity planning* paper [24]. Given access to the source code, rather than re-implementing them in OMNeT++, they were adjusted to function within the OMNeT++ simulation environment.

6.3.1 External Functions

As the script defining the data generator rules discussed in Section 5.3 was developed with the intention to be used in OMNeT++, it was written as a stand alone script. To utilize it in OMNeT++, the source code of the script and its header file had to be placed in the directory where OMNeT++ was installed. A header file simply declares the name of the functions implemented in the script in order to inform the compiler of their existence and location [37].

On the other hand, to integrate the optimization algorithms into the OMNeT++ environment, a different approach had to be used.

6.3.2 External Objects

In the source code made available to me, each of the four algorithms were written to be launched as a stand alone script. Across these four algorithms, a common functionality was the IO operation which reads input data files following a specific format and outputs the optimization results into newly generated files. As all four optimization algorithms were integrated in the OMNeT++ simulation environment, to reduce code redundancy, the IO operations for the four algorithms were merged into a centralized script. The remaining methodology of the algorithms were left unchanged.

Additionally, each optimization algorithm had code fragments across multiple files. To execute a single algorithm, the code fragments related to it had to be linked when being compiled [38]. However, OMNeT++ does not trivially allow for multiple external source code files to be linked when the simulation is launched.

That being said, OMNeT++ allows a user to link object files which can be used within the simulation. Object files are source code files compiled into a low level language readable by a computer. Consequently, each optimization algorithm was compiled into its own object file readable by the OMNeT++ environment [39]. Of the disadvantage of object files, their content is not readable for humans. Thus, a user must refer to the source code of the implemented optimization algorithms to modify their content. Once modified, the object files have to be generated once again to reflect the user's changes.

7 Stage 4: Testing & Performance Analysis

To assure a software functions as expected, it is important to perform software testing to discover any failures or design defects [40]. In the development process of the design DES model, there were two main types of testing performed; unit and system testing.

7.1 Unit Testing

Unit testing is oriented towards assuring that each implemented component functions as expected. This type of testing is conducted as soon as a function is implemented to uncover any bugs prior to moving on the next component. This process simplifies the debugging process in the long run as it avoids multiple components being built on top of a faulty one [41].

To perform the unit test, a white-box testing approach was utilized [42]. In white-box testing, the internal structure of the unit being tested is known. To test the proper functioning of the unit, the tester injects a set of inputs to the unit and tracks its progress through the unit assuring the expected output is produced. Testing all the different types of expected inputs the unit may receive allows to uncover any design flaws.

Out of the performed unit test, a sample would be the tests conducted for the routing approach described in Section 6.2.3. The following input cases were tested.

1. A route which only has an origin and destination without intermediary stops
2. A route which has multiple intermediary stops
3. A route which revisits the same node multiple times
4. A route which includes an undefined node
5. A route which attempts to visit a node it is not connected to

For each of the indicated inputs, the expected output at each step within the function was noted prior to performing the test. The first three inputs were expected to finish successfully while the last two were expected to return an index value of -1 indicating a faulty input. The function was adjusted until the test for each possible input type produced the expected behaviour.

7.2 System Testing

System testing is performed once all functions of the system are integrated. Rather than focusing on each implemented function, system testing looks to evaluate the proper functioning

of the designed system from end-to-end [41].

7.2.1 Tested System Configuration

Once the system was complete and each component tested independently, the system as a whole was tested. The current set of optimization models available are only designed to perform the shipper-to-carrier assignment for a simple physical network of two terminals with one shipper facility connected to each. Consequently the system testing was conducted on the physical network presented in Figure 5.

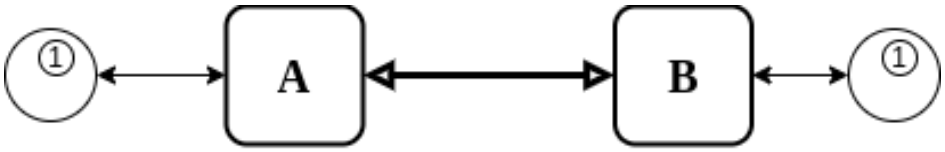


Figure 5: Three Segment Network

As the implemented DES model was designed to be able to accommodate more diverse physical networks, the fact that the system testing was only performed on the presented network resulted in more emphasis being put on the unit testing. For instance, when unit testing features such as the routing logic presented in Section 6.2.3, they were performed on a network similar to that presented in Figure 1 in Section 5.1. This reduces the likelihood of bugs related to these unit surfacing when optimization models able to accommodate more complex networks are introduced.

For the three segment network in Figure 5, it was tested against possible combinations of input parameters presented in Appendix B.3. This includes the different optimization models, varying current implementation and look ahead period windows, networks with different modes and distances between nodes, etc. The only parameters for which not all cases were tested are the parameters which indicate if a shipper request or carrier offer may be split. In other words, this parameter indicates whether items within one shipper request may be delivered through multiple carriers. Similarly, it indicates whether vehicles from a single carrier offer may be split and depart at different times. As the current optimization models do not support item and carrier splitting, the system testing was performed with this parameter staying constant.

7.2.2 Performance Testing

Throughout the system testing, performance metrics were tracked to verify if some of the implemented functions could be improved. More specifically, it was observed that the run time between two periods was significant. Additionally, after launching a few runs, my system emitted warnings indicating the computer is running low on disk space. Performing

some debugging, the main cause behind both issues was related to the process of generating the request-to-offer assignment cost described in Section 5.3.3.

When first implemented, the cost of assigning a request to an offer at a specific period was recorded in a file. The file consists of an $Nx4$ matrix where the first column is the requestID, followed by the offerID, the period and the assignment cost. That being said, in the case where we are generating 100 requests and 15 offers per period for a simulation lasting 10 periods, at period 1, N would be $15,000$ or $100 * 15 * 10$. At the second period, there would be $13,500$ new entries given the 100 new requests, 15 new offers and 9 remaining periods. Additionally, for requests and offers carried forward from period 1, they would also have an assignment cost when respectively paired to new offers and requests generated at period 2. Consequently, the number of request-to-offer-to-period tuples increase rapidly as the simulation progresses.

Not only does the above implementation take time to traverse the file to find a specific cost, the file size can become significant when the simulation concludes. Consequently, the implementation was adapted where the totality of the assigned cost data is not stored in a file. Instead, when passing input data to the optimization model, a temporary file only containing the assignment cost for requests and offers being currently considered by the optimization model is generated. The temporary file only contains data for the number of periods equating to the OPH.

To exemplify the difference in performance, the simulation was launched using both implementations for the same set of parameters. These parameters were as follows:

- Requests per period: 500
- Offers per period: 25
- Simulation periods: 20
- Look Ahead Periods: 7
- Current Implementation Periods: 3
- Optimization Model: HM4

The computational platform on which the tests were performed was a computer with 8 cores, 32GB of memory, a 1.60GHz processor and running the operating system Ubuntu 20.04. The obtained results are presented in the Figure 6.

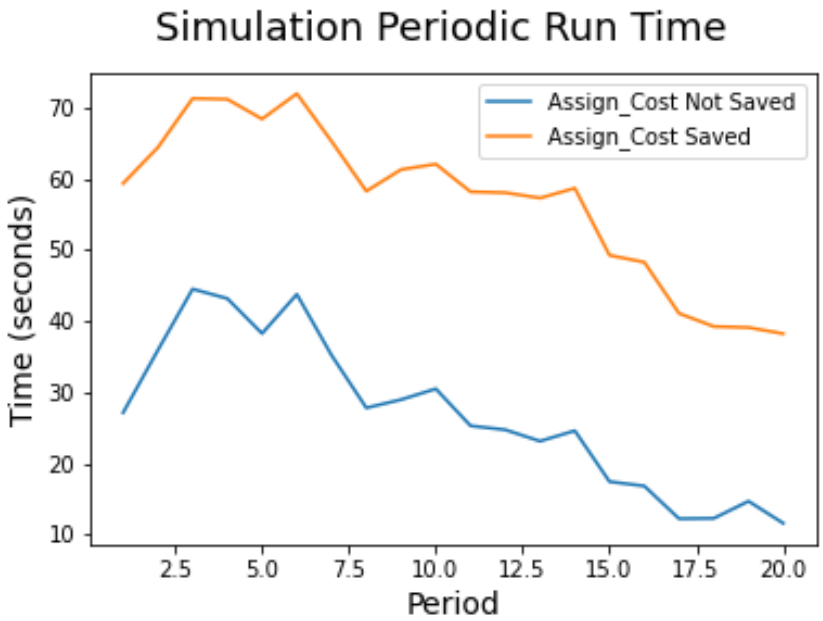


Figure 6: System Run Time

As displayed in Figure 6, the updated implementation which doesn't store the totality of the assigned costs data in a file reduced the run time by around 52.83%. Additionally, in the original implementation, the resulting file holding the assigned cost data consumed 48.3 megabytes of disk storage. Although this value is not large, for a simulation configuration with more requests, offers or periods, the value may increase steeply.

8 Gained & Improved Knowledge

Throughout my six months internship with the CIRRELT, I had the opportunity to build on both my soft and technical skills. From improving my communication skills by working on a user guide detailing the built system to gaining analytical aptitudes by analyzing various simulation software, I got to cover a wide array of topics.

Additionally, in the process of training two master's students on how to use the simulation model, I gained a lot of practice in presentation skills and breaking down a complex subject into clear smaller components. This experience increased my appreciation towards the ability of clearly presenting a complex subject to individuals unfamiliar with the topic at hand.

That being said, the two main areas where I believe I learned the most are the topic of simulation and development in the C/C++ environment.

8.1 Simulation

Prior to commencing my master's degree, my knowledge of simulations was limited to a broad definition of their purpose being to imitate a process. Throughout my coursework, I gained interest towards the field of operations research. When learning about topics such as queuing theory, I gained some additional sense towards how simulations may be utilized, but my knowledge was still quite limited.

Since starting my internship with the CIRRELT, my knowledge towards discrete event simulations increased substantially. To begin with, while covering some literature to familiarize myself with the theory of DES, I had the chance to learn about various use cases in which DES has been applied. These range from simple queuing models for simulating elevator traffic to models replicating the security system at airports. The amount of details in those problems gave me a sense of how to approach designing a DES model.

Going through a design phase, I got to apply the concepts I had covered through the read literature. Although breaking down the complex system into smaller components proved to be challenging, having hands-on experience with designing a model greatly helped putting into perspective the components of DES and their importance within the overall system.

A challenge faced while first working on the design was to fully grasp the random data generator component of a DES model. Coming from a background of mainly machine learning and data science, my expectation was that the data used in a DES model would be collected from real life sources. Working on the design and reading on this aspect, I came to understand that DES models are intended to be used with either accurate random data, actual data or both. Additionally, it made me better understand how useful generating accurate random data may be given the complexity collecting real life data may entail.

8.2 Development in the C/C++ Environment

In my previous educational and work experiences, the bulk of my programming knowledge was centered around the Python and Java programming languages. My knowledge of C/C++ was limited to a single introductory class taken during my undergraduate studies.

Throughout the development phase of the internship, I got to apply and better understand C/C++ concepts which are not found in Python and Java. These include header files, pointers, references, memory management, etc. As my understanding of these concepts and general C/C++ best practices improved, throughout the development phase, I was able to revisit some implemented functions and simplify them using my newly acquired knowledge. For instance, a common practice in Python is returning multiple variables from a single function. In C/C++, there is no native support to return multiple variables from a function. Consequently, I was initially implementing functions which return tuples holding multiple variables to work around this limitation. Through some additional research, I found the

most efficient solution to return multiple variables from a function is C/C++ is to instead use variable references. This approach allows for any amount of variables to be given as input to a function and have their value updated according to a logic implemented within the function.

Moreover, when integrating the external components described in Section 6.3, I learned a lot about turning source code into a shareable object and library files. This aspect did not only prove to be useful within this use case, but could be applied to future projects I work on in other programming languages supporting libraries, such as Python and Java.

9 Conclusion

In conclusion, during my six months internship with the CIRRELT, I worked towards building a DES model based on an M1M system which makes automated decisions on which shipper request to assign to a set of available carrier offers. The goal behind this simulation model is to imitate the designed system under different variables and optimization models to analyze its performance under the given parameters.

In the process of building this DES model, I was able to cover multiple tasks such as performing software analysis, designing a DES model, implementing the designed DES model, performing software testing and writing a user guide for users and future contributors. The challenges met during this internship have helped me improve my computer programming skills and gain a vast amount of knowledge in a wide array subjects, especially those related to the field of DES.

For future projects, I would be interested in integrating my acquired knowledge of DES with my other field of interest, machine learning and data science. In the fields of machine learning and data science, the base steps to perform is data collection to build models or perform analysis on top of. While working on building the DES model for the M1M system, I did some research and found some work has been made where simulation generated data is used towards building machine learning models. Consequently, it would interest me to further explore how data generated from a simulation could be integrated with collected data to improve the performance of a machine learning model.

References

[1] MultiMedia LLC. Notre mission. Available at https://www.cirrelt.ca/?Page=MISSION (2021-06-17), 2020.

[2] Barrett JS, Jayaraman B, Patel D, and Skolnik J. Discrete event simulation. Available at https://www.med.upenn.edu/kmas/DES.htm (2021-06-17), 2008.

[3] Teodor Gabriel Crainic, Michel Gendreau, and Walter Rei. Optimization for operational planning of m1m systems on hyper-corridors. Unpublished Manuscript, 2021.

[4] What is open source? Available at https://opensource.com/resources/what-open-source (2021-06-17).

[5] University of Hamburg Department of Computer Science. Desmo-j - a framework for discrete-event modelling and simulation. Available at http://desmoj.sourceforge.net/home.html (2021-06-17), 2014.

[6] Michael J Allen. Facsimile simulation library. Available at http://facsim.org/ (2021-06-17), 2020.

[7] JaamSim Development Team. Jaamsim. Available at https://jaamsim.com/ (2021-06-17), 2021.

[8] Jsimplesim - a simple framework for multi-agent simulation in java. Available at https://jsimplesim.org/ (2021-06-17), 2021.

[9] Manpy. Available at https://www.manpy-simulation.org/ (2021-06-17), 2016.

[10] ns-3 network simulator. Available at https://www.nsnam.org/ (2021-06-17), 2021.

[11] Omnet++ discrete event simulator. Available at https://omnetpp.org/ (2021-06-17), 2021.

[12] Opensimply. Available at https://opensimply.org/ (2021-06-17), 2020.

[13] Simjulia. Available at https://simjuliajl.readthedocs.io/en/stable/welcome.html (2021-06-17), 2019.

[14] Team SimPy. Simpy discrete event simulation for python. Available at https://simpy.readthedocs.io/en/latest/ (2021-06-17), 2020.

[15] Pierre L'Ecuyer. Ssj: Stochastic simulation in java. Available at http://simul.iro.umontreal.ca/ssj/ (2021-06-17), 2018.

[16] Desktop operating system market share worldwide. Available at https://gs.statcounter.com/os-market-share/desktop/worldwide/ (2021-06-17), 2021.

[17] TIOBE The Software Quality Company. Tiobe index for june 2021. Available at https://tiobe.com/tiobe-index/ (2021-06-17), 2021.

[18] Jaamsim users discussion group. Available at https://groups.google.com/g/jaamsim-users (2021-06-17), 2021.

[19] ns-3-users. Available at https://groups.google.com/g/ns-3-users (2021-06-17), 2021.

[20] Omnet++ users. Available at https://groups.google.com/g/omnetpp (2021-06-17), 2021.

[21] Omnest - omnet++ comparison. Available at https://omnest.com/comparison.php (2021-06-18), 2021.

[22] Omnet++ 6.0 preview 11. Available at https://github.com/omnetpp/omnetpp/releases/tag/omnetpp-6.0pre11 (2021-06-18), 2021.

[23] Roger B. Dannenberg. Discrete event simulation. Available at https://www.cs.cmu.edu/~music/cmsip/slides/02-discrete-events.pptx.pdf (2021-06-21), 2019.

[24] Teodor Gabriel Crainic, Franklin Djeumou Fomeni, and Walter Rei. Multi-period bin packing model and effective constructive heuristics for corridor-based logistics capacity planning. *Computers & Operations Research*, 132:105308, 2021.

[25] Teodor Gabriel Crainic, Franklin Djeumou Fomeni, and Walter Rei. Data set related to the paper: "multi-period bin packing model and effective constructive heuristics for a corridor-based logistics capacity planning". Available at https://doi.org/10.17632/z3rv8dhm37.2 (2021-07-13), 2020.

[26] Truck comparison guide. Available at https://www.enterprisetrucks.com/truckrental/en_US/vehicles/commercial-truck-comparison-guide.html (2021-06-22), 2021.

[27] A guide to truck trailers. Available at https://www.allencounty.us/homeland/images/lepc/docs/TruckTrailerGuide.pdf (2021-06-22).

[28] Martello Silvano and Paolo Toth. *Knapsack problems : algorithms and computer implementations*. J. Wiley & Sons, Chichester ; New York, 1990.

[29] Agile development: Iterative and incremental. Available at https://www.visual-paradigm.com/scrum/agile-development-iterative-and-incremental (2021-06-25), 2021.

[30] Omnet++ simulation manual - omnet++ version 5.6.1 - simple module. Available at https://doc.omnetpp.org/omnetpp/manual/#sec:ned-lang:simple-modules (2021-06-25), 2020.

[31] Omnet++ simulation manual - omnet++ version 5.6.1 - compount module. Available at https://doc.omnetpp.org/omnetpp/manual/#sec:ned-lang:compound-modules (2021-06-25), 2020.

[32] Omnet++ simulation manual - omnet++ version 5.6.1 - the cmessage class. Available at https://doc.omnetpp.org/omnetpp/manual/#sec:msgs:cmessage (2021-06-25), 2020.

[33] Omnet++ simulation manual - omnet++ version 5.6.1 - gates. Available at https://doc.omnetpp.org/omnetpp/manual/#sec:ned-lang:gates (2021-06-25), 2020.

[34] Omnet++ simulation manual - omnet++ version 5.6.1 - the ned language. Available at https://doc.omnetpp.org/omnetpp/manual/#cha:ned-lang (2021-06-25), 2020.

[35] Omnet++ simulation manual - omnet++ version 5.6.1 - parameters. Available at https://doc.omnetpp.org/omnetpp/manual/#sec:ned-lang:parameters (2021-06-25), 2020.

[36] Robert Sedgewick and Kevin Wayne. *Algorithms*. Addison-Wesley, Upper Saddle River, NJ, 4 edition, 2011.

[37] Colin Robertson, Kent Sharkey, Andrew Rutherfurd, Logan Saether, and Matthew Sebolt. Header files (c++). Available at https://docs.microsoft.com/en-us/cpp/cpp/header-files-cpp?view=msvc-160 (2021-06-26), 2019.

[38] The linker. Available at https://riptutorial.com/c/example/4360/the-linker (2021-06-25).

[39] Gnu - creating object files. Available at https://www.gnu.org/software/libtool/manual/html_node/Creating-object-files.html (2021-06-25).

[40] Cem Kaner, Jack L. Falk, and Hung Quoc Nguyen. *Testing Computer Software*. John Wiley & Sons, 605 Third Ave. New York, NY, United States, 2 edition, 1999.

[41] Software testing methodologies. Available at https://smartbear.com/learn/automated-testing/software-testing-methodologies/ (2021-06-26).

[42] White box testing: A complete guide with techniques, examples, & tools. Available at https://www.softwaretestinghelp.com/white-box-testing-techniques-with-example/ (2021-06-26), 2021.

Appendix A Introduction

This documentation is divided into two major sections; a user and developer guide. The user guide section covers the installation process, setting up the environment and running the simulation model as is. The simulation's workflow is also covered in Section B. Note that this document is written with the assumption that the reader is familiar with the *Optimization for Operational Planning of M1M Systems on Hyper-Corridors* document [3].

For adding or modifying some functionalities or logic in the simulation model, refer to the developer section. Note that information in the developer section assumes OMNeT++ was installed and the environment was configured as per Sections B.1 and B.2.

Appendix B User Guide

B.1 Installation

The simulation model was developed and tested using OMNeT++ version 5.6.2. It is recommended to use version 5.6.2 of OMNeT++ as some features in later versions are not backward compatible.

To install OMNeT++, on their website's *download page* [1], choose your operating system and click download.

NOTE: As per the OMNeT++ documentation, if you would like to run parallel simulations, it is recommended to use Linux, macOS or other unix-like platforms if available.

Once downloaded, extract the compressed folder in a directory of your choice where you would like to have OMNeT++ installed. Navigate to the folder *../omnetpp-5.6.2/doc/* where there is a file named *InstallGuide.pdf* containing installation information specific for each operating system. Install OMNeT++ according to said document.

[1] OMNeT++ Download Page: `https://omnetpp.org/download/`

B.2 Importing Project & Setting Up Environment

B.2.1 Installing External Packages

Once OMNeT++ is installed, external C/C++ packages which are not natively included as part OMNeT++ must be added. Currently, the only external package required is *rapidjson*.

1. In the zip folder *M1M-V1*, locate the folder named *rapidjson*.
2. In a different window, navigate to the directory where you installed OMNeT++ and open the *include* folder.
3. Copy and paste the *rapidjson* folder into the *include* folder. By default, the file path should resemble the following: *../omnetpp-5.6.2/include*

B.2.2 Copying the Project To OMNeT++

To launch the simulation, OMNeT++ needs to have access to the source code of the simulation's implementation. The code is to be added through the following steps

1. In the zip folder *M1M-V1*, locate the folder named *M1M-V1*.
2. In a different window, navigate to the directory where you installed OMNeT++ and create a folder named *workspace*.
3. Copy and paste the *M1M-V1* folder into the *workspace* folder.

B.2.3 Launching OMNeT++

The next step involves launching OMNeT++ following the steps below.

NOTE: The first time OMNeT++ is launched on your machine, it may take several minutes to install some projects included by default with OMNeT++.

---

1. Open the terminal of your operating system. For Windows users, it is recommended to use the *mingwenv* terminal located in the *omnetpp-5.6.2* folder.

2. Through the terminal, navigate to the path of the directory *../omnetpp-5.6.2*
3. Run the command *omnetpp*
4. A pop up window will ask you to specify the directory including the project. Select the *workspace* folder where the *M1M-V1* folder was copied.

NOTE: It is also possible to create a launch shortcut from your desktop. In the OMNeT++ installation guide, for each operating system, a section named *"Starting the IDE"* outlines the necessary commands to create the shortcut.

B.2.4 Importing The Project

Once OMNeT++ is launched, the next step is to import the folder containing the simulation project to the OMNeT++ interface.

1. Under the *"File"* tab in the menu bar, click *"Open Projects From File System..."*
2. Next to *"Import source"*, click the *"Directory"* button
3. Navigate to the *workspace* folder created in step B.2.2.
4. Select the folder *"M1M-V1"* and click open
5. Click the *"Finish"* button

B.2.5 Specifying The Path Of External Objects

As part of the simulation, optimization models built independently of the simulation model are used. As these models were developed outside of OMNeT++, the environment needs to be aware of the location of these models' files.

1. On the *"Project Explorer"* tab found in the left hand side in the OMNeT++ interface, navigate to the folder *M1M-V1/src*
2. Open the *makefrag* file

3. On each line of the file, there is a file specified of the format *PATH/omnetpp-5.6.2/workspace/M1M-V1/src/externalObjects/...* Replace the *PATH* keyword with the absolute path where OMNeT++ is installed.
    - For Windows, a sample absolute path resembles the format *C:/../omnetpp-5.6.2/workspace/M1M-V1/..*
    - For unix like systems, a sample absolute path resembles the format */home/USERNAME/omnetpp-5.6.2/workspace/M1M-V1/..*
4. For each line starting with the keyword *EXTRA_OBJS*, the path contains the following sequence of folders *'../M1M-V1/src/externalObjects/OS/..'*. Replace the *OS* keyword in the file path by either *windows* or *other* depending on the operating system used.
5. In the OMNeT++ interface, right click the folder *M1M-V1* folder and click *Clean Project*
6. After cleaning the project, right click on the same folder and click *Build Project*

The last two mentioned steps are essential whenever making changes to the files found in the *externalObjects* directory, including changing the directory's structure. This assures that when launched, the system is not using code artifacts from prior versions.

NOTE: If you are new to OMNeT++, it is beneficial to go through their *Tic Toc tutorial* [2]. It covers the basic concepts of OMNeT++ and familiarizes the user with the user interface.

B.3 Parameters

Before running the simulation, the last step involves setting up the parameters of the simulated environment. The parameter files can be found in the *M1M-V1/src/params* folder. Parameters are split into five groups; network, zones and terminals, carriers, shippers and general.

B.3.1 Network Parameters

OMNeT++ requires that the network be defined in a NED file. For the *M1M-V1* project, the defined network can be found in the *src/M1M.ned* file. It can be observed that within the NED file, the mode and distance of the arcs between terminals and zones is not defined. The NED file simply indicates a connection between both locations is possible.

[2] OMNeT++ Tic Toc tutorial: `https://docs.omnetpp.org/tutorials/tictoc/`

In this section, we define a node as location in the network. It could either be a shipper facility in a zone, a consolidation terminal or a transfer terminal. Nodes follow a specific naming convention to be able to distinguish between their types. The name definition rule is as follows.

- Consolidation terminals are named according to the format "CT_ID". For instance, the consolidation terminal "A" would be denoted as "CT_A".
- Transfer terminals are named according to the format "TT_ID". A transfer terminal "B" would be denoted as "TT_B".
- Shipper facilities within zones have the name format "Z_connectedTerminal_zoneNumber_facilityNumber". Assume a node is named "Z_A_2_3". This would indicate it is the third shipper/consignee facility in the second zone connected to terminal A.

Moreover, to define if a path actually exists between two nodes and the distance between them, the user must specify this information in the *params/network.csv* file. Each line of the CSV file indicates a mode specific distance between two locations. Note that on each line of the CSV, the indicated origin and destination must have a direct connection (no intermediary node between them) established in the NED file. Additionally, the distance between two nodes is not symmetric. Thus, the distance between nodes A and B may be different than the distance between B and A.

Parameter	Description
mode	*String* Variable indicates the mode of the vehicle. The current implementation accepts the "vehicle" and "train" modes.
origin	*String* The name of the origin location. Value must match the name indicated in the NED file.
destination	*String* The name of the destination location. Value must match the name indicated in the NED file.
distance	*Whole Number* The distance between both nodes in kilometers

Table 2: Network Parameters

B.3.2 Zone & Terminal Parameters

For each zone and terminal in the network, we assign them a value indicating the speed at which they process items per period (1 day). We define processing as the actions of either loading, unloading, transferring, sorting or storing items. For each action, the user indicates the total volume of items which can be processed for the node in question within a single period. If a service is not available at node, for instance loading or unloading services at a transfer terminal, the indicated value defaults to 0.

If a node has a larger volume of items to process per period than they can accommodate, this causes delays. For instance, if a carrier is transporting items totaling a volume of 15,000 cubic feet and the terminal loads items at a speed of 5,000 cubic feet per period, it will take 3 periods for the loading operation to be completed. Thus, the values assigned for the volume which can be processed per period directly correlates to possible delays experienced by a carrier.

Additionally, for consolidation terminals only, there is also a storage capacity metric. This value indicates the volume of items the terminal can hold in storage. Note that for transfer terminals and zones, storage capacity is automatically set to zero as these locations don't offer storage services. Below is the expected format of each line in the *params/zoneAndTerminal.csv* parameter file.

Parameter	Description
node name	*String* The name of the node for which the parameters apply. The node name must match that specified in the NED file.
Loading Volume Per Period	*Whole Number* Speed metric indicating the volume of items which can be loaded into a vehicle per period.
Drop Off Volume Per Period	*Whole Number* Speed metric indicating the volume of items which can be dropped off from a vehicle per period.
Storing Volume Per Period	*Whole Number* Speed metric indicating the volume of items which can be moved to storage per period.
Sorting Volume Per Period	*Whole Number* Speed metric indicating the volume of items which can be sorted per period.
Cross Dock Volume Per Period	*Whole Number* Speed metric indicating the volume of items which can be cross docked (transferred) from one vehicle to another per period.
storage capacity	*Whole Number* Variables indicates the storage space at a consolidation terminal. Value must be greater or equal to 0.

Table 3: Zone & Terminal Parameters

B.3.3 Shipper Parameters

The list of shippers found in the *shippers.csv* indicate the possible shippers which will make shipping requests throughout the simulation's run. To add new shippers, the user manually inputs new entries in the *shippers.csv* file following the specifications below. Shippers are defined as follows

Parameter	Description
shipperID	*Whole Number* A unique numerical ID identifying the shipper
category	*String* Attribute indicating the category of the shipper. In the current version, this attribute is not taken into consideration and can be left blank.

Table 4: Shipper Parameters

B.3.4 Carrier Parameters

The carriers added in the *carriers.csv* represent the list of carriers (represents a company) which will make offers to transport goods. To add new carriers, the user manually inputs new entries in the *carriers.csv* file following the specifications below.

Note that a *carrierID* may be listed multiple times when the carrier in question can generate offers across various categories and modes. For instance, a carrier holding two types of trucks and one type of train would be denoted across three lines in the CSV file:

```
1,24f,vehicle,...
1,40f,vehicle,...
1,10-53f,train,...
```

Carriers are defined according to the attributes below.

Parameter	Description
carrierID	*Whole Number* A unique numerical ID identifying the carrier
category	*String* This attribute indicates the category of vehicle the carrier holds. This value affects the capacity of the generated offers. Accepted values for vehicles are • 24f: A 24 feet truck with a capacity of 1500 cubic feet • 40f: A 40 feet truck with a capacity of 2385 cubic feet • 53f: A 53 feet truck with a capacity of 3830 cubic feet Accepted values for trains are • N-40f: A train with N containers of 2385 cubic feet • N-53f: A train with N containers of 3830 cubic feet
mode	*String* The mode of the carrier. Accepted values in the current version are "vehicle" and "train"
serviced terminals & zones	*String* A pipe (|) delimited list of zones and terminals the carrier services. For instance, a carrier that only services zone Z_A_1_1 and terminals CT_A and CT_B is indicated as Z_A_1_1| CT_A|CT_B

Table 5: Carrier Parameters

B.3.5 General Parameters

General parameters are parameters which apply to the overall workflow of the simulation and not to a specific component. General parameter include the following list

Parameter	Description
seed	*Whole Number* Value is the seed used for any number randomly generated. The seed allows for the same results to be obtained for each run having the same seed. Value must be a whole number greater or equal to 0.
shipperRequestPer Period	*Whole Number* The number of shipper requests to generate per period for both known and predicted requests. Value must be greater than 0.
carrierOfferPerPeriod	*Whole Number* The number of carrier offers to generate per period for both known and predicted offers. Value must be greater than 0.
maxSimulationPeriods	*Whole Number* The total number of periods the simulation runs for. Value must be greater than 0.
lookAheadPeriods	*Whole Number* The number of look ahead periods. Value must be smaller than the maxSimulationPeriods.
finalDecisionPeriods	*Whole Number* The window of periods for which the decisions are final. This attribute refers to the *Current Implementation* window described in the M1M operations paper. For instance, with a final finalDecisionPeriods value of 2, any decision taken for the current period and the subsequent two periods is final and can not be altered. Value must be smaller than the lookAheadPeriods.
regenerateForecast Data	*Boolean* Boolean value indicating whether forecast (predicted) data is regenerated at each period or only one additional period is added. In other words, if true, at period $t + 1$, we discard all previous predicted data and generate new predicted data for the look ahead periods. If false, at period $t + 1$, we carry forward predicted data for periods $t + 2$ to T generated at period t and only generate new predicted data for the new additional look ahead period.
splitShipperRequest	*Boolean* Boolean value indicating whether the items in a shipper's request may be split or not. The current implementation does not allow splitting, thus the value must be false.

splitCarrierOffer	*Boolean* Boolean value indicating whether the vehicles in a carrier's offer may be split or not. The current implementation does not allow splitting, thus the value must be false.
optimizationModel	*String* The value indicates the optimization model used to perform the item-to-bin assignment. Current accepted values are "HM1", "HM2", "HM3" and "HM4".
inputDataFile Directory	*String* A path containing the files shipperRequests.csv, carrierOffers.csv, spotMarket.csv and assignedCost.csv. If the path is left blank, the data is generated randomly. If not blank, the format these files must follow is described in Section B.4.
deleteOptimization InputFiles	*Boolean* The optimization model requires data to be in a specific file format. If the variable is true, these temporary files are deleted to not take disk storage space on the computer. Additionally, if the value is set to true, deleteAssignCostFile is set to true by default.
deleteAssignCostFile	*Boolean* The optimization model requires a file holding the item-to-bin assignment cost for each period. This file could get quite large depending on the indicated values for shipperRequestPerPeriod and carrierOfferPerPeriod. If set to true, the files are deleted at the end of the simulation run to not take disk storage space on the computer.

Table 6: General Parameters

B.4 Input Data Files

The simulation model is adopted to function whether a user has collected data or not. In the case the user wishes to provide their own data, there are four files which must be provided. If no files are provided, the simulation model will generate data. Below we cover the format of each expected file and the expected order of columns in the CSV files. A sample of the expected format can be found in the *M1M-V1* zip file under the *sample_input_data* folder.

B.4.1 Shipper Requests

The shipper requests file is expected to be named *shipperRequests.csv*. It's content revolves around the shipper requests for the duration of the simulation. The CSV column names and their format is as follows.

- shipperID: A numerical ID identifying the shipper
- requestID: A numerical ID identifying the request
- item number: A numerical value indicating the item number within a request. If a request contains more than one item, each item should be listed separately in the case shipper request are to be split
- category: A string value indicating the shipper's category
- origin: The shipper facility the items depart from
- destination: The consignee facility items are to be delivered to
- volume: The volume of the item
- loading rules: The loading rules
- request reception time: The period the IDSP receives the requests
- latest acceptance time: The period by which the IDSP must accept or reject a request. Must be greater or equal to the request reception time
- earliest release: The period the shipper makes the items available
- latest release: The latest period the item can be picked up. Must be greater or equal to the earliest release
- earliest delivery: The beginning of the time window at which a delivery is considered early.
- earliest target delivery: The end of the time window at which a delivery is considered early and the beginning of the target delivery time window.
- latest target delivery: The end of the target delivery time window and the beginning of the late delivery time window.
- latest delivery: The end period of the late delivery time window.

B.4.2 Carrier Offers

The carrier offers file is expected to be named *carrierOffers.csv*. It contains all carrier offers received throughout the simulation's lifetime. The CSV column names and their format is as follows.

- carrierID: A numerical ID identifying the carrier
- offerID: A numerical ID identifying the offer
- vehicle number: The vehicle number in case an offer contains more than one vehicle of the same type
- category: The category of the offer
- mode: The mode of the offer
- origin: The location from which the carrier departs
- destination: The location at which the carrier ends their route
- route: A pipe delimited list indicating the origin, destination and travel time of each leg. A single leg is a hyphen separated list denoted as *origin-destination-travelTime*. For instance, a carrier originating from the shipper facility Z_A_1_1 travelling to the consignee facility Z_B_1_1 is denoted as: Z_A_1_1-CT_A-1|CT_A-CT_B-2|CT_B-Z_B_1_1-0. In the example, the first leg has a travel time of 1, the second leg a travel time of 2 and the last leg a travel time of 0 periods.
- capacity: A pipe delimited list of the capacity of the carrier for each leg. For instance, if a carrier has three stops in their route, their capacity is denoted at 2000|1500|2350
- loading rules: The loading specifications
- request reception time: The period the IDSP receives the offer
- latest acceptance time: The period by which the IDSP must accept or reject an offer
- earliest availability: The earliest period the carrier may start working
- latest availability: The latest period the carrier may start working
- total travel time: The aggregate travel time of each leg from the origin to the destination
- fixed cost: Base cost of selecting the offer
- variable cost: The unit cost of transporting an item per unit of volume and of distance

B.4.3 Spot Market Cost

The spot market cost information is to be located in a file named *spotMarket.csv*. It contains the information of sending an item through the spot market for each period of the simulation.

The spot market file contains three types of columns; the requestID, the item number and the cost. Each period for which the simulation is to run has its own cost column. For instance, a simulation running for 6 periods would have the following csv header:

requestID,item number,1,2,3,4,5,6

The content of each column is as follows.

- requestID: The request for which the spot market costs apply to
- item number: The item number related to the request ID
- cost columns: The cost of transporting an item through the spot market for a carrier departing at a period matching the column number. If the column is outside the request's availability window (before the earliest release and after the latest release), a cost of -1 is assigned.

B.4.4 Assigned Cost

The assigned cost information is to be located in a file named *assignedCost.csv*. The CSV file is formatted as below

- requestID: The ID of a request
- offerID: The ID of an offer
- period: The assignment period
- cost: The cost of assigning the requestID to the offerID at the indicated period.

B.5 Running Simulation & Its Workflow

With the input parameters described above set, the next step is to run the simulation. To run the simulation, open any file ending with the ".cc" extension under the *src/* folder. Once the file opens, click the run icon on the top menu bar as displayed in Figure 7. Note that all steps described in the following step are done automatically by the software. The user does not need to perform any manual action from this stage onwards.

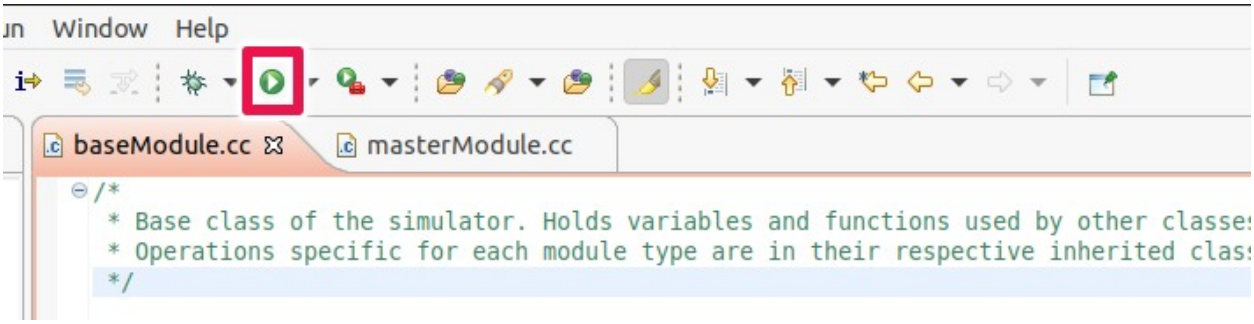


Figure 7: Running Simulation

Once launched, a new window opens up displaying a visual representation of the simulation. Clicking on the play button, as displayed in Figure 8, in the opened window starts the simulation events.

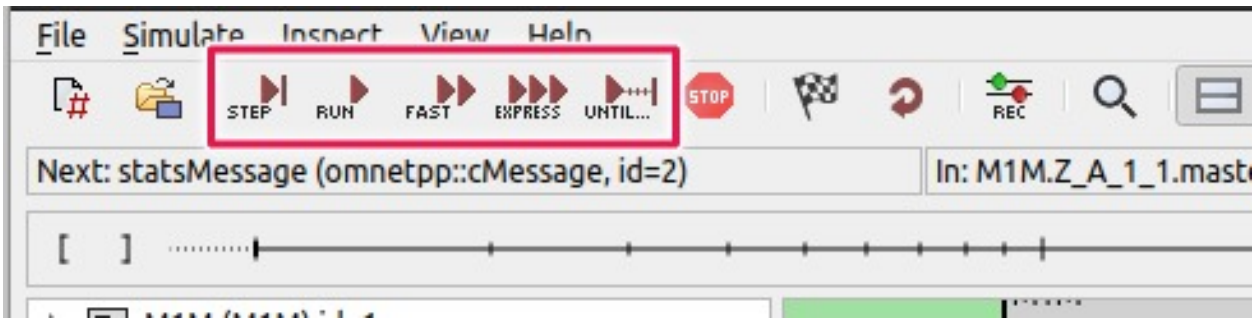


Figure 8: Simulation Play Controls

In the following subsections, we will briefly cover in chronological order the simulation events which occur at each period. In the simulation's design, a period was defined as being one day. Figure 9 displays the general workflow of the simulation model described below.

B.5.1 Data Generator

At each period of the simulation, the first step is to generate data. In the case the user provides data files, only predicted data is generated. If no data is generated by the user, both actual (known) and predicted data are generated randomly. The generated files follow the structure described in Section B.4.

At each period, the number of known requests and offers generated is limited to the number specified by the user as input data. On the other hand, for predicted data, for each period within the look ahead window, a number of requests and offers matching the value specified by the user are generated.

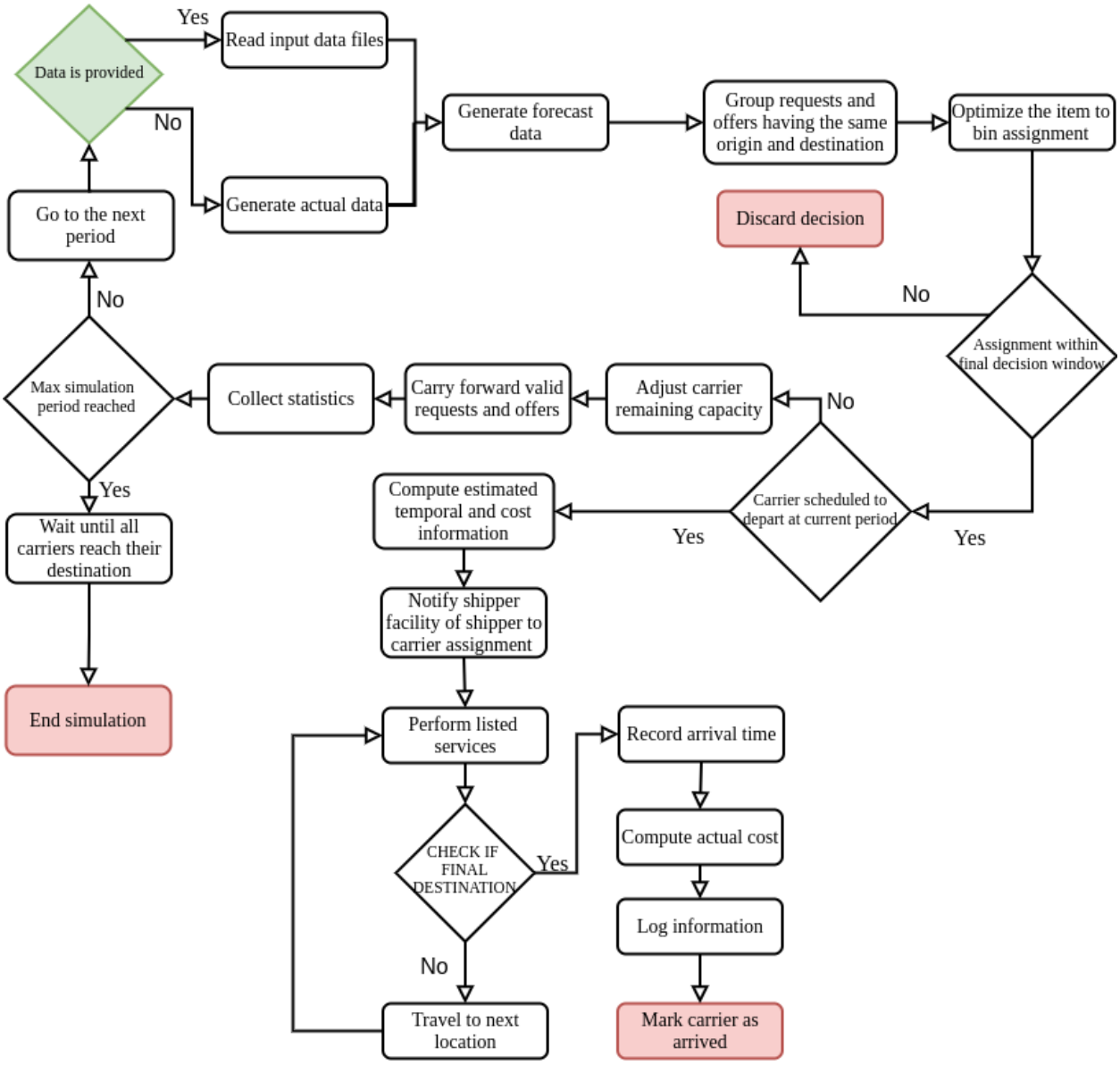


Figure 9: Simulation Workflow

Once the data is randomly generated and in the expected file format, the next event is the optimization process.

B.5.2 Optimization

The next step in the simulation process revolves around optimizing the item-to-bin assignment. With this step, we first need to prepare the data in a format readable by the currently implemented optimization models.

The used optimization models do not consider the origin and destination facility of the items to assign to bins as part of their parameters. Consequently, the first step within this event groups all shipper requests for items which have the same origin and destination facility. Moreover, a similar process is performed for carrier offers where offers having the same origin and destination facilities as the shipper request are retained.

Moreover, the implemented models expect the data to be in 8 distinct files following a specific format. In the file formats described below, an item refers to items the shipper wishes to have delivered and a bin refers to the medium of transportation offered by the carrier. In the OMNeT++ interface, these files can be found in the under the file directory *input_files/CURRENT_DATE/CURRENT_TIME/PERIOD/ORIGIN-DESTINATION/optimization*. Note that the user does not need to create these files as they are automatically generated from the input data files described in Section B.4. The tab delimited files are formatted as follows.

1. **Assign_Costs**
   The Assing_Costs file contains the cost of shipping an item, for a given bin, during a specific period. The file consists of an $Nx4$ matrix where the first column is the itemID, followed by the binID, the period and the assigned cost. In a scenario where we are generating 100 items, for 20 bins, across 5 periods, the resulting file will have a size of $10,000x4$.

2. **Bins_Capa**
   This file is composed of an $Nx2$ matrix where N is the number of bins generated. The first column is the binID and the second column is the volume capacity of the bin.

3. **Bins_Cost**
   The Bins_Cost file is composed of an $Nx3$ matrix where N is the number of bins generated. The first column is the binID, the second is the fixed cost and the last column is the variable cost of the bin.

4. **Items_Size**
   This file again follows a similar structure to the Bins_Capa file. However, in this instance, N represents the number of items generated and the second column holds information regarding the item's volume.

5. **param**
   The param file is a file holding a single vector of size 3. This vector holds metadata information about the data generated at the current period. The first item in the vector is the number of items generated, followed by the number of bins, the number of periods the simulation runs for.

6. **Spot_Cost**
   The Spot_Cost file consists of an $Nx(P+1)$ matrix where N is the number of items and P is the number of periods. The first column of the matrix is the itemdID and the remaining P columns are the spot-market cost of the item for each period. If the item is not available at a certain period, it holds a cost of -1.

7. **Time_Bins**
   This file contains an $Nx4$ matrix where N is the number of bins. The first column is the binID, the next two columns hold the availability time window of the bin and the fourth column is its travel time.

8. **Time_Items**
   The Time_Items file holds a matrix of size $Nx5$ where N is the number of items. The first column holds the itemIDs. The next four columns hold the following time variables regarding the item: earliest release, earliest target delivery, latest target delivery and latest release.

The above files contain both the known and predicted data for the operations planning horizon. We differentiate between both types of data by adding the keyword "F" at the beginning of each predicted request and offer.

Once the optimization models ingest the given data, there are two types of assignment completed. The first type is an item assigned to a bin departing at a specific period. The second type is an item to be transported through the spot market at a specific period. The optimization results are located in the *input_files/CURRENT_DATE/CURRENT_TIME/PERIOD/ORIGIN-DESTINATION/optimization* file directory and have the below format. Note that although both known and predicted data are given as input, the output files only contains the item-to-bin assignment of known requests and offers.

- **binSchedule.csv**
  This file holds the shipping information for items sent through regular bins. This file consists of three columns representing the period, the carrier ID and the list of items to be shipped. In the CSV file, the column containing the list of items to be shipped is a single cell where items are in a pipe delimited list (ex: item1|item2|item3|....).

- **spotMarketSchedule.csv**
  This file holds information about items to be shipped through the spot-market bins. This CSV file contains two columns indicating the period in which the item is to be delivered via the spot-market.

As post processing steps to the optimization, the results are analyzed to retain decisions that fall within the *Current Implementation* window. Decisions falling outside this window are discarded as the item-to-bin assignment may change in the subsequent periods. For shipper requests and carrier offers for which their decision has been discarded, their requests and offers are carried forward to the subsequent period to be reconsidered within the optimization process.

For the retained decisions, we first mark the assigned shipper request and carrier offers as accepted. Similarly, for requests and offers which have not had an assignment and the current period is their latest acceptance period, they are marked as rejected.

Moreover, for the retained decisions, we distinguish between those scheduled to depart their facility during the current period and those scheduled to leave in the future. For carriers scheduled to depart in the future, their offer is carried forward to the subsequent period while taking into account their remaining capacity given the current list of items assigned to them. For carriers departing this period, the results are sent to the next event responsible of initializing the movement of the carrier within the physical network.

The above process is repeated for every possible origin and destination facility combination. It is assumed that the origin and the destination facility are not the same.

B.5.3 Shipping Orchestrator

The shipping orchestrator is responsible of collecting the optimization results for carriers scheduled to depart the current period. For each carrier, the shipping orchestrator records information that is available prior to departure. These include the expected departure period, the expected arrival period, the estimated cost (based on the carrier's fixed cost and the sum of the assignment costs of items it carries), etc. These metrics become useful when uncertainty factors are introduced or to track delays given the time a carrier was at a node being serviced.

Once the information is recorded, each shipper facility is notified of the results of the optimization process indicating the list of items each carrier will transport.

B.5.4 Carriers Travel Through the Network

At this stage, each shipper facility is aware of the item-to-bin assignment for carriers departing the current period. Consequently, we perform the initial consolidation where the item is picked up by the carrier. To simulate this action, a time delay representing the time it took to load the items into the vehicle is introduced. If the items were all loaded within a single period, the carrier departs on time. If it takes more than one period to load the items, the carrier only leaves once all items are loaded.

Once a carrier departs the shipper facility, it travels to the next location as part of its route based on the travel time indicated in the offer. Once at the next location, the services required (loading, unloading, transferring, sorting or sorting) for items within its cargo are performed. If no services are required at the terminal, the carrier simply passes nearby the next location in its route within necessarily stopping. These steps are repeated until the carrier reaches their final destination.

At their final destination where the items are due, the carrier goes through an unloading process. A carrier is only marked as arrived once all items within its cargo are unloaded. Thus, if it takes more than one period to unload the cargo, additional delays are recorded.

Based on departure and arrival delays, the estimated arrival may not be reflective of the actual arrival period. Consequently, for each item, we verify if the actual arrival is the same as the expected arrival period. If delays occurred , a penalty may be applied to the cost. To set the applicable penalty cost, refer to Section C.2.8.1.

B.5.5 Log Items & Statistics

As the final step in the list of periodic events, logs and statistics are collected. For logs, new information is appended to the files described below throughout the simulation up until the final period. The files below can be found in the directory *logs/CURRENT_DATE/CURRENT_TIME/*.

- **carrierOffers.csv**
  File contains all carrier offers which were generated throughout the simulation including their physical, temporal, cost and route information. Additionally, the acceptance and rejection status of each offer is indicated.

- **shipperRequests.csv**
  File contains all shipper requests which were generated throughout the simulation including their physical and temporal information. Additionally, the acceptance and rejection status of each request is indicated.

- **spotMarket.csv**
  The spot market file contains the spot market assignment cost of each request for each period of the simulation.

- **processedCarrierOffers.csv**
  This file logs information about a carrier once it arrives at its destination. It holds information about temporal and cost information which were computed prior to the carrier's departure. It also holds the information on the actual time and cost information which are logged based on events which occurred during the carrier's trajectory. Additionally, there are two columns,departure and route delays, indicating the number of periods the carrier was delayed. A departure delay indicates the time by which a carrier left the terminal later than expected. An arrival delay indicates a delay which

occurred after the carrier left its origin location. Arrival delays usually indicate a carrier could not have its load processed on arrival and had to wait additional periods to be processed (unloaded). Lastly, a list of items transported through the carrier is included.

- **processedShipperRequests.csv**
  This file logs information about a shipper requests once it arrives at its destination. It contains similar information as carriers, but on a per item basis.

In addition to the logs collected about the shippers requests and carrier offers, statistics are also collected for zones and terminals. More specifically, for each zone and terminal, we collect periodic departures, arrivals, number of items processed, delays and average time to service (perform on of the 5 possible services) a carrier.

As these statistics are collected through OMNeT++ functions, their results are located under the *src/General.anf* file. Should the file not be presented, it can be automatically generated by opening the *src/results/General.vec* file. Note that if a version other than OMNeT++ 5.6.2 was installed, these files may be corrupted.

To add more statistics to be collected, refer to Section C.2.8.2.

B.5.6 Dry Run

In this section, a minimal example representing what happens at each step of the simulation is presented. Note that this example is only to exemplify some of the details presented in the previous sections and is not based on an actual run.

In this example, we assume we have a simple network where two shipper facilities, A and B, are directly connected. We will only consider the case of items going from facility A to facility B.

We set the following initialization parameters:

- Number of periods: 3
- Look ahead periods: 2
- Final decision window: 2
- Travel time between A and B is 1 period

For simplicity sake, we have one carrier which generates one offer per period for the mode vehicle. All offers depart from facility A to facility B with a direct route. Additionally, we have 1 shipper making 2 requests each per period.

B.5.6.1 Period 1

Data generator: we randomly generate the following requests and offers at period 1

Shipper Requests							
Request ID	Volume	Reception Period	Latest Acceptance	Earliest Release	Earliest Target Delivery	Latest Target Delivery	Latest Release
1	12	1	3	2	2	3	3
1	15	1	2	1	1	3	3

Spot-Market Cost			
Request ID	Period 1	Period 2	Period 3
1	-1	2	5
2	3	4	6

Carrier Offers								
Offer ID	Capacity	Reception Period	Latest Acceptance	Earliest Availability	Latest Availability	Travel Time	Fixed Cost	Variable Cost
1	100	1	2	1	3	1	20	0.25

Regular Bin Assign Cost			
Request ID	Offer ID	Period	Cost
1	1	1	1
1	1	2	1
1	1	3	3
2	1	1	2
2	1	2	4
2	1	3	7

Optimization We first format the data into the format explained in the user guide's Section B.5.2. The assigned cost and spot market cost are already in said format. The remaining files are simply filled with information from the shipper request and carrier offers tables.

Passing the bin capacity, cost and time constraints, the item's size and time constraints, the spot market cost and assignment cost to the optimization algorithm, assume we get the following optimization results. Note these results below are random assignments and not based on any optimization algorithm.

Bin Assignment		
period	Offer ID	Request ID
3	1	1

Regular Bin Assign Cost	
Period	Request ID
1	2

Since the final decision window is two periods, only assignments for period 1 and 2 are honored. Other assignments are ignored. Consequently, we ignore the assignment of item 1 to bin 1 as it is scheduled to depart at period three. However, since both are not at their latest acceptance period yet, we carry forward the request and offer to the next period.

Moreover, we have item 2 scheduled to leave through the spot market at the current period. We notify the shipper orchestrator about this item.

Shipping Orchestrator: The shipper orchestrator receives all carriers departing this period. We have only one carrier departing in this example. We record the following information for the carrier:

- carrierID: S2 (For items leaving through spot market, the ID is the request's ID)
- Mode: Vehicle
- Type: Spot Market
- Volume transported: 15
- Expected departure: period 1
- Expected arrival: period 2 (with a travel time of one period)
- Estimated cost: 3$
- Route:
  - Leg 1
    - origin: facility A
    - destination: facility B
    - travel time: 1
    - Actions to perform: loading
  - Leg 2
    - origin: facility B
    - destination: none (since we are at the final destination)
    - travel time: none (since we are at the final destination)
    - Actions to perform: unloading

Travelling through the physical network: Assume the current facility can load items at a speed of 10 cubic feet per period. Since this item is of size 15, it will take two periods to load. We thus have a departure delay and the item loading will be completed the next period. As no other items are scheduled to depart this period, this step is done.

Logs & Statistics: Logs about shippers and carriers are only recorded once they arrive at their destination or they are marked as rejected. In this instance, we have nothing to log during the first period.

We collect statistics about each facility (A and B) noting the following statistics for the current period: number of departures, arrivals, number of delays, remaining processing units, etc.

Period 1 is now complete and we jump to period 2.

B.5.6.2 Period 2

Data generator: we randomly generate the following requests and offers at period 2 in addition to carried forward request and offer

Shipper Requests							
Request ID	Volume	Reception Period	Latest Acceptance	Earliest Release	Earliest Target Delivery	Latest Target Delivery	Latest Release
1	12	1	3	2	2	3	3
3	21	2	3	2	2	3	3
4	101	2	2	2	2	2	2

Spot-Market Cost			
Request ID	Period 1	Period 2	Period 3
1	-1	2	5
2	-1	7	8
4	-1	5	-1

Carrier Offers								
Offer ID	Capacity	Reception Period	Latest Acceptance	Earliest Availability	Latest Availability	Travel Time	Fixed Cost	Variable Cost
1	100	1	2	1	3	1	20	0.25
2	105	2	2	2	3	1	23	0.33

Regular Bin Assign Cost			
Request ID	Offer ID	Period	Cost
1	1	2	1
1	1	3	3
1	2	2	2
1	2	3	4
3	1	2	2
3	1	3	3
3	2	2	3
3	2	3	5
4	1	2	1
4	2	2	2

Optimization We repeat a similar process as the previous period of placing the data in the file format described in Section B.5.2 of the user guide. Assume we obtain the results below

Bin Assignment		
period	Offer ID	Request ID
3	1	1|3

Regular Bin Assign Cost	
Period	Request ID
2	4

Since both decisions are within the final period window, both are accepted. Additionally, since offer 1 is to depart next period, we carry it forward while reflecting its reduced capacity given two items are assigned to it.

Shipping Orchestrator: The shipper orchestrator receives all carriers departing this period. We have only one carrier departing in this example. We record the following information for the carrier:

- S4 (For items leaving through spot market, the ID is the request's ID)
- Mode: Vehicle
- Type: Spot Market
- Volume transported: 11
- Expected departure: period 2
- Expected arrival: period 3 (with a travel time of one period)

- Estimated cost: 5$
- Route:
  - Leg 1
    - origin: facility A
    - destination: facility B
    - travel time: 1
    - Actions to perform: loading
  - Leg 2
    - origin: facility B
    - destination: none (since we are at the final destination)
    - travel time: none (since we are at the final destination)
    - Actions to perform: unloading

Travelling through the physical network: Since at the last period we didn't complete the loading, we first start by completing the loading operation for carrier S2 transporting item 2. Once loaded (within the period) it departs along its route and we record its actual departure time as period 2. We also have enough processing capacity for carrier S4 to depart on time.

Logs & Statistics: We still have no carriers which arrived at their destination, so we have nothing to log on that front. However, the carrier offer #2 had a latest acceptance for the current period (period 2). As it wasn't accepted, we log it as rejected.

We collect statistics about each facility (A and B) noting the following statistics for the current period: number of departures, arrivals, number of delays, remaining processing units, etc.

Period 2 is now complete and we jump to period 3.

B.5.6.3 Period 3

Data generator: we randomly generate the following requests and offers at period 3 in addition to carried forward offers. Note that the carried forward requests (request 1 and 3) are not noted as they have already been assigned. Additionally, note that carrier 1's capacity has been adjusted reflecting the volume of both items loaded.

Shipper Requests							
Request ID	Volume	Reception Period	Latest Acceptance	Earliest Release	Earliest Target Delivery	Latest Target Delivery	Latest Release
5	10	3	3	3	3	3	3
6	28	3	3	3	3	3	3

Spot-Market Cost			
Request ID	Period 1	Period 2	Period 3
5	-1	-1	7
6	-1	-1	11

Carrier Offers								
Offer ID	Capacity	Reception Period	Latest Acceptance	Earliest Availability	Latest Availability	Travel Time	Fixed Cost	Variable Cost
1	100	1	2	1	3	1	20	0.25
3	75	3	3	3	3	1	17	0.38

Regular Bin Assign Cost			
Request ID	Offer ID	Period	Cost
5	1	3	2
5	3	3	5
6	1	3	4
6	3	3	5

Optimization We repeat a similar process as the previous period of placing the data in the file format described in Section B.5.2 of the user guide. Assume we obtain the results below

Bin Assignment		
period	Offer ID	Request ID
3	1	1|3|5|6

Regular Bin Assign Cost	
Period	Request ID
-	-

Since the carrier offer has already been accepted and the item assignment is final, we move to the next step.

Shipping Orchestrator: The shipper orchestrator receives all carriers departing this period. We have only one carrier departing in this example. We record the following information for the carrier:

- carrierID: 1
- Mode: Vehicle
- Type: Bin

- ID of items in the carrier's cargo: 1—3—5—6
- Volume transported: 71
- Bin total capacity: 100
- Expected departure: period 3
- Expected arrival: period 4 (with a travel time of one period)
- Estimated cost: fixedCost+assignedItemCost = 20+3+3+2+4=32
- Route:
  - Leg 1
    - origin: facility A
    - destination: facility B
    - travel time: 1
    - Actions to perform: loading
  - Leg 2
    - origin: facility B
    - destination: none (since we are at the final destination)
    - travel time: none (since we are at the final destination)
    - Actions to perform: unloading

Travelling through the physical network: We only have one carrier departing this period. Assume the facility has enough processing capabilities and no delays are experienced.

We also have two carriers scheduled to arrive at facility B this period. We first process carrier S2 transporting item 2. We log its arrival as period 3 and apply a penalty since it was expected to arrive at period 2, but departure delays made it arrive late.

Carrier S4 is also at facility B this period. Assume the facility can only unload 100 cubic feet per period. Since the item is of size 101, the unloading will be completed the next period.

Logs & Statistics: We have one carrier which arrived and was processed. We log the information recorded prior to its departure (expected travel times and cost) and the actual departure, arrival and computed cost with penalty.

Additionally, the carrier offer #3 had a latest acceptance for the current period (period 3). As it wasn't accepted, we log it as rejected.

We collect statistics about each facility (A and B) noting the following statistics for the current period: number of departures, arrivals, number of delays, remaining processing units, etc.

Period 3 is now complete and we jump to period 4.

B.5.6.4 Period 4

The simulation was only scheduled to run three periods as per the input parameters. However, as some carriers haven't reached their final destination yet, the simulation continues until all carriers have been marked as arrived. Note that no new data is generated and no optimization is done during those additional periods.

Travelling through the physical network: We start by processing carrier S4 for which we didn't have the capacity to process. We log its arrival time as period 4 indicating there was an arrival delay. We recompute its actual cost accordingly.

We also have carrier 1 transporting 4 items which arrived. We have enough processing units to unload the carrier.

Logs & Statistics: We have two carriers which arrived this period. One with a delay (S4) and one without delays (carrier 1). We thus log the information for each reflecting their actual temporal and cost information.

We collect statistics about each facility (A and B) noting the following statistics for the current period: number of departures, arrivals, number of delays, remaining processing units, etc.

Period 4 is now complete and we jump to period 5.

B.5.6.5 Period 5

We are past the last simulation period and all carriers have arrived at their final destination. The simulation ends.

Appendix C Developer Guide

C.1 External Functions & Objects

In this section, we cover components that were developed outside the OMNeT++ environment. Having these components independent of OMNeT++ allows for them to be changed without requiring extensive changes on the OMNeT++ side. Of these components, we have both the data generator and the optimization models.

C.1.1 Data Generator

The data generator module is responsible for generating random data to be used by the optimization models. This includes both known and predicted data. The source code of the data generator can be found in the *M1M-V1/src/externalFunctions* directory as displayed in the below tree.

```
externalFunctions
├── header
│   ├── dataGenerator.h
│   └── graph.h
├── dataGenerator.cc
└── graph.cc
```

There are four main functions to visit in the *dataGenerator.cc* file which cover the logic behind the generated data.

NOTE: The code lines (variable names and for loops) are copied verbatim from the source code. They can be located by using the find function in the text editor.

The first function is named *generateShipperRequest* and it generates shipper requests. For each shipper, it generates between 0 and 4 requests containing between 1 and 8 identical items. To modify the parameters of the generated shipper request, the code of interest is located within the for loop *for(i=0; i ¡ numberOfRequestsPerShipper; i++)*. Requests are generated up until the desired number of requests to generate per period is reached.

Moreover, the second function in the data generator, *generateSpotMarketCost*, determines a spot-market cost for each request generated. To change the parameter determining the cost, modify the content of the variable *spotMarketCostVector[i][t-1]* found in the for loop *for(int t = earliestRelease; t ¡= latestRelease; t++)*

The third function, *generateCarrierOffer*, functions in a similar fashion to *generateShip-*

perRequest, but for carrier offers. For each carrier, it generates between 0 and 4 offers containing between 1 and 5 identical vehicles or a single train. To modify the parameters of the generated carrier, the portion of the code to modify is found within the brackets of the for loop *for(i=0; i ¡ numberOfOffersPerCarrier; i++)*. Offers are generated up until the desired number of offers to generate per period is reached.

The last function of main interest in the file is named *computeAssignmentCost*. This function takes the temporal and size information of a request, the variable cost of an offer and the assignment period to determine a cost. To modify the ruling of the assigned cost, the variable to modify within the *computeAssignmentCost* function is named *assignedCost*.

Once the data is generated, the populated matrices are passed to the function *writeDataToFiles* which writes the data into the format described in Section B.4.

The second file found in this directory, *graph.cc*, is a class which runs the depth first search algorithm. In essence, this algorithm creates a graph corresponding to the network defined in the *params/network.csv* and finds all possible routes between each pair of origin and destination for each mode. These routes are used to randomly assign a route within carrier offers. Additionally, for the spot-market, we use the shortest route returned between the origin and destination.

C.1.2 Optimization

The file directory *M1M-V1/optimizationModels* containing the optimization models is structured as below.

```
optimizationModels
├── HM*
│   ├── include
│   ├── objectFiles
│   └── src
│       ├── HM*.c
│       ├── MostEfficient.c
│       ├── Profitable.c
│       ├── VCSBPPAC.c
│       └── main.c
├── IO
│   ├── include
│   ├── objectFiles
│   └── src
│       └── IO.c
└── generateObjectFiles.sh
```

There are 4 optimization models currently available, HM1-HM4, and they all follow the same folder structure. The file HM*.c contains a function called HM*, which is used to call

each algorithm. For each model, there is also a *main.c* file which allows the user to specify an input directory containing files in the format described in Section B.5.2. This script was added in order to test optimization models before importing them to OMNeT++. The generated object file for the main function can be launched using the command *'./HM*/main'*.

C.1.2.1 Generating Object Files

To generate the object files, once the source code modifications are complete, the script *generateObjectFiles.sh* is to be launched. The script will delete previously compiled object files and generate new ones based on the current state of the code.

To launch the script, in a linux or macOS environment, navigate to the *optimizationModels* directory in the command line. Running the command
'bash generateObjectFiles.sh' will output the object files in the *objectFiles* directory.

On a Windows environment, it is recommended to install *Git Bash* [3] to run the bash script. To produce the object files, from the Git Bash command line, navigate to the *optimizationModels* directory and run the command
'sh generateObjectFiles.sh'.

NOTE: Depending on the compiler, the generated object files may only function on the operating system they were built in. For instance, object files generated in a linux environment may not be recognized in a Windows environment

---

C.1.2.2 Importing Object Files To OMNeT++

To import the generated object files in OMNeT++, copy the files to their respective directories in the *src/externalObjects* directory in OMNeT++. If generated using Windows, place the files in the *externalObjects/windows* directory. If generated in a unix-like operating system, they are to be placed in the *externalObjects/other* directory.

Once copied, make sure the path in the *makefrag* file matches the location the files were placed in for both the header and object files.

In the situation the *makefrag* file is accidentally deleted, it can be generated by following the below steps.

- Right click on project → properties → OMNeT++ → Makemake → src: makemake... → options... → custom

[3] Download Git Bash: `https://git-scm.com/downloads`

- For each library, add EXTRA_OBJS += -LPATH -lLIBRARY_NAME
- For headers, add add CFLAGS += -IPATH/include
- Apply the changes

NOTE: The object file name must start with the word lib. For example, the HM4 library file would be libHM4.a and indicated as -lHM4 in the EXTRA_OBJS field. Failing to add the lib keyword will result in the object file not being recognized.

Whenever a change is made to the imported objects, it is imperative to clean and rebuild the project to assure the latest version of the imported libraries is being used. This can be achieved through the following steps.

- Right click on the project's folder and click the *Clean Project* button
- Once the cleaning process is complete, right click on the project's folder and click the *Build Project* button

---

NOTE: The optimization algorithms are written in C. When importing their header in C++, it must be specified that these are C objects. This is done as follows:

```
extern "C" {
    #include "../externalObjects/include/HM.h"
}
```

C.2 OMNeT++

The simulation environment was built on top of the OMNeT++ API. In this section, we cover the overall tasks performed by each defined class. This information could be helpful in locating where a specific operation is implemented in the case the user wishes to change its logic. For details on the specifics of each function found in the class, extensive comments were left above each function definition describing its use. All described files can be found directly under the *src* folder. We start by covering concepts which are specific to OMNeT++ before exploring each defined class.

C.2.1 OMNeT++ Concepts

C.2.1.1 Modules

Before further covering the implementation details, we start by defining concepts at the forefront of OMNeT++; simple modules and compound modules. A simple module represents an entity able to perform a specific task [30]. For instance, in our use case, a simple module can be viewed as an entity which performs one of the 5 possible tasks at a terminal (pickup, sorting, cross dock, storing, drop off). From an object oriented point (OPP) of view, a simple module is a class.

On the other hand, a compound module is a dummy module that merely englobes one or more simple modules. Compound modules in themselves can not perform any actions. Their functionalities are fully dependent on those of the simple modules they contain [31]. In our model, a compound module could be represented by a terminal holding 5 simple modules depicting the 5 possible operations.

C.2.1.2 Messages

In OMNeT++, two modules have the capabilities of sending each other messages. Messages can be either simple words or data structures holding information populated according to the developers requirements [32].

C.2.1.3 Gates

In OMNeT++, a corridor between two locations is defined by gates [33]. Two modules, whether compound or simple, need to be connected in order to send each other messages. OMNeT++ presents three types of gates; input gates capable of ingesting a message, output gates capable of sending a message and inout gates able to both ingest and send messages.

C.2.1.4 NED

In OMNeT++, the simulation module network is defined using the NED programming language [34]. The NED file is where modules, gates and how the overall components connect to one another is defined. The NED programming language loosely uses multiple OPP practices, in particular, inheritance.

C.2.1.5 Parameters

Each simple and compound module may be defined by some parameters. For instance, a sample parameter may be a value indicating the storage space of a module. In OMNeT++,

module parameters allow for information specific to a module to be read or updated along the simulation run time [35]. Each module can hold a unique value for each defined parameter.

C.2.2 M1M.ned

With the OMNeT++ concepts defined above, we start by covering the *M1M.ned* file. The NED file contains the definition of all simple modules and compound modules as well as the physical network.

When defining a simple module, it won't have any functionalities until it is connected to a C++ class. To connect a simple module to a class, add the *@class(className);* tag within its definition. For the given use case, the simple modules include C++ classes which perform the data generation, optimization, shipping orchestration and classes specific to each service (loading, unloading, transfers, sorting and storing) offered by a zone or terminal. Additionally, all classes inherit the simple module *baseModule* which defines functions shared across multiple simple modules.

Moreover, the NED file contains 4 compound modules; the IDSP, zones, transfer terminal and consolidation terminal. The IDSP compound module groups the data generator, optimization and shipping orchestration module into one. Having the three simple modules into one compound module facilitates making sure operations are being performed in the expected order.

Additionally, zone and terminal modules contain the simple modules specific to the services they can perform. In other words, a zone module can have items picked up or dropped off, a transfer module can have items sorted or cross docked and a consolidation module contains all five services.

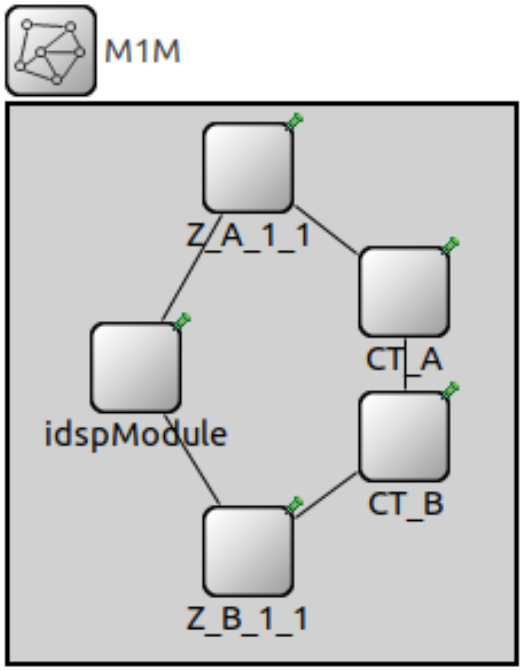


Figure 10: Three Segment Network

C.2.3 M1M.msg

The message file is used to define data structures which can transport information from one module to the next. In our implementation, there is one main message defined named *shippingJobDetails*. When a carrier travels across its route, it is in fact the *shippingJobDetails* message object which travels across the network. This message contains information about the carrier, its departure, arrival, items contained in its cargo and the route it takes.

Variables can be easily added to a message by adding their type and name to the message object body. As the message is built on top of the OMNeT++ *cMessage* class, each variable has getter and setter methods automatically generated. Note that these functions are defined in a class generated by OMNeT++ called *M1M_m.cc*. It is important to not alter the content of the *M1M_m.cc* class or change its file location for OMNeT++ to properly find its definition.

C.2.4 baseModule Class

With the network and message defined, we explore the user defined classes. For OMNeT++ to recognize a user defined class is in fact a simple module, the class must inherit the OMNeT++ defined *cSimpleModule* class. From the functions inherited from the *cSimpleModule* class, the main two used in our simulation module are the *initialize* and *handleMessage* functions.

When the simulation model is launched, the *initialize* function is triggered for each module. Thus, this function is used to set class variables or validate user specified parameters.

On the other hand, the *handleMessage* function is the entry to a simple module during the simulation's run. This function is used to define the logic of steps to perform every time the simple module is triggered.

In our implementation, we defined the *baseModule* class which inherits the *cSimpleModule* class. The *baseModule* is the class that all remaining implemented classes inherit. Of its functionalities, the *baseModule* validates the user input parameters following the specifications indicated in Section B.3. Additionally, it finds all possible routes, their distances and stores the information in C++ map object to avoid performing this operation every time data is generated. Lastly, the class contains helper functions used by multiple child classes such as functions to retrieve the next terminal to travel to, information about a shipper or carrier, etc.

C.2.5 dataGeneratorModule Class

At the beginning of each period, the first function triggered is the *handleMessage* function of the *dataGeneratorModule* class. In essence, this class calls the functions defined in Section C.1.1 to generate both known and predicted data. Additionally, it removes predicted data from previous periods which are no longer valid. These include predicted offers which have reached their latest acceptance period or any predicted data applicable to the current simulation period.

Once the data is generated, the *dataGeneratorModule* sends a notification message to the *optimizationModule* informing them the period's data is generated.

C.2.6 optimizationModule Class

When the *optimizationModule* is triggered, it first loops through all possible combinations of origin and destination facilities. For each pair, it formats the data into the structure described in B.5.2.

Once the data is in the expected format, it calls the optimization model and receives an item-to-bin assignment. The remaining steps of this class involve determining if an offer is accepted (has an assignment within the final decision period window) and carrying forward requests and offers which haven't been assigned or scheduled to depart at a future period. Additionally, for requests and offers which haven't been accepted and the current period is their latest acceptance, the *optimizationModule* discards them and marks the request or offer as rejected.

C.2.7 shippingOrchestratorModule Class

Once all optimization operations are completed, the *shippingOrchestratorModule* class receives a notification with the item-to-bin assignment for carriers scheduled to depart at the current period.

For each carrier scheduled to depart, the *shippingOrchestratorModule* creates a *shippingJobDetails* message described in Section C.2.3. Once the message is created, it notifies the masterModule class.

C.2.8 masterModule Class

The master module is considered the brain of each individual zone and terminal. For conciseness, we will refer to zones and terminals as nodes for the remainder of this section.

As carriers don't necessarily stop at each node and may perform only a few services out of the possible 5, a *masterModule* was introduced. When a carrier passes by a node, the master first verifies if there are any services to be performed at their node. If no services are required, the carrier simply continues their route to the next node and considers this stop a pass by. If services are to be performed, the master routes the carrier to the service at hand. After each service, the carrier verifies with the master if any more services are required. Once there are no more services to be performed, the master verifies the next node the carrier is expected to travel to and routes them accordingly.

C.2.8.1 Logging Carrier's Arrival

When a carrier is at their destination node and all services are performed, the *masterModule* adds the information found in the carrier's *shippingJobDetails* message to the log files. This includes route information, expected departure, arrivals, delays, etc.

Additionally, the *masterModule* verifies if an item has a penalty applied based on delayed deliveries. The current penalties were introduced to test the logic behind this aspect. To adjust the penalty value, the user may modify the formulas found in the *computeCostWithPenalty* function in the *masterModule.cc* file.

C.2.8.2 Collecting Statistics

In addition to its previously mentioned functionalities, the *masterModule* also records statistics specific to each node at the end of each period. New statistics variables can be added through the steps below

```
private:
    //Statistics tracking variables
    long VARIABLE_Name;

    //Tracking Vectors
    cOutVector VECTOR_NAME;
```

Listing 1: Creating Variables

```
void masterModule::initialize(int stage){
    if(stage == 0){
        VARIABLE_NAME =0;
        WATCH(VARIABLE_NAME);

        VECTOR_VARIABLE.setName("name");
```

Listing 2: Initializing Variables

1. In the *src/header/masterModule.h* file, there is a section where private variables are defined. Add the name of the variable you wish to track under said section
2. Next, in the same section, create a *cOutVector* where the collected statistics will be recorded at each period. Refer to listing 1 as a reference.
3. In *src/masterModule.cc* file, in the *initialize* function, assign an initial value to the metric created in step one.
4. Add a line of the format *WATCH(VARIABLE_NAME);* where VARIABLE_NAME is the name of the variable created in step one.
5. In the same *initialize* function, set a name for the vector created in step 2 by adding a line of the format *VECTOR_VARIABLE.setName("name");*. Refer to listing 2 for steps 3 to 5.
6. Place the created variable at the location in the code which affects its value. For instance, if you wish to track delays, find the part of the code which detects when a carrier arrives at its destination. On arrival, a condition can be added which detects if a carrier is delayed. If the condition is met, the variable is incremented.
7. In the *handleMessage* function, there is a portion of the code enclosed within a condition of the following format *msg→ isSelfMessage() && msg→getKind()== 2*. In this section, record the variable's value for the current period and reset it's value for the subsequent period. Listing 3 demonstrates the expected code format.

NOTE: The implementation for the statistics collection is based on the OMNeT++ Tic

```
void masterModule::handleMessage(cMessage *msg)
{
    if(msg-> isSelfMessage() && msg->getKind()== 2){
            VECTOR_NAME.record(VARIABLE_NAME);
            VARIABLE_NAME = 0;
```

Listing 3: Recording Statistics

Toc tutorial. Refer to *this section of the tutorial* [4] for more details on statistic collection.

C.2.9 serviceModule, pickUpModule & sortingModule Classes

The five possible services which can be performed in a terminal are defined across three classes. All three classes have the same composition.

When a service is being performed, we take the total volume of items in the carrier's cargo. We verify how long it will take to process the items. The master module at the node in question will be notified at X periods in the future once all operations are done. X refers to the amount of periods required to perform the service for the carrier in questions.

For the sorting action, defined in the *sortingModule* class, in addition to introducing a time delay for the time to store items, for items to be added to the storage, we add their information to a map object containing the items stored at the current terminal. As information, we note the requestID the item belongs to as well as its volume.

Lastly, for the pick up action, it implements the same time delay logic as other services. However, when an item is picked up, we verify if it was placed in the terminal's storage. Should this be the case, the item's volume is deduced from the terminal's occupied storage and removed from the map object.

[4] OMNeT++ Tic Toc Statistic Collection Tutorial: `https://docs.omnetpp.org/tutorials/tictoc/part5/#52-adding-statistics-collection`