FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Distributed Simulation and Exploration of a Game Environment

João Francisco Veríssimo Dias Esteves

Mestrado Integrado em Engenharia Informática e Computação

Supervisor: Daniel Castro Silva

July 23, 2020

Distributed Simulation and Exploration of a Game Environment

João Francisco Veríssimo Dias Esteves

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

July 23, 2020

Abstract

In order to accomplish certain missions, such as the transport of cargo or the detection of a forest fire, different kinds of autonomous vehicles may be used together. A computer simulation of vehicles cooperating in missions can be used to analyze and better plan these situations. However, like any software, the complexity of feasible missions in terms of area and vehicles is limited to the processing power available. This is the case of a multi-agent simulation platform developed by FEUP's Artificial Intelligence and Computer Science Lab on top of Microsoft Flight Simulator X (FSX). However, since the simulation engine, FSX, runs on a single computer, this limit can be greatly increased by distributing it to multiple computers.

A literature study has been conducted into several distributed simulation architectures introduced in the past decades (HLA, DDS, DIS, TENA, FMI) and into techniques for temporal synchronization in distributed computing. This resulted in the choice of DDS initially; however, it proved to be inadequate in practice considering the specific requirements of the platform, and so, RabbitMQ was used instead as the communication middleware following the conclusions of a parallel study. The techniques studied for temporal synchronization also weren't adequate, so an ad-hoc algorithm was developed.

A distribution approach has been proposed and implemented. It consists of having different instances of FSX on unique and distant locations in the virtual map, having each one in a different PC and simulating only a certain surrounding area. Thus, the vehicles crossing between the instances' areas migrate from instance to instance, minimizing the workload in each one whenever possible. Multiple components of the platform have been adapted to support this change of paradigm from 1 to many FSX instances. Periodically, an algorithm migrates vehicles based on two criteria: if they're getting too close to their instance's border, or if they can be migrated to an instance with an inferior load for the sake of load balancing. Another periodic algorithm synchronizes the internal times of the FSX instances by estimating the communication latencies between instances and adjusting the time of *slave* instances based on the time of one *master* instance.

A number of tests validate and showcase the results from both the vehicle migration and temporal synchronization algorithms, proving the main functionalities of the distribution adaptation of the platform work. However, some tasks are left for future work and their proposed solutions are properly explained.

Keywords: distributed simulation; multi-agent simulation; flight simulation

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
ATC	Air Traffic Control
BPMN	Business Process Model and Notation
COTS	Commercial-Off-The-Shelf
DARPA	Defense Advanced Research Project Agency
DDS	Data Distribution Service
DIS	Distributed Interactive Simulation
DoD	U.S. Department of Defense
FEUP	Faculty of Engineering, University of Porto
FOM	Federation Object Model
FMI	Functional Mock-up Interface
FSX	Microsoft Flight Simulator X
FTP	File Transfer Protocol
GPS	Global Positioning System
HLA	High Level Architecture
HTTP	Hyper Text Transfer Protocol
HTTPS	Hyper Text Transfer Protocol Secure
IEEE	Institute of Electrical and Electronics Engineers
LIACC	Artificial Intelligence and Computer Science Laboratory
M&S	Modelling & Simulation
MDA	Missile Defense Agency
MOM	Management Object Model
NATO	North Atlantic Treaty Organization
NTP	Network Time Protocol
OMG	Object Management Group
OMT	Object Model Template
OSF	Objective Simulation Framework
PDU	Protocol Data Unit
PKI	Public-Key Infrastructure
PTP	Precision Time Protocol
QoS	Quality of Service
RTI	Run-Time Infrastructure
SOM	Simulation Object Model
STANAG	NATO Standardization Agreement
TCP/IP	Transmission Control Protocol / Internet Protocol
TENA	Test and Training Enabling Architecture
TSO	TimeStamp Ordered
UDP	User Datagram Protocol
UI	User Interface
U.S.	United States (of America)

Chapter 1

Introduction

The *Artificial Intelligence and Computer Science Laboratory* (LIACC) at FEUP has been developing a multi-agent simulation platform with Microsoft Flight Simulator X (FSX) as its engine in order to simulate the execution of missions by multiple heterogeneous vehicles, be it planes, land vehicles or submarines [50]. Although one of the main uses of FSX is as a game, that is not the platform's purpose. Its purpose is to simulate the strategies of multiple autonomous vehicles on missions such as fire fighting or high valued targets detection, in order to hopefully serve as a learning tool for the entities behind these activities. The role of FSX here is solely to provide a physics engine for the simulation. In addition, Microsoft sold FSX to Lockheed Martin in 2009, who revamped it and sells it today as a separate product, Prepar3D[1]. It is aimed at commercial uses and is still actively developed as of June 2020, with DirectX 12 support in the 5.0 update of April 2020. Although it's a divergence from FSX, both products share a lot of their functionalities which contributes to the idea that FSX can be more than just a game.

1.1 Motivation and Research Questions

Currently, the FSX instance at the platform's core runs on a single computer, limiting the complexity of the missions executed to the processing power of this computer, particularly in the number of vehicles. This is not only due to the processing power required by the simulation itself, but also due to the required communications with FSX since every vehicle has to continuously exchange information with the platform.

There is also a documented limitation set by FSX [33] on the simulated area: only a circular area with a 200Km radius with the player at its center is simulated, the *reality bubble*. Outside this area, non-AI objects are completely removed from the system and new AI objects cannot be created [47]. This is a problem when missions have a big enough scope. A mission spanning the entire landmass of mainland Portugal is impossible like this.

[1] https://www.prepar3d.com/

In order to focus the research done in this thesis' area, the following four questions will be answered:

- How can the processing power of multiple computers be harnessed to expand the functionalities of the platform?
- What are the concrete benefits of such a distribution?
- Are there industry standards for distributed simulation that can simplify this process?
- Can a proper load balance be achieved amongst the participating computers?

1.2 Methodology and Expected Results

To answer the first question, this dissertation aims to solve the problem of limited processing power by distributing the FSX instance between different computers. By having each computer simulating different vehicles, more power is available for more vehicles in the simulation as a whole. As for the problem of the *reality bubble*, these FSX instances can be placed far enough from each other in the virtual realm so the total area covered by them is as large as required for the simulation. This would, for example, make possible the execution of missions throughout the whole Portuguese continental landmass with just two distributed instances. The second question ends up being tackled with this, as it is expected this will allow a higher number of vehicles and a larger geographical area.

As for the third question, a market study and literature review is conducted to identify what others have done to solve their problems in distributed simulation. This information is taken into account to decide on what is the most adequate solution to the work's needs and is afterwards put into practice.

Another problem that is derived from this thesis, addressed by the last question, is *load balance*, which is how loaded each instance will be in relation to its peers. The thesis will explore the possibility of using an automatic mitigation against it by diverting vehicles to less loaded instances whenever possible.

It is expected that this work will not harm the performance of the platform in any way. In fact, it is expected to improve it in terms of supporting missions covering a larger geographical area and maintaining a higher performance when a lot of vehicle agents are introduced. In short, more agents and larger maps.

1.3 Document Structure

The rest of this document begins, in chapter 2, by presenting some architectures used in distributed simulation that, even though some might have been created in the last few decades, are still used today. It also exemplifies cases where these architectures have been used and analyzes several implementations of the architectures. It then gives alternatives, message-oriented middlewares, and goes on to outline some techniques for temporal synchronization. Finally, some work related to this thesis is presented.

Chapter 3 gives some context to the simulator engine and the platform, argues and chooses the technologies used for the thesis' solution, and introduces the approach to the solution. Chapter 4 reports on all of the solution's details and chapter 5 presents several tests done to validate the implementation in regards to both geographic distribution and temporal synchronization. Chapter 6 explains many improvements that can still be done in different areas of the solution and sums up the document with a few closing remarks.

Chapter 2

State of the Art

For the state of the art, several architectures for distributed simulation were studied in terms of their purpose, architecture, and implementations available. This section has the goal of presenting possible architectures to adopt in this thesis, laying out the unique features and the pros and cons for each promising option while also showcasing some examples of their real-world uses. In addition, it also presents works related to this thesis.

2.1 Distributed Simulation and its Standards

Simulation, in a general sense, aims to imitate the operation of a system with the goal of analyzing complex systems without the costs or infeasibility associated with the operation of their real-life counterparts. This could be the simulation of road traffic flow, microscopic particle physics, or hardware such as an engine or a missile [43] [12] [24].

As these systems become more complex, there might not be enough computational power in a single computer to run the simulation at the desired performance. So, the simulation can be distributed in order to use the computational power of multiple computers. One option is to split the simulation into different areas, assigning each one of them to a single computer solely responsible for simulating the entities in that area. Another option is to distribute the components of a system among different computers, leaving each computer responsible for a car in a road traffic flow simulation [18], or for one of the different features of a missile simulation [13]. Another perspective is that of human-in-the-loop simulations, where people use physical simulators to train their aircraft piloting skills for instance and distributed simulation allows for the communication between these devices [1].

Therefore, distributed simulation holds the key for ever more complex software simulations and here are presented various standards for it. Some are given a shorter treatment and are discarded straightaway while those more suitable for this thesis are discussed in a more elaborate way by their architecture and distinguishing features.

[1]AVES (Air Vehicle Simulator) by DLR (German Aerospace Center) at https://www.dlr.de/ft/en/desktopdefault.aspx/tabid-1387/1915_read-38610/

2.1.1 DIS

Distributed Interactive Simulation was a standard for distributed simulation developed by the United States' Defense Advanced Research Project Agency (DARPA) for real-time wargaming across multiple host computers [48]. Its main building block is the Protocol Data Unit (PDU), a formatted message with simulated state information exchanged between the hosts, of which there are different types arranged into families, among which the *Warfare family*, *Logistics family* and others. These PDU's and all their characteristics are part of the standard, meaning they have specific use cases. It's not possible to, for example, add a field to a PDU and still be compliant with the standard. This renders DIS a rigid architecture for only certain expected purposes, and distributed simulation across multiple FSX instances is certainly not one of them. Although there have been updates to the standard, HLA has come out as a successor to introduce distributed simulation in a more general perspective, and thus DIS isn't worth looking into further in this thesis.

2.1.2 HLA

High-Level Architecture is a standard for distributed simulation. Originally devised by the US government, it is now represented by the IEEE 1516-2010 series of standards [20].

Figure 2.1 presents an overview of the components of an HLA system. It contains one or more *federates*, a *Run-Time Infrastructure* (RTI), and an *Object Model Template* (OMT). The federates are the various cooperating simulators, while the RTI is the interface between them. The OMT is a presentation format and syntax used to describe *Federation Object Models* (FOM), *Simulation Object Models* (SOM) and *Management Object Models* (MOM), defining the object and interaction classes exchanging data. Multiple simulations connected via the RTI using a common FOM are referred to as a *federation*.

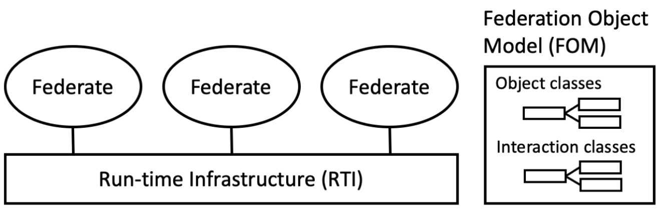


Figure 2.1: Overview of HLA's components (extracted from [56])

The services offered by the RTI are aggregated in the following groups [32]:

- Federation Management: create and destroy federation executions, joining and resigning of federates, pause/resume execution, save/restore execution;
- Declaration management: manages the publisher/subscriber model for information exchange, offering services such as subscribing to an object class attribute;
- Object management: manages the life cycle and message passing for object instances, offering services such as to register/discover an object, send/receive an interaction, and remove object;

- Ownership management: allows attribute ownership to be transferred across instances, offering the services of assume/divest attribute ownership, acquire/release attribute ownership, and notification of ownership changes;
- Time management: coordinates the time advancement of each federate along the federation time axis, offering services such as request time advance, request next event and notification of granting of next event - see further details below;
- Data Distribution management: transmits data between federates, using routing spaces to direct data only to the interested parties;
- Support services: general functionality for joined federates, such as name-to-handle transformation and RTI start-up and shutdown.

HLA has services for time management in the sense of creating an order between events sent by federates. Their purpose is to create a total order between events despite each federate having their own definition of the current time. There are two kinds of events: timestamp-ordered (TSO) or receive-ordered. TSO events will contain a timestamp indicating when they should act, while receive-ordered events can be processed as soon as they arrive. Each federate may be *regulating*, *constrained*, both or neither. For an event to be TSO, it must be indicated as being a TSO event, it must be sent by a regulating node, and it must be received by a constrained node - if these conditions do not hold, the event is receive-ordered.

Further, HLA also has mechanisms in place for save and restore functionality. Any federate can request the system's state to be saved, initiating a state saving procedure over all federates. This state can then be restored in the future if required.

HLA also specifies rules that federations and federates must adhere to. The federation rules are as follows:

- Must have a FOM documented using the OMT
- All object representations occur in the Federates, not in the RTI
- Data exchange between instances of objects in different Federates occurs via the RTI
- Federates must interact with the RTI in accordance with the HLA Interface Specification
- During Federation Execution, an instance attribute may be owned by at most one federate at any given time

And the federate rules are as follows:

- Must have a Simulation Object Model (SOM) documented using the OMT
- Must be able to update/reflect instance attributes and send/receive interactions as specified in their SOM

- Must be able to dynamically transfer/accept ownership of attributes during federation execution as specified in their SOM
- Must be able to vary the conditions under which they provide attribute updates as specified in their SOM
- Must manage their local time in a manner which allows them to coordinate data exchange with other federates

2.1.3 DDS

Data Distribution Service is a standard for data connectivity [42]. Developed by the *Object Management Group* (OMG) since 2001, it is currently still being worked on with version 1.4 released in 2015 [39]. Several vendors are making available their own implementations, with both commercial and open-source alternatives available in the market.

It may not be specifically made for distributed simulation, as it aims at a more general market than distributed simulation proper, but it is also very much related to it. Like HLA, it has a publisher-subscriber model but it's centered around *Topics*. Subscribers subscribe to the topics they're interested in, while publishers send their messages to a desired topic. Thus, this is a model where publishers and subscribers do not necessarily need to know about each other and *plug-and-play* is inherently supported. An overview of DDS is presented in Fig. 2.2. It contains a *DDS Domain*, a network of logical applications that can communicate with each other via their *Data Readers* and *Data Writers* sending and reading messages from *Topics*. Applications must belong to the same domain to communicate with each other.

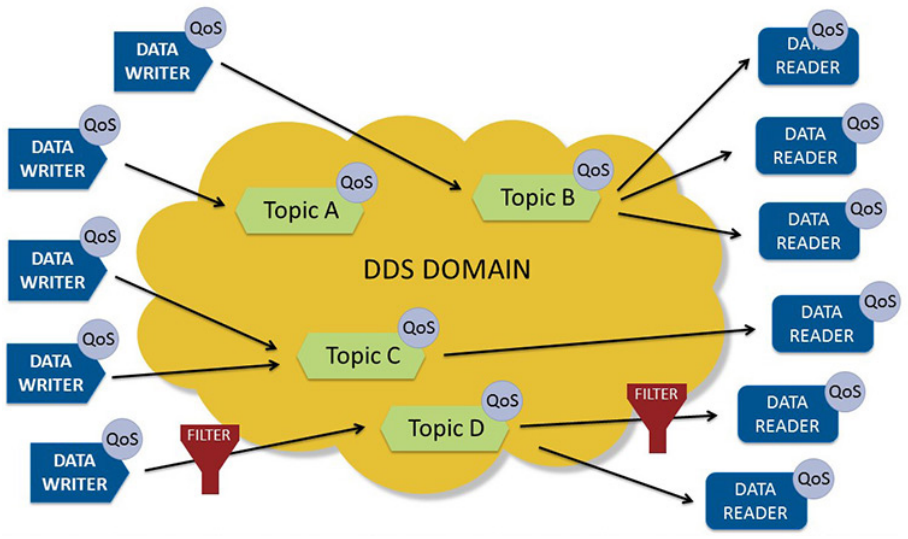


Figure 2.2: DDS overview (extracted from [40])

The standard has a rich set of configurable Quality of Service policies that allow for extensive customization of desired functionality [49]. These policies are presented in Fig. 2.3, divided in groups. These groups are explained in the following list, with a QoS policy example for each:

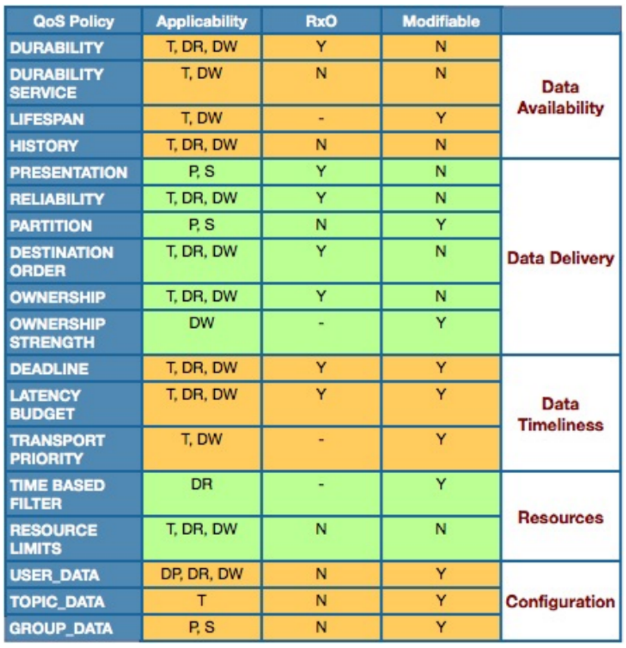
- *Data Availability* controls the availability of data to domain participants. For example, the *LIFESPAN* QoS policy controls the interval of time during which a data sample is valid, with the default value being infinite.

- *Data Delivery* controls how data is delivered and which publishers are allowed to write a specific topic. For example, the *RELIABILITY* QoS policy controls the level of reliability associated with data diffusion, with possible values being *RELIABLE* and *BEST_EFFORT* distribution.

- *Data Timeliness* controls the timeliness properties of distributed data. One such policy is the *TRANSPORT_PRIORITY* policy which allows applications to control the importance associated with a topic or with a topic instance, thus allowing a DDS implementation to prioritize more important data relatively to less important data.

- *Resources* controls the network and computing resources that are essential to meet data dissemination requirements. For example, the *RESOURCE_LIMITS* policy allows applications to control the amount of message buffering performed by a DDS implementation.

- *Configuration*, a less important category, supports the definition and distribution of user-specified bootstrapping information. For example, the *USER_DATA* policy allows applications to associate a sequence of octets to domain participant data readers and data writers. This data is then distributed by means of the *DCPSParticipant* built-in topic. This QoS policy is commonly used to distribute security credentials.



QoS Policy	Applicability	RxO	Modifiable	
DURABILITY	T, DR, DW	Y	N	Data Availability
DURABILITY SERVICE	T, DW	N	N	
LIFESPAN	T, DW	-	Y	
HISTORY	T, DR, DW	N	N	
PRESENTATION	P, S	Y	N	Data Delivery
RELIABILITY	T, DR, DW	Y	N	
PARTITION	P, S	N	Y	
DESTINATION ORDER	T, DR, DW	Y	N	
OWNERSHIP	T, DR, DW	Y	N	
OWNERSHIP STRENGTH	DW	-	Y	
DEADLINE	T, DR, DW	Y	Y	Data Timeliness
LATENCY BUDGET	T, DR, DW	Y	Y	
TRANSPORT PRIORITY	T, DW	-	Y	
TIME BASED FILTER	DR	-	Y	Resources
RESOURCE LIMITS	T, DR, DW	N	N	
USER_DATA	DP, DR, DW	N	Y	Configuration
TOPIC_DATA	T	N	Y	
GROUP_DATA	P, S	N	Y	

Figure 2.3: DDS QoS policies (extracted from [49])

2.1.4 TENA

Test and Training Enabling Architecture [19], developed by the US Department of Defense, aims to unify and reuse the assets used in various test and training range systems which are usually created as *stovepipe* systems, meaning they work independently of each other and thus can lead to duplicated or wasteful effort. Requirement studies for a common T&E (Testing and Evaluation) range infrastructure began in the mid-1990s, with the first release in October 2001. From 2005 to 2016, it has been downloaded more than 10.000 times by its users.

Figure 2.4 depicts an overview of the architecture of TENA. It has at its core a set of software called "TENA Middleware" that performs real-time data exchange between systems with C++, Java, and .NET languages, supporting a wide range of platforms with several versions of Windows since XP, multiple Linux like Fedora and Red Hat, and even embedded devices with Overo Gumstix.

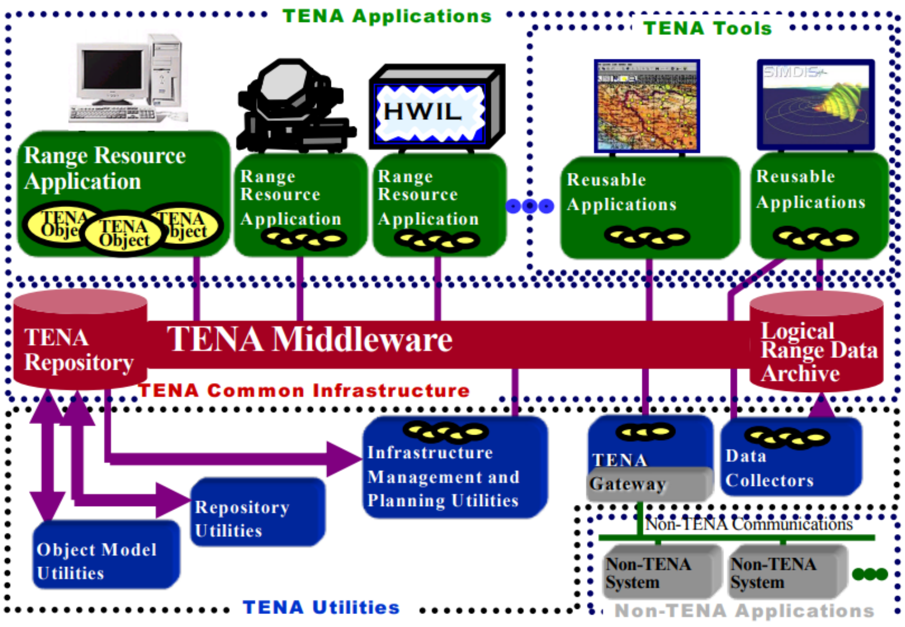


Figure 2.4: TENA architecture overview (extracted from [19])

Figure 2.5 presents a diagram highlighting the kind of systems expected to be interconnected by TENA, such as radar stations, GPS systems, and telemetry displays. TENA has the goal of eliminating all proprietary interfaces from these systems in test and training ranges, while also standardizing various characteristics such as computer/networking hardware, programming languages, and network protocol. These concerns stem from the fact that many ranges have been locked into particular computer vendors or network technologies, while also having problems maintaining code with old languages and old compilers. In addition, many ranges' protocols only support UDP which can cause problems when connecting with external networks.

Since this is an architecture that doesn't have much to do with the context of distributing a virtual simulation by virtual land area and also because the US government still controls whoever has access to it, TENA won't be analyzed further in this thesis.

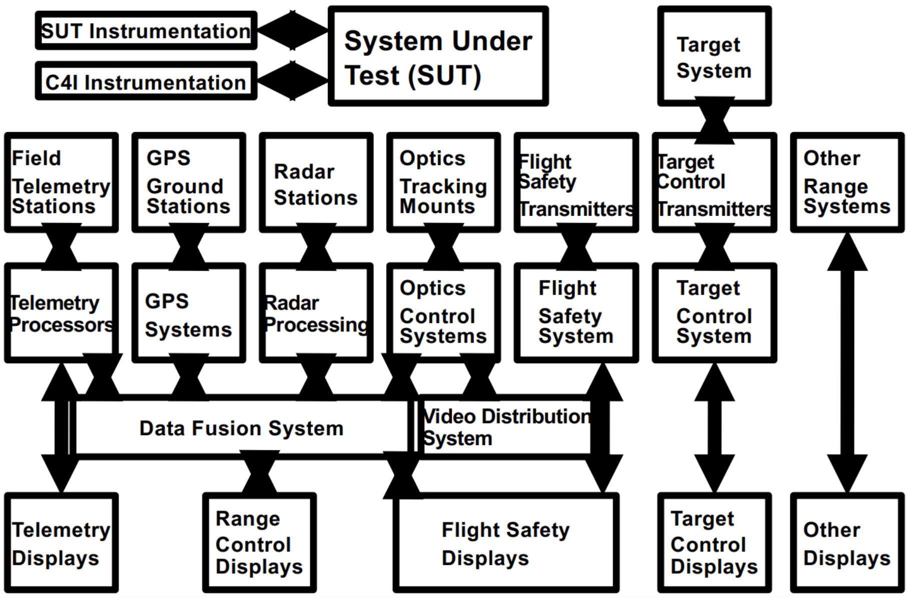


Figure 2.5: Example of systems interconnected by TENA (extracted from [19])

2.1.5 FMI

Functional Mock-up Interface development is currently coordinated by the Modelica Association, which encompasses various organizations such as BOSCH, Siemens PLM, and Daimler AG. The FMI specification defines two use cases: dynamic model exchange and co-simulation [5]. In the former, FMI aims to generate a C code representation of a dynamic system model that can be utilized by other modeling and simulation environments, where models are described by equations. In FSX, the platform's simulation engine, the simulation of the vehicles is done in a closed form, the platform simply reads the positions of the vehicles in FSX in each instant. This use case is therefore not suitable for the platform. In the latter use case of co-simulation, it intends to provide an interface standard for the coupling of simulation tools in a co-simulation environment, where subsystems can communicate with each other and in the time between two communication points each subsystem is solved by its individual solver.

FMI is thus a tool geared towards the distinct field of dynamic systems modeling and therefore is also not analyzed further in this thesis.

2.1.6 Interoperability

There has been a big push for interoperability between different systems, as to uniformize the available tools and minimize costs. Both HLA and DDS standards have been developed to allow for interoperability between different vendor implementations for each standard.

This interoperability is valued to the point where the nations of NATO have agreed for roughly the last decade to apply HLA to all of its new applications in Modelling & Simulation systems as part of the NATO Standardization Agreement (STANAG) 4603 [38], first published in 2008 and updated in 2015.

In 2009, the *DDS Interoperability Demo* was showcased with 3 vendors who developed their own independent implementations of the DDS-RTPS Interoperability Wire Protocol 2.1 specification [9]. The vendors were PrismTech, RTI, and TwinOaks. These are companies already well established in the area with several years of expertise. The demo had 3 goals: show interoperability works, show it along the many dimensions of the standard, and show it doesn't compromise performance. For this, multiple scenarios were tested including the discovery mechanism, different platforms (Windows, Linux), filtering, reliability, robustness to network interruptions, multiple topics and instances, and time and content filters. Since different vendors could come up with a completely interoperable platform, it proves the DDS standard is complete and usable.

2.2 Evaluation of Standards

Several standards were presented in section 2.1. DIS was an important architecture for distributed simulation, but its status as a predecessor to HLA and its rigid architecture disqualifies it from adoption in this thesis. Likewise, TENA and FMI may be promising standards in their own right, but they are not quite suitable for this thesis, not to mention the restrictions of the US government in the case of TENA. This leaves HLA and DDS as the most promising standards for adoption.

Joshi and Castellote present a very complete overview of both HLA and DDS [23]. It presents equivalences between terms and concepts in both, as well as the different features present in one but not the other. Then it goes into a more technical overview of how one can map HLA to DDS, from the perspective of a distributed application developer.

HLA has a few features that DDS doesn't. It does have a state save/restore mechanism, but note it can also be implemented in DDS as detailed later in this section and in section 3.4. Although the platform has no use for it at the current point in time, the situation may change with future work. In addition, HLA has time synchronization mechanisms. Although they may sound appealing, they add significant overhead. The possibility of using them is also detailed in section 3.4.

On the other hand, DDS also has its own set of unique characteristics. DDS has a strongly typed data model, in contrast to HLA's un-typed and un-marshaled approach, which means the DDS developer does not need to worry with low-level details such as how to encode the data to be transmitted. Further, DDS has state propagation semantics and this is how a save/restore functionality could be implemented in the future, as a state representing the full current state of the simulator can be saved by a custom implementation to set all required participants to it at a later time. Access to meta-data is another feature by DDS, allowing any participant to know about events such as a new participant joining a node or the creation of topics.

Hakiri et al. evaluated the performance of HLA and DDS by using PLATSIM, a distributed interactive simulation platform where users interact with each other over publish-subscribe middleware, such as HLA and DDS [17]. The considered application is used for remote education in driving schools. An overview of the tested platform is given in Fig. 2.6. For the HLA side, MAK Real Time RTI was used with 2 federates. For DDS, two participant processes were associated through one topic. Both latency and throughput were the targets of the evaluation, and for that multiple configurations with different packet sizes were tested. Although the paper observes that both HLA and DDS are well suited for real-time distributed applications given the results, the latency of communication in the DDS version is generally better than HLA's. DDS and HLA go from obtaining roughly the same latency at several packet sizes to DDS reaching over 30% lower latency than HLA at packets of roughly 4000 bytes. In terms of throughput, DDS has better performance all around with improvements ranging from 1.14 to 3 times the throughput of HLA with packet sizes between 10 and 5000 bytes.

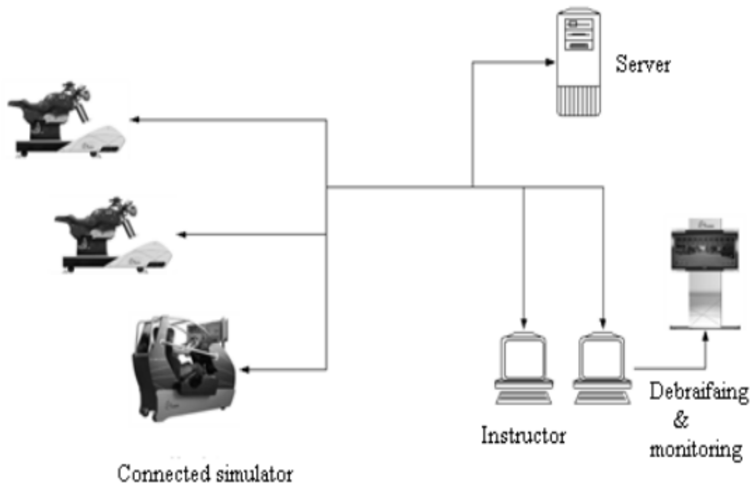


Figure 2.6: PLATSIM hardware testbed (extracted from [17])

2.3 Distributed Simulation Standard Implementations

Several implementations of both HLA and DDS are listed here along with some important characteristics of each: availability (open source, commercial), the supported languages, documentation available, and compliance to the respective standard. A table cell with only a ″-″ means the information is unknown as, for instance, the documentation of a commercial implementation may only be available after the product is acquired. In the case of available documentation, three levels are used: "Full" for the cases where there is lengthy documentation available with code samples and in most cases support from the community; "Some" when the documentation lacks in some aspects with no samples available; and "None".

The various implementations of HLA and DDS are listed in Tables 2.1 and 2.2. The information listed in this section has been retrieved from Wikipedia's RTI entry [55] and the websites of the implementations' vendors. It should be noted the standard implementation must support C#/.NET in order to be considered, as the platform to be applied to is written in C#/.NET.

[2]VT MAK, `https://www.mak.com/products/link/mak-rti`
[3]Loyola Enterprises, `https://www.loyola.com/partners/nads/simware-rti.html`
[4]Pitch Technologies, `http://pitchtechnologies.com/products/prti/`
[5]U.S. RDECOM, `https://apps.dtic.mil/dtic/tr/fulltext/u2/a500177.pdf`
[6]ONERA (Frech Aerospace Lab), `https://savannah.nongnu.org/projects/certi/`

Table 2.1: HLA implementations (adapted from [55])

	Availability	Languages Supported	Documentation	Standard Compliance
CAE RTI	Commercial	C++	-	1.3, IEEE 1516
MÄK RTI [2]	Commercial	C/C++, C#, Java, others	Full	1.3, IEEE 1516-2000, IEEE 1516-2010 (HLA Evolved)
SimWare RTI [3]	Commercial	C++	-	1.3, IEEE 1516-2000
Pitch pRTI [4]	Commercial	C++, Java, Web Services	-	1.3, IEEE 1516-2000, IEEE 1516-2010 (HLA Evolved)
RTI NG Pro	Commercial	C++, Java	-	1.3, IEEE 1516-2000, IEEE 1516-2010 (HLA Evolved)
MATREX RTI [5]	US Govt.	C++, Java	-	1.3
CERTI [6]	Open Source	C++, Fortran90, Java, Matlab, Python	Some	1.3 partial, IEEE 1516 partial
EODiSP HLA [7]	Open Source	Java	Full	IEEE 1516 partial
The Portico Project [8]	Open Source	C++, Java	Full	1.3, IEEE 1516, IEEE-1516e
Open HLA [9]	Open Source	Java	None	1.3, IEEE 1516-2000, IEEE 1516-2010 (HLA Evolved)
OpenRTI [10]	Open Source	C++, Python	Some	Partial for: 1.3, IEEE 1516-2000, IEEE 1516-2010

Table 2.2: DDS implementations

	Availability	Languages Supported	Documentation	Standard Compliance
OpenDDS [11]	Open Source	C++, [.NET]	Full	DDS 1.4
Vortex DDS [12]	Open Source / Commercial	C/C++, C#, Java, JavaScript, CoffeeScript, Scala, others	Full	DDS 1.4
Connext DDS [13]	Commercial	C/C++, C#, others	-	DDS 1.4
GurumDDS [14]	Commercial	C/C++, Java, C#, Python	-	DDS 1.4
InterCOM DDS [15]	Commercial	C++, Java, C#, ADA	-	-
Mil-DDS [16]	Commercial	C++, C#/.NET, Java	-	DDS 1.2
CoreDX DDS [17]	Commercial (source available)	C/C++, C#, Java	Full	-

[7]P&P Software GmbH, https://www.pnp-software.com/eodisp/
[8]Calytrix Technologies, http://www.porticoproject.org/
[9]Michael Newcomb, https://sourceforge.net/projects/ohla/
[10]Mathias Frölich, https://sourceforge.net/p/openrti/wiki/Home/
[11]Object Computing, https://opendds.org/

Note that OpenDDS' .NET support is made available by OpenDDSharp[18], an unofficial .NET wrapper for OpenDDS.

Both standards have a good number of documented, standard-compliant implementations. Roughly half of the HLA implementations are open source while the rest are commercial or restricted, and in DDS the majority of them are commercial. The vast majority of the implementations support C++, with Java as a popular option too, but unfortunately not many support C#. As for HLA, a commercial implementation would have to be considered as there are no open-source alternatives with C# support. However, if DDS is chosen, one is safe to go open source as there are at least two options with C# support which are both documented and fully compliant with the latest version of the standard: OpenDDS and Vortex DDS.

2.4 Distributed Simulation Use Cases

This section presents some use cases and context behind the adoption of HLA and DDS in the industry, defense, and research studies/proposals, as to showcase that these are mature standards used in the real world.

2.4.1 HLA in Defense

According to STANAG 4603, from 2008 and updated in 2015, NATO is applying HLA to its applications in Modelling & Simulation (M&S) systems [38]. In fact, NATO itself has an ongoing so-called "activity", MSG-163, which, among other objectives, aims to actively contribute to the development of the HLA standard [36]. Further, NATO is developing a certification process to verify the HLA compliance of simulation components [37]. It's currently, from 2018 to 2020, in "Initial Operational Capability", where the program is funded by NATO's nations and customers can use and test it for free. It's planned to reach "Full Operational Capability" in 2021, the final release phase in which customers will have to pay to use it.

An example of a long time use of HLA in defense can be found in the VIKING exercises, coordinated by the Swedish Armed Forces [36]. This is an exercise that combines both military and civilian elements such as the police and prison authorities in peacekeeping operations and international crisis management in a fictional country with participating organizations from NATO, European Union, and others. A recent exercise was VIKING 18, conducted in 2018 at nine sites in six countries with around 2500 people participating [52]. A diagram of the system in use at this particular exercise is displayed in Fig. 2.7.

[12]ADLINK Technology Inc., https://www.adlinktech.com/Products/IoT_solutions/Vortex_DDS/Vortex_DDS?lang=en

[13]Real-Time Innovations, https://www.rti.com/products

[14]Gurum Networks, http://www.gurum.cc/?page_id=2129&lang=en

[15]Kongsberg Geospatial, https://www.kongsberggeospatial.com/applications/intercom-dds

[16]MilSOFT, http://www.milsoft.com.tr/en/portfolio/mil-dds/

[17]TwinOaks, http://www.twinoakscomputing.com/coredx

[18]OpenDDSharp is available at https://github.com/jmmorato/openddsharp

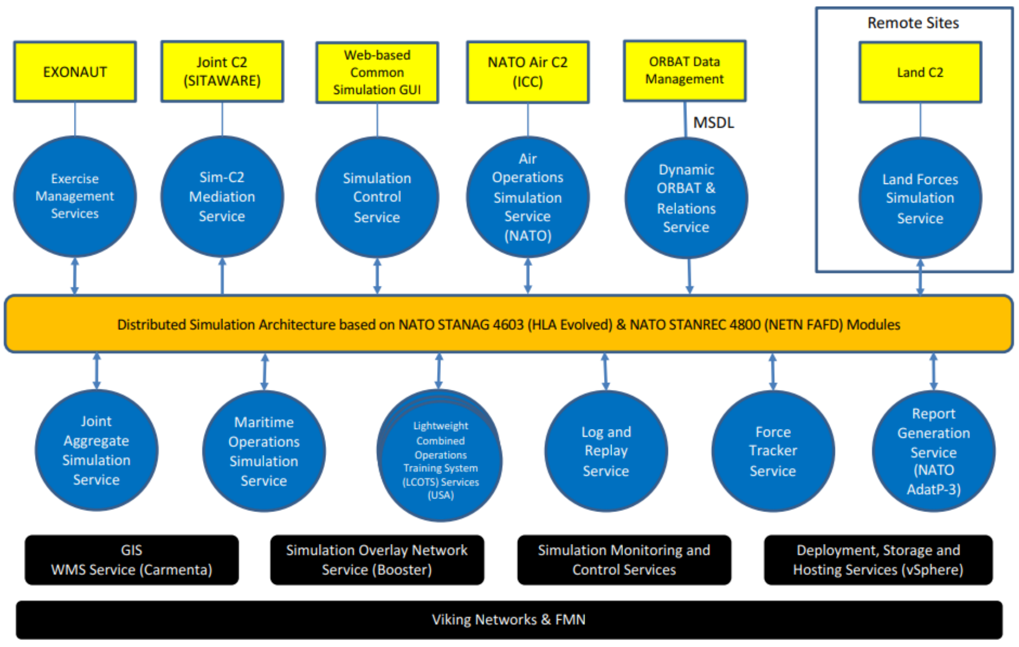

Remote Sites
EXONAUT
Joint C2 (SITAWARE)
Web-based Common Simulation GUI
NATO Air C2 (ICC)
ORBAT Data Management
Land C2
MSDL
Exercise Management Services
Sim-C2 Mediation Service
Simulation Control Service
Air Operations Simulation Service (NATO)
Dynamic ORBAT & Relations Service
Land Forces Simulation Service
Distributed Simulation Architecture based on NATO STANAG 4603 (HLA Evolved) & NATO STANREC 4800 (NETN FAFD) Modules
Joint Aggregate Simulation Service
Maritime Operations Simulation Service
Lightweight Combined Operations Training System (LCOTS) Services (USA)
Log and Replay Service
Force Tracker Service
Report Generation Service (NATO AdatP-3)
GIS WMS Service (Carmenta)
Simulation Overlay Network Service (Booster)
Simulation Monitoring and Control Services
Deployment, Storage and Hosting Services (vSphere)
Viking Networks & FMN

Figure 2.7: VIKING 18 overview of M&S services and federation design with HLA as the integration middleware (extracted from [29])

2.4.2 HLA in the Industry

As far as simulation is concerned, the industry tends to use commercial-off-the-shelf (COTS) packages [51] [6]. These are software packages that abstract the low-level details of simulation. While HLA is heavily used in the defense sector, for instance with NATO nations enforcing its use on all new applications, COTS packages still don't incorporate it fully. This is due to a number of reasons, mainly the time costs and the relative benefit of adoption. The fact that COTS packages don't make HLA available leads to the companies using them to not consider HLA.

Programmers using such simulation packages will have a lot of low-level details abstracted, yet HLA is an extensive standard, one that would require a different way of thinking and dealing with low-level details pertaining to distributed simulation. Thus, the time for companies to incorporate it into their products is regarded as not being worth it.

While in the defense sector HLA is regarded as a transparent abstraction of distributed simulation, in the industry it's perceived as the exact opposite because it views HLA in a more practical, cost-to-benefit approach. They view it as more of an obstacle due to the large number of functions it makes available, turning it to be less intuitive. Also, as already said, HLA is an extensive standard, however, the use cases of companies in general don't need all the functionality HLA provides.

2.4.3 HLA in Research

In 2000, a group at the Goddard Space Flight Center investigated the use of HLA for future NASA missions [46]. A prototype was developed in which, in a virtual setting, two Earth-orbiting spacecraft collect data, a tracking station monitors the two spacecraft, and an "Earth" generates data for the onboard instruments to collect. The group reached favorable conclusions, claiming "[HLA] is solid, well designed, and should have a place within NASA and the wider simulations and modeling community".

In 2018, Gorecki et al. proposed to use Business Process Model and Notation (BPMN) diagrams to model the execution scenario of an HLA distributed simulation of business processes, where a diagram is fed to a master HLA federate [16].

In the context of cloud simulation, in 2011 Zhang et al. investigated the use of HLA [57]. They found HLA to not have any security features so they implemented two protocols on top of it based on HTTPS and public-key infrastructure (PKI) to manage authentication, data authentication, and data confidentiality.

2.4.4 DDS in the Industry

In 2011, Komatsu chose Real-Time Innovation's Connext DDS for their mining machinery, collecting and transmitting data to remote equipment [44]. Thus, a machine can operate by itself inside a mine while transmitting data to and be controlled from the surface.

In 2013, the company operating Canada's airspace, NAV CANADA, announced it had selected Real-Time Innovation's Connext DDS middleware to replace the distribution architecture of its system [3]. DDS is now used to automate and integrate flight information from multiple sources in what is the second-largest airspace by volume of air traffic in the world, after the United States [2].

Volkswagen has been using DDS for the past years to integrate heterogeneous systems of autonomous cars. In V-Charge, a collaborative research project between Bosch, Volkswagen AG, and several universities, researchers investigated driverless valet parking fully automated through the command interface of a mobile phone application [45]. For this, several sensors and the drive control had to be integrated, which meant integrating several computers used by the research partners. For this, *Real-Time Innovation*'s Connext DDS was picked as the integration middleware.

2.4.5 DDS in Defense

Thales Naval, a defense contractor in the Netherlands, signed with PrismTech in 2007 to extend their license of OpenSlice DDS middleware to Thales Naval's worldwide naval operations [34]. As of today, Thales Naval still uses DDS in their products, such as the M-CUBE (Mine Counter Measure Management System) [53].

The US Department of Defense's Missile Defense Agency (MDA) is developing the *Objective Simulation Framework* (OSF) to support MDA's applications related to military simulation [35]. It is expected to be deployed on several laboratories and fielded sites throughout the United States and abroad, including mobile platforms. It incorporates Vortex DDS as its underlying technology to share data among participating simulator elements in real-time [1].

2.4.6 DDS in Research

In 2014, Kim et al. proposed AddSIM-DDS, an integration of distributed simulation through DDS in AddSIM which is a component-based simulation environment that has been developed for weapon system modeling and engagement simulation [26]. The authors describe three possible approaches of integration and analyze an anti-surface ship warfare scenario. The paper reaches positive conclusions and encourages further research on the topic.

In 2016, another study built on top of the previous contribution by using AddSIM-DDS to build a distributed simulation for anti-air missile systems [25]. The authors explain how DDS has a big advantage over HLA in this scenario due to the low latencies of DDS, and proceed to evaluate AddSIM-DDS against an integration of HLA with AddSIM called AddSIM-HLA in this anti-air simulation scenario. AddSIM-DDS came out on top, finishing the simulation faster than its HLA counterpart with results becoming exponentially more distant as the frequency of data updates increases. At the highest frequency of 10 000 updates per unit simulation time, AddSIM-HLA was between ten and twenty times slower than AddSIM-DDS. However, the paper does warn to not take this benchmark as an official HLA vs DDS showdown due to the particular constraints and assumptions of the scenario.

In 2013, NASA investigated the possibility of using DDS for distributed simulation across its local area network with extensibility to other NASA centers and partners, effectively investigating the use of DDS across the cloud [31]. One drawback found was the lack of security features in free implementations of DDS. Although application-level isolation and compartmentalization are provided, data still flows freely. In fact, DDS vendors offer tools that allow a user to join any domain, an isolated set of DDS participants, and view all of its traffic.

2.4.7 Summary

Although HLA is an established IEEE open standard and heavily promoted by defense organizations such as NATO, it doesn't see much interest from the industry. On the other hand, DDS has had widespread adoption not only by the defense sector but also across the industry. Studies in both HLA and DDS raise concerns regarding the feasibility of security in distributed simulations implemented with these architectures.

2.5 General Message-Oriented Middleware

An alternative to using a distributed simulation middleware is to use another more general message-oriented middleware, a more generic approach to communication that is independent of the use case of distributed simulation. A message-oriented middleware is an application-layer hardware or software infrastructure to send and receive messages between distributed systems, taking care of low-level details like supporting multiple operating systems and network protocols. HLA and DDS also fall in this category, however, they are very extensive protocols and have mostly commercial implementations.

A particularly popular example of such a middleware is the Advanced Message Queuing Protocol (AMQP), whose architecture is displayed in Fig. 2.8 and for which there are many implementations. AMQP works in the application layer, like FTP and HTTP. In AMQP, messages sent by producers are passed through a centralized broker which routes them to their indicated message queues, owned by consumers. This allows users of AMQP to not be tightly coupled, they can even communicate from across different programming languages and kinds of devices. It not only supports sending messages to individual consumers but also the broadcast to several, among other patterns. It is a free protocol, contributing to its advancement and availability of multiple implementations.

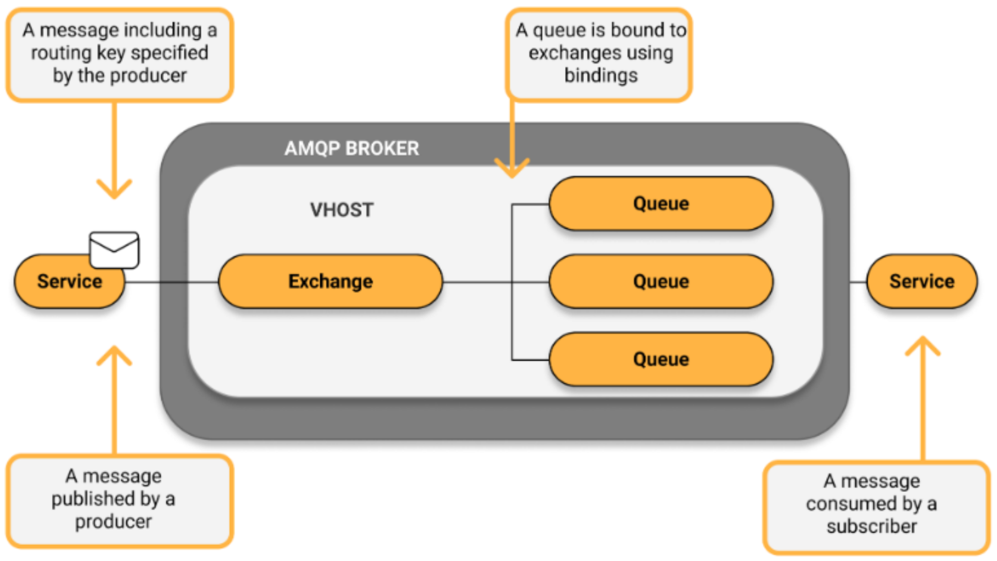

A message including a routing key specified by the producer
A queue is bound to exchanges using bindings
AMQP BROKER
VHOST
Service
Exchange
Queue
Queue
Queue
Service
A message published by a producer
A message consumed by a subscriber

Figure 2.8: AMQP architecture diagram (extracted from [8])

Costa has presented research on message-oriented middlewares in his master's thesis for the purpose of, concurrently with this thesis, improving communications in the same platform this thesis aims to improve [10]. ZeroMQ, Apache Kafka, RabbitMQ, and ActiveMQ are analyzed, going at length on their functionality, security features, availability, maintenance, and real use cases. Also of note, the latter two of these offer implementations of AMQP, among other protocols.

2.6 Temporal Synchronization

Time is a great problem in distributed computing, specifically distributed simulations. If the participating components aren't sufficiently synchronized, simulated entities may fall behind or be ahead-of-time relatively to the global simulation.

2.6.1 Network Time Protocol

The Network Time Protocol (NTP) addresses the need to synchronize multiple computers across local and wide-area TCP/IP networks to Coordinated Universal Time (UTC)[19]. One important distinction is that the protocol does not synchronize computers with each other, it synchronizes all computers to UTC. It relies on client-server exchanges for which there are hierarchy layers, the *stratum* levels. A computer with a dedicated UTC source, such as a GPS transmitter, is categorized as *stratum-0*, while a computer directly connected to it is a *stratum-1*. A computer that receives its time from a *stratum-1* is a *stratum-2* and so on until *stratum-15*, reducing the degree of accuracy per each level. Synchronization is more precise on smaller networks, down to a single millisecond in a local area network and within tens of milliseconds over the internet.

H.A.M. Luiijf and R. van Kampen tested NTP across transoceanic sites: two at Orlando, and one at The Hague [30]. The experiment concluded that only one stratum-1 server with a GPS receiver was needed to synchronize the clocks at various remote sites in a WAN, as long as the WAN connections remained active all the time. If not, a Stratum 1 server was required at each site. As a measure of clock desynchronization, the clock offsets measured between stations were sub-1ms between a Stratum 2 and a Stratum 1 in the same site, below 4ms between a Stratum 2 and a Stratum 1 in different Orlando sites, and mostly within 4ms with occasional spikes up to 10ms between a Stratum 2 in The Hague and a Stratum 1 in Orlando. These results show that NTP can be a very precise synchronization protocol, given the appropriate hardware.

2.6.2 Precision Time Protocol

The Precision Time Protocol (PTP) is a protocol similar to NTP in purpose, but while NTP achieves millisecond-level accuracy, PTP aims for microsecond-level accuracy. Martin Levesque and David Tipper give an overview of the protocol and present synchronization results in their implementation of ptp++, a PTP simulation model [28]. PTP is similar to NTP in technical details, however, not only is it more precise by nature, it adds optional specialized hardware to make it even more precise, such as routers with PTP support. The aforementioned authors obtained results in a variety of circumstances, with synchronization values in the order of the tens of microseconds. Indeed, PTP is much more precise than NTP, however, it requires hardware as costly as NTP's, or more.

2.6.3 "Intolerant" Synchronization

Jafer, Liu, Wainer, and Fujimoto have a comprehensive list of temporal synchronization algorithms for distributed simulation [21] [14]. Both papers classify the presented synchronization algorithms as either conservative or optimistic. Conservative algorithms consist of absolute, solid, intolerant synchronization, where the simulation steps of each participant are interlocked to a certain degree so they are constantly in sync and simulation in multiple computers will be executed just like if it

[19] A concise explanation of NTP is available at https://searchnetworking.techtarget.com/definition/Network-Time-Protocol

was executed in a sequential computer. Optimistic algorithms allow for more flexible constraints, detecting and recovering from eventual out-of-sync violations. One example of such an optimistic algorithm is the Time Warp mechanism [22] which rolls back a participant's state when an event with an older timestamp is received, reprocessing all events in timestamp order. This means the algorithm requires a state saving functionality so it has checkpoints to revert to, which demands a continuously increasing usage of memory. These optimistic algorithms generally have these extra requirements because, although they may allow simulation flaws to pass, the end result is still a perfect synchronization of simulation participants.

2.7 Related Work

A proof of concept has already been successfully implemented at LIACC [47]. The approach consisted of two instances of FSX, far enough from each other on the virtual map, in their own virtual machines with a migration protocol in place. Once an airplane traveling from one instance's zone to the other's reached the common area between instances, the migration protocol would remove it from the original instance and instantiate it in the next one with the same state: speed, altitude, attitude, etc. It was observed this transition took place smoothly, and the resulting flight path was taken just as planned.

As part of an open-source project called *VirtualAir*, HLA plugins have been developed for FSX, FlightGear and X-Plane [27]. These plugins have been demonstrated [20] connecting FSX and FlightGear, where the input to one simulator's plane would result in the other simulator mimicking the movement. The implementation used was CERTI, an open-source implementation of the HLA RTI [41]. This proves it is possible to use HLA with FSX. The source code has been made available in Sourceforge [21].

2.8 Summary

The study of the state of the art found and filtered several standards related to distributed simulation. Section 2.1 described them, discarding the less adequate ones straight away (DIS, TENA, FMI) and picking HLA and DDS while going more in-depth in their respective architectures. Section 2.2 continued this effort by pitting HLA against DDS in terms of both their similar and distinguishing features.

As they're only standards, vendors offer their own implementations of the standards. Several of these were listed in section 2.3 while presenting some of their important characteristics such as the documentation available and supported languages. From this, it's concluded DDS has a couple of suitable open-source implementations while a commercial alternative would need to be considered in HLA.

[20] Video available at `https://youtu.be/JUzt5cON4lg`
[21] Code available at `https://sourceforge.net/p/virtualair/code/HEAD/tree/`

To investigate the maturity of these technologies, several use cases of HLA and DDS from different categories were found in section 2.4. It just so happens that HLA has virtually no uses in the industry although DDS has plenty of popularity, and both are used extensively in defense. Both also share attention in research efforts, in which studies were found raising concerns about the lack of security features in both standards' implementations.

Section 2.5 presents alternatives to these standards: message-oriented middlewares. These are simpler, more generic middlewares that only handle the transmission of messages. On a different subject, section 2.6 talks about techniques to temporal synchronization, which is essential in distributed simulation.

Finally, relevant work related to this thesis is presented in section 2.7. Most important is the proof of concept implemented at LIACC for precisely the work done in this thesis [47], although the integration of HLA with FSX and other flight simulators can also be a helpful source [27].

Chapter 3

Approach

This chapter aims to introduce the thesis' approach to solve the problem. It begins by presenting a rough overview of FSX in section 3.1 and then outlining the current state of the platform in section 3.2. It then lists a number of requirements for the new platform in section 3.3 and capitalizes on some of them by choosing the technologies to be used in section 3.4. It finally goes on in section 3.5 to give a high-level view of the implemented solution, which is drilled down on chapter 4.

3.1 Microsoft Flight Simulator X

FSX is a flight simulator developed by Microsoft and released in 2006, greatly expanded throughout the years by both its developers and its large modding community[1]. The game features hundreds of different flyable planes in great detail and is capable of simulating flight dynamics to a realistic scale. It has a realistic global map with satellite imagery and real-life airports. FSX runs only on Windows.

It has an API called *SimConnect* for data retrieval and modification [33]. This has allowed the development of simulator machinery such as displays mimicking real aircraft displays or gadgets allowing for GPS navigation[2]. It is also thanks to this API that the simulation platform can instantiate the vehicles of the simulated agents and control them. There are libraries available for C++, C#, and, to a lesser extent, Visual Basic.

SimConnect can connect to a FSX instance running in either the same or a remote computer by passing the respective *configuration index*, a concept that is present in multiple of the following sections. *SimConnect* reads a local file named *SimConnect.cfg*[3] in the *Documents* folder of Windows to associate the *configuration index* to the respective FSX instance. The file is divided in numbered sections directly related to the *configuration indexes*, each one with an IP, Port, and

[1]A significant quantity of addons for FSX is available for download at `https://flyawaysimulation.com/`

[2]For example, RealSimGear sells such accessories at `https://realsimgear.com/`

[3]The file syntax is available in the following URL. Despite that Prepar3D is a separate product, the syntax of *SimConnect.cfg* is the same. `https://www.prepar3d.com/SDKv4/sdk/simconnect_api/configuration_files/configuration_files_overview.html#SimConnectClientConfiguration`

other information associated. A strange, non-documented behavior was found in which *configuration index* 0 was always the FSX instance in *localhost*, no matter what IP and Port were written in section 0 of the *SimConnect.cfg* file.

3.2 The Platform

The goal of the platform is to simulate vehicles that coordinate the execution of missions, serving as a study basis for different mission strategies. There are several kinds of missions intended for this platform, such as fire fighting, search and rescue, pollution detection, detection of illegal activities, and transportation. With it, simply a set of waypoints or maneuvers can be provided to FSX and it'll simulate the movement of the vehicles to satisfy them, freeing the developed platform components from having to deal with this kind of physics calculations.

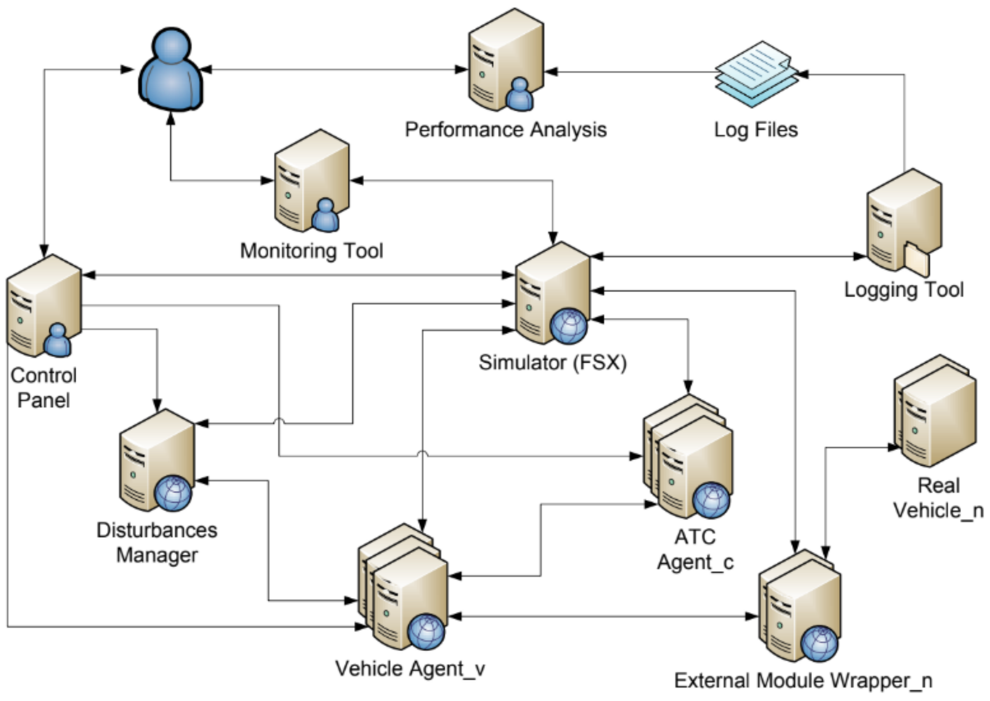


Figure 3.1: Original platform architecture (extracted from [50])

The current state of the platform is presented in Fig. 3.1. It contains the simulator, FSX, at its core and builds additional features on top. The main interface component is the Control Panel, allowing to configure the scenario, teams, disturbances, and missions. The two other most relevant modules are the ATC Agent and Vehicle Agent. The Vehicle Agent is connected to the vehicle inside FSX, managing its operations, while the ATC agent controls the airspace of an airport or another specific zone, effectively controlling the vehicles inside it by communicating with them via messages. It contains a Disturbances Manager, allowing for the simulation of environmental events the core simulator is incapable of, such as a chemical in the air forcing vehicles to react [4]. In addition, the platform provides an extensive Monitoring Tool providing features such as visual feedback of the state of the simulation, a list of all the vehicles present, and individual monitoring of vehicle state. Not visible in the diagram is the communications middleware Agent Service [54]. Originally, Agent Service was a centralized middleware responsible for the transmission of messages between components and *yellow pages* agent registration services.

Different paradigms of the platform have also been changed and improved in parallel with this thesis. Costa had to improve the platform's communication middleware while introducing security mechanisms [10]. This resulted in the adoption of RabbitMQ with its AMQP implementation to replace Agent Service's message transmission, while keeping Agent Service's *yellow pages* services. He also allowed the different components of the platform to run in different computers, as originally intended, with the introduction of Simulation Nodes, placing one in each computer. A Simulation Node is an entity responsible for receiving certain messages and acting on them, allowing the Control Panel to request any computer to start executing a component of the platform such as a vehicle agent. Concurrently, Damasceno improved the Disturbances Manager by integrating an external simulator of environmental disturbances into the platform [11]. FSX, despite being the world simulation engine of the platform, does not actually simulate environmental disturbances, which is where the Disturbances Manager comes in. With Damasceno's work, the manager now has the capability of simulating the spread of fire by integrating the manager with an external simulator that does exactly that. And Carvalho had to introduce the actual execution of many missions in order to achieve his main goal of collecting and analyzing data from simulations in the platform [7].

The platform's modules are nearly completely written in C#, and therefore the communication with FSX is handled via the *SimConnect* C# API. This is an object-oriented API in which communication with an FSX instance is handled by a *SimConnect* object. It allows for bidirectional communication in the sense that data such as the local time of FSX can be set and data such as a vehicle's position can be retrieved.

3.3 Requirements

The one FSX instance is to be replaced by possibly many instances, each one with a distinct area of simulation. The user must be able to configure the new instances and analyze the simulation state both globally and in a per instance basis, however, the functioning of the distributed simulation should be transparent as to not have to deal with low-level details. For example, the user should not have to specify in which instance a vehicle is to be placed in, instead the placement of vehicles should function exactly as it does originally.

Henceforth, the requirements for the new platform architecture are:

1. Support one or more instances of FSX

2. All instances must be synchronized in terms of simulation time, weather conditions, and other aspects of the simulation

3. Details of distributed simulation should be abstracted away as much as possible

4. Configuration should have virtually the same complexity as originally when there is only one FSX instance

5. Extra complexity in configuration should only come from setting the number of FSX instances and locations of each

6. User should be warned in the control panel when the planned vehicle trajectories cross zones outside of the instances' simulation area

7. User can monitor the global state of the simulation as originally possible, without taking into account the different FSX instances

8. User can monitor the local state of the simulation in each FSX instance

9. Vehicles should be load balanced between instances whenever possible

3.4 Technological Choices

Having a distributed simulation standard not only simplifies the implementation from a programming perspective, but it also standardizes the communication between several entities of the platform and hence it becomes more modular. The FSX instances become participants - or *federates* in HLA terms - and communication with them would require the definition of the types of objects to be exchanged, and later, in the light of future work, these objects could be extended or modified. In addition, the remaining entities of the platform like the disturbances manager and logging mechanisms might also become compliant with the distributed simulation standard. Therefore, using such a standard helps to pave the way for future work on a more modular platform, theoretically. However, a more general message-oriented middleware also simplifies the programming perspective while making the implementation more loosely coupled with the chosen technologies. Ultimately, RabbitMQ was chosen, a general message-oriented middleware, and this section explains the reasoning behind it. In the initial phase of this thesis, research was conducted to determine the best standard among the distributed simulation standards and the resulting arguments are presented first.

Requirement 2 calls for all federates to have the same global weather conditions and the same time of day. At simulation start, these are settings that can be set to the same values, however, the simulation in each participant may run at different speeds which would lead to the divergence of these values. To solve this, it is possible to set different simulation rates and to pause an FSX instance so others can catch up. Here should also be noted that FSX's simulation rate is not to go above 4, as higher values may lead to abnormal behavior with SimConnect and loss of precision with the simulation[4]. With a bound on the simulation rate, so is bounded the divergence of simulation states and thus it may not be needed to keep the FSX instances synchronized at all times, instead relying on specific instants or periods of time as synchronization periods as to have

[4]https://www.fsdeveloper.com/forum/threads/simconnect-simulation-rate.165450/

a lower overhead leading to superior performance. So, while it may be handy to have builtin time management services such as HLA's to synchronize this effort, there isn't a need for a very strict model of event ordering. An application-level approach can be taken to introduce an adequate time management service to the system to make up for the lack of such services in DDS.

One other advantage of HLA over DDS is the save and restore feature, where the global state of the system can be saved to be restored to at a later time. At the present moment, the platform has no need for such a feature; however future work could provide for some interesting use cases. It would not be too complicated to introduce such a feature with DDS, as Joshi and Castellote describe [23]: "an application can specifically create a SaveRestoreTopic or a SynchronizationPointTopic, and require all participants subscribe to them. The QoS policies on these topics can be set to ensure the correct operational behavior semantics, e.g. reliable and ordered". Seeing as a save and restore functionality is not immediately needed and it can be done in the future with DDS, this is not a strong point for choosing HLA.

As claimed in section 2.2, DDS is advertised to have better performance and thus be more suitable for real-time systems. This can make a big difference in the scalability of the platform, so it's a strong point for DDS.

Another advantage of using DDS is shown in section 2.3, where two alternatives of open source DDS implementations are presented. If HLA was picked, the platform would have to be locked to a commercial, typically expensive, implementation. It is also of note that studies comparing the different implementations don't tend to be carried out, so a lengthy analysis of the several commercial implementations would be needed to avoid the costly decision of an inadequate implementation.

The aforementioned reasons make DDS a winner in its category, however, there are strong cons to its choice. Sections 2.4.3 and 2.4.6 point to the lack of security features in the free implementations of the standard, which is in direct opposition to the requirements of the work being conducted in parallel by Costa [10]. An application-level approach to introduce security would be possible, but this would add a lot of complexity and maintenance costs to the solution. Despite this problem, the open-source *OpenSplice DDS* was experimented with in order to analyze its feasibility but the lack of proper documentation and support even for basic features proved to be a fundamental problem if it were to be adopted. In addition, Damasceno's concurrent thesis ideally required a middleware that had interfaces not only for the platform's C# but also for Python, due to his integration with an external simulator, but the differences in both these interfaces for *OpenSplice DDS* were vast and also proved to be very problematic [11].

The previous problems make the general message-oriented middlewares a lot more promising, taking into account from the get-go that all of them have built-in security features. Costa conducted extensive tests of RabbitMQ, ActiveMQ, ZeroMQ, and Apache Kafka, with and without their security features [10] and found RabbitMQ to be the overall winner in performance, ease-of-use, and documentation. All of these are free middlewares that are still actively supported by their developers. Additionally, RabbitMQ offers similar interfaces for both C# and Python, going in line with Damasceno's work [11].

3.5 Architecture

Having chosen the technological backbone in the previous section, the proposed solution is laid out as follows. For the simulation setup, the Control Panel is put in charge of calculating and assigning the areas to the FSX instances that are to simulate them (Fig. 3.2). The details of this area distribution between the FSX instances are detailed in section 4.1 while the required modifications to the Control Panel are in section 4.4.1.

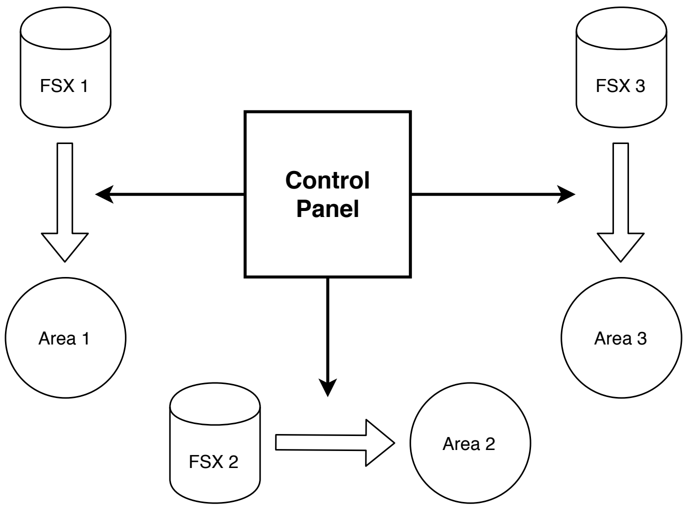


Figure 3.2: Representation of the assignment of areas to FSX instances

To manage the distribution simulation as it executes, the Distribution Agent has been introduced to the platform. As Fig. 3.3 suggests, each agent is connected to one FSX instance and it has the responsibility of handling the migration of vehicles by communicating with each other. The migration algorithm implemented by the agent is explained in section 4.2. The agents also manage the temporal synchronization among the FSX instances (Fig. 3.4), communicating with each other to pass on a specific *master* FSX instance's time to all the other instances via an algorithm explained in section 4.2. All the technical details of the Distribution Agent are in section 4.4.2.

The Control Panel and Distribution Agent are the main modifications to the platform, however, the other components require changes as well. The ATC agent needs to be able to connect to the possibly many instances of FSX whose simulation area overlap with the ATC's, and the vehicle agent needs to be able to migrate between FSX instances, that is, it must be able to disconnect from an instance and connect to another once ordered by a message from the Distribution Agent. All of this is reported in chapter 4, with additional modifications to be done in the future described in chapter 6.

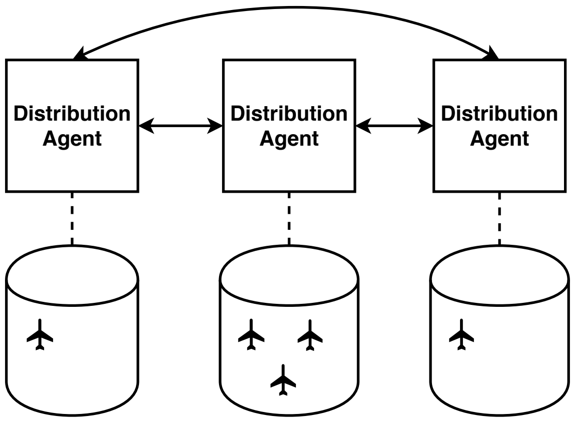


Figure 3.3: Representation of the distribution agent's responsibility in migrating vehicles

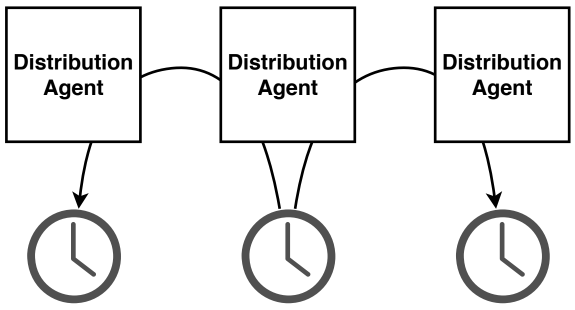


Figure 3.4: Representation of the distribution agent's responsibility in synchronizing the FSX clocks

Chapter 4

Implementation

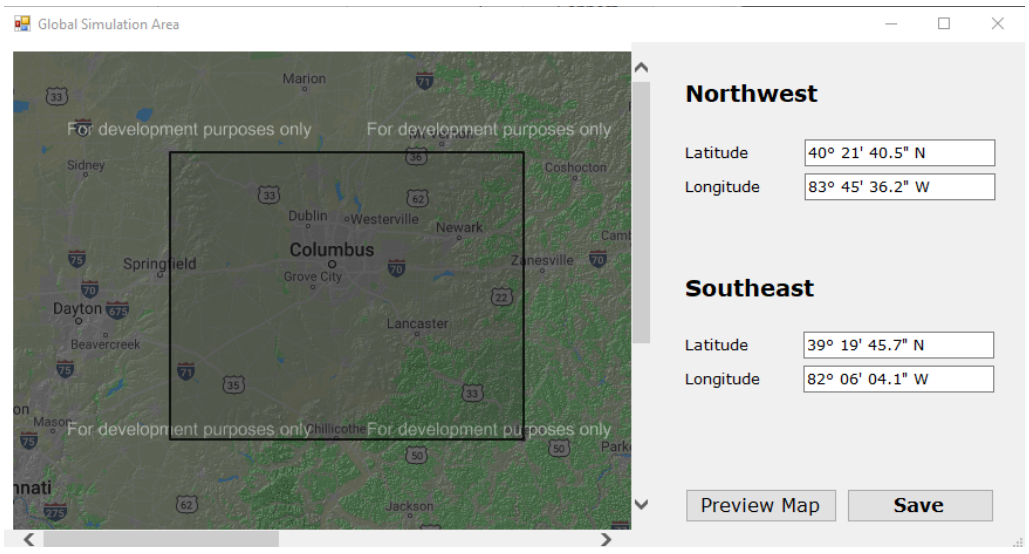
This chapter gives all the details of all the modifications to the platform resultant from the work in this thesis to approach the problems of geographic distribution and temporal synchronization. It first outlines the algorithms in a more platform-independent perspective and then describes their concrete implementation in the platform's components.

4.1 Geographic Distribution

As already stated, each FSX instance will simulate a different area in a circular shape. The radius for such shapes is at most 200Km, a limit set by FSX. These areas may overlap, with the goal of forming a global simulation area larger than any individual simulation. The global simulation area is input by the user to form a rectangular area (Fig. 4.1), followed by the automatic calculation of the individual simulation areas (Fig. 4.2). This calculation is implemented via circular meshing, filling the entirety of the global rectangular area with overlapping circles with centers arranged in a two-dimensional grid. Although this was the chosen implementation, this geographic distribution is modular and can be swapped out to another, possibly more intelligent distribution of instances to take into account factors such as load balancing.

Circular meshing takes as input the global rectangular area A. The radius for the instances will be a constant R = 200Km, and as per circular meshing the instances will be aligned in a two-dimensional matrix with $\sqrt{2} * R$ kilometers between columns and lines. First, the validity of the chosen configuration is tested, as there may not be enough instances available to fill the global area. There must be at least $\left\lceil \frac{\Delta_{longitude}}{\sqrt{2}*R} \right\rceil * \left\lceil \frac{\Delta_{latitude}}{\sqrt{2}*R} \right\rceil$ instances present, where $\Delta_{longitude}$ and $\Delta_{latitude}$ are respectively the longitudinal and latitudinal lengths of the chosen global simulation area. Then the global area is iterated latitudinally and longitudinally, initiating a connection to a different FSX each time.

A number of improvements can be made in different areas of this geographic distribution and are enumerated in section 6.2.1.



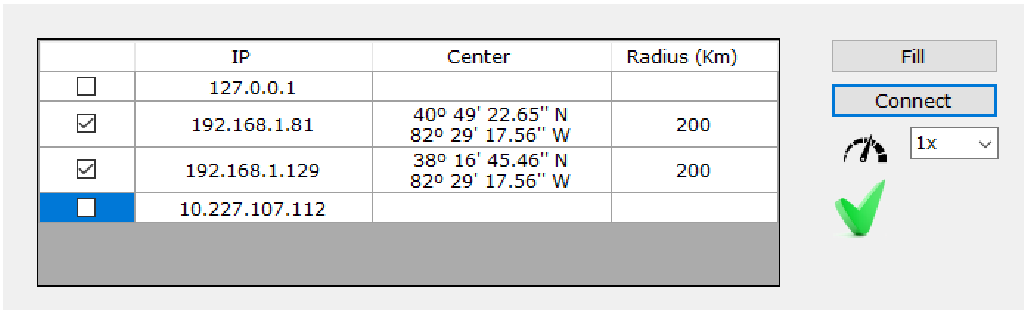
Northwest

Latitude	40° 21' 40.5" N
Longitude	83° 45' 36.2" W

Southeast

Latitude	39° 19' 45.7" N
Longitude	82° 06' 04.1" W

Figure 4.1: User input for the global simulation area



	IP	Center	Radius (Km)
☐	127.0.0.1		
☑	192.168.1.81	40º 49' 22.65" N 82º 29' 17.56" W	200
☑	192.168.1.129	38º 16' 45.46" N 82º 29' 17.56" W	200
☐	10.227.107.112		

Figure 4.2: Implemented UI for the end result of the calculation of the area assigned to each instance

4.2 Migration Protocols

With multiple simulations contributing to a whole, vehicles need to be able to transfer between instances seamlessly. This is most important when vehicles approach the border of the instances they are in but also when it's possible to offload vehicles from an overloaded instance. When a vehicle migrates, it must cease existing in its current instance and resume all of its operations in its new instances with the same state: same velocity, same altitude, same position, and other such variables. This is a straightforward concept, however, the real challenge is in deciding when and where to migrate what vehicles. For this, two distributed protocols were defined and implemented in the platform, running periodically in the new distribution agent (described in section 4.4.2) associated with each instance. These are the Border Protocol and the Load Balancing Protocol, handling the migration of vehicles in two different sets of circumstances, and both need common functionality before and after their execution.

4.2.1 Common Functionality

Although both protocols have different goals, they share a significant amount of common logic. They both require the calculation of the remaining times in the instance's area, and they both migrate vehicles. Both also run periodically, starting an iteration every 5 seconds of simulation time.

First, the algorithms require an estimation of the remaining time the vehicles have until they reach the boundary of the instance's area. For this estimation, it is assumed they keep their current heading, pitch, and velocity indefinitely. As the vehicles travel between waypoints, another approach would have been to estimate based on the heading to the next waypoint. However, this could lead to vehicles dangerously approaching the border or even bypassing it in some cases. Take for instance Fig. 4.3, where a big airliner reaches waypoint A and now has to turn to B. It requires a significant distance to be able to turn, needing to migrate to a more suitable instance, but the estimation of the time inside the instance will claim it still has plenty of space. Instead, if the estimation takes only the current heading into account, this problem is prevented as the border protocol will take action to migrate the vehicle to a more suitable instance, if available.

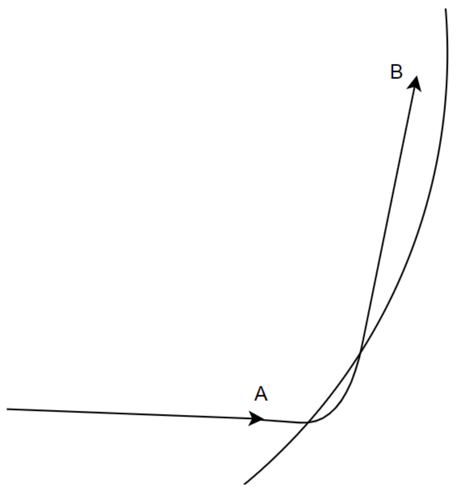


Figure 4.3: Case where a vehicle can not turn hard enough to keep within the same instance

Taking this time data into account, the protocols will take action to decide which vehicles to migrate to what instances as detailed in sections 4.2.2 and 4.2.3. After this, the migration phase kicks in. It consists of removing the vehicle from the previous instance and creating it in the new one with all the same data whose variables are listed in Table 4.1. However, these variables aren't enough for a complete migration, and extra care must be taken due to the multithreaded nature of the vehicle agent, as detailed in section 4.4.4.

4.2.2 Border Protocol

The border protocol is meant to decide, periodically, which vehicles have to migrate to what instances so they never travel out of their instances' simulation areas. Taking into account the estimation of the remaining time for each vehicle in its instance's area, calculated beforehand, the

Table 4.1: Vehicle migration parameters

Param.	Type	Observations
Vehicle model	*string*	-
Latitude	*double*	-
Longitude	*double*	-
Altitude	*double*	-
Pitch	*double*	-
Bank	*double*	-
Heading	*double*	-
Airspeed	*uint*	-
On ground	*uint*	Boolean indicating if vehicle is landed
Tail number	*string*	Only for planes
Waypoints	*List of Waypoints*	List of all waypoints in the vehicle's trajectory, including past ones
Current waypoint	*int*	Index of the next waypoint in the waypoints list

decision of which instance to migrate to, if any, derives mainly from the partition of vehicles by remaining time thresholds. This is to be sure the migration is worthwhile in the sense of spending more time in the new instance than it would if it didn't migrate. The thresholds are defined as follows, measured in simulation time:

- $T_{Light} = 20$ seconds
- $T_{Moderate} = 10$ seconds
- $T_{Critical} = 5$ seconds
- $\Delta_{Light} = 10$ seconds
- $\Delta_{Moderate} = 10$ seconds

So, a new instance will only be chosen for migration if it obeys the logic in Fig. 4.4; let t_{orig} and t_{new} respectively be the remaining times in the original instance and in a new instance considered for migration. Most vehicles will migrate based on the T_{Light} and $T_{Moderate}$ thresholds to instances whose new remaining times are minimally bounded to be considerably higher. The $T_{Critical}$ threshold is only meant as a fail-safe to guarantee vehicles always have a chance to migrate. The values above have not been obtained from a scientific analysis, they were chosen simply due to being seemingly reasonable and could be further tweaked in the future. Likewise for the 3 temporal levels (*Light*, *Moderate*, and *Critical*), which are an attempt at forcing migrations of vehicles to instances where they still have enough time until they hit the new border. So, for example, if a vehicle is estimated to reach the border in 19 seconds, it can only migrate to instances where it'd take over 30 seconds to reach the new border.

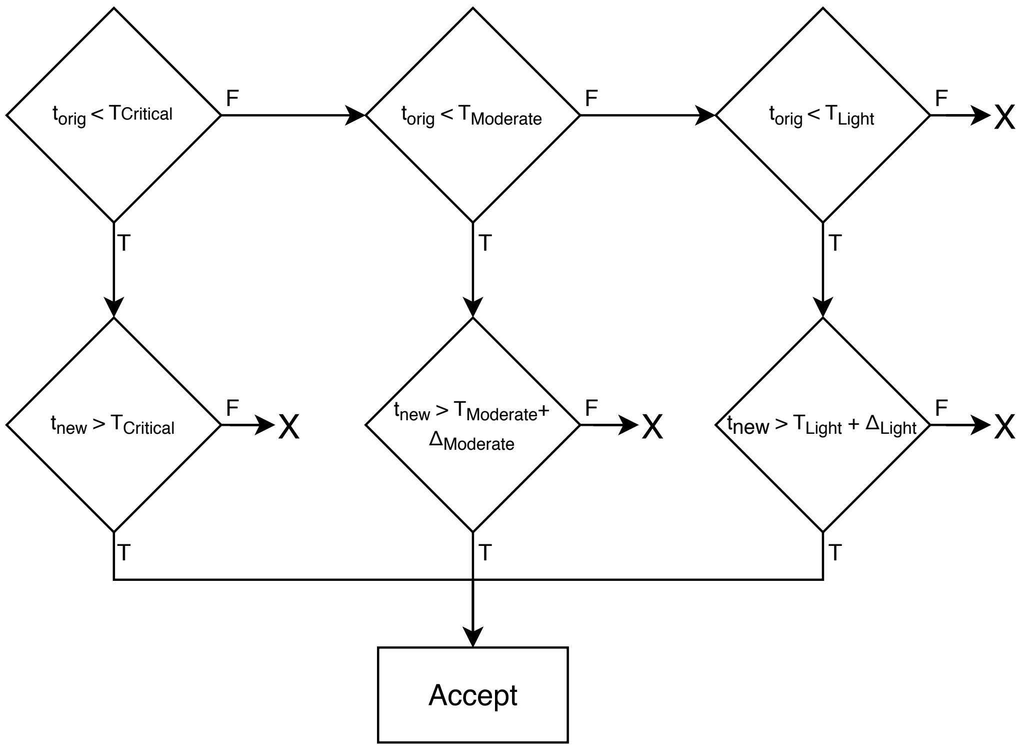

$t_{orig} < T_{Critical}$
F
T
$t_{orig} < T_{Moderate}$
F
T
$t_{orig} < T_{Light}$
F
X
T
$t_{new} > T_{Critical}$
F
X
T
$t_{new} > T_{Moderate} + \Delta_{Moderate}$
F
X
T
$t_{new} > T_{Light} + \Delta_{Light}$
F
X
T
Accept

Figure 4.4: Border protocol's instance acceptance diagram

This defines the migration acceptance criteria, so now the communication aspect between instances needs to be clarified. Figure 4.5 presents a flow diagram for the communication between two instances.

The first message is a broadcast from each instance to all of its neighboring instances. It broadcasts a list with all the relevant data of its vehicles for which $t_{orig} \leq T_{Light}$, the data being t_{orig}, position coordinates, altitude, velocity, heading, and pitch, in the case of an airplane. Other kinds of vehicles require slightly different data, such as the replacement of altitude by depth in the case of a submarine. On reception of this broadcast, each instance verifies if it can accept the vehicles with the data it receives. It will verify if the vehicles are inside its own simulation area and calculate the new remaining time, t_{new}, in order to verify if it is well inside its borders as per the acceptance diagram of Fig. 4.4. From this criteria, it will form a subset of vehicles from the original received list and send a list with its IDs back to the sender instance. In the case of several instances accepting the same vehicles, the original instance accepts the first and ignores the following. This means the protocol is not optimal, but the alternative would have been to wait for a better migration opportunity which would add significant complexity while demanding more hardware resources; the solution found here is deemed to be good enough, and the vehicles will gradually find a proper instance to be simulated in. The original instance now knows which vehicles to migrate, and the algorithm moves on from the decision phase realized by the border protocol.

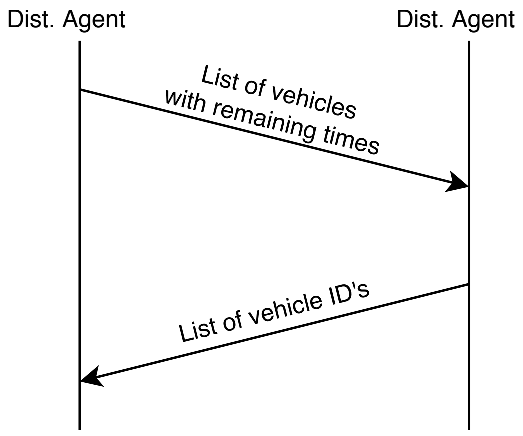


Figure 4.5: Border protocol's communication flow diagram

4.2.3 Load Balancing Protocol

The load balancing protocol is also meant to decide periodically which vehicles to migrate to what instances, but it instead migrates those well within their instances' borders to improve load balancing between instances. The load balancing achieved will be gradual in each step, migrating vehicles to less loaded instances albeit not necessarily the optimal ones. Each instance takes two parameters solely for this protocol's execution: the relative load allowed, r_l, and the load difference allowed, d_l. In the current implementation, these values were chosen to be $r_l = 1.5$ and $d_l = 1$, which means that any instance is allowed to have up to 1.5 times the load that this instance has, and that any other instance is allowed to have 1 more vehicle than this instance; "allowed to" meaning vehicles aren't migrated within those limits. This is to avoid cases where vehicles could be caught in a loop of migrating back and forth between instances, and also to reduce the number of unnecessary migrations. Like in the temporal thresholds of the border protocol, these values were chosen because they made sense and were not the object of a scientific study. So, an instance is only allowed to take in new vehicles from another instance when it satisfies the constraints 4.1 and 4.2. Let l_{sender} and l_{self} be the number of vehicles in the sender instance and the number of vehicles in the current instance, respectively.

$$r_l < \begin{cases} \frac{l_{sender}}{l_{self}}, & \text{if } l_{self} > 0 \\ l_{sender}, & \text{otherwise} \end{cases} \tag{4.1}$$

$$d_l < l_{sender} - l_{self} \tag{4.2}$$

Like the border protocol in section 4.2.2, it takes into account the estimation of the remaining time for each vehicle in its instance's area, calculated beforehand, and it has a similar communication diagram as shown in Fig. 4.6. It also broadcasts the first message with a list of vehicle data, but now these are the vehicles for which $t_{orig} \geq T_{Light}$, and the data does not include t_{old}. In addition to the list, the broadcast includes the number of vehicles the instance is simulating. The recipient instance will check if the constraints 4.1 and 4.2 are satisfied, and if so it will pick a subset of vehicles that are inside the new instance's simulation area, as before, but now must have $t_{new} > T_{Light}$. The subset has at most a size of $\lfloor (l_{sender} - l_{self})/2 \rfloor$ vehicles, prioritizing those with the highest t_{new}. If the subset has at least 1 vehicle, a list with the IDs of the vehicles in it will be replied to the original instance signaling that it accepts to take these vehicles; additionally, the protocol in this instance will cease accepting new vehicles for the next 10 seconds in simulation time. In the case of several instances accepting the same vehicles, the original instance accepts the first and ignores the following. Like in the Border Protocol, this means the protocol is not optimal, but the alternative would have been to wait for a better migration opportunity which would add significant complexity while demanding more hardware resources; the solution found here is deemed to be good enough, and the vehicles will gradually find a proper instance to be simulated in. The original instance now knows which vehicles to migrate, and the algorithm moves on from the decision phase realized by the load balancing protocol.

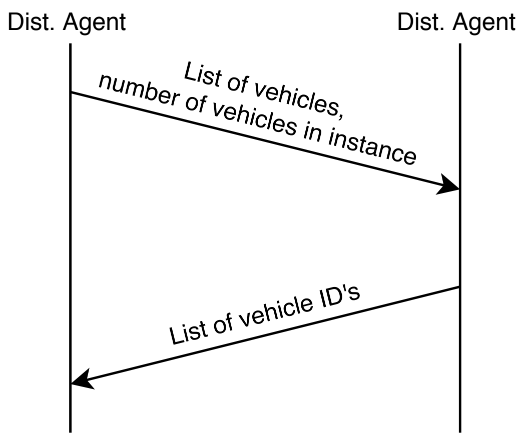


Figure 4.6: Load balancing protocol's communication flow diagram

4.3 Temporal Synchronization

FSX, as the platform's simulation engine, has its own internal time. Running multiple FSX instances independently introduces the danger of their clocks diverging due to either individual slowdowns during the simulation or offsets from the start of each simulation, that is, not every simulation starts at the same real-world time instant. This leads to errors in the simulation because the simulated entities are dependent on the internal time. If an FSX instance's time is delayed, then its vehicles' states haven't advanced as much as they should. This section, therefore, addresses the simulation time synchronization aspect of requirement 2.

Chapter 2 presented multiple alternatives to address this problem. NTP and PTP achieve very good results, but they synchronize all computers to a real clock, UTC, when the FSX internal clocks represent a virtual time that is not indicative of real-world time. In fact, the internal clocks can be sped up to run faster simulations or even slowed down. This means NTP and PTP do not adequately reflect the work's needs. On top of that, NTP and PTP only achieve its high precision with specialized, expensive hardware such as GPS transmitters and routers with PTP support.

The chapter also references a comprehensive list of temporal synchronization algorithms in section 2.6.3. These are very strict algorithms, despite including so-called "optimistic" algorithms, that can also require complex features from software and hardware such as state rollbacks. Taking into account the nature of the required communication between participants and the relatively low severity of the problems resultant from desynchronized participant simulations, this level of scrutiny was deemed unnecessary. From experimental observations and as expected, the temporal divergence between FSX instance during simulation, in which these algorithms specialize, isn't significant compared to the initial offset of simulation start.

Thus, an ad-hoc, simpler algorithm was designed and implemented to periodically make sure all clocks are synchronized. This is a periodic, master-slave algorithm with a period of 30 seconds in simulation time. At simulation start, one instance is chosen as having the master time, meaning it's the one that contains the real time for the simulation while all others, the slaves, must synchronize with it throughout the simulation. The flow of messages from this algorithm is present in Fig. 4.7.

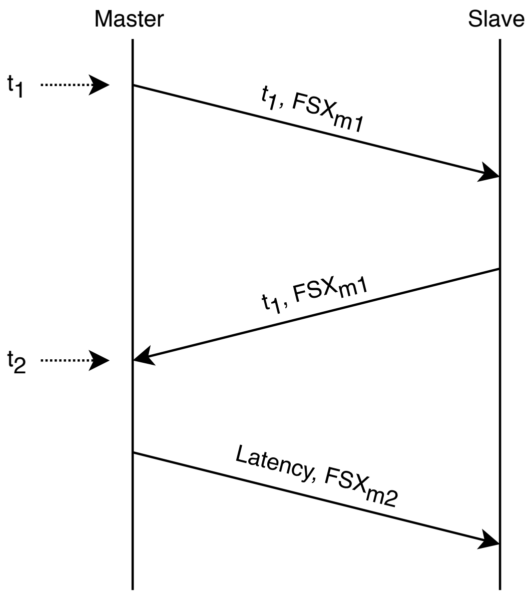


Figure 4.7: Temporal synchronization protocol's communication flow diagram

The protocol is based on calculating the round-trip time, rtt, to estimate the latency of a message so the slaves can estimate their own internal FSX times. The first message is a broadcast from the master FSX instance to all slave instances. It includes a timestamp of the master computer's clock, t_1, and a timestamp of the master FSX's internal clock, FSX_{m1}, at that moment. It should be noted that the value of FSX_{m1} is not actually used for anything and its inclusion here is merely to mimic the size of the third message for the purpose of message latency estimation, as will be seen in this protocol. Upon reception, a slave will merely redirect the same message back to the master. Let t_2 denote the timestamp of the master computer's clock upon reception of the slave's reply. This allows the master to calculate the rtt of the preceding pair of messages between master and slave and thus estimate the $Latency$ of a single message by calculating $Latency = \frac{rtt}{2} = \frac{t_2 - t_1}{2}$. This $Latency$ is sent on the third and final individual message back to the slave, along with a timestamp of the master FSX's internal time at that moment. This allows an FSX slave instance to calculate the internal time it should be on, $FSX_{\tilde{m}}$, by $FSX_{\tilde{m}} = FSX_{m2} + Latency * r$, where r is the rate at which the global simulation is running, the base simulation rate.

With the slave's supposed time known, $FSX_{\tilde{m}}$, then its current time, FSX_s, must be adjusted by $\Delta FSX_s = FSX_{\tilde{m}} - FSX_s$. The adjustment is made by increasing or decreasing, when ΔFSX_s is respectively positive or negative, the slave FSX's simulation rate by a factor of 2 for a limited amount of time Δt_s. The time will only be adjusted if there's a significant amount of variance, so the adjustment will only proceed if $|\Delta FSX_s| > t_{tolerance}$, where $t_{tolerance}$ is a parameter required by all slave FSX instances. The diagram of Fig. 4.8 shows a visualization of the adjustment required in a temporal referential where the basis is the temporal velocity, or simulation rate, r, at the instant the final message is received by the slave, which means $FSX_{\tilde{m}}$ is relatively stationary and FSX_s has to have an additional simulation rate R to match $FSX_{\tilde{m}}$. If the simulation rate is doubled, in this referential, R becomes $2 * r - r = r$. If the simulation rate is halved, in this referential R becomes $\frac{r}{2} - r = -\frac{r}{2}$. Let $\Delta t_{adjustment}$ be the real time for which the adjusted rate will be applied. From this, equation 4.3 gives the real time for which the modified simulation rate must be applied before resetting to the previous simulation rate.

$$\Delta t_{adjustment} = \frac{\Delta FSX_s}{R} = \begin{cases} \frac{\Delta FSX_s}{r}, & \text{if simulation rate doubled} \\ \frac{\Delta FSX_s}{\frac{r}{2}}, & \text{if simulation rate halved} \end{cases} \tag{4.3}$$

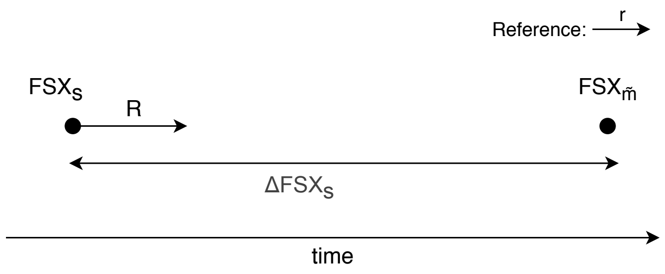


Figure 4.8: Temporal referential for time adjustment

One metaphoric example to understand these concepts, particularly how $\Delta t_{adjustment}$ is smaller when the simulation rate doubles relatively to when it's halved, is one where the times are translated to movements. Assume there are two cars, one in front of the other, traveling at 10 Km/h and they want to meet up. In option A, the car behind doubles its velocity to 20 Km/h, and in option B, the car in front halves its velocity to 5 Km/h. In which of these options do the cars meet up faster? Quick thinking might say they take the same amount of time to meet, but this is false. Relatively to the car that doesn't change its speed of 10 Km/h, in option A the other car will have a relative speed of 20 - 10 = 10 Km/h while in option B the other car will have a relative speed of 10 - 5 = 5 Km/h. So, naturally, option A is faster, its car is 5 Km/h faster than the car in option B, relatively to the car traveling at 10 Km/h. Taking these concepts back to the realm of time, these cars represent FSX_s and $FSX_{\tilde{m}}$ in Fig. 4.8. R is, in magnitude, higher when the simulation rate is doubled than when it is halved, just like the car's relative velocity in the example.

One concern with this adjustment algorithm is that the effective maximum simulation rate supported by FSX is 4x. Although a user can increase the simulation rate far beyond 4x in the client application of FSX, AI vehicles disappear above this threshold. Thus, when the base simulation rate already is 4x, instances cannot catch up to the master's time and will fall behind. A possible, not implemented approach to solve this issue with the algorithm is detailed in section 6.2.5.

4.4 Platform Modifications

Having discussed the algorithms that handle the new distributed nature of the platform, this section details the more technical details of the required modifications to implement them while giving an overview of the platform's components before and after the work from this thesis.

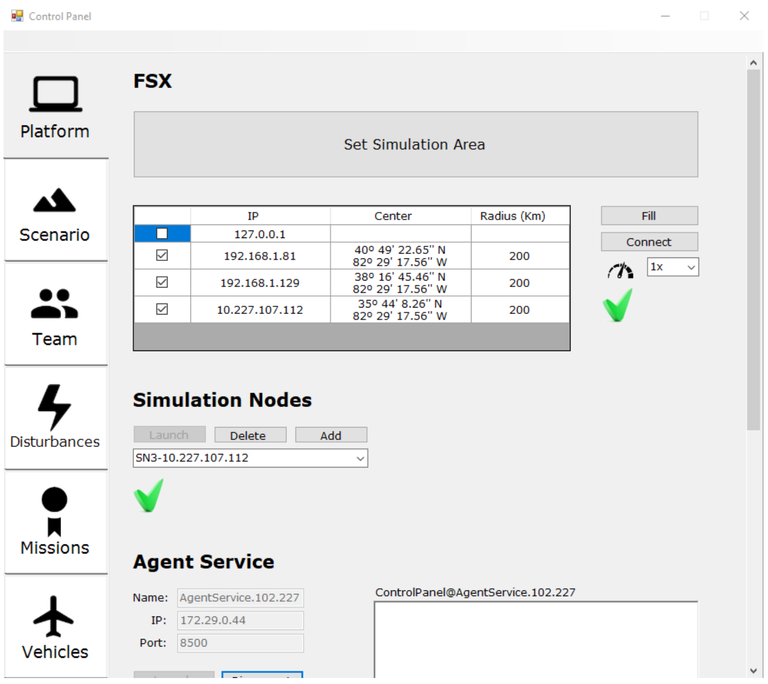
All communication between components was previously done by AgentService messages, such as the communication between ATC and vehicle agents. Now, it's been fully adapted to use RabbitMQ for its messages, a change done by Costa [10].

4.4.1 Control Panel

The Control Panel is the main entity responsible for the execution of the simulation, hence modifications throughout its functions need to be carried out so it communicates with all FSX instances. At startup, it connected to the only instance of FSX running at a specified location. This has been changed so it instead connects to the various FSX instances which must already be running at the specified locations. This component is the one responsible for distributing the multiple FSX instances geographically as per the algorithm presented in section 4.1. Once the control panel connects to and distributes the instances, it sets a common local time for each of them and initiates periodic requests to the instances for their simulation rates and local times, so it can display them. The control panel also allows to modify the instances' simulation rates, allowing to increase or decrease the global simulation's speed.

The control panel is also responsible for initializing the air traffic (ATC) and vehicle agents. These agents were all simply placed in a single instance before, but now they must be placed in their proper instances. The ATC agent can connect to multiple FSX instances, as explained in section 4.4.3, so the control panel doesn't make any judgments regarding which instances each ATC agent will connect to. The control panel limits itself to sending to the ATC agents all the required data about the instances it is connected to, from their simulated areas to their network access data, so the ATC can connect to any instances it requires. The vehicle agent on the other hand is only connected to one instance at any given time, as explained in section 4.4.4, so it does not receive the data of multiple instances. The control panel instead calculates which instance's simulation area has its center closest to the vehicle's initial position and sends its network access data to the vehicle agent.

The Control Panel's main "Platform" tab is shown in Figs. 4.9 and 4.10. Here the user can open the global simulation area map from Fig. 4.1, select and connect to the available FSX instances, configure the simulation nodes available, connect to Agent Service, and other secondary features. It should be noted this UI underwent a major refactor. The previous UI was already very cluttered, adding the necessary user interface to connect to multiple instances would have greatly exacerbated the problem. So, in order to properly add this thesis' distribution features, a new design was made which favored simplicity and free space and in which the input flow went from top to bottom. The user no longer needs to jump back and forth between sections, as, for example, to launch a scenario before, the user would have to go into the respective tab, go to the top of the window outside of the tab's area, click the usual *File* -> *Open* menu to read an XML file defining the scenario, and go back down into the tab's contents to click the *Launch* button. The file opening step was very non-intuitive, so instead the user is now presented with a file input box at the top of, but inside the tab's contents to indicate this is a mandatory setup step to launch a scenario.



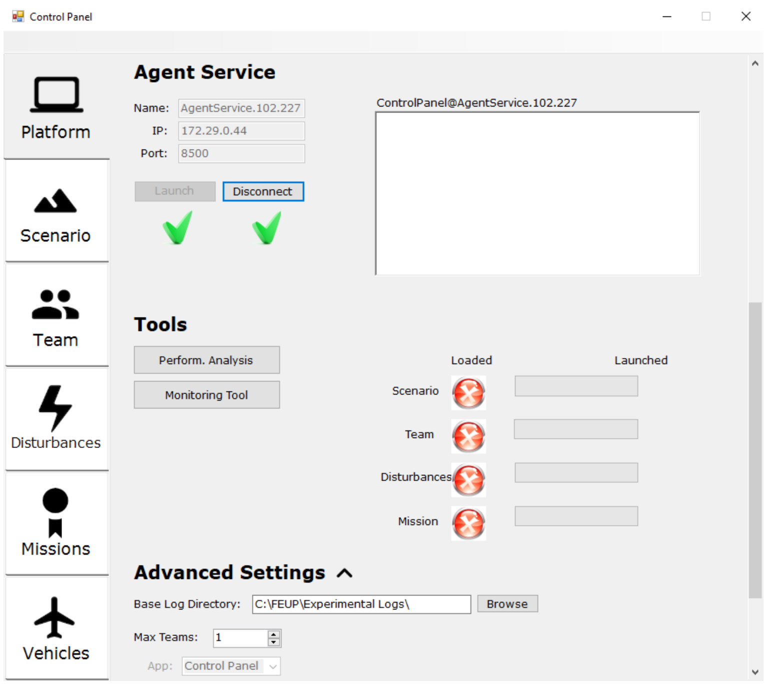
	IP	Center	Radius (Km)
☐	127.0.0.1		
☑	192.168.1.81	40º 49' 22.65" N 82º 29' 17.56" W	200
☑	192.168.1.129	38º 16' 45.46" N 82º 29' 17.56" W	200
☑	10.227.107.112	35º 44' 8.26" N 82º 29' 17.56" W	200

SN3-10.227.107.112

Name: AgentService.102.227
IP: 172.29.0.44
Port: 8500

ControlPanel@AgentService.102.227

Figure 4.9: Control Panel UI with the required modifications - part 1



Agent Service

Name: AgentService.102.227
IP: 172.29.0.44
Port: 8500

ControlPanel@AgentService.102.227

Tools

	Loaded	Launched
Scenario	✗	
Team	✗	
Disturbances	✗	
Mission	✗	

Advanced Settings

Base Log Directory: C:\FEUP\Experimental Logs\
Max Teams: 1
App: Control Panel

Figure 4.10: Control Panel UI with the required modifications - part 2

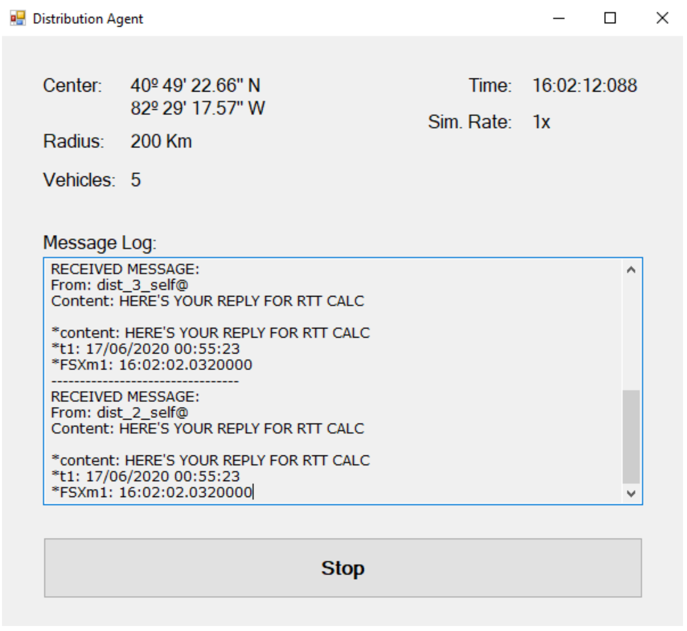
4.4.2 Distribution Agent

The distribution agent is a new type of agent created for the platform in this thesis to implement the algorithms related to vehicle migration and temporal synchronization, which were given an overview of in sections 4.2 and 4.3. It is the distribution agent that periodically runs these algorithms. There exists a distribution agent for each FSX instance, running in the same computer to minimize network traffic as each agent is continuously reading data from its corresponding FSX instance. The distribution agent takes the parameters in table 4.2.

Table 4.2: Distribution Agent parameters

Param.	Type	Description
c_0	*uint*	Configuration index of its FSX instance
c_i	*uint[]*	Configuration indexes of neighboring FSX instance $i \in [1,n]$ with n as the number of neighboring instances
l_r	*double*	Maximum relative load allowed
l_d	*double*	Maximum absolute difference of load allowed
b_{MT}	*bool*	Whether it contains the simulation's master time
r_G	*double*	Global simulation rate

The implemented UI for this agent is present in figure 4.11. It shows relevant data from its associated FSX instance, namely its area's center coordinates and radius, its internal time, and its simulation rate, whilst informing how many vehicles are being simulated by it. It also presents a log with all the messages received by the agent, and a button to toggle the vehicle migration and temporal synchronization procedures executed by the agent.



Center: 40º 49' 22.66" N
82º 29' 17.57" W
Radius: 200 Km
Vehicles: 5
Time: 16:02:12:088
Sim. Rate: 1x

Message Log:

```
RECEIVED MESSAGE:
From: dist_3_self@
Content: HERE'S YOUR REPLY FOR RTT CALC

*content: HERE'S YOUR REPLY FOR RTT CALC
*t1: 17/06/2020 00:55:23
*FSXm1: 16:02:02.0320000
----------------------------------
RECEIVED MESSAGE:
From: dist_2_self@
Content: HERE'S YOUR REPLY FOR RTT CALC

*content: HERE'S YOUR REPLY FOR RTT CALC
*t1: 17/06/2020 00:55:23
*FSXm1: 16:02:02.0320000
```

Figure 4.11: Distribution Agent UI

Migration Protocols

Both border and load balancing protocols need to be able to broadcast a message to the neighboring instances, thus each distribution agent must have a known unique identifier. This is achieved with the configuration index of its FSX instance, unique for every distribution agent and known for all of its neighbors. With this unique index, two channels of communication are created for each agent, each with its unique routing key: one to receive messages and another to broadcast messages to all of its neighbors. The routing keys can be constructed solely from knowing the configuration index, as they are of the formats $dist\_c_0\_self$ and $dist\_c_i\_neighbors$ (e.g. $dist\_0\_self$ and $dist\_13\_neighbors$) for respectively the reception and neighbor broadcast channels. Any message sent will have the $dist\_c_0\_self$ key in its envelope to tag who sent it and allow for individual replies.

Practically, on startup, a distribution agent subscribes to the $dist\_c_0\_self$ routing key to receive individual messages and subscribes to all the $dist\_c_i\_neighbors$ routing keys to receive the broadcast messages from its neighbors. Back to Figs. 4.5 and 4.6, the first message is a broadcast to routing key $dist\_c_i\_neighbors$, received by all $c_i, i \in [1,n]$, while the second message is an individual reply to routing key $dist\_c_i\_self$ obtained from the first message's envelope.

Once the migration parameters are decided, or in other words what vehicles to migrate between which instances, a message is sent to each of the vehicle agents about to migrate. This message contains only the configuration index of the new FSX instance the vehicle is to migrate to. The reception of this message by the agent will trigger its migration, further explained in section 4.4.4.

Temporal Synchronization

For the temporal synchronization algorithm, the distribution agent knows if it has the master time by the corresponding initialization parameter. All distribution agents that do not have the master time subscribe to the routing key $dist\_time\_master$, while the agent that does have it periodically broadcasts the first message with t_1 and FSX_{m1} to this routing key as outlined in section 4.3.

In the synchronization phase, section 4.3 explains the kind of messages that are exchanged between distribution agents and how the slave instances' are sped up or slowed down. This act of changing the simulation rate is realized by messages to *SimConnect*. *SimConnect* does not allow to set the simulation rate to a value, instead it works by doubling or halving the current rate. So when, for example, a slave simulation needs to be sped up temporarily to become synchronized with the master instance, its distribution agent transmits a message to it ordering it to double its simulation rate. After the required time to become synchronized has passed, the distribution agent will transmit another message to the simulation ordering it to halve its simulation rate, effectively resetting it to its original simulation rate.

4.4.3 ATC Agent

The ATC agents are responsible for controlling vehicle traffic similarly to a real-life airport traffic controller in an assigned zone, be it at an airport or somewhere else. Originally, it connected to the one FSX instance in order to find vehicles within its assigned area. Now, it can connect to multiple instances, but not necessarily all. Each ATC agent has to cover the vehicles in an area, so it only connects to the FSX instances which have overlapping simulation areas. To do this, it first receives all the necessary data on all the FSX instances from the control panel on startup. Then it checks if each of these overlaps with the ATC's coverage area, and if so makes it possible for the ATC to connect to it.

Fig. 4.12 has an example of an ATC agent connecting to only some FSX instances. Here the ATC covers an oddly-shaped area while the FSX instances are represented by the circles. It only connects to instances 1 and 2, since it overlaps with them, but not with instance 3.

4.4.4 Vehicle Agent

Each vehicle agent is associated with a vehicle in FSX. The agents are responsible for the vehicle's behavior, sending their waypoints to FSX at the beginning. A waypoint is a position on the map toward which vehicles will eventually travel to, and a vehicle in FSX has a list of waypoints that make up its travel plan. When a vehicle agent has to modify its planned trajectory, it modifies the list of waypoints for its respective vehicle in FSX. The agent continuously receives data about its respective vehicle from FSX, such as its current position and velocity. In addition, the vehicle

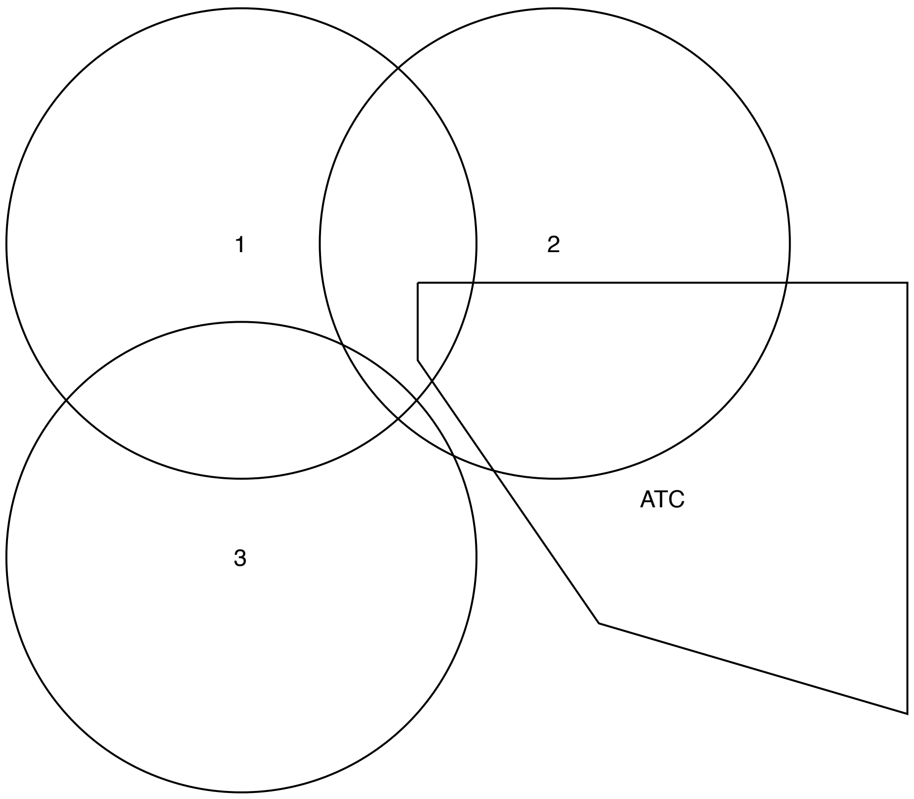


Figure 4.12: ATC overlapping FSX instances 1 and 2

agent is the one that instantiates its vehicle in FSX once it is created by the Control Panel and is connected to FSX.

In this new paradigm, the vehicle agent is still connected to only one FSX instance, the one simulating its vehicle, but it's not immediately clear which instance to first start simulating the vehicle in. It instead connects to the instance whose center of simulation is closest to the vehicle's initial location, something that is calculated by the control panel and whose configuration index is passed as an initialization argument to the vehicle agent, as explained in section 4.4.1.

The vehicle agent initiates its migration to another FSX instance when it receives the order from a distribution agent to do so. Two methods were tested for this migration procedure. Both consist of disposing the vehicle agent's connection to the current instance, which destroys the simulated vehicle within the instance by association, establishing a connection to the FSX instance to which it's been ordered to migrate, and creating a vehicle in it with the same state as it last registered in the original instance. The two methods only differ in the timing of these actions, this is explained afterwards. The vehicle's state is known because the vehicle agent requests FSX to continuously send data about it, with the precise variables listed in Table 4.1. However, these variables don't make up the entire state of the vehicle, there are others that cannot be set by the *SimConnect* API. *SimConnect* has a significant amount of variables that do not work for AI vehicles, they only work for the user's vehicle. This is something that Garrido also encountered when trying to inject faults into vehicle components, although most of the variables he listed as not working for AI vehicles are related to the vehicle's engine [15]. Throttle, aileron position, elevator position, and rudder position were found to not be working either, which are the main and most basic controls of an airplane, hence when an airplane migrates it can abruptly change its trajectory and speed before quickly readjusting itself to continue to the next waypoint as before. This abrupt change is only expected to be noticeable when the plane is in the process of maneuvering, as when it's using its elevators to change pitch and suddenly the elevators go to their default position.

In both methods, a hot swap occurs where the connection to the original FSX instance is replaced by the connection to the new instance, and the original connection is terminated. In the first method, the hot swap occurs immediately after opening a connection to the new instance. However, the lack of some settable variables results in state discrepancies from an instance to the next, as tests showcase in section 5.1. In particular, the inability of setting the engine power manually means the engine starts at low power in the new instance, and the vehicle, if it was moving in the original instance, will slow down before the engine power increases back to nominal levels and the vehicle accelerates. Thus, a second method was developed in which the hot swap is delayed some amount of time after establishing a connection to the new instance, let this time be the stabilization period. The vehicle is first created with the original instance's state, and during the hot swap its state is set again, but this time to the new state in the original instance at the time of the hot swap. This is to give time to the new instance to stabilize the vehicle, hopefully letting it reach a state with an adequate engine power by the time its state is set the second time.

One small quirk of this migration's hot swap is that the connection to FSX is handled by a *disposable* object in the *SimConnect* C# API, which is not thread-safe when being disposed of. This is a problem because the vehicle agent is multi-threaded in nature, handling tasks like managing the UI, receiving *SimConnect* data, and receiving messages from distribution agents, so if the *SimConnect* object is disposed of it's not feasible to keep the agent stable. The implemented workaround to this problem is to switch to the new FSX instance before disposing of the last one, copying the latter's *SimConnect* object to a temporary variable until it is disposed of. This was found to be stable, with the agent handling the switch between connections graciously.

Another required modification is to change one of its associated properties in *AgentService*'s yellow pages. *AgentService* knew an agent's vehicle ID inside FSX in order to be able to associate it with FSX, but now with multiple FSX instances the single ID is no longer unique, each FSX instance attributes vehicle IDs independently of each other. This property has been changed to also include the ID of the FSX instance in the format f_v, where f is the FSX instance ID and v is the vehicle ID within the instance. Whenever the vehicle migrates, this property is changed to reflect the new FSX instance ID and vehicle ID. This means the platform has the means to know which agent a message from a vehicle corresponds to, in any FSX instance.

Chapter 5

Experimental Results

After presenting the details of the implemented solution, this chapter showcases the results of multiple tests to both the migration of vehicles and the temporal synchronization between FSX instances. All tests were repeated 5 times to ensure consistency.

5.1 Vehicle Migration

Different scenarios were tested in which FSX instances are placed in different locations to test different kinds of migrations. All tests have 2 instances except for the last one, which has 3.

5.1.1 Validation of vehicle migration

As talked about in section 4.4.4, two methods of vehicle migration were implemented. Three tests are presented here: the first test evaluates the first method with an immediate hot swap of FSX instances, while the second and third tests assess the second method with a stabilization period of 1 second and 5 seconds, respectively. The tests read the state variables of an airplane - latitude (Lat), longitude (Lng), altitude (Alt), heading (Hdg), pitch, bank, and airspeed - in every simulation frame of the FSX instance it is present in as it migrates from one to the other. The results are showcased in Fig. 5.1 and Table 5.1, Fig. 5.2 and Table 5.2, and Fig. 5.3 and Table 5.3, for respectively the first, second, and third tests. These tests have the objective of establishing the integrity of the vehicle's simulation during migration. The variables of the graphics have been normalized between 0 and 1 in order to represent their changes despite their vastly different magnitudes.

As predicted in section 4.4.4, the first graphic shows an abrupt change of multiple variables at the instant the migration takes place, roughly in the middle of the graphic's horizontal axis, but according to the first table, these changes do not represent a large difference in magnitude. The graphic suggests the most problematic variable transition to be the airspeed, even though the difference is only of 1.5 knots. The deduced reason for this disparity is the lack of engine control in the creation of vehicles. The vehicle is instantiated at no or low power and steadily increases power, which means the vehicle slows down a bit until it gradually accelerates from the reestablished engine power.

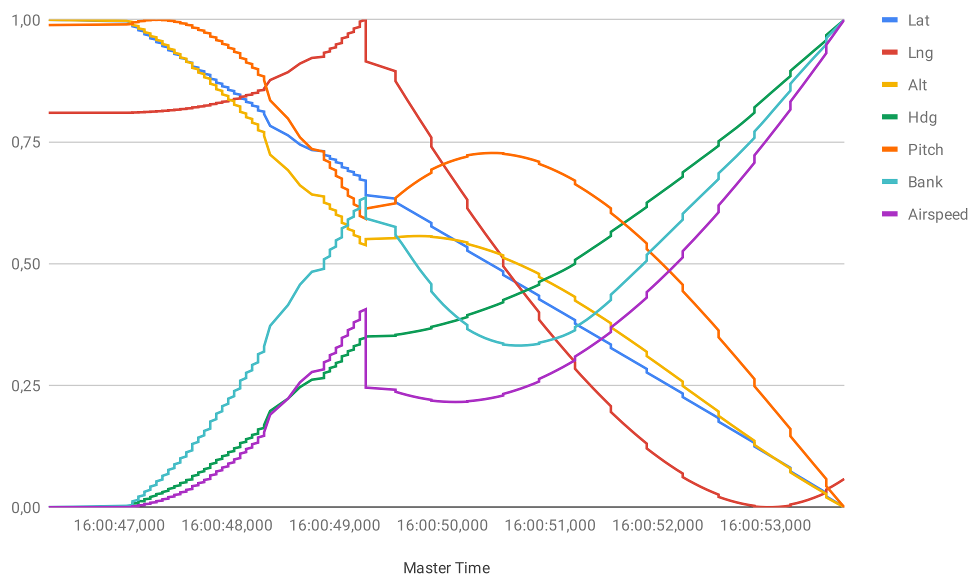


Figure 5.1: Normalized state variables of an airplane as it migrates once between FSX instances

Table 5.1: Values of an airplane's state variables in a 7 second timespan as it migrates once between FSX instances. Latitude and longitude in degrees; altitude in feet; heading, pitch, and bank in degrees; airspeed in knots.

	Average	Min	Max	Transition
Latitude	39,9943	39,99077	39,9966	39,99467 → 39,9945
Longitude	-82,88139	-82,88141	-82,88138	-82,88138 → -82,88138
Altitude	3035,35	3018,696	3046,854	3033,838 → 3034,185
Heading	180,526	180,035	181,429	180,52 → 180,524
Pitch	-0,778	-3,746	0,45132	-1,263 → -1,174
Bank	0,938	0,095	2,224	1,448 → 1,356
Airspeed	188,475	185,9	194,95	189,587 → 188,136

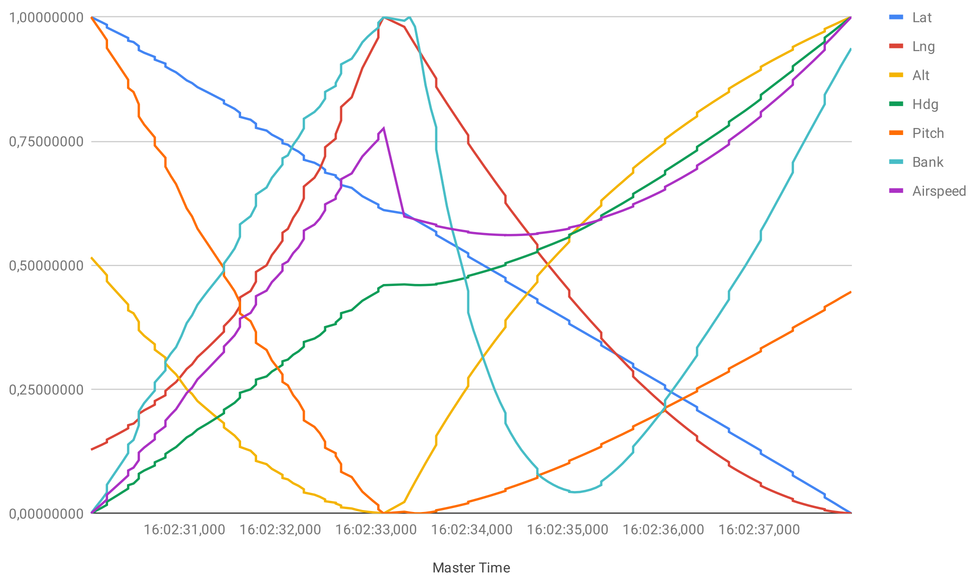


Figure 5.2: Normalized state variables of an airplane as it migrates once between FSX instances with a 1 second lag

Table 5.2: Values of an airplane's state variables in a 7 second timespan as it migrates once between FSX instances with a 1 second lag. Latitude and longitude in degrees; altitude in feet; heading, pitch, and bank in degrees; airspeed in knots.

	Average	Min	Max	Transition
Latitude	39,99096	39,98741	39,99448	39,99173 $\rightarrow$ 39,99168
Longitude	-82,88137	-82,88139	-82,88133	-82,88133 $\rightarrow$ -82,88133
Altitude	3030,802	3019,626	3044,139	3019,626 $\rightarrow$ 3020,191
Heading	181,559	180,57	182,571	181,49 $\rightarrow$ 181,493
Pitch	-4,69	-5,982	-1,572	-5,982 $\rightarrow$ -5,967
Bank	2,263	1,537	3,141	3,141 $\rightarrow$ 3,129
Airspeed	197,861	190,083	203,991	200,869 $\rightarrow$ 198,401

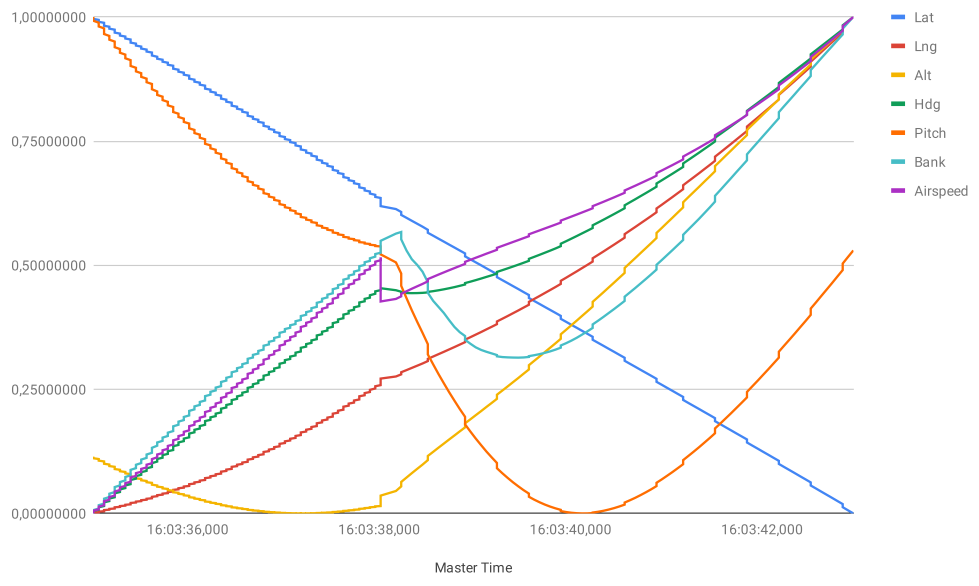


Figure 5.3: Normalized state variables of an airplane as it migrates once between FSX instances with a 5 second lag

Table 5.3: Values of an airplane's state variables in a 7 second timespan as it migrates once between FSX instances with a 5 second lag. Latitude and longitude in degrees; altitude in feet; heading, pitch, and bank in degrees; airspeed in knots.

	Average	Min	Max	Transition
Latitude	39,99093	39,98614	39,99369	39,99091 $\rightarrow$ 39,99081
Longitude	-82,8813	-82,88137	-82,88111	-82,8813 $\rightarrow$ -82,8813
Altitude	3031,578	3020,047	3077,756	3020,971 $\rightarrow$ 3022,115
Heading	181,589	180,839	182,803	181,732 $\rightarrow$ 181,73
Pitch	-6,464	-10,294	-3,042	-6,403 $\rightarrow$ -6,51
Bank	3,075	2,092	4,717	3,484 $\rightarrow$ 3,533
Airspeed	202,488	193,457	215,666	204,91 $\rightarrow$ 202,938

The second test, in Fig. 5.2 and Table 5.2, still shows an abrupt change of values in the transition between instances despite the 1 second stabilization period, while the third test's graphic (Fig. 5.3), with its 5 second stabilization period, seems to have a smoother transition. Still, when verifying the associated tables' real values, there's a similar magnitude in the difference between instances, particularly around 2 knots of airspeed. Even though the engine has had more time to increase power in the new instance, so did the engine in the original instance, which might explain how there's still an abrupt change in airspeed.

Although the migration procedure is not perfectly smooth in any test, it still positions the vehicle correctly and keeps its travel plan. In the case of abrupt changes in the vehicle's state, it quickly corrects itself to nominal values, be it with a stabilization period or not. It is not worth it to include a stabilization period, judging by how it does not contribute to a smoother state transition.

5.1.2 Validation of the border protocol

This test aims to validate the border protocol at a base level. The initial scenario is represented in Fig. 5.4. It has two instances, one to the north and one to the south. It creates 5 planes at an airport, represented by the figure's singular airplane icon, which is located at the intermediary zone between both instances and is very near the southern border. The planes immediately takeoff and go south towards the southern instance's exclusive zone. For this test, even though the planes are nearer the center of the southern instance than the northern one, they are hardcoded to be created in the northern instance. Thus, this test aims to see if the border protocol acts quickly to migrate the vehicles to the southern instance as otherwise they would cross outside their instance's boundary in the next moments.

Indeed this is what happens. It was observed that the load balancing protocol took precedence by migrating only 2 of the 5 planes to the southern instance, but, very shortly after, the border protocol did migrate the remaining 3 planes to the southern instance.

The global area was input as having its northwest point at (42^{o}54'42.9" N, 85^{o}00'00.2" W) and its southeast point at (39^{o}18'52.6" N, 82^{o}45'14.7" W), resulting in the northern instance being placed at (41^{o}38'24.35" N, 83^{o}43'41.56" W) and the southern instance at (39^{o}5'47.15" N, 83^{o}43'41.56" W).

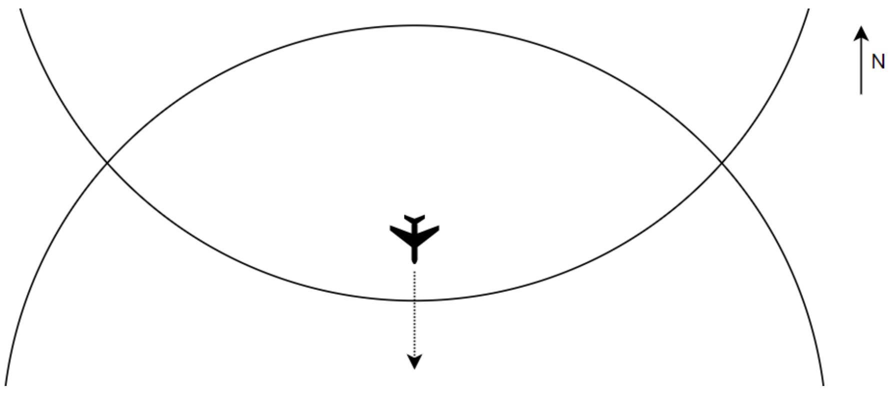


Figure 5.4: Initial scenario for the validation test of the border protocol

5.1.3 Validation of the load balancing protocol

This test aims to validate the load balancing protocol at a base level. The initial scenario is represented in Fig. 5.5, and the conditions are very similar to the validation test of the border protocol. Like the former, there are two instances, one to the north and another to the south, and 5 planes are created at an airport represented by the figure's singular airplane icon, but this one is further north, although still in both instances' intermediary zone. Here the planes are also created in the northern instance but this is as intended naturally since it's the instance with the closest center to them. The planes also takeoff and go south, but in this one the border protocol is not meant to act as the planes are sufficiently far from the southern border. The purpose of this test is to, therefore, verify if the load balancing protocol migrates half of the vehicles to the southern instance, or in this case 2 or 3 of them since there are 5 in total.

This test is also successful, observing an immediate migration of 2 planes to the southern instance in 4 out of 5 experiments. In the first experiment, only one plane migrated immediately but a second migration occurred 15 seconds afterward, so it was also a success. One possible explanation for this outlier is that the vehicles are not all created at the same instant so the load balancing protocol could have had an iteration right in the middle of the vehicles' creation. The reason for the 15-second delay between the migrations in the first experiment is mostly due to the 10-second timeout between migrations with the rest of the delay filled by the 5-second polling interval of the load balancing protocol.

The global area was input as having its northwest point at (42°05'41.3" N, 83°45'36.2" W) and its southeast point at (38°17'01.3" N, 82°06'04.1" W), resulting in the northern instance being placed at (40°49'22.65" N, 82°29'17.56" W) and the southern instance at (38°16'45.46" N, 82°29'17.56" W).

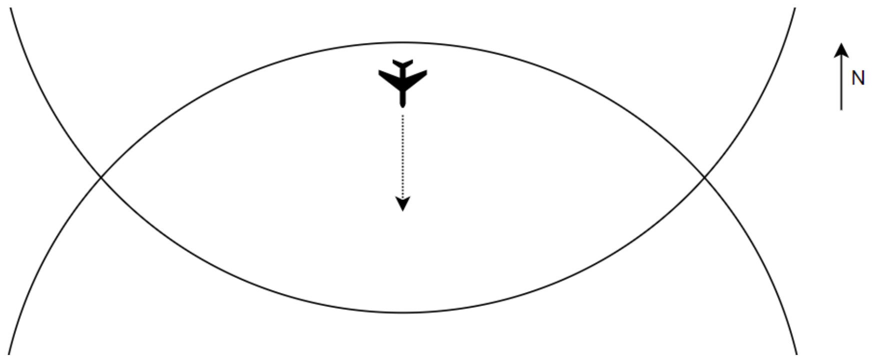


Figure 5.5: Initial scenario for the validation test of the load balancing protocol

5.1.4 Continuous testing of both migration protocols

Here are presented two tests in two different scenarios to validate and evaluate both the border protocol and load balancing protocols. The first test has a simpler scenario with two instances while the second test has three overlapping instances.

In this first test, 10 planes travel from the center of one instance to the next, exposing both border and migration protocols at different stages. Again there are two instances, one to the north and one to the south, with the north instance's center being an airport. The 10 planes are created in the same spot at this airport but they are created 20 seconds apart, with each one taking off immediately and traveling south. This results in the line of planes traveling south seen in Fig. 5.6, taken early on from the radar of an ATC agent in the area. This is also represented in the graphic from Fig. 5.7 with the first increase of the blue line from 1 to 10 vehicles, as will all of the next events in this test.

The planes start in the exclusive zone of the northern instance, but as they travel south they will enter the intermediary zone and will then finally enter the southern instance's exclusive zone. Then, it is expected that roughly half of the planes eventually migrate to the southern instance and also that later on all of them will be in the southern instance. This is verified to indeed happen in the aforementioned graphic, which can be split into three major event groups. First, as already mentioned, the 10 planes are created in the northern instance, 20 seconds apart from each other. Then, the load balancing protocol kicks in as the planes reach the intermediary zone to migrate 4 of the 10 planes. Although the ideal solution would have been to migrate 5 vehicles, the protocol's parameters have a certain tolerance to make it more robust in complex situations, preventing cases of vehicles continuously migrating back and forth between instances. Finally, the border protocol takes action to migrate all of the remaining 6 planes to the southern instance, as they approach the border of the northern instance and enter the southern instance's exclusive zone. The graphic shows several migrations of only one vehicle each, this is because the vehicles were created 20 seconds apart from each other and therefore travel with a significant distance between them, so they enter the instances' zones in different intervals.

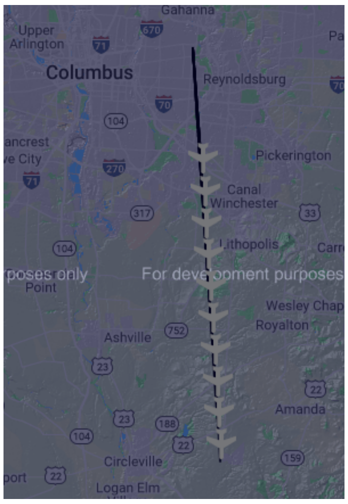


Figure 5.6: Initial scenario for the validation test of the load balancing protocol

For the second test, three instances have been set up with planes traveling southwards as seen in Fig. 5.8. The instances have been arranged in this geographic disposition specifically for this test rather than letting the implemented circular meshing algorithm place them. This is to test the

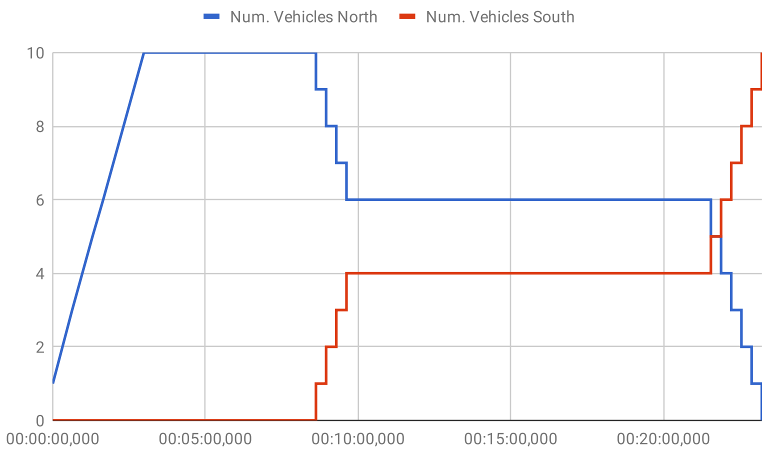


Figure 5.7: Vehicles migrating between 2 instances

migration protocols' capabilities across a more complex system where every pair of instances has an intermediary zone and all 3 share a common intermediary zone at the center. As before, the figure's singular airplane icon represents 10 planes which are to travel south and are created 20 seconds apart, taking off as soon as they are created. The airplanes are created on the north instance's exclusive zone, and will proceed to travel to the intermediary zone between instances north and east, the intermediary zone between all instances, the intermediary zone between instances east and south, and finally on to the exclusive zone of the southern instance. As in the figure, let the north instance be instance 1, east instance be instance 2, and south instance be instance 3.

As Fig. 5.9 shows, the test went as expected. The planes are progressively created in instance 1 since they are closest to its center. They arrive very quickly to the border of instance 2, so the load balancing protocol does its job and migrates the vehicles into a stable configuration of 6 to 4 vehicles. They then arrive at the intermediary zone common to all instances so the load balancing protocol migrates the vehicles into a 4-3-3 configuration for a while. As they leave instance 1, the border protocol migrates them out to instances 2 and 3 while the load balancing protocol concurrently balances the latter, resulting in a 0-6-4 configuration. Finally, vehicles reach the border of instance 2 and the border protocol migrates all of the remaining ones out to instance 3. Remember the vehicles aren't created at the same time, meaning they reach the instances' borders at different times and so they mostly migrate one at once, sometimes two, hence the jagged edges in the graphic.

Both tests show how both protocols contribute to the distributed simulation. Not only does the border protocol keep the simulation running on distances greater than those a single instance would be able to cover, but the load balancing protocol helps to offload instances that would have to do more work otherwise.

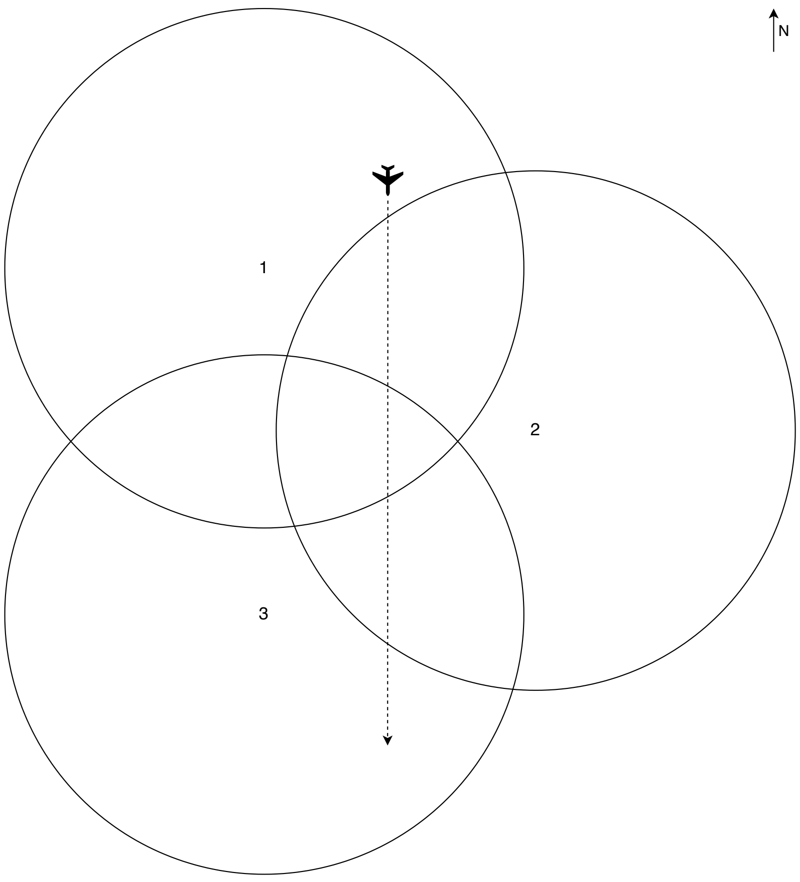


Figure 5.8: Initial scenario for the test with 3 instances

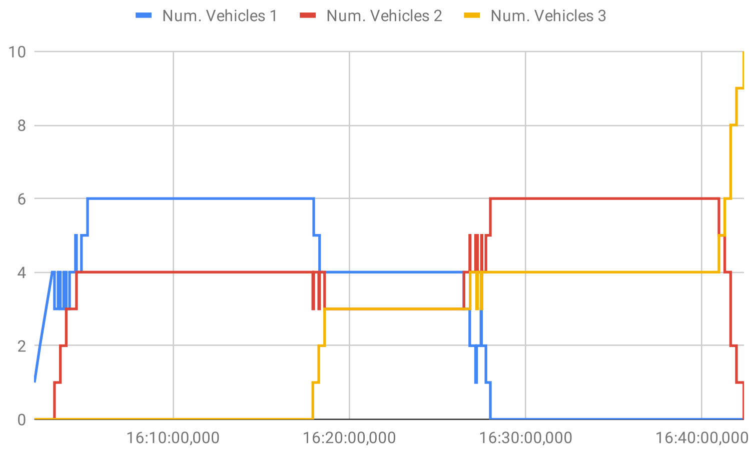


Figure 5.9: Vehicles migrating between 3 instances

5.2 Temporal Synchronization

Detecting the real offset between clocks in distinct computers is hard since they might be in fact at different real-time instants, but the implemented synchronization algorithm estimates this offset periodically and is thus the target of analysis in this section, for which two tests were made. In both tests, two computers ran 5 simulations for between 12 and 13 minutes of simulation time each. The two tests differ only on their simulation rate, with the first running at 1x the simulation rate and the next one at 2x. The computers were connected via Wi-Fi and the *ping* utility from Windows calculated the network latency between the two to be between 1 and 2ms on average with rare spikes up to 15ms. There's a graphic for each test with data intervals of 30 seconds of simulation time.

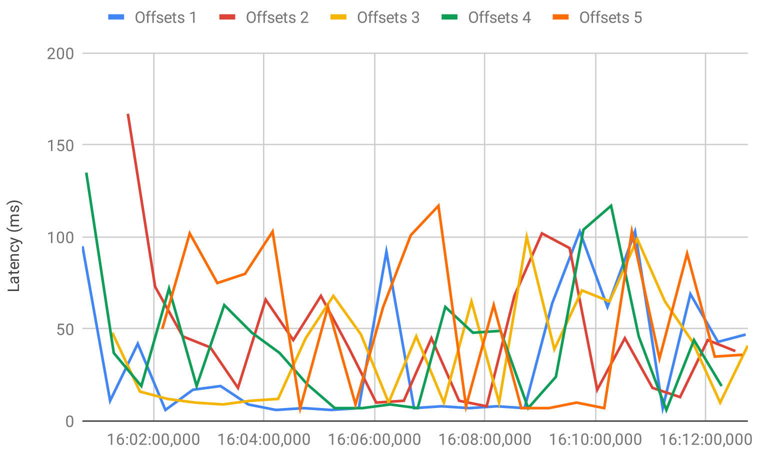


Figure 5.10: Estimated latencies between FSX instances in 5 runs at 1x sim. rate

The test results for the first one are shown in the graphic of Fig. 5.10. The graphic shows latency to vary greatly but it is overall bounded to 100ms. One special case was the first test, with most values forming a line between 6 and 9ms. Unfortunately, the tests had to be run via Wi-Fi on a home network which was also in use by other applications and devices, so this was most likely the main cause for the high variance in the estimated latency.

Note how all the simulations except for the last one have a bigger latency at the start, in fact, the last one also presented such but the data point was removed because it was in the order of seconds and it broke the scale for the rest of the graphic. The initially high latency is expected since on FSX connection establishment the local times are set to be the same but it's not a synchronized effort. The temporal synchronization algorithm immediately detects and corrects this abnormally high latency in its first iteration.

The results for the second test are displayed in Fig. 5.11. The 5 individual runs were run between 12 and 13 minutes of simulation time at a simulation rate of 2x, which corresponds to between 6 and 6.5 minutes of real time. Similarly to before, the first data point for two of the runs was removed to preserve the visible scale of the graphic as they were in the scale of seconds.

A lot of the measured latencies are still within 100ms, but this time a lot of the measurements go over this limit to the 100-200ms range. This could be a result of the increase in simulation rate, increasing the divergence of the internal times between FSX instances, but it could also be, again,

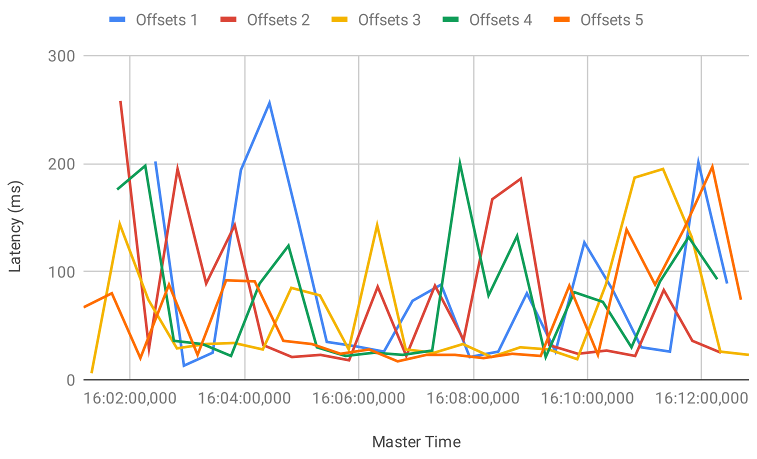


Figure 5.11: Estimated latencies between FSX instances in 5 runs at 2x sim. rate

due to the less-than-ideal conditions of the network, in which the two computers were connected by WiFi and other applications and devices were using the network.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

An investigation into distributed simulation standards was conducted, finding DIS, HLA, DDS, TENA, and FMI. DDS was chosen from these as being promising, but it failed to be adequate in practical experimentation. Message-oriented middlewares were analyzed extensively by Costa, so, from his research, RabbitMQ was chosen [10]. As temporal synchronization is an important aspect of distributed simulation, temporal synchronization techniques were researched. It was concluded that a simpler, ad-hoc solution would be used instead.

The context of the problem was described, in particular the state of the platform in which the solution was to be developed. This solution has been implemented and detailed in this thesis, with future work to improve it explained afterward in this chapter.

At the beginning of this document, four research questions were placed:

- How can the processing power of multiple computers be harnessed to expand the functionalities of the platform?
- What are the concrete benefits of such a distribution?
- Are there industry standards for distributed simulation that can simplify this process?
- Can a proper load balance be achieved amongst the participating computers?

As for the first question, multiple computers have been used to distribute multiple instances of FSX, one FSX per computer. FSX by itself runs a centralized simulation, but the platform that runs on top of FSX connects all of the instances to have a distributed vision of the simulation. As for the second question, enlarging the map available to the global simulation is a considerable benefit, since a single FSX would be limited to a circular area with a radius of 200 Km. Supporting more vehicles in the simulation due to offloading them from a single centralized instance could also have been an advantage, however, the platform's agents proved to be the bottleneck by far in terms of CPU time and memory usage, rendering the usage of system resources by FSX relatively insignificant.

As for the third question, industry standards were investigated but those particularly for distributed simulation were not adequate. Instead, RabbitMQ, a message-oriented middleware, was adopted. For the final question, a load balance protocol has indeed been implemented which balances vehicles among FSX instances whenever possible.

And then, in chapter 3, nine requirements for the solution were laid out. Requirement 1 has been ticked already by the answer to the first research question. Requirement 2 gets a partial tick, as temporal synchronization has been established but weather synchronization has been left for future work. Still, temporal synchronization can be considered a much more important problem to solve than weather synchronization, especially when the weather can be set to be static in the simulation. Requirements 3, 4, and 5 have been tackled too as the user does not have to do overly complicated actions. Instead, this thesis only added the complexity of choosing which instances to connect to. Everything else, like the instances' virtual locations on the map or in which instances the vehicles are created, is handled automatically by the platform. In fact, the new Distribution Agent is only created when there's more than one instance to connect to, which means simulations with only 1 instance function just as before. Unfortunately, requirement 6 was not approached and is left for future work. Likewise for requirements 7 and 8, since it was enough during development and testing to simply analyze logs, see the vehicles traveling in each FSX, and see vehicle agent data such as their location on the ATC agents' UI. Concluding this list, requirement 9 has been accomplished as per the answer to the fourth research question.

6.2 Future Work

The algorithms developed here are expected to work in other simulations, as their theory presented in sections 4.1 and 4.2 is ultimately not tied into the platform nor FSX. One disadvantage of using FSX is that it is proprietary, closed software. There's no direct access to the simulation besides an API that doesn't have the best documentation and is somewhat lacking. For example, for the ATC agent to look for the vehicles in a certain area it has to request all of the vehicles in the instance and only then filter them by area. This is inefficient when the area covered by the ATC agent is smaller than the instance and there are a lot of vehicles outside the ATC agent's area. FSX should be able to communicate only the subset of the desired vehicles to reduce stress on the network and the agent's processing power. Another case is talked about in the validation test of vehicle migration (section 5.1.1), where the lack of controls for AI vehicles in the API leads to a somewhat rough vehicle migration. So, it would be interesting to see an open simulator distributed geographically using the algorithms of vehicle migration and temporal synchronization described in this thesis.

Even though the core of the solution has been deemed as finished, a number of tasks are left for future work and their proposed solutions are explained in this section. Unfortunately, implementing features is a slow process in this platform, from the lack of automated tests, to the incomplete documentation of the Microsoft *SimConnect* API, and numerous bugs within the platform itself. Running a test in the platform implies a series of manual steps in the GUI, and then waiting for the test to conclude, something that might take minutes since the simulations are not instantaneous, they're locked to a maximum simulation rate of 4x. Therefore, a good number of adaptations to make the whole of the platform truly compatible with the implemented distributed approach were left out from being implemented, although most of the adaptations were considered and designed to some level.

6.2.1 Geographic Distribution

In the current state of the geographic distribution of instances, the user inputs a rectangle for the global simulation area in the form of text boxes for the coordinates. This could be upgraded to set the points via clicking on the map and also by allowing any polygon instead of just a rectangle. Although a rectangle fits a lot of desired shapes like Portugal's, allowing a generic polygon would be more adequate in a lot of cases such as Spain's airspace. Currently, if one wanted to simulate just the Spanish airspace, the global simulation area would also have to include Portugal, potentially requiring more FSX instances than needed.

Another upgrade in this area would be to introduce alternatives to circular meshing in the placement of the FSX instances. Circular meshing allows to efficiently cover the entire area of the user's global rectangular map, but another approach could be desired. One example of such an alternative is to focus the instances on vehicle traffic hotspots. One iteration of the simulation could be first run to analyze the traffic and generate a heat map. This heat map would be fed into the control panel to place the instances adequately, possibly setting them with an inferior radius. Take for instance a very crowded airport, an instance could be placed on it with a small radius to help control the load on the instance, with bigger instances surrounding it.

Additionally, the user could manually set the geographic details of the instances. This is what was intended for the UI of the control panel in figure 4.9. The "Fill" button would use circular meshing to place the selected instances and then the user could edit the centers' coordinates and radiuses if so was desired. Then the "Connect" button would connect to the instances and place them on the input locations. However, in the current implementation, the "Fill" button is only a placeholder, the user cannot edit the table with the instances' details, and the "Connect" button not only connects to the instances but also executes circular meshing.

Finally, in light of requirement 6, the user should be warned of setup incompatibilities that the platform would not know how to solve. For example, the control panel could warn the user before the simulation starts when a vehicle or an ATC is supposed to be instantiated in a region that isn't simulated by any of the FSX instances.

6.2.2 Plug n' Play

As it stands, the FSX instances are not PnP, meaning they must be connected to at the beginning of the simulation by the Control Panel and must not be shut down. However, nothing in the migration and temporal synchronization protocols require them to be static. With some extra functionalities implemented, FSX instances could be actively added and, with some constraints, removed from the simulation.

Each FSX instance must have an associated Distribution Agent, which itself has a list of all the other FSX instances whose simulation areas overlap with its own. This list is the only challenge in making this a PnP system. A new FSX instance requires the knowledge of all the other instances with which it overlaps, and likewise, those new instances must know about the new neighboring FSX instance, so the distribution protocols in place start including it. This can be done by the Control Panel, calculating the overlapping instances and including the neighbors' configuration indexes on the creation of the new instance while sending a message to which of the neighbors with the new instance's configuration index. The ATC Agents in the area must also know about the new FSX instance, so the Control Panel must also check for area overlaps, as it already does at the beginning, and inform the overlapping ATC's of the new FSX instance they must connect to.

Something similar can also be done to dynamically remove FSX instances, however, any remaining vehicles would first have to be flushed out to other instances. This means that the node can not be safely removed if it has vehicles in areas that do not overlap with other instances. If all the vehicles can be pushed out, then the ATC's can be told to disconnect from the instance and the neighboring FSX instances can be informed to remove the instance from their lists of neighboring instances, and then the FSX instance can be shut down.

Being able to add and remove instances also makes the simulation more robust to failing nodes, allowing to divert a simulation's failing area to, say, a backup node.

6.2.3 Weather Synchronization

As each FSX instance is independent, the weather in each must be maintained as a whole to avoid cases such as a plane migrating from clear weather to a tropical storm. When starting an FSX instance, the user can set the weather to either be static or dynamic. In the case of static weather, the user only has to start every FSX instance with the same kind of weather, but for dynamic weather, the changes would have to propagate between the FSX instances to ensure simulation integrity.

This topic wasn't a priority of the implementation, considering all the missing functionality in the platform and the required changes for this thesis. However, this section proposes a way to synchronize the weather. *SimConnect*, FSX's API, allows to retrieve and set the weather at several weather stations around the map while also allowing for the addition and removal of weather stations from the map, providing a mechanism to effectively set the weather anywhere. Each instance has a fixed period of time between weather updates. Like in the temporal synchronization, a master instance can be defined as having the correct weather and this information would be propagated to neighboring instances at the time instants that weather is updated, reading the weather stations at the overlapping areas and setting them on the other FSX instances. The instances that receive the weather propagation then also communicate their own, other overlapped weather stations to the remaining neighbors, and so on until all FSX instances are updated.

6.2.4 Vehicle Migration

The implementation of vehicle migration is missing a step in order to ensure simulation integrity between all of the platform's components. After the protocols decide what migrations to do and before these actually take place, the ATC agents should be warned that these are about to happen. This is to prevent two known issues within the ATC when it's connected to both FSX instances the migration is taking place in: one being when it receives the vehicle deletion event from an instance and deletes its associated state even though it reappears in another instance and thus the vehicle gets its ATC state reset, and the other when the vehicle reappearance in an instance is detected before the deletion in another instance resulting in the reappearance being ignored and the vehicle being deleted from the ATC.

So, the solution would be for the distribution agent to warn the ATC agents connected to neighboring instances that certain vehicles will migrate and wait for their acknowledgment before proceeding, as shown in Fig. 6.1. By warning the ATC of upcoming migrations, the ATC would wait for a while when detecting a deletion or appearance of a vehicle ID it knows would eventually migrate. If it detects a corresponding appearance or deletion, respectively, of the same vehicle ID before timing out then it would ignore the deletion and carry on as before and both issues would be fixed.

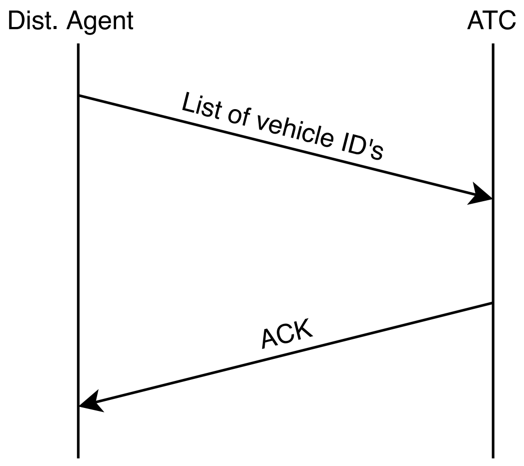


Figure 6.1: ATC is warned of vehicles that will eventually migrate

6.2.5 Temporal Synchronization

The current implementation of temporal synchronization has a fault. A simulation cannot run at a higher rate than 4x, otherwise problems arise in it, namely AI vehicles disappearing, but the temporal synchronization algorithm works by either increasing or decreasing the simulation rate in the FSX slave instances while never modifying the master. This means that if a simulation is globally running at 4x and a slave falls behind the master, it cannot speed up to catch up to the master. A solution is to allow the master to pause itself in such cases until the slaves can catch up.

The communication flow for the updated temporal synchronization protocol herein proposed is displayed in Fig. 6.2. It is the same as the original protocol until the reception of the 3rd message but if on this reception the slave detects it already is running at 4x by default (not by the result of being sped up for another adjustment already) and it requires a speed up, then instead of speeding up it will calculate the time during which the master should slow down and it will send a message back to it with this calculated time. The master receives this message and slows down for the requested time. As the master slows down and the other slaves did not have to catch up, the original algorithm will act to slow them down. With this, the simulation is now synchronized.

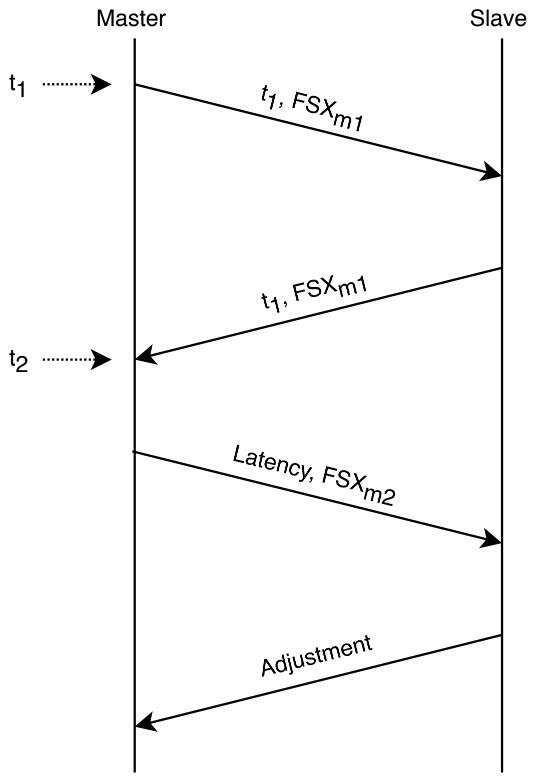


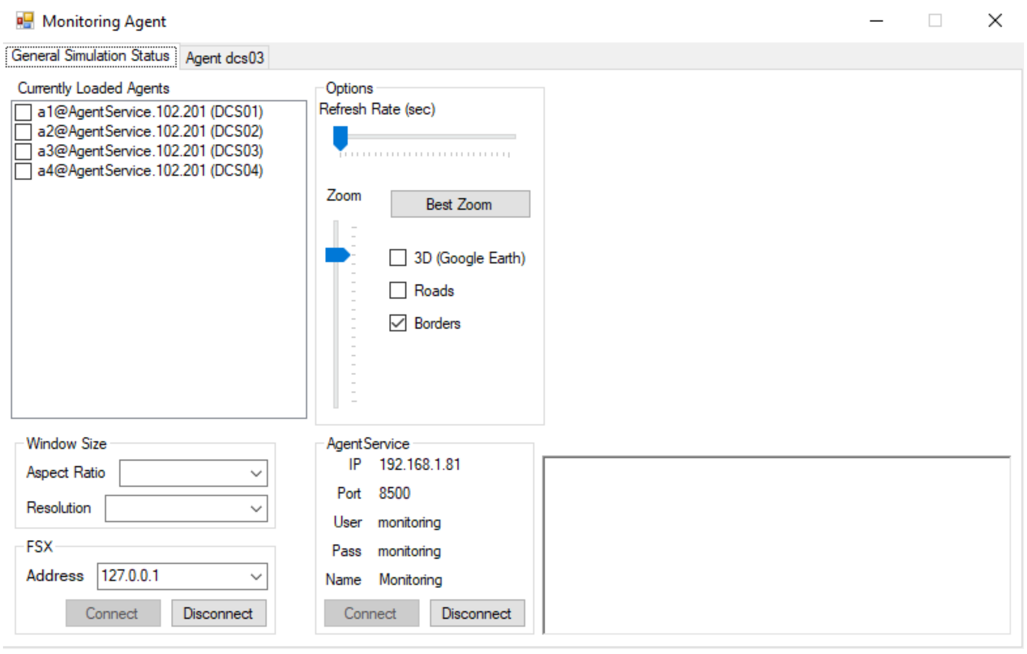
Figure 6.2: Proposal for the temporal synchronization protocol's communication flow if global simulation rate is 4x

6.2.6 Other Platform Components

Two other platform components are missing the distributed simulation treatment: the Disturbances Manager and the Monitoring Tool.

Although FSX can't simulate disturbances to the required fidelity by itself, it still has visual assets to display such as fire. One of the objectives of the Disturbances Manager is to associate with FSX in order to display these assets on the disturbances the manager is simulating. It also communicates with FSX to know which vehicle agents are within the area of the disturbance. Hence, the manager would also need to be modified to display these assets and to be able to get all the vehicles within an area across all FSX instances which share the disturbances' locations, connecting to multiple instances similarly to how the ATC agent connects to all instances overlapping with its coverage area. Unfortunately, the Disturbances Manager was not given as much priority as the remaining adaptations and thus could not have been given a chance to be modified. This stems in part due to the concurrent thesis of Damasceno [11] which actively involves improving the Disturbances Manager and would require some level of cooperation, and also from the fact that the Disturbances Manager is not as much of a fundamental component to running simulations as are the Control Panel, ATC Agents, and Vehicle Agents; after all, simulations can be run without any environmental disturbances.

The other component that was not modified is the Monitoring Tool, displayed in Fig. 6.3. It'd have been very useful even during the implementation of the other components to have a tool dedicated to visualizing the state of the simulation. However, this tool isn't even functional. Figure 6.3 shows the Monitoring Tool opened after starting a simulation in only one FSX instance with 5 vehicles. Only 4 are displayed because those aren't even the real vehicles, they're hardcoded. There should also be a map on the right, but it's not shown. Since it's in such a precarious state, it was given the lowest priority and was not worked on.



Currently Loaded Agents
☐ a1@AgentService.102.201 (DCS01)
☐ a2@AgentService.102.201 (DCS02)
☐ a3@AgentService.102.201 (DCS03)
☐ a4@AgentService.102.201 (DCS04)

☐ 3D (Google Earth)
☐ Roads
☑ Borders

Aspect Ratio
Resolution
Address 127.0.0.1

AgentService
IP 192.168.1.81
Port 8500
User monitoring
Pass monitoring
Name Monitoring

Figure 6.3: Monitoring Tool

References

[1] ADLINK Technology Inc. Vortex DDS Supports Teledyne Brown Engineering on the Objective Simulation Framework. 2014. Available online at https://www.adlinktech.com/en/vortex-dds-client-Teledyne (accessed 2020-01-17).

[2] Air Line Pilots Association International. Improving Commercial Aviation Safety in the Far North. 2019. Available online at http://www.alpa.org/-/media/ALPA/Files/pdfs/news-events/white-papers/white-paper-improving-commercial-aviation-safety-far-north.pdf?la=en (accessed 2020-07-02).

[3] Airport Technology. NAV CANADA improves air traffic management with RTI platform. 2013. Available online at https://www.airport-technology.com/uncategorised/newsnav-canada-improves-air-traffic-management-rti-platform/ (accessed 2020-01-16).

[4] João Almeida. *Simulation and Management of Environmental Disturbances in Flight Simulator X*. Master's thesis, University of Porto, 2017.

[5] Modelica Association. Functional Mock-up Interface for Model Exchange and Co-Simulation. 2019. Available online at https://fmi-standard.org/downloads/ (accessed 2019-12-09).

[6] Csaba A. Boer, Arie de Bruin, and Alexander Verbraeck. Distributed simulation in industry - a survey Part 3 - the HLA standard in industry. In *Proceedings of the 2008 Winter Simulation Conference (WSC'08), Miami, FL, USA*, pages 1094–1102. IEEE, dec 2008. DOI: 10.1109/WSC.2008.4736178.

[7] Tiago Carvalho. *Data Collection and Analysis in a Distributed Simulation Platform*. Master's thesis, University of Porto, 2020.

[8] CloudAMQP. What is AMQP and why is it used in RabbitMQ? Available online at https://www.cloudamqp.com/blog/2019-11-21-what-is-amqp-and-why-is-it-used-in-rabbitmq.html (accessed 2020-06-30).

[9] Angelo Corsaro, Gerardo Pardo-Castellote, and Clark Tucker. DDS Interoperability Demo. 2009. Available online at https://www.slideshare.net/GerardoPardo/dds-interop-demo-rtew09 (accessed 2020-07-01).

[10] Davide Costa. *Secure Communication in a Distributed Simulation Platform*. Master's thesis, University of Porto, 2020.

[11] Rafael Damasceno. *Co-Simulation Architecture for Environmental Disturbances*. Master's thesis, University of Porto, 2020.

[12] V. Daniel Elvira. Impact of detector simulation in particle physics collider experiments. *Physics Reports*, 695:1–54, jun 2017. ISSN: 03701573. DOI: 10.1016/j.physrep.2017.06.002.

[13] Chen Dong, Tao Chao, Songyan Wang, and Ming Yang. Distributed simulation method for homing missiles guidance, navigation, and control. In *Communications in Computer and Information Science: Proceedings of the Asia Simulation Conference 2012 (AsiaSim 2012), Shanghai, China*, volume 323, pages 463–471, oct 2012. DOI: 10.1007/978-3-642-34384-1_55.

[14] Richard M. Fujimoto. Parallel and distributed simulation systems. In *Proceedings of the 2001 Winter Simulation Conference (WSC'01), Arlington, VA, USA*, volume 1, pages 147–157. IEEE, dec 2001. DOI: 10.1109/WSC.2001.977259.

[15] Daniel Garrido. *Fault Injection, Detection and Handling in Autonomous Vehicles*. Master's thesis, University of Porto, 2019.

[16] Simon Gorecki, Youssef Bouanan, Gregory Zacharewicz, Judicael Ribault, and Nicolas Perry. Integrating HLA-Based Distributed Simulation for Management Science and BPMN. *IFAC-PapersOnLine*, 51(11):655–660, jan 2018. ISSN: 24058963. DOI: 10.1016/j.ifacol.2018.08.393.

[17] Akram Hakiri, Pascal Berthou, and Thierry Gayraud. Addressing the challenge of distributed interactive simulation with Data Distribution Service. In *Proceedings of the 2010 International Simulation Multi-Conference, Ottawa, Canada*, pages 103–111, jul 2010.

[18] Peter Heywood, Steve Maddock, Jordi Casas, David Garcia, Mark Brackstone, and Paul Richmond. Data-parallel agent-based microscopic road network simulation using graphics processing units. *Simulation Modelling Practice and Theory*, 83:188–200, apr 2017. ISSN: 1569190X. DOI: 10.1016/j.simpat.2017.11.002.

[19] Gene Hudgins. Test and Training Enabling Architecture (TENA) and the Joint Mission Environment Test Capability (JMETC). 2016. Available online at `https://www.tena-sda.org/attachments/TENA-JMETC-ITEA-Tutorial-2016-05-DistA.pdf` (accessed 2020-07-01).

[20] IEEE. 1516-2010 - IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA). Standard, 2010.

[21] Shafagh Jafer, Qi Liu, and Gabriel Wainer. Synchronization methods in parallel and distributed discrete-event simulation. *Simulation Modelling Practice and Theory*, 30:54–73, 2013. ISSN: 1569190X. DOI: 10.1016/j.simpat.2012.08.003.

[22] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(3):404–425, jul 1985. ISSN: 0164-0925. DOI: 10.1145/3916.3988.

[23] Rajive Joshi and Gerardo Pardo-Castellote. A Comparison and Mapping of Data Distribution Service and High-Level Architecture. 2006. Available online at `https://www.rti.com/hubfs/docs/Comparison-Mapping-DDS-HLA.pdf` (accessed 2020-07-04).

[24] Mohamed M. Kamel, Karim M. Ali, Mohammed A.H. Abozied, and Yehia Z. Elhalwagy. Simulation and modelling of flight missile dynamics and autopilot analysis. In *Proceedings of the 18th International Conference on Aerospace Sciences & Aviation Technology*

(ASAT-18), Cairo, Egypt. Institute of Physics Publishing, oct 2019. DOI: 10.1088/1757-899X/610/1/012033.

[25] Dohyung Kim, Hyun Shik Oh, and Seong Wook Hwang. A DDS-based distributed simulation for anti-air missile systems. In *Proceedings of the 6th International Conference on Simulation and Modeling Methodologies, Technologies and Applications (SIMULTECH 2016), Lisbon, Portugal*, pages 270–276. SciTePress, jul 2016. DOI: 10.5220/0005980402700276.

[26] Dohyung Kim, Ockhyun Paek, Taeho Lee, Samjoon Park, and Hyunshik Bae. A DDS-Based Distributed Simulation Approach for Engineering-Level Models. In *Proceedings of the 2014 Winter Simulation Conference (WSC'14), Savannah, GA, USA*, pages 2919–2930, dec 2014.

[27] Tomáš Králícek. *Building an air traffic simulation platform using open-source components*. Master's thesis, Masaryk University, 2011.

[28] Martin Levesque and David Tipper. ptp++: A Precision Time Protocol Simulation Model for OMNeT++ / INET. In *OMNeT++ Community Summit 2015*, Pittsburgh, 2015.

[29] Björn Löfstrand. Ledningsträningssystem i Viking-övningen – HLA och NATO designregler i praktisk tillämpning. 2016. Available online at http://sesam.smart-lab.se/seminarier/Hostsem16/161124BL.pdf (accessed 2020-01-17).

[30] Eric Luiijf and R. Kamper. Time Synchronization in Distributed Simulations. Technical report, TNO Physics and Electronics Laboratory, The Hague, 1996.

[31] Michael M. Madden and Patricia C. Glaab. Distributed Simulation using DDS and Cloud Computing. In *Proceedings of the 50th Annual Simulation Symposium (ANSS 2017), Virginia Beach, VA, USA*, volume 49, pages 24–35. Society for Modeling and Simulation International (SCS), apr 2017. DOI: 10.22360/SpringSim.2017.ANSS.009.

[32] Roger McFarlane. An Introduction to the HLA Part 1. Available online at http://www2.econ.iastate.edu/tesfatsi/HLAIntro.RMcFarlane.pdf (accessed 2020-07-02).

[33] Microsoft Corporation. ESP SDK Overview. 2008. Available online at https://docs.microsoft.com/en-us/previous-versions/microsoft-esp/cc526948(v=msdn.10) (accessed 2020-01-24).

[34] Military & Aerospace Electronics. Thales Naval extends partnership with PrismTech, use of OpenSplice DDS middleware. nov 2007. Available online at https://www.militaryaerospace.com/home/article/16724974/thales-naval-extends-partnership-with-prismtech-use-of-opensplice-dds-middleware (accessed 2020-07-02).

[35] Missile Defense Agency. Objective Simulation Framework. 2015. Available online at http://web.archive.org/web/20170107024147/https://www.mda.mil/global/documents/pdf/OSF_Product_Sheet_Final_20150818.pdf (accessed 2020-07-02).

[36] NATO. MSG-163: Evolution of NATO Standards for Federated Simulation. Available online at https://www.sto.nato.int/Lists/test1/activitydetails.aspx?ID=16496 (accessed 2020-01-17).

[37] NATO. NATO HLA Certification. Available online at https://www.mscoe.org/nato-hla-certification-home/ (accessed 2020-01-17).

[38] NATO Standardization Office. STANAG 4603, Edition 2. 2015. Available online at https://www.mscoe.org/content/uploads/2017/12/STANAG-4603-Ed.-2.pdf (accessed 2020-07-02).

[39] Object Management Group. Data Distribution Service Category - Specifications Associated. Available online at https://www.omg.org/spec/category/data-distribution-service/ (accessed 2020-01-24).

[40] Object Management Group. What is DDS? Available online at https://www.dds-foundation.org/what-is-dds-3/ (accessed 2020-01-18).

[41] Office National d'Etudes et de Recherches Aérospatiales. CERTI. Available online at https://savannah.nongnu.org/projects/certi/ (accessed 2020-01-28).

[42] OMG. Data Distribution Service (DDS) - Version 1.4. Standard formal/2015-04-10, Object Management Group, 2015.

[43] Larry E. Owen, Yunlong Zhang, Lei Rao, and Gene McHale. Traffic flow simulation using CORSIM. In *Proceedings of the 2000 Winter Simulation Conference (WSC'00), Orlando, FL, USA*, volume 2, pages 1143–1147, dec 2000. DOI: 10.1109/WSC.2000.899077.

[44] Real-Time Innovations. RTI Customer Snapshot: Komatsu - Modular Software Infrastructure to Monitor, Control and Collect Real-Time Equipment Analytics. 2017. Available online at https://info.rti.com/hubfs/Collateral_2017/Customer_Snapshots/RTI_Customer-Snapshot_60007_Komatsu_V7_0718.pdf (accessed 2020-07-02).

[45] Real-Time Innovations. RTI Customer Snapshot: Volkswagen - Solving the Research Collaboration Challenge In Driverless Valet Parking. 2017. Available online at https://info.rti.com/hubfs/Collateral_2017/Customer_Snapshots/RTI_Customer-Snapshot_60016_Volkswagen_V6_0718.pdf (accessed 2020-07-02).

[46] Michael R. Reid. An Evaluation of the High Level Architecture (HLA) as a Framework for NASA Modeling and Simulation. In *Proceedings of the 25th NASA Software Engineering Workshop (SEW-25), Greenbelt, MD, USA*, pages 1–9, Maryland, nov 2000. NASA.

[47] Cristiano Rodrigues, Daniel Castro Silva, Rosaldo J.F. Rossetti, and Eugénio Oliveira. Distributed Flight Simulation Environment using Flight Simulator X. In *Proceedings of the 10th Iberian Conference on Information Systems and Technologies (CISTI 2015), Aveiro, Portugal*, pages 1293–1297, jun 2015. DOI: 10.1109/CISTI.2015.7170615.

[48] Peter Ryan and Lucien Zalcman. The DIS vs HLA Debate: What's in it for Australia? *Proceedings of SimTecT 2003, May 26-29 2003, Adelaide, Australia*, 2003.

[49] Douglas C. Schmidt, Angelo Corsaro, and Hans Van't Hag. Addressing the challenges of tactical information management in net-centric systems with DDS. *CrossTalk*, 21(3):24–29, mar 2008.

[50] Daniel Castro Silva. *Cooperative Multi-Robot Missions: Development of a Platform and a Specification Language*. PhD thesis, University of Porto, 2011.

[51] Steffen Straßburger. Overview about the High Level Architecture for Modelling and Simulation and Recent Developments. *Simulation News Europe*, 16(2):5–14, 2006.

[52] Swedish Armed Forces. Welcome to Viking 18. 2018. Available online at https://www.forsvarsmakten.se/en/news/2018/04/welcome-to-viking-18/ (accessed 2020-01-17).

[53] Thales Group. M-CUBE - Mine Counter Measure Management System. Available online at https://www.thalesgroup.com/en/worldwide/defence/m-cube-mine-counter-measure-management-system (accessed 2020-01-18).

[54] Christian Vecchiola, Alberto Grosso, and Antonio Boccalatte. AgentService: a framework to develop distributed multiagent systems. *International Journal of Agent-Oriented Software Engineering*, 2(3):290, 2008. ISSN: 1746-1375. DOI: 10.1504/IJAOSE.2008.019421.

[55] Wikipedia. Run-time infrastructure (simulation). 2019. Available online at https://en.wikipedia.org/wiki/Run-time_infrastructure_(simulation) (accessed 2020-01-21).

[56] Wikipedia. High Level Architecture. 2020. Available online at https://en.wikipedia.org/wiki/High_Level_Architecture (accessed 2020-01-21).

[57] Zenghui Zhang, Xudong Chai, and Baocun Hou. System security approach for web-enabled HLA/RTI in the cloud simulation environment. In *2011 6th IEEE Conference on Industrial Electronics and Applications*, pages 245–248. IEEE, jun 2011. DOI: 10.1109/ICIEA.2011.5975588.