FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Decentralized Autonomous Vehicles and Control Stations Data Sharing for Partially Disconnected Operation Locations

Eduardo de Mendonça Rodrigues Salgado Ramos



Mestrado Integrado em Engenharia Informática e Computação

Supervisor at FEUP: Pedro Alexandre Guimarães Lobo Ferreira do Souto

Supervisor at Laboratório de Sistemas e Tecnologia Subaquática: Paulo Sousa Dias

July 21, 2020

© Eduardo de Mendonça Rodrigues Salgado Ramos, 2020

Decentralized Autonomous Vehicles and Control Stations Data Sharing for Partially Disconnected Operation Locations

Eduardo de Mendonça Rodrigues Salgado Ramos

Mestrado Integrado em Engenharia Informática e Computação

Approved in oral examination by the committee:

Chair: André Monteiro de Oliveira Restivo
External Examiner: Paulo Sérgio Soares Almeida
Supervisor: Pedro Alexandre Guimarães Lobo Ferreira do Souto
July 21, 2020

Abstract

Marine research is, nowadays, still one of the most challenging research fields in today's science. Reasons for this include factors like the adverse environments, the elevated costs and isolation introduced by the vastness of the earth's oceans.

To produce this work I worked together with Laboratório de Sistemas e Tecnologia Subaquática (LSTS), a research laboratory that develops software and hardware solutions for Marine exploration. Our goal was to develop a software system that could easily be integrated into the existing toolchain and provide a solution to existing problems. Said existing problems concern limitations in data sharing between connected systems, laborious manual processes concerning data sharing and information loss in certain recurring scenarios. After analyzing the existing system, the operation context and the desired functionalities, we defined the expected system as a data replication system capable of tolerating network partitions while working in both local and wide-area networks and being integrated into LSTS' target systems, Dune and Neptus.

We divide the solution into four modules. The core module bridges the interaction of this system with the target systems in the toolchain, it is responsible for receiving and sending updates of the existing system. LSTS uses its application layer protocol for data exchange so the second module is the communication interface provided by the target systems. The third module and by far where most of my work was directed, is the consistency manager. In this module, I implement a solution for consistent data replication capable of supporting network partitions. This solution leverages the Strong Eventual Consistency properties provided by Conflict-Free Replicated Data Types to provide the system with high availability and a guarantee of eventual consistency. I implement State-based CRDTs including the documented Grow-Only Set and Observed-Remove Set, as well as, custom ones built from these, Dictionary and LSTS Plan. Lastly, the election manager is a module required to allow efficient wide-area communication and aims at solving the problem of leader election in a dynamic network where partitions occur.

Testing procedures were conducted under controlled conditions due to the COVID-19 pandemic. Two scenarios were developed by LSTS to simulate real operations. Both scenarios were completed successfully and a comparison in performance between the pre-existing system with and without the developed framework is presented. The preliminary results obtained in a controlled environment are very promising for future, real-world, testing.

Integration in Neptus was limited to the Plan data type. Integration in Dune was relatively simpler allowing for an extended integration to vehicle state data types. The prototype system solved most of the problems initially set, but since it is essentially a proof of concept, it can still be improved. The existing system had no problem when faced with an eventual consistency model, which was an initial concern. CRDTs proved to be an effective solution for high availability when faced with network partitions and eventual consistency fulfills the system's requirements.

Keywords: data replication, eventual consistency, conflict-free replicated data types

Acknowledgements

I wish to express my deepest gratitude to everyone who helped me make this work possible.

Firstly, I would like to thank Pedro Souto, my supervisor, for the outstanding availability and support he showed in answering any of my doubts, propositions and requests. Also remarkable were his efforts in making this document engaging, correct and best deliver the contents of the developed work.

Secondly, I thank Laboratório de Sistemas e Tecnologia Subaquática, for welcoming my work and making their systems available throughout the development of this work. Specifically to Paulo Dias, who closely accompanied my work and showed unparalleled availability and commitment to make this work a reality.

Thirdly, I express my gratitude to my friends and family who were always there to fulfill my needs and listen to my ramblings.

Eduardo Salgado Ramos

"To be neutral does not mean to be indifferent or insensitive.
You don't have to kill your feelings. It's enough to kill hatred within yourself."

Andrzej Sapkowski, Krew elfów

Contents

Abstract i

1 Introduction 1
- 1.1 LSTS Stack . . . 1
- 1.2 Motivation and Goals . . . 2
- 1.3 Approach . . . 4
- 1.4 Results . . . 5

2 Background 7
- 2.1 Distributed Architectures . . . 8
- 2.2 Network Discovery . . . 10
- 2.3 Election . . . 10
- 2.4 Data Replication . . . 12
  - 2.4.1 Consistency Models . . . 12
  - 2.4.2 Conflict-free Replicated Data Types . . . 14
- 2.5 CAP Theorem . . . 16
- 2.6 LSTS Toolchain . . . 17
  - 2.6.1 Neptus . . . 18
  - 2.6.2 Vehicles and Dune . . . 20
  - 2.6.3 Manta . . . 21

3 Automated Consistent Data Sharing 23
- 3.1 System Architecture . . . 23
- 3.2 Consistency in a local network . . . 24
  - 3.2.1 Grow-Only Set (G-Set) . . . 27
  - 3.2.2 Observed-Remove Set (OR-Set) . . . 27
  - 3.2.3 Dictionary . . . 28
  - 3.2.4 LSTS Plan . . . 29
- 3.3 Consistency in Wide-area Networks . . . 29
- 3.4 Local Speaker . . . 32
  - 3.4.1 Model . . . 32
  - 3.4.2 Algorithm . . . 33
  - 3.4.3 Limitations . . . 34
- 3.5 Data Exchange Protocol . . . 35

4 Implementation 39
- 4.1 Integration with Neptus . . . 39
- 4.2 Integration with Dune . . . 44

5 Results **47**
5.1 Individual Module Testing 47
5.2 Testing in real scenarios 48
5.3 Impact on LSTS' Workflow 50
5.4 Resulting Software 51

6 Conclusions and Future Work **53**
6.1 Conclusions 53
6.2 Future Work 54

References **55**

List of Figures

1.1 LSTS' Toolchain Visualization 3

2.1 Neptus Interface with the Plan Editor tool active 18
2.2 Two Noptilus category submarine vehicles 20
2.3 Manta with an antenna and acoustic modem installed 21

4.1 Representation of the 4 modules constituent of this work and their interaction . . 40
4.2 Developed Neptus Interfaces 41
4.3 Representation of the 4 modules implementation in Neptus and their interaction . 42
4.4 Data layers in Neptus and data flow between them 43
4.5 Representation of the four modules implementation in Dune and their interaction 45
4.6 Data layers in Dune and data flow between them 46

List of Algorithms

1 State-based Grow-only Set (G-Set) 27
2 State-based Observed-Remove Set (OR-Set) 28
3 State-based Dictionary 30
4 State-based LSTS Plan 31
5 Initial Election process for any node *N* 34
6 Re-election process for any node *N* 35
7 Improved Election process for any node *N* 36

Abbreviations

AUV	Autonomous Underwater Vehicle
ASV	Autonomous Surface Vehicle
CCU	Command and Control Unit
CRUD	Create, Read, Update, Delete
CRDT	Conflict-Free Replicated Data Type
CS	Critical Section
EC	Eventual Consistency
IP	Internet Protocol
JVM	Java Virtual Machine
LAN	Local Area Network
LSTS	Laboratório de Sistemas e Tecnologia Subaquática
M2M	Machine-to-Machine
OT	Operational Transformation
P2P	Peer-to-Peer
SEC	Strong Eventual Consistency
TCP	Transmission Control Protocol
UAV	Unmanned Aerial Vehicle
WAN	Wide Area Network

Chapter 1

Introduction

Marine exploration is still one of the greatest challenges in today's science. This thesis focus on potentiating the work developed by Laboratório de Sistemas e Tecnologia Subaquática(LSTS), a research laboratory based in Faculdade de Engenharia da Universidade do Porto, which focus on providing hardware and software solutions for marine research. LSTS also conducts its missions in partnership with other institutions even taking part in international collaborative missions that bring together several research groups from different countries.

The focus of this work is on improving the distributed data replication capabilities of the current software by building a platform through which data can be shared automatically and consistently with minimal user intervention. At first sight, such a problem has already been solved for several different scenarios but in the context of LSTS' operation, some limitations arise that make existing solutions less viable or outright non-applicable. The main hurdle to overcome is the support of a very dynamic and weakly connected network. Such network properties originate from the limited communication range in extensive operational areas, limitations in surface to underwater communications and the dynamic nature of vehicles moving in operational courses. The limited time of connectivity to vehicles is also an issue since only when these surface can communication be established. In short, this work proposes to develop a framework that improves on the current data sharing between nodes, establishing a network of consistent replicated data despite adverse connectivity environments.

1.1 LSTS Stack

LSTS and its partners participate in different types of missions which cover a very wide scope. We can define two environments, aerial and marine, some examples of aerial operations include surveillance of forest fires for spread path predictions or surface water temperature scan over extensive oceanic areas. Submarine operations compose the bulk of LSTS research and include several scenarios such as collecting sensory data important to marine biology research and ocean floor mapping, which includes 3D modeling of shipwrecks. To support all these operation scenarios a wide range of hardware and software is required

LSTS' range of hardware includes two main kinds of autonomous vehicles (aerial and submarine), each equipped specifically for an operation scenario. Vehicles are categorized by their capabilities, mainly battery life which directly correlates to vehicle size and maximum operating range, but they all have one thing in common, the autonomous control software and communication devices. Another major hardware item which is not explicitly featured in Figure 1.1, is called Manta. Mantas run the same software as vehicles but in this case, the focus is communication, since they do not require autonomous control or sensory properties. The purpose of a Manta in LSTS' hardware kit is to bridge the communication gap between vehicles and a ground station (located either on land or aboard a vessel), operators' main location during a mission. Mantas provides a flexible and portable communication infrastructure as they can be equipped with a range of radio and ultrasonic transmitter for surface and underwater communication. Furthermore, it is also able to exchange data transparently between IP, acoustic, and satellite links, as well as, using mobile internet networks like 3G and 4G.

In terms of software, LSTS has two solutions focused on local operation (Neptus and ACCU), and one cloud-based solution (Ripples), presented in Figure 1.1. Neptus is the main multi-purpose software that allows for mission planning, data analysis and vehicle operation and configuration. The ACCU, short for Android Command and Control Unit, is a more limited version of Neptus that runs on any Android-based device and is mainly used for live vehicle remote control. Any system running either of these software is referred to as a Command and Control Unit (CCU) since its main goal is to configure, command, and control the vehicles prior to autonomous operation. Ripples is a cloud-based solution useful for data aggregation and dissemination, it has got a web interface where the user can quickly view operating vehicles, weather information and, for example, other marine traffic all in near real-time. The other portion of LSTS' software stack includes Dune, IMC and GLUED, presented in Figure 1.1. Dune is the software that autonomously operates and controls every vehicle used by LSTS. Since vehicles have limited resources, a minimal distribution of the GNU/Linux Kernel is curated and named GLUED, this is the operating system that runs in the vehicles' main processing core in which Dune is executed. Finally, the Inter-Module Communication Protocol (IMC), is an application-level protocol developed by LSTS so that all communication between nodes running their software is standardized.

1.2 Motivation and Goals

To better understand the motivation for this work, let us start with what is missing, could be improved, or would be a valuable feature. Empirical observation and usage of the current system provide insight into some points that, if addressed, could highly improve the ease-of-use and the assurance that no data will be lost in case of an unexpected crash. Exposing two relatively simple scenarios will help understand these limitations. **(1)** A handover between operating teams occurs after one has been operating for an extended period of time. The full picture of the currently operating vehicles must be transferred between instances of Neptus, including executing plans, and various data types that compose a vehicle's state. **(2)** Another scenario is when one vehicle

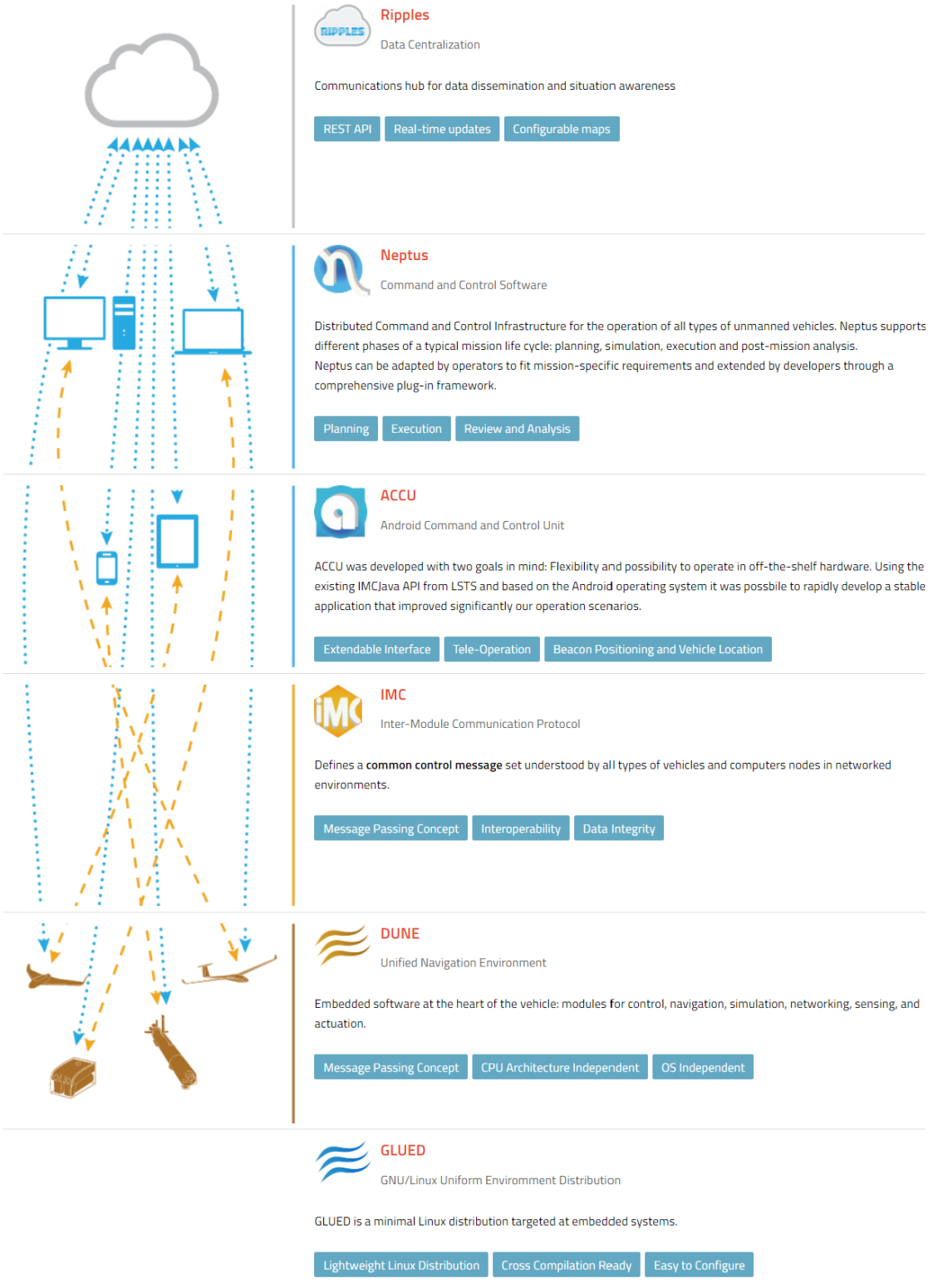


Figure 1.1: LSTS' Toolchain Visualization

performs the role of a data mule to rendezvous with currently operating vehicles to share new data and fetch the existing one. The data mule is commanded from a base operation station to which it returns after completing the rendezvous to share the collected data. I now outline the current problems in these scenarios:

- Sharing information between CCUs is not complete, important information about the vehicle state cannot be shared. This includes accurate positioning, battery level, subsystems running. Excluding vehicle related data some other desirable data types are also not shareable and require the use of external software to do so;
- Sharing data between CCUs and Vehicles is a fully manual process, that involves individual data item sharing;
- There is no stable data storage to prevent information loss in case of an unexpected system shutdown. More precisely, some critical pieces of information that are lost include vehicle properties such as accurate positioning, battery level and active subsystems. In extreme cases even the plans that are currently being executed can be lost;
- Data sharing between vehicles is a laborious task that is custom implemented for every occasion;

Facing these issues takes time and attention away from the real goals of operations. Improving these would bring more consistent and less risky operations allowing researchers to focus on the main operational goals and their work. The problems' origin can mostly be traced to the limitations the system presents in automated data sharing and, indirectly, communication between components.

With this work, we aim to develop a framework that can be integrated into any component of LSTS' toolchain and would allow these systems to consistently replicate data between them. Replication provides better performance or even availability to data and helps prevent data loss, the innovative proposition is that we aim to achieve this in a partially disconnected environment with a very dynamic network. The system should also be able to cope with the different communication infrastructures that LSTS operations involve, meaning being able to optimize local and wide-area communication. A successful solution to these problems would, not only, considerably improve LSTS' operational efficiency but could also be further adapted to other systems.

1.3 Approach

In this work, I targeted two big issues in LSTS' toolchain: information loss; information sharing. With the correct solution, both of these could be simultaneously solved. Data replication would prevent information loss to a significant level by sharing and keeping replicated information over several machines. To share data the underlying communication infrastructure was critical and so it was carefully analyzed. I could separate local-area and wide-area communications in terms of

the network properties assumptions. Nonetheless, these were not significant to require a different solution for each context, only slight adjustments would do.

I make use of Conflict-free Replicated Data Types (CRDT), a relatively recent technology in the field of data replication that is built with object-like data types which guarantee eventual consistency once all systems exchange the expected messages and provide near infinite availability independent of a system's network state. CRDTs can be state-based or operation-based, both of which come with their requirements, benefits and drawbacks. I use a state-based only approach as this allowed for a solution that did not require causal delivery, which would be extremely costly to achieve in a highly dynamic network.

To optimize wide-area communication between local-area systems, a local leader, dubbed speaker, is elected and takes on the responsibility of sharing local data and receiving external data to share in the local network. The election algorithm used was an adaptation of existing ones to best suit the assumptions of the local network and provide a stable leader to avoid overhead work intrinsic to a change in leadership.

Concerning the implementation, together with LSTS, we defined data categories, based on data features, to establish different standard types of data we wanted the systems to support. To achieve consistent data sharing I made use of documented state-based CRDTs to build my CRDTs and provide a solution that would be able to cover all the standard data categories defined before.

Flexibility was key to the integration in the different components of LSTS' toolchain, both current and future. Unfortunately, achieving a high degree of integration with Neptus proved harder than initially predicted so instead of providing a full integration I opted for maintaining the flexibility and integrate only one of the main data objects in the toolchain, the Plan. Other CRDTs were implemented based on existing data types future outlooks on useful data to share, however, these were only verified in artificial testing without the full integration.

1.4 Results

The work was deemed successful as CRDTs proved to be a great solution when working with dynamic networks and partitioned environments. This work resulted in a highly independent framework that can be quickly adapted to other systems and guarantee eventual consistency as long as good system integration is achieved.

Testing was conducted in a controlled environment using Wi-Fi connections with more than one machine, no physical testing with vehicles has been conducted at the time of writing because of the COVID-19 pandemic and the limitations in place. The testing procedures, initially, targeted individual components such as the full range of CRDTs and the election algorithm. At a later stage, testing was then expanded to simulated scenarios devised by LSTS. A detailed comparison is presented assessing the testes scenarios using the pre-existing system with and without the developed data-sharing framework. Significant improvements to the workflow and user dependency were verified.

At first, the use of CRDTs raised some concerns about the negative impact its consistency model would have on the system. I verified that some ambiguity is introduced when concurrent edits occur and it is not desirable to just merge or choose one of the instances. Another negative note about CRDTs is their complex implementation, however, most of the time, it can be overcome by the aggregation of different documented ones. The positive points, on the other hand, are overwhelming, on LSTS' toolchain updated data is key for optimal results and the implemented systems allow any machine to automatically replicate data to/from other machines in the same network, making data flow seamless to the user.

The resulting software architecture is also worth noting. We achieved a framework-like architecture, that can easily be adapted to any system. Four modules can be distinguished and only two of then are strictly application dependent. A communication module that may already exist or be implemented from scratch and adhere to application layer protocol. The election manager is fully independent of the integrating system but uses the communication module to send messages. The Core module manages the election manager and bridges the interface between the existing system data and the consistency manager. Finally the consistency manager depends on the data provided by the Core and is responsible for guaranteeing consistency between nodes. This architecture was proven by the effortless porting of the solution developed for Neptus to Dune, a radically different system, that shares some data types and the same application layer protocol.

The rest of this document is structured as follows. A background analysis and literature review mainly on the topics of distributed data replication and distributed systems architecture is presented in the first part of Chapter 2, still on this chapter, election algorithms, network discovery and CRDTs are also covered. A deeper look at LSTS' systems is presented in the second part of Chapter 2. Chapter 3 focuses on the methods used to solve the problems targeted by this work, starting with how data is shared in both local, including the choice of CRDTs. Secondly, I cover how the local network solution was adapted to wide-area networks. Finally, still in Chapter 3, I expose the solution implemented to elect a local speaker and the data exchange protocol. Chapter 4 covers the concrete implementation of the solution to work with Neptus and Dune. The impact on LSTS' workflow is discussed in detail in Chapter 5. Conclusions are drawn from the work accomplished and suggestions for future work are discussed in Chapter 6.

Chapter 2

Background

Distributed Systems, the area of research that aims at developing complex systems where several computers connected over a network must work together to achieve a common goal and *appear as a single coherent unit to its users* (Tanenbaum and Van Steen, 2007). LSTS' tool-chain itself is already a distributed system, any number of CCUs interact between themselves and with vehicles to form a cohesive working system that potentiates marine exploration.

A Distributed System is highly complex with several different and sometimes uncontrollable parts. In general, we can think of a distributed system as an aggregation of these, usually tightly coupled, modules:

- Communication - In most cases, this is the major source of limitations and problems, it refers to how systems can communicate with each other. In most cases this will be out of reach for the developers to make any changes and control its behavior, hence being a source of limitations. Distributed systems can even be developed so far that as to adapt to different communication mediums and still work without a major negative impact. The communication infrastructure can be reduced to a few crucial parameters which include the time range of possible message delivery delay and whether messages can be lost or duplicated. Establishing a good picture of the communication available is a crucial starting point for any work in distributed systems.

- Architecture - This point covers how participants relate to each other. Depending on what we aim to achieve the privileges of each participant of the system can be different, we might have participants that only save and provide data, some might just do processing while others can just serve as an inspecting tool. This allows for the most diverse hierarchies and layouts and highly impacts the efficiency of the system. To better explain this last point, lets assume we have one participant in Europe who wants to read some data, he should read data from the closest possible data storage instead of using one that is on another continent, so the network should be able to define some sort of structure as to guide the participants about where to find resources.

- Data - By far one of the most complex problems in modern distributed systems, systems rely on this aspect on varying levels, one can be more data focused like cloud-based storage which lets you store, change and read your files from any point in the globe with very good consistent results and others like a distributed computing network which does not support data-heavy operations but still need to guarantee that work is not being double processed or lost. Both systems have extremely different needs when it comes to data accessibility and consistency which gives rise to several different methods and algorithms that help a system manage its data while keeping to a stricter/looser set of guidelines.

In this Chapter, we will go over existing literature, concepts inherent to the problem and analyze existing technologies and proven systems that could help build, support and explain the solution that solves the problem. No works were found that addressed a similar challenge, this can be attributed to the detailed and customized specification of the problem at hand and the fact that we are limited to the use of existing software. Instead, research was conducted following the topics presented above as a guideline. Since the communication layer is already in place and provided by Neptus and Dune no research was conducted in this direction, in Section 2.6 we cover this communication layer and other topics related to the existing system.

2.1 Distributed Architectures

In this work there were no architecture specifications so it is necessary to select an appropriate architecture, Neptus and Dune already work in their architecture but it does not mean we are tied to it. To define the architecture that works best for our objective we will start by analyzing one big differentiating factor when it comes to distributed systems, their architecture. This is an important first step when analyzing existing research to find solutions that are adequate for our proposed architecture. Also, defining an architecture is a key step in analyzing a system's predicted qualities and its ability to accommodate changes. Bass et al. (2003) defend that software architecture is determinant in enabling or inhibiting a system of certain features. One key aspect, before we start, is to differentiate system from software architectures. While software architectures concern how different components of a software interact with each other, independent of their physical location (can even be on different machines), system architecture covers how these software components' instances are placed in different machines. This section will cover both types starting with software.

Tanenbaum and Van Steen (2007) identify four important software architectural styles: Layered, Object-based, Data-Centered, Event-Based. From these we will focus mainly on the data-centered and event-based styles as these are the more pertinent for this work. The data-centered style is defined by a central common repository which is used as the main means of communication between processes, the most common application of this style is a distributed file system. As for the event-based style, processes communicate through a shared channel where they can publish and subscribe to Events, this being the main means of interaction. Furthermore, the author defends that a combination of both these styles can be an efficient solution to build what he refers

to as *shared data spaces*. The combination of these architectures would allow for the decoupling in time of every system since a system no longer need be active when an event is published to observe its impact. For this joint architecture to work two things are necessary, a persistent data space and all participants must have access to it.

The author continues to develop the topic by analyzing exiting system architectures and makes the distinction between centralized, decentralized and hybrid. A centralized approach defines a partitioning in at least two parts of the full application logic, often referred to as client-server architecture. The amount of split is application dependant and there have been several successful examples of different split levels. The split levels refer to how the complete application components: user interface, application logic, data storage; are distributed over the existing machines. The way these can be split between client and server can also be applied to being split between different machines on the server side in full transparency for the client. An important aspect to clarify is that centralized does not mean non-replicated, the inherent feature of a centralized architecture is that some part of the application is in exclusive control of a subset of the machines that compose the system. We can think of these networks as being like a dictatorship where a group of machines has exclusive control over the application, independently of how many or few machines they control. If we are looking for a more "democratic" solution we must look into decentralized architectures.

The characteristic feature of a decentralized architecture is that all participants are equal, acting both as a client and server. Contrary to centralized architectures, there is no expected data flow as node communication is symmetric, to solve this another layer of structuring is required. Overlay networks, as they are referred to, represent the processes as nodes and the possible communication channels between these as edges. Overlay networks can be structured or unstructured, in sum, structured networks provide deterministic identifiers for both nodes and data items making it possible for any process to locate the data item it is looking for and who is currently holding it. There are many more complex mechanics to the workings of structured peer-to-peer networks but they are not important for this work. Structured networks entail high costs and complexity in defining the deterministic processes to obtain identifiers and whenever a node leaves or joins the network it must transfer all data items it is holding or should hold. As such, in a very dynamic network like the one the problem presents this is far from an ideal solution. Unstructured peer-to-peer networks construct their overlay network in a close to random manner, meaning there are no guarantees of data exchange flows raising the problem of network flooding when a request needs to be made. At the same time, since there is no information about the full network, some nodes might become overloaded if they contain a lot of adjacent peers. However there is not a clear cut separation between these two types of network structuring, with careful management of the adjacent peers, an unstructured network can become organized enough to allow for balancing of the number of connections and prevent random look-ups and network flooding. One of the most popular methods for data lookup in unstructured peer-to-peer networks is the introduction of super-peers. These may or may not have extra responsibilities like maintaining an index of data items, these extra responsibilities introduce a soft hierarchy to a network otherwise flat but with

the advantage of having peers which any other peer can rely on for operation that require higher coordination (data lookup is one of the most common use cases). Since these peers are very important for the network to work at its best they are expected to have very high up-times, but in case some fail, the issue of selecting another peer to elevate to super-peer class arises.

2.2 Network Discovery

Network discovery is the process of mapping other processes to which a connection can be established. This challenge can be separated into two different approaches for this work. For this work, we intend the protocol to work in a Local Area Network (LAN) environment and a Wide Area Network (WAN) environment. In both environments, possessing the knowledge of which systems are actively connected provides significant insight into the network allowing for potentially simpler and more efficient solutions. When specifically considering a wide-area environment, it is even more important, as solutions like multicast or broadcast, are not an option. Discovering other active systems over a wide-area connection is critical to share data with these.

In the case of a LAN environment, we can use the multicast or broadcast capabilities of the Internet Protocol (IP), in fact, Neptus already accomplishes local network discovery and each instance has access to a list of currently active systems, this will be further covered in Section 2.6.

As for a wide-area environment, there are not many options available, given our assumptions: systems can connect and disconnect at any point in time and are few in total so creating a stable network is not an option, IP solutions like multicasting are also blocked in wide-area networks unless some kind of tunneling is being used which defeats the purpose of discovery. One of the most popular solutions is to rely on well-known machines which can provide a list of active systems to which a connection can be established. These well-known machines are expected to keep a list of connected machines and have very high up-times to provide the discovery service all the time.

2.3 Election

The election of a leader break what could otherwise be a flat hierarchy and delegates one leader to perform the role of coordinator, initiator or just have an elevated role. In this Section, we will go through some of the more traditional election algorithms, the concept of a stable leader and how to achieve it finishing with a small reflection about the topic.

One of the most famous and flexible algorithm for leader election is the Bully algorithm (Garcia-Molina, 1982), this algorithm assumes every process is defined by a unique identifier. It can be described in the lines of the bigger identifier forcing smaller ones into submission. It is extremely simple but effective if the communications network proves reliable. A process that identifies the leader as being unresponsive, broadcasts an election message, processes with higher IDs respond to the broadcaster and broadcast another election themselves. One a broadcaster does not obtain an answer it means he is the one with the highest active ID in the network. At any point

recovering processes can send a coordinator message to bully smaller IDs into accepting him as the leader.

Awerbuch (1987) makes one of the earliest mentions in the literature that an election algorithm can be solved by the construction of a tree-like structure. This is an important advancement due to the properties of tree structures and how these can be balanced and healed after losing a node. The leader will be whoever ends up at the root of the tree but unfortunately building a tree-like structure is very complex in dynamic and unreliable networks.

Aguilera et al. (2001) introduce the idea of a stable leader election, this concept can be reduced to the fact that as long as a leader maintains a correct state it will continue to be the leader. A more complete definition of stability is that if process p is the leader at time t and is accessible during the period $[t-k\delta, t+1]$ at time $t+1$ process p will still be the leader. This work also includes several solutions for a stable leader election algorithm when faced with different assumptions. The basic proposed algorithm builds upon Ω, introduced by Lamport (1998), outlining two problems that do not guarantee stability and solving these. Two solutions are proposed, one that works with the assumption of no message loss and a second one that is resilient to message loss. The latter one, however, includes an added level of complexity as it requires clock synchronization with a bounded drift to eliminate delayed messages. The advantage of having a stable leader in a dynamic network is preventing some costly setup work for the leader to perform its elevated role.

Tanenbaum and Van Steen (2007) surveys some well-known election algorithms. The author refers to the work by Vasudevan et al. (2004) when covering wireless environments. Vasudevan proposes a solution specialized in mobile, ad hoc networks. Said solution is capable of electing a leader in conditions where node mobility, node crashes, link failures, network partitions and merging of partition occur. One of the big innovative factors is the capacity for choosing the best leader and it works with the concept of neighbours to propagate an election message through the network, each node that receives an election message will then also propagate back another message with the best result found for a network leader.

Another interesting algorithm is presented by Franceschetti and Bruck (1998), because it covers election in a complete graph model and, also important, for the problem at hand it addresses partial link failures which is a scenario we might be dealing with. Again several solutions are presented that build upon a base solution, the author defends a solution focused on agreeing on the network state instead of the usual selection of a leader. The algorithm works with candidate and acceptance messages, in short, a node that detects a link failure to some other node will share it with the network through a candidate message, to which nodes will respond with an accept message if they also detected a failure. A node is elected if it receives the same number of accept messages as the number of connected links. Since all processes can send a candidate message, nodes will only answer to the system with the smaller ID so that a common decision is made about the selected process. This system guarantees that all nodes agree on a failed process before it is declared as failed and a new election takes place, worth noting is that only the leader knows that it has been elected.

2.4 Data Replication

Data Replication refers to the process of storing, accessing and editing data on two or more locations. In this section we will analyze the basic principles of data replication and how to define the level of consistency a system provides, to help us with that we introduce four of the most well-known consistency models definitions.

2.4.1 Consistency Models

The work on data consistency was first aimed at analyzing multi-processor shared-memory systems. This research aimed at defining models and methods to make the behaviour of write and read operations predictable and optimized when processes used a shared memory space. Another very popular application of these models were caches, data stores that kept replicated data from each other. Fast-forward some years and the same research was now being applied in the context of shared data distributed systems. The principles are the same, replicated data stores that all should store the same exact data, and, although the implementations are very far from each other, it does not concern the model. Tanenbaum and Van Steen (2007) survey, the more up to date, consistency models which they separate in terms of data-centric and client-centric.

2.4.1.1 Data-Centric Models

Data-centric models focus on the data storage interaction with accessing processes, models should provide a certain set of rules that if correctly implemented by both the processes and data storage should guarantee a certain level of consistency. Starting with the more strict and loosening constraints we will start with sequential consistency.

Sequential Consistency was first defined by Lamport (1979) in the context of correctly executing multi-process programs in multi-processor systems with shared memory. The author defines sequential consistent data stores as those that satisfy the following: *the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program*. A complex concept that is also later covered by Tanenbaum and Van Steen (2007) in a more in-depth analysis. In other words, a sequentially consistent data store is one that for a multi-processor system (possibly with processors in different machines) guarantees that *all processes see the same interleaving of operations* no matter their ordering, a sequence of operations is established for all processors.

Causal Consistency is a model introduced by Hutto and Ahamad (1990) with the goal of weakening the Sequential Consistency model to achieve better performance. They base their model on the work of Lamport (1978) and the notion of potential causality. They suggest the restriction of sequencing only to operations that are causally related, in other words, any pair of operations

where a relation of potential causality can be established must respect the sequential model. Operations outside this scope, called concurrent, may differ in the order they are observed by each process.

Operations Grouping is the consequence of evolution and the shift in the target for the data-centric models. While sequential and causal consistency models concern individual operations or memory access (as they were developed to work at a hardware level) this level of granularity is not practical for application programming so the concept of operation grouping was outlined. Read and Write operations can be grouped into critical sections (CS), a critical section includes a set of operations that are guaranteed to have exclusive access to the data that is being operated on and the most up to date values and thus lowering the level of granularity. In reality, the simple interface of a CS hides a lot of precise working for guaranteeing exclusive access and updated data, synchronization variables helped processed manage how had effective access to the what shared data and tight protocols defined how synchronization variables could be claimed to have exclusive (usually for write operations) and non-exclusive (usually for read operations) access to the data store.

2.4.1.2 Session-based

To understand Session-based models, also called client-centric in Tanenbaum and Van Steen (2007), we need to first review their origin, which stemmed from the introduction of Eventual Consistency (EC) shared-data systems. Eventual consistency focus on systems where several processes have permissions to execute write operations and how quickly these updates are actually required to be viewed by the reading processes. In essence, this model allows systems to reach high levels of inconsistency, assuming they can tolerate them. The inconsistency derives from allowing concurrent updates to the same data object which may lead to different processes (running on different machines) having different values for that object. If once the updates stop, replicas progress towards and eventually reach a consistent state then the system guarantees eventual consistency. An obvious requirement for eventual consistency is thus an update propagation mechanism that guarantees eventual propagation to all replicas. One of the biggest issues when implementing an eventually consistent system is how conflicts can be resolved and how often this process is expected to occur. In case of seldom concurrent writes and easy resolution of the conflicts, eventual consistency is a very cost-effective model. However, if conflict resolution is a complex and heavy process, expected to happen often, eventual consistency can quickly become inefficient.

Another problem introduced by the weakening of consistency is how clients perceive data changes. If a client can connect to different replicas and these are allowed to have different values for the same data object this difference will also be reflected on the client interface. The introduction of Session-based consistency models focuses on solving this problem within the scope of a session although in some cases they also affect other sessions. no guarantees are established when it comes to accesses by different clients. The seminal work on these models was conducted in the development of the Bayou database system (Terry et al., 1995, 1994; Demers et al., 1994) and originated these four guarantees:

- *Monotonic Reads* - For a given arbitrary process read operation either return the last read value or a more recent one for the same data item;
- *Read Your Writes* - For a given arbitrary process, a write operation will always be observed by a successive read operation on the same data item;
- *Writes Follow Reads* - If a read operation R1 precedes a write operation W2 in a session, then any process which already knows of W2 must also have the knowledge of any write operations present at the process which performed R1 and state that W2 follows all those write operations. This guarantee is different from the previous two, in that it also affects clients in other sessions;
- *Monotonic Writes* - For a given arbitrary process only after a write operation has been completed can any consecutive writes be executed to the same data item; Like *Write Follows Reads*, this guarantee also affects users in other sessions.

Another important aspect of the Bayou System for this work lies in the way it implements a process of anti-entropy to resolve conflicts. This process is executed pair-wise between two communicating replicas, it starts by performing an analysis and detection of conflicts followed by a resolution process adequate for the problem, which at times results in a highly undesirable rollback.

2.4.2 Conflict-free Replicated Data Types

Eventual consistency is a good flexible solution for systems where consistency is not critical but even when applied with session-based models still presents some serious problems. The allowance for high availability comes at the compromise of consistency which originates replicas with conflicting values. When faced with concurrent modification in different replicas, a complex and/or costly resolution processes must be executed and there is the possibility of this ending up in a rollback if it's not possible to resolve the conflicts.

The bulk of this work is toward resolving the delicate balance between a strong enough consistency while keeping the system highly available, CRDTs are one of the more recent investigation topics when it comes to distributed data replication that tackles eventual consistency problems while keeping a very high availability level. Although the concept of CRDT is recent its precedent work is quite extensive. Wuu and Bernstein (1984) studied the problem of a replicated log and key-value stores. The area of Operational Transformation (OT) potentiated concurrent modification of sequence-like structures and was the base for what today are Google Drive's collaborative solutions. Preguiça et al. (2009) introduce the abstraction of a CRDT for the first time, this concept is then further formalized in Shapiro et al. (2011c,b).

The definition of Strong Eventual Consistency (SEC) (Shapiro et al., 2011b) is crucial for the concept of a CRDT, it takes the definition of Eventual consistency (EC), which states that *an update delivered at some correct replica is eventually delivered to all correct replicas* and *correct replicas that have delivered the same updates eventually reach equivalent state*, after all

methods terminate EC is guaranteed. SEC builds on this definition with an aspect called strong convergence: *Correct replicas that have delivered the same updates have equivalent state*. The difference lies in the fact that SEC does not require costly or complex resolution processes since the underlying object (CRDTs) already provides methods that aim at resolving these conflicts in a streamlined manner. This concept now requires formal proof of how CRDTs achieve it, two possibilities are available: state-based or operation-based.

2.4.2.1 State-based

A state-based object or an object in the state-based style is defined by a tuple (S,s0,q,u,m), set S is the possible states of the object which individually produce the payload of the object, $s0 \in S$ is the initial state, q is the set of query methods, u is the set of update methods and m is the merge method which resolves differences and merges the payload of two objects into one. A state-based CRDT must be equipped with one fundamental inner structure, a join semi-lattice, therefore the comparison of any two CRDTs of the same type must produce a partial order $\leq$ equipped with a least upper bound. This structure allows for the introduction of a new definition the *Monotonic semi-lattice object*, an object is included in this definition if the set S of its payloads defines a semilattice ordered by $\leq$; the merging method of the object with a similar one calculates the LUB; the state is monotonically non-decreasing between updates. Shapiro et al. (2011b) theorize that any state-based object which is also a monotonic semi-lattice object will provide SEC as long as eventual delivery and termination are assumed, a formal proof can be found in the referenced work. Eventual delivery is defined by *"An update delivered at some correct replica is eventually delivered to all correct replicas"* and termination is the requirement that all methods executing at all replicas eventually terminate.

2.4.2.2 Operation-based

Another possible implementation of a CRDT can be operation-based, informally these can be abstracted to an optimization of a replicated state-machine, every CRDT has its own internal state which can be changed by applying operations which are then communicated to other replicas. Using state-machine replication, all replicas must apply operations in the same order, the way CRDTs optimize this definition is by defining commutative and non-commutative updates. This allows commutative updates to be immediately delivered without requiring any order while non-commutative must respect causal-delivery. Causal delivery, however, is a problem when network partitioning can occur, we must keep a log of all registered operations so that at a future date this log can be used to update the partitioned network and vice-versa. This raises the problem of knowing when to prune the log or otherwise we incur the risk of it growing indefinitely (Wuu and Bernstein, 1984). To effectively prune the log without losing any information from either the local (emitter) or the remote (receiver) CRDTs we would require a confirmation that all nodes in the network acknowledged the update.

An object in the operation-based style is defined by a tuple (S,s0, q, t, u, P). Similarly to the state-based object we have a set of payloads S, an initial state s0 and a set of queries q. As expected the merge methods are no longer needed since we are not merging states, instead, we make use of a prepare (t) and effect (u) sets of methods. The way these work is a method from (t) executes before an update is effective at the local replica, immediately followed by the update method from (u). Method u is then delivered to all other replicas, parameters included. Like queries (q) prepare-update methods (t) do not have any side effects so they do not enter the causal history (C) of an object, an object's causal history is solely composed by the update methods executed at that object so we say an update $u_j(a)$ has been delivered to replica i when $u_j(a) \in Ci$. From this causal history a "happened before" relation is easily extracted in the form of (t,u) happened before (t',u') *iff* when (t',u') executes, (t,u) is already present in C. In case updates happen concurrently it is not possible to define a happened before relation so we must prove that concurrent methods are commutative between them, another required measure that needs to be implemented is that of a delivery precondition (P), in other words, when is an update allowed to be accepted by a replica, but for simplicity sake the authors restrict the scopes of preconditions to those where the precondition is assured by causally-ordered broadcast.

2.5 CAP Theorem

One of the most accepted works in defining limitations to distributed data replication is the CAP theorem. The CAP Theorem, first presented by Brewer (2000), defends that shared-data systems are limited to just two of the following three properties: consistency, availability, tolerance to network partitions. In practice a system can either be exposed to network partitions or not, so what the theorem implies is that if network partitions are present a compromise must be made between consistency and availability. If there are no network partitions a system can be both consistent and available. Consistency refers to the values read from the different machines that compose the system, a system is consistent if the values read from every constituent machine will output the same value at any moment in time. Availability refers to how "available" the system is to accept write and read operations, a system with high availability will accept any write or read operations even if the values read are not the most up to date ones. Tolerance to network partitions refers to the system's ability to continue functioning as expected when components lose communication with each other. In this work network partition is inevitable so we can expect there will be no solution that provides a high level of both consistency and availability. The most important take away from the CAP theorem is understanding that when faced with network partitions to optimize a distributed data system what we are truly optimizing is the balance between availability and consistency.

2.6 LSTS Toolchain

In this Section, I will cover LSTS' toolchain in detail to better understand the pre-system this work will be integrated with. LSTS works with several different systems, autonomous vehicles, communication modules, and software that is used to program and control the vehicles as well as analyze data. For all these systems to work together an application level protocol was developed internally, the Inter Module Communication Protocol or IMC. This protocol defines a message structure of header and body fields for a set of known message types.

The IMC Protocol is provided as a java library which includes all components necessary to make an application comply with the IMC protocol. The library includes all the necessary definitions and interface methods, including binary serialization and de-serialization methods. Above all, it contains the definition of an extensive set of IMC defined messages, for this work we used the Event type as a placeholder since the implementation of a new message type would require an update throughout the toolchain which was not truly necessary for the time being. The structure of the Event message was very flexible and hence its selection for this work, message headers do not contain any specific information so there is no need to cover them, the body of an Event message contains two String fields, labeled topic and data.

This work's focus resides mainly in Neptus since it provides a very stable and flexible platform for development which can, ideally, be adapted with ease to work with other systems in the toolchain. Vehicles are also a very important link in LSTS' data flow as they provide a mobile platform for data to be shared in these scenarios:

- Data can be stored on a vehicle that will travel between vessels where crews are operating using Neptus. When close to a vessel data can be transferred over a wireless network instead of using paid satellite links to communicate between vessels.
- For long-range missions, a smaller, short-range, vehicle will rendezvous with one performing a long-range plan to transfer data back to the main vessel where researchers can immediately start to analyze it.

These, among other less common scenarios, justifies the necessity of vehicles also being included in this work. Hence a solution was also developed around the Dune platform which runs in GLUED, a light-weight Linux distribution tailored for the use-case of LSTS. Dune is the heart of the vehicles, it controls navigation, sensory activities, networking and the interchangeable systems present in each vehicle. It also provides a simulation mode which proved extremely useful for development as no physical vehicle was necessary for deployment.

A very important step for development was categorizing the data types which flow between the different systems. These three data categories were defined:

- Timestamped - This type of data includes a pair (device, timestamp) that provides a permanent unique identifier which allows for simple ordering since the timestamps all originate from the same process and cannot be edited;

- Read-only - This type of data is read-only, as the name indicates, but the same data item can be produced by different systems leaving the system ID as the unique identifier;
- Editable - editable data can be edited by anyone with the proper permissions;

2.6.1 Neptus

Neptus is the flagship software of LSTS, it is a Java-based GUI application that provides users with many different tools. In Figure 2.1, I present a labeled picture of Neptus' interface. The toolbar allows the user to quickly access the most used tools without the need to navigate the menu bar. The Mission data panel concentrates the direct controls of the selected vehicle, immediately above it, and also information like the transponders and plans to be used in the operation. The map display is where the user can visualize the active vehicles' live position and the selected plan course. Other important data like restricted navigation zones or other necessary data layers are also displayed. In Figure 2.1, the Plan Editor tool is active so we have a detailed view of the selected plan and, on the right the Plan Editor properties interface.

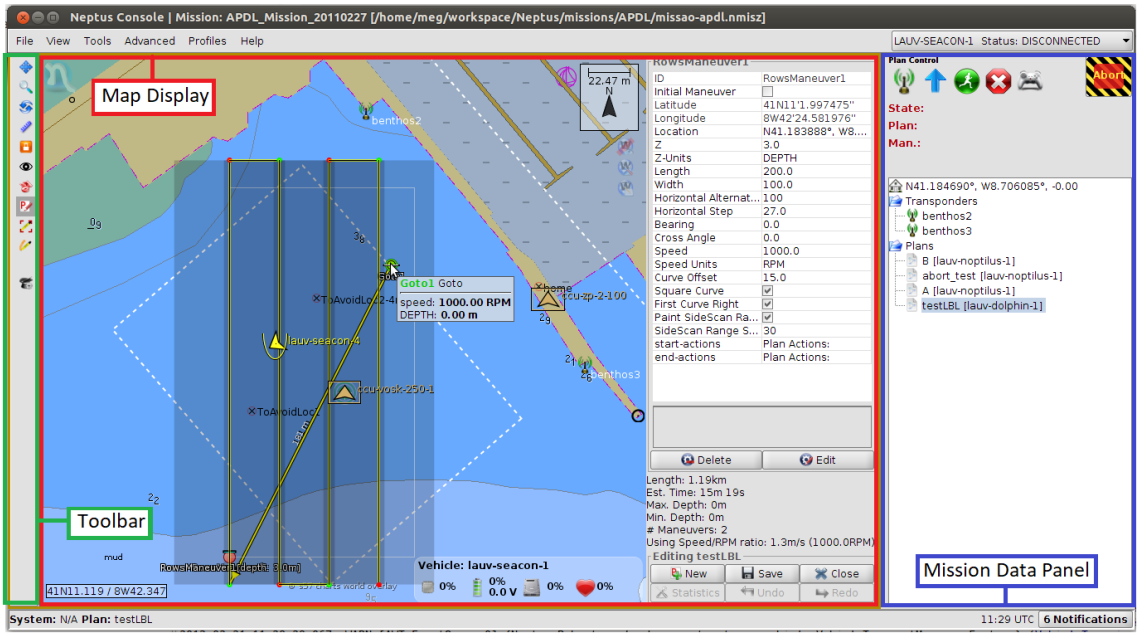


Figure 2.1: Neptus Interface with the Plan Editor tool active

Neptus architecture relies on plugins for expansion, this is useful in the sense that a plugin does not need to be fully integrated with the rest of the code structure but can still access it. A plugin can take different forms, the main ones are a Console Panel or a Map Layer. A Console Panel allows for a fully customizable interface window while the Map Layer is restricted to drawing over the existing map. I will not cover the Map Layer as it was not used and its functionality is mostly similar except for the user interface. All plugins have direct access to Neptus core code, in this work we make use of mainly four big modules:

- IMC Java - This is the IMC protocol library, it provides message objects and all the serialization process related to them. It also provides some other required definitions for IMC;
- IMC Message Manager - The IMC message manager aggregates a lot of information for IMC based communication, including **(i)** active IMC enabled nodes in the local network; **(ii)** the local system's IMC ID; **(iii)** a range of individual properties of connected nodes. The major feature of the message manager is the interface for IMC message sending, it includes a generic send message method that can be parametrized with which transport to use (UDP, TCP) and whether or not to use multicast/broadcast. It also takes responsibility for filling in the header fields of messages;
- Message Subscription - Neptus uses Java Annotations[1] to provide a subscription mechanism to its message bus. Meaning, methods annotated with the Subscription tag will be invoked every time a relevant message is received by the Neptus instance;
- Properties Provider - One of the most useful features in Neptus is the system-wide properties saving and loading mechanism, this allows plugins to save and load configurations with ease, the only requirement on the plugin side is to annotate a field with the Property annotation[1].

Network properties were analyzed using the IMC Communication Manager. The communication interface provided both TCP and UDP transport layers, TCP's properties are very well-documented so I will not go into detail (Postel, 1981). The UDP transport is the most interesting to cover, there are no significant properties outlined in the transport definition, so these are usually added in the application layer when needed. In local networks, Neptus communication interface guarantees, by managing the optional parameters, the following communication properties:

- No message duplication;
- Messages may be lost;
- There are no transmission errors, if node i receives message M from node j the received message is the same as sent by node j;
- Received messages must at some prior point in time have been sent by another node, in essence, no spontaneous messages are generated.

An important Neptus data type to cover, due to its impacting role in this work is the Plan. A Plan in Neptus is a representation of the planned course for a vehicle to execute. This course is defined by waypoints called Maneuvers which are connected by Transitions, an abstraction to a graph is easily achieved. Maneuvers store several data items, concerning the behaviour of the vehicle, which are not relevant for this work. Transitions also store some data but not on the same scale as Maneuvers. Both Maneuvers and Transitions are identified by a string type field which must be unique within the containing Plan.

[1] Java Annotations documentation can be consulted here:
https://docs.oracle.com/javase/7/docs/technotes/guides/language/annotations.html - accessed 4th July 2020

Other features of Neptus include various data visualization tools, map exploration with overlays for different data types, plan simulation with capabilities to account for tides, among several others which are not relevant for this work and for that reason will not be covered.

2.6.2 Vehicles and Dune

Vehicles are the most dynamic system in LSTS' toolchain, these are focused on the autonomous operation in different conditions which are generalized to aerial and submarine environments. LSTS possesses vehicles with different ranges of operation and sensory equipment but all of them share the same control software, Dune. Dune is a C++ program that provides autonomous control and interfaces with the several systems aboard vehicles. An image of two Noptilus class vehicles is presented in Figure 2.2.



Figure 2.2: Two Noptilus category submarine vehicles

Most of the current sub-marine vehicles possess two computing systems. The main one is where Dune runs and has access to external communications and on-board systems/sensors. A secondary processor, dubbed backseat, is connected to the main one via a network switch which allows for two-way communication. This backseat system is usually used to run non-critical applications that may interface with Dune over a TCP connection.

Dune instead of plugins uses the concept of tasks, tasks operate primarily on an event-based behaviour and are the symmetric to plugins in Neptus (Section 2.6.1). Tasks are usually specifically designed to interact with sensors or internal electronic systems. They are also provided a

system-wide configuration system which loads on startup and can be edited as plain text files of key-value entries before startup. For example, the communication with the backseat processor is the responsibility of a task that can be configured with respect to which types of IMC messages are sent through.

Contrary to Neptus (Section 2.6.1), Dune's data types are mostly in the format of IMC messages. Lets take the example of the Plan data type, its representation as an IMC message contains mostly the same data but with a different structure. Maneuvers and Transitions are now their IMC Message counterparts and data fields present in the Neptus Plan are no longer available the same way new data fields are available in the IMC message Plan.

2.6.3 Manta

Manta is not a vehicle but its inner workings are very similar, it also operates using Dune so the same architecture of tasks and configuration applies. A Manta works as a portable communication interface, with support for several communication mediums provided the correct attachment is connected. As such, it is a very interesting system for our work since it processes a lot of data going through the network.

For example, Mantas can be used to connect a vehicle using sub-aquatic acoustic communication to systems using other more regular communication methods, like Ethernet and WiFi or even some less common ones such as satellite or 3G/4G mobile networks. In essence, a Manta serves as a proxy for switching communication technology and allows a more extensive coverage of the communication space. In Figure 2.3, I present a picture of a Manta (black box) equipped with an antenna and an acoustic modem (blue cylindrical shape).



Figure 2.3: Manta with an antenna and acoustic modem installed

Chapter 3

Automated Consistent Data Sharing

This Chapter will present and argue the proposed solution that aims at solving the problems presented in Section 1.2. In Section 3.1 I will discuss how the system architecture was selected. Then in Section 3.2, I will start by covering how consistent data sharing was achieved in a local network. This includes the analytic process which lead to the selection of CRDTs as the approach to follow and the reasoning behind which CRDT should compose the solution to achieve consistent data replication. Following, in Section 3.3, I discuss how the solution implemented at a local level was adapted to also work at a wide-area level, highlighting the limitations of the local-area solution and how these were solved or minimized, taking into account the restrictions of LSTS' operation scenarios. Section 3.4 covers the solution for a leader election problem introduced by the support of wide-area consistency. Finally, Section 3.5, defines the communication protocol and the reasoning behind it.

One of the most important steps I considered during development was defining the underlying network properties of the existing communication layer provided by both Neptus and Dune. Since designing a new protocol was not in the scope of this work I had to keep with these communication layers. With this information in mind, selecting the appropriate architecture, an equally important step in building a distributed system, ensued.

3.1 System Architecture

According to LSTS' specification the system assumes all nodes in the network should be replaceable with or without prior notice, effectively meaning no number of machines can hold enough significance that their disconnection would compromise the correct behaviour of other nodes. Referring back to Section 2.1, this clearly points at a decentralized solution similar to a peer-to-peer network, even though the name usually refers to large scale networks and we are talking about a network size of, at most, around ten machines. A **decentralized** architecture was ideal to cope with the common network partitioning that occurs at any time. To keep availability in systems that become isolated, each machine keeps its own record of all the data. The high amount of partitioning is justified by the particular dynamics of the nodes. The main causes are vehicles being

able to freely move in and out of communication range and the connection inside the vessels being vulnerable to relocation of the operators' machines rendering it unstable.

With this dynamics in mind I opted for a less costly **unstructured** network overlay over a structured one. This implementation was possible partly because Neptus and Dune, in their communication interface, already provide a list of connected nodes and the possibility to broadcast messages. Another somewhat related factor supporting this decision was the fact that we do not require any network resource lookup (one of the biggest features of structured network overlays).

An important aspect to consider in other areas of development was also outlined during this stage. Although it is not possible to distinguish a node based on it is communication, it is communication identifier provides insight into what kind of system we are communicating with. Different systems types (Section 2.6), can be expected to behave in different ways, physically speaking, a vehicle can be expect to leave and re-enter a network more often than a CCU or a Manta.

In sum, the goal of this architecture is to make it possible that any system remains available as long as itself persists. The same logic indirectly applies to network data, as long as there is one active system, data contained on that system can potentially be re-introduced into a broader network.

Network Security was considered as well, LSTS has no security layer built into it is software, although this may seem like a serious oversight and considerations took place to assess if it should be included in this work's scope, we opted to leave it out due to time limitations and the deep, complex integration that would need to be achieved for it to have any significant impact on LSTS' work. Since communication on a public network is seldom and not sensitive, this problem could be over-looked.

3.2 Consistency in a local network

Achieving local network consistency was the main area of development, I analyzed the different degrees of consistency presented in Section 2.4, outlining existing algorithms to achieve them while tolerating network partitions.

To start with, we will needed a communication infrastructure to share data with other nodes, the IMC communication interfaces of Neptus and Dune were used (Section 2.6). Referring to the CAP Theorem (Section 2.5), I based my analysis on the consistency level, starting with the strongest consistency models and weakening them to introduce the desired availability while always keeping partition tolerance. The desired levels were defined in conjunction with LSTS' empirical input and expected system behaviour. Once the acceptable levels of availability, in the presence of partition tolerance, was achieve I considered to have found the strongest consistency level possible for the system.

One of the strongest models defined is that of sequential consistency, we analyzed the more traditional algorithms that fulfil the specification. These, proved very capable in terms of consistency, mostly providing a consistency level stronger than effectively necessary. However, the major drawback was the requirement of tight synchronization among processes, which cannot

be satisfied in the presence of network partitions or when nodes join/leave the network. Some other drawbacks include high latency in read and write operations and high message exchange rates. More specifically, we analyzed the primary-based algorithms and found these offered a very compelling solution by having a single system being the owner of a data item, thus eliminating inconsistencies between replicas. A simple switch in ownership would suffice for leaving the network, but this approach requires ownership management. Ownership management is complex and costly especially if, first, we want to distribute the load over the different nodes to improve scalability, second, we want to support both local and wide area scenarios.

One of the most popular models immediately weaker than sequential consistency is causal consistency which aims to reduce the sequential requirements solely to operations with a causal order (Section 2.4.1.1), unfortunately there are still limitations in terms of availability. These limitations are caused by the high levels of collaboration required to create a causal order of events. The limitations are reflected in reduced partition tolerance and possible high latency of read/write operations.

Sequential and causal consistency algorithms were grouped as those that provide, elevated consistency levels following every operation. This levels of consistency are not fully required in cases where our system may write to the same object with a high frequency (i.e. a sensor's live readings) and real-time data readings are not critical. Most data requires processing during or after mission completion to be analyzed. In sum, the negative aspect of these was the compromise between availability and partition tolerance versus consistency. The compromise mostly tended towards consistency leaving a lot to be desired in terms of availability and partition tolerance. The advantage of providing real-time readings of any data object was not as valuable as availability. Algorithms more focused on providing good availability and mechanisms to deal with partition tolerance were required.

I narrowed the search to algorithms that provided eventual consistency. An eventual consistency model would allow for more inconsistencies and provide a very acceptable level of availability and tolerance to network partitioning. These algorithms and technologies were analyzed: Bayou-System, Conflict-Free Replicated Data Types.

The Bayou System was, by far, one of the solutions that best resembled our system and with a lot of literature reporting and detailing its inner workings (Demers et al., 1994; Terry et al., 1995; Edwards et al., 1997). I predicted it to be a good solution with a few caveats, but nonetheless applicable and better than any of the previously analyzed options. It required a process of *anti-entropy* that would perform an analysis and resolution of conflicts between replicas, unfortunately this process was developed to be pair-wise, not to mention complex and highly domain-specific, which, after all, was the exploitation of the Bayou system. Out of the gate the Bayou system fit the problem very well providing incremental progress that could be interrupted if partitioning occurred, update replication instead of full data base transfers (saving on data transfers) and a node leaving the network need only communicate with another one to correctly process its exit. Data sharing is also quite simple, as with most algorithms based on the eventual consistency model, there are no complex processes to prevent a system from writing or reading an object, the single

requirement is that updates eventually reach all replicas. However, also as most algorithms based on eventual consistency, it struggles when it comes to resolving inconsistencies that are prone to occur.

The other system analyzed in the scope of eventual consistency were CRDTs, these are one of the most modern documented technologies about data replication in distributed systems' literature. CRDTs improve on the Bayou system in the aspect of conflict resolution. A CRDT is mathematically proven to always resolve it is own conflicts through the use of the integrated merge method. So, we keep the eventual consistency advantages of high availability and small to null read/write latency, but have now eliminated domain-specificity, complex conflict-resolution and limitations introduced by *anti-entropy* processes.

It sounds too good, so what are the draw-backs, considering this specification it would be complex and time-consuming to produce a proof of correctness for a new CRDT, however, there is already a comprehensive list of very capable CRDTs available in the literature which were used as a building block for this work. it is also important to remember what we were giving up by using them, CRDTs are not the perfect solution and involve an heavy compromise to consistency in favor of great availability, an inevitable choice when network partition occurs (Section 2.5). The network will never be fully consistent as long as data is still being exchanged.

As described in Section 2.4.2, CRDTs are quite complex and offer different implementation flavors, a deeper analysis was conducted to decide which CRD-Types would be used in this work. While an operation-based approach may seem simpler and possibly even feasible to implement from scratch, it hides quite a lot of underlying requirements. As covered in Section 2.4.2.2, operation-based CRDTs require specific network behaviour that unfortunately our scenario does not allow for. It is not feasible to guarantee delivery to all nodes in the network since new unknown systems may enter the network at any moment, nor is it logical to keep an indefinitely large log of operation for a single data object. At this point an operation based implementation is pretty much out of consideration.

A state-based approach, on the other hand, does not have such underlying requirements. The problem of guaranteeing delivery to all nodes is overcome at the cost of having to share the whole payload of the CRDT versus solely the operation. Considering a system may be absent from the network for undetermined amounts of time, when thinking about objects with a high rate of write operations, the amount of data in the state might even be smaller than the log of missed operations, refering to the occasion of resolving network partitions. The biggest hindrance of using a state-based approach is the requirement of providing a merge method, which is an heavy deterrent for developing our own CRDTs from scratch. An important note is that there are proposed methods to transform state-based to operation-based CRDTs, unfortunately, these do not provide a solution for delivery acknowledgement and the maintenance of a log. In the following subsections I will enumerate and go over all of the implemented state-based CRDTs in detail.

3.2.1 Grow-Only Set (G-Set)

Sets are one of the most basic data structures and are the underlying structure present in others, like Maps and Graphs. *Add* and *Remove* are considered the only mutational operations, however, these do not commute and for this reason the sequential specification of a set becomes complex to implement as CRDT. One of the easiest ways to remove this complexity is to remove one of the operations, in this case, **Remove**.

By making the **Remove** operation unavailable we can build the Grow-Only Set, a set that grows indefinitely and is easily implemented as a state or operation-based CRDT. I used a state-based approach documented in the work by Shapiro et al. (2011a). The merge method is as simple as computing the union of the payload sets of the merging objects. A detailed implementation can be found in Shapiro et al. (2011a), the implementation used in this work is presented in Specification 1.

The G-Set was not used in the integration with Neptus or Dune data objects nor in the building of any other CRDT. Its implementation was simple and was added as a nice to have item, too keep any sort of log for example.

Specification 1 State-based Grow-only Set (G-Set)

1: **payload** set S
2: **initial** $\varnothing$
3: **update** *add* (element e)
4: $S := S \cup e$
5: **query** *lookup* (element e) : boolean b
6: **let** $b = (e \in S)$
7: **merge** (A,B) : payload R
8: **let** $R.S = A.S \cup B.S$

3.2.2 Observed-Remove Set (OR-Set)

Since a G-Set does not support the expected behaviour of a Set data structure, I also implemented a set-like CRDT with support for both *Add* and *Remove* operations. There are several documented implementations of Set emulating CRDTs supporting both operations, for this work, however, most of them presented limitations. Some examples include, "tombstones" of removed elements that grow indefinitely, not being able to add an element once this is removed (2P-Set, Shapiro et al. (2011a)), relying on timestamps for element selection (LWW-Set, Shapiro et al. (2011a)).

For this work I implement an OR-Set, documented in the work by Bieniusa et al. (2012). This implementation of the set provides the regular *Add* and *Remove* operations with the expected behaviour. In this work they present both a simple and an optimized version of a state-based OR-Set. While the simple one relies on an unbound tombstone set, the optimized one provides a more complex merge method and leverages the use of unique tags associated with each add operation. Said unique tags are implemented as a pair of timestamp and a replica ID. Each replica keeps a set of tuples $< element, timestamp, replicaID >$ and a version database, an add operation will insert

a new element in the set of tuples associated with the current timestamp and the local replica ID, followed by an update to the version database, setting the local replica ID entry value to the same timestamp. The merge method will then use both the set and the version database to compute which elements are still in the set, giving priority to add in case of concurrent *add* and *remove* operations. "Old", unecessary tuples are also eliminated in the merging process. A complete implementation description can be found in Bieniusa et al. (2012). The implementation for this work is presented in Specification 2.

The OR-set was not used in the integration with Neptus or Dune data objects but is used in the building of the Dictionary CRDT (Section 3.2.3).

Specification 2 State-based Observed-Remove Set (OR-Set)

const $localID$ ▷ Represents the unique identifier of the replica
payload LinkedHashSet E, HashMap v ▷ E: set of tuples (element e, timestamp c, replica i)
▷ v: last known update timestamp, mapped by replica ID
initial $\varnothing, \{localID \rightarrow 0\}$
query *elements* () : set S
let $S = \{e | \exists c, i : (e,c,i) \in E\}$
query *contains* (element e) : boolean b
let $S = (\exists c, i : (e,c,i) \in E)$
update *add* (element e)
let $r = localID$
let $c = v.get(r) + 1$
let $O = \{(e,c',r) \in E | c' \leq c\}$
$v.add(r \rightarrow c)$
$E := E \cup \{(e,v,i)\} \setminus O$
update *remove* (element e)
let $R = \{(e,c,i) \in E\}$
$E := E \setminus R$
merge (B)
let $M = (E \cap B.E)$
let $M' = \{(e,c,i) \in E \setminus B.E | c > B.v.get(i)\}$
let $M'' = \{(e,c,i) \in B.E \setminus E | c > v.get(i)\}$
let $U = M \cup M' \cup M''$
let $O = \{(e,c,i) \in U | \exists (e,c',i) \in U : c < c'\}$
$E := U \setminus O$
$v := \{k \rightarrow val | k \in v.keys \cup B.v.keys : val = max(v.get(k), B.v.get(k))$

3.2.3 Dictionary

A dictionary or a key-value map, is a highly flexible structure and as such it was important to include it in this work. Although there exist designs of dictionary CRDT's (Shapiro et al., 2011b,c), unfortunately they are operation-based. My implementation, Specification 3, takes my OR-Set implementation and adapts it to work with a dictionary. The intuition is that we

will use the OR-Set with entries of a dictionary, which raises a problem. We are now generating a multi-map, because we allow for different elements with the same key being added in the merge method (the add method only involves an intuitive modification of removing any elements with the same key). The merge method of OR-Set uses the full tuple for comparison, so elements, in this case dictionary entries, with the same key but different values will be considered distinct elements and be part of the resulting set, causing two distinct values for the same key. $(< A \rightarrow 1, timestamp, replicaID > \neq < A \rightarrow 2, timestamp, replicaID >)$

I made the decision of using the entry timestamp as the deciding factor for entries with the same key, in practice, at the end of the merge method the resulting set is filtered to only include the entries with the most recent timestamp for the same key. This implies that for the expected concurent behaviour of the map it relies on a global clock synchronization which we do not guarantee, however, for LSTS' data flow this is not a major problem as most scenarios would provide a unique data source as we have seen in Section 2.6 or concurrent updates are rare.

In terms of convergence it is assured by the use of the OR-Set's merge method with the modification working as a deterministic filter that will not introduce any divergence on different replicas. The Dictionary was used in the integration with Dune data objects but not Neptus nor is it used in the building of any other CRDT.

3.2.4 LSTS Plan

The LSTS Plan CRDT, presented in Specification 4, provides a CRDT object that aims to support the Plan Object of the existing toolchain, a native Plan we will call it. The native Plan can be represented by a graph so, following the design presented in the works by Shapiro et al. (2011a,b), I used my OR-Set implementation as the supporting sets of edges and nodes, both of which work independently. Similarly to a graph this CRDT supports the *Add* and *Remove* operations for both nodes and edges with parameters being a nodes and edges representation from the native Plan.

One particularity that was adapted due to Neptus' Plan structure was that, since node names are unique within a plan, concurrent updates will most likely originate conflicts. Conflicts occur, for example, when two new nodes are added concurrently, the added node's name will be automatically assigned in a deterministic fashion, causing a collision. We make the assumption that nodes with the same name are the same object and will follow the update rules of the OR-Set. The alternative would have been, when faced with a concurrent node update, to keep both nodes and change the name of one, however, from the perspective of the user this would most of the times mean correcting the plan to remove the extra node.

The LSTS Plan was used in the integration with Neptus and Dune data objects but not in the building of any other CRDT.

3.3 Consistency in Wide-area Networks

It is often desirable to synchronize data with systems not accessible through a local network and for that effect a wide-area network must be used. The solution implemented for a local Network

Specification 3 State-based Dictionary

const *localID* ▷ Represents the unique identifier of the replica
payload LinkedHashSet E, HashMap v ▷ E: set of tuples (entry e, timestamp c, replica i)
▷ v: last known update timestamp, mapped by replica ID
initial $\varnothing, \{localID \to 0\}$
query *elements* () : set S
let $S = \{e | \exists c, i : (e,c,i) \in E\}$
query *get* (entry.key k) : entry.val r
let $S = \{e | \exists c, i : (e,c,i) \in E, e.key = k\}$
update *add* (entry e)
if $get(e.key) \neq NULL$ **then** ▷ An entry for $e.key$ already exists, so it is removed
$remove(e.key)$
end if
let $r = localID$
let $c = v.get(r) + 1$
let $O = \{(e,c',r) \in E | c' \leq c\}$
$v.add(r \to c)$
$E := E \cup \{(e,v,i)\} \setminus O$
update *remove* (entry.key k)
let $R = \{(e,c,i) \in E | e.key = k\}$
$E := E \setminus R$
merge (B)
let $M = (E \cap B.E)$
let $M' = \{(e,c,i) \in E \setminus B.E | c > B.v.get(i)\}$
let $M'' = \{(e,c,i) \in B.E \setminus E | c > v.get(i)\}$
let $U = M \cup M' \cup M''$
let $O = \{(e,c,i) \in U | \exists (e,c',i) \in U : c < c'\}$
$E := U \setminus O$
let $R = \{(e,c,i) \in E | \exists (e',c',i') \in E : e.key = e'.key, c < c'\}$ ▷ Calculate key conflicts
$E := E \setminus R$ ▷ Remove conflicting Keys with a lower timestamp
$v := \{k \to val | k \in v.keys \cup B.v.keys : val = max(v.get(k), B.v.get(k))$

was not adequate for wide-area networks for the following reasons:

- System Discovery - the listing of active nodes functionality, provided by Neptus and Dune's communication layer, does not extend to wide-area network nodes;
- In a local network there is nearly no restriction to data flow volumes, this becomes an issue because the wide-area communication aboard vessels is usually paid, so the minimization of transferred data is highly desired.
- For the same reason as the last item, it is desired that a single system in a local-network is responsible for performing wide-area communication to avoid duplicate data being shared.
- Communication - the local network solution relied heavily on the broadcast functionality of the IP protocol, which is not available in wide-area networks;

Specification 4 State-based LSTS Plan

```
payload OR-Set V, OR-Set E                    ▷ E: set for edges, V: set of vertex
    initial ∅, ∅
query edges () : set S
    let S = E.elements()
query vertex () : set S
    let S = V.elements()
update addEdge (edge e)
    if not E.contains(e) then
        E.add(e)
    end if
update removeEdge (edge e)
    if E.contains(e) then
        E.remove(e)
    end if
update addVertex (vertex v)
    if not V.contains(v) then
        V.add(v)
    end if
update removeVertex (vertex v)
    if V.contains(v) then
        V.remove(v)
    end if
merge (B)
    E.merge(B.E)
    V.merge(B.V)
```

To solve system discovery, referring back to Section 2.2, a simple and efficient solution that could easily be managed by the user was adopted. The user may input a list of addresses of well-known machines that provide almost guaranteed availability and are running the same data-sharing system. There is no "discovery" process involved, only verification that a connection can be established, the user edits a list of addresses that he knows will most-likely be available at the time and pretends to synchronize data with. This method entails there is no truly automatic discovery of other services and the user must have sufficient knowledge of internet addressing.

Regarding the exchanged data volumes, we need to keep the amount of updates over wide-area connection to a reasonable low amount. Since automation would be a very delicate process of balancing and testing the best option was to allow the user to manage it. The user is provided a set of configurable properties through which he can decide which CRDTs types to synchronize over local or wide-area networks, also there is now the possibility of controlling the update period when using wide-area connections, a feature not present for local networks.

To address the problem of controlling which systems should synchronize over wide-area connections a decision was made to limit each systems responsibility. By default a system is not enabled to send and request updates over wide-area connections unless expressly indicated by the

user or, the regular scenario, it is elected as the representative, in other words, fulfills the role of speaker of the local network it is inserted in. A node with this added responsibility is required to attempt synchronization to the addresses mentioned in the previous paragraph concerning system discovery. In case of a successful connection can be established, it will send and request data from the remote system to achieve the most consistent result possible. An important point of reasoning for this problem was the possibility of existing and future features also benefiting from the election of a leader.

Finally, the communication problem was solved by switching the interface methods of the communication layer to use TCP unicast. The difference of communicating with a single process at a time instead of all the available processes was solved using a local speaker mentioned in the previous paragraph. The implementation of the local speaker meant communication over wide-area networks only concerns two specific processes so there is no longer a need for a broadcast-like functionality.

3.4 Local Speaker

In this section I will expose how a solution was designed for the selection of a local speaker, a concept introduced in Section 3.3. Firstly, to better define what the process of selecting a local speaker should achieve, I detail the responsibilities of the local speaker and how it should be perceived by other nodes in the network. The process of selecting a local speaker should, in a local network environment, **(i)** distinguish a unique process that has elevated responsibilities; **(ii)** the selected process recognizes itself as the local speaker.

A solution was required that would guarantee the completion of this process. Defining a new algorithm from scratch in the available time frame was very unrealistic so I based the design on existing work. Analysing the literature covered in Section 2.3 I came to the conclusions that algorithms that relied on a tree-like structure were impracticable when we are facing a dynamic and unreliable network. Franceschetti and Bruck (1998) and Aguilera et al. (2001) propose solutions for fully connected unreliable networks, Aguilera's solution being very complete but also more complex while Franceschetti's is less complete but better suited due to its simplicity. Franceschetti's solution, based on the work of Singh (1992), focuses primarily on handling faulty links and proposes four levels of progression to achieve the most complete solution. Franceschetti's proposes a protocol he refers to as Φ, said protocol aims at verifying that all nodes have the same view with respect to a given system of the network, usually the leader, so that a re-election process may be initiated. This mechanism was very helpful in achieving the goal of maintaining a stable local speaker when faced with partial link failure.

3.4.1 Model

Defining a model of the system is crucial to proving the correctness of the algorithm. LSTS' communication infrastructure abstracts a model of a fully connected graph defined by a set of nodes (N) that represent active and correct processes and a set of edges (E) which define the

communication links between these. Active nodes can be uniquely identified by their ID and messages can only be received and sent from/to active nodes. Nodes may fail silently. The network is considered to be asynchronous since there is no global clock and both message transmission and processing times are unbounded. Each node is capable of identifying changes in the available communication links through an already existing failure detector. Finally, communication between nodes is defined by the following assumptions:

- No message duplication;
- No spontaneous message generation;
- No transmission errors;
- No messages are lost, if a link is considered available;
- Point-to-Point communication;
- Messages successfully sent over the same link arrive at the destination in a FIFO fashion.

3.4.2 Algorithm

I use Franceschetti's Φ protocol design as the solution for leader election in this work, the core logic is that the process with the smallest ID in a fully connected network will become the leader. The initial election process in this protocol starts with nodes trying to capture the role of leader by sending out an election message with their identifier to all available edges (node i would send the message $elect(id(i))$). Other active nodes on the network upon the reception of an election message compare their own identifier to that of the election message and respond accordingly. For example, node j after receiving $elect(id(k))$ from node k will compare $id(j) < id(k)$: if the condition is true node j does not answer; otherwise, it sends an $accept_elect(id(k))$ message to node k. A node that receives an $accept_elect$ message from all available edges declares itself the leader to all other nodes. A node can be in three states, *Normal* and *Leader* or *Dead*. When in *Normal* or *Leader* state a node must have at least one available link to another node and the protocol is executed as expected. The other possible state is *Dead*, where a node does not have any communication links available and will not take any action except to periodically check if a new communication link is available. Every node has knowledge of its current state, which is kept in a variable.

The process of handling a failure detection is also implemented in Franceschetti's design of the Φ protocol. The idea behind its design is that for a re-election process to be successful nodes must agree on their view of the possible failing leader, only if all nodes cannot establish a link to the current leader will a re-election process take place. A node that detects a link failure will, in case it does not have a link to a node with a smaller ID, try to elect itself by sending a candidate message with the ID of the lost node to all available link (node i would send $cand(j)$ if it lost the link to node j and all other available links are to any node k so that $id(k) > id(i)$). Another possibility is that the node that detected the failure does not have any other links, in which case it

goes into a *dead* state and takes no further action. Dead nodes are restored to a normal state after a protocol message is received. Nodes receiving candidate messages perform two checks **(i)** is the sender the node with the smallest ID in its available links; **(ii)** a link to the lost candidate is not available. If both conditions are true then a response is sent in the format of an accept message (node i would answer with $accept(j)$ to edge e after receiving $cand(j)$ from edge e and verifying both conditions are true), Otherwise no answer is sent. For a node to be elected as leader the same condition is imposed as for the initial election process, a node that receives accept messages from all available edges declares itself the leader to all other nodes.

Franceschetti's design for an initial election process which is started by a node's startup procedure is defined in Algorithm 5. In Algorithm 6 is defined the process executed when a leader failure is detected and the network must perform a re-election. A formal proof of correctness is presented in the work Franceschetti and Bruck (1998).

Algorithm 5 Initial Election process for any node N

```
1:  let nA = number of available links from N
2:
3:  upon node initialization do:
4:      if nA = 0 then
5:          state ← Dead
6:      else
7:          state ← Normal
8:      end if
9:      send elect(id(N)) to all available links
10: upon elect(i) received from node s do:
11:     if i ≤ id(N) then
12:         send accept_elect(i) to s
13:     end if
14: upon accept_elect(j) received do:
15:     if has received accept_elect(j) message from nA unique nodes then
16:         state ← Leader
17:     end if
```

3.4.3 Limitations

Franceschetti's design presented some limitation when applied to our scenario, especially regarding its network partition tolerance. The Φ protocol considers only the behaviour of singular nodes connecting or disconnecting from a fully connected network component. As such, it was not prepared to handle network partitions and consequent healing. Said limitations imply that when a partitioned network heals, it is highly likely that two leader processes exist in the same fully connected component, a failure of the protocol.

To fix this limitation of the original protocol and adapt its behavior to our situation, we modified it so that when a leader detects a new available link it will try to "defend" its position by executing the initial election algorithm. In addition, to ensure a more robust behavior in the presence

Algorithm 6 Re-election process for any node N

1: **let** $nA =$ number of available links from N
2: **let** $K = \{i : id(i) < id(N)\}$
3: **let** $nK =$ number of available links from N to nodes $\in K$
4:
5: **upon** detection of link loss to node j **do:**
6: **if** $nA = 0$ **then**
7: $state \leftarrow Dead$
8: **else**
9: **if** $nK = 0$ **then**
10: $\forall$ available link i send $cand(j)$
11: **end if**
12: **end if**
13:
14: **upon** $cand(j)$ received from a node s **do:**
15: **let** $nS =$ number of available links to nodes $\{p : id(p) < id(s)\}$
16: **if** (link to node j has failed) $\bigwedge$ $(nS = 0)$ **then**
17: send $accept(j)$ to s
18: **end if**
19: **upon** $accept(j)$ received **do:**
20: **if** has received $accept(j)$ message from nA unique nodes **then**
21: $state \leftarrow Leader$
22: **end if**

of partitions, we replaced the condition required to answer to an election message with a condition similar to that used in Algorithm 6. With these changes, Algorithm 5 becomes Algorithm 7.

Another limitation of Franceschetti's design is the restricted choice of leader, a very simple logic of selecting the node with the smallest ID is employed. A better solution, for the problem in question, would be selecting the process which is more adequate to be the leader in the current fully connected component. This solution would be more appropriate since different nodes have varying degrees of connectivity and choosing the right one would mean fewer reelections and improved leader stability. This limitation was not critical and, as such, was not developed in the scope of this work but is a possible future improvement.

3.5 Data Exchange Protocol

A simple protocol was required for nodes to communicate and inform other about the types of actions they wanted to perform, something in the lines of Create, Read, Update and Delete (CRUD) operations. To protect against delayed messages each CRDT is associated with an ID, said ID is flagged as invalid if no updates should be accepted for the CRDT associated with it.

Messages referring to CRDTs should be of the IMC type *Event* and contain a topic with the prefix "crdt_" followed by one of:

Algorithm 7 Improved Election process for any node N

let $nA =$ number of available links from N

upon node initialization **do:**
    if $nA = 0$ **then**
        $state \leftarrow Dead$
    else
        $state \leftarrow Normal$
    end if
    send $elect(id(N))$ to all available links
upon $elect(i)$ received from node s **do:**
    let $nS =$ number of available links to nodes $\{p : id(p) < id(s)\}$
    if $nS = 0$ **then**
        send $accept_elect(i)$ to s
    end if
upon $accept_elect(j)$ received **do:**
    if has received $accept_elect(j)$ message from nA unique nodes **then**
        $state \leftarrow Leader$
    end if
upon new available link **do:**
    if $state$ is not $Normal$ **then**
        $state \leftarrow Normal$
        send $elect(id(N))$ to all available links
    end if

- data - informs the receiver of an update to the provided ID. The message body contains all the necessary information for the creation or update of a new CRDT at the receiver. In case the receiver does not know the ID, he will proceed to create a new CRDT entry for that respective ID. In case the result is an update, the receiver will proceed to execute the merge method with the existing local CRDT associated with the same ID.

- removed - this is used to explicitly inform that a given ID will not be considered valid for any future updates in the current instance of the network. This will prevent late messages from instantiating or updating CRDTs when these have already been deleted. Receiving a removed message means the local instance of the CRDT associated with that ID is deleted. Removed IDs are restored back to the available pool of IDs with a full network restart, but no conflicts are expected since we will be using a UUID object to generate the IDs;

- query - the query serves as a request for data, nodes recovering from a partition or just starting up should direct their request at the local speaker (Section 3.4), instead of just relying on future updates. The contents of the message may contain a list of specific IDs, data types such as "plan" or "vehicle state", or instead if the keyword "all" is part of the list, a complete set of the receiver's IDs and corresponding CRDTs is returned to the sender in the format of individual "crdt_data" messages.r

Another important piece of communication is to remotely set the configuration of the speaker. This feature was important to allow the dynamic configuration of systems without interface, like Vehicles or Mantas, which only rely on a static configuration file, loaded at startup. The IMC Event message with the topic $data_configuration$ is used and the structure of the body is defined as a map of property name and values. To reduce transferred data Boolean values will be added to the map in case of a true value and missing otherwise.

Chapter 4

Implementation

In this Chapter I will cover the implementation details of the methods discussed in Chapter 3. Looking at the methods proposed in Chapter 3 a clear separation of responsibilities can be outlined. I propose an architecture of four weakly connected modules, respectively, a Core, a Communication Manager, an Election Manager and a Consistency Manager (Figure 4.1). **(i)** The core is responsible for interfacing with the pre-existing system, handling data update listeners and notifications. **(ii)** The communication manager does not have any dependency on other modules and represents the existing communication layer interfaces available in the pre-existing system. In case the pre-existing system does not include a communication layer interface one must be made from scratch. **(iii)** The Consistency Manager interfaces with both the core, to receive and notify data changes and the communication manager, to send/receive messages. Its responsibility is to manage CRDT data flow both in and out of the system to achieve consistency, it maintains a local, volatile database of CRDTs which is automated but ultimately controlled by the user. **(iv)** The election manager is responsible for guaranteeing that at any point in time there is an available leader in the current local network. It interfaces with the core to notify state changes and information about the leader. It also relies on the communication module to communicate using application level-protocols.

In the following sections I will expose more details about how the integration with Neptus (Section 4.1) and Dune (Section 4.2) was implemented.

4.1 Integration with Neptus

To achieve a practical integration with the Neptus system we were required to keep to its constraints and other specifications outlined in Section 2.6.1. The preferred solution would be in the format of a plugin with a user interface to allow the user insight into the system's current state. This approach did not present any limitation and as such it was the implementation path followed.

When it comes to user interface, as a plugin, we have access to a set of preferences in the format of key value and a user interface window we can customize to our needs. The inner workings of Neptus are responsible for making these available to the user, in case of the preferences interface

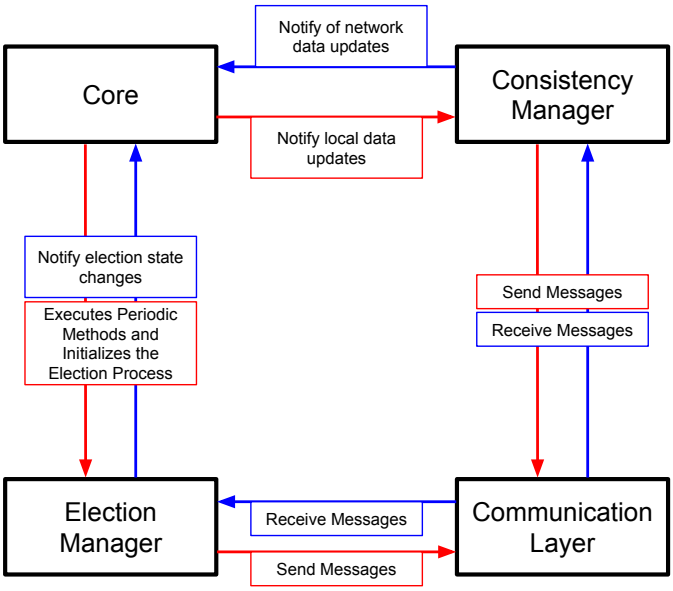


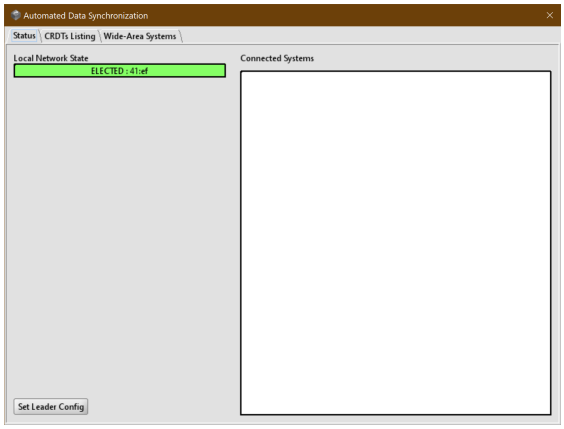
Figure 4.1: Representation of the 4 modules constituent of this work and their interaction

(Figure 4.2d), this is made available from a provided tree like structure of key-value entries, both the loading and saving processes are the responsibility of Neptus. It allows for management of the data types that are automatically synchronized and provides a comprehensive set of options to allow the user to fully configure the data share process. These include wide-area update period and which data types should be synchronized in local, wide-area or no synchronization.

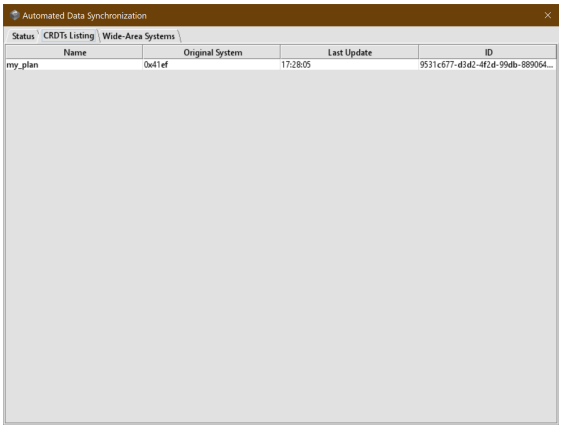
As for the interface window we built it to our specification based on a basic Swing[2] JPanel object which is then used by Neptus. This interface includes three tabs, the first one (Figure 4.2a) provides an overview of the system showing user a list of currently connected systems at a local level, it also provides insight into the election process through a "current state" color coded label and who the current local speaker is. As the leader is responsible for wide area communication, this screen also allows for the local configuration to be sent to the leader, in short, allowing for remote configuration of another connected node. The second screen (Figure 4.2b) displays a sortable table listing the CRDTs currently stored in the system, colums are name, original system, last update time, and ID. Options to save as a distinct local data object not associated with the CRDT and delete with a right mouse button interaction. The third screen (Figure 4.2c) includes wide area synchronization information which is a list of well-known addresses for remote systems. To add a system the user can use the available button and the remove option is included in the right mouse button interaction.

In Figure 4.3, I present the actual modules implemented in Neptus and their interaction, the blocks outlined in green represent modules already available in the pre-existing system that suffered no modification.

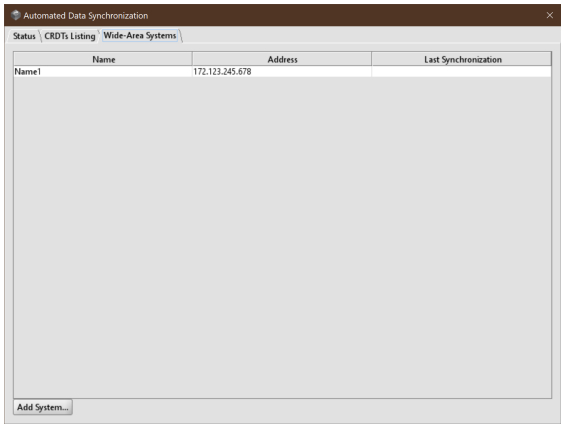
[2]Java swing is a tool used to build window-based applications in Java. It is part of the Java Foundation Classes, documentation can be consulted here: https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html - accessed 2nd July 2020



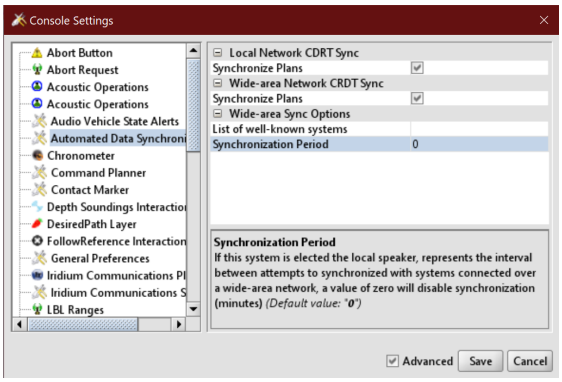
(a) Interface Tab 1 - Status



(b) Interface Tab 2 - List of CRDTs



(c) Interface Tab 3 - Wide-Area Systems



(d) Properties Panel

Figure 4.2: Developed Neptus Interfaces

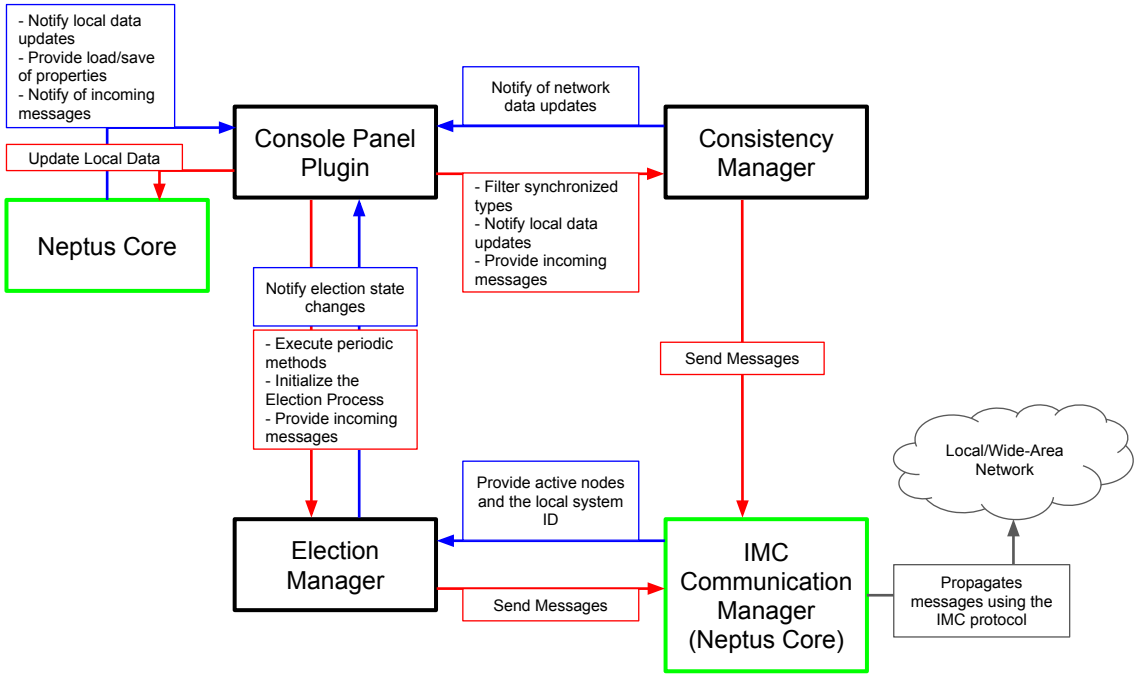


Figure 4.3: Representation of the 4 modules implementation in Neptus and their interaction

As a plugin we can also make use of all available interfaces and libraries loaded in Neptus, in this work I utilize the extensive IMC Communication Manager to process all outgoing communications, this interface provides both UDP and TCP transports, as well as, reception listeners in case the TCP transport is used. For message reception I used the subscription-based bus of incoming messages that allows for a message type specific subscription and is integrated in the plugin base class that interfaces with Neptus. This subscription interface allowed us to define listeners for specific IMC message types without requiring any sorting.

One of the most challenging bits in this integration was, by far, the ability to fully synchronize the various useful data types, responsibility of the Console Panel Plugin. This was due to time-limitations and because of the way Neptus provides, or is unprepared to provide, update listeners and notifications to/from a plugin. So, in conjunction with LSTS, we made the decision to focus on integrating the developed system with a single core data type as a proof of concept. The selected type was the Plan Object covered in Section 2.6.1 because the update listeners and notification structure was already in place. To receive updates on the our proof of concept data type, Plans, I needed to get notification once any plan had been updated. This was achieved by registering a listener on the global Mission object which concentrates all information about the current state of mission planning, like available plans, vehicles and waypoints. This listener receives updates mainly on selected vehicle changed, and plan database changes (creation, update, deletion of plans).

To achieve integration of this external Object with the Consistency Manager it had to capable of making existing Neptus structures, CRDT compatible, we analyzed two different approaches,

changing existing structures or creating new ones. One possibility was to start from existing Neptus objects and edit them so they would become CRDTs, meaning we could just use these edited objects in our CRDT sharing logic. As expected this solution presented a lot of challenges and attrition by LSTS, since it would mean a major change to some of the main data structures in most workflows. Editing existing data structures was considered a last-resource case. If no changes were to be made to the existing data structures, we have again two possibilities, a CRDT could be built based on existing data structures by following the strict proofing process of strong eventual consistency. However, this would cause any future addition to the available CRDTs to also go through the same proofing process. The second option is to make use of already existing CRDTs to build the ones we needed and make the translation between the CRDT object and the existing Neptus object. The latter presented a much more flexible and streamlined solution for the problem of integrating CRDTs with Neptus, for this reason it was the implemented solution. This decision in conjunction with the decision of keeping a database of existing CRDTs parallel to the existing objects allowed for an added layer of user control and flexibility. So, in short, the process was as simple as including four extra methods for the Plan CRDT object: **(i)** a method that receives a Neptus Plan and creates/updates its payload from it; **(ii)** a method that returns a Neptus Plan from the existing payload; **(iii)** a method that returns an IMC protocol message so it can be easily shared; **(iv)** a method that receives an IMC protocol message and can creates/updates the payload.

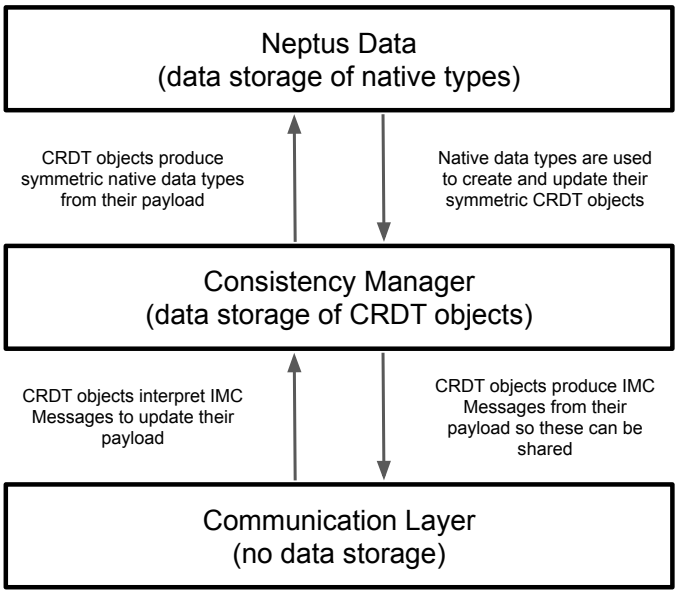


Figure 4.4: Data layers in Neptus and data flow between them

The complete solution to solve full data integration in Neptus is composed of three layers presented in Figure 4.4. Neptus data (as configured by the user), CRDT data, network data. All layers can be updated independently or together, although, currently, no use for updated CRDT data not being reflected in the network data, and vice-versa, was found so this restriction is not

implemented. The flow from Neptus data to network goes as follows: Neptus to CRDT is controlled by the user using the configurable properties; CRDT to Network data has no restriction. The reverse flow, from network data to Neptus data, is as follows: no restrictions from Network to CRDT data; CRDT data to Neptus data is restricted by the user who can select whether this update should be automated or manual.

This solution for data integration entails that a double copy of replicated types is kept at any moment in time meaning extra system memory is required. Another draw back is the complexity introduced in keeping sure data is being shared and updated. Since the data we are visualizing may not be not fully connected to the networked data its required to carefully configure the available properties to our specification and guaranteeing they are as expected before operation.

4.2 Integration with Dune

The most impactful difference for integration between Neptus and Dune systems is the different programming languages they are written in. While Neptus provides an extremely flexible solution that runs on any JVM, Dune is programmed in C++ and optimized to run on the tailor built systems LSTS uses for automation and communication. The decision was made that instead of using the task based feature development of Dune (Section 2.6.2) I could make use of its excelent communication abilities and develop a systems that would be external to Dune by relying on the existing IMC protocol messages for communication. As explained in Section 2.6.2, Dune can forward messages for the secondary processing core, the backseat, so that other programs can listen to them. In the case of a Manta, covered in Section 2.6.3, messages forwarded to its only processing core. Our solution makes use of this backseat communication and, as such, works with any system capable of running a JVM and a reliable communication channel to an instance of Dune is provided. This helped in the development process by allowing for the re-use of most of the code and logic developed for Neptus, and apply it as a standalone Java application using the IMC Java library.

Most of the logic mentioned in Section 4.1, but we are still missing some features which were provided by Neptus but not Dune. The missing features are namely, configurations load/save and user interface. The configuration of the system, including which data should or should not be synchronized is loaded from configuration file to set the initial values on startup and these keep the same value as long as no action is taken. The user interface was deemed unnecessary in vehicles, leaving their configuration static. The configuration becomes dynamic in the case of the vehicle becoming a speaker, in that case, the remote configuration set covered in Section 3.5 is available.

The four module architecture is also being used with Dune, Figure 4.5 represents the actual implementation of these. Again boxes outlined in green present the already existing modules. No communication layer is independently provided in this diagram, this is because Dune is responsible for managing message sharing and delivery. The core also suffered some changed to its original responsibilities. It now needs to provide complete, headers included, IMC messages to send to Dune, as well, as receive them. I would put it as a sort of communications proxy that

fills in the message header and, transparently to other modules, provides a TCP connection to Dune. Since communications with Dune are performed only using IMC messages, the solution implemented with Neptus for data listeners and notifiers had to be modified.

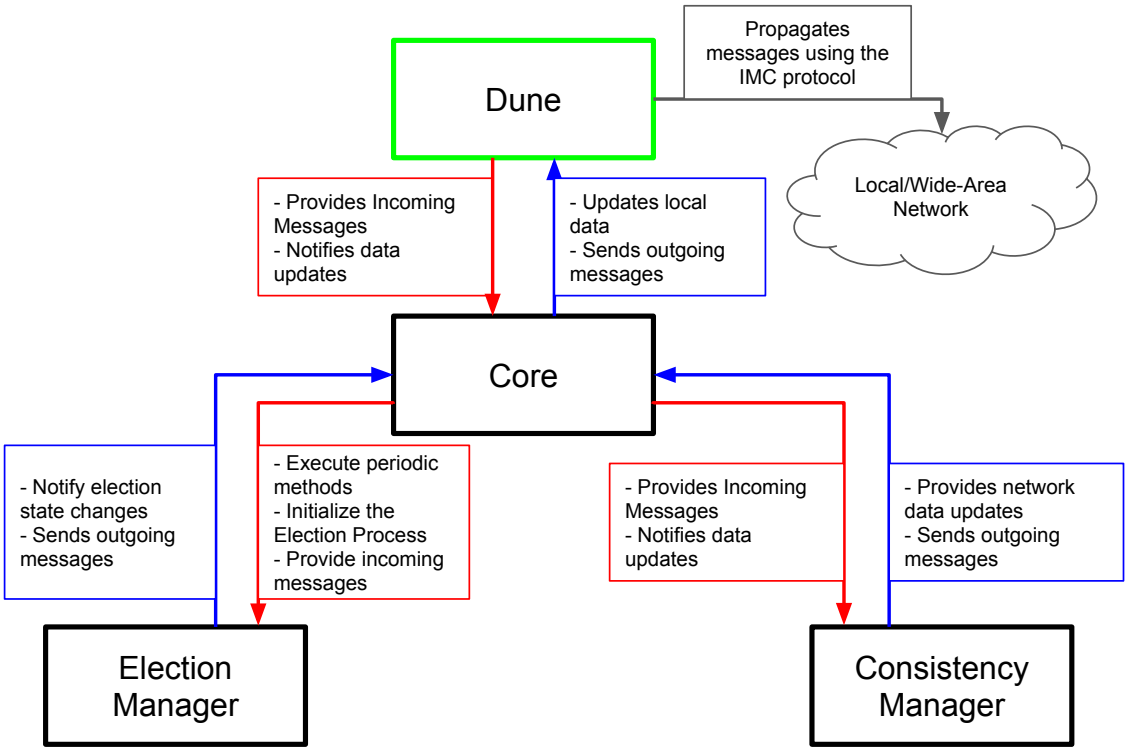


Figure 4.5: Representation of the four modules implementation in Dune and their interaction

Given Dune uses another language, the native objects will not be in the same format as Neptus so a different solution for data translation had to be implemented. In this case, the changes made to the Neptus implementation of the CRDTs focused on the return values of payload methods which are used to obtain a native object type from the CRDT and update it from a native object. There was no room for decision on this matter, since communication with Dune is performed only using IMC messages. The new payload data obtained from the CRDTs is now always an IMC message which is not necessarily the same being used for the sharing of the CRDT with other machines. This method raises a slight limitation about how flexible a CRDT can be, not covered in Section 4.1. To generate the appropriate native object some extra information may be needed about exactly what types of data we are keeping. A practical implementation example is the sets used for the Plan CRDT, which were implemented from the same base CRDT class, the OR-Set, but an extra parameter was added to identify which data types were being handled. Once we have a compatible data type for Dune we make use of the backseat communication to send the message directly to Dune making sure the appropriate header fields' values are filled to prevent the message from leaving the vehicle unintentionally. Dune tasks are then responsible for interpreting these messages and updating Dune's internal data state. A diagram of this interaction is presented

in Figure 4.6.

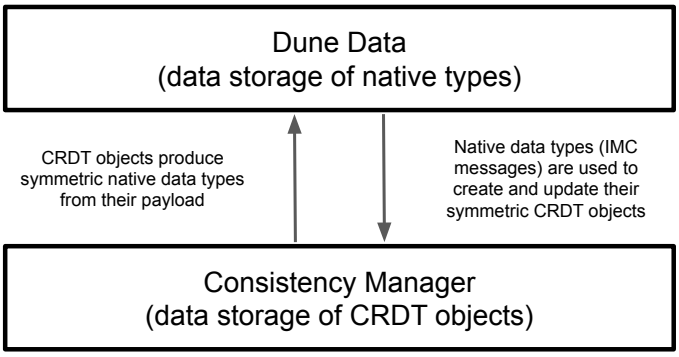


Figure 4.6: Data layers in Dune and data flow between them

We also had to address the opposite direction of data flow, how to get notified of data being updated in Dune. Fortunately, Dune already includes a lot of notifications, in the form of internal messaging, about changes to local data. Messages are transmitted over an internal software bus to which all tasks can directly tap in. Some of these tasks are responsible for filtering which of these messages are suppose to leave the vehicle or, in our case, be transmitted to the backseat. Which messages types the backseat receives can be configured prior to startup by adding the desired message name to in a static key-value configuration file. My system still has to filter out some extra messages, being sent to other applications also running in the backseat, that we are not using.

Chapter 5

Results

In this chapter I will cover the validation of the implemented work and testing methodology. At a superficial level, to assess the correctness of this work I started by verifying modules individually. Followed by a deeper test of completing two scenarios developed by LSTS to evaluate the system. Lastly I will discuss the impact of this work by comparing the process of completing both scenarios with and without the use of this work. A note about the software architecture is also presented at the end.

All testing was conducted in a controlled environment using three regular desktop computers connected over the same Wi-Fi network. Due to the COVID-19 pandemic, no physical testing was allowed with the vehicles, but these are expected as soon as the situation permits. To produce this report I used the available Dune simulator to accurately represent the vehicles used in the test scenarios. The instances of Neptus and Dune were distributed over the available machines using a method of round-robin. Process were assigned an arbitrary machine minimizing the number of processes on each machine and maximizing the number of machines used.

5.1 Individual Module Testing

As individual modules I considered the Consistency and Election Managers. Each CRDT type was subjected to an artificial sequence of operation that were meant to emulate a real use case of concurrent modifications and verify the predicted SEC guarantees, no lack of consistency was detected after all methods had terminated. As expected there was no guaranteed consistency until the termination point as some inconsistencies were noted.

Another component that was tested in isolation was the election algorithm, I devised the following procedures based on real world scenarios experienced during LSTS' operations.

(i) The first scenario verified concurrent start up and network join after startup, I concurrently started two processes, after a leader had been established, another two process were started individually, joining the network and recognizing the existing leader;

(ii) The second scenario targeted leader loss and the re-election process, with four processes in the network the leader process was unexpectedly shutdown and a new leader was elected among the three remaining processes, the shutdown process recovered and rejoined the network being re-elected leader;

(iii) The third scenario covered network partitioning and recovery. Again with four processes in the network I simulate a partition by isolating two processes from the other. In the current leader partition no action is taken while in the other partition a new leader is elected. When partitions came together again to form a single network the first leader was re-elected as continuing leader for the new network;

(iv) The fourth scenario tested the correct procedure for when a process becomes isolated, processes were shutdown individually and the last remaining one entered a dead state.

The testing procedure for the election algorithm included several runs of the above items to simulate potential communication layer failures. All test produced the expected results, no events worth noting occurred during any of the runs. The timing in leader election was not accurately measured as there was not a significant sample of different network configurations and results were mostly constant or random. The election of a leader at start up was dependent on the processes load time until my system's code was executed. In the case a leader was already present in the network a new system would immediately attempt to become leader either successfully or not. The detection of network partitions is as fast as the first process to detect a loss of the link to any process, a verification completed during the periodic active nodes update.

5.2 Testing in real scenarios

These individual modules were then tested together with the rest of the work in two scenarios developed by LSTS to evaluate the system. The **data-muling scenario** targets a long-range mission where vehicles operate for extended periods of time. Support vehicles perform the data-muling task of carrying data between the base station, where a team is operating, and the running vehicles. The context is one co-located team, using an instance of Neptus to operate two Autonomous Underwater Vehicles (AUV) and one supporting Autonomous Surface Vehicle (ASV). For this scenario, the sequence of events is as follows:

(i) An ASV is operating within communication range of the main vessel with the goal of supporting the two AUVs. No communication to the AUVs can be established at this time;

(ii) The operation team aboard a vessel commands the ASV to sail to a, previously agreed, rendezvous point. Before communication to the ASV is lost, the team publishes two plans, one for each AUV. At the time only the ASV can see this publication and stores the plans in its database;

(iii) After reaching the rendezvous point the ASV holds position waiting for the AUVs;

(iv) AUV1, as it will be identified henceforth, surfaces near the stationary ASV and a network connection is established; The ASV shares the plan destined for AUV1 and, simultaneously, receives the state (internal systems state, location, plan execution progress) of AUV1. Before proceeding with the, just received, plan, AUV1 updates it with the new calculated time and place for rendezvous, sharing it with the ASV;

(v) AUV2 eventually surfaces in proximity to the ASV and establishes a connection. In a similar fashion to item **(iv)**, plans and states are exchanged. The particularity of this case is that AUV2 also receives the state and plan of AUV1 and plans its course to match the rendezvous set by AUV1, also sharing the updated plan with the ASV;

(vi) The ASV returns to the vicinity of the vessel where the team is located and information is synchronized with Neptus.

So, in this first scenario, we are presented with a common rendezvous for information exchange, this allows for savings in paid satellite communications and, as such, represents and important exercise in an operation scenario. The system behaved as expected, successfully sharing and receiving data after establishing a connection between any two machines. Time to reach consistency between any two nodes was not a concern but results were very positive with information being exchanged in a matter of, at most, a few seconds depending on the amount to be shared. I will now elaborate on which parts of the process were impacted by our systems and which were responsibility of the existing system. Vehicle commanding, autonomous navigation and decision making is of full responsibility of Dune and Neptus.

With Neptus and Dune the data sharing system's behaviour is event-driven so when, in this case a new connection is established or a data update is received, the system immediately acts to share or request data from other nodes. As such, all of the data exchange processes were managed by the developed work using the available communication infrastructure. In the case of Neptus, the integrated data types are limited to plans so no information about state of vehicles is available to the user even though its in the CRDT database.

Moving on to the **Handover Scenario**, this represents another common occurrence in long-range missions where teams take turns managing the operating vehicles. The context for this scenario is a follow-up to the previous one, the team had been operating for a long time and is preparing to handover operational control to another team in a different vessel. The sequence of events is as follows:

(i) As the scheduled time for handover is nearing, Team A, the longer operating team, contacts Team B and debriefs over what happened in their operational time;

(ii) Team B which does not have physical access to Team A's devices needs to receive data like vehicles states and executing plans only Team A has access to;

(iii) A Wi-Fi connection is established between the vessels and data is transferred.

The second scenario presents a simpler exercise than the first one, but nonetheless represents an important operation especially when several teams are in operation. As specified for the **Data-Muling Scenario** the developed work will automatically synchronize data and share updates with any connected system, so, as long as the configuration properties are set correctly, the test didn't present any issue difference to the expected results. Again it's important to mention that when operating with Neptus only plans are fully integrated and have value for the user.

5.3 Impact on LSTS' Workflow

For this work its not only important to validate the expected functionality but understand its impact on LSTS' workflow and how it compares to the existing system. I will start with the first scenario, for the current system the data-muling exercise is one that requires, as described to me by LSTS, a custom implementation for each operational situation. As can be expected a laborious task as this one mostly caused avoidance in using it. This work will allow the data-muling task to be seamless and only require the vehicles to be in close proximity. If the synchronization properties are leveraged correctly the precise types of information needed can be transferred as wished.

The handover scenario poses a more simplistic exercise that is not fully achievable with LSTS' current system. Synchronization between two Neptus instances is quite limited. Concerning the exercise itself, only plans can be shared and they have to be shared individually by user interaction. Our system, on the other hand, provides the user with the power to select which data types to synchronize, removing limitations to an acceptable level and the whole process is automatic.

Another important point to discuss is how Neptus and Dune behaved when faced with eventual consistency, did the CRDT approach cause any unexpected side-effects? Comparing to the existing system we observe a difference in the behaviour of plans, which are a complex data type and required a, hard to tune, balance of automatic updates versus full manual version control. A demonstration can be made using the Plan data type. It is possible for a user to be editing an older plan and concurrently receive an update for it. If the user had chosen to synchronize plans automatically a merge process will ensue, most likely leading to a loss of unsaved data (this point is up to the user to decide, whether to loose unsaved changes or overwrite the just updated plan). This would not occur in the pre-existing system since operations were manual and, when exchanging plans, usually all users were fully aware of which plans they were handling. As for the data types that represent a vehicle's state and were only integrated with Dune, the visible impact of these was very positive.

In LSTS' toolchain, up to date data is key to allow the user to have a near real-time perception of what is happening. The prospect of systems autonomously updating to the latest possible data they came in contact with, thanks to the integration with this work, proved very beneficial.

5.4 Resulting Software

The designed software architecture was also a positive takeaway. The adopted architecture turned a domain specific project into something more flexible and framework-like. The developed work can be fully separated into four modules, a core, an election manager, a consistency manager and a communication infrastructure. The election manager is highly independent providing just a tool for the data sharing logic. If the communication module is available in the integrating system, depending on the type of interface provided, it may just be weakly connected. Otherwise, a new communication module can also be designed from the ground up, for this work we took advantage of the existing one, which provided a great interface and adhered to LSTS' IMC protocol. The consistency manager and core modules are the only application specific ones as they must, respectively, handle application specific data types and bridge the gap between data updates to and from the integrating system. I support this claim by the ease experienced in porting the Neptus solution into the Dune environment, a radically different program. This process only required the change of data updates' handlers and notification, keeping in mind both systems used a common application level communication protocol.

Chapter 6

Conclusions and Future Work

In this section I will expose the conclusions that were taken from the developed work and its analysis and comparison with the pre-existing systems in LSTS toolchain. Moreover, propositions for future work are presented in section

6.1 Conclusions

LSTS' toolchain presented some critical limitations concerning automated data sharing that were successfully overcome by the integration of this work with the Neptus and Dune systems. Integration with Neptus and Dune, which ultimately was the goal of this project, was verified in two scenarios developed by LSTS to evaluate the system performance in situations that were previously impractical or unavailable. In both scenarios the framework was successful in guaranteeing consistent data share between any two nodes that established a network connection. Concerning LSTS' workflow, this framework helped achieve more consistent result by removing the human factor (all data sharing is automated), eliminated laborious task that provided custom, one time use, solutions and allowed for an extended range of shareable data-types.

The, proof of concept, prototype achieved consistent data replication in an emulated highly dynamic and partition prone network, one of the main problems to solve. Another aspect that was successfully covered was the limited connectivity times. The conducted tests show that data synchronization between replicas is completed within a short period of time after a network connection is established. Thus, suggesting that even when vehicles surface momentarily, this solution still works.

The implemented solution, at least under controlled test conditions, solved all problems presented in Section 1.2, respectively, manual data share between CCUs; limitations to the data types that could be shared between CCUs; data losses in unexpected shutdown scenarios; and laborious one time use solutions for data sharing between. Another critical aspect, that the solution addressed was data sharing in wide-area networks.

Existing systems that guaranteed eventual consistency, thus providing the required high availability and network partition tolerance, were very domain-specific and for that reason presented a

less than ideal solution to our problem. I focused on keeping my solution flexible enough so that it could be easily adapted to other applications or domains. This flexibility proved useful even when integrating the developed framework with LSTS' different systems Neptus and Dune.

Worth noting is that the work was limited at some stages by the COVID-19 pandemic, especially in testing. However, thanks to the dedication of Paulo Dias, my LSTS advisor, we were able to design the test scenarios described above. Thus minimizing the impact of the pandemic on the produced work. The results obtained gave us confidence that the implemented solution will also work in real-world tests, once the public health situation allows it.

6.2 Future Work

Although this is a fully functional framework there is still some interesting work to be produced. An important aspect for a work like this would be comparing it to other solutions. CRDTs, being a relatively recent solution for data replication, still lack significant literature comparing them to other popular data replication systems.

Another line of future work is improving on the current election algorithm or even selecting a better solution. The implemented election algorithm presented some limitation and possible improvements. Due to time limitations neither reduction of limitation nor improvements could be conducted but, if achieved, would considerably improve the system's performance.

Another interesting, but more complex endeavour, would be adapt LSTS systems to use CRDTs as native types. A research path in developing new CRDTs to cover the full data types spectrum available could be followed. LSTS' system being so reliant on data share could take full advantage of this by eliminating the sort of middle-ware implemented for this work.

References

Aguilera, M. K., Delporte-Gallet, C., Fauconnier, H., and Toueg, S. (2001). Stable leader election. In International Symposium on Distributed Computing, pages 108–122. Springer.

Awerbuch, B. (1987). Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In Proceedings of the nineteenth annual ACM symposium on Theory of computing, pages 230–240.

Bass, L., Clements, P., and Kazman, R. (2003). Software architecture in practice. Addison-Wesley Professional.

Bieniusa, A., Zawirski, M., Preguiça, N., Shapiro, M., Baquero, C., Balegas, V., and Duarte, S. (2012). An optimized conflict-free replicated set. arXiv preprint arXiv:1210.3368.

Brewer, E. A. (2000). Towards robust distributed systems. In Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing, page 7. ACM.

Demers, A., Petersen, K., Spreitzer, M., Terry, D., Theimer, M., and Welch, B. (1994). The bayou architecture: Support for data sharing among mobile users. In 1994 First Workshop on Mobile Computing Systems and Applications, pages 2–7. IEEE.

Edwards, W. K., Mynatt, E. D., Petersen, K., Spreitzer, M. J., Terry, D. B., and Theimer, M. M. (1997). Designing and implementing asynchronous collaborative applications with bayou. In Proceedings of the 10th annual ACM symposium on User interface software and technology, pages 119–128.

Franceschetti, M. and Bruck, J. (1998). A leader election protocol for fault recovery in asynchronous fully-connected networks. Technical Report ETR024, Paradise,California Institute of Technology.

Garcia-Molina, H. (1982). Elections in a distributed computing system. IEEE transactions on Computers, 31(1):48–59.

Hutto, P. W. and Ahamad, M. (1990). Slow memory: Weakening consistency to enhance concurrency in distributed shared memories. In Proceedings., 10th International Conference on Distributed Computing Systems, pages 302–309. IEEE.

Lamport, L. (1978). Time, clocks, and the ordering of events in a distributed system. Commun. ACM, 21(7):558–565.

Lamport, L. (1979). How to make a multiprocessor computer that correctly executes multiprocess programm. IEEE transactions on computers, 28(9):690–691.

Lamport, L. (1998). The part-time parliament. ACM Transactions on Computer Systems, 16(2):133–169.

Postel, J. (1981). Transmission Control Protocol. RFC 793, RFC Editor.

Preguiça, N., Marques, J. M., Shapiro, M., and Letia, M. (2009). A commutative replicated data type for cooperative editing. In 2009 29th IEEE International Conference on Distributed Computing Systems, pages 395–403. IEEE.

Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011a). A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA.

Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. (2011b). Conflict-free replicated data types. In Symposium on Self-Stabilizing Systems, pages 386–400. Springer.

Shapiro, M., Preguiça, N. M., Baquero, C., and Zawirski, M. (2011c). Convergent and commutative replicated data types. Bull. EATCS, 104:67–88.

Singh, G. (1992). Leader election in complete networks. In Proceedings of the eleventh annual ACM symposium on Principles of distributed computing, pages 179–190.

Tanenbaum, A. S. and Van Steen, M. (2007). Distributed systems: principles and paradigms. Prentice-Hall.

Terry, D. B., Demers, A. J., Petersen, K., Spreitzer, M. J., Theimer, M. M., and Welch, B. B. (1994). Session guarantees for weakly consistent replicated data. In Proceedings of 3rd International Conference on Parallel and Distributed Information Systems, pages 140–149. IEEE.

Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. (1995). Managing update conflicts in bayou, a weakly connected replicated storage system. ACM SIGOPS Operating Systems Review, 29(5):172–182.

Vasudevan, S., Kurose, J., and Towsley, D. (2004). Design and analysis of a leader election algorithm for mobile ad hoc networks. In Proceedings of the 12th IEEE International Conference on Network Protocols, 2004. ICNP 2004., pages 350–360. IEEE.

Wuu, G. T. and Bernstein, A. J. (1984). Efficient solutions to the replicated log and dictionary problems. In Proceedings of the third annual ACM symposium on Principles of distributed computing, pages 233–242.