



Deusto

Facultad de Ingeniería
Universidad de Deusto

Ingeniaritza Fakultatea
Deustuko Unibertsitatea

Grado en Ingeniería Informática **Informatikako Ingeniaritzako Gradua**

Proyecto fin de grado **Gradu amaierako proiektua**

**Design and Implementation of a Distributed
Network of Autonomous and Heterogeneous
Services**

Mikel Solabarrieta Román

Director: Diego Casado Mansilla

Bilbao, junio de 2020



Deusto

Facultad de Ingeniería
Universidad de Deusto

Ingeniaritza Fakultatea
Deustuko Unibertsitatea

Grado en Ingeniería Informática Informatikako Ingeniaritzako Gradua

Proyecto fin de grado Gradu amaierako proiektua

Design and Implementation of a Distributed
Network of Autonomous and Heterogeneous
Services

Mikel Solabarrieta Román

Director: Diego Casado Mansilla

Bilbao, junio de 2020

A handwritten signature in black ink, appearing to read 'Diego Casado Mansilla'.

Abstract

This degree thesis documents the design, implementation and test of a network of autonomous and heterogeneous services to enable automation of service discovery and utilization. On the one hand, autonomy of the services is defined by their capacity to operate independently from other services. On the other hand, heterogeneity implies services have distinct properties and purposes. Finally, decentralization refers to a network whose peers act on local information that they obtain from and transmit to other peers.

The services will be described using the Blindingly Simple Protocol Language (BSPL) which was designed with these types of networks in mind. The network, peer discovery and communication; and therefore, service discovery and interservice communication, will be designed and implemented using the LibP2P networking stack and its implementation in the Go programming language.

The degree thesis is divided in four main stages. The first one is the implementation of BSPL in Go for compatibility with the rest of components of the thesis. The second one is the design of the distributed network; mainly peer discovery and communication based on the LibP2P specification. The third one is the implementation of the distributed network and its integration with the partial BSPL implementation. The final stage is the development of a testing scenario for the analysis of the performance and behavior of both the individual peers or services and the network as a whole.

Descriptors

Autonomous Services, BSPL, Distributed Systems, LibP2P, Peer to Peer

Contents

1	Introduction	1
1.1	Context	1
1.2	Motivation	2
1.3	Problem Statement	3
1.4	Justification	4
2	Scope	7
2.1	Project Scope	7
2.2	Ethical Considerations	7
2.2.1	Source Code Availability and Licensing	8
2.2.2	Network Accessibility	8
2.3	Objectives	8
2.4	Project Structure	9
3	State of the Art	11
3.1	Multi-Agent Systems	11
3.2	Interaction-Oriented Design	12
3.3	Peer-to-Peer Networks	13
4	Technologies	15
4.1	Blindingly Simple Protocol Language	15
4.2	LibP2P	17

4.2.1	Discovery	18
4.3	Programming Language	19
4.3.1	Characteristics	19
4.3.2	Justification	22
5	Technical Objectives and Requirements	25
5.1	Functional Objectives	25
5.2	Non-Functional Objectives	27
5.3	Additional Specifications	29
5.3.1	Stakeholders	29
5.3.2	Use Cases	29
5.3.3	Restrictions	30
5.3.4	Design Specifications	31
6	Budget and Planning	33
6.1	Budget	33
6.2	Planning	34
7	Methodology and Resources	39
7.1	Methodology	39
7.2	Resources	40
7.3	Development Methodology	40
7.3.1	Version Control	40
7.3.2	Testing	41
7.3.3	Dependencies	43
8	Development	45
8.1	BSPL Module	45
8.1.1	Sub-modules	46

8.1.2	Presentation	53
8.2	NaHS Module	54
8.2.1	Events	54
8.2.2	Network	56
8.2.3	Operation	57
8.2.4	Presentation	59
8.3	Demonstration	59
8.3.1	Simple Demo	59
8.3.2	Complex Demo	64
8.4	User Manual	66
9	Results	69
9.1	Libraries	69
9.1.1	BSPL	70
9.1.2	NaHS	71
9.2	Demo	72
10	Conclusions	73
10.1	Future Work	73
10.2	Personal Assessment	74
	Bibliography	77

List of Figures

1.1	NaHS logo	4
1.2	Network topology	5
2.1	Project structure	10
3.1	Agent logic level and network level	12
3.2	Different levels of network centralization inside a distributed network	14
4.1	Causality determined by information flow	16
4.2	LibP2P logo	17
4.3	Simplification of Rendezvous	19
4.4	PHP, Java, Node and Go server comparison	21
4.5	Go concurrency example with code samples and a flow diagram	22
5.1	NaHS agent architectural overview	31
6.1	Project schedule	37
7.1	Kanban board	39
7.2	Screenshot of the header section of the BSPL README.md file	41
7.3	Output shown in the editor after automatic tests	42
7.4	Coverage shown in the editor after automatic tests	42
7.5	Report of CI tools after a successful commit	42

8.1	UML diagram of bspl.proto	46
8.2	Circular parameters between BSPL messages	47
8.3	Overview of the BSPL parser	48
8.4	UML diagram of bspl.parser	51
8.5	UML diagram of bspl.reason	51
8.6	UML diagram of bspl.implementation	53
8.7	UML diagram of nahs.events	55
8.8	From Go objects to serialized data	55
8.9	Example of hash, multihash and ID	56
8.10	Information flow when reasoner sends a message	58
8.11	UML diagram of nahs.net	60
8.12	Section of the output log of the complex demo execution	67
9.1	Screenshot of the README.md file from the NaHS repository	70

List of Tables

5.1	Requirements of objective O1	25
5.2	Requirements of objective O2	26
5.3	Requirements of objective O3	26
5.4	Requirements of objective O4	27
5.5	Requirements of objective O5	27
5.6	Requirements of objective O6	28
5.7	Requirements of objective O7	28
5.8	Requirements of objective O8	28
5.9	Requirements of objective O9	29
5.10	Requirements of objective O10	29
6.1	Cost of human resources	33
8.1	BSPL submodules	45
8.2	BSPL Go types	46
8.3	Descriptions of some of the methods of the Instance interface	52
8.4	Descriptions of some of the methods of the Reasoner interface	52

Listings

4.1	BSPL bike rental example	15
4.2	DB management example with defer	20
4.3	DB management example without defer	20
4.4	Defer, panic, recover example	22
7.1	Test example	43
7.2	TestMain example	43
7.3	Contents of bspl/go.mod	43
7.4	Contents of bspl/go.sum	43
7.5	Partial contents of .profile	44
7.6	Partial contents of .gitconfig	44
8.1	Definition of SortParameters	47
8.2	Definition of sort.Interface	47
8.3	JSON describing a deterministic finite automaton to process a BSPL protocol	49
8.4	Token table generated by the lexical analyzer	49
8.5	Final parsing function	50
8.6	Type aliases in BSPL module	53
8.7	Section of BSPL protocol discovery source code	57
8.8	Exported types and functions at NaHS package	59
8.9	Bike rental protocol definition	61
8.10	Person and personReasoner types	61
8.11	Example of an Instantiate method	62
8.12	Example of an Update method	63
8.13	Bike ride protocol	65
8.14	Bike request protocol	65
8.15	Bike transport protocol	65
8.16	Bike storage protocol	66
8.17	Station search protocol	66
9.1	BSPL module usage example	71
9.2	NaHS node creation example	71

Chapter 1

Introduction

1.1 Context

Automation and digitalization of tasks and services have been gradually increasing over the past decades. The first reported cases of automation date back to thousands of years ago but didn't become really relevant until the industrial revolution. At this point, it became clear the potential of automation of processes, which combined with other factors led to the automatized society we live in today. Compared to the past, the amount of automatized processes is extreme. From a contemporary perspective, however, there is still great automation potential that needs some requirements to be met before being possible. One of those requirements is the potential interactions automatized processes may have between each other to have more autonomous functionality. Autonomous refers to the capacity of the process to be carried out without requiring a person to carry parts of it out. For example, in this context, an autonomous payment process is one that requires no cashier.

It is worth saying that currently, the greater part of automating processes is the logic behind them. This logic requires knowledge of what needs to be done, how it needs to be done, and what must be considered while doing it. All of this is digitalized, which is why many of the current problems and business actions involve digitizing relevant data and procedures.

Buying train tickets, office supplies, renting a car, etc. are cases that traditionally required the interaction of multiple people. However, currently only require one person. As imagined, these require manual interaction that go beyond just stating what is wanted in most cases (e.g. further information about the train, or the platform where the train will stop). The specific cases that only require minimal interaction do so because great effort has been put into enabling the automatizing of very specific actions.

Considering these services, the next step after digitizing services and automatizing them to some degree seems to be able to connect them. In this way, it is possible to make the information they produce understandable by software that can later interact with the others according to that information. This would enable a convenient and generalized way of looking for and interacting

1.Introduction

with service providers that should be open and accessible to anyone at any time. To meet these requirements, service providers and consumers should form a network in which anyone can take part and be available for others.

For example, an agent representing a company that wants to obtain paper should be able to join a network where many other agents are consuming and providing different, heterogeneous services and be able to see if any of the agents offers a paper providing service. If such an agent were found, the way the two agents (consumer and provider) will interact should be specified by some kind of protocol. In this context, an agent is a node on the network that can offer and/or consume services with other nodes of the network. In the example, obtaining paper might not be the only task the agent is managing, thus it is behaving autonomously in two senses: following its own directives and not depending on the nodes to perform other tasks.

The goal of this project is to create a network of autonomous, heterogeneous agents that enables this kind of scenario and tests it with a simulation scenario. The project has been named “Network of Autonomous and Heterogeneous Services” or “NaHS” for this reason.

1.2 Motivation

The work of Munindar P. Singh and Amit K. Chopra in 2017 was the start launcher this project. In [1] they develop the idea of automation-dependant fields require human intervention and both centralized knowledge and logic that started this project.

Advancements such as modern low-power processors and the recent deployment of 5G networks would enable almost anything to be part of the network, making the entry-point of data transfer rates and processing power more accessible to people and devices. Complementary to these developments, there are two important factors that need to be considered:

- The demand for automation is, for now, ever-increasing;
- Enormous amounts of data are being generated, many of which have more potential uses than the one they are subjected to.

Putting it all together, the situation seems to be that there is a lot of data that could be used to automate processes and services, there is enough device and infrastructure technology to automate the processes and services, and there is a societal demand to do so. Not only for the sake of automation but also useful exploitation of data that will be generated anyway.

Due to the social nature of this project and the impact a successful network may have, it is essential that it is free software accessible for and usable by anyone. It is also essential that agents are independent and have the freedom to choose what other agents they interact with. For this, the network must be distributed and agent reasoning must be based on their local information, similar to a multi-agent system. This is not to say this project will result in the ideal network, but it should be developed with an exemplary attitude towards future advancements in the fields.

Personal interest in subjects such as automation or distributed systems and the recognition of the relevance of systems intended for wide adoption to be developed with openness in mind are also important factors regarding the motivation behind this project.

1.3 Problem Statement

Despite recent advancements in fields such as IoT that require by definition high degrees of automating and distribution of both logic and data, most modern systems centralize logic, data and require manual intervention of humans to operate. An example will be provided to illustrate the problem. A person that wants to rent a bicycle needs to manually search for a bicycle provider with centralized information lookup services such as the Google search engine and check that the provided ones meet their criteria. Something similar happens when a system or person needs meteorological data, news feeds, etc. In all of these cases different agents look for different services that are described and interacted with in very different ways.

As mentioned in the previous section, there is also a potential for interaction between automated systems that isn't being met yet. The exploitation of this potential should be made gradually, openly and with flexible protocols and procedures that allow change and growth so they don't fall behind as the field matures. It is also crucial that the system is not centralized so that control is distributed, participants are free to choose their interactions and the interest of a central authority doesn't compromise the project.

The proposed solution which is presented in this degree thesis is a network where services offered can be described, publicized and consumed for and by agents openly and independently. The characteristics, needs and reasoning of each agent may be different but they still require a way of interacting with each other. To achieve independence of the agents the network must also be distributed, avoiding central points of knowledge, control and failure. By making it truly distributed and standardizing but not imposing interactions, more complex systems such as private networks within the network can be created. These characteristics describe multi-agent systems, where each agent has autonomy, a local view of the network as a whole and no single agent has control over the network.

The resulting code of the project is split between three repositories:

1. **BSPL** - <https://github.com/mikelsr/bspl>
2. **Networking** - <https://github.com/mikelsr/nahs>
3. **Demo** - <https://github.com/mikelsr/nahs-demo>

Each of the repositories will now be explained. The BSPL repository manages protocols. For two agents to interact, there must be a set of rules they must follow to make the interaction possible. Protocols described with BSPL specify those rules. The agent acting as a service provider defines the rules of interaction with a protocol, and the consumer agents understand the rules from that protocol. The BSPL package will therefore implement protocols and protocol enactments, as well



Fig. 1.1: NaHS logo

as providing ways to share them in plain text. In the paper-obtaining example, the protocols would allow the company to request paper from a supplier that has defined a protocol for that specific interaction. BSPL would define how to request and obtain paper. Once service interactions are defined, the agents still need to find each and physically exchange data in order to interact. The networking repository takes care of this meta-interactions, by providing tools to form a peer-to-peer network where each agent is a node and has tools to discover other nodes and send/receive that that contains the protocols and enactments described in the BSPL package. Agents interact directly with each other, therefore having a peer-to-peer model is vital as it mirrors the logical interactions, in addition to the benefits described before. In the example, a computer would need to locate other computers that might provide paper and contact them, which is what the networking module does. Finally, the demo package creates a scenario where heterogeneous agents interact with each other to provide and consume different services. The scenario is centered around bicycle rentals and has different agents: some rent bikes, others transport or charge them... Each of the libraries is further explained in their corresponding development section of chapter 8, including a detailed explanation of the demo in section 8.3.

1.4 Justification

Many of the ideas presented in the previous section are not new. Distributed networking technology already exists. However, descriptive automation is still very immature field of research. The networking stack used in this project is described and justified in 3.3, but it is worth advancing that the chosen distributed networking stack is LibP2P [2]. LibP2P is a modular, documented and actively developed distributed networking stack that has been proved in projects such as the notable IPFS [3]. IPFS an initiative for a distributed web that has a similar topology to the one intended for this project. Being built over LibP2P proves the networking stack to be a great choice for this project. Addressing how to automate interaction, the Blindingly Simple Protocol Language [4] has proven to be a conceptually matching proposal that while immature, serves its purpose well.

This project defines how nodes should interact and builds a networking library to enable interactions. More specifically, it implements BSPL from the ground up and builds a networking library on top of it to match the goals of the network proposed in “Problem Statement” section.

Figure 1.2 shows the topology of the network. Each dot represents a member of the network: a person or device that consumes or offers a service. Each line represents an interaction between two nodes. Nodes can explore the network by communicating with others and find for nodes that offer something they want. They can also choose to accept or reject interactions with other nodes and

are therefore in control of their actions. The group of three nodes at the bottom connected to the main network with a discontinuous line represents a private network of three nodes, but one of the nodes requires something from a node that is part of the public network.

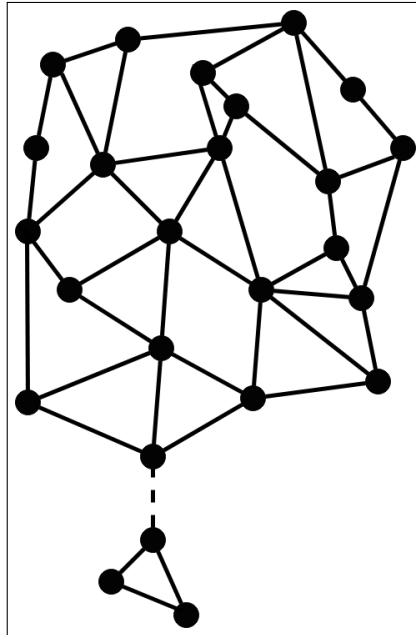


Fig. 1.2: Network topology

The network is intended to be unstructured and dynamic: nodes will rise, interact and fall without breaking the network. For this discovery mechanisms need to be implemented, discussed in chapters 3, 4 and 8. This topology and instability also mean interactions between nodes need to be asynchronous: nodes need to be able to continue functioning even if other nodes haven't responded to their requests yet. For this purpose, internal node logic and interactions between nodes need to be designed with asynchrony in mind.

Chapter 2

Scope

2.1 Project Scope

The project aims to create the foundations of a distributed network in which any type of agent can interact with others to offer or consume services. The network needs to be distributed to maintain users' freedom and self-governance. Furthermore, it should be flexible so agents may connect and disconnect without disrupting other nodes. Finally, it should be standardized so even if two nodes have no services in common they can still exchange information about the known nodes for the benefit of both of them.

The final result of the project will be:

- A software implementation of a protocol description language: BSPL.
- The design of a distributed network of heterogeneous, autonomous devices.
- A software implementation of the network, using the BSPL implementation and LibP2P networking stack.
- A real use case of the project using the implemented software that contains multiple service providers and consumers that interact with each other.

2.2 Ethical Considerations

This section discusses the ethical consideration of the project from two perspectives: the availability of the source code and the accessibility of the network that is the goal of the project.

2.2.1 Source Code Availability and Licensing

This project is possible thanks to selfless contributions of others. As explained in section 4.3, the language is free software that doesn't compromise their users. LibP2P, initially a part of IPFS started by Juan Benet and Protocol Labs, is also free software that offers complete peer-to-peer functionalities and has not only the source code publicly available but also the discussion and research behind it. Other components, such as BSPL, have also been publicly exposed as part of research efforts.

All of these projects follow an ethical attitude, that not only can be generalized but should be generalized. It propels and enriches research, makes science and technology almost universally accessible and inspires others to do the same. Efforts have been made to develop this project using only free software and produce libraries that are publicly accessible and comply with the GNU definition of free software [5].

2.2.2 Network Accessibility

As mentioned previously, the network needs to be open to everyone. The first requirement for this is that the code-base is not only available but can be legally used, modified and distributed. The second requirement is that the network as a whole doesn't discriminate nodes. This is not to say nodes are forced to interact with others: they choose to do so. There are multiple reasons why a node may refuse to interact with another: breaches of trust, arbitrary decisions... but it must never be because a centralized authority has decided so. In an extreme case, every single node in the network may decide to avoid interactions with a specific node. From the neglected node's perspective, the network is against him, but the global perspective is that each node has decided on its own ignore that node. This anarchical approach has room for initiatives as trust systems, systems that define how the success or failure to fulfill a compromise with another node is communicated through the network so that other nodes take it into consideration when interacting with it. This topic alone brings up a lot of questions: what if a node lies or there was an unavoidable error? This could shift the network to a more democratic behavior, which would change its implications. These topics are not, however, in the scope of this project, but might be interesting as general reflection and future work.

2.3 Objectives

The final results of section 2.1 are derived from the objectives of the project, which are listed below. Each objective and their corresponding requirements are analyzed in depth in chapter 5.

- **O1** Provide a functional, tested BSPL parser.
- **O2** Provide a functional, tested BSPL implementation.
- **O3** Provide a networking library based on BSPL to enable the creation of NaHS agents.

- **O4** Provide a demo where agents interact with each other to meet their individual goals.

Non-functional objectives:

- **O5** Code must compile without any errors or warnings with the Go compiler.
- **O6** All critical functions must be tested.
- **O7** The most relevant types and functions must be exposed with the main package.
- **O8** The code must be clear and understandable.
- **O9** Documentation must be provided for the resulting code and underlying ideas.
- **O10** Software licenses must be compatible.

2.4 Project Structure

This section defines the structure of the project development and the methodology. From laying foundations with research to delivering the final product. In short, the first step is research previous languages and existing solutions. Then, design/implementation/testing the different components. Finally, ending with analysis, conclusions and documentation revision. Each step shown in figure 2.1 is further detailed below.

1. **Research protocol description and distributed networking:** Search for information about how to describe interactions between parties, modern P2P networking libraries, etc. Understand all the foundations and choose the appropriate approach and libraries for this project degree.
2. **General design of NaHS:** Create a general design of the network. Answer the following questions: What is a node of the network? How does it reason? How does it interact with others?
3. **Design, implementation and testing of the BSPL parser:** Develop a library to, given a raw-text BSPL protocol, create a class/structure that represents the protocol. This involves the lexical analysis of the text, parsing the generated token table and generating the corresponding data structures. This should bring about a library released in free software and ready to use by anyone
4. **Design, implementation and testing of the BSPL instances:** After creating protocols, and as explained in section 4.1, interactions need to be instances of protocols. This requires a more thorough understanding of BSPL and is separated to the parser to allow independent, future reworks. Interface-based design allows the two modules to exist separately.
5. **Design, implementation and testing of the networking components:** Build a networking library over LibP2P that revolves around BSPL. More specifically, BSPL reasoners as explained in section 4.1.

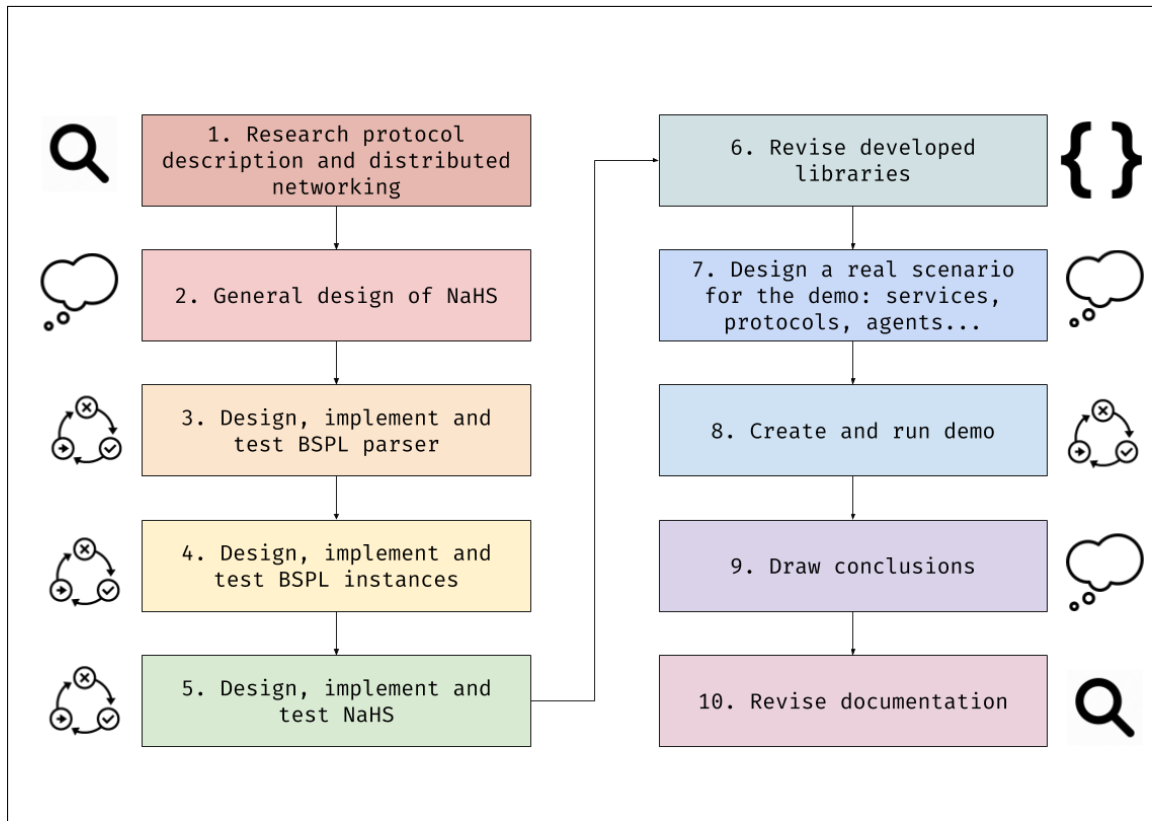


Fig. 2.1: Project structure

6. **Revision of the previous steps:** Check that the project is consistent and that NaHS agents can indeed be built using the created libraries.
7. **Design of a real, plausible scenario for the demo:** Create a real example where NaHS could be used. Answering the following questions: Who takes part in the interactions? How should they be carried out? How is that represented digitally?
8. **Creation and execution of the demo:** Develop the demonstration, gradually increasing the complexity by adding new actors.
9. **Analysis and conclusions:** Analyse the results and draw conclusions. Lay down future work.
10. **Documentation revision:** Revise the project and code documentation.

Chapter 3

State of the Art

This chapter will briefly describe three relevant topics for this project: multi-agent systems, interaction-oriented design and peer-to-peer networks. The concepts are closely related, as they refer to how different entities are designed, communicate and interact. The objective of the chapter is to provide some insight into the project and give more context before the technologies are explained.

3.1 Multi-Agent Systems

Multi-agent systems are systems composed of agents that interact with each other to fulfil a purpose. “Agent” in this context refers to any intelligent entity (representation of a person, device or object) that interacts with others, while intelligence is defined by the capacity of the agent to correctly interact with others and process the results of the interactions.

Multi-agent systems have three main characteristics that have previously been discussed on this document. The first one is the autonomy of the agents, who must be able to function even without interactions with other agents. The second one is that agents have only a local image of the network, which may be incomplete as all the information they receive comes from other agents they’ve been in contact with. The third one is that no single agent has the specific task of controlling the network. In this project, the MAS will be implemented with peer-to-peer networking, as it is the model that best fits the design NaHS.

Usually, multi-agent systems are used to solve complex problems by modeling individual components of a physical or logical scenario into agents that will then interact in a similar way the real component would. As an example, a MAS might represent a bee hive by creating bee agents that mimic the behaviour of real bees [6], or a MAS system with car agents might be created to predict traffic in a city.

How agents interact with each other in a logical level or how they discover each other or send/receive information will be described in the BSPL (section 4.1) and networking 8.2.2 sections, respectively.

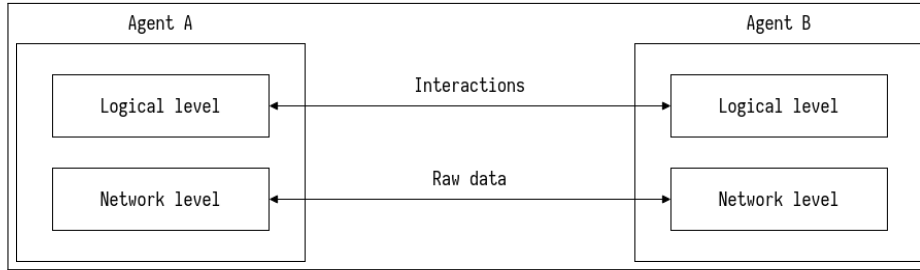


Fig. 3.1: Agent logic level and network level

For now it should be said that the agents will communicate directly with each other, forming a peer-to-peer network. Figure 3.1 shows how the logic level communication represents interaction between two nodes (has meaning) while the network level uses raw data (without meaning). The network level is also used to discover other agents.

Although multi-agent systems cover a wide range of sub-types, the main characteristics of NaHS is that agents interact on two levels: the logical level and the networking level. The logical level is defined by interactions: enactments of protocols described with BSPL as will later be explained. The network level involves discovery and the transmission of data before the data is interpreted by the logical level.

3.2 Interaction-Oriented Design

The meaning of interaction-oriented design might depend on what is being designed. In the context of this project, it refers to designing agents of a multi-agent system around their interactions. Similar ideas have been discussed previously, for example interaction-oriented software design [7] is based on the idea that interactions in complex environments can be hard to predict and regulate. By defining interactions with high-level abstractions the design of the components can be made more flexible and robust than if interactions were defined as sequences of processes, messages and events. On the same line, [8] expands the definition of interactions between MAS agents from just message exchanges to the representation of physical interactions to create MAS based simulations.

The principal benefit of interaction-oriented design is separating the interaction of the implementation. When two parties interact by following a protocol, they must be abstracted of how the other party implements the protocol as long as they follow it. By shifting the design of the agents from a set of events and messages that consider internal implementation to well defined, abstract interactions, the project intends to provide flexibility that will enable the network to contain many different agents that are able to interact with each other regardless of their own internal processes. Therefore, the only thing agents must implement regarding communication with other nodes are the interactions described with high-level abstractions without worrying about how other agents might have implemented them.

In NaHS, every interaction between agents is via the enactment of protocols described with BSPL [4], which will be explained in section 4.1. To put it into perspective with the example of chapter 1, an agent wanting to obtain paper will be given an interaction protocol by the paper supplier. If the

first agent follows the given protocol it will be able to obtain paper regardless of how the provider agent implements any of the processes executed to provide paper such as accessing databases or internal component interactions.

3.3 Peer-to-Peer Networks

Peer to peer networks are computer networks where each computer acts as an equal to the rest, instead of the client/server behavior traditional networks use. The role of a client is usually to initiate connections and the role of the server is to respond to it, however in P2P networks every node can implement the behavior of both client and server. Members of the network are referred to as peers. Peers exchange not only data and information about other peers between each other. There are multiple P2P network models, in the one used in this project each node maintains a local image of the network but does not update it with a “centralized”, controller node such as the tracker nodes used in the BitTorrent protocol. Instead, all the information a node has about the network comes from the interaction with other, equal nodes. It should be mentioned that BitTorrent also has measures to avoid centralization, such as using distributed hash tables. DHTs are an interesting topic and represent the state-of-the-art in distributed storage. They have gained traction with the rise of distributed applications such as cryptocurrencies and could be studied for their use in NaHS in some future iteration, for example, to look for service providers.

In the case of P2P networks such as the one proposed in this project. Distributed networks, including this one, could still have some levels of centralization. In the case of NaHS, if few nodes offer services consumed by many, the state of the network would have a higher centralization degree than if the distribution of service providers/consumers were more regular. Figure 3.2 shows two distributed networks: the one on the left has a smaller degree of centralization than the one on the right. It should be noted that more centralization can have negative implications, such as centralized control contrary to the design of the network, but can also have positive ones, such as discovery nodes that help maintain connectivity on the network.

The network proposed in this project contains an unstructured distributed network, where no fixed network topology exists, nodes can appear/disappear at any time, and the information each node has about the global network is entirely based on local interactions. NaHS agents interact directly with each other, with no intermediary entities. For this reason, P2P networking fits the design of agents as it makes agents directly connected, without intermediaries.

Despite their many advantages, P2P networks have some issues that require mentioning. The first one is sometimes P2P are not possible due to the network configuration of one of the nodes. For example, if it is behind restrictive firewalls or multiple NAT layers, establishing and maintaining P2P connections might not be possible. Another problem might be the verification of node identities. The last one mentioned here is the latency issues such a network may have, specially when compared to centralized networks. Fortunately LibP2P, the P2P networking stack used in this project, implements features to solve these problems to some degree. The pending work is discussed in section 10.1.

There were several choices for P2P networking libraries, and although LibP2P is the one actually

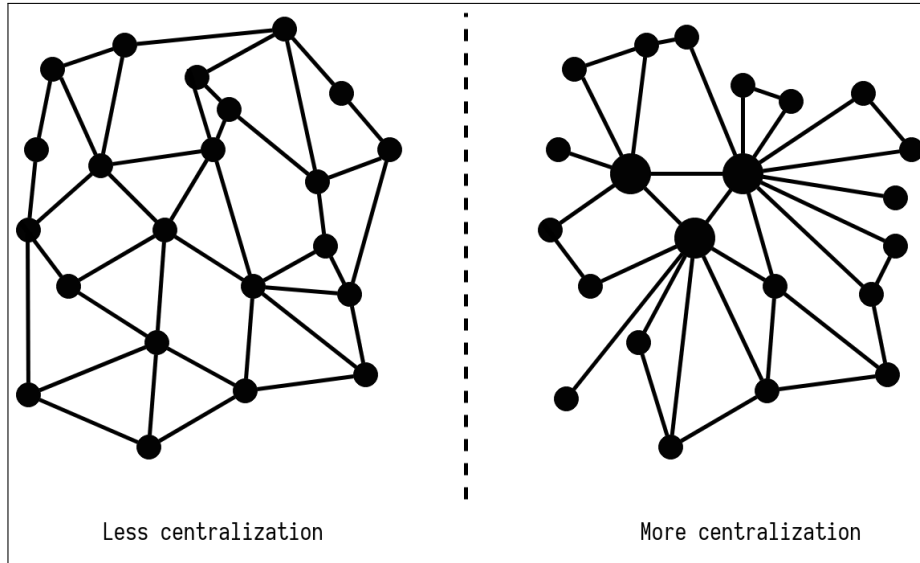


Fig. 3.2: Different levels of network centralization inside a distributed network

used, Noise [9] deserves special mention. Noise is a P2P networking framework developed in Go (the programming language is discussed in 4.3) intended to provide easy to use P2P capabilities. It is designed with performance, security and developer-friendliness in mind. It has many distinctive features worth reading, either in the official documentation [9] or in the article written by Kenta Iwasaki [10]. As a final note, LibP2P has its own implementation based on noise, under the “go-libp2p-noise” repository [11].

Chapter 4

Technologies

This chapter describes many of the technologies used in the project, some of which are closely related to the topics discussed in the State of the Art chapter. Some methodologies are described in chapter 7 instead of in this one because they heavily influence the development methodology, explained in that chapter.

4.1 Blindingly Simple Protocol Language

BSPL is a protocol description language introduced by Munindar P. Singh in [4]. In the article BSPL is described as a way of describing communication protocols for interaction-oriented applications.

A BSPL protocol consists of a protocol name, two or more roles, parameters and actions. Roles are used to determine what role each enacting participant will play. The parameters section contains the information about the protocol and will give meaning to the enactment once values are given to the parameters. Actions contain an origin role, a destination role and a message. Messages contain parameters that give them meaning. If BSPL protocols were programming classes, enactments would be instances (in fact in the BSPL implementation included in this project calls them “instances”). Roles are assigned and parameters take values, always as strings for the sake of simplicity. Each of the components is explained after taking a look at an example.

```

1 BikeRental {
2     role Customer, Renter
3     parameter in origin, out ID key, out bikeID, out price, out rID
4
5     Customer -> Renter: request[in origin, out ID]
6     Renter -> Customer: offer[in ID, in origin, out bikeID, out price]
7     Customer -> Renter: accept[in ID, in bikeID, in price, out rID]
8     Customer -> Renter: reject[in ID, in bikeID, in price, out rID]
9 }
```

Listing 4.1: BSPL bike rental example

Listing 4.1 shows the protocol used in the demo to rent a bicycle and will become clearer along this section. There are two roles: a customer that rents the bike, and the renter that provides it. The protocol has five parameters: `origin`, `ID`, `bikeID`, `price` and `rID`. `origin` refers to the initial location of the customer. `ID` identifies the enactment of the protocol. `bikeID` identifies the rented bike and `price` the price the renter offers the customer for the bike. `rID` is used to accept or reject the rental. There are four possible actions: request a bike from a renter, offer a bike to a customer, and either accept or reject the offer.

BSPL is based on four principles, as explained in [4]:

- **Information orientation.** The approach is based on the objective of interactions: share or obtain information. In BSPL this is done via protocol parameters: items of information exchange between the protocol participants.
- **Explicit causality.** The causality of the protocol and the instance is derived from the parameters. This is explained later in this section.
- **No states.** The state of the protocol enactment is explicit in the values of the parameters, no global state of the enactment must be maintained. The state of the other participants is irrelevant to each participant.
- **Separation between structure and meaning.** The protocol structure is defined by the names of the components, but the meaning is given by the values assigned to them by the implementations.

The components of a protocol and the values given to each enactment are immutable to ensure consistency between message exchanges, even asynchronously. Each enactment must also be unique, thus the union of key parameters must not coincide with that of other enactments.

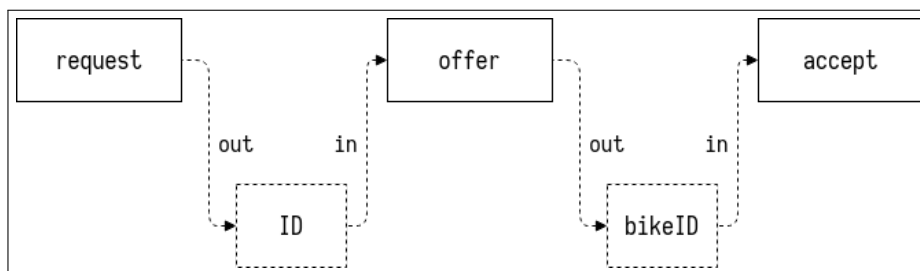


Fig. 4.1: Causality determined by information flow

As seen in the aforementioned bicycles example, parameters are adorned with keywords. These keywords are `key`, `in`, `nil` and `out`. `key` indicates that it is a key parameter used to determine the uniqueness of the enactment. `in` indicates the parameter is known when the protocol or message is first enacted. `out` indicates that the parameter is unknown at the beginning of the enactment. `nil` is ambiguous and indicates that the parameter might not be known at any time. Messages are not ordered in the protocol definition, instead, the order is derived from causality. Figure 4.1 shows how, in the example of listing 4.1, the order of the actions depends on the flow of information. Request generates the `ID` parameter and therefore must precede any other action that requires the



Fig. 4.2: LibP2P logo

ID. In the same way, offer assigns a value to `bikeID` and `price` variables, and thus it must precede actions that require them. In the figure, messages are represented with rectangles, parameters with dotted rectangles and causality with discontinued arrows.

The author gives the idea that protocols must be enacted via local state transfers or LoST [12]. The local state of a participant refers to what the values the participant has of public parameters of the protocol. This, as previously explained, could allow the extraction of actions from the parameter difference between the remote state and the local one. For example, in listing 4.1 if the renter didn't know the ID and receives the local state of the client containing it, `request` must have happened.

4.2 LibP2P

LibP2P is a modular network stack. The modularity is due to the project being a combination of specifications, protocols and libraries that form LibP2P as a whole. One of its main objectives is to provide a set of P2P functionalities that can be cleanly separated so that developers use only what they require. The logo, shown in figure 4.2, is supposed to represent the modular components forming the whole networking stack. It is currently implemented in JavaScript, Go and Rust; but Haskell, Java and Python implementations are already in development.

Originally, LibP2P was a part of IPFS: a project to create a distributed, open web [3]. Building an application over LibP2P based on the shared values with IPFS, LibP2P and the rest of the projects started by Juan Benet and developed by Protocol Labs is one of the reasons LibP2P was chosen over Noise. Despite this choice, the value of Noise is undeniable. The values and objectives of IPFS are very much in line with what was described in section 2.2, and IPFS stands as one of the P2P projects with more potential for technological and societal impact.

Back on libP2P, the variety of components that form it gives developers a lot of flexibility. The design decisions and specifications are all available in the GitHub repository for libp2p specs [13]. Specifications are language independent and allow understanding of the stack without knowledge

of specific programming languages. It also serves as a knowledge pillar for new implementations of libp2p. Besides the specifications, both libp2p and the implementations have abundant documentation, albeit still in development. Accessibility seems to be important for the team behind libp2p and basic concepts of P2P networking are also clearly explained in the documentation page. Transport, peer identity, content routing, security, etc. Each of the components is explained, documented and forms the building blocks of libp2p. The library also introduces a personally new concept: stream multiplexing. Stream multiplexing is used to handle multiple communication protocols in a single established connection.

The implementation used in NaHS is the Go implementation, found in multiple repositories but the main one being `go-libp2p` [14]. The repository contains types, functions and interfaces that are well structured and allow for software design before using each specific component hosted in other repositories under the same GitHub account. It also contains a list of every other component of the Go implementation, their build status, code coverage and description.

Due to its ties with IPFS, there is plenty of deployed infrastructure using libP2P. In the case of NaHS, the nodes IPFS have been deployed for peer discovery serve as a rendezvous point for nodes that want to join the network but lack contact information about any other peer. A protocol with the name Rendezvous is used in NaHS as a discovery mechanism, using the servers provided by IPFS.

LibP2P nodes use multiaddresses, which contain information about the network addresses and protocols. An example of a multiaddress is `/ip6::<1/tcp/4242`. A node listening in that multiaddress is expecting connections to come from the `::1` loopback address and TCP connections to go to the 4242 port.

As a closing statement, libp2p is an actively developed project with input from both Protocol Labs and the community and represents an opportunity for anyone who wants to develop a decentralized, peer-to-peer application.

4.2.1 Discovery

NaHS uses the rendezvous protocol [15] for agent discovery. It is a lightweight discovery mechanism that allows the discovery of peers of multiple networks, but NaHS nodes only look for other NaHS nodes.

The following is a simplified explanation of the Rendezvous protocol with a hypothetical example. Node A wants other nodes of NaHS to find it, so it contacts a bootstrap node and tells it to include it on a list of nodes running NaHS. The bootstrap node is a rendezvous point and is natively known by every NaHS node. Now, Node B wants to find other NaHS nodes so it asks the rendezvous point for contact information of other nodes running NaHS. The point then sends a list of nodes running NaHS to node B, which includes node A. When they contacted the rendezvous point, the nodes might have been given a cookie. The cookie allows the point to know what information has already been sent to each node and therefore can, for example, inform them when a new node registers for NaHS without having to send them the full list. Another example is paginating responses to make queries more manageable. Each node can only register itself, therefore node A couldn't register

node B or the other way around. The rendezvous point can be another node, but making it a server makes it easier to find.

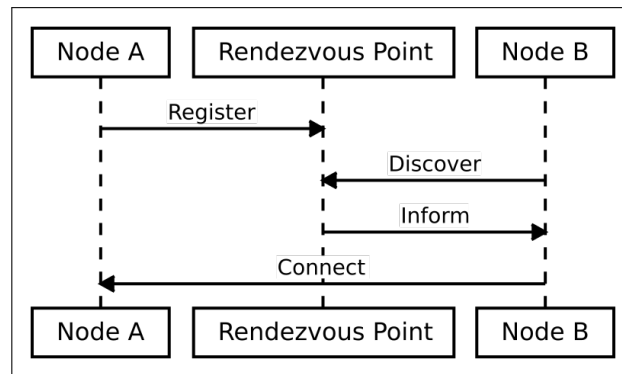


Fig. 4.3: Simplification of Rendezvous

4.3 Programming Language

The project is entirely developed in Go, excluding some utility shell scripts. This section goes through the main characteristics of the language as well as the reasons behind this choice.

4.3.1 Characteristics

Go is a multi-paradigm programming language that runs on most major operating systems. Its BSD-based license makes it compatible with the values this project is developed with. It has two implementations, the one developed by Google and a GCC front-end. The former was used in this project. Native tools such as testing and dependency management are not mentioned in this section but can be found in sections 7.3.2 and 7.3.3 respectively.

4.3.1.1 Syntax

The syntax is very similar to the C syntax. It does however give more importance to simplicity which makes the code more readable and favors simple designs. Types are static and can be either explicitly declared or inferred from the initial value of the variable.

The readability is improved by many factors. A good example is how declaring types after variable names avoids the spiral rule [16] or how starting a name with a capital letter means the type, variable or function that name represents is exported; in contrast to languages like Java where the keywords `public` must be used. Dave Cheney’s “Clear is better than clever” [17] covers good practices in this regard.

The `defer` keyword offers a convenient way of running cleanup operations. `defer` adds a function to a stack, every function of the stack is executed before exiting the function even if it exits due to an error. In the case of this project, this is specially convenient when handling descriptors and

streams. Listing 4.2 shows how to handle a database with defer and listing 4.3 shows how it could be without defer. Note that Go does not have try/catch statements, as explained in section 4.3.1.5.

```
1 db, := NewDB()
2 defer db.Close()
3
4 ...
```

Listing 4.2: DB management example with defer

```
1 db, := NewDB()
2 try {
3     ...
4 }
5 catch(Exception) {
6     ...
7 }
8 db.Close()
```

Listing 4.3: DB management example without defer

Syntax formatting tools ensure that the code is formatted in a certain way, making projects more consistent not only as individual projects but also as part of the Go library which uses mostly the same formatting.

4.3.1.2 Types

Go has structures instead of classes and therefore there is no inheritance. Instead, a similar concept is achieved by interfacing and embedding. Interfaces define the methods the structure will have while embedding, whether by composition or delegation, provide a way of implementing those methods. While inconvenient at times, it does frequently result in clear designs. There are pointers that have the same syntax as in C using the `*` and `&` characters, which are useful for referencing and mutability but can harm performance if used incorrectly.

Methods are not declared inside the types but rather by association to types.

4.3.1.3 Safety and Performance

Go offers competitive performance despite being garbage collected. The performance depends on the quality of the code, the specific application and the configuration of the GC, but the performance/abstraction balance can be impressive. The impact of the Gc is expected to be reduced as the GC matures. The GC avoids some memory leaks, ensures memory safety by checking that no reference is out of bounds and avoiding pointer arithmetic if not used from the `unsafe` builtin module. Figure 4.4 shows the potential of Go. It is taken from an article by Brad Peabody [18] that shows the number of concurrent connections per second a server can handle, out of 5000. Higher is better. The servers are written in PHP, Java, Node and Go. The article contains an analysis of the results.

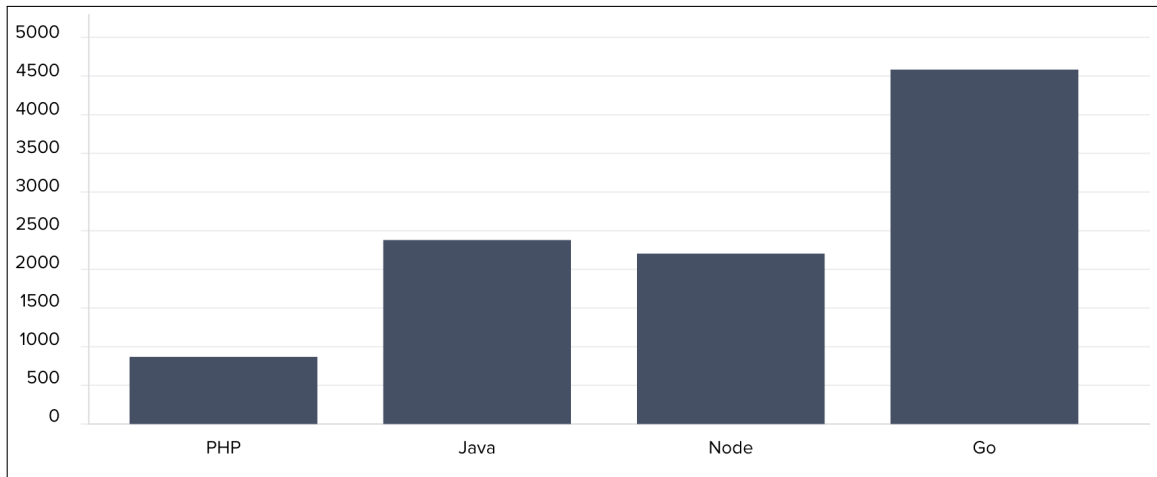


Fig. 4.4: PHP, Java, Node and Go server comparison (source: [18])

4.3.1.4 Concurrency

Quoting “Effective Go” [19], “A goroutine has a simple model: it is a function executing concurrently with other goroutines in the same address space. It is lightweight, costing little more than the allocation of stack space. And the stacks start small, so they are cheap, and grow by allocating (and freeing) heap storage as required. Goroutines are multiplexed onto multiple OS threads so if one should block, such as while waiting for I/O, others continue to run. Their design hides many of the complexities of thread creation and management”.

Communication between goroutines can be achieved in multiple ways. The preferred one is the channel built-in type. A channel a typed variable (meaning each channel has a type, e.g. `chan int` or `chan bool`) that acts as a reference to a variable of that type. Writing to a channel is as simple as `channel <- value` and reading from it is as simple as `value = <-channel`. This communication model avoids mistakes that happen when implementing shared memory communication by ensuring type safety and that only one goroutine accesses a value at any given time. Reading from a channel blocks execution of the routine so others can be executed meanwhile. Other traditional tools such as mutexes can be used too and the model is not perfect as mutability and shared memory result in race conditions being possible in Go. It is worth mentioning that the `select` control structure is equivalent to a `switch` control structure that refers to communication operations instead of values. Figure 4.5 shows an example of a simple concurrent Go program. When launched, function `a` receives a channel, waits three seconds and writes a value to the channel. Function `main` is run at the start of the program, creates a channel, calls `a` with that channel, reads a value (which blocks the execution of the main routine until a value is available) and prints it. The program will therefore print a string after three seconds of starting.

4.3.1.5 Errors

Errors in most programming languages are just variables that indicate that a procedure can't be completed. In many programming languages, the apparition of an error interrupts the flow of the program. The possibility of the error appearing is also not explicit in many situations, for example,

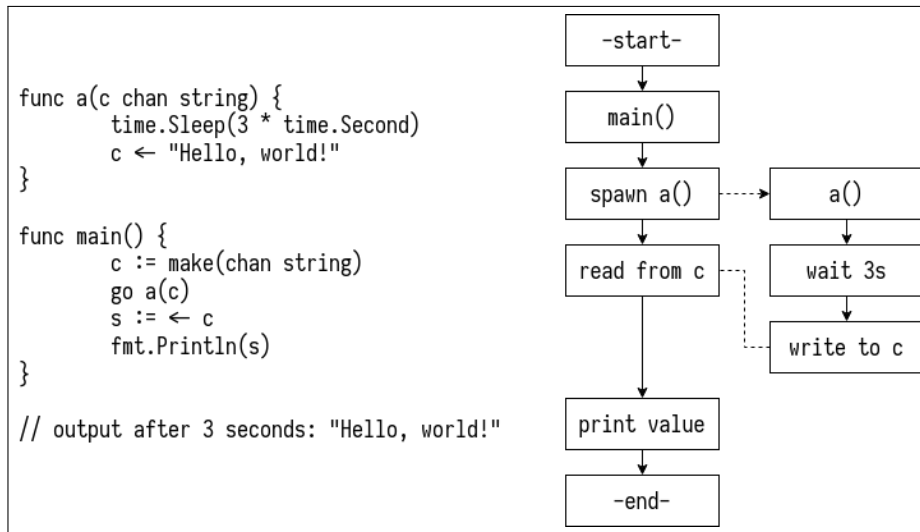


Fig. 4.5: Go concurrency example with code samples and a flow diagram

a Java or C++ method might throw an exception that is not in the method definition.

Go solves this problem in two ways. The first one is treating errors the program can handle as standard variables of type `error`. The error is declared in the function signature as another output parameter, so when the caller receives it, it is handled without interrupting the execution flow. If no error happened, a null value is given to the error.

The second way is the `panic` facility. `panic` is used when the error is vital and the program can't continue, the error is severe enough to interrupt execution. There is a facility to recover from panics, appropriately named `recover`. It is a necessary tool, for example to recover from tests that intentionally panic, but should not be used as a try/catch mechanism. The previously mentioned `defer` keyword. An example of the so-called “defer, panic, recover” dynamic is shown in listing 4.4, which outputs “Recovered from error someError”.

```

1 func f() {
2     defer func() {
3         if r := recover(); r != nil {
4             fmt.Print("Recovered from error ", r)
5         }
6     }()
7     panic("someError")
8     fmt.Println("This will not be executed")
9 }

```

Listing 4.4: Defer, panic, recover example

4.3.2 Justification

The most binding reason is that the project dependencies, such as one of the most complete IPFS implementations and the lexical analyzer are written in Go. The characteristics described above, e.g. language design and concurrency, were also important factors for choosing Go. Python

would also have been a good choice, specially with the relatively recent development of `mypy` for static typing and `asyncio` for concurrency, but the dependency factor made Go stand out. The performance and memory advantage of Go over Python could also be relevant if this project is deployed in low-power devices.

Chapter 5

Technical Objectives and Requirements

Each of the objectives listed in section 2.3 are defined and explained in this chapter. Each objective has certain requirements. The full identifier of a requirement is obtained by appending the requirement identifier to the objective identifier. For example, the full identifier for requirement **R1** of objective **O1** is **O1R1**. Requirements that require additional specifications have a specification section. Specifications are fully identified by appending the objective, requirement and specification identifiers. For example, **O1R1S1**.

5.1 Functional Objectives

Objective 1

Objective **O1** makes reference to the BSPL parser that needs to be created to be able to work with protocols described in it. The final product of this objective is a library that allows to intuitively feed it a BSPL protocol and receive a structure representing the protocol. As explained in chapter 4c, the chosen language for the project is Go. This means that the resulting product must be a Go package that parses raw-text and outputs a Go type defining a protocol. Concepts such as instance and reasoner are explained in section 4.1.

Table 5.1: Requirements of objective O1

Requirement	
R1 There must be Go types defining each component of a BSPL protocol.	
Specification	Description
S1	Types representing main and atomic BSPL components must be exported.

5. Technical Objectives and Requirements

S2	Types related to instances must be declared as interfaces but not implemented in this package.
S2	Types related to reasoners must be declared as interfaces but not implemented in this package.
Requirement	
R2 Functions for protocol serialization and parsing must be provided.	
Specification	Description
S1	The parsing functionality must be exposed a single function, as a “black box”.
Requirement	
R2 The instance interfaces must include (des)serialisation methods aside of the functional methods.	

Objective 2

The second objective is to provide the implementation of BSPL protocol instances that were declared as interfaces in **O1**. Developing the rest of the components using the interfaces makes it possible to transparently if any changes are made in the future. For this, the result of this project will be a package that implements all the instance interfaces and provides as little package-dependant code exposition as possible. This is to say, the exported code of this library that is not strictly implementing instance interfaces must be minimized.

Table 5.2: Requirements of objective O2

Requirement
R1 Every interface declared in O1 but the reasoner must be implemented.
R2 The implementation must be information based. Actions are derived from changes in the instance information.

Objective 3

O3 refers to the networking components of the project. The result must be a library that provides ways to discover other nodes, exchange service information, instantiate protocols, update instances and belong to private networks.

Table 5.3: Requirements of objective O3

Requirement
R1 There must be a type representing a node of the network.
R2 Each node must have a BSPL reasoner as an attribute. The reasoner is implemented by each network user but the interface is defined O1 .
R3 Nodes must be able to discover other nodes on the network.
R4 Nodes must be able to exchange the offered services with other nodes.

R5 Nodes must be able to engage (instance protocols) with other nodes.	
Specification	Description
S1	The node that instantiates the protocol must do so according to the correctness of the protocol.
S2	The node that receives the instanced protocol must check the validity of the role asignation.
R6 Nodes must be able to update instances with other nodes.	
R7 Nodes must be able to create private networks with other nodes.	

Objective 4

According to objective 4 a demonstration must be created for NaHS using **O1**, **O2** and **O3**. The demo must include multiple, different agents that consume and offer different services, and act in their own interest.

Table 5.4: Requirements of objective O4

Requirement
R1 There must be at least four different agents.
R2 There must be at least five different protocols (five services offered between four agents).
R2 Each agent type must implement a BSPL reasoner.

5.2 Non-Functional Objectives

Objective 5

The fifth objective is straightforward, packages must compile without warnings or errors. This includes pre-compiling steps, such as writing style verification and documentation generation from comments.

Table 5.5: Requirements of objective O5

Requirement
R1 Style must correspond to <code>gofmt</code> .
R2 No warnings must be raised when compiling.
R3 No errors must be raised when compiling.

Objective 6

Objective 6 tries to ensure that the most relevant of the code are fully covered by tests. Go provides a native testing tool that, as specified by requirements **O6R2** and **O6R4**, will be the tool used in this project.

Table 5.6: Requirements of objective O6

Requirement
R1 Every non-utility function must be tested, considering utility functions as the ones that simplify processes but don't implement relevant processes.
R2 Tests must be carried out with the <code>go test</code> tool.
R3 Tests must be carried out locally and whenever new commits are pushed to the repository using CI tools.
R4 Coverage reported with the <code>go test</code> tool must be visible from the git repository.

Objective 7

Go packages usually have multiple sub-packages. The go syntax implies that each of the sub-packages must be manually imported for their corresponding types and functions to be accessible. However, Go supports type aliasing, which when combined with function wrapping allows exposing any sub-package types and methods from the main package.

Table 5.7: Requirements of objective O7

Requirement
R1 Each BSPL interface, protocol type and the global parsing function must be exposed from the main package.
R1 The NaHS network Node type must be exposed from the main package.

Objective 8

O8 is ambiguous. Type, function and variable names must be descriptive. Comments must be used to explain the functionality and flow of the code. `gofmt` standards must be followed. The intention of objective 8 is to make it as easy as possible for someone new to the library to understand the code while reading it. This, combined with other factors such as documentation, is vital for maintainability.

Table 5.8: Requirements of objective O8

Requirement
R1 Descriptive names must be used for types, functions and variables.
R2 Comments must be used to explain relevant sections of the code.
R3 Code must be formatted using <code>gofmt</code> .

Objective 9

Documentation, both in the form of this document and in the form of generable documentation from the comments of the source code must be provided with the project.

Table 5.9: Requirements of objective O9

Requirement
R1 The documenting comments in the source code must be able to generate documentation with the <code>godoc</code> tool.

Objective 10

O10 may look irrelevant but is of great importance for the reasons described in section 2.2.1.

Table 5.10: Requirements of objective O10

Requirement
R1 Only dependencies compatible with the Mozilla Public License [20] must be used.
R1 Compatibility between MLP and the dependency licenses must be verified.

5.3 Additional Specifications

This section details the stakeholders, basic use-cases, restrictions and design specifications of the project. Some specifications, such as the use-cases of the demo are described in their corresponding development sections.

5.3.1 Stakeholders

The project has a general purpose that could be adopted by anyone for any type of application, which makes stakeholders impossible to define. The demo scenario, however, has stakeholders represented by each of the agents implemented on it. The demo is explained in section 8.3.

5.3.2 Use Cases

The use cases of the project are determined by what an agent is natively able to do. All use cases involve two agent actors, except for “Parse protocols” that involves only one agent.

- **Discover other agents.** This use case requires another one: for one agent to be discoverable it must be able to announce itself. Nodes must be able to receive announcements from other nodes, even if it is by polling.

5. Technical Objectives and Requirements

1. Connect to a rendezvous node.
 2. Connect request and receive data from the rendezvous node.
- **Announce an agent.** Each node will be able to tell other nodes it has joined the network and offers services described with certain protocols.
 1. Connect to a rendezvous node.
 2. Connect send contact data to the rendezvous node.
 - **Create private networks.** Two agents that share a key can avoid communicating with nodes that don't share the same key.
 1. Create a key.
 2. Use the key when initializing the node.
 - **Enact protocols.** This use case is the combination of two use cases: understanding and communicating protocols.
 - **Parse protocols.** An agent must be able to understand, at least on a syntactical level, protocols described with BSPL.
 1. Create a token table from a BSPL protocol.
 2. Convert the tokens into Go structures.
 - **Send protocol instance updates.** When a protocol enactment is updated, the new information is sent to the other agent.
 1. Open a connection to another agent.
 2. Serialize the message.
 3. Send the message.
 - **Send protocol instance updates.** When another agent updates an enactment, it sends the new information.
 1. Accept a connection from another agent.
 2. Unmarshal the message.
 3. Apply the new information.

5.3.3 Restrictions

The project presents the following restrictions:

- Internet connectivity to contact the rendezvous nodes and any nodes outside of the system an agent is hosted at.
- Go version equal or greater than 1.14.4. Earlier versions might work, but it is not guaranteed.
- External libraries. The specific versions of all the dependencies are listed in the `go.mod` file of each repository.
- The project provides examples of agent implementation, but any other methods might be used for this purpose.

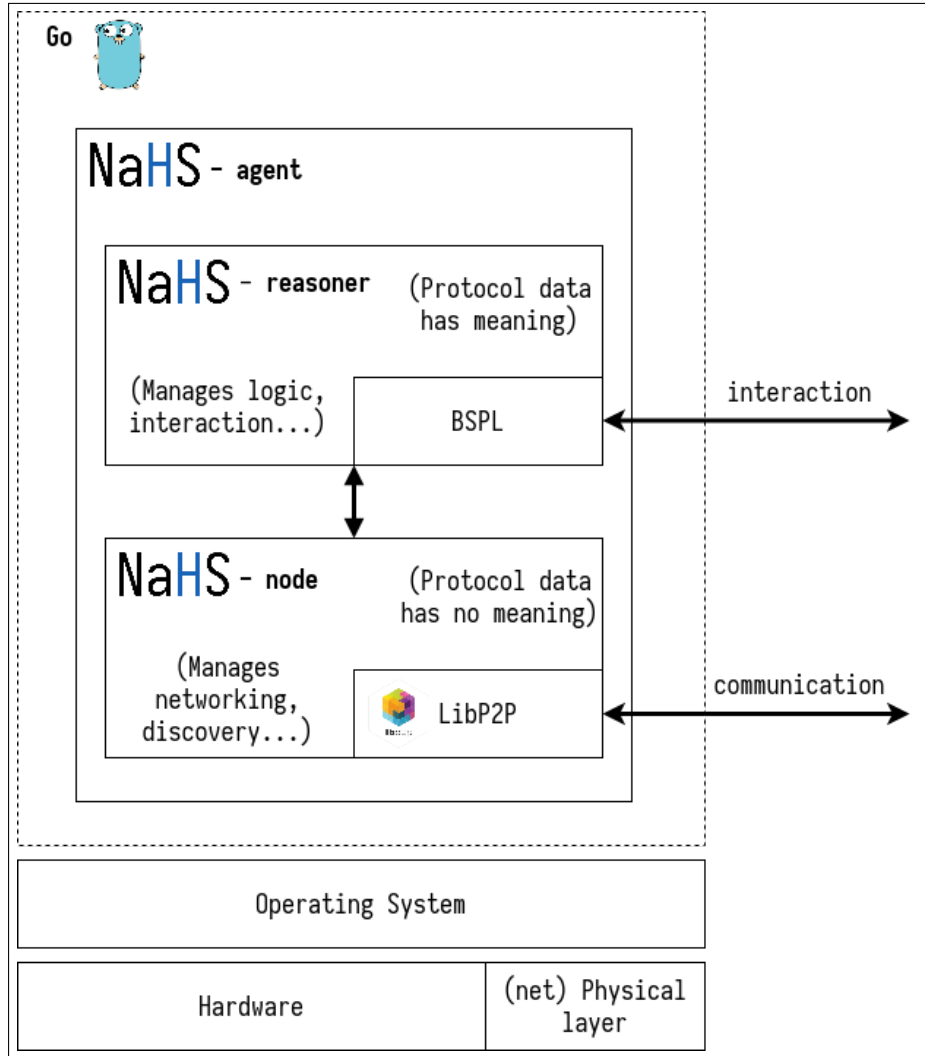


Fig. 5.1: NaHS agent architectural overview

5.3.4 Design Specifications

Figure 5.1 shows the architectural overview of a single NaHS agent. It shows how the reasoner and networking node combine to form the agent, that the reasoner uses BSPL for interaction and the node uses LibP2P for communication, and that all is compiled into a Go program. Each one of these components is further explained in the chapters 4 and 8. Go has no classes or objects, but alternatives to class diagrams are provided in the development section of each module in chapter 8.

Chapter 6

Budget and Planning

6.1 Budget

The human resource budget needs to take into account the job profiles involved in the project and the amount of hours worked by each profile. There are two people involved in the project: the project director and the student. The project director is expected to invest 60 hours, while the student is expected to invest around 300 hours. The schedule of section 6.2, however, indicates an estimate of 388 hours. The research and development stages require however different profiles, this the budget shown in figure 6.1 shows the estimated budget for the project with work profiles and hour distribution broken down in accordance with the project schedule.

- **Researcher.** Research of communication protocols and peer-to-peer networking falls under this role.
- **Protocol engineer.** Designer and developer of the Blindingly Simple Protocol Language lexical analyzer, parser and implementation.
- **Networking engineer.** Designer and developer of the NaHS peer-to-peer networking components.
- **Test designer.** Designer of tests for the libraries developed by the protocol and networking engineers.

Table 6.1: Cost of human resources

Profile	€/hour	Hours	Total cost [€]
Project Director	45	60	2,700
Researcher	36	184	6,624
Protocol engineer	40	96	3,840
Networking engineer	38	76	2,888
Test designer	36	32	1,152
Total cost:			17,204

The schedule of section 6.2 is made with Microsoft Project. An approximate cost of a Project license for four months could be amounted to 120€, therefore the final cost would be 17,324€.

6.2 Planning

Meetings with the project director were organized weekly to report the state of the project, evaluate the results and plan the next actions. While initially meetings took place on the University of Deusto, the *COVID-19* outbreak resulted in remote meetings for the majority of the project.

Figure 6.1 contains a Gantt diagram with the expected schedule of the project. The schedule was planned around the project structure defined in section 2.4. The first stage is intended for research and evaluation of current technologies that will later be used in the project. Research is followed by the development of a BSPL parser and BSPL instances and after that come the networking components of NaHS. The last stages involve creating a demo for the libraries and documenting the process. Each of the phases will be explained next, with each point representing a milestone.

- **Initial research.** Expected start and finish dates: *from 2020-02-05 to 2020-02-17*. This initial milestone will be achieved after researching the most relevant fields for this project, mainly autonomous services, peer-to-peer networks and protocol description languages. These tasks are vital as the correct development of the next stages depends heavily on the knowledge acquired in this phase. *Result: definitive roadmap of the next phases.*
- **BSPL development.** Expected start and finish dates: *from 2020-02-18 to 2020-03-26*. This milestone marks the completion of all BSPL-related development, from initial design to final testing and preparations. It is divided in other two main milestones: the development of the BSPL parser and the development of BSPL instances. Each one of these sub-phases is composed of several tasks, as can be seen in figure 6.1.
 - **Parser development.** Expected start and finish dates: *from 2020-02-18 to 2020-03-10*. This sub-milestone marks the completion of the BSPL parser. Development is considered complete when the result is tested and validated one last time after designing and writing the source code. This means the project will be able to take a plain-text protocol and create the correct Go structures that represent it, as well as functions to comply with every one of the related requirements and specifications defined in chapter 5. *Result: functional, validated BSPL parser.*
 - **Protocol implementation.** Expected start and finish dates: *from 2020-03-11 to 2020-03-23*. BSPL protocols need to be enacted to be really useful, this milestone will be achieved when the BSPL enactments are designed, implemented, tested and validated. The networking components can't be fully developed without a functional enactment or at least an interface definition, this phase and all its tasks need to be completed. *Result: enactable BSPL protocols.*

After development is done, the library should be prepared to be imported from other projects by creating top-level type aliases and function wrappers.

- **Networking.** Expected start and finish dates: *from 2020-03-27 to 2020-04-28*. When networking is completed, nodes of the NaHS network must be able to join or leave the network, discover other nodes and interact with other nodes via protocol enactments. The phase before reaching the milestone includes multiple tasks, from designing the network to testing it. The sub-milestones listed below contain more details. *Result: a peer-to-peer network where nodes discover each other and each other's BSPL protocols, and interact by enacting protocols.*
 - **Node design.** Expected start and finish dates: *from 2020-03-27 to 2020-04-03*. This phase involves the low-level interactions between NaHS nodes, from BSPL-related ones to peer discovery. LibP2P needs to be well examined and its use in the node design must be planned. The node is the atomic networking component of NaHS, and defines most of its functionality. *Result: NaHS node specification.*
 - **Development.** Expected start and finish dates: *from 2020-04-06 to 2020-04-23*. The development involves implementing and perfecting the previous design before testing and validating it. The initial functionalities will be developed locally but will later be tested in an open environment (not all nodes are local). Once the networking functionalities are completed, the milestone is reached. *Result: functional, validated networking components.*
- **Demonstration.** Expected start and finish dates: *from 2020-04-29 to 2020-05-18*. With protocol and networking features implemented, the integration needs to be tested in scenarios that emulate a real system. The demonstration milestone contains two sub-milestones, explained below. *Result: Demonstration of behavior of NaHS nodes.*
 - **Simple demonstration.** Expected start and finish dates: *from 2020-04-29 to 2020-05-04*. This milestone marks the completion of a scenario where two nodes enact a single protocol to validate the library integration. The scenario needs to be designed, implemented and run. *Result: Demonstration of one protocol enactment between two nodes.*
 - **Complex demonstration.** Expected start and finish dates: *from 2020-04-05 to 2020-05-18*. Once the simple demonstration is completed, new agents and protocols will be added to the environment to observe their behavior when running the demo. Having multiple agents interact means it will likely be a more demanding task than the simple demo. *Result: Demonstration of multiple protocols being enacted between multiple different, heterogeneous nodes.*
- **Documentation.** Expected start and finish dates: *from 2020-05-19 to 2020-06-17*. The documentation milestone will be reached when the project documentation is complete, having been checked and improved several times as well as verified by the project director. *Result: deliverable documentation.*

There is a final milestone, “Project closure”, which involves delivering documents and publishing repositories. Tasks have been divided so that none of them take longer than two weeks, with the exception of the final documentation. This makes it easier to adapt to unforeseen changes on the schedule, as changing tasks that take a lot of time is harder than changing smaller, more specific

6. Budget and Planning

tasks. Milestones, on the other hand, can take longer as they are composed of tasks that need to be completed to reach a milestone.

Although the schedule shows a total amount of 97 days, it takes into account that external factors, such as professional workload or academically demanding periods, as a way of risk management. All tasks are ordered linearly to make the schedule easier to stick to and adjust, but some of the tasks may be run in parallel. For example, the development and testing of the libraries are complementary, intertwined actions run on parallel. The documentation may also be advanced as the project progresses instead of being written entirely at the end of the project. Each day represents four hours of work, the approximated time corresponding to the project. The structure is the same as the one shown in section 2.4 but gives a more detailed insight to each of the planned steps.

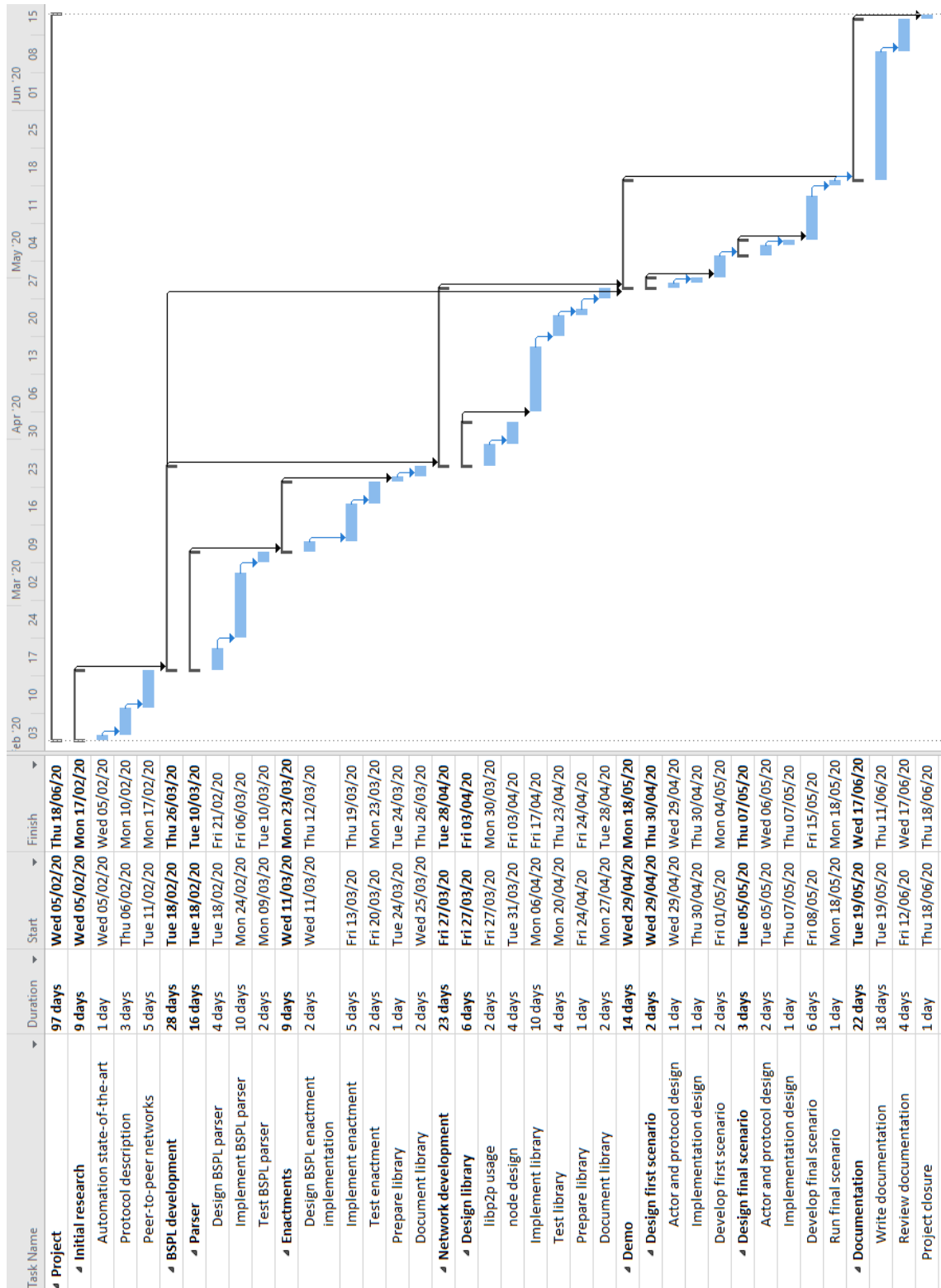


Fig. 6.1: Project schedule

Chapter 7

Methodology and Resources

7.1 Methodology

The project was developed using the Kanban methodology. Its origins are Japanese, from tracking the life cycle of a product throughout all the fabrication process. Similar to that, Kanban tracks the state of each task from their creation to their completion in a project. It is usually done in a physical board, composed of columns. Columns are ordered, and each column can hold any number of cards. Cards represent tasks and can be moved from their current column to adjacent ones.

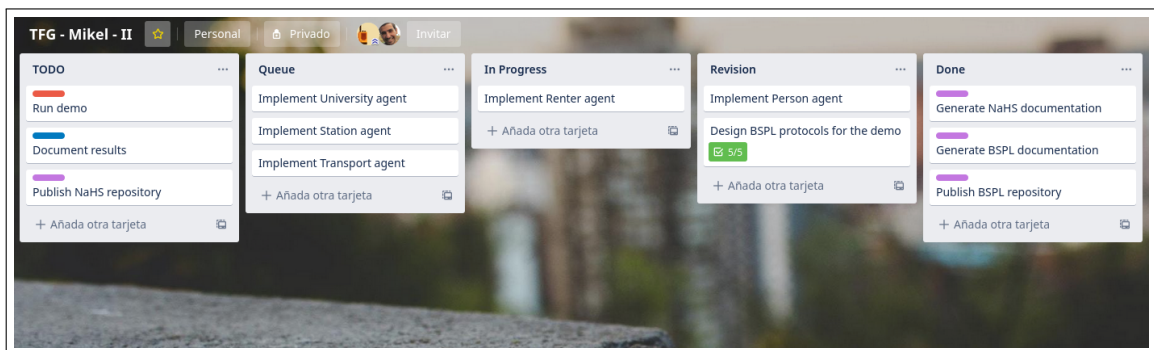


Fig. 7.1: Kanban board

A digital board was used in this project, shown in figure 7.1 and provided by Trello. Trello is a free (in this case) platform to plan projects, either individually or collaboratively. It has many features, the most noteworthy ones being collaboration tools, integration with other services such as BitBucket or GitHub to manage commits, issues... There were five columns in the Kanban board of this project. When a task was created, its card was added to the first column. When a task was finished, its card reached the final column and was archived.

- **TODO.** Tasks that have not been started and will not be started soon.

- **Queue.** Tasks that have not been started but will soon be.
- **In Progress.** Tasks being currently developed.
- **Revision.** Completed tasks that need verification.
- **Done.** Completed, verified tasks.

This set of columns allows for flexible management of tasks, easy tracking and an intuitive way of knowing what has been done, what is being done and what will be done soon. Most tasks follow a linear path from **TODO** to **Done**, but it is also common for tasks to repeatedly move between **In Progress** and **Revision** after design choices, other code or project requirements change. This is an example of the agility Kanban boards provide to project organization.

7.2 Resources

The author and the director are the only human resources involved in this project. In a professional project there should have been more, as the different roles played by them (researcher, developer, test engineer...) should ideally be played by a specialized professional. The same can be said for physical resources: a laptop and a desktop computer are all that is required. No heavy processing power is required for this project: the more demanding parts are tests as they try to push the libraries. Remote physical resources, such as machines to run CI tests, are counted as digital resources. Digital resources are abundant. Academic articles, web pages, software libraries, documentation... All available at no cost for the researcher. The project was developed on a GNU/Linux system using open source editors and tools. The effort put by Protocol Labs towards documenting P2P and their contributions are deeply appreciated.

7.3 Development Methodology

This section describes some general methods with which the software was developed. Version control tools were used to keep track of changes and unit tests were continuously made to the code while it was being developed.

7.3.1 Version Control

Git was used for the BSPL, NaHS and demo version control. Git is a VC system that tracks changes of a directory, originally created by Linus Torvalds in 2005. The inner workings of Git deserve its own section, [21] gives very detailed information about it. In short, git creates different objects for file contents and links them together in a way that makes it easy to reconstruct the project at any point in its version history with data structures, namely Merkle trees. Local versions of the repository (or project directory) in the local devices of developers are saved to and compared against the remote, central version of the repository, which contains the “official” version of the repository.

The git provider used in this system is GitHub. It is a company that offers free repository hosting. Its popularity has made it the default go-to for many developers, resulting on a very rich ecosystem of projects being hosted on it. It offers many services not limited to repository hosting, usually oriented to enhancing workflows of said repositories. Changes pull requests, issues... All can be intuitively examined and commented, giving individuals and teams a lot of facilities to develop a project. GitLab is also a very competent alternative that even has more arguments in its favor than GitHub. It has a lot of native functionalities such as CI tools, free private repositories, and has a community-driven, free sourced community version [22]. The results of this project published on GitHub will be mirrored in GitLab for increased availability. In both GitHub and GitLab, repositories are presented with README files when viewed with web browsers. README files usually contain the most relevant information someone examining the project should have, offering quick explanations and pointing to more in-depth information sources. Figure 7.2 shows a screenshot of the header of the README file for the BSPL repository. Information about repository organization and library usage is provided after the header.

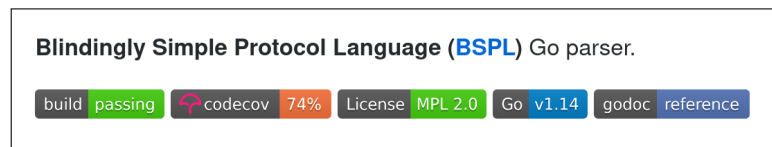


Fig. 7.2: Screenshot of the header section of the BSPL README.md file

Mercurial could have been a great choice for this project too. Mercurial is a VC system, arguably more simple to use than Git and this producing cleaner, easier to examine documentation. Despite its possible shortcomings in branch management, this project was developed by a single person, with a very linear approach to development that did not require extensive branch management. There are many free platforms that provide mercurial support, including BitBucket which is owned by Atlassian, the same company that owns Trello.

7.3.2 Testing

The methodology was the same for all of the modules. In the cases of interface and structure design, techniques similar to test-driven development were used. A pseudo-test was created to define how the elements should behave. Then real code and tests were written and run. Despite this, as it was not strictly TDD, the coverage is not as high as if it were.

As a quality assurance measure, the editor was configured to run code formatting, tests and report coverage on save. Code formatters ensure syntax consistency in every file. The testing and coverage results can be seen in figure 7.3. The coverage is shown in the editor as shown in figure 7.4, green meaning the line was run in the tests and red meaning it wasn't.

To prepare the project for collaborative development, the tests are also run with a CI tool when pushing new commits. The success of the tests and coverage improvement are considered before accepting a new commit. This improves traceability, ensures every change is tested and reported even if new developers don't do it locally.

```

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE
Running tool: /usr/bin/go test -timeout 5s -coverprofile=/tmp/vscode-goc7C1sF/go-code-cover github.com/mikelsr/bspl/implementation -v

== RUN TestIntance_Key
--- PASS: TestIntance_Key (0.00s)
== RUN TestInstance_Equals
--- PASS: TestInstance_Equals (0.00s)
== RUN TestInstance_Diff
--- PASS: TestInstance_Diff (0.00s)
== RUN TestInstace_Update
--- PASS: TestInstace_Update (0.00s)
== RUN TestMarshalAction
--- PASS: TestMarshalAction (0.00s)
== RUN TestUnmarshalAction
--- PASS: TestUnmarshalAction (0.01s)
== RUN TestInstance_MarshalAndUnmarshal
--- PASS: TestInstance_MarshalAndUnmarshal (0.00s)
PASS
coverage: 80.0% of statements
ok github.com/mikelsr/bspl/implementation 0.009s coverage: 80.0% of statements
    
```

Fig. 7.3: Output shown in the editor after automatic tests

```

146 func (i *Instance) Update(j reason.Instance) error {
147     _, values, err := i.Diff(j)
148     if err != nil {
149         return err
150     }
151     // Set parameter value
152     for k, v := range values {
153         i.SetValue(k, v)
154     }
155     return nil
156 }
    
```

Fig. 7.4: Coverage shown in the editor after automatic tests

All checks have passed
3 successful checks




✓		Travis CI - Branch — Build Passed	Details
✓		codecov/patch — Coverage not affected when comp...	Details
✓		codecov/project — 74.49% (+0.00%) compared to 3...	Details

Fig. 7.5: Report of CI tools after a successful commit

Tests in Go are different to some other languages. Test files are stored inside the tested module: they are in the same directory as the files being tested. The test-file of `proto/action.go` is `proto/action_test.go`. The test functions are exported and take a reference to `testing.T` as a parameter. `testing.T` is defined by the Go documentation [23] as “a type passed to Test functions to manage test state and support formatted test logs”. To put it into context, a test that runs the function `ok()` looks like the listing 7.1.

```

1 func TestOk(t *testing.T) {
2     if !Ok() {
3         t.Fail()
4     }
5 }

```

Listing 7.1: Test example

Set-ups and tear-downs are handled differently too. A `TestMain` function must be defined that takes a reference to `testing.M` as a parameter. The run function (`testing.M.Run()`) runs the tests, therefore the set-up code must be called before the run function and the tear-down code after it. An example is shown in listing 7.2.

```

1 func TestMain(m *testing.M) {
2     // setup
3     m.Run()
4     // teardown
5 }

```

Listing 7.2: TestMain example

7.3.3 Dependencies

Module dependencies are handled with `go mod`, the default dependency management tool since Go version 1.11. Module version and dependencies are stored in the `go.mod` file (shown in listing 7.3), checksums of the dependencies are stored and kept track of in the `go.sum` file (shown in listing 7.4).

```

1 module github.com/mikelsr/bspl
2
3 go 1.14
4
5 require bitbucket.org/mikelsr/gauzaez v1.0.0

```

Listing 7.3: Contents of `bspl/go.mod`

```

1 bitbucket.org/mikelsr/gauzaez v1.0.0 h1:N1qsZ0Tm8Ao1WHrvk9AyZMwX5RP5wpUIQqXBrnA4H9E
  =
2 bitbucket.org/mikelsr/gauzaez v1.0.0/go.mod h1:uRxDJYAEn0imoBIeSXoHcJ/
  hBQ1zNZDoigJvTctq7KM=
3 github.com/mattn/go-runewidth v0.0.7/go.mod h1:H031xJmbD/WCDINGzjvQ9THkh0rPKHF+
  m2gUSrubnMI=
4 github.com/mattn/go-runewidth v0.0.9 h1:Lm995f3rfdpd6TsmuVCHVb/QhupuX1Yr8sCI/QdE
  +0=
5 github.com/mattn/go-runewidth v0.0.9/go.mod h1:H031xJmbD/WCDINGzjvQ9THkh0rPKHF+
  m2gUSrubnMI=

```

```

6 github.com/olekukonko/tablewriter v0.0.4 h1:vHD/YYe1Wolo78koG299f7V/VAS08c6IpCLn+
  Ejf/w8=
7 github.com/olekukonko/tablewriter v0.0.4/go.mod h1:zq6Qw10f5S1nkVbMSr5EoBv3636FWnp+
  qbPhuo021uA=

```

Listing 7.4: Contents of bspl/go.sum

During most of the development the repositories containing the libraries developed in this thesis were not publicly available. To keep this from being an issue when using these libraries as dependencies some configuration is needed. To use private repositories as dependencies in other Go projects, the `GOPRIVATE` environment variable must be set. The `GOPRIVATE` environment variable tells Go that modules in that path are not publicly available. This way it avoids using the default Go proxy and fetches the module from the original host. All the libraries of this thesis are hosted in GitHub, therefore, the value of `GOPRIVATE` is set to the following in the `.profile` file.

```

1 export GOPRIVATE='github.com/mikelsr/*'

```

Listing 7.5: Partial contents of .profile

The authentication method being used for GitHub, in this case, is SSH. To configure git to use SSH instead of the default HTTPS for the repositories in question the `.gitconfig` file must be edited to add the rule.

```

1 [url "git@github.com:"]
2     insteadOf = https://github.com/

```

Listing 7.6: Partial contents of .gitconfig

Chapter 8

Development

This chapter documents the development section and gives details into how components were developed and why design choices were made. Before going in depth into each module, a general overview will be provided to give them context. In the example of the company that wanted to obtain new paper, the agent representing the company will first join a network with more agents. In this network, the company will be able to announce itself and look for any services other agents might offer, including paper provision. This will be done with the networking library, explained in section 8.2. After finding a paper provider, the provider and the company will interact by following a protocol described by the provider with BSPL. The protocol will be enacted by the two agents, and if the enactment is correct (follows the protocol description) the company will have been provided paper. How protocols are described and enacted is described in section 8.1.

8.1 BSPL Module

The development of the Blindly Simple Protocol Language module is organized in four sub-modules described in table 8.1.

Table 8.1: BSPL submodules

proto	Go structures to form a BSPL protocol.
parser	Standalone BSPL parser implemented using a lexer written while taking the Computability and Complexity course.
reason	Interface definition for implementing a reasoner and protocol instances.
implementation	Draft implementation to use in another project.

Before proceeding any further, the objective of each of the packages should be briefly described. The goal of the `proto` package is to represent protocols described with BSPL in Go. For that, types and structures need to be created. The objective of `parser`, on the other hand, is to take a BSPL raw-text file and convert it to the types of the `proto` package. The sub-package `reason` defines how protocol enactment should be with interfaces and the `implementation` package implements the enactments.

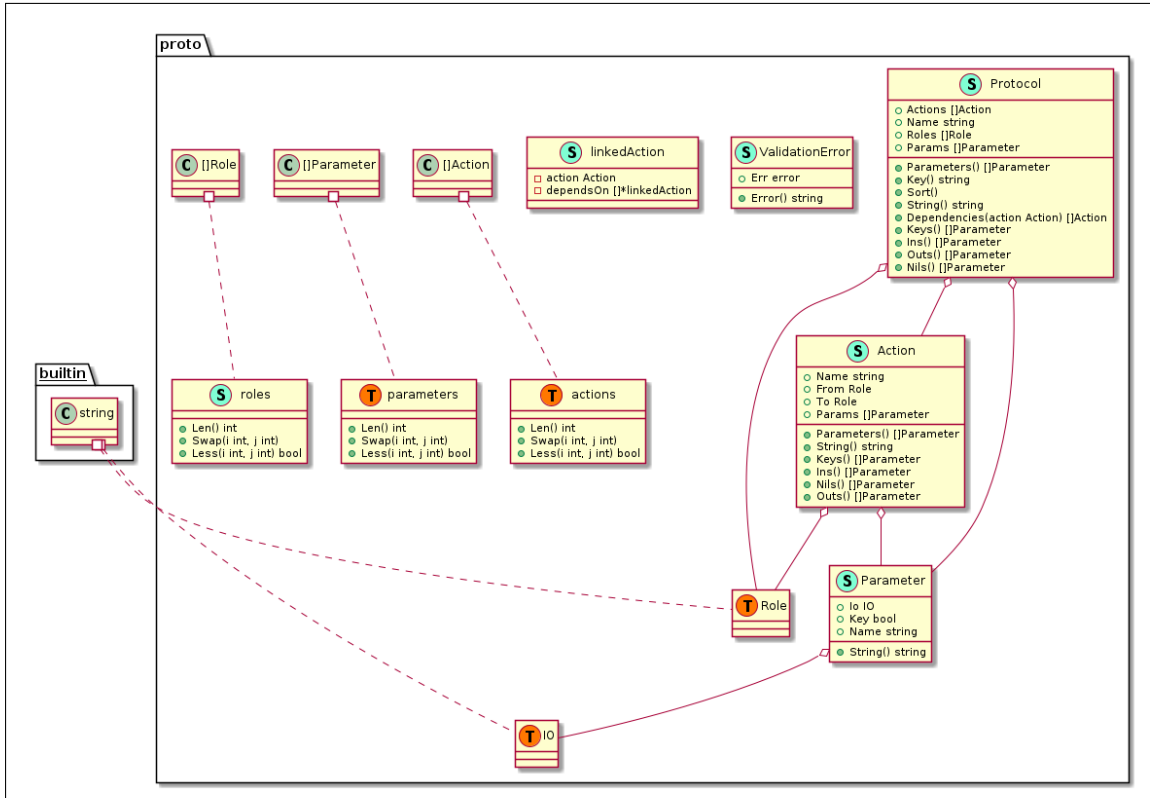


Fig. 8.1: UML diagram of bspl.proto

8.1.1 Sub-modules

This section contains a detailed view of each of the sub-modules in the BSPL module.

8.1.1.1 proto

This packages contains the definition of types and structures that represent a BSPL protocol in the Go programming language.

Table 8.2: BSPL Go types

Protocol	Structure defining a Go protocol. It has a name, roles, parameters and actions.
Role	Roles of a protocol. It is an alias for the <code>string</code> type. Each role identifies the expected behavior of a participant in a protocol.
Parameter	Structure defining a Parameter of a BSPL protocol or action. It has a scope, a name and a key attribute.
IO	<code>string</code> alias defining the scope of a parameter. It can only take one of the following values: <code>In</code> , <code>Nil</code> or <code>Out</code> .
Action	Structure defining an action of a BSPL protocol. It has source and target role, a name and a set of parameters.

The generated UML diagram from the sub-module is shown in figure 8.1.

The package also contains some functions related to these types. The most noteworthy are the validation and sorting functions.

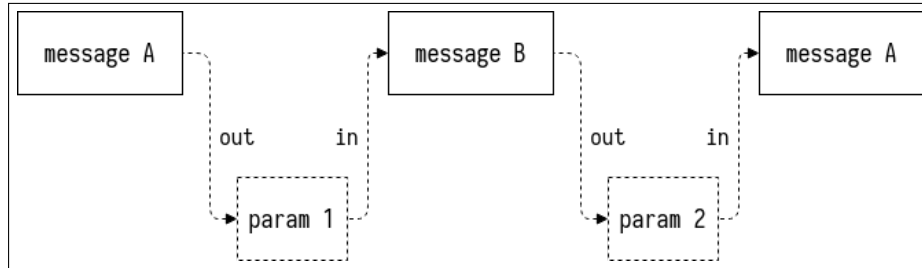


Fig. 8.2: Circular parameters between BSPL messages

The validation functions have multiple uses but are mainly used by the parser to verify BSPL protocols. The order of BSPL actions is inferred from their parameters as explained in section 4.1. To check that the order can be inferred, the `Validate(p Protocol) error` function creates a linked list of actions and parameters and checks for circular dependencies. If none are found, it will return a `nil` value instead of an error. This process equivalent to checking the structure of a tree data structure. Parameter verification is vital and needs to take into account message branching. Figure 8.2 shows an example similar to the one on figure 4.1 but with circular dependencies.

The sorting functions of actions, parameters and roles make it trivial to compare them by first sorting them and then comparing the output of their `String() string` methods. The most convenient to implement the sort functions is to make each of the types to be sorted implement the `sort.Interface`, which is an interface defined in the respective documentation [24] as shown in listing 8.2. It is worth noting the types to be sorted are not actions, parameters nor roles but slices of them. For example, it is not `Parameter` but `[]Parameter`. For that, three local types have been created:

- `actions = []Action,`
- `parameters = []Parameter,`
- `roles = []Role.`

These three local types implement `sort.Interface` and are transparently used from the exported sort functions. To provide an example, to sort parameters the `SortParameters` function is called, avoiding the explicit use of local types as shown in listing 8.1.

```

1 func SortParameters(params []Parameter) {
2     // cast params to the parameters type pass it to sort.Sort.
3     // Go manages slices with references instead of values, the
4     // reference to the original slice now points to the sorted one.
5     sort.Sort(parameters(params))
6 }
    
```

Listing 8.1: Definition of SortParameters

```

1 type Interface interface {
2     // Len is the number of elements in the collection.
3     Len() int
4     // Less reports whether the element with
5     // index i should sort before the element with index j.
6     Less(i, j int) bool
7     // Swap swaps the elements with indexes i and j.
    
```

```

8 |   Swap(i, j int)
9 | }

```

Listing 8.2: Definition of sort.Interface

8.1.1.2 parser

Figure 8.3 shows an overview of the parser, from the raw-text received as input to the final representation of the package in Go. The protocol goes through three stages when being parsed: first it is described as text, then as tokens extracted from the text and finally as a Go structure. The next sections will go into more detail about each of the steps. The generated UML diagram from the parser sub-module is shown in figure 8.4 later in this section.

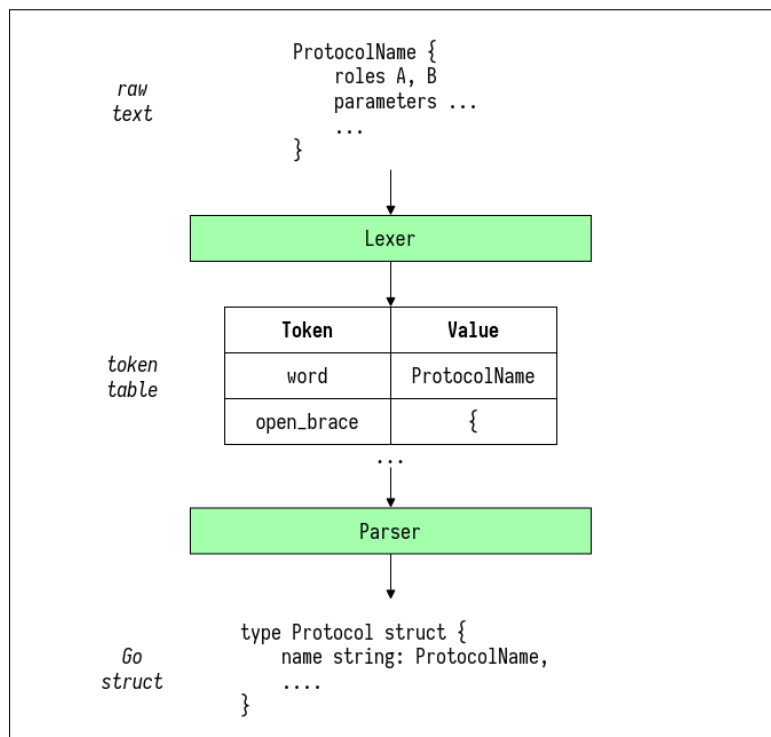


Fig. 8.3: Overview of the BSPL parser

Lexical Analyzer

The objective of the parser sub-module is to provide a way to read a raw-text protocol described BSPL and convert it to a Protocol structure from the proto package.

The first step is to pass the text through a lexical analyzer, or lexer, to create a token table. There are multiple ways of doing this, the chosen one is a tool written in Go while taking the Computability and Complexity course that saves a lot of work. It is a configurable lexer that takes a deterministic finite automaton described in a JSON file and runs the source text through it to generate a token table. A more detailed explanation can be found at the article of the lexer [25].

These automatons contain nodes and transitions. Nodes can be final or not final. A transition has a source and a destination. To go from the source to the destination, a condition has to be met (a character has to meet a regular expression in this case). In the JSON file, nodes have a boolean attribute describing whether the node is final, and paths each with a regular expression (transition condition) and a destination (another node). The full JSON describing the finite automaton can be found at the BSPL repository, in the `config/lexer.json` file.

```

1 {
2   "tokens": [
3     "arrow",
4     "close_brace",
5     "close_bracket",
6     "colon",
7     "comma",
8     "newline",
9     "open_brace",
10    "open_bracket",
11    "whitespace",
12    "word"
13  ],
14  "nodes": {
15    "q0": {
16      "final": false,
17      "paths": {
18        "[A-Za-z_]+$": "q1",
19        "[ |\\t]+$": "q2",
20        "\\n$": "q3",
21        "\\{": "q4",
22        "\\}": "q5",
23        "\\[$": "q6",
24        "\\]": "q7",
25        ":$": "q8",
26        ",$": "q9",
27        "\\-$": "q10"
28      }
29    },
30    "q1": {
31      "final": true,
32      "token": "word",
33      "paths": {
34        "[A-Za-z_]+$": "q1"
35      }
36    },
37    ...
38  }
39 }

```

Listing 8.3: JSON describing a deterministic finite automaton to process a BSPL protocol

The lexer outputs a token table described in the listing 8.4. `Lines`, `LinePosI` and `LinePosE` are useful for error reporting but the parser only strictly requires `Tokens` and `Values`. `Tokens` contains the type of token, e.g. `word`, and `Values` contains the characters it contains, e.g. `some_value`.

```

1 type TokenTable struct {
2   Tokens []Token // type of token
3   Values []string // value of the token

```

```

4 |   Lines    []uint    // line the token is located at
5 |   LinePosI []uint    // initial column of the token
6 |   LinePosE []uint    // end column of the token
7 | }

```

Listing 8.4: Token table generated by the lexical analyzer

Some lessons learnt about good code design from a talk by Dave Cheney [17] have been applied to both the lexer, the parser and the networking components of this project among others. The clearest example is applying the Interface Segregation Principle [26] to describe with interfaces the methods expected from parameters instead of the types, for example requiring an `io.Reader` interface instead of an `os.File` type.

Parser

Once the token table is complete, the only thing left is to parse it. The lexer extracted tokens given a set of syntax rules, but the tokens had no meaning. It is the job of the parser to find meaning on those tokens by verifying that they compose a valid BSPL protocol. There is a Python tool that can be used to verify the validity of BSPL protocols [27], however the protocols would still need to be converted to Go. To reduce keep the number of dependencies of the package to a minimum, the validation has been implemented on the parser instead of depending on external projects. To keep the parsing state a protocol builder (`ProtoBuilder`) structure has been created, with methods to parse each of the components of a protocol described in BSPL: name, roles, parameters and actions. While parsing these, the protocol builder validates the protocol. Some of the validations were described in section 8.1.1.1. Others, such as the validity of the syntax are implemented in this sub-module. The methodology of every parsing function is similar: consume tokens from the table and when a checkpoint is reached check the validity of the current state of the protocol being created. If the check fails, return the error through the call levels until the highest level caller handles it. The maintainability of this section of the code has room for improvement and can be considered for future work, changing the internals to perhaps compare the obtained token structure to the expected one or use some other parsing tool. It does however fulfill its function efficiently. Space complexity is also acceptable taking advantage of how the programming language handles slice references.

The process takes many steps and is perhaps too complicated to be used by someone unfamiliar with the library when importing it. To make it cleaner and easier, there is a function to wrap lexical analysis and the parsing without requiring initializing structures such as the protocol builder. The final function is in listing 8.5, abstracting the caller from explicitly calling the lexer, parser and builder.

```

1 | func Parse(in io.Reader) (proto.Protocol, error) {
2 |     tokens, err := LexStream(in)
3 |     if err != nil {
4 |         return proto.Protocol{}, err
5 |     }
6 |     stripped := Strip(*tokens)
7 |     b := new(ProtoBuilder)
8 |     if err := b.Parse(stripped.Tokens, stripped.Values); err != nil {

```

```

9     return b.Protocol(), err
10  }
11  return b.Protocol(), nil
12 }
    
```

Listing 8.5: Final parsing function

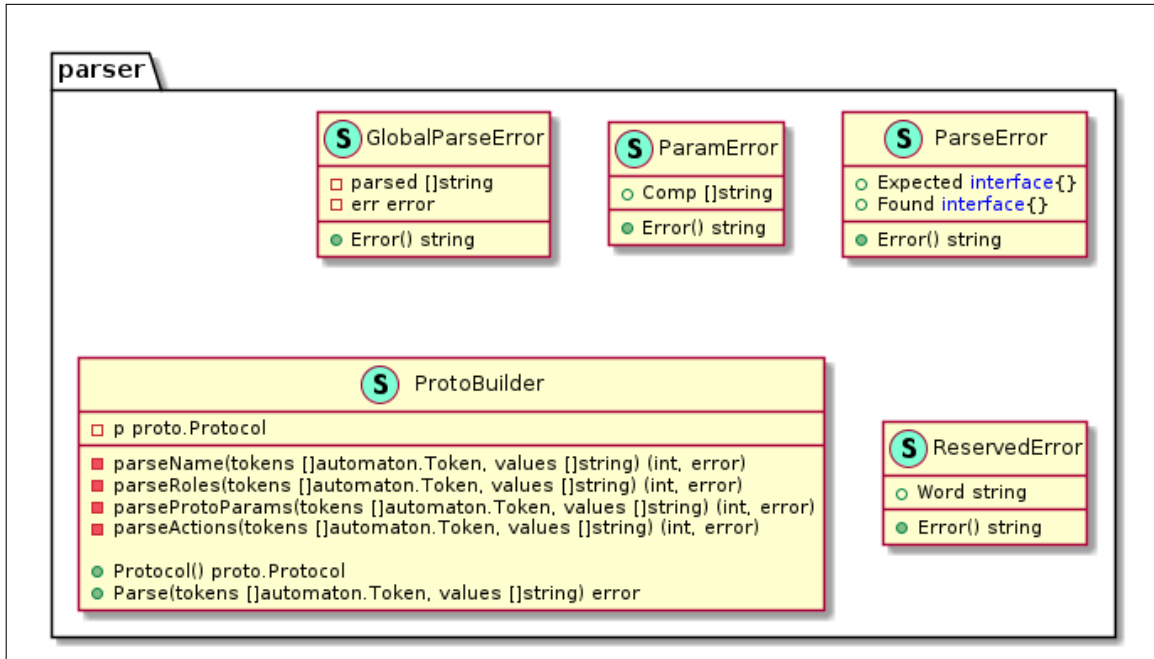


Fig. 8.4: UML diagram of bspl.parser

8.1.1.3 reason

The generated UML diagram from the sub-module is shown in figure 8.5.

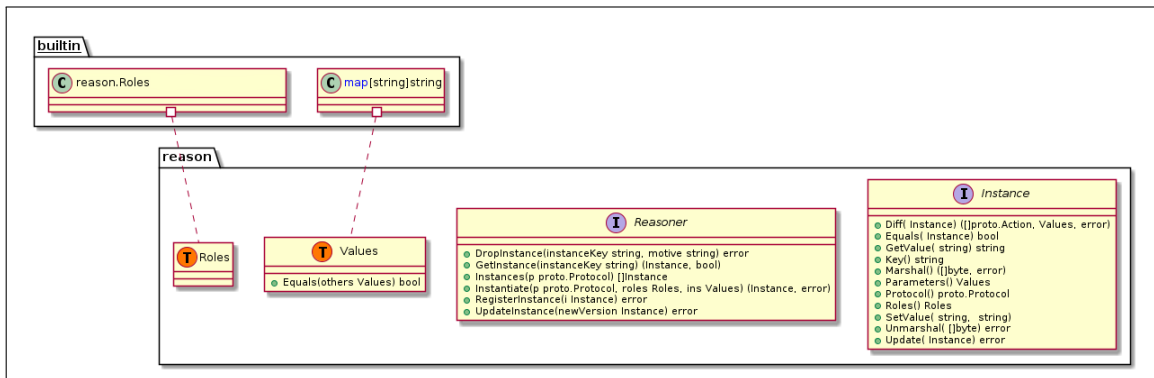


Fig. 8.5: UML diagram of bspl.reason

The reason sub-module contains the interfaces that define a BSPL reasoner and an instance of a protocol. There is a main reason to define the reasoner as an interface: it allows to define the logic of each of them while maintaining design consistency. Instances, however, were not such a clear

choice between interfaces and structures. The deciding factors were the ISP and that it would enable development of the NaHS module, which has the BSPL module as a dependency, without implementing it. This was important because the way to implement it was not clear at first and was reworked during development.

The reasoner interface will be implemented by the agents or some component of the agents in the NaHS network, but can also be implemented by any other service providers and consumers. The instance interface is implemented in the implementation sub-module of BSPL. That is the implementation used in this project but it is easily replaceable if a better one is created.

The following is a description of each one of the interfaces. The generated UML diagram shown in figure 8.5 contains both of them to complement this section.

The `Instance` interface defines an instance of a BSPL protocol between two nodes. Each node plays a role and runs some actions depending on the role they are playing, the previously run actions and the internal logic of their reasoner. Each action of an instance can only be created by the origin role, as explained in section 4.1. The order of the actions is derived from its parameters and in some cases, an action requires some other actions to have been run previously. During the lifespan of an instance, the parameters of the protocol it instantiates take values. These values are relevant for instance handling and for instance identification. The most relevant methods of the interface are described in table 8.3.

Table 8.3: Descriptions of some of the methods of the Instance interface

Diff	Diff identifies what action has been run between two versions of an instance.
Key	Generates the key or identifier of the instance, based on the protocol name, key parameters and their values.
Marshal	Serializes an instance into bytes.
Unmarshal	Creates an instance from marshalled bytes.
(Get Set)Value	Retrieve or assign the value of a parameter of the instantiated protocol.
Update	Update an instance given a future version of it. A future version of it is the same instance but with new actions having been run.

The `Reasoner` interface defines a BSPL reasoner. A reasoner enacts or instances protocols according to its internal logic. It keeps a local state of each instance and deduces the actions based on new states of them. The purpose of the reasoner is made clearer when considering every method has an instance either as an input or output parameter. The most relevant methods are described in table 8.4.

Table 8.4: Descriptions of some of the methods of the Reasoner interface

Instantiate	Creates an instance of a protocol. This belongs to the reasoner and not the instance interface because the initial values given to the instance depend on the reasoner.
RegisterInstance	Creates a new local state of an instance given as an input parameter.
UpdateInstance	Updates the local state of an instance given a future version of it.

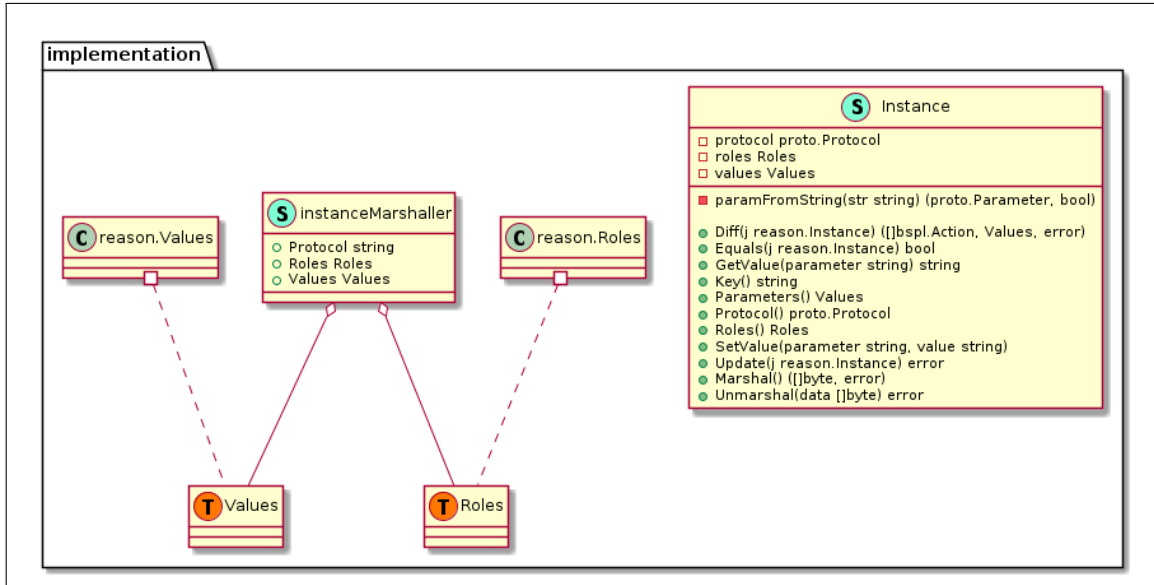


Fig. 8.6: UML diagram of bspl.implementation

8.1.1.4 implementation

The implementation module contains an example implementation of the `reason.Instance` interface. It does this by storing the values of the parameters with an internal map, using the `String()` method of each parameter as the key and the given value as the value. The marshalling and unmarshalling are done with the JSON format. Values of the JSON object are encoded in base64 to ensure that the contents of the instance don't negatively interfere with the marshal and unmarshal processes. The implementation initially contained message types that represented the instance of actions, but they were deleted due to the redundancy of actions being inferred from two states of an instance.

The generated UML diagram from the sub-module is shown in figure 8.6.

8.1.2 Presentation

Having the package structured in sub-modules is very practical for development, but not so much for exporting the package. In Go, each of the sub-modules needs to be individually imported. The types from the sub-modules can be exposed from the main module by taking advantage of the type aliasing that Go allows. Contrary to type definition, type aliases are equal to the original type. This is to say, in a type definition (`type T1 T2`), `T1` and `T2` are not the same type. But in a type alias (`type T1 = T2`), `T1` and `T2` are the same type. Part of the final result is shown in listing 8.6, which contains a section of `bsp1/bsp1.go`. There is a usage example in section 9.1.1.

```

1 type (
2     Action = proto.Action
3     IO = proto.IO
4     Parameter = proto.Parameter
5     Protocol = proto.Protocol
6     Role = proto.Role
7 )
    
```

```

8 | Reasoner = reason.Reasoner
9 | Instance = reason.Instance
10 | Roles = reason.Roles
11 | Values = reason.Values
12 | )

```

Listing 8.6: Type aliases in BSPL module

8.2 NaHS Module

The NaHS module integrates the BSPL with LibP2P. It contains two main sub-packages, a utility sub-package and three folders. The two main sub-packages are `events` and `net`, explained in sections 8.2.1 and 8.2.2 respectively. The utility package contains functions useful for both packages, specifically to locate test resources. The other three folders are described below.

- **config.** Contains the pre-shared key (PSK) of the private network the node will use. If there is no PSK file, the node will join the public NaHS network.
- **scripts.** A shell script that generates PSK files is located in this folder.
- **test.** Multiple tests resources, including BSPL protocol files and NaHS node keys, explained on section 8.2.2.

8.2.1 Events

Events are used to share BSPL-related messages between nodes. There are three possible actions that might happen between two nodes when enacting a BSPL protocol.

- **Protocol instantiation.** The BSPL protocol is instantiated and the roles are assigned values.
- **Messages.** An action occurs and is communicated in the form of a message, containing new values for the protocol instance.
- **Cancel enactment.** Nodes might decide to drop instances for multiple reasons: to notify the other node they won't be available, trust breaches, internal errors, etc. A cancel event is not required but is useful for both parties.

The actions have been named "new", "update" and "drop". Each event represent an action, therefore three event types have been created in Go, as shown in the generated UML diagram for the `events` sub-package shown in figure 8.7 at the very end of section 8.2. All three types implement the `Event` interface, which unifies event behavior. Events need to be easily identifiable, in this case with an `ID` function that returns a UUID in string form. The event type must also be identifiable so it can be cast to a specific type when necessary. This is done with a `Type` function that returns the event type based on an enumeration named `EventType` as shown in figure 8.7. For convenience when developing the marshal/unmarshal functions explained in the next paragraph, a function to

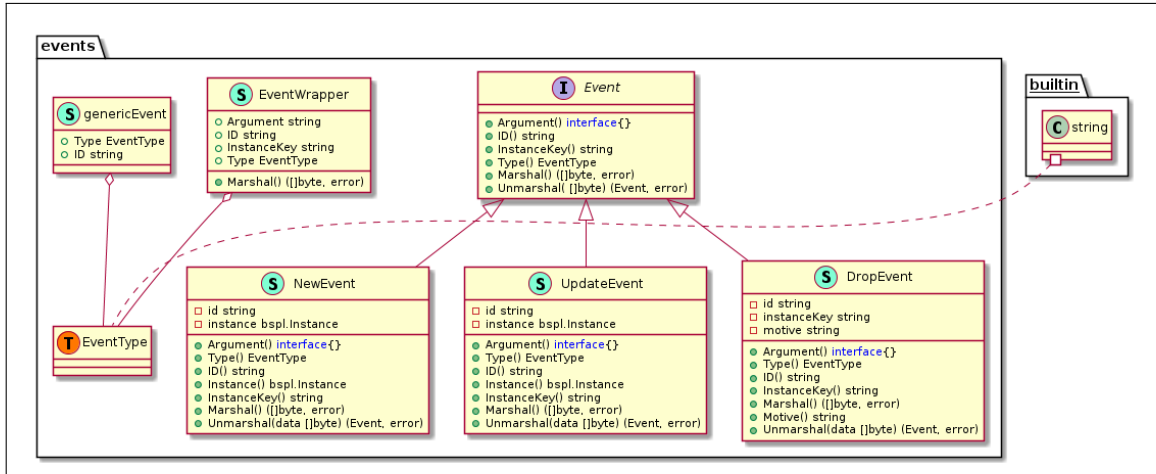


Fig. 8.7: UML diagram of nahs.events

return the arguments of the event is also useful. In this case this is done with an `Arguments` function that returns an `interface{}` type. The `interface{}` type is an interface with no methods, in other words, it is an interface implemented by any type in Go. This allows using parameters without defining their type, as long as the receiver knows exactly how to handle them.

Marshalling and unmarshalling are also essential in order to transmit events and instances from one node to another. One of the most intuitive ways to do so in Go is with JSON, however, the parameters of instanced protocols may have conflicting value with the notation. Fortunately, values on BSPL are always strings and can therefore be easily encoded or have their conflicting characters escaped. The most simple option is encoding the values with base64, as none of the 64 characters used for base64 encoding conflicts with JSON. Figure 8.8 shows a Go structure at the top, the serialized bytes at the bottom, and all the in-between steps in order.

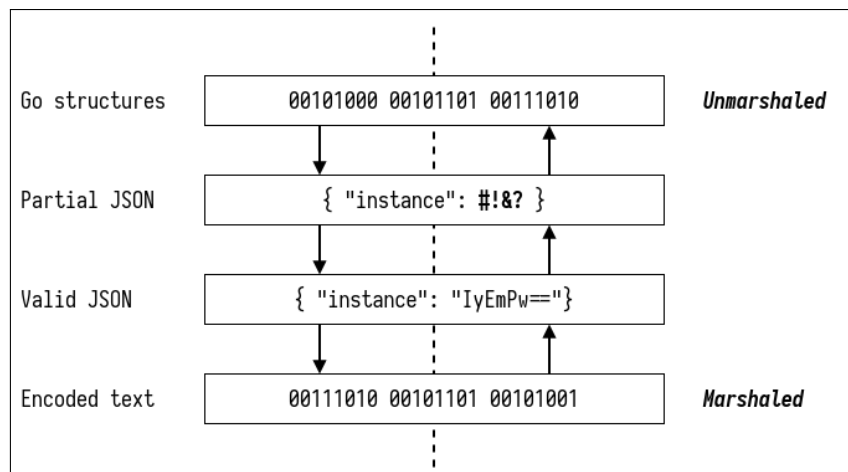


Fig. 8.8: From Go objects to serialized data

A brief explanation of each event implementation is due. Events for instance creation contain an instance with at least one key parameter with a value, and the NaHS IDs (see section 8.2.2) of the node that created the event and the node it will be sent to assigned to the roles of the protocol.

Update events contain a version of the instance with more parameters than when the protocol was instantiated. There might be consecutive updates for a single instance, the order of which is determined by their messages. Finally, the drop event contains the key of the instance to be canceled so the receiving node can identify the instance. The rest of the types seen in the UML diagram, `genericEvent` and `EventWrapper`, are used for testing and marshaling.

There is one last function exported in the package: `RunEvent`. It takes an instance and a reference to a BSPL reasoner as arguments, determines the type of event, and calls the corresponding reasoner function. As an example, if the event were of the type "update", `reasoner.UpdateInstance` would be called.

8.2.2 Network

The networking features of NaHS are implemented in the `net` sub-package. Most of the package functionality is implemented in the `Node` type. As the name indicates, this type represents a single node of NaHS. This section will analyze the node functionalities and behavior. It is worth mentioning the node type has a libP2P peer (or host) as an attribute and each libp2p peer has an ID, which itself is represented with a string containing base58 (alphanumeric) characters. In reality, the ID is a multihash encoded in base58, while a multihash is the encoded output of a cryptographic hash function that contains not only the hash but also the hashing method [28]. Figure 8.9 contains an example of how to turn a sha256 hash into a libp2p peer ID.

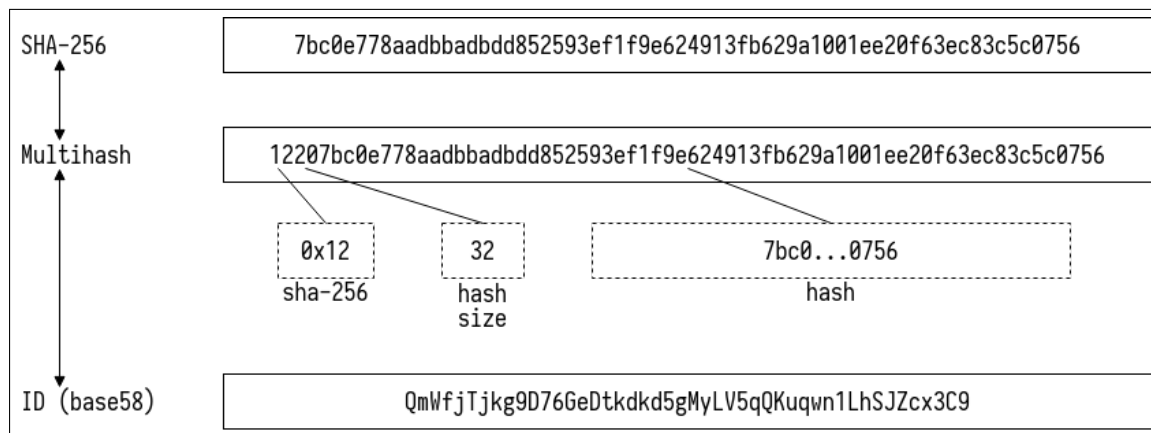


Fig. 8.9: Example of hash, multihash and ID

8.2.2.1 Discovery

The first thing a node does is look for other nodes. The default wait it does so is using the rendezvous protocol (explained in chapter 4 for discovery, detailed in the LibP2P specification [15]).

NaHS nodes use the default nodes provided by and used in IPFS. The node type contains a method to register itself at the rendezvous point and another one to discover other nodes, which can be launched at the developer's discretion. This raises the question of why not to always announce oneself at boot and constantly run a discovery routine on the background. The answer is because

it is not always desired behavior, as low-latency cases such as local tests don't require them and can use the `node.AddContact` method to manually introduce two nodes.

Upon discovering one another, nodes exchange their known and offered BSPL protocols. This is done by calling the `discoveryHandler` function shown in listing 8.7. The function is a wrapper for two other functions, one to read the protocols sent by the remote node and another one to send protocols to the remote nodes. After storing the remote contact information and opening a stream with the remote peer, both of these functions are called as goroutines to be either waited for or timed out. When protocols are received, they are stored and mapped to the node that offers them so the node can be contacted when the protocol is enacted.

```

1 func (n *Node) discoveryHandler(stream network.Stream) {
2     // defer recovery function to log error and close stream
3     defer func() {
4         if r := recover(); r != nil {
5             logger.Errorf("Recovered from error in protocol exchange: %s", r)
6         }
7         stream.Close()
8     }()
9
10    logger.Debug("Opened new BSPL protocol discovery stream")
11    n.addRemotePeer(stream)
12
13    var wg sync.WaitGroup
14    rw := bufio.NewReaderWriter(bufio.NewReader(stream), bufio.NewWriter(stream))
15
16    wg.Add(2)
17    go n.discoveryReadData(rw, &wg, stream.Conn().RemotePeer())
18    go n.discoveryWriteData(rw, &wg)
19    wg.Wait()
20 }

```

Listing 8.7: Section of BSPL protocol discovery source code

8.2.3 Operation

After discovery is taken care of, what a node does depends on its reasoner. This will create a circular reference: the reasoner will have a reference to the node and the node will have a reference to the reasoner. While arguably inelegant, it allows for the reasoner to trigger the node to send events which is very useful. The reasoner can, however, be separated into two concepts: one that manages BSPL information, and one that manages how the agent (member of NaHS) behaves by itself. Reasoner implementation is however up to the developer, different design choices suit different use cases.

The node type offers methods for agents to use as they see fit. To send events, the `SendEvent()` method is called, passing it the ID of the target node and the event to be sent. The function should be launched as a goroutine and marshals the event before sending it to the target node, which runs the event handler when it receives the event. The event handler receives the event, passes it to `events.RunEvent` and responds to the sender if necessary. As explained before, `events.RunServer` calls the reasoner, thus this is the way BSPL messages wrapped in events reach the reasoners. If

something happened along the way and the event couldn't be processed by the remote node, the node will be notified and can react accordingly.

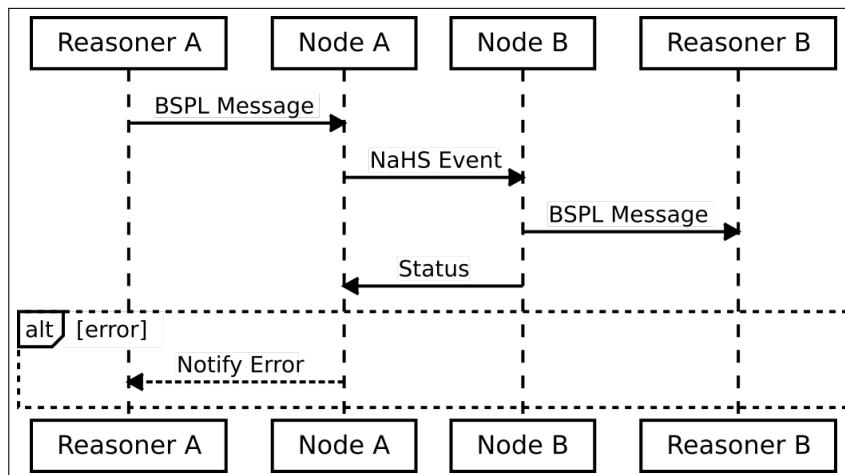


Fig. 8.10: Information flow when reasoner sends a message

Looking more into the development process, each node has a libp2p “Host” as an attribute. As previously explained, each host has a set of private/public keys that uses to communicate and to generate the ID. The private key can be exported using the `ExportKey()` of the NaHS node type. When a new node is created using one of the three constructors described later in this section, the private key can be imported effectively creating a node with the same ID.

LibP2P host options should be discussed now. LibP2P hosts represent a node that can play the role of both client and server in a P2P network. It has many relevant attributes for this project, but the main one is the peer-store. The peer-store is used to store information about other peers, in this case other NaHS nodes. It is the data structure where contact information and peer metadata is stored after being either manually introduced or discovered as described in section 8.2.2.1. Hosts also have very useful methods for opening, closing connections and streams or managing them in more ways... The constructor functions is also noteworthy, as it accepts any number of options, “Option” being a Go type described in the same library as Host (host and option implementations can be further studied in the libp2p repositories [14] [29]). LibP2P hosts are widely configurable, allowing modifications of multiple parts of the network stack: transport, relaying, security... The most relevant options for this project are simpler ones such as the addresses to listen at, the default NAT manager and transports, and the configuration of private networks. Private networks are created by identifying other nodes belonging to the same private network with a pre-shared key, generated using the `scripts/gen_psk.sh` file found in the NaHS repository.

NaHS provides three node constructors by default, but all of them allow introducing any option compatible with libp2p nodes. All run the default NaHS requirements such as setting stream handlers or configuring protocols, but they have noticeable differences.

- **NewNode.** Creates a default NaHS node and launches the rendezvous node registration/discovery routine.
- **LocalNode.** Creates a default NaHS node without setting up any discovery mechanism.

- **NodeFromPrivKey**. Creates a default NaHS node in the private network specified by the pre-shared key passed as a parameter.

The offered and consumed BSPL protocols of a node can be set either when creating it or at any other moment. There are also some utility functions to manually change information about other nodes or to locate contacts in the peer address book that offer a certain role in a BSPL protocol, e.g. they play the Renter role in a BikeRental protocol.

8.2.4 Presentation

```

1 type (
2     // Node of the NaHS network.
3     Node = net.Node
4 )
5
6 // NewNode creates a new NaHS node. LibP2P options can be passed
7 // to configure the node.
8 func NewNode(reasoner bspl.Reasoner, options ...libp2p.Option) *Node {
9     return net.NewNode(reasoner, options...)
10 }
11
12 // MakeNode creates a node with the specified private key so the
13 // node maintains the ID it previously had.
14 func MakeNode(reasoner bspl.Reasoner, sk crypto.PrivKey, options ...libp2p.Option)
15     *Node {
16     return net.NodeFromPrivKey(reasoner, sk, options...)
17 }

```

Listing 8.8: Exported types and functions at NaHS package

Very much like the BSPL modules, giving package users the most simple way of accessing the most relevant components is important. For that purpose, listing 8.8 shows the elements exported at the top-level package, mainly the node type and two simplified construction function. Any other type or function can also be imported by explicitly importing its sub-package.

8.3 Demonstration

The demonstration development was divided into two stages. The first one was a simple interaction between two agents by enacting one protocol. The second one was more complex and involved multiple agents and protocols.

8.3.1 Simple Demo

The proposed scenario for the simple demonstration contains two agents: a person that wants to rent a bicycle and a renter that rents a bicycle. The renter has defined a protocol to rent a bike, shown in listing 8.9. The renter doesn't know what kind of agent will rent a bike, it has

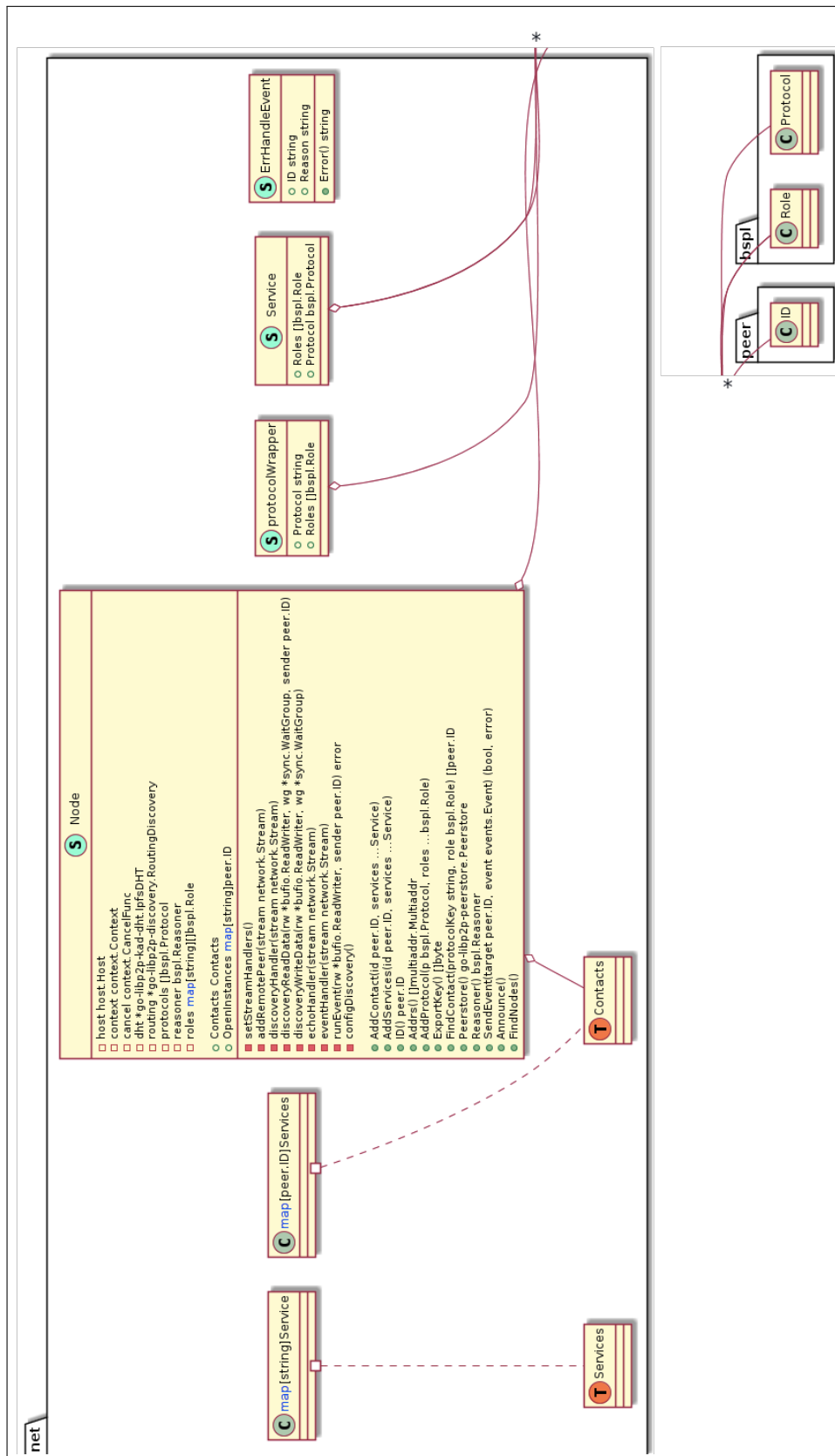


Fig. 8.11: UML diagram of nahs.net

just defined the behavior expected of a “Customer”. In the same way, it has defined behavior for “Renter” without needing to specify any information about itself. These two roles can therefore be played by anyone willing to implement that behavior.

```

1 BikeRental {
2     role Customer, Renter
3     parameter out ID key, out bikeID, out price, in origin, in destination, out
      rID
4
5     Customer -> Renter: request[in origin, in destination, out ID]
6     Renter -> Customer: offer[in ID, in origin, out bikeID, out price]
7     Customer -> Renter: accept[in ID, in bikeID, in price, out rID]
8     Customer -> Renter: reject[in ID, in bikeID, in price, out rID]
9 }

```

Listing 8.9: Bike rental protocol definition

Four actions are defined on the protocol, that generate six parameters. The messages of the actions are `request`, `offer`, `accept` and `reject`. When a customer requests a bike to go from somewhere to another place, the locations are specified in the `origin` and `destination` parameters so the renter knows where the bike is expected to be picked up and where it will be dropped. After receiving a request, the renter can offer a bike to the customer for a price. The bike is identified by the `bikeID` parameter and the offer price by `price`. Finally, the user can either accept or reject the offer by assigning specific values to `rID`. The flow of information is clear in this example: `ID` states that requests must be sent before offers, and the `bikeID` and `price` parameters indicate that accepting or rejecting an offer can't happen before making one.

Two agents will therefore be created for this demo: a person that plays the customer role and a renter that plays the renter role. The client will try to rent a bike and either accept or reject the offer the renter makes.

8.3.1.1 Implementation

The proposed solution for the demo agent implementation is creating an exported type for each agent so it can be managed from outside the package and a local reasoner that won't be accessible by other packages. Listing 8.10 shows a simplified version of the person and person-reasoner types. The reasoner will implement the logic, while the person type will be the highest level component, which calls the reasoner and is affected by it. For example, the reasoner will implement all the methods described in the reasoner interface in the BSPL package, one of the methods being instantiating protocols. When called to instantiate a bike rental protocol given the origin and destination values, the reasoner will do so by looking for a valid renter and send it an event notifying the enactment of the protocol. The `personReasoner` abstracts `personReasoner` from requiring to know either BSPL or NaHS data.

```

1 // Person is a NaHS agent representing a single person
2 type Person struct {
3     maxPrice float64 // maximum euro/min a person is willing to pay for a bike
4     node      *nahs.Node
5     reasoner  *personReasoner
6 }

```

```

7 |
8 | type personReasoner struct {
9 |     node *nahs.Node
10 |     consumedServices map[string]bspl.Protocol
11 |     offers           chan float64
12 | }

```

Listing 8.10: Person and personReasoner types

A simplified example of protocol instantiation is shown in listing 8.11. The exported function `Instantiate` is called, which delegates the instancing to other functions according to the protocol. In this case, the task is delegated to `instantiateBikeRental`, which looks for a valid renter, checks the validity of the parameters, creates the values outputted by the action and creates a new instance with those values. Once the instance is created, the message needs to be sent to the renter. For this, the `sendEvent` method of the NaHS node is called with a new event and the renter as a target. The method is called as a new goroutine to avoid blocking execution while the other node is notified. Once the bike rental is requested, `Person` will keep doing unrelated things until the reasoner receives an update from the renter with an offer and a bike identifier. When the offer is received, the `personReasoner` will compare the price with the one specified by the person and accept or reject the offer accordingly before notifying the person.

The objective of this approach is to show how the agents can continue functioning and are not, at any point, forced to wait for responses. In some cases, the best approach will be to in fact wait until a bike is found, but in many others it may not be the case. This is to say, it will always be the choice to wait for new information as a logical requirement but never as a technical one. It is also worth noting how neither `Person` nor `personReasoner` have to know how the renter is implemented, the only thing they have to do is examine its behavior from the protocol definition. The same goes for the renter, who just defined `customer` in the protocol. This means that any other node, no matter how different, could play the renter role as long as they implement the behavior. This, while an intended objective of the project, would imply that in order to create a secure bike rental protocol more measures would need to be added for authority verification. It is nonetheless a great example of how heterogeneous agents can interact in the network, and how the behavior of the reasoner can be implemented in any way, interactions with other NaHS agents being driven by voluntary actions and the information available.

```

1 | func (pr *personReasoner) Instantiate(p bspl.Protocol, roles bspl.Roles, ins bspl.
  |   Values) (bspl.Instance, error) {
2 |     switch p.Key() {
3 |     case bikeRentalProtocol.Key():
4 |         return pr.instantiateBikeRental(roles, ins)
5 |     case ...
6 |     ...
7 |     }
8 |     return nil, fmt.Errorf("Unkown protocol '%s'", p.Key())
9 | }
10 |
11 | func (pr *personReasoner) instantiateBikeRental(roles bspl.Roles, values bspl.
  |   Values) (bspl.Instance, error) {
12 |     renter, err := personReasoner.findRenter()
13 |     if err != nil {
14 |         return errors.New("No renter found")

```

```

15     }
16     id := uuid.New().String()
17     params := make(map[string]string)
18     required := []string{"in origin", "in destination"}
19     for _, r := range required {
20         v, found := values[r]
21         if !found {
22             return nil, fmt.Errorf("Missing parameter: '%s'", r)
23         }
24         params[r] = v
25     }
26     i := imp.NewInstance(bikeRentalProtocol, roles)
27     i.SetValue("ID", id)
28     i.SetValue("destination", params["in destination"])
29     i.SetValue("origin", params["in origin"])
30     // notify the renter of the new instance
31     go pr.node.SendEvent(renter, events.MakeNewEvent(i))
32     return i, nil
33 }

```

Listing 8.11: Example of an Instantiate method

Listing 8.12 contains a simplified version of how the reasoner of the renter will update the protocol enactment when the customer decides to either accept or reject the offer. When the NaHS networking node receives the update, it calls the `Update` method of the reasoner assigned to the node, in this case `renterReasoner`. The reasoner will then check if the update belongs to an existing instance, extract the actions represented by the messages and act accordingly. In this case, the `rID` paramter will be extracted, analyzed, and if the offer was accepted the rental will be registered.

The specific implementation of each agent and reasoner can be found at the demo repository: <https://github.com/mikelsr/nahs-demo>. The files for the simple demo are under the directory `demo/v1`. While simple, the demo provided insight into the combined usage of the libraries developed throughout the project and demonstrated how an information and interaction-oriented applications can be developed without much difficulty and when combined to the asynchronous design of BSPL provide a lot of flexibility for both design and behavior. The programming language choice also was also proved right as the asynchronicity of the logic and the easy concurrency are a great combination.

```

1 func (rr *renterReasoner) UpdateInstance(j bspl.Instance) error {
2     i, found := rr.Node.OpenInstances[j.Key()]
3     actions, _, err := i.Diff(j)
4     if err != nil {
5         return err
6     }
7     switch j.Protocol().Key() {
8     case bikeRentalProtocol.Key():
9         if len(actions) != 2 {
10            return errors.New("Unexpected actions")
11        }
12        ... // other verifications such as
13            //ensuring actions are accept or reject
14        rID := j.GetValue("rID") // rID should be "accept" or "reject"
15        if rID == "accept" {
16            rr.registerRental(j.GetValue("ID"))

```

```

17     }
18     case b...
19     ...
20     }
21     return i.Update(j)
22 }

```

Listing 8.12: Example of an Update method

8.3.2 Complex Demo

To comply with requirement **O4R1**, the complex demo must have at least four agents that together implement at least five protocols. Building upon the simple example, the agents and motivations of each agent are explained next.

- **Person.** A person wants to go from one point of a city to another. To do so fast, it wants to ride a bike from the origin to the destination. Each person has a maximum price they are willing to pay for a bike ride.
- **Renter.** A renter's goal is to rent as many bikes as possible to as many customers as possible and will therefore offer bikes to any customer as long as there are bikes available. A renter owns multiple bike stations through the city used to pick and drop bikes. When a customer asks for a bike on an origin, the bike will be at the station nearest to the origin.
- **Station.** The goal of a station is to charge the electrical bicycles that are docked on it and will therefore release the bikes with the highest battery levels first. For this reason, the reasoner will obtain the bike IDs from the offers from the stations. Bikes were implemented in the form of a queue, where the bikes stationed the longer are the first ones to be released to emulate charging time.
- **Bike.** Bicycles were added to the example as agents to show that any entity capable of understanding and processing the protocols while acting according to their own logic can be a NaHS agent. In this very simple example, bikes want to be docked at stations after being released from one.
- **Transport.** Transports can move an arbitrary amount of bikes from one station to another. Their goal is to do so, as in a real example this would provide income. Transports can be imagined as vans with room for many bicycles.
- **University.** To promote green methods of transportation, universities want as many as their students and workers to ride bicycles instead of using cars. For that, universities ask renters to have a number of bicycles ready at the nearest station at a certain time, when a lot of people are expected to leave the campus.

There is one great thing about this example: while people, transports and university are independent, renters control the stations which control the bicycles. This is, however, unknown to the

customers and transports and they are still capable of consuming the services and performing the tasks.

Each of the new protocols will be shown and explained (the bike rental protocol is implemented too but has already been explained), starting with bike rides shown in listing 8.13. This protocol was added to give meaning to the bicycle agent by representing bike pickups and drops. The protocol has the “Rider” and “Bike” roles. When a bike is picked, the instance is created with a rental ID representing that the rider has indeed rented the bike. When the bike is dropped, the ID of the station is found and added.

```

1 BikeRide {
2     role Rider, Bike
3     parameter out ID key, out rentalID, out dropStation
4
5     Rider -> Bike: pick[out ID key, out rentalID]
6     Rider -> Bike: drop[in ID key, in rentalID, out dropStation]
7 }

```

Listing 8.13: Bike ride protocol

The bike request that universities use is shown in listing 8.14. The request does not necessarily have to be made by universities, universities play the “Requester” role. Requesters ask for a certain amount of times at a certain date and time, and after evaluating if it is possible the renters confirm the request.

```

1 BikeRequest {
2     role Requester, Renter
3     parameter out ID key, in bikeNum, in datetime, in station, out offerNum,
4     out rID
5
6     Requester -> Renter: request[out ID, in bikeNum, in datetime]
7     Renter -> Requester: accept[in ID, out rID, out offerNum]
8     Renter -> Requester: reject[in ID, out rID, out offerNum]
9 }

```

Listing 8.14: Bike request protocol

Bike transportation can be triggered for multiple reasons: a station is running low on bikes, high demand is expected at a certain time or a bike request has been made and there are not enough bikes. The protocol shown in listing 8.15 shows how an agent can request a number of bicycles to be transported from one station (`src`) to another station (`dst`) at a certain time. The transport can then either accept or reject the request. If accepted and carried out, the transport can be successful or unable to transport the bikes if, for example, there were not enough bikes to transport.

```

1 BikeTransport {
2     role Requester, Transport
3     parameter out ID key, in bikeNum, in src, in dst, in datetime, out rID, out
4     result
5
6     Requester -> Transport: request[out ID, in bikeNum, in src, in dst, in
7     datetime]
8     Transport -> Requester: accept[in ID, out rID]
9     Transport -> Requester: reject[in ID, out rID]
10    Transport -> Requester: success[in ID, in rID, out result]

```

```

9   Transport -> Requester: failure[in ID, in rID, out result]
10 }

```

Listing 8.15: Bike transport protocol

The bike ride protocol gave no autonomy to the bicycle. For this, and to fulfill the bike’s objective of docking at stations, the bike storage protocol shows the logical relationship between stations and bikes. Bikes dock to stations, while stations release bike when a rental is processed.

```

1 BikeStorage {
2     role Bike, Station
3     parameter out ID key, in rentalID
4
5     Bike -> Station: dock[ID key]
6     Station -> Bike: release[ID, in rentalID]
7 }

```

Listing 8.16: Bike storage protocol

The last protocol is used by any agent to locate nearby bike stations. An agent that wants to find the nearest station can ask another agent able to play the “Locator” role for the ID of the station. The position of the agent playing “User” is given with coordinates.

```

1 StationSearch {
2     role User, Locator
3     parameter out ID key, in coordinates, out stationID
4
5     User -> Locator: request[out ID, in coordinates]
6     Locator -> User: inform[in ID, out stationID]
7 }

```

Listing 8.17: Station search protocol

A reasoner was developed for each of the agents with the same philosophy as the simple demo. As an example scenario involving the agents, four bikes were created and docked evenly in two stations belonging to one renter. A person near one of the stations requested to travel near the other, while the university requested two bikes to be available in the station the bike was taken from in the immediate future, triggering the transport request for more bikes. When the person rents a bike, picks it, rides and drops it, the rest of the protocols are also involved. Figure 8.12 shows the log outputted by this scenario. It should be noted that due to the asynchronous nature of the application and the concurrency of the implementation design, it is possible for an event to happen in one node, the event to be sent to another node, logged at the other node and then logged at the original node, resulting on the log statements showing up in what could appear to be random order but is really a result of the design.

8.4 User Manual

Documentation has been created for the BSPL and networking repositories. It contains short explanations about each exported type and method. The URLs of the pages the documentation is

```

Created bike with ID LLAQ (QmcaLXdAC1RWfGJvyeZ6MugBEqoDSXVgFXh3x6RUD9LLaQ)
Created bike with ID XWVJ (QmPftWe8DNkt7HqEN6kAmD6njWM3dar7XJA6bnujqkxWwJ)
Created bike with ID WEAF (QmYJb9ovv1LbAuwocv7roxP9VzTrjZzUfJLLTgoL5weaf)
Created bike with ID FCNS (QmQFUixwJM8XychHHfPtMn5o9x4yLJCcZbKexkfPVnFCNs)
Created station with ID BXGE (QmRSZSGMnkbGkj9MzippGrKZoChBMDLiHGLgeQ666JBxgE)
[BXGE] Bike LLAQ docked
[BXGE] Bike XWVJ docked
Created station with ID XUC8 (QmRcQxF3nWPEZLBNrWUoW8sWX1TpVBczEDHLtayBpmxuc8)
[XUC8] Bike WEAF docked
[XUC8] Bike FCNS docked
Created transport with ID PQUB (QmX1gr24SNdYwEnvR5KwvRj8CShQdMSozURgyK2y81PqUB)
Created university with ID 7BTP (QmXDCzLmytcWGzciWLSHAxc5ZKTJeG1NZe4Kz8DCXd7bTp)
Created renter with ID W7GW (QmVMhuBdVxtsm3pykK4fbgtS92sLE9oAbiE1FJstKCw7gw)
Created person with ID ABZJ (QmPqcXKsudZ86Py2iFHNrfeAHK1ck19MnM3bFugEgkABZJ)
[ABZJ] Search for station
[W7GW] Send event 'update:4261' to node ABZJ (instance key: StationSearch,ID:dccd7a0a-b791-4f2a-a41f-99d7ea8575f5)
[ABZJ] Nearest station found: BXGE
[ABZJ] Sent rent request to W7GW
[W7GW] Send event 'update:D951' to node ABZJ (instance key: BikeRental,ID:708d84b4-6017-42eb-842c-42429ebc96ee)
[ABZJ] Received offer for bike LLAQ at price: '0.03'
[ABZJ] Accepted offer for price '0.03'
[ABZJ] Bike with id LLAQ rented
[ABZJ] Waiting to discover node LLAQ
[ABZJ] Send event 'update:9585' to node W7GW (instance key: BikeRental,ID:708d84b4-6017-42eb-842c-42429ebc96ee)
[ABZJ] Discovered LLAQ
[ABZJ] Send event 'new:0FED' to node LLAQ (instance key: BikeRide,ID:f4f48beb-3f82-4f2b-8e75-6df9744d0dc0)
[W7GW] Send event 'update:C64D' to node ABZJ (instance key: StationSearch,ID:ff0ba9fb-150f-4b4c-a9c2-4371ad2675a0)
[W7GW] Response from ABZJ for bike LLAQ offer: accept
[ABZJ] Nearest station found: XUC8
[ABZJ] Dropping bike LLAQ at station XUC8
[7BTP] Requesting 2 bike(s) from W7GW to station BXGE at 2020-06-18 23:53:00.364080301 +0200 CEST m=+3.593868373
[LLAQ] New rider ABZJ
[ABZJ] Send event 'update:ED90' to node LLAQ (instance key: BikeRide,ID:f4f48beb-3f82-4f2b-8e75-6df9744d0dc0)
[LLAQ] Dropped at XUC8 by ABZJ
[W7GW] Received request for 2 bikes at station BXGE and time 2020-06-18T23:53:00.364080301+02:00
[W7GW] Request bike transport from PQUB
[W7GW] Send event 'new:8AD1' to node PQUB (instance key: BikeTransport,ID:3943c250-8160-4eed-a2a3-8849a2778d43)
[PQUB] Send event 'update:239B' to node W7GW (instance key: BikeTransport,ID:3943c250-8160-4eed-a2a3-8849a2778d43)
[W7GW] Accepting request 'BikeRequest,ID:275a3301-a629-4e04-adc6-1c36b64af162'
[W7GW] Send event 'update:E9A6' to node 7BTP (instance key: BikeRequest,ID:275a3301-a629-4e04-adc6-1c36b64af162)
[7BTP] Success requesting '2' bikes
    
```

Fig. 8.12: Section of the output log of the complex demo execution

hosted at. There are two links for each: one to `godoc.org` and another one to `pkg.go.dev`. The former is intended to be the host of documentation for packages hosted on multiple Git repository hosting providers, while the latter is intended to create a centralized place to access resources about Go and the Go ecosystem.

- **BSPL**

- <https://godoc.org/github.com/mikelsr/bspl>
- <https://pkg.go.dev/github.com/mikelsr/bspl>

- **Networking**

- <https://godoc.org/github.com/mikelsr/nahs>
- <https://pkg.go.dev/github.com/mikelsr/nahs>

Chapter 9

Results

The objectives listed in section 2.3 and detailed in chapter 5 have been achieved. As a result, two software libraries and one demo scenario have been produced. At the same time, each one of the libraries has been documented and the documentation has been made publicly available and has been referenced from the libraries to make it more accessible. Section 9.1 will go over the libraries, while section 9.2 discusses the demo.

9.1 Libraries

Objectives **O1**, **O2** and **O3** all stated that software libraries had to be created with specific purposes. The functionalities of objectives **O1** and **O2** were to be implemented under the same library but in different sub-packages, and this is the final result. The repository containing the library has the name “BSPL”. Objective **O3** referred to a library with the networking components, which is what the “NaHS” repository contains. Both libraries have been continually tested during development, both locally before staging changes and in continuous integration tools once the changes reached their respective central repository. Both libraries have been published and can therefore be directly imported in any Go program.

The repositories of each library contain a short README file that gives relevant information about the project at first glance. This information includes a short description of the project and each of the sub-packages and sub-folders, the status of the last tests, test coverage percentage, project license, the location of the documentation and the version of Go the project was built with. Many of this information can be more thoroughly examined, for example, the test results and coverage reports link to web pages with much more detailed information. The README is however intended to quickly provide context and relevant information about the project, the task of giving more detailed information belongs to each of the other information sources. Figure 9.1 contains a screenshot of the first thing users see when browsing the NaHS repository, which is similar in content to what is seen when browsing the BSPL repository.

Section 9.1.1 reports the final status of the BSPL library and provides a simple example of how to

use it while section 9.1.2 does the same for the NaHS library.

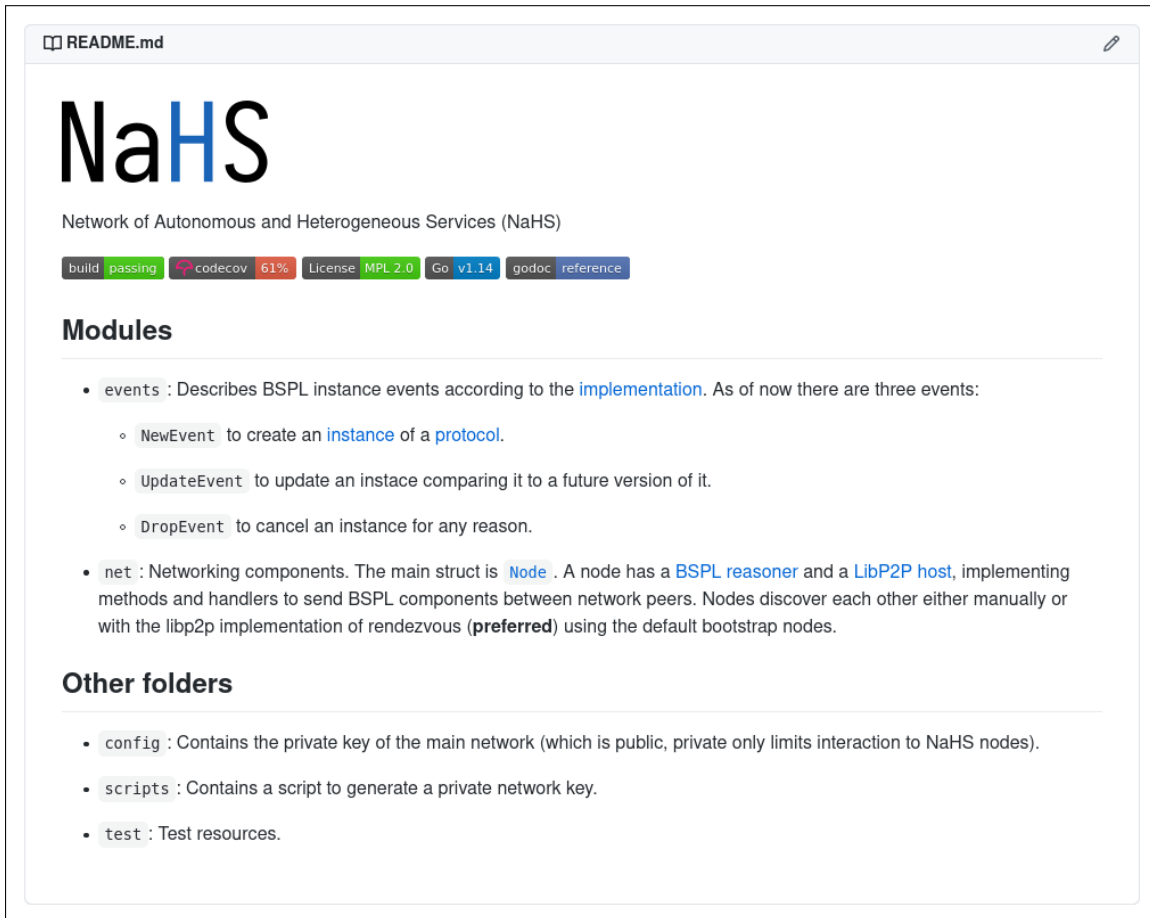


Fig. 9.1: Screenshot of the README.md file from the NaHS repository

9.1.1 BSPL

As planned, the BSPL library contains a BSPL parser that receives a raw-text protocol as input and outputs its equivalent Go structure. It also contains interfaces defining the behavior of instances and reasoners, while providing an implementation of instances that implements said interface. The library can be accessed at <https://github.com/mikelsr/bspl> and the documentation can be accessed at any of the links provided in section 8.4.

Listing 9.1 shows the most basic usage example of the BSPL parser. It assumes that the `path` variable contains a string with the path to a raw-text file containing a BSPL protocol, opens the file and passes the reader to the `bspl.Parse` function, which returns the resulting BSPL protocol represented in Go and an error, if the protocol could not be correctly parsed. It should be noted that `bspl.Parse` can receive any type that implements the `io.Reader` interface and is not limited to files. It should also be noted that while the original function and types are implemented inside the `bspl/parser` and `bspl/proto` packages, the aliases and wrappers created at the top-level package make them available from the top-level `bspl` package.

```

1 package main
2
3 import (
4     "fmt"
5     "os"
6
7     "github.com/mikelsr/bspl"
8 )
9
10 func main() {
11     source, _ := os.Open(path)
12     protocol, err := bspl.Parse(source)
13     if err != nil {
14         panic(err)
15     }
16     fmt.Println(protocol)
17 }

```

Listing 9.1: BSPL module usage example

9.1.2 NaHS

The NaHS module contains all the required networking features. Networking is centered around the `node` type, which allows NaHS agents to announce themselves, discover other agents, interact with other agents via events and form private networks. At the same time, `node` is built around the `bspl.Reasoner` interface, as NaHS agents will communicate with BSPL protocols. Events are implemented in their own sub-package and define communication from a lower-level aspect than BSPL protocols, which are wrapped in events. NaHS agents use BSPL protocols to interact, while NaHS networking nodes use events. The project is available at <https://github.com/mikelsr/nahs> and the documentation is accessible at the links provided in section 8.4.

```

1 package main
2
3 import (
4     "fmt"
5
6     "github.com/mikelsr/bspl"
7     "github.com/mikelsr/nahs"
8 )
9
10 func main() {
11     var reasoner bspl.Reasoner
12     node := nahs.NewNode(reasoner)
13     fmt.Printf("Created a new NaHS node with ID: %s\n", node.ID())
14 }

```

Listing 9.2: NaHS node creation example

Having the package designed with the `bspl.Instance` and `bspl.Reasoner` interfaces means that even if the implementations of each of them change, the networking components won't need to be modified. Figure 9.2 shows the most basic example of a NaHS node creation by initializing a reasoner variable

and creating a new node. If the program wouldn't exit right after creating the node, the mode would automatically contact the rendezvous point to announce itself and search for other nodes.

9.2 Demo

Two demo scenarios have been created: one where two agents interact with a single protocol and one where multiple agents interact with each other by using many different protocols. Both of them are functional and show how the two libraries combine to form a network of heterogeneous agents that interact with each other to achieve their own goals. The most useful aspect of the demo agents is that they implement the reasoner interface defined in the BSPL repository and use many of the networking features to communicate with each other while the interaction logic is dictated by protocols. The demo is available at <https://github.com/mikelsr/nahs-demo> and have been documented on section 8.3.

It is an example of how the resources provided by the NaHS and BSPL libraries might be used to create agents that have autonomy while being able to interact with other nodes when they offer something of interest to the agent. It is now, however, designed to be the standard of how agents are created and there are many other ways to do so.

Chapter 10

Conclusions

The project was successful in what it set out to do. The implement the functionalities specified in chapter 5 while meeting the requirements and hide the underlying complexity of processes by exposing only the top-level functions in the main package, but still provide developers with the capacity to manipulate the libraries at a lower level by importing sub-packages. The demo provides an example of how agents can be implemented but the design of the agents is entirely up to the development and might be better approached from other perspectives depending on each agent.

In the network created in this project, an agent, which can be any kind of entity, defines the services it offers by using the Blindingly Simple Protocol Language and interprets what other agents offer with that same language. Agents join a network of agents where contact information and protocols are exchanged, increasing the number of services the agent has access to. Agents can then interact by enacting specific protocols with the tools provided by the NaHS libraries and with the logic and decision criteria implemented in their reasoners. Agents that enact protocols don't need to know how the other agent and the reasoner of said agent are implemented, minimizing the requirements of a protocol enactment. Agents retain local states of the enactments and communicate by updating the local states with messages that represent actions. The information-driven approach means interactions are based on the information available to each agent, and the asynchronous design allows the agents to continue functioning without requiring waiting for new messages.

NaHS, in its current state, is a first idea of what the network it defines could be and needs work before it is used as in a real-world system. There are some ways in which NaHS could be immediately improved and the next section discusses some ideas to do so.

10.1 Future Work

Designing and adding an accountability system could give nodes autonomy when consuming and offering services. A protocol enactment can be interpreted as a contract between two agents. What happens if one of them breaks it? Is it a one-time event or is the agent frequently taking advantage of other agents? An accountability system shared between the agents of the network could help

the network fight these issues. At the same time, if an agent reports a breach of trust, how can it be verified? Many questions arise when developing this topic and developing it would mature NaHS beyond its current state.

The design of the agents was more complicated than expected, and this should be worked on to make NaHS as accessible as possible. Analyzing the most common difficulties when designing agents could give insight into what can be improved, possibly by developing the networking module to add more features and reduce the manual work done by agents. Care should be taken into designing and implementing these features without restricting what can be done with the NaHS libraries. This should be the first milestone in the future work roadmap and be frequently revisited to improve the quality of service of NaHS.

Finally, it would be interesting to see the project applied to a real scenario as a first test of what a real network of autonomous and heterogeneous services would be like, even if the libraries are not mature. Analyzing the behavior of the agents in an uncontrolled environment would possibly be different from the one observed in the controlled environment of the demos. The real scenario could have different types of agents offering and consuming new services as their needs change. The agents could range from apps used by actual people to IoT devices, physical machines, online platforms, and service providers.

10.2 Personal Assessment

This project has been not only a great learning experience but also consolidated the interest on research. Finding a problem by reading academic publications and researching designing a possible solution was interesting, as it presented new problems that could be solved with a lot of flexibility regarding what path the solution would take. Requiring the student to research heavily on topics to be able to come with a solution was motivating and the range of topics the project covers kept it interesting as research and development advanced. The implementation of the solution in a relatively new programming language with different design patterns than the ones studied in university subjects, while challenging at times, provided valuable insight into programming that will surely prove useful in the future. The topics studied in the project have also led to new ideas and topics that show how much there is still to learn.

Overall, the project has been a more than appropriate way of applying many things learnt during the degree and has provided new knowledge that complements the one acquired in the university as both a student and a research intern.

Bibliography

- [1] M. P. Singh and A. K. Chopra. The internet of things and multiagent systems: Decentralized intelligence in distributed computing. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1738–1747, June 2017.
- [2] Protocol Labs. “libp2p: A modular network stack”. <https://libp2p.io/>. [Online; accessed 11-Jun-2020].
- [3] Protocol Labs. “InterPlanetary File System”. <https://ipfs.io/>. [Online; accessed 11-Jun-2020].
- [4] Munindar P. Singh. Information-driven interaction-oriented programming: Bspl, the blindingly simple protocol language. In *10th International Conference on Autonomous Agents and Multiagent Systems*, volume 1 of *AAMAS '11*, pages 491–498. International Foundation for Autonomous Agents and Multiagent Systems, 01 2011.
- [5] Free Software Foundation. “What is free software?”. <https://www.gnu.org/philosophy/free-sw.html>. [Online; accessed 11-Jun-2020].
- [6] Nyree Lemmens, Steven Jong, Karl Tuyls, and Ann Nowe. Bee behaviour in multi-agent systems, 01 2007.
- [7] Michael N. Huhns. Interaction-oriented software development. *International Journal of Software Engineering and Knowledge Engineering*, 11:259–279, 06 2001.
- [8] Yoann Kubera, Philippe Mathieu, and Sébastien Picault. Ioda: An interaction-oriented approach for multi-agent based simulations. *Autonomous Agents and Multi-Agent Systems*, 23:303–343, 11 2011.
- [9] Trevor Perrin. “The Noise Protocol Framework”. <http://www.noiseprotocol.org/noise.html>. [Online; accessed 12-Jun-2020].
- [10] Kenta Iwasaki. “Noise v1.1.2: Fearless, decentralized p2p networking in Go”. <https://medium.com/perlin-network/noise-v1-1-2-fearless-decentralized-p2p-networking-in-go-bf3afdd77230>, 2020. [Online; accessed 12-Jun-2020].
- [11] Protocol Labs. “go-libp2p-noise”. <https://github.com/libp2p/go-libp2p-noise>. [Online; accessed 12-Jun-2020].

Bibliography

- [12] Munindar P. Singh. BSPL, the Blindingly Simple Protocol Language. <https://www.csc2.ncsu.edu/faculty/mpsingh/local/SOC/s12/slides/BSPL-intro-talk-v1.pdf>, April 2011. [Online; accessed 12-Jun-2020].
- [13] Protocol Labs. “libp2p specification”. <https://github.com/libp2p/specs>. [Online; accessed 12-Jun-2020].
- [14] Protocol Labs. “The Go implementation of the libp2p Networking Stack”. <https://github.com/libp2p/go-libp2p>. [Online; accessed 12-Jun-2020].
- [15] Protocol Labs. “Rendezvous Protocol”. <https://github.com/libp2p/specs/tree/master/rendezvous>. [Online; accessed 17-Jun-2020].
- [16] David Anderson. “The Clockwise/Spiral Rule”. <http://c-faq.com/decl/spiral.anderson.html>, 1994. [Online; accessed 14-May-2020].
- [17] Dave Cheney. “Clear is better than clever”. <https://dave.cheney.net/paste/clear-is-better-than-clever.pdf>, 2019. [Online; accessed 14-May-2020].
- [18] Brad Peabody. “Server-side I/O Performance: Node vs. PHP vs. Java vs. Go”. <https://www.toptal.com/back-end/server-side-io-performance-node-php-java-go>, 2017. [Online; accessed 14-May-2020].
- [19] The Go Authors. “Effective Go”. https://golang.org/doc/effective_go.html#concurrency. [Online; accessed 14-May-2020].
- [20] Mozilla Foundation. “Mozilla Public LicenseVersion 2.0 — Mozilla Foundation”. <https://www.mozilla.org/en-US/MPL/2.0/>. [Online; accessed 12-June-2020].
- [21] Scott Chacon and Ben Straub. *Pro Git*. Apress, USA, 2nd edition, 2014.
- [22] GitLab. “GitLab.org repository”. <https://gitlab.com/gitlab-org/gitlab>. [Online; accessed 17-Jun-2020].
- [23] The Go Authors. “Documentation of the testing package”. <https://golang.org/pkg/testing/>. [Online; accessed 13-May-2020].
- [24] The Go Authors. “Documentation of the sort package”. <https://golang.org/pkg/sort/#Interface>. [Online; accessed 13-May-2020].
- [25] Mikel Solabarrieta. “Building a configurable lexer from a deterministic finite automaton”. <https://blog.mikel.xyz/posts/building-configurable-lexer-from-a-deterministic-finite-automaton>, 2019. [Online; accessed 13-May-2020].
- [26] Wikipedia. “Interface segregation principle — Wikipedia, the free encyclopedia”. <http://en.wikipedia.org/w/index.php?title=Interface%20segregation%20principle&oldid=937349235>. [Online; accessed 14-May-2020].
- [27] “Protocheck: Temporal logic implementation for verifying correctness properties of multi-agent protocols”. <https://github.com/shader/protocheck>. [Online; accessed 20-Jun-2020].

- [28] Protocol Labs. “Multihash. Self describing hashes - for future proofing.”. <https://github.com/multiformats/multihash>. [Online; accessed 15-Jun-2020].
- [29] Protocol Labs. “Go libp2p core”. <https://github.com/libp2p/go-libp2p-core>. [Online; accessed 12-Jun-2020].

Acronyms

BSPL	Blindingly Simple Protocol Language
CI	Continuous Integration
DB	Database
DHT	Distributed Hash Table
GCC	GNU Compiler Collection
GNU	GNU's Not Unix!
HTTPS	Hypertext Transfer Protocol Secure
IOP	Interaction-Oriented Programming
IPFS	InterPlanetary File System
ISP	Interface Segregation Principle
JSON	JavaScript Object Notation
LoST	Local State Transfer
MAS	Multi-Agent System
NaHS	Network of Autonomous and Heterogeneous Services
OS	Operating System
PSK	Pre-shared Key
SHA	Secure Hash Algorithm
SSH	Secure Shell
TDD	Test Driven Development
UML	Unified Modeling Language
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
VC	Version Control

Acknowledgements

I would like to thank Diego Casado Mansilla for his friendship and mentoring these five years, his help has been of inestimable value both as my professor and my supervisor. I would also like to thank the entire MORElab team for welcoming me into their research unit and making my experience in Deusto a very pleasant one. I am really looking forward to crossing paths again in the future.