DEPARTMENT OF
COMPUTER SCIENCE

DIOGO FILIPE MARCOS BARRETO
Bachelor in Computer Science

GENERIC DECENTRALIZED MEMBERSHIP AND COMMUNICATION ABSTRACTIONS FOR EDGE SYSTEMS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
December, 2023

DEPARTMENT OF
COMPUTER SCIENCE

GENERIC DECENTRALIZED MEMBERSHIP AND COMMUNICATION ABSTRACTIONS FOR EDGE SYSTEMS

DIOGO FILIPE MARCOS BARRETO

Bachelor in Computer Science

Adviser: João Carlos Antunes Leitão
Associate Professor, NOVA University Lisbon

Co-adviser: Nuno Manuel Ribeiro Preguiça
Full Professor, NOVA University Lisbon

Examination Committee

Chair: João Baptista da Silva Araújo Júnior
Associate Professor, NOVA University Lisbon

Rapporteur: Miguel Filipe Leitão Pardal
Assistant Professor, University of Lisbon

Adviser: João Carlos Antunes Leitão
Associate Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
December, 2023

Generic Decentralized Membership and Communication Abstractions for Edge Systems

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.1) [35].

To family and friends.

Acknowledgements

First I would like to thank my adviser, Prof. João Leitão, for his enormous support, valuable suggestions, and reviews during the execution of this dissertation. Thank you for guiding me through this world of distributed and decentralized systems not only during this dissertation but also throughout my academic journey. I would also like to thank my co-adviser Prof. Nuno Preguiça for his important lessons on distributed systems during my master's and bachelor's years.

I extend my acknowledgements to the Department of Computer Science of the NOVA School of Science and Technology where, throughout the last five years, I had many good moments and where I learned much of what I know today.

To Pedro Ákos Costa and Pedro Fouto, for all the help, with ideas and suggestions that made this work better.

A special acknowledgement to Jacinta Sousa, Diogo Paulico, and David Antunes, for your contributions with ideas for improving this dissertation, in our meetings and outside them, as well as all the support during these last months. These acknowledgements are also extensible to everyone at *our* Room 248 where we worked a lot, but also had a lot of fun.

For enduring me during the last five years, a special acknowledgement to Diogo Barata, who I already knew for many years, and Márcia Matias, who is the example of how university can bring incredible friends. A special thanks to someone outside of university, David Nunes, who also supported me during all these years. No one gets anywhere without great friends, and you are the ones.

An acknowledgement to my girlfriend, Maria Inês, for her fantastic support during the years and an apology for all the missing moments. Thank you for everything and specially for your support during all the projects, evaluations, and this dissertation.

Finally, to my parents, Helena Correia and Orlando Barreto, who always supported me, no matter what, and always gave me all conditions to follow my dreams. You made me what I am today.

A last word to João Maria, I know you are watching from somewhere in the universe.

"Science is a way of thinking much more than it is a body of knowledge." (Carl Sagan)

Abstract

Nowadays, many systems rely on decentralized architectures as a way to provide services to many users that require high availability, fault tolerance, and scalability. Avoiding a centralized component, which is often the source of bottlenecks and a single point of failure, makes these systems more autonomous and robust. Additionally, not relying on proprietary infrastructures also brings privacy-related benefits. Systems that require some form of coordination between different components in order to perform a particular task, for instance in the context of edge computing, usually benefit from relying on decentralized architectures. Examples of these systems include swarms of satellites, IoT appliances, industrial machines, clusters of computers processing large amounts of data, among others.

Decentralized architectures need to provide two fundamental functionalities: the management of peers participating in the system, commonly called membership management, and mechanisms to support communication between peers in order to send and receive information. Many solutions providing these functionalities were already developed in the context of peer-to-peer systems, specially when considering the development of protocols based on overlay networks. While, on one hand, the existence of several decentralized protocols, some of which providing similar services, allows for applications to choose the most suitable one for their operation. On the other hand, should the need for a change of protocol providing a given service arise, applications may have to be extensively rewritten due to the differences between interfaces. This also makes it difficult to reuse solutions based on the same decentralized services across different applications.

This work studies existing decentralized protocols with a focus on their operation, exposed interfaces, and provided services. Then, leveraging on the lessons learned from the performed study, we propose generic abstractions for interacting with decentralized protocols based on the implemented services. Moreover, our architecture provides mechanisms to simplify the management of protocols that can be leveraged by applications based on decentralized protocols to take advantage of an edge computing approach. In this document, we also present the implementation, based on our architecture, of a set of applications and existing protocols.

Keywords: Distributed systems, Decentralized systems, Decentralized applications, Overlay networks, Edge computing, Peer-To-Peer systems

Resumo

Hoje em dia, vários sistemas baseiam-se na utilização de arquiteturas descentralizadas para fornecer serviços com garantias de alta fiabilidade, disponibilidade e escalabilidade. Evitar a utilização de um componente centralizado permite evitar pontos de estrangulamento bem como a existência de um único ponto de falha, tornando estes sistemas mais autónomos e robustos. Adicionalmente, não depender de infraestruturas proprietárias traz também benefícios relacionados com a privacidade. Uma das aplicações mais comuns para este tipo de arquiteturas são sistemas que necessitam de alguma forma de coordenação entre os diferentes atores que os compõem para realizar uma determinada tarefa, por exemplo, no contexto da computação na *edge*. Estas tarefas podem tratar-se de coordenação entre grupos de satélites, dispositivos IoT, máquinas industriais ou computadores responsáveis pelo processamento abundante de dados, entre outros.

De uma forma geral, sistemas baseados em arquiteturas descentralizadas necessitam de fornecer duas funcionalidades fundamentais: mecanismos de gestão dos nós que participam no sistema, conhecidos como gestão de filiação (*membership management*), e mecanismos de comunicação entre esses mesmos nós. Diversas soluções já existem, em particular no domínio dos sistemas entre-pares (*peer-to-peer*), onde protocolos baseados em redes sobrepostas (*overlay networks*) são habitualmente utilizados. Assim, apesar da existência de múltiplos protocolos, capazes de fornecer serviços similares, permitir às aplicações escolher o mais adequado à sua operação, a substituição de um protocolo por outro pode levar a que uma aplicação tenha de ser significativamente modificada. A reutilização de código entre aplicações baseadas na utilização dos mesmos serviços também se torna difícil devido às diferentes interfaces expostas pelos protocolos.

Neste trabalho diversos protocolos descentralizados foram estudados, considerando a sua operação, interfaces expostas e serviços disponibilizados, de forma a propor um conjunto de abstrações genéricas para interagir com os mesmos baseadas nos serviços fornecidos. A arquitetura apresentada permite também uma gestão simplificada dos diversos protocolos utilizados por aplicações, baseadas em computação na *edge*, para obter os serviços necessários. Neste documento, apresentamos ainda a implementação, baseada na nossa solução, de um conjunto de aplicações e protocolos já existentes.

Palavras-chave: Sistemas distribuídos, Sistemas descentralizados, Aplicações decentralizadas, Sistemas entre-pares, Redes sobrepostas, Computação na *edge*

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Nowadays, distributed architectures are widely used as a way to build robust, scalable, and fault-tolerant systems. These architectures are those in which components cooperate with one another over a network in order to perform a given task. We can define a network, such as the Internet, as a set of interconnected machines (or processes) able to communicate between themselves [37].

Systems that operate on top of a network can take advantage of a more centralized or decentralized approach. In a centralized system all information needs to be sent to a (potentially logic) central point, like a data center, to be processed, thus making this central component responsible for a significant part of the operation. Alternatively, in a decentralized architecture the nodes present on the network can work together and cooperate, by sending and receiving messages directly among themselves, to perform the required operation. Steen and Tanenbaum discuss in [37] the difference between a *distributed system* and a *decentralized system* stating that both may rely on multiple machines performing a certain operation but in the first case "processes and resources are sufficiently spread across multiple computers", e.g., an email service that relies on multiple servers as a way to distribute load and improve fault-tolerance, yet on the second case "processes and resources are necessarily spread across multiple computers" making the distribution of processes a core aspect of the system.

A decentralized approach offers many advantages when compared to a single process or a group of autonomous processes with no communication between themselves sending and receiving information to/from the same central location. These include improved availability by avoiding single points of failure, because a node can substitute another one in the execution of a given task, or better scalability by opening possibilities for clients and data to be distributed across nodes [29, 37]. Decentralized systems may also improve reliability by replicating information on different nodes minimizing the risk of data loss. These approaches may even increase privacy and tolerance to malicious attacks due to the lack of a single target and the possibility of using the nodes in the network to hide the identity of users or the exchanged information among them [10, 11].

On the other hand, centralized systems can be considered easier to maintain as information only needs to be sent to a central point [37]. As a consequence, they do not have to deal neither with the heterogeneity questions related with the characteristics of each node nor the membership management and communication issues regarding the distributed nature of the infrastructure. In fact, when comparing centralized and decentralized systems it is not uncommon to consider decentralized systems algorithms to be more complex and difficult to understand and implement than their centralized counterparts.

Distributed systems can vary from large sets of globally interconnected machines and processes relying on proprietary network infrastructures, in many cases running in the same data centers as is the case of large companies like Google, Microsoft, or Amazon [2]; to smaller scale networks of connected devices based on already existing networks like the Internet.

A particular type of network architectures, consisting of direct information exchange and coordination by connecting a set of computers among themselves, are known as *Peer-To-Peer* (P2P) networks. The exchange of information can be done by leveraging an underlying network on top of which connections, like TCP channels, are open between machines. Although a *global view* of the system could be maintained, this would require every node to know about every other peer in the system leading to problems in dynamic network environments where a high number of nodes entering and leaving the system is expected, as this information would need to be constantly updated. In order to deal with this challenge, a common solution is to allow nodes to only maintain connections to a small set of other peers, i.e., relying on a *partial view* of the system. If these connections are defined at the application level they originate structures known as *Overlay Networks* which can act as the membership protocol for a P2P system [29].

In overlay networks, a node should maintain a set of other nodes, often called neighbors, to whom connections are established. The connections between nodes then form a graph that allows messages from one node to reach, eventually with the help of other ones, any process on the network. Generally it is possible for new nodes to join a pre-existent overlay network by contacting current participants [29]. On one side, these networks can have numerous connected users cooperating, for instance, to process large amounts of data or to share information in a decentralized way, as in the BitTorrent [47] protocol. On the other side, small networks of nodes cooperating in the execution of a given task also exist. This is the case of *Internet Of Things* (IoT) sensors in a house sending information between themselves [11], the industrial machines on a factory exchanging data [46, 36, 52], decentralized solutions for the management of energy grids [27, 45, 6], or swarms of satellites in space communicating and autonomously adjusting their positions to avoid collisions [54, 59]. Many decentralized protocols, responsible for providing a given set of services to applications, rely on building overlay networks to interconnect the different nodes on a logical network, in order to exchange the required data for the system operation.

In 2020, according to Forbes [48], 59 zettabytes of data were "created, captured, copied, and consumed in the world", which compares to the 33 zettabytes two years early. The large amount of data being produced, alongside with the need for a faster and more efficient processing of such data, paved the way for a new distributed computing paradigm called *edge computing*. The idea behind edge computing is allowing computations to be executed closer to clients and outside big data center facilities [33, 38]. This approach allows for faster response times for clients and, at the same time, reduces the load induced in each component of the network, from machines in data centers, to the amount of traffic on networks [9]. As we will discuss further ahead, the use of peer-to-peer architectures that take advantage of the interconnected devices can play a significant role in the development of new approaches to edge computing. In fact, nodes in a peer-to-peer network can be considered "on the edge" of the network and closer to end-users, enabling applications that want to take advantage of the edge computing paradigm, to rely on these networks. However, contrary to peer-to-peer systems, edge computing systems can still rely on centralized infrastructures like cloud datacenters.

1.1 Motivation

Independently of what type of decentralized system is considered two main capabilities should be guaranteed to enable its operation: *i)* mechanisms for *membership management* (or membership protocols [29]) that are responsible for managing the nodes that are part of the system, the ones entering and the ones leaving either by failure or explicit request and *ii)* mechanisms for allowing nodes to communicate between them to coordinate actions and exchange information.

Applications taking advantage of the edge computing paradigm can rely on decentralized protocols as a way to obtain the membership management and communication services needed for their operation. These applications interact with decentralized protocols through the interfaces exposed by them to access the services provided and needed for the operation of applications.

Many protocols have been proposed, in particular in the context of peer-to-peer architectures, that address in different ways the challenges related with both the membership management and support for different communication primitives between peers. In particular, when considering peer-to-peer systems, overlay networks can be used, as discussed previously, to address the challenges related with the system membership. In fact, various examples of protocols relying on overlays networks have been proposed, in the context of P2P architectures, such as Chord [57], Kademlia [42], Freenet [10], HyParView [30], among others [34, 62, 23, 8, 28] with two main types arising based on the way peers are organized and connected between themselves forming the topology of the network: *structured* and *unstructured* overlay networks. The first ones are defined by enforcing specific patterns on the connections between peers, thus leading the overlay network to evolve towards a specific topology, usually more beneficial to the task that needs to be performed (e.g., nodes can form a ring, a tree, among others). The second ones do not enforce any type of network topology, allowing nodes to organize freely in a flexible way (usually randomly) [29, 38, 26].

Each protocol provides a specific set of services to applications that, in turn, rely on them for their operation by interacting through the exposed interfaces. Although the existence of multiple solutions providing similar services allows application developers to have a wide variety to choose from, this also leads to the existence of multiple interfaces to interact with, even when protocols provide similar services. In fact, the lack of a set of standard and generic abstractions to interact with the different decentralized services, leads not only to highly different (and incompatible) interfaces between similar protocols but also between distinct implementations of the same protocols.

The specific abstractions that are exposed by protocols to provide membership management or communication mechanisms bring many challenges to developers. First, the developer needs to be aware of multiple interfaces, not only when developing applications based on different services, but also when relying on different protocols, even if these conceptually provide the same abstractions [14, 3]. Also, this impacts the maintainability and improvement of systems as this heterogeneity means that the re-usability of an application code is limited when using different underlying protocols, for instance, to better cope with a particular execution environment. Moreover, this leads to a lack in the availability of generic solutions for instantiation and management of decentralized protocols, regardless of their operation or services provided, whose existence would be quite beneficial for applications relying on multiple services and protocols.

As discussed previously, the motivation for this work comes from the need of developing new generic abstractions that allow performing the operations required by applications leveraging on the edge computing approach, namely the ones related with membership management and communication. These abstractions should be related with the services provided by protocols to applications, in opposition to the current protocol-dependent ones that pose challenges when developing new systems as well as when improving and/or maintaining current ones.

1.2 Contributions

The contributions that result from this dissertation are the following:

- We provide a study of multiple protocols and approaches employed, both for membership management and communication, in the context of decentralized systems. This study has the main objective of understanding the differences and similarities between the operations provided by each protocol, thus allowing the definition of a set of generic abstractions to interact with them.

- Based on the previous study, we identified a set of decentralized services, with protocols being categorized based on the services that they provide. For each service, a generic interface was developed to allow the interaction between applications and decentralized protocols through service-based interfaces, instead of a protocol-based ones.

- We propose an architecture, integrating the generic interfaces mentioned above with a set of other components, whose objective is to provide a simple, easy-to-use, and extensible solution for applications that require the use of services provided by decentralized protocols. This solution acts as a *middleware* between applications and protocols and integrates aspects related with protocol instantiation, management, and application-protocol interaction.

- We provide a reference implementation, based in Java, of the architecture mentioned above. This reference implementation also includes the implementation of some well known decentralized protocols based on the proposed architecture, namely regarding the interfaces of the services provided. We also provide some applications that take advantage of the decentralized services considered in our solution.

- Based on the reference implementation, as well as the applications developed to take advantage of our solution, we also present and discuss an experimental evaluation whose objective is to understand the impact of our solution both in terms of application development and performance.

1.3 Research Context

The work presented here has been partially motivated by the *Trustworthy and Resilient Decentralised Intelligence for Edge Systems* (TaRDIS) [59] European project aimed at "supporting the correct and efficient development of applications for swarms and decentralized distributed systems, by combining a novel programming paradigm with a toolbox for supporting the development and executing of applications".

1.4 Document Structure

Besides this introductory chapter the dissertation is structured as follows:

- Chapter 2 presents the related work regarding the topics discussed in this dissertation. In this chapter the important concepts, necessary for a clear understanding of the work presented in this dissertation, are also presented.

- Chapter 3 details the challenges that we address in this work as well as our study regarding the services provided by decentralized protocols and the generic abstractions devised to interact with each one. An architecture for dealing with the presented challenges, based on the defined generic abstractions, is also presented in this chapter.

- In Chapter 4 a reference implementation of the solution proposed before is presented. The chapter begins with an overview of the main aspects of that implementation, followed by a detailed description of each implemented component. Then, an explanation of how our solution can be leveraged to develop applications that rely on a decentralized approach is provided together with a comparison between applications relying and not relying on the solution proposed in this dissertation.
- In Chapter 5 we present the methodology on which we relied to evaluate the proposed solution. This chapter also presents the results obtained from the evaluation, together with a discussion regarding those results.
- The document ends with Chapter 6 where we present the main conclusions as well as the work directions that may be followed in the future based on the solution proposed here.

2 Related Work

In this chapter we present relevant work that serves as a basis for this dissertation and discuss the important concepts for a clear understanding of this work.

First, the aspects related with peer-to-peer architectures (P2P) and edge computing are introduced. Then, we provide an overview of what are overlay networks, existing architectures, characteristics, and properties identifying the main positive and negative aspects of each one. We further discuss several applications that rely on decentralized protocols, based on peer-to-peer architectures and overlay networks, for their operation, like publish-subscribe or file sharing applications. A study of existent decentralized protocols is also presented as well as the frameworks for simplifying their development and the work already developed on devising generic approaches to interact with those protocols. At the end of this chapter, a summary of the contents is provided.

2.1 Edge Computing and Peer-To-Peer architectures

In this section both the edge computing paradigm and peer-to-peer architectures will be presented in-depth. We will also discuss how systems can leverage on the combination of both approaches to complete a given task.

2.1.1 Edge Computing

As discussed earlier in this document, the development of systems that rely on edge computing paradigms can be considered, nowadays, as a way to tackle multiple challenges. These include *i)* the significant increase in the amount of data that needs to be processed [48], and *ii)* the need of a fast and efficient information processing, both from end-users and systems that are required to quickly detect changes and trigger response actions [9]. This scenario raises questions related with the scalability capacity of centralized architectures and shows the necessity of removing computations from centralized environments, like data centers. A solution for these challenges can be the transfer of computations, partially or totally, towards the edge of the network near the client devices or even, if possible, to client devices, while minimizing as much as possible the dependency on centralized components or infrastructures [33, 9].

A particular example of using the network edge to accelerate computational tasks, especially when considering Internet of Things (IoT) devices, is called *Fog Computing* [38, 33, 9]. This approach relies on the use of nodes (*fog nodes*) available in the network between the clients and large scale data centers. These nodes could be used to distribute the execution of computations, considering properties like location, processing capacity, among others. Consequently, this allows for a more expedite and efficient data processing and, at the same time, reduces the amount of data that needs to be transmitted throughout the network. Various types of fog nodes can be used and, for instance, data can be analyzed first in nodes near the client to provide a quick result and, then, sent to a data center for gathering and statistical treatment [9]. This would be important in situations such as if we consider a sensor that triggers some automation based on the analysis of the information retrieved. Mechanisms relying on fog computing can also play a role in situations where privacy concerns need to be addressed by allowing critical data, such as medical information, to be analyzed in the nearest nodes with only more generic and anonymized information being sent to data centers. As a result, fog computing can even help to deal with privacy regulations at a regional level [38, 33].

An evolution of fog computing consists in performing the computations on the actual devices that retrieved the information by taking advantage of their embedded processing capabilities. This evolution is called *Mist Computing* [40] and, by itself, does not imply that some data cannot be sent to fog nodes and/or central processing facilities for further analysis and aggregation.

The distribution of computation between machines on the edge of the network relying on decentralized architectures is not something new as it has been employed for decades. An example of this is the SETI@Home [1] project, which allows computers to cooperate in processing large amounts of scientific data with the objective of studying the existence of extraterrestrial life by analyzing radio signals retrieved from space. This project is an interesting case study as it effectively uses end-user devices, like computers, to help process large amounts of data without the need of central processing. In this case a central server was effectively used for distributing the computations between client devices as the data is retrieved by radio telescopes and, as so, a central point of operation needs to exist in order to provide the information. Still, this is an example of how devices on the edge of the network can be used, relying on decentralized mechanisms, to process large amounts of information. When we consider situations where data sources are in itself decentralized (like in the case of IoT devices), this approach can even become more interesting. Notably, the computation model employed in SETI@Home (and similar projects) is composed of small and fully independent computational tasks (named *embarrassingly parallel*). More complex computations require more complex approaches in order to be executed on the edge.

One of the challenges when considering edge computing paradigms is related with the heterogeneity that may exist due to significant differences between the characteristics of nodes in terms of processing power, storage capacity, network capabilities, reliability, among others [33, 40]. Some studies [33] even consider the separation of the network nodes into distinct categories based on their characteristics, forming multiple levels of nodes. These may vary from the ones "in the center" of the network, with high processing capacity and availability, like data centers, to devices on the edge, with limited availability and mixed performance, like IoT sensors. To allow nodes to be split into levels, as presented before, authors suggest the study of a set of characteristics such as processing and storage capabilities or availability guarantees. The expected increase of devices more close to clients emphasizes even more the need to develop methods that can cope and be performant even in presence of significant heterogeneity as the computational resources can be limited.

The heterogeneity between the conditions of different networks discussed before, depending on the characteristics of the nodes present in each one, supports the existence of multiple decentralized protocols providing the same services to applications. This way, applications can benefit from relying on a decentralized service provided by a protocol more performant on the expected network conditions.

2.1.2 Peer-To-Peer architectures

Peer-to-peer architectures allow processes to interact and cooperate directly without the need for a centralized point of coordination. This is achieved by having nodes directly connected to a set of known neighbors with which they cooperate to perform a given operation, thereby removing the need to contact a central point, like a server. This connection between peers generally happens on top of an existing network, like the Internet, effectively creating what is called an *overlay network*. As we will discuss later in this chapter, in most architectures a peer does not even need to know all the other network participants, therefore relying on a partial view of the network [29, 38]. In these solutions, although a node is only connected to a subset of network participants, it should be able to communicate throughout the entire network due to mechanisms such as having nodes relaying messages to their respective neighbors, and so on until reaching the intended destination. One of the most well known protocols leveraging on a P2P architecture is the BitTorrent protocol in which peers communicate between themselves to perform file transfers in a distributed manner, with files being divided into chunks that can then be traded between peers and combined to reconstruct the complete file [47, 38].

Peer-to-peer networks have many advantages when compared with centralized solutions, namely when considering that they can overcome challenges related with the existence of a single point of failure/attack, thus possibly improving reliability and security. Privacy questions can also be tackled as their decentralized nature allows for data to be dispersed and processed along the network instead of doing so using machines in central locations, like data centers [11]. Some P2P networks, like the one proposed in Freenet [10], even strive for anonymization of the data exchanged between peers, making it harder to know which peers have a specific segment of data or which ones are looking for it. On the other hand not relying on a central server means that for creating, maintaining, and operating a peer-to-peer system, specialized mechanisms for allowing peer management and communication are required to exist as they need to be able to manage a possibly large number of participants and allow for communication between them. In fact, overlay networks can be effectively considered as a mechanism for management of peers to be used in P2P architectures [29].

Napster is considered the pioneer in peer-to-peer file sharing architectures. Created in 1999, the system brought the idea of a set of users sharing content stored on their own machines, in a decentralized manner, by cooperating with each other over the network. This way, peers could directly download the content from the host machines without the need for a central storage server [29]. Although this approach effectively allowed for a decentralized file-sharing system it employed a centralized component, responsible for providing the index of contents available in the network and their location, effectively materializing a centralized *resource location* service. Consequently, the Napster approach relied on the use of a centralized directory server that should be contacted to find which peers in the network had a specific resource, coordinate with them, and obtain the files [8]. Napster was closed due to legal reasons related to copyright and, although the legal matters of this closure are outside the scope of this document, the technical aspects that allowed this outcome should be considered. The use of a centralized server was effectively the reason that enabled the closure, as it represented a central point of failure and, if offline, the system was not capable of operating properly with users being unable to discover other peers storing the desired content [8, 7].

It is when the P2P mechanisms for peer management and communication are considered, that a relation can be established between edge computing and peer-to-peer systems. On one hand, as seen before, edge computing allows for computations to be transferred from central points, like data centers, to machines near the clients or even into the client devices. On the other hand, peer-to-peer networks allow for multiple devices to be connected without the need of a centralized architecture. We can now understand why peer-to-peer architectures can contribute to the edge computing paradigm by taking advantage of the solutions for both membership management and communication as a way to interconnect devices "on the edge" of the network. In fact, the distributed nature of devices on the edge matches the principles behind P2P systems and overlay networks as they are "highly decentralized, robust and can be easily adapted to promote hierarchical topologies" [33] like the ones considered in fog [9] or mist computing [40].

It is worth noting, however, that multiple protocols for maintaining peer-to-peer systems exist nowadays [10, 42, 57, 23, 8, 47, 43, 58, 28, 30, 62], each one employing its own mechanisms for managing the network of nodes (i.e., the overlay network) and allowing information to be exchanged between them. Some of these protocols provide the same services, but relying on different approaches for both membership management and communication, each one more suitable to be used on a specific set of conditions. Furthermore, each protocol exposes a specific interface to allow applications to interact with the services provided [14, 3], originating a heterogeneous set of interfaces exposed by protocols providing the same services, and sometimes, even when comparing multiple implementations of the same protocol.

This heterogeneity is what motivates the importance of developing generic abstractions for allowing the use of different solutions, considering the ones already employed in the context of P2P, under the same set of service-based generic programming interfaces. Moreover, applications that rely on decentralized approaches, in particular those who need to leverage on multiple protocols for different services, would benefit from the development of mechanisms to simplify the management of decentralized protocols, independently of their actual implementation or services provided.

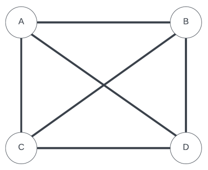
2.2 Overlay Networks

In this section a more in depth study of overlay networks will be conducted regarding their main characteristics and properties, in particular when considering the network view and structure. First an overview of overlay networks will be carried out regarding their basic concepts and main properties, followed by a comparison between global and partial views. Then, both structured and unstructured overlay networks will be presented, as well as a comparison between the positive and negative aspects of each one. An overview of decentralized communication protocols will also be provided in this section.

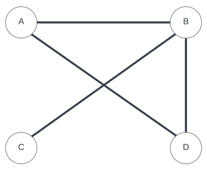
As described before in this document, an *Overlay Network* can be defined as a set of nodes connected between themselves on top of an underlying network (hence the name *overlay*). We can consider that each node represents a process and maintains a set of other nodes as its neighbors. In fact, in many cases overlay networks rely on the Internet as the underlying network and processes open TCP connections between themselves creating links [38, 30, 34]. In an overlay network nodes can be connected to other ones independently of their physical location or the physical links that exist between them. As an example, in a network operating on top of the Internet, a node located in Portugal can have neighbors located in Japan because, although a direct connection may not exist, logical links are created on top of the underlying network. Many P2P systems rely on overlays as membership management mechanism because this type of networks enables processes to be effectively interconnected with the other nodes on the network acting as peers in the system [29]. In the literature, overlay networks are often described as graphs $G = (V, E)$ where the vertices of the graph correspond to the nodes and the edges correspond to the connections established between them [32, 31, 30, 26, 8].

2.2.1 Global view vs Partial view

Before presenting a more in-depth study of overlay networks, it is important to characterize them based on the view that each node has of the system membership. Therefore, P2P architectures can be characterized, based on the connections established by each node participating in the system with the other ones, in *Global View* or *Partial View* architectures. Each one of these approaches has a profound impact not only on the mechanisms that need to be employed for membership management and communication but also on the performance of the network on a given operational scenario.



(a) Global View



(b) Partial View

Figure 2.1: Comparison between global view and partial view on a P2P system network

2.2.1.1 Global View

Systems relying on a global view are characterized by having each node know all other nodes in the network, therefore, having "access to the full membership information" [29], i.e., if we consider Π as the set of all network participants, the set of neighbors of a node A is $\Pi \setminus \{A\}$. A global network view allows every node to directly communicate with any other node present in the network, thus ensuring a very efficient network topology for both direct communication or broadcast of messages. In fact, if we consider broadcast and a global view, a node is only required to send a message to all its neighbors to accomplish broadcast.

Simple membership management protocols can also be used for maintaining overlay networks based on a global view. Consider this example: when a new peer (A) joins the network, by connecting to an already participating process (B), A can receive all the neighbors of B and establish connections with them. These connections are performed by relying on the underlying network, using a protocol like TCP [38], gaining a global view of the system and letting the other nodes know about the new one joining. On the other side, when a node leaves the network, the neighbors only need to detect the fault — or be informed by the leaving node if possible — and no longer consider it as neighbor.

The main disadvantage of a global view is related with the high load put on the network participants due to the number of neighbors that each node needs to manage, especially in situations where the network has more than a few peers and/or a significant level of *churn* is expected. The churn is a term related with the number of nodes entering and leaving a P2P network at a given time [29]. This problem leads to architectures relying on a global view only being useful in situations where both the number of peers and the churn effect are expected to be low, like in local networks built in controlled environments.

2.2.1.2 Partial View

Systems relying on a partial view [29, 32] are characterized by having each node to be aware of only a small portion of the peers in the network. This means that a node A may not be able to communicate directly, i.e., using a direct link such as a TCP channel [38, 30, 34], with a node B also participating in the network. In these architectures, nodes need to rely instead on other participants that should relay information — possibly multiple times — until it reaches the intended destination. When considering overlay networks, relying on partial views is the most common architecture as it solves the scalability and capability issues related with keeping track of the entire membership that arise when considering global views. On the other hand, this approach means that more sophisticated and complex membership management mechanisms are needed in order to define which (and how many) nodes should be chosen as neighbors, how changes in the overlay membership are handled when a new node wants to join or a participant becomes disconnected, as well as how nodes can communicate efficiently with others throughout the network. Many approaches for dealing with these aspects will be considered when studying the different overlay network solutions [30, 34, 42, 57, 8, 10].

Figure 2.1 shows the comparison between a global view and a partial view P2P systems, both with the same set of nodes $V = \{A, B, C, D\}$ but with a different set of edges. Note that the architecture of the global view is, in fact, represented as a complete graph.

2.2.2 Overlays Properties

Below we present some important properties of overlay networks, namely *connectivity* and *accuracy*, that have a direct impact on the correctness of operation, and *network diameter*, *clustering*, and *node degree*, that should also be considered to improve performance.

Connectivity and Accuracy In order to enable the correct operation of an overlay network both *connectivity* and *accuracy* should be guaranteed [29, 30]. On one hand, connectivity is related with the ability of any node to exchange information with any other, by leveraging on the overlay links established between them, which should guarantee that exists "at least one path from each node to all other nodes" [29]. If a node, or a set of nodes, loses connectivity this means that it is no longer possible to contact these peers using the network. On another hand, peers should also eventually drop existent connections to failed nodes in the network to guarantee higher accuracy, i.e., that a node does not have a significant number of neighbors that are no longer available. The accuracy value can be calculated, per node, as the division between the number of available neighbors and the total number of neighbors and, at a network scale, as the average of accuracies. In fact a lower accuracy value can have impact on mechanisms such as random walks, described in Section 2.3.1, causing messages to be sent more often to neighbors that are not available [29].

Network Diameter, Clustering, and Node Degree Some aspects should be considered when studying overlay networks with the objective of improving efficiency, namely *network diameter*, *clustering*, and *node degree* [29, 30]. The network diameter is related with the lengths of the paths between two peers and, therefore, networks should try to maintain a small diameter in order to make it easier for messages to reach the entire network. The network diameter can be considered as the average of path lengths between peers. The degree of a node is related with the number of neighbors that a node have, and the network should ensure that a uniform degree is maintained in order to not have a significant discrepancy between the number of neighbors of nodes, thus originating load imbalance. Clustering is related with the heterogeneity of neighbors between processes as overlays should strive to keep a low clustering level by maintaining significant differences in neighbor sets between peers, i.e., heterogeneity should exist between the neighbors of different nodes. The *clustering coefficient* can be defined, per node, as "the number of edges between that node's neighbors divided by the maximum possible number of edges across those neighbors" [30]. The average of clustering coefficients allows the evaluation of the network clustering level with higher values (between 0 and 1) meaning higher levels of message redundancy and high probability of network segments isolation, therefore having a negative impact on fault-tolerance [30].

2.2.3 Overlays Topology

When studying the characteristics of overlay networks two main types of network topologies arise: *structured* and *unstructured* overlay networks. This characterization is related with how the peers are linked together between themselves, which has a significant impact on which operations can leverage on the network and how they are implemented. Both of these categories will be discussed in this section.

It is important to note, at this point, that in this section only networks based on partial views will be considered because when a global view approach is used the questions related with the topology of the network no longer matter. In fact, when considering a global view, the only possible topology is one with all nodes connected between themselves in a complete graph and always able to communicate over direct links. However, nodes on an overlay network based on a partial view can have, under certain conditions, a global view of the network. This could happen, for instance, if the number of peers participating in the network is low (as in the special case of a single node).

2.2.3.1 Structured Overlay Networks

Structured overlay networks can be described by relying on peer management protocols that strive to maintain a specific network topology with the objective of making a certain operation, generally search within the network, more efficient. In structured overlays the set of links maintained between processes on the network are not random, and those links are created (or discarded) with the objective of maintaining a certain topology in the structure of the network, such as a ring, a tree, among others [38, 57, 42]. This specific topology should be chosen taking into consideration the expected operation of the overlay, network conditions and/or peer characteristics with the objective of ensuring a reliable and efficient operation.

One of the most common situations for the use of structured overlay networks, in the context of P2P, is resource location by relying on *Distributed Hash Tables* (DHTs) [29, 38]. A DHT works, on a high level view, in the same way as a common hash table, i.e., data is organized in key-value pairs where the key is composed by a hash of some information correlated to the data (or the data itself) and the value represents the information to store. The difference, when considering DHTs, relies on how key-value pairs are actually stored in the nodes using a distributed approach. Generally the set of values S, that can be issued by a hash function h, is distributed between all nodes present in the network becoming each node responsible for a subset of S (which could be chosen, as an example, taking into consideration the identifiers of the nodes). Then, when data related with a key k needs to be located, a node can perform a hash of the key $h(k)$ and contact the peer which is expected to be responsible for storing the information [38]. When a new node joins the network the subset of values that will become his responsibility should be computed and, possibly, information can be transferred from other nodes to it [57].

As discussed before, when relying on structured overlay networks for implementing DHTs, the topology of the network should primarily ensure the efficiency of the search operation, i.e., that a node searching for a key could as efficiently as possible contact the node responsible for that key. Two well known P2P systems relying on structured overlays for providing a DHT-based key-value pairs storage are Chord [57] and Kademlia [42].

2.2.3.2 Unstructured Overlay Networks

Contrary to structured overlays, *unstructured overlay networks* do not strive to impose a specific topology to the connections performed between peers participating in the network [38, 29]. In this case the links maintained by nodes with their neighbors depend on many factors and the topology of the network cannot be inferred *a priori*. These factors include the moment when nodes joined the network, the peer to which a node connects when joining, or even properties of nodes such as location, processing capacity, network connection, reliability, among other aspects.

Generally when considering unstructured overlays, a new process joins the network membership by contacting an already participant one who is then responsible for enabling the joining node to obtain a set of neighbors. This could be done, for instance, by forwarding the information about the join throughout the network to allow current participants to establish connections with the new node [30, 34].

The membership management mechanisms of unstructured overlays are expected to be more simple and less computationally expensive, when compared to the structured counterparts, as a specific topology does not need to be enforced [29] and, as so, each node is responsible for choosing the set of neighbors based on previously defined heuristics and possibly also taking into consideration constraints regarding a minimum and maximum number of neighbors. In fact, it is expected that when a node needs to add a new neighbor, due to not having a sufficient amount of neighbors or a connection failure, it should try to connect with other processes in the network, obtained through its neighbors, choose the best one according to the heuristics and attempt to establish a connection. In some cases nodes can even drop currently available neighbors in order to substitute them for newer ones [30, 8, 62].

Generally we can consider all nodes participating in an overlay network as contributing equally, nevertheless, there are architectures where a few nodes contribute, in some form, more that others to the network operation.

Super-Peers A relevant architecture commonly used in unstructured overlays is related with the definition of special nodes called *Super-Peers*. This approach is leveraged in many overlay designs, like the ones proposed in Overnesia [34] as well as (newer versions of) Gnutella [5, 25]. The super-peers approach can be defined by the existence of a specific type of peers in the network that have some set of characteristics that allow them to be "promoted", based on aspects defined by the network implementation [29]. These characteristics are often related with processing capability, storage and/or reliability. Generally, the idea behind super-peers relies on biasing network participants to prefer connections with a certain set of peers to help build a more stable and efficient network. It is important to note that the use of this mechanism does not impose a topology on the network, thus not forming a structured overlay, but instead the mechanisms for choosing a super-peer as neighbor are inserted into the heuristics on which nodes rely to choose a neighbor over others [8]. As an example, when considering protocols performing search operations, we can think of super-peers as nodes expected to be able to store more information than others, which explains the preference for performing more connections to this type of peers. Although relying on a super-peer based approach can improve efficiency, this strategy puts in question the uniformity of node degrees discussed in Section 2.2.2 and, in some situations, can lead to negative consequences as a failure in a super-peer will produce a major effect on the network when compared with a common node [34]. The definition of the candidates to super-peers also constitutes a challenge as the characteristics might not be trivial to define [29].

2.2.4 Decentralized Communication Protocols

If a system relies on a centralized approach processes are only required to contact a central node, e.g., a server, in order to obtain the desired resources or execute a given operation. On the other hand, if a decentralized P2P system is in place, specially considering architectures leveraging on overlay networks, more complex communication protocols are required.

In the case of P2P systems relying on structured overlay networks the communication is often performed by contacting directly the node responsible for the identifier of a given resource [29], as discussed in Section 2.2.3.1. However, this does not mean that a (generally) small set of neighbors does not need to be contacted in order to reach the node responsible for the resource, as a partial view of the system is maintained, hence requiring mechanisms, like the ones proposed in Kademlia or Chord, to find the desired node, e.g., the node with the nearest identifier to the key [29, 57, 42].

The communication protocols applied to P2P systems leveraging on unstructured overlays are much different from the ones employed in structured overlays, as the absence of a specific topology makes it harder for one peer to contact another one, responsible for providing a given resource. In many situations systems rely on *Gossip-based Dissemination Protocols* [29] where peers collaborate in exchanging messages throughout the network by sending them consecutively to f neighbors, which should then relay the message again if they have not received it yet. The parameter f is known as *fanout* and in many systems is configurable, with higher fanout values leading to faster message dissemination in exchange for more redundancy and load in the network [29].

2.2.5 Structured *versus* Unstructured Overlay Networks

As presented in Section 2.2.3.1, structured overlay networks have the main advantage of allowing an efficient discovery of a node or resource if the information about the full identifier is known, however this also brings some disadvantages. Chawathe et al. discuss in [8] some challenges of relying on structured overlay networks. First, the effort that is required for maintaining the expected topology and perform the management of the information stored is expected to be higher. This is even more important when considering large and/or dynamic networks, with a significant churn effect due to a high number of nodes continuously joining and leaving the network. Furthermore, another challenge of the DHTs is related with searching for a resource based on a partial key instead of the complete one (known as *exact-match queries*). Consider this example: the hash of "NOVA School of Science" can — and will most likely — be completely different from "NOVA School of Science and Technology" and if the first expression is used, on a DHT, to search for a resource whose key is, in fact, the second one the resource would not be found.

Unstructured overlay networks, on the other hand, are more suitable for situations where a high number of exact-match queries are not expected, and the searches are mainly done using partial query expressions. This is explained by the use of gossip-based dissemination protocols, where the search for a given resource is made locally at each node when it receives the query, with all matching results being sent to the source node [8]. Another advantage of the unstructured overlays approach is related with the membership management mechanisms, as these are expected to require less computational resources [29], due to not imposing a specific topology. This leads to networks able to work more efficiently in highly dynamic environments with significant levels of churn and/or a high number of peers.

The main disadvantages of an approach based on unstructured overlays are related with the expected high level of communication in the network due to the possibility of a significant number of messages being retransmitted throughout the network. Even though adequately configured mechanisms, such as *time-to-live* (TTL) values, discussed in detail in Section 2.3.1, and fanout parameters can mitigate this problem, if incorrectly configured or implemented the network can become useless due to a high load. An additional possibility is relying on a super-peer mechanism, described in Section 2.2.3.2, which contributes to the mitigation of this challenge by reducing the traffic in the network [29].

Although more resilient to unstable network conditions and churn than structured overlays, in unstructured overlays the mechanisms for neighbor selection should also be considered to not create an excess of dynamism in the network, therefore undermining their operation. One other problem arises when using unstructured overlay networks to store resources: it is not guaranteed that a resource is located even though it is present in the network. This could happen when a combination of the defined parameters like TTL, network diameter, topology, and/or randomly chosen walks lead a query returning no result, although a matching resource exists in the network. This problem has more importance if a high level of *needles* [8], i.e., lower replicated resources, are expected to be present in the network. In fact, the replication of data is not only important in unstructured overlays to guarantee that information is not lost even when nodes become disconnected, but also to improve the efficiency and recall of search operations.

2.3 Applications relying on Peer-To-Peer protocols

Multiple applications can be built by relying on peer-to-peer systems taking advantage of overlay networks to manage the network membership. In this section some of the most common types of applications that leverage on the use of P2P architectures will be described. These types of applications provide mechanisms such as *Resource Location*, *Broadcast*, *Publish-Subscribe*, *Distributed File Sharing*, and *Distributed Computation*.[1]

2.3.1 Resource Location

This is the most commonly described application when considering peer-to-peer architectures and this could be explained by the fact that the first widely used P2P systems, like the already presented Napster and Gnutella networks, were designed exactly for this purpose as explained before in this document. Generally speaking *Resource Location* consists on relying on a network to find a given resource which can be, for instance, a file, a machine with specific resources or a node with a given identifier.

[1]While here we refer these as applications, in some contexts like [29] these are referred as services.

As stated in Section 2.2.3 an application providing resource location can leverage on P2P systems relying on structured or unstructured overlay networks. In the first case, consistent hashing [57] is generally employed to find a resource to which the full identifier is known while, on the second, gossip-based dissemination techniques, like *flooding* or *random walks* [38, 29], are usually employed to allow the retrieval of a resource even without knowledge of its full identifier.

Flooding Flooding consists on sending a query throughout the entire network by forwarding it to all neighbors that should then forward the query again. This approach can put a high level of load into the network so, to deal with this situation, mechanisms like *Flooding with Limited Horizon* [29] might need to be put in place. This mechanism relies on adding a time-to-live (TTL) value to queries that, when reached, leads the query to be discarded. If the matching resource is found, the node that contains the resource can reply directly to the origin node or send the result back through the network peers [38, 10]. It is important to note that even if a matching resource is found, queries may continue to cross the network leading to the possibility of obtaining multiple results [8].

Random walks Random walks, on the other hand, consist on a node sending a query only to a small set of neighbors (or even one), chosen at random. The nodes receiving the query then verify if the requested resource is present and, if successful, reply to the origin node as described before in the flooding method. When a node receives a query, it should relay it again using the same procedure. Although originating much less stress on the network when compared to flooding, random walks also need to use a TTL-like mechanism to stop queries from continuing indefinitely and end up in a situation where too much load is put on the network [38, 8].

2.3.2 Broadcast

Broadcast consists, as the name suggests, in disseminating data throughout an entire network in order to reach every participating peer. Although questions related with the effort put on the network should be considered, broadcasting applications can leverage on the use of underlying P2P architectures. A simple implementation of broadcast can be done by relying on the flooding mechanisms already discussed in Section 2.3.1 [39]. Therefore, if all messages that are flooded through the network contain a unique identifier, like an UUID, the participating nodes are able to verify if a message was already received by them and, if not, deliver it locally and retransmit throughout all neighbors [29].

When studying broadcast services provided on top of peer-to-peer systems, the topology of the network can have a significant impact on the operation. As an example, a tree based topology would allow a reduction in the load put on the network, as message redundancy is avoided due to the absence of cycles, however, a failure on a link could lead to a network partition [39]. An interesting solution for building a tree for broadcast and, at the same time, increase fault tolerance is relying on protocols such as Plumtree [28] which are able to maintain a tree structure, even in presence of node failures, on top of a protocol relying on an unstructured overlay acting as *peer sampling* service [29, 38], like the HyParView [30] protocol described in Section 2.4.

Applications may also implement *Multicast* services aimed at delivering information only to a subset of nodes. As in broadcast, a multicast service can be built on top of overlay networks, with Steen and Tanenbaum [39] proposing a solution where nodes maintain multiple sets of neighbors, each one from a different multicast group. This way messages can be flooded only in the context of each group that acts as a separate overlay. Additionally, protocols like Plumtree can also be considered in these situations through the construction of multiple trees (each one maintained by a Plumtree instance), instead of only one.

2.3.3 Publish-Subscribe

Publish-Subscribe services are nowadays widely used in a variety of situations with companies like Google [64] or Amazon [63] offering this type of service through their respective cloud computing platforms. Publish-subscribe systems allow for an asynchronous form of event-driven communication where information consumers subscribe to a set of topics, related with the data they wish to receive and, on the other hand, producers send messages tagged with those topics, that should only be delivered to participants in the network that subscribed them [38, 15].

In some publish-subscribe systems it is even possible for a consumer to receive information sent while it was disconnected from the network when it becomes online again [15]. This can be important in situations where it is expected that a node does not always have a stable network connection, such as vehicles in an IoT network, thus leading to possible reliability issues that can be tackled with this approach.

A simple approach to publish-subscribe based on P2P networks is to rely on broadcast operations. In fact, if a message is disseminated throughout an entire network with information about the topic (or the set of topics) related to it, each node can then decide if it should deliver the information received by comparing the message topics with a local subscriptions set. An advantage of this approach is related with the capability of allowing the subscription and unsubscription operations to be done locally, without the need for any exchange of information between network participants. However, it could lead to a significant load on the network if a high number of messages is sent, which is expected when considering publish-subscribe services. Moreover, this will generate a high quantity of unuseful traffic related with the retransmission of messages that will not be delivered by most nodes because the topic was not subscribed. Nevertheless, other approaches exist for implementing publish-subscribe services on top of P2P networks like Scribe, "a large-scale event notification infrastructure for topic-based publish-subscribe applications" [56] which relies on an overlay network called Pastry [55].

2.3.4 Distributed File Sharing

Peer-to-peer protocols, such as BitTorrent, are also widely used to build distributed file sharing applications. The BitTorrent protocol allows decentralized downloads of files by providing mechanisms to clients that enables them to transfer chunks from other network participants and then merge them together to obtain the complete file. [47, 38]

Protocols related with distributed file sharing provided over P2P architectures should strive to ensure that users are required to collaborate in sharing information and discourage a practice called *free riding* [38, 25], where a user takes advantage of the network to obtain resources but does not contribute. This practice can lead to a significant imbalance on the system by creating situations where, although having a high number of data requests, the number of nodes effectively providing resources is low. The free riding practice is a challenge which was already studied in the context of the Gnutella network in [25]. The BitTorrent protocol limits this problem by imposing a mechanism that requires users to effectively exchange chunks of information who own at approximately the same rate at which data is being received [38].

2.3.5 Distributed Computation

The distribution of computation between machines with the objective of parallelizing data processing, in an approach commonly called *Grid Computing*, is one more example of an application where the use of P2P architectures can be leveraged. In fact, the SETI@Home [1] infrastructure already discussed in Section 2.1 provides an example of how nodes connected on a peer-to-peer network can contribute to the parallel processing of data. However, this service relied only partially on a decentralized architecture as a central server was needed for both data distribution and post-processed data retrieval.

Foster and Iamnitchi [17] present a comparison between the Grid Computing and P2P paradigms and, according to the authors, although the main objective of both approaches is related with "resource sharing within virtual communities" and both rely on the same approach, in some aspects differences exist between the two. These differences are related with the computational power of nodes, which is expected to be higher in the case of Grid Computing, or the mechanisms in place to deal with a high number of nodes, lower guarantees of availability or enforcing a fair usage of the system by clients, which are more developed in the context of the P2P paradigm.

However, when considering both the increase in the size of computer grids and the increase in computational performance of day-to-day devices, both approaches seem to converge into each other. This leads the use of P2P mechanisms to handle grids of machines executing distributed computations to be considered as a possibility in some situations. An example of this overlapping between both approaches is the work by Verbeke et al. [61] where a framework for enabling Grid Computing on top of a P2P architecture, operating on a heterogeneous environment is presented.

2.4 Examples of Peer-To-Peer protocols

In this section some examples of peer-to-peer protocols, relying on overlay networks, will be discussed. The study of the protocols presented here, namely regarding their operation, services provided, and properties, served as starting point for the solution presented in Chapter 3.

Kademlia Kademlia [42] is one of the most well known P2P protocols reliant on structured overlay networks and is commonly used to implement DHTs. The system relies on a network of peers, each one with an assigned identifier, that store neighbor nodes on buckets, called *k-buckets* and seen as leaves in a binary tree. The routing operation, as well as the storage of the nodes in the k-buckets, is based on a XOR distance metric between the binary representation of the identifiers. When a node wants to contact other node with (or near) a given identifier, it first contacts α peers (where α is a configurable parameter) from the nearest k-bucket, based on a XOR operation, and should receive, from the contacted nodes, the closest k known nodes on which the operation is then repeated until no other node with the nearest identifier is received. This solution allows for queries in Kademlia to approach, step by step, a node with a given identifier or the closest one which could, for instance, be responsible for storing a value related with that identifier. An interesting scenario for using Kademlia are situations where the interaction of a node with the closest ones, regarding the XOR metric applied to the identifiers, is more frequent as peers are expected to have more information about the network topology near their identifier due to the way nodes are split between buckets.

Chord The Chord protocol [57] is, alongside with Kademlia, one of the most well known P2P protocols leveraging on the use of structured overlay networks. The main service provided by Chord is the same as the one provided by Kademlia, i.e., the capability of finding the nearest node to a given identifier, although one of the differences is related with the capacity of Kademlia for providing the set of the closest nodes, while Chord is only expected to provide the closest one. Like Kademlia, the Chord protocol is generally used to implement DHTs with each resource being stored on the node with the closest identifier, leveraging on consistent hashing. To operate correctly the Chord protocol employs mechanisms to maintain, even in the presence of failures, a ring-shaped structure where each node keeps information about its successor, i.e., the network node with the next key on the identifier space. When implementing a DHT based on Chord, the successor of a resource identifier is also the node that should store the resource. For performance reasons, each node also maintains a set of entries called *fingers* to expedite the execution of queries by contacting directly the nearest finger to an identifier, therefore avoiding, in most situations, the need for iterating through consecutive nodes.

Kelips The Kelips [23] protocol is a decentralized protocol for building a Distributed Hash Table (or DHT). As in the case of other DHTs, like Kademlia or Chord, the Kelips protocol relies on consistent hashing to attribute a given identifier to nodes and resources stored. Then, nodes are put together in *affinity groups* with other nearest nodes by considering the identifiers. Each node maintains information about nodes and resources stored in its affinity group as well as some additional information about contact nodes from other groups. The information maintained at a node, in Kelips, is associated with a heartbeat count that, if not updated, will lead to the deletion of the entry. The updates are performed through the dissemination of messages throughout the affinity group but also between affinity groups by relying on the contacts known, by each node, in other groups. The mechanisms employed in Kelips allow for efficient resource lookup and store operations, as a node who wants to obtain a resource just needs to perform a hash on the resource identifier and send the query to a contact within the respective affinity group. The contact will then look for the resource in the index containing the resources stored on the affinity group and return the address of the node that effectively stores the requested resource. The operation to add a new file is equivalent to the one described before but the node, in the group, responsible for storing the new resource will be chosen at random.

Gnutella After the already discussed Napster approach, Gnutella was devised as a fully decentralized file-sharing peer-to-peer protocol taking advantage of a flooding technique with limited scope [8] as a way to eliminate the need for a centralized search model. In the first versions of Gnutella a simple approach was implemented where each network node formed links, using TCP connections, to a set of other nodes and resource location was performed by flooding queries throughout the network, expecting nodes with matching resources to reply directly, with the requested data, to the nodes that acted as origin of the queries [53]. This approach presented scalability issues as the usage increased, leading the Gnutella network to became flooded with queries and, although a solution based on time-to-live (TTL) values was put in place this, on the other hand, lead to queries failing to obtain the expected results as only a few nodes were visited [25, 8].

Gia Gia [8] is a file-sharing P2P system reliant on an unstructured overlay network developed with the objective of improving the mechanisms already proposed in Gnutella. Gia relies on biased random walks (an approach also described in [29]) to find resources, leading nodes to choose higher capacity peers when forwarding a query, although also relying on a token based flow control mechanism where a node that handles more queries earns the right, from its neighbors, to issue a higher number of queries. In the approach followed by Gia, in addition to a TTL mechanism, a limitation on the maximum responses that a query should originate is also put in place to tackle the load induced by the possible existence of multiple results. Other solutions are also used in the overlay that supports Gia, such as a mechanism for topology adaptation that strives to put nodes with low capacity near higher capacity ones, with the objective of increasing connectivity on high capacity nodes, effectively following a super-peer approach.

Cyclon Cyclon [62] is a peer-to-peer protocol devised with the objective of managing an unstructured overlay network, by maintaining a partial view of the network in each node. To this end, Cyclon employs a shuffle mechanism that forces nodes to periodically exchange messages containing a subset of the nodes from their views as well as updating the local views with the information received from incoming messages. The protocol aims to be robust in the presence of node failures and strives to maintain a set of beneficial properties in the graph formed by the overlay network, like low diameter and clustering, as well as symmetric node degrees. In fact, Cyclon can act as a *peer sampling* protocol, i.e., a protocol that is responsible for providing a sample of peers that can then be used by other protocols for their operation [29, 38]. This sample is obtained through the partial view that is maintained by each node, and can be important for the operation of other protocols, such as the ones responsible for providing message dissemination services on top of an overlay network. An interesting aspect of Cyclon is the dynamic sample of nodes provided by the protocol, that changes over time due to the cyclic strategy [31] employed through the exchange of periodic messages to update the views of the nodes.

HyParView HyParView [30] is a protocol responsible for maintaining an unstructured overlay network devised with the objective of supporting a reliable broadcast of messages in a gossip-based approach. Unlike other peer sampling protocols, like Cyclon, which build an overlay network by relying on a single partial view composed by the known neighbors, the HyParView overlay relies on two distinct partial views: a smaller one known as *active view*, from which the sample of peers is effectively obtained, and a larger one, called *passive view*, used to replace the nodes of the first one when needed. The active view relies on a reactive strategy with nodes being replaced only when failures occur, while the passive view is updated by a cyclic strategy with each node periodically exchanging information about its known set of peers, from both views, with a member of the active view. The main advantages of HyParView are related with its high reliability even in the presence of a high level of node failures and the stability of the active view, due to the reactive approach. This stability is important for the operation of some protocols that may rely on the peer sampling service provided, like the Plumtree protocol described below.

Plumtree The Plumtree [28] protocol is a decentralized protocol responsible for the dissemination of data throughout a network. Plumtree employs an interesting approach as it builds a tree on top of an unstructured overlay network maintained by an underlying peer sampling protocol, like HyParView. This makes the dissemination of messages more efficient, due to the structured approach where messages do not need to be flooded to all neighbors of each node while, at the same time, provides fault tolerance even in conditions where a significant level of churn is expected. This is due to the underlying unstructured overlay network, which is expected to be more resilient in unstable conditions when compared to a structured one. For maintaining the tree structure the Plumtree protocol relies on the messages received from neighbor nodes to detect cycles in the graph formed by the underlying network. Therefore, if a message is received by a node from more than one neighbor, a cycle is detected and one of the connections is considered redundant, leading the following messages to not being transmitted through that connection. On the opposite side, nodes periodically exchange messages, through flooding with all their neighbors, containing the identifiers of the messages previously received, allowing a node to detect if a message was lost. When a node discovers that a disseminated message was not received, it can request the message from the (active) neighbor from whom it discovered the absence of the message. Additionally, the node will also connect to the active one, therefore healing the tree.

Freenet Freenet [10] is a peer-to-peer protocol aimed at providing decentralized file storage with privacy guarantees. The objective of the protocol is to create a network able to store resources and efficiently route queries to obtain them while, at the same time, providing privacy guarantees for the nodes that store or request a given resource. The Freenet protocol relies not only on hashing mechanisms to define the keys for the resources, but also on a set of cryptographic primitives to guarantee privacy and security related properties, like the integrity of the data. To perform the retrieval of a resource, queries containing the resource key are routed through the network by relying on the routing table of each node. The routing tables maintain information about the keys stored in the local node or in other nodes, enabling a search for a given key by contacting the node related with the nearest key available on the routing table, if the key is not contained in the local node. Then, the same operation is performed by the next nodes until the resource is found or a maximum hops limit is reached. When a resource is found, it is returned to the node who performed the request through the inverse path, which enable the nodes in between to cache the resource. To ensure privacy, the Freenet protocol relies on a set of special mechanisms such as the capability of nodes to hide the origin of a resource by claiming themselves as the origin, even though the resource was obtained from other node. Moreover, in order to tackle attackers that might try to introduce excessive load on the network, nodes can unilaterally decide to drop a given query if an excessive maximum hops value is detected.

2.5 Frameworks for developing Peer-To-Peer protocols

With the objective of simplifying the development of peer-to-peer protocols multiple frameworks were already developed. Examples of these frameworks include Appia [44], Libp2p [49], and Babel [19]. These frameworks generally provide a set mechanisms to simplify the development of decentralized systems by enabling the developers to take advantage of common communication, data exchange, and coordination primitives, when developing decentralized protocols, eliminating the need for development from scratch. These include primitives to allow the communication between different processes on a network, e.g., by relying on protocols like TCP, UDP, or QUIC. Usually, mechanisms for exchanging information between distinct protocols, running on the same process, are also provided with the objective of composing multiple protocols together in order to develop more complex ones.

Appia [44] is a framework focused on the development of decentralized systems by combining multiple protocols together in a stack. Appia introduces the concept of channels, a stack of protocol instances (or sessions) that should be crossed by messages, using a FIFO mechanism, to impose the required properties. A session is, in fact, the implementation of a layer encapsulating a given set of properties and multiple layers provide a given Quality of Service (instantiated through channels). One of the main advantages of Appia is related with the capability of inter-channel coordination by sharing sessions between distinct channels. The communication between distinct sessions is performed through events triggered by one session and received by others, who should specify their interest in those events.

Libp2p [49] is a modular framework providing a set of libraries and protocols aimed at acting as building blocks for developing protocols and applications where interaction between distinct peers on a network is required, therefore simplifying the development of solutions based on P2P approaches.

Libp2p provides mechanisms for identifying peers and addressing them on a network as well as the capability of exchanging information through a set of communication protocols. The discovery and addressing mechanisms are supported on identifiers generated from key pairs together with the use of mDNS for local network discovery and/or a Kademlia DHT for discovery and routing. Regarding the communication aspects, implementations of transport protocols like TCP, UDP, QUIC, among others [50] are available to be used in decentralized solutions leveraging on Libp2p. Other modules are also provided like secure communication over TLS or *hole punching* to allow communication between nodes behind firewalls and NATs. Depending on the modules required, the implementations are available in multiple languages such as Go, Swift, or Rust [50, 51].

In the following section we focus on Babel as this is the framework on which our implementation of the solution presented in this dissertation relies on. The rationale for choosing the Babel framework as basis is discussed in Chapter 4, where the complete implementation is presented in detail.

2.5.1 The Babel framework

Babel [19] is a Java framework focused on simplifying the development of decentralized systems by allowing programmers to focus on writing algorithm related logic, instead of spending time dealing with low level aspects like network communication, thread

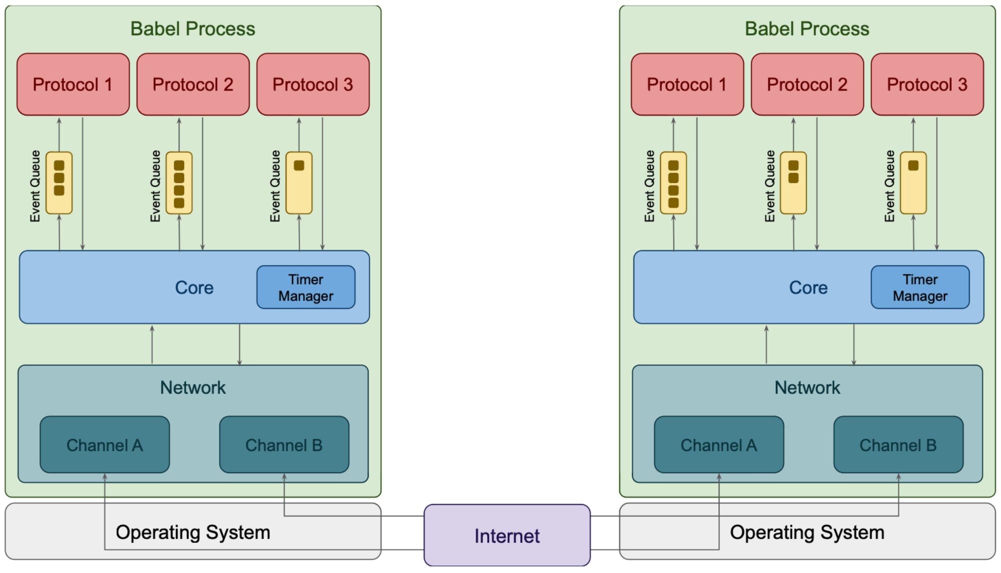


Figure 2.2: Overview of the Babel framework architecture (extracted from [19])

management, or concurrency. Figure 2.2, retrieved from the paper that describes the framework [19], provides a high level overview of the Babel framework architecture.

In Babel, multiple protocols run on the same process and interact with each other using an event-driven approach where interaction is performed through primitives like *requests*, *replies*, and *notifications*. Timers are also available and can be set to trigger at time intervals or a given moment to deal with both periodic and non-periodic tasks. Babel protocols need to define when to send an event and how events arriving at the protocol will be handled by implementing and registering callbacks (or handlers) for each event type. Both protocols and events are defined by programmers, in Babel, through the extension of Java classes. Each protocol has a unique identifier and should be registered in Babel, which will then run it on a single thread where events are queued and executed in serial, thus removing the need for dealing with concurrency. All events in Babel, namely messages, requests, replies, notifications, and timers, also have a unique identifier to verify which event was triggered and perform the necessary actions, e.g., executing the registered callbacks.

Communication, through the network, between distinct processes is also available by using network abstractions called *channels*, through which *messages* can be sent. When a protocol needs to communicate with another one on a different process, it should send a message containing the data, through the channel, to the destination. Then, when arriving at the destination, Babel messages are delivered, through the respective channel, to the protocols related with that channel where they are handled accordingly, as any other event, based on the callbacks registered. To allow the serialization and deserialization of messages, respectively, from and to the Java classes that represent them, serializers and deserializers should also be defined and registered, for a given type of message, on the channel. These serializers are required, when sending a message, to serialize the object into a byte array format or vice-versa when a message is received by the channel.

2.6 Generic solutions for interacting with Peer-To-Peer protocols

Some works have already been proposed with the objective of defining generic solutions for interacting with peer-to-peer protocols, in particular those relying on overlay networks. An initial approach was presented by Dabek et al. in [14], devising some interesting mechanisms for interacting with P2P protocols through a generic interface. However, this approach falls short in some aspects.

First, the presented *Application Programming Interface* (or API) is only suitable for interacting with structured overlay networks. Moreover, although considering mechanisms for interacting with DHTs and message dissemination, these are not described in detail, focusing only on an API called KBR (or *Key-based Routing*) containing operations related with the routing of messages to nodes, given an identifier. We also consider that this solution makes many assumptions about the operation of the protocols maintaining the structured overlay, such as the proposed `forward` upcall which informs applications about the routing of a message through the node. As an example, we believe that this upcall creates two challenges: *i)* it is not in line with the expected behavior of protocols like Kademlia, where queries are not effectively routed from node to node, and *ii)* allows applications to overwrite the operation of protocols by changing values related, for instance, with the next hop.

Regarding the mechanism for overwriting the operation of protocols, although, as discussed in the work, this could be an advantage due to the increased flexibility, we believe that it delegates to the application a responsibility of the protocol. Additionally, the `range` operation also cannot be implemented in protocols such as Kademlia [3]. Finally, the work does not devise any operations for allowing a node to join or leave the network maintained by the decentralized protocol (although an `update` upcall is defined to inform applications about the occurrence of these events).

In summary, the approach presented in [14] proposes relevant mechanisms for allowing a generic interaction with protocols based on structured overlay networks and the services provided by them. These include the specification of callbacks to notify applications about events occurred on the underlying protocol (for instance when a neighbor joined or left), as well as the definition of an API for key-based routing. However, the authors approach has an excessive focus on the operation of overlay networks, instead of focusing on the programmers that are interested in building applications relying on the services provided by decentralized protocols.

Relying on the work mentioned before as a starting point, an architecture known as OverArch was proposed by Baumgart et al. in [3]. This architecture considers both structured and unstructured overlay networks and strives to address some issues regarding the original approach proposed in [14]. In OverArch mechanisms for both key-based routing over structured overlays and dissemination over unstructured overlays are proposed, as well as additional components responsible for object location and storage, and anycast or multicast operations. These mechanisms further extend the ones presented in [14], allowing protocols like Kademlia to be supported by the architecture.

Although providing a significant improvement, the solution put forward by OverArch still maintains a high level of control over the operation of protocols, exposing interfaces that are not simple to be used by programmers not familiar with the internal operation of decentralized protocols as some protocol-related logic is exposed by the interface and needs to be understood by the programmer. In fact, as in [14], OverArch even allows for applications to overwrite the operation of protocols through the exposed API. Moreover, OverArch does not simply operate as a layer of abstraction between decentralized protocols and applications, as it provides a series of components related with communication and routing table management, leading to the exposure of protocol-related logic through the API, making it difficult to distinguish between the protocol and the abstraction.

Additionally, in the proposed architecture each interface is related with a specific type of overlay (structured or unstructured), e.g., Key-based Routing for structured overlays and *Key-independent Message Dissemination* for unstructured overlays. The possibility of the same interface being used by protocols relying on both structured and unstructured overlays is not considered.

Furthermore, none of the solutions discussed in this section provide mechanisms for simplifying the management of multiple decentralized protocols required by an application, namely regarding the generic instantiation of protocols by defining the services or properties required. Additionally, different models of interaction between applications and the exposed interfaces, for instance providing synchronous and asynchronous operations, are also not considered. These aspects are proposed in the solution presented in this dissertation, together with the development of service-based interfaces focused on simplifying the implementation of applications that rely on services provided by decentralized protocols. We believe that programmers developing applications based on our solution, through the mechanisms and generic abstractions provided, do not even need to have a significant knowledge of the underlying protocols providing the decentralized services as well as their operation.

2.7 Summary

In this chapter the fundamental concepts and related work for this dissertation were presented. The chapter started with an overview of edge computing and peer-to-peer architectures, followed by a study of overlay networks by presenting the key aspects and work developed in this area that are taken into consideration throughout this dissertation. Then, an overview of applications that can rely on the use of peer-to-peer protocols was presented as well as a set of protocols able to provide services based on a decentralized P2P approach. Additionally, we presented in this chapter a set of frameworks aimed at simplifying the development of P2P protocols and decentralized systems. Finally, a discussion about other works focusing on the development of generic solutions for interacting with decentralized protocols, in particular considering overlay networks, was also provided.

In the next chapter we present the generic abstractions devised, for each service provided by decentralized protocols, as well as the proposed architecture for our solution.

3 Generic Decentralized Abstractions

In this chapter, we propose a set of generic abstractions, focused on key decentralized services, devised with the objective of aiding programmers in the development of decentralized applications. These are expected to simplify development, improve maintainability, and facilitate further improvements, while abstracting, as much as possible, the implementation details or internal operation of specific protocols.

Our objective is to propose an architecture that allows programmers to easily choose (and switch between) the decentralized protocols used in an application by interacting with service-based interfaces instead of protocol-based ones. Furthermore, the solution should provide ways to easily instantiate new protocols, as well as manage the ones that are currently running. Figure 3.1 provides a simple representation of the considered model.

The chapter is organized as follows. First, in Section 3.1, we describe the challenges to which these abstractions intend to answer and then, in Section 3.2, an overview of the different services provided by decentralized protocols is presented, as well as the generic interfaces proposed for interacting with each one. In Section 3.3, we present the proposed solution architecture by describing each component in detail. The chapter ends with a summary in Section 3.4.

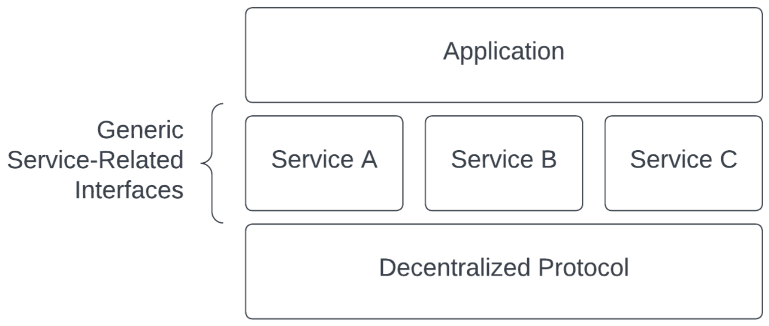


Figure 3.1: Overview of the solution model

3.1 Problem Description

Nowadays, multiple applications rely on decentralized protocols as a way obtain the necessary services to operate. Leveraging on these services, applications can then provide other high-level services like resource location, distributed file storage, publish-subscribe mechanisms, distributed computation or messaging, among others.

A significant number of decentralized protocols already exist, many of which providing the same types of services but relying on different approaches more suitable for specific execution scenarios or operational conditions. By comparing these protocols is possible to understand that, even when considering ones that provide the same set of services, the interfaces exposed to enable application-protocol interaction are, in most cases, completely different and unique to interact with each one. This, although contributing to the existence of a diverse set of tools to be chosen from depending on the operational conditions, creates challenges to developers, which include, but are not limited to:

- Developers need to be aware of multiple interfaces even when considering protocols providing the same service;
- Changing the protocol used by an application to another one providing the same set of services might require major application changes;
- Switching between protocols (possibly at runtime), providing a specific service, in response to changes in operational conditions is not possible due to the individual interfaces provided. This can happen, for instance, if in a decentralized application that became more popular, the protocol in use needs to be changed due to the increase in the number of users;
- If an application relies on multiple decentralized protocols for its operation, having to deal with specific interfaces to interact with each one not only decreases maintainability but also is more prone to errors.

As an example, we can consider protocols like Kademlia [42] and Chord [57] or HyParView [30] and Cyclon [62]. By comparing the first two it is possible to understand that both protocols implement a service able to provide applications with the node (or the set of nodes) near a given identifier. In the second case the same situation can be seen as both HyParView and Cyclon protocols are able to provide a peer sampling service, by maintaining an overlay network and returning a subset of nodes, among all currently active in the network, to applications. In both pairs of protocols, although the services provided by each one are essentially the same, the interfaces exposed for allowing the interaction with those services are different, which leads to the challenges described above in this section.

Another challenge faced when building applications relying on decentralized protocols is related with the code complexity needed to instantiate and interact with those protocols, in particular when an application needs to rely on multiple protocols. To simplify the use of protocols by applications, not only generic interfaces should be defined regarding each service as to overcome the challenges stated earlier, but also the instantiation of the protocols providing those services should be simplified. This can be done by allowing applications to only define the decentralized services needed, the properties required from the protocols providing them, and a set of parameters related with the instantiation and operation of those protocols. These parameters can include, for instance, their own network address or the contact nodes to be used when joining the network.

3.2 Decentralized Services Interfaces

Part of the work required when proposing a solution for the challenges presented in Section 3.1 is related with devising which decentralized services will be considered. This includes the generic interfaces, related with each service, that must be implemented by protocols providing those services and exposed to applications. It is worth noting however that, as we will see in Section 3.3, the solution presented here is generic, thus allowing the extension of the services proposed, through the definition and registration of new interfaces.

Two main challenges arise when devising the interfaces of services. The first one is related with the definition of the services itself by splitting protocols, and even operations inside the same protocol, into each different service (based on functionality), and therefore interface. On one hand, if interfaces have many, loosely related, operations this will result in an excessive number of largely different protocols being required to provide the same set of operations. This will lead to an unuseful solution where many protocols, although providing a service, are not able to implement many of its operations, do not implement them as expected, or have low performant implementations. This is particularly important because applications should be able to use a decentralized protocol just by specifying the services needed for their operation and expect an adequate protocol to be provided. On the other hand, defining multiple, very similar, interfaces just to accommodate minor differences between the operation of different protocols will result in numerous interfaces, many of which only implemented by a very reduced number of protocols, defeating the purpose of this effort.

The second challenge is related with the definition of the API for each operation. Excessively generic operations will result in difficulties when using the API, for instance, by forcing programmers to be aware of multiple parameters to be able to use the operations correctly. Conversely, over specific operations will result in multiple, very similar and closely related, operations together in the same service interface therefore resulting, once again, in a hard-to-use programming interface.

When designing the interfaces for each service, we not only considered the operations that are available to applications, but also the existence of notifications. We defined notifications as information that applications can receive (asynchronously) from decentralized protocols, through each service interface, but that are not a direct result of a requested operation, e.g., information about a status change of another network node.

We also considered the existence of service-related properties in each interface, like the type of overlay (structured or unstructured) in which a protocol, implementing the interface, relies on. As a result, a set of properties were defined for each service. These properties are optional and each protocol, providing a set of services and therefore implementing a set of interfaces, can then define what properties are provided and their respective values. Service-related properties can be used by applications to choose the best protocol for the required operation, among all that provide the same service.

Before presenting each one of the service-based interfaces considered, it is important to offer explanations for some parameters that are widely used in the operations described. In some operations we present parameters with the `Host` data type. This type can be considered as the information required to identify and contact a given node on the network, e.g., a `<IP, Port>` pair. Some operations also contain a parameter, called `requestId`, described as a byte array. This parameter makes it easier to use the interfaces of services in an asynchronous way, relying on a request/reply event-driven architecture, by guaranteeing that the identifier provided by applications on the request, can be returned on the reply, therefore allowing applications to easily match a reply with the respective request.

Based on the study of the applications and protocols discussed, respectively, in Sections 2.3 and 2.4, as well as the remaining related work, we propose four services that can be provided by decentralized protocols to applications.

As many protocols [30, 62, 42, 57, 8, 10, 23] require operations to manage the network (generally an overlay), e.g., in order to join or leave it, a *Membership Management* service is proposed, providing the required operations. This service becomes particularly important when considering protocols whose objective is exactly maintaining a network of nodes like the peer sampling protocols. To allow the dissemination of messages throughout the network, a service provided by multiple decentralized protocols [30, 62, 28, 31, 58, 43], the *Dissemination* service is also defined.

Additionally, to support the capability of finding a given node (or set of nodes) by providing a key, a resource identifier, or query data a *Routing* service is proposed [42, 57, 8]. Finally, the *Resource Storage* service is also presented with the objective of supporting protocols [42, 57, 23, 10, 8] providing a decentralized resource storage service.

It is important to note that, contrary to what is proposed in [14] and [3], we do not make any assumption about whereas a service should be provided by protocols relying on structured or unstructured overlay networks. As an example, we can consider the Routing service. This service can be provided both by protocols relying on a structured overlay to obtain the nodes near a given identifier and by protocols leveraging on an unstructured overlay to return the nodes that are related with a given query. The same logic can be applied to the Dissemination service as it can be provided by a protocol relying on a structured overlay (e.g., using a tree to remove cycles) or an unstructured overlay through flooding. Similarly, in the remaining services no assumptions are made regarding the network structure.

In Sections 3.2.1, 3.2.2, 3.2.3, and 3.2.4 we present and discuss each one of the different services described before, as well as the operations, notifications, and properties defined in the interfaces related with each one.

3.2.1 Membership Management

The Membership Management service interface is related with the network management operations that are provided by the decentralized protocols. We expect that almost all protocols need to implement this interface as they are required to provide a management service with operations for joining or leaving a network and getting the current neighbors.

Although we expect many protocols to implement this interface, it is particularly important when considering protocols that provide a peer sampling service, like HyParView or Cyclon because these protocols can take advantage of the `GetNeighbors` operation to provide the sample of network nodes to applications.

The following sections describe, respectively, the operations, notifications, and properties considered for the Membership Management service interface.

3.2.1.1 Membership Management Operations

For the Membership Management service interface three operations were considered: `Join`, `Leave`, and `GetNeighbors`. The detailed description of each operation is presented below.

`Join(Set<Host>)` The `Join` operation is responsible for allowing a node to join a network given a set of nodes (defined here with the `Host` data type) that should be used as contact nodes. The way each protocol takes advantage of this set for joining the network is protocol-dependent, e.g., a protocol can use only the first node on the set (eventually contacting other ones if some are not available), multiple nodes with no specific order, or multiple nodes by the order they appear on the set.

`Leave()` The `Leave` operation is responsible for allowing a node to leave the network. This operation does not require any parameters and protocols should implement it in a way that is in accordance with the expected protocol behavior when a node stops or crashes and no longer takes part on the network. A node that left the network using this operation should be able to join again using the `Join` operation. As an example, if a node A requests the `Leave` operation, the network should be on the same state that would be expected if A crashes and then restarts (without performing the `Join` operation).

`GetNeighbors(byte[], Integer) → (byte[], Set<Host>)` The `GetNeighbors` operation is responsible for providing the set of nodes that are neighbors of the node that requests the operation. The definition of a neighbor is protocol-dependent and each protocol providing the Membership Management service, and therefore implementing this operation, can provide the set of neighbors based on the specific protocol definition.

As explained before, this operation is particularly important when considering protocols that provide a peer sampling service, as they can take advantage of this operation to provide the sample of network nodes.

The `GetNeighbors` operation receives the `requestId` and `numNeighbors` parameters. The `requestId` parameter can be defined, when making the request, as a unique identifier (a byte array) to simplify the operation use in asynchronous environments. The `numNeighbors` is an optional Integer parameter allowing applications to define how many neighbors should be returned. Protocols can use this parameter as intended (or not consider it at all), e.g., providing all neighbor nodes when `numNeighbors` <= 0 or at most `numNeighbors` nodes if `numNeighbors` > 0.

The operation returns the `requestId` provided, as explained before, and a set of nodes (defined here with the `Host` data type) that represent the neighbors of a node. The size of the node set may be variable, depending on the protocol parameters and operation, like the active view size on the HyParView protocol or the cache size on Cyclon, and/or the `numNeighbors` parameter if it is defined and the protocol supports it.

3.2.1.2 Membership Management Notifications

Two (asynchronous) notifications, `NewNeighbor` and `DropNeighbor`, were devised to be triggered, by protocols, to applications interacting with them through the Membership Management service interface. The detailed description of each notification is presented below.

`NewNeighbor` We considered the `NewNeighbor` notification to be sent, by protocols, to applications interacting with them through the interface when a new neighbor is added, i.e., when a new node is present on the set returned by the `GetNeighbors` operation, if no limitation on the set size is requested. The notification only has one field, represented here with the `Host` data type, containing the information about the new neighbor.

`DropNeighbor` We considered the `DropNeighbor` notification to be sent, by protocols, to applications interacting with them through the interface when a neighbor is dropped, i.e., when a node is removed from the set returned by the `GetNeighbors` operation. The notification only has one field, represented here with the `Host` data type, containing the information about the removed neighbor.

3.2.1.3 Membership Management Properties

A set of properties, related with the Membership Management service interface, was also defined in our solution. As explained before these properties are optional and allow applications to choose the best protocol, considering their specific requirements, between all protocols providing the same service. Therefore, protocols may or may not provide values to those properties, but, if provided, these properties can help applications to choose the best protocol for their operation. The properties considered for this service are presented below.

View Type The View Type property defines the type of network view, `Global` or `Partial`, of a given protocol. As discussed in Section 2.2.1, we consider that protocols rely on a global view of the network if every node knows all the other network nodes as neighbors. On the opposite side, we consider a partial view if a node might only know a subset of all network nodes as neighbors. If a protocol maintains a partial view of the network, a mechanism needs to exist in order to update it, more details on this are provided below when describing the Peer Sampling Type property.

Peer Sampling Type This property can be used by peer sampling protocols to define its peer sampling service as `Static` or `Dynamic`. We consider that a peer sampling protocol provides a static sampling service when the sample of nodes returned, i.e., the set of nodes returned by the `GetNeighbors` operation, does not change during the operation and after a stabilization period, if no nodes join or leave the network. HyParView is an example of a static peer sampling protocol. Any peer sampling protocol that maintains a total network view is considered static because, if the view contains all nodes on the network, then it will only change if a node joins or leaves.

A peer sampling protocol can be considered as dynamic if the sample of nodes returned suffers changes over time, even if no nodes were added or removed from the network. The Cyclon protocol is an example of a dynamic peer sampling protocol.

The terms *Reactive*/*Cyclic* can also be used to express a similar property, although these terms are more related with the update mechanism for the partial view of protocols [31]. On one hand, reactive peer sampling protocols only update their network view in reaction to nodes entering or leaving the network, thus maintaining a static sample when no changes occur on the network membership. On the other hand, cyclic peer sampling protocols employ a cyclic strategy to update their views by periodically exchanging peers with other nodes, thus providing a dynamic sample of nodes that changes over time. The preference, in this work, for the usage of the *Static*/*Dynamic* terms is explained by our focus on the user taking advantage of our solution, more interested in the service provided by the protocol rather than its internal mechanisms. This becomes clear when studying protocols like HyParView which employs a combination of a reactive and cyclic strategies, although, from a user standpoint, the view provided is effectively static.

Request Nodes This property is closely related with the `numNeighbors` parameter of the `GetNeighbors` operation and is defined as a boolean value. If the property is set as true this means that the protocol supports the definition of a maximum number of nodes to be returned when performing the `GetNeighbors` operation, i.e., considers the `numNeighbors` parameter. If the property is set as false the protocol does not consider the `numNeighbors` parameter.

Overlay Structure This property allows the classification of a protocol as structured or unstructured, based on the structure of the overlay network on which it relies. As explained in Section 2.2.3, in a structured overlay network nodes are required to perform connections with other nodes to enforce a specific network topology, generally more beneficial to the task being performed. Conversely, in an unstructured overlay network the connections between nodes are not enforced and the network topology, at a given time, cannot be inferred *a priori* because it depends on factors like the moment in which a given node joined or left the network, or even random events like membership shuffles between nodes.

3.2.2 Routing

The Routing service interface provides the operations related with the capability of obtaining a node or a set of nodes, present on the network, based on a query, therefore effectively *routing* the query to a node. Many protocols, like Kademlia and Chord, provide this service by returning the nearest nodes to an identifier based on a protocol-dependent distance definition.

Protocols focused on resource location both based on exact-match queries, like Kademlia or Chord, or non exact-match queries, like Gnutella [53, 25, 8] or Gia [8], can take advantage of this interface by implementing the `FindNodes` operation.

In the following sections we describe the operations and properties considered for the Routing service interface. In this interface no notifications were defined.

3.2.2.1 Routing Operations

For the Routing service interface we considered a `FindNodes` operation, described below, to be used by applications interacting, through the interface, with a decentralized protocol providing the service.

`FindNodes(byte[], byte[])` $\longrightarrow$ `(byte[], Set<Host>)` The `FindNodes` operation is responsible for allowing the routing of a query, throughout the network, to a given node or set of nodes. The operation receives two parameters, `requestId` and `searchData`. The `requestId` parameter, defined as a byte array, can be leveraged by applications, as explained before, when interacting with protocols asynchronously, whereas the `searchData` parameter contains the query to be verified when performing the routing operation.

With the objective of maintaining the operation as generic as possible, we do not make any assumption on the data type of queries that can be sent to protocols to perform the routing operation and, as so, we consider the `searchData` parameter as being a byte array. Thus, protocols implementing the operation are responsible for processing the query data received in accordance with their specific operation, e.g., protocols like Gnutella or Gia may consider the `searchData` to be a textual representation of a resource, whereas protocols like Kademlia or Chord can perform a hash on the data provided and use the result to return the nearest nodes based on some distance metric.

The operation returns the `requestId` provided and a set of nodes (defined here with the `Host` data type) containing the result of the routing operation. The use of a set as a return value is explained by the necessity of maintaining the operation generic, in order to allow it to be implemented by multiple routing protocols with different necessities. This way, as an example, protocols like Gnutella or Gia can provide the set of nodes where matching resources were found, with no specific order, Chord can provide a set containing only one element, representative of the nearest node to the identifier, while Kademlia can return an ordered set of the nearest nodes to the given identifier, based on the XOR metric.

3.2.2.2 Routing Properties

Regarding the properties of the service, we present below the Multiple Results property.

Multiple Results The Multiple Results property consists on a Boolean value specifying if a protocol can return multiple nodes when performing the `FindNodes` operation, i.e., if the returned set may contain more than one element, or otherwise, if the protocol only returns at most one element.

3.2.3 Resource Storage

The Resource Storage service interface provides operations that can be leveraged by protocols implementing mechanisms for resource storage and location, like Distributed Hash Tables (or DHTs). Many protocols that provide the Routing service, presented in Section 3.2.2, are also expected to provide this one, however we consider that both services have significant differences regarding their operations and, as so, they should be separated into two distinct services.

Therefore, while the Routing operations are related with obtaining a set of network nodes by routing queries throughout the network, the Resource Storage operations are related with the ability of providing decentralized resource storage, independently of the underlying mechanisms. Even so, many protocols that provide a Routing service, by returning the nearest nodes for a given query, generally relying on an identifier, are also capable of storing resources on those nodes, by considering the resource identifier, effectively implementing a DHT. This is the case of protocols like Kademlia and Chord, already presented as an example previously on Section 3.2.2.

We can also consider the Freenet protocol [10] as a possible protocol for exposing this interface to provide its operations. In fact, Freenet is an example of a protocol that might not provide the Routing service but provides the Resource Storage one.

The following sections present the operations and notifications devised for this service interface.

3.2.3.1 Resource Storage Operations

We considered three operations, related with the storage of resources, to be included in the interface of the service: `PutResource`, `GetResource`, and `RemoveResource`. Each operation is presented below.

`PutResource(byte[], byte[])` The `PutResource` operation allows the storage of a new resource on the system. Our interface defines two parameters for this operation: the `key` parameter, used as key for the resource (e.g., the name of the resource), and the `data` parameter, which contains the content. As we do not make any assumption about the key or data types, we consider both fields as being an array of bytes. Protocols implementing this operation should then store the resource in some node(s) present on the network, taking into consideration both the key and content, for allowing later retrieval.

It is important to note that, regarding resource persistence on the decentralized system, e.g., when dealing with node failures, our solution does not impose any type of guarantees and, consequently, this depends entirely on the protocol in use. However, many decentralized protocols [10, 42, 57] store resources on multiple nodes to guarantee some degree of persistence even in the presence of failures.

This operation should also be used to update a given resource, by providing the resource key and the new content to update. Once again, the guarantee that the resource(s) with the given key, present on the network, will be updated depend on the protocol in use.

`GetResource(byte[], byte[])` → `(byte[], Boolean, byte[], byte[])` This operation allows the retrieval of a resource, stored on the decentralized system, by providing the respective key. We propose two parameters for performing this operation: a `requestId` parameter, defined as a byte array, to facilitate the use of this operation in asynchronous environments, as explained before, and a `key` parameter, a byte array which contains the identifier of the resource to obtain.

The operation should return the `requestId` provided during the call, a boolean `found` indicating if the requested resource was obtained, the key used to perform the resource search, and the content (defined as a byte array), which is only valid if the `found` field is set to true. As before, the guarantees regarding the retrieval of a resource with a given identifier, even if present on the network, or which one is returned, if multiple exist with the same key on different nodes of the system, are protocol-dependent.

`RemoveResource(byte[])` The `RemoveResource` operation allows for the removal of a resource stored on the network, given an identifier. The operation only receives a `key` parameter, defined as a byte array, which represents the identifier of the resource to be removed. As in the above operations, the guarantees regarding the removal of resources, i.e., if all resources stored on the system with the provided identifier will be removed, are up to the protocol in use.

3.2.3.2 Resource Storage Notifications

We propose two notifications to be triggered by protocols and sent to applications interacting with them through this service interface: a `NewResource` notification and a `RemovedResource` notification. These notifications should be triggered when a resource is inserted or removed on a node of the decentralized protocol implementing the service. More details about both notifications are presented below.

`NewResource` The `NewResource` notification should be triggered when a new resource is inserted on a given network node of the protocol implementing the resource storage interface. The notification contains the key of the inserted resource. As an example, if we consider the Kademlia or Chord protocols, this notification can be triggered, on a given node, after a resource is stored on it, i.e., the node is the nearest one when considering the key of the resource. If a resource is stored on multiple nodes, e.g., for persistence purposes, the `NewResource` notification should be triggered in all of them.

`RemovedResource` The `RemovedResource` notification should be triggered when a resource is removed on a given network node of the protocol implementing the resource storage interface. The notification contains the key of the removed resource. As before, if we consider the Kademlia or Chord protocols, this notification can be triggered, on a given node, when a resource is removed from the node. If a resource stored in multiple nodes is removed from all of them, a `RemovedResource` notification is expected to be triggered in each one.

3.2.4 Dissemination

The Dissemination service interface is responsible for enabling the interaction with message dissemination operations. This interface should be implemented by protocols that are able to disseminate a message throughout the network, for instance providing a broadcast or multicast service.

Protocols responsible for the dissemination of information, like HyParView or Cyclon (when considering the use of these directly as dissemination protocols), Plumtree [28], GoCast [58], or Araneola [43], are examples of protocols that can implement this interface to provide the dissemination service.

In this work we consider the existence of a `Disseminate` operation that should be implemented by protocols providing the service. This operation enables applications, relying on dissemination protocols, to interact with the dissemination service, independently of the underlying protocol, through a common interface.

The following sections describe the operations and notifications considered for the Dissemination service interface.

3.2.4.1 Dissemination Operations

For the Dissemination service interface, we considered a `Disseminate` operation, described below, that can be used by applications interacting, through the interface, with a decentralized protocol providing the service in order to disseminate information throughout the network.

`Disseminate(byte[])` The `Disseminate` operation is responsible for allowing the dissemination of data throughout the network and should be implemented by protocols providing this service. The operation considers only one parameter, `data`, which contains the information to be disseminated. As we do not make any assumption on the type of information that can be disseminated by protocols providing this service, the `data` parameter can be considered as an array of bytes.

3.2.4.2 Dissemination Notifications

A `DataReceived` notification can be sent, by protocols, to applications interacting with them through the Dissemination service interface. We present below a detailed description of this notification.

`DataReceived` The `DataReceived` notification can be leveraged to notify the applications, relying on a protocol providing the dissemination service, that new data, disseminated by any node on the network, was received. The `DataReceived` notification contains the `data` field, a byte array responsible for storing the disseminated data.

3.3 Proposed Solution

In this section we propose a solution for the challenges presented in Section 3.1. Our solution consists on multiple components, working together to provide an abstraction layer between applications and decentralized protocols. The purpose of this abstraction layer, acting as a *middleware*, is providing a generic, standard, and simple way for allowing

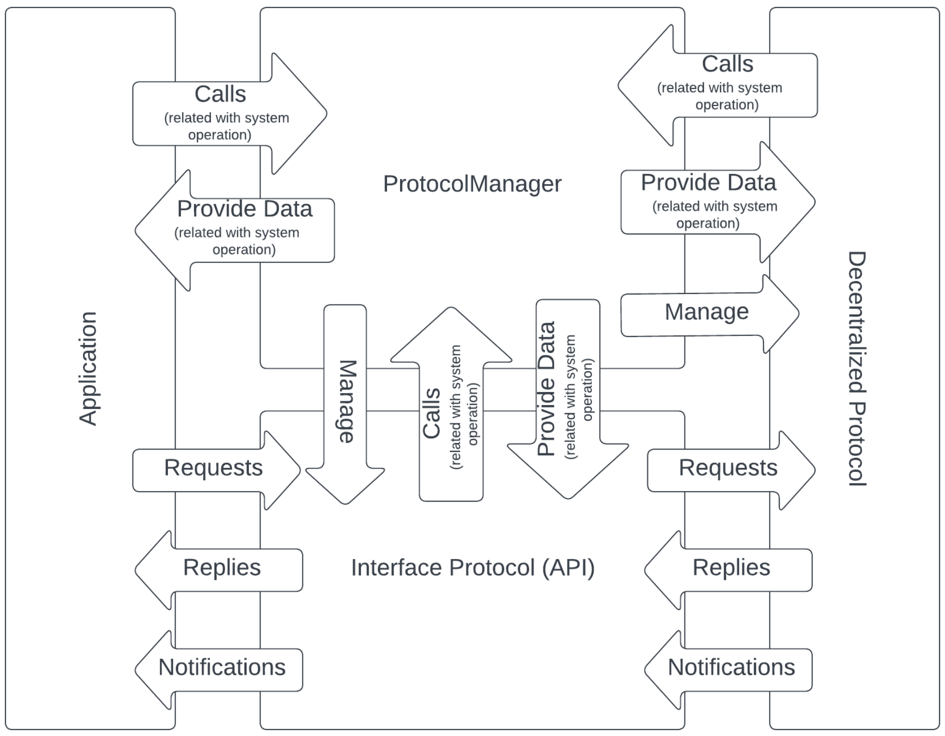


Figure 3.2: Overview of the solution components

applications to interact with decentralized protocols, through generic interfaces defined per service, in opposition to the common per protocol approach.

Additionally, the solution presented here also aims at simplifying the choice and instantiation of the most adequate protocol, providing the decentralized services required by an application, as well as allowing a simple management of the multiple decentralized protocols being executed simultaneously to provide the required services.

Our solution consists on four main components:

- The Protocol Manager;
- A set of generic interfaces for allowing applications to interact with decentralized protocols;
- The decentralized protocols providing a set of services;
- The applications that require services provided by the protocols.

Figure 3.2 presents an overview of the components as well as the interactions between them. Sections 3.3.1, 3.3.2, 3.3.3, and 3.3.4 describe, in detail, each one of the components considered.

3.3.1 Protocol Manager

The Protocol Manager component is responsible for managing all the other system components as well as maintaining information about their operation. In our solution we consider that, in each process, only one instance of the Protocol Manager should exist, and every other component should be able to access the Protocol Manager to obtain the necessary resources or information about the system.

Among other relevant information, the Protocol Manager should maintain information about:

- The decentralized protocols available, as well as the services provided by each one;
- The interfaces available (related with each service) for interacting with decentralized protocols;
- The decentralized protocols running at a given moment;
- The generic interfaces instantiated, at a given moment, to enable the interaction between the applications and the running protocols;
- Global system properties;
- Properties that are required when instantiating the protocols.

The Protocol Manager provides the necessary operations for allowing the instantiation of a new decentralized protocol, providing a given service required by an application. We considered that, in our solution, applications can request the instantiation of a new decentralized protocol both by providing the protocol name (e.g., Kademlia) or the services provided (e.g., Routing). As a result, the Protocol Manager should expose two distinct operations, described below and presented in Listings 3.1 and 3.2, to enable the use of each one of the different mechanisms. Besides the parameters presented here the operations can also be extended with other, implementation-specific, ones.

It is important to note that, when requiring the instantiation of a new protocol, an identifier, the `protocolId`, should be provided to be associated with the protocol. Applications, as well as other system components, can then rely on this identifier to refer to the specific running protocol when interacting with the Protocol Manager, e.g., for requesting the interfaces to interact with it.

Both operations, for instantiating a new protocol, should return the required information about the running protocol. This information should contain, at least, the protocol identifier, as well as the generic interfaces to be used when interacting with each one of the services provided by the protocol. Other implementation-specific information may also be provided as a return value of the operation.

The operation presented in Listing 3.1 should be used to initialize a new decentralized protocol by name. The operation receives the parameters described below.

Listing 3.1: Protocol instantiation by name

```
<Integer,Set<GenericAPI>> newProtocol
   (DecentralizedProtocol protocol, Integer protocolId,
  Boolean useExisting, Host host,
  Properties runProperties)
```

`protocol` The `protocol` parameter should contain information about the protocol to be instantiated.

`protocolId` The `protocolId` parameter should be used, by applications, to define an identifier for the protocol to instantiate. This identifier can then be used, by other system components, to refer to a specific protocol, for instance, when interacting with the Protocol Manager.

`useExisting` The `useExisting` parameter can be used by applications to define what should happen if an instance of the requested protocol is already running. If this parameter is set to true and an instance of the requested protocol is already running, the Protocol Manager does not create a new one and, instead, returns information about the existent one. If this parameter is set to true, applications should check the identifier of the protocol returned (`protocolId`) as a result of the `newProtocol` operation, because it might not be the one requested if another one is already running.

`host` The `host` parameter allows applications to specify the host information, e.g., IP address and port, that the protocol should use for its operation. If the `useExisting` parameter is set to true and a protocol instance is already running, this parameter should not be considered.

`runProperties` The `runProperties` parameter provides applications with the ability to specify the static configurations that a protocol should use when running. As an example, if we consider the Cyclon protocol, these properties allow us to define the cache size, the shuffle interval, or how many nodes should be exchanged in each shuffle. It is important to note that we do not make any assumptions about the existence of default configurations specifying the values, for each property, that should be used if no others are provided. However, if those configurations exist, they must be overwritten by the ones defined in the `runProperties` parameter. Moreover, if the `useExisting` parameter is set to true and a protocol instance is already running, this parameter should not be considered.

The operation presented in Listing 3.2 can be used to initialize a new protocol by specifying the set of services it should provide, i.e., without needing to choose a specific protocol. This operation enables the use decentralized protocols, even without requiring applications to choose a specific protocol. Below we present the parameters that should be supplied when requesting the operation. We omitted the explanations related with the parameters that were already discussed before.

Listing 3.2: Protocol instantiation by service set

```
<Integer,Set<GenericAPI>> newProtocol
   (Set<Services> servicesSet,
   Integer protocolId, Boolean useExisting, Host host,
   Properties requiredProperties, Properties runProperties)
```

`servicesSet` The `servicesSet` parameter consists on the set of services that should be provided by the protocol. This way applications do not need to define exactly which protocol will be instantiated, but only the services required. Each service matches a generic interface presented in Section 3.2.

`requiredProperties` The `requiredProperties` parameter enables applications to provide a set of properties to be used when choosing the protocol to be instantiated, between all protocols that provide the services requested in the `servicesSet` parameter. The properties provided in the `requiredProperties` parameter differ from the ones in `runProperties`: whereas the first ones are related with each decentralized service (presented in Section 3.2) and allow choosing the most appropriate protocol for a specific scenario, the second ones are related with the protocol execution, as explained before.

We do not enforce a system behavior when multiple protocols are available that match both the set of services and properties required by an application. Therefore, each implementation can define its own behavior in this situation, e.g., by having a pre-defined order to choose from when instantiating a new protocol.

3.3.2 Generic Interfaces

The generic interfaces are special protocols that allow the interaction between applications and decentralized protocols. Each interface consists on a set of operations, notifications, and properties, related with a concrete decentralized service. We presented the interfaces considered during this work in Section 3.2, however, if there is a need for an extension, more can be defined by specifying the operations, notifications, and properties available. This allows for our solution to be generic and suitable to be used with a wide variety of decentralized protocols, even if the service provided was not yet considered.

Although each interface instance is unique to interact with a service provided by a protocol, multiple applications should be allowed to take advantage of it to interact with the service. All interfaces related with a protocol are created when the protocol is instantiated and, if a new application needs to use a service provided by a running instance of a protocol, it should request the appropriate interface through the Protocol

Manager (using the protocol identifier) and rely on it to perform the interactions. When a new protocol is instantiated through the Protocol Manager, it will provide applications with the instances of the generic interfaces through which the operations can be requested, as seen before.

We propose three mechanisms for allowing the interaction between applications and generic interfaces in order to request the execution of an operation made available by a service:

- A fully asynchronous interaction based on event-driven mechanisms with requests and replies;
- A Promise-based interaction where applications are able to decide how to wait until the completion of an operation;
- A synchronous (blocking) interaction where applications always wait until the operation is complete.

By considering a wide range of interaction mechanisms we aim at: *i)* providing applications with different possibilities of interaction, depending on their specific necessities and *ii)* not limiting the languages or frameworks that can be used to implement our solution, even if the implementation of some mechanisms is not possible or needed.

The fully asynchronous interaction model is based on requests, made by applications through the interfaces, that are then handled by the decentralized protocol implementing it. Upon completion of the triggered operation, and if applicable, the interface will then forward the reply to the application. In fact, we expect that interfaces should allow applications to register handlers when performing an asynchronous operation request, that will be executed when the result is available. Handlers can also be registered on the interface, through specific operations, to deal with notifications sent from protocols to applications, as defined in Section 3.2.

When a notification is to be delivered, the interface protocol is expected to obtain all handlers registered for that type of notification and execute them. Similarly, when the result of an operation is available, the interface should obtain the handler registered when requesting the operation and execute it.

The promise based interaction model allows implementations to take advantage of the constructions present in many programming languages, like Java [21], Scala [24], C++ [20], or Rust [22], where an asynchronous operation can return a Promise (or Future) that can later be tested (or waited) for completion. This allows applications to control the amount of concurrency and deffer the execution of operations, e.g., by requesting an operation, obtaining a Promise, performing some other computations, and verifying later if the operation is completed and the results are available in order to continue execution.

In the synchronous interaction model, applications will immediately block and wait for the completion of the requested operation, if applicable. This mechanism, although not allowing any type of concurrent execution, provides developers with a simpler and less error-prone model of interaction as they do not need to define handlers for the results of asynchronous operations nor deal with Promises.

Even through, as stated before, three types of application-interface interaction models are possible within our solution, some operations, due to its nature, might not provide all interaction options. As an example, we can consider the `Disseminate` operation of the Dissemination service interface. This operation does not return any value to the application requesting it, as it triggers the dissemination of data throughout the network, although the application who requested it will also receive the data in the form of a notification. In this case, in general, only one option is available for requesting the operation, as the distinction between synchronous and asynchronous interaction does not exist in most (obvious) implementations.

In summary, generic interfaces not only need to provide the operations required to allow the interaction (synchronous and asynchronous) between applications and protocols, but also ways for registering handlers for notifications and replies that can be delivered asynchronously by the service. Moreover, to operate correctly, we believe that generic interfaces need to maintain data structures able to store and retrieve information about which handlers are registered for a given notification or reply, as well as about the operations currently in progress. Those structures should be as efficient as possible because, for instance, if the retrieval of the handlers that need to be executed is inefficient, this will result in a negative impact on the operation of applications relying on the service.

3.3.3 Decentralized Protocols

The decentralized protocols are the core component of the system, and they implement the services that can be leveraged by applications to perform their operations. As an example, if we consider the Kademlia or Chord protocols they are able to provide the operations related with the Routing service, as well as the ones related with Resource Storage.

With our solution, we strive on allowing developers to implement decentralized protocols, that can be used together with the proposed abstraction layers, by imposing as few restrictions as possible on the development. This way protocols must define which services to provide and, as a result, the required interfaces, among the ones already devised or by defining new ones. Then, protocols need to implement the logic to receive operations, process them, return results, and trigger the notifications exposed by the service interface. Although not mandatory, properties related with each service provided may also be defined, to allow applications to choose the most suitable protocol for their operation. We believe that even protocols already implemented on a given programming language and/or framework can be easily adapted to our solution, just by specifying which services are provided and implementing the logic required to deal with the related interfaces.

In addition to the impositions discussed before, regarding the necessity of implementing the operations and notifications related with the interfaces of the services, some additional restrictions may also exist related with the methods and parameters required when instantiating the protocols. These restrictions depend on the implementation of the Protocol Manager and are related with how the necessary parameters are transmitted to the protocol, e.g., the protocol identifier, the host information, or the configurations related with the protocol operation. In fact, although this might be implementation-dependent, we expect the way each protocol is instantiated, i.e., the methods that need to be executed on instantiation and the initialization parameters, to be common to all protocol implementations in order to allow the Protocol Manager to always use the same instantiation logic, regardless of the protocol.

As all other components in the system, protocols can also rely on the Protocol Manager to obtain information about the system operation, namely other running protocols, system configurations, generic interfaces to interact with other protocols, among others.

3.3.4 Applications

The applications are the system components that act as a client of a set of services, provided by a decentralized protocol. Applications should obtain from the Protocol Manager instances of protocols providing the services required for their operation, both by requesting a new instance to be created, or by obtaining information about a currently running one. Applications, as well as any other component, can also request from the Protocol Manager other relevant information about the system operation.

As explained before, applications interact with the decentralized protocols by requesting operations and registering handlers through the generic interfaces provided. Consequently, application programmers are able of changing the protocol providing a given service to another one, just by modifying the instantiation call to the Protocol Manager, without making further changes on the code, as the interfaces remain the same. However, applications should never interact directly with protocols and, as such, requests for obtaining a protocol instance or to execute an operation must always be performed, respectively, through the Protocol Manager or the returned generic service interfaces.

The solution presented here also makes it easier to reuse code, therefore simplifying application development. If two applications A and B are expected to rely on the same decentralized services, even though provided by different protocols, we expect that code from application A can be reused in B, and vice-versa. This happens because, although the underlying protocols are not the same, the interfaces through which both applications interact with the protocols are exactly the same. As expected, protocols can also be reused in multiple applications without any change on both protocols and applications due to the common interfaces implemented.

In fact, with this solution, we consider the possibility of also having decentralized protocols acting as applications. This happens, for instance, if we consider dissemination

Interface	Services			
Components	Membership Management	Routing	Resource Storage	Dissemination

Operations	`Join`
	`Leave`
	`GetNeighbors`
		`FindNodes`	`PutResource`
			`GetResource`
			`RemoveResource`
				`Disseminate`

Notifications	`NewNeighbor` `DropNeighbor`	—	`NewResource` `RemovedResource`	`DataReceived`
Properties	View Type Peer Sampling Type Request Nodes Overlay Structure	Multiple Results	—	—

Table 3.1: Summary of the proposed service-based generic programming interfaces

protocols, such as Plumtree, that rely on a peer sampling service to obtain the sample of network nodes required for its correct operation. In these situations, the dissemination protocol has a double role, acting as a protocol providing the dissemination service and, at the same time, as a client application of protocols providing, through the Membership Management service interface, the peer sampling service. The dissemination protocol can even provide a Membership Management interface for applications to interact with, by forwarding the operations to the underlying protocol. As an example, if the `Join` operation of the dissemination protocol is requested, this operation will effectively be performed by the underlying peer sampling protocol which is responsible for maintaining the network.

Using decentralized protocols as client applications of other underlying protocols, required for their operation, like in the dissemination example discussed above, allows for an even more generic approach with more configurable, although simple to use, decentralized protocols. As an example, we can have multiple instances of the same protocol, running on the same process, relying on distinct underlying protocols, among all available, for providing a given service. This could be done just by defining a property on the `runProperties`, during protocol instantiation, which will then lead to a change in the underlying protocol, requested from Protocol Manager, during the upper protocol initialization.

3.4 Summary

In this chapter we have presented a proposal that aims at simplifying the interaction between applications and decentralized protocols with a positive impact on the development and maintainability of decentralized applications.

Our solution acts as an abstraction layer or middleware between applications and protocols, providing the first ones with generic interfaces focused on the services provided, instead of the protocols itself. Moreover, through the definition of the Protocol Manager, we also provide simple and easy-to-use mechanisms to manage all decentralized protocols running on a process, making it easier for building and maintaining applications requiring multiple decentralized services to work.

To allow the development of the solution presented here, it was also necessary to study various already existent decentralized protocols, as well as the distinct services provided by them, in order to define coherent, comprehensive, and useful generic interfaces. Table 3.1 provides a summary of the operations, notifications, and properties proposed for each generic programming interface.

Throughout this chapter we have not made any assumption regarding implementation aspects, namely the ones related with programming languages or frameworks. We believe that our architecture is generic enough to be implemented in multiple programming languages and frameworks depending on the use case and/or developers preferences.

In the following chapter we propose a reference implementation, based in Java, of the solution presented here.

4 Implementation

In this chapter we present a reference implementation of the solution laid out in the previous chapter. As stated previously, our solution is generic enough to be implemented in a wide variety of programming languages and frameworks, hence, when describing it in Chapter 3 we have not made any assumption regarding implementation aspects. Here, we present an implementation of our solution for generic abstractions based on the Babel [19] Java framework. In this implementation we not only implemented the programming interfaces to interact with decentralized protocols, but also the management mechanisms to allow a simple use, instantiation, and management of decentralized protocols and services by applications.

The chapter is structured as follows: first, in Section 4.1 we provide an overview of our implementation as well as a discussion on the usage of the Babel framework as cornerstone. Then, in Section 4.2 we describe in detail the implementation of each component of the solution. Section 4.3 further specifies how the solution presented here can be used to develop an application and provides a comparison with the development of the same application without using it. The chapter ends with a summary of contents in Section 4.4.

4.1 Implementation Overview

Our reference implementation of the generic abstractions for interacting with decentralized services aims at allowing applications to easily take advantage of services provided by decentralized protocols, by relying on the common programming interfaces and the architecture presented in Chapter 3. For this purpose we relied extensively on the Java inheritance mechanisms both by using Java interfaces and class inheritance to make the implementation of the solution as generic and easy to use as possible. Moreover, although the Babel framework was used as base in our implementation, applications do not need to interact directly with it and programmers are not required to be aware of any Babel mechanism to use the interfaces exposed by services and protocols already developed, only those who wish to develop extensions to protocols or services need to be aware of the operation of the framework.

Listing 4.1: Example of a simple message dissemination

```java
public static void main(String[] args) throws Exception {
    ProtocolManager.registerProps(args);
    ProtocolManager protocolManager = ProtocolManager.getInstance();

    Host myself = new Host(InetAddress.getByName("127.0.0.1"), 10000);
    Host contactNode = new Host(InetAddress.getByName("127.0.0.1"), 10001);

    ProtocolInformation protocolInformation =
            protocolManager.newProtocol(Collections.singleton(AvailableAPI.DISSEMINATION),
                ↪ (short) 10000, true,
                true, myself, null, null);

    MembershipManagementAPI membershipManagementAPI = protocolInformation.getAPI(
        ↪ AvailableAPI.MEMBERSHIP_MANAGEMENT);
    DisseminationAPI disseminationAPI = protocolInformation.getAPI(AvailableAPI.
        ↪ DISSEMINATION);

    membershipManagementAPI.joinRequest(new JoinRequest(Collections.singleton(contactNode)
        ↪ ));

    disseminationAPI.disseminationRequest(new DisseminationRequest("Hello␣World!".getBytes
        ↪ ()));

    Runtime.getRuntime().exit(0);
}
```

Listing 4.1 represents a very simple application that relies on our solution to disseminate a message throughout the network. In a high level view we can describe the presented operation in the following steps: first an instance of the Protocol Manager is obtained and, through it, we request the instantiation of a protocol able to provide a dissemination service. Then, the application obtains the required interfaces for performing operations related with network membership and dissemination, through which it requests the `Join` and `Disseminate` operations for, respectively, joining the network and disseminating a "Hello World!" message. The node is located at port `10000` and relies on another node running in port `10001` as contact node.

As illustrated by this example, the application code does not even need to define which specific decentralized protocol to use, on the contrary it only needs to indicate which services are required and interact with them through the respective interfaces. In fact, with our solution, even if we explicitly choose the protocol to be instantiated, the change from one protocol to another, providing the same set of services, is as simple as changing the Java enumerate provided as the `protocol` parameter of the `newProtocol` method.

To develop the reference implementation of the solution presented in Chapter 3, we relied on the Babel [19] framework. Our decision to use Babel as the underlying framework for this reference implementation is based on two aspects: *i)* the architecture of Babel allows programmers to focus on the development of the solution rather than on, potentially complex and time-consuming, support mechanisms for providing communication, thread management, concurrency, or events management and *ii)* implementing our solution in Java allow us to take advantage of the inheritance mechanisms and strong type system of the language, which are important when developing abstractions that are expected to be generic and extensible, for instance, by enabling the enforcement of some restrictions and behaviors.

Below, in this chapter, the detailed implementation of each system component defined in Section 3.3 will be presented, as well as the concrete applications developed to test and evaluate our implementation. All listings presented in this chapter refer to a Java 8 development environment.

4.2 System Components

In this section we will describe in detail the implementation of each one of the components presented before in Section 3.3. This section is divided as follows: in Section 4.2.1 we present the implementation of the Protocol Manager component, followed by the Generic Interfaces in Section 4.2.2. Then, in Section 4.2.3, we describe the implementation of the supported decentralized protocols and, in Section 4.2.4, we present each application developed based on our generic abstractions.

4.2.1 Protocol Manager

The Protocol Manager, as explained in Section 3.3.1, is responsible for the coordination of all other system components, namely the instantiation of new protocols and interfaces and the management of the running ones. It is also responsible for maintaining all the necessary information about the system operation, namely the current system properties and running protocols, and to expose operations to access that information. As an example, applications (or even other protocols) can rely on the Protocol Manager to obtain the interfaces required for interacting with the services provided by protocols that are running at a given moment.

We need to ensure that, in each process, only one instance of the Protocol Manager exists, being delegated on it all the management operations. For that purpose we implemented a singleton pattern where applications should always call a static `getInstance` method exposed by the `ProtocolManager` class, without any parameters, to obtain a Protocol Manager instance. If an instance already exists it will be returned, otherwise a new one will be instantiated. The Protocol Manager can obtain the running configuration either from a configuration file or from an array of Strings representing properties obtained from the command line, however, in the latter case, properties should be registered with the `registerProps` method, that receives an array of Strings as parameter, before performing the first call to the `getInstance` method.

As in our reference implementation we leveraged on the Babel framework, the Protocol Manager is also responsible for launching the Babel instance as well as register all the necessary protocols in Babel. Therefore, the instantiation and initialization of Babel is done during the instantiation of the Protocol Manager.

To maintain information about the system, the Protocol Manager relies on a Map data structure containing the identifiers of the running protocols as keys and objects of the type `ProtocolInformation`, with information about the running protocol, as values. The `ProtocolInformation` class holds information about an instantiated decentralized protocol, its running configuration, the programming interfaces (or APIs) that can be used to interact with each service provided by the protocol, and a Boolean representing if the protocol is initialized. The generic programming interfaces to interact with the protocol are stored, in the `ProtocolInformation` object, on a Map structure with a Java enumerate, representing the decentralized service, as key and the interface instance as value. This key-value mapping is possible because, as explained in Section 3.3.2, for each service provided by a protocol only one interface should exist, and all applications should rely on it to interact with the service. More details about both the enumerates representing the available services and the classes representing the generic interfaces will be provided in Section 4.2.2. Regarding the Boolean value storing if a protocol is initialized, it is related with an `init` operation that all decentralized protocols must have, when implemented using Babel. This operation, after which the Boolean value will be set as true, should be called, through the Protocol Manager, only once for each protocol.

In our implementation we also considered concurrency aspects because, when interacting with the Protocol Manager, multiple threads might require operations to be performed simultaneously, leading the operations to require thread-safety guarantees. This is required so that we can support multithreaded applications using our implementation, as well as due to the multithreaded architecture of Babel where distinct protocol instances run on distinct threads. This may result in multiple components of our solution, developed as Babel protocols, interacting simultaneously with the Protocol Manager at a given time.

To handle the concurrency requirements discussed before, different data structures were employed: on one hand, the data structure, described before, for storing information about the running protocols on the Protocol Manager, is implemented by a Java `ConcurrentHashMap` because, as an example, a request can be made by an application, for instantiating a new protocol, while other component is requesting information about a running protocol, causing writes and reads on the Map to occur simultaneously. On the other hand, the data structure storing, on the `ProtocolInformation` class, the instances of the interfaces for the services provided was implemented relying on a Java, non-concurrent, `HashMap` as data is only inserted during protocol instantiation, when all interfaces to interact with it are also instantiated. Furthermore, we had to define synchronized regions when developing the Protocol Manager to avoid the occurrence of data-races when instantiating a new protocol, e.g., to guarantee that no other protocol is instantiated with a given identifier after verifying that the identifier of a new protocol is available. However, we do not expect this to pose a significant impact on the performance because it was only required during the operation to instantiate a new protocol, which we do not expect to be performed regularly.

Besides the `registerProps` and `getInstance` methods, discussed before, the Protocol Manager exposes the `newProtocol`, `initProtocol`, and `provideProtocolInformation` methods to applications. Each method is described in detail in the following sections.

4.2.1.1 `NewProtocol` method

The `newProtocol` method is responsible for the instantiation of a new decentralized protocol. This method represents the implementation of the operation with the same name discussed in Section 3.3.1. Two different signatures, presented in Listings 4.2 and 4.3, are available. The first one should be used when specifying, by name, the exact protocol to instantiate, while the second one should be used when instantiating a protocol by the set of services that it provides.

Listing 4.2: Signature of the `newProtocol` method (by name)

```
public ProtocolInformation newProtocol(AvailableProtocol protocol, short protocolId,
    ↪ boolean init, boolean useExisting,
                                        Host host, Properties runProperties)
        throws FailureCreatingAPIInstanceException,
            ↪ FailureCreatingProtocolInstanceException,
        FailureInitializingProtocolException, ProtocolIdAlreadyExistsException
```

The description of each parameter presented in Listing 4.2 is available below. Then, the operation of the method will be described.

`protocol` The `protocol` parameter is used to specify the exact protocol to be instantiated and expects a Java enumerate of type `AvailableProtocol` representing the decentralized protocol, among the ones available.

`protocolId` The `protocolId` parameter expects a Short value representing the identifier of the protocol to instantiate. If a protocol with the given identifier already exists, the method will throw a `ProtocolIdAlreadyExistsException`.

`useExisting` The `useExisting` parameter receives a Boolean value representing if a new instance of the protocol should be created, if another instance of the protocol is already running. Applications setting this flag as true when requesting the instantiation of a new protocol should verify, in the object returned, if the protocol instantiated has the identifier requested or, otherwise, what is the identifier of the instance returned.

`init` The `init` parameter receives a Boolean value representing if the protocol should be initialized after instantiation. If this parameter is set to false, the initialization should be performed later by calling the `initProtocol` method described in Section 4.2.1.2. This parameter will not be considered if the `useExisting` parameter is set to true and the new protocol instance is not created because one already exists.

`runProperties` The `runProperties` parameter receives a `Properties` object containing the set of configurations to be used when running the protocol, such as the active view size in HyParView [30], or the shuffle interval in Cyclon [62]. Besides this parameter, our implementation also considers the existence of a mechanism based on a configuration file where the default properties, for each decentralized protocol available, can be defined. Therefore, the properties provided in the `runProperties` parameter will overwrite the ones with the same key, defined in the global configuration file. This parameter will not be considered if the `useExisting` parameter is set to true and the new protocol instance is not created because one already exists.

`host` The `host` parameter receives a `Host` object representing the IP address and port where the protocol is going to be made available. Using the global configuration file, or the properties provided in the `runProperties` parameter, it is also possible to define an IP address and port by providing properties with keys in the format `protocol.[protocolId].address` and `protocol.[protocolId].port` containing, respectively, the IP address and port. This parameter will not be considered if the `useExisting` parameter is set to true and the new protocol instance is not created because one already exists.

After performing a request to obtain a new protocol, through the `newProtocol` method of the `ProtocolManager` class, the following operations will be performed in order:

1. If the `useExisting` parameter is set to true, an iteration is performed through all running protocols in order to find if one matching the request is available. If a matching protocol is found, the respective `ProtocolInformation` object will be returned;

2. If the `useExisting` parameter is set as false, or if no running protocol matches the requested one, a new decentralized protocol will be instantiated with the identifier provided (a verification is also performed to ensure that no other protocol already uses the identifier);

3. The configurations required for running the protocol are obtained by merging the ones defined on the global configuration file with the ones provided on the `runProperties` parameter;

4. If no host is provided through the `host` parameter, the Protocol Manager obtains it from the configuration file;

5. The protocol is instantiated by relying on a Java Class object, related with the protocol, registered on the `AvailableProtocol` enumerate. This operation is done, through the Java reflection mechanism, by calling the constructor of the protocol class with the required parameters, i.e., the `Properties` object representing the running

configuration, the protocol identifier, and the protocol host. After instantiation the protocol is also registered on Babel;

6. All required instances of classes that will act as programming interfaces (or middleware) for interacting with the services provided by the protocol are instantiated by matching the services defined in the `AvailableAPI` enumerate with the Java interfaces implemented by the protocol. A Class instance, related with the API of each service, is obtained from the respective enumerate entry, from the ones that match the services implemented by the protocol, and instantiated relying on the Java reflection mechanisms. These interface objects are, in fact, also developed as Babel protocols (more details in Section 4.2.2) and, as so, the registration on Babel and initialization is performed like any other Babel protocol;

7. A new `ProtocolInformation` object is created containing information about the decentralized protocol, its running configuration, and the set of protocols, acting as programming interfaces, instantiated for each service. All information is stored on the data structures of the Protocol Manager, as previously discussed;

8. If the `init` parameter is set to true, the protocol `init` method is called and the execution of the protocol inside the Babel framework starts by starting the respective Babel thread;

9. Finally, the `ProtocolInformation` object containing all information about the running protocol is returned.

Listing 4.3: Signature of the `newProtocol` method (by set of services)

```
    public ProtocolInformation newProtocol(Set<AvailableAPI> protocolsApis, short
        ↪ protocolId, boolean init, boolean useExisting,
                                            Host host, Properties requiredProperties, Properties
                                                ↪ runProperties)
            throws FailureCreatingAPIInstanceException,
                ↪ FailureCreatingProtocolInstanceException,
            FailureInitializingProtocolException, ProtocolIdAlreadyExistsException
```

The method shown in Listing 4.3 can be executed to request the instantiation of a new protocol, by providing the set of decentralized services required, instead of the specific protocol. This method receives a `protocolsApis` parameter, instead of the `protocol` parameter, presented in Listing 4.2. Moreover, a `requiredProperties` parameter is also available. The description of each parameter is presented below.

`protocolsApis` The `protocolsApis` parameter receives a set of Java enumerates, with type `AvailableAPI`, that represent the decentralized services provided by protocols, among the ones available. This parameter allows defining which services must be provided by the protocol requested. Therefore, when calling the `newProtocol` method, our implementation guarantees the retrieval of a protocol that provides, at least, all the services specified in the `protocolsApis` parameter or, if such protocol is not available, the operation will throw a `FailureCreatingProtocolInstanceException`.

`requiredProperties` The `requiredProperties` parameter receives a `Properties` object representing the properties of the service that must be met by the requested protocol, like the peer sampling type (static or dynamic) or the view type (total or partial). These properties, already described for each decentralized service in Section 3.2, allow applications to choose the best protocol for their operation among all that provide the same services. When calling the `newProtocol` method, applications are neither required to provide any property nor all the properties for a given service. If any property is provided, the presented implementation guarantees the retrieval of a protocol whose properties match the ones provided or, if such protocol is not available, the operation will throw a `FailureCreatingProtocolInstanceException`. These properties should not be confused with the ones provided in the `runProperties` parameter: while the first ones are related with the properties provided by the protocol, regarding the services it implements, the second ones are related with the configuration of the protocol.

The execution of the `newProtocol` method presented in Listing 4.3 is similar to the one presented in Listing 4.2. In fact, in our implementation, the method presented in Listing 4.3 calls the one in Listing 4.2 after discovering an adequate protocol, by taking into consideration both the set of services provided in `protocolsApis` and the properties defined in `requiredProperties`.

To choose an adequate protocol we iterate over all decentralized protocols implemented and registered on the `AvailableProtocol` Java enumerate to discover one that is able to provide all the services required, i.e., implements all the Java interfaces related with those services. This verification is performed by relying on Java reflection to obtain all interfaces implemented by the class that represents the protocol.

Moreover, to verify if the properties of a protocol match the ones requested, the values of the requested properties are compared with the values of the respective protocol properties, which are obtained through a static `provideProtocolProperties` method exposed by the protocol class. As discussed in Section 3.3.1, the choice of the protocol to instantiate if multiple are available matching the requirements, i.e., the services and properties requested, is implementation dependent. Therefore, in our implementation we choose the first protocol that meets all the requirements considering the order in which protocols are registered on the `AvailableProtocol` enumerate.

A note should be made regarding the use of Java reflection mechanisms, in particular when calling the constructors for instantiating the decentralized protocols and interfaces, as well as when verifying the interfaces implemented by a protocol class. Although relying on Java reflection is known to be slow [60, 16], we consider that its use does not pose a performance issue, as it is only done during the instantiation of a protocol, which is a non-recurrent operation that is expected, in most applications, to only take place a limited number of times.

4.2.1.2 `InitProtocol` method

Listing 4.4: Signature of the `initProtocol` method

```
public void initProtocol(short protoId)
        throws ProtocolNotInstantiatedException, FailureInitializingProtocolException
```

The `initProtocol` method, whose signature is presented in Listing 4.4, allows the initialization of an already instantiated protocol, i.e., the execution of the event handlers and the `init` method of the protocol. The method only receives a parameter, `protoId`, with the Short data type, representing the identifier of the protocol where the operation will be performed. If no protocol is instantiated with the given identifier, the method throws an exception.

Protocols must only be initialized once and, therefore, calling this method on an already initialized protocol will not produce any result. A protocol may have already been initialized if *i)* the `initProtocol` method was already executed for the protocol with the given identifier, or *ii)* the `init` parameter of the `newProtocol` method was set to true when requesting the protocol instantiation.

4.2.1.3 `ProvideProtocolInformation` method

Listing 4.5: Signature of the `provideProtocolInformation` method

```
public ProtocolInformation provideProtocolInformation(short protoId)
        throws ProtocolNotInstantiatedException
```

The `provideProtocolInformation` method, shown in Listing 4.5 returns an object with all information about an instantiated protocol, given the protocol identifier in the `protoId` parameter. The `ProtocolInformation` object, already presented before in this chapter, contains all information about a running protocol, namely, the protocol instance, the running configurations, the set of generic interfaces to interact with the services provided, and information about the protocol initialization. If no protocol is instantiated with the identifier provided, the method throws an exception.

4.2.2 Generic Interfaces

The generic programming interfaces act as a middleware layer, allowing the interaction between both applications and protocols providing decentralized services through a common, service-oriented, *Application Programming Interface* (or API). Although we named these components *interfaces*, this terminology is not related with the concept of Java interfaces, and should not be confused. The programming interfaces considered here are built through the implementation of components acting as a layer of abstraction between applications and protocols, allowing operations to be requested through a common API and then executed on the respective protocol. These components are also responsible for redirecting to the application the results of operations, as well as notifications originating on the underlying protocol. Each interface exposes operations to interact with a given service, among the ones available, and is responsible for implementing all the necessary logic to support the distinct interaction mechanisms available. In summary, the components acting as generic interfaces are responsible for providing a common and generic abstraction layer for interacting with a decentralized service, independently of the underlying protocol, by taking advantage of multiple (synchronous and asynchronous) interaction mechanisms.

The interface components were developed as special Babel protocols that expose the required APIs to execute the operations described in Section 3.2 both asynchronously and synchronously, when applicable. Methods are also available for registering notification handlers in order to deal with incoming notifications.

Although the programming interfaces are implemented as Babel protocols, we do not require the interaction between applications and interfaces to be performed through Babel as, in fact, we do not expect programmers of decentralized applications to know anything about the operation or usage of Babel in order to be able to use our solution. Therefore, although interface components interact with the protocols by relying on the event-driven mechanisms of the Babel framework, i.e., by sending requests and receiving replies and notifications, the interaction between applications and interfaces is possible through the mechanisms listed below:

- Synchronous mechanisms where an application requests the execution of an operation and blocks until the respective result arrive;
- Mechanisms based on Java Futures, where an application requests an operation and the interface returns a Java Future object that will only be completed, by the interface component, when the respective reply arrives from the decentralized protocol;
- Asynchronous mechanisms where applications, when requesting an operation, register a handler (implemented as a Java Consumer) to be executed when the respective reply arrives. For notifications, handlers are registered based on the notification class to deal with all notifications of that type;
- Asynchronous mechanisms based on event-driven requests, replies, and notifications provided by Babel to be leveraged by applications written using the Babel framework, but also relying on the generic abstractions proposed by our solution.

The programming interfaces can be developed by extending the `GenericAPI` Java abstract class and should implement all the service specific logic regarding the operations that can be performed through them, as well as expose the methods to perform those operations. As a result, the subclasses should expose methods for performing each service

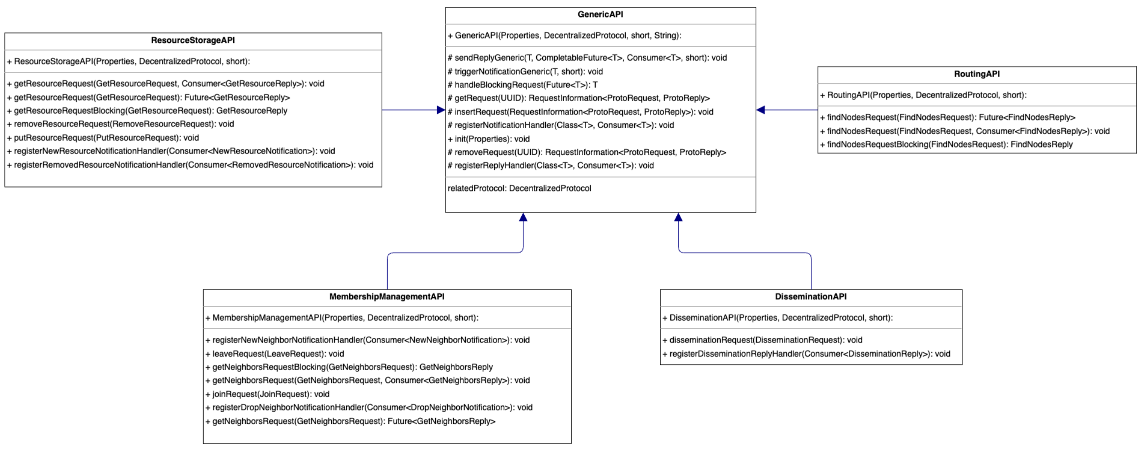

ResourceStorageAPI
+ ResourceStorageAPI(Properties, DecentralizedProtocol, short):
+ getResourceRequest(GetResourceRequest, Consumer<GetResourceReply>): void
+ getResourceRequest(GetResourceRequest): Future<GetResourceReply>
+ getResourceRequestBlocking(GetResourceRequest): GetResourceReply
+ removeResourceRequest(RemoveResourceRequest): void
+ putResourceRequest(PutResourceRequest): void
+ registerNewResourceNotificationHandler(Consumer<NewResourceNotification>): void
+ registerRemovedResourceNotificationHandler(Consumer<RemovedResourceNotification>): void
GenericAPI
+ GenericAPI(Properties, DecentralizedProtocol, short, String):
# sendReplyGeneric(T, CompletableFuture<T>, Consumer<T>, short): void
# triggerNotificationGeneric(T, short): void
# handleBlockingRequest(Future<T>): T
# getRequest(UUID): RequestInformation<ProtoRequest, ProtoReply>
# insertRequest(RequestInformation<ProtoRequest, ProtoReply>): void
# registerNotificationHandler(Class<T>, Consumer<T>): void
+ init(Properties): void
# removeRequest(UUID): RequestInformation<ProtoRequest, ProtoReply>
# registerReplyHandler(Class<T>, Consumer<T>): void
relatedProtocol: DecentralizedProtocol
RoutingAPI
+ RoutingAPI(Properties, DecentralizedProtocol, short):
+ findNodesRequest(FindNodesRequest): Future<FindNodesReply>
+ findNodesRequest(FindNodesRequest, Consumer<FindNodesReply>): void
+ findNodesRequestBlocking(FindNodesRequest): FindNodesReply
MembershipManagementAPI
+ MembershipManagementAPI(Properties, DecentralizedProtocol, short):
+ registerNewNeighborNotificationHandler(Consumer<NewNeighborNotification>): void
+ leaveRequest(LeaveRequest): void
+ getNeighborsRequestBlocking(GetNeighborsRequest): GetNeighborsReply
+ getNeighborsRequest(GetNeighborsRequest, Consumer<GetNeighborsReply>): void
+ joinRequest(JoinRequest): void
+ registerDropNeighborNotificationHandler(Consumer<DropNeighborNotification>): void
+ getNeighborsRequest(GetNeighborsRequest): Future<GetNeighborsReply>
DisseminationAPI
+ DisseminationAPI(Properties, DecentralizedProtocol, short):
+ disseminationRequest(DisseminationRequest): void
+ registerDisseminationReplyHandler(Consumer<DisseminationReply>): void

Figure 4.1: Class diagram of the generic programming interfaces

operation, considering the multiple mechanisms available, register notification handlers, as well as implement the necessary logic to receive service-specific operation results and notifications returned by the underlying protocol. Then, they can rely on the `GenericAPI` superclass to take advantage of the generic mechanisms that are common to all operations and notifications. These mechanisms include, as an example, the maintenance of data structures required for storing handlers and information related with the operations requested. In Figure 4.1 we present the class diagram of the components implemented that act as interfaces for decentralized services. The development of new programming interfaces to interact with other services can be performed by further extending the `GenericAPI` class.

Besides the middleware components described above, executed as special Babel protocols, we also defined a set of Java interfaces that should be implemented by the decentralized protocols providing a service. These Java interfaces force protocols to implement the methods that will handle the requests sent from the middleware component to the protocol. Each decentralized protocol is required to implement all interfaces related with the services provided. In Listing 4.6 we provide an example by considering the Java interface that should be implemented by protocols providing a Membership Management service.

For clarity purposes it is important to, once again, distinguish between two different concepts that are described here with the *interface* expression. On one hand, we implemented generic programming interface components that act as a middleware layer between applications requiring decentralized services and protocols implementing those services. These components are Java classes that are executed over the Babel framework and implement the necessary logic to deal with all synchronous and asynchronous interaction mechanisms described above, providing applications with service-based common APIs. On the other hand, Java interfaces were defined as a way to force protocols to implement the necessary methods to deal with the requests sent to them, from the Babel protocols implementing the middleware layer.

Listing 4.6: Membership Management Java interface

```java
public interface MembershipManagementProtocol extends GenericProtocolAPI {
   Class<? extends GenericAPI> relatedAPI = MembershipManagementAPI.class;

   void uponJoinRequest(JoinRequest request, short sourceProto);

   void uponLeaveRequest(LeaveRequest request, short sourceProto);

   void uponGetNeighborsRequest(GetNeighborsRequestInternal request, short sourceProto);

   static Class<? extends GenericAPI> provideRelatedAPI() {
      return relatedAPI;
   }
}
```

By analyzing Listing 4.6 it is possible to verify that the protocols implementing the Membership Management service are required to handle three types of requests related with the `Join`, `Leave`, and `GetNeighbors` operations. These methods should be implemented and registered on the protocol as handlers for each defined request class. The interface also stores, on the `relatedAPI` field, the service to which it is related. Moreover, the Java interface presented extends another one, named `GenericProtocolAPI`, that should be extended by all Java interfaces defined for being implemented in protocols related with a given decentralized service.

In our implementation, we provide support for all services devised in Section 3.2. Additional services can be supported by implementing classes extending the `GenericAPI` abstract class, to act as middleware, as well as defining new Java interfaces, extending the `GenericProtocolAPI` interface, to be implemented by protocols providing those services. Moreover, a Java enumerate describing the properties related with the decentralized service should also be developed. Finally, all the available services as well as the related interfaces should be registered in the Java enumerate `AvailableAPI`.

Listing 4.7: Representation of the `AvailableAPI` enumerate

```java

   MEMBERSHIP_MANAGEMENT(MembershipManagementAPI.API_NAME, MembershipManagementAPI.class,
       ↪  MembershipManagementProtocol.class, MembershipManagementProperties.class),
   ROUTING(RoutingAPI.API_NAME, RoutingAPI.class, RoutingProtocol.class,
       ↪ RoutingProperties.class),
   DISSEMINATION(DisseminationAPI.API_NAME, DisseminationAPI.class, DisseminationProtocol.
       ↪ class, DisseminationProperties.class),
   RESOURCE_STORAGE(ResourceStorageAPI.API_NAME, ResourceStorageAPI.class,
       ↪ ResourceStorageProtocol.class, ResourceStorageProperties.class);
```

Listing 4.7 presents the implementation of the Java enumerate `AvailableAPI` which represents all decentralized services available in our implementation, i.e., the services for which generic interfaces are defined. This is the same enumerate whose values should be provided in a set when calling the `newProtocol` method, through the Protocol Manager, to create a new protocol instance based on the services provided. The Java enumerate stores, respectively, information about the name of the service, the class that implements the component acting as middleware, the Java interface that represents a decentralized service and should be implemented by protocols providing it, and a Java enumerate class indicating which properties are defined for the service. These service-related properties are the ones that can be provided, if required, in the `requiredProperties` parameter when requesting the provisioning of a new protocol.

In the following we present the operation of the middleware components, or generic interfaces, regarding the instantiation, execution of operations, and notifications handling.

4.2.2.1 Instantiation of Generic Interfaces

As explained before, each interface component allowing the interaction with a decentralized service is instantiated, by the Protocol Manager, after the request for a protocol providing that service. Due to the implementation of the interface component as a Babel protocol the constructor is expected to receive a `Properties` object, allowing the access to the running configurations if required, a `DecentralizedProtocol` Java class representing the decentralized protocol for which the component is acting as middleware, and a randomly generated Short identifier to act as interface identifier.

The decision about which interface components to create, after the instantiation of a new protocol, is done by leveraging on Java reflection to get the set of Java interfaces implemented by a protocol, between all defined on the `AvailableAPI` enumerate, thus obtaining the services provided. Finally, an interface component is instantiated for each service, by executing the constructor method of the Java class, related with the service, registered on the `AvailableAPI` enumerate as the middleware component (or programming interface). As the instantiation of the Java class is made using a Java `Class` object registered on the Java enumerate, the constructor is also executed through Java reflection mechanisms.

4.2.2.2 Requesting operations

The execution of operations can be requested, by applications, on each decentralized protocol through the interface components, acting as middleware, by relying on the synchronous and asynchronous mechanisms presented before. The interaction mechanisms allow applications to choose how to expect the results of an operation, e.g., by blocking until they are available, continuing execution and verifying later, or relying on even-driven mechanisms with handlers. However, we do not need to always consider all mechanisms, as operations that are not expected to return any result (e.g., `Join`, `Leave`, or `Disseminate`) are inherently asynchronous, which discards the use of synchronous ones.

Conversely, when an operation is not expected to return any result, the request should be made to the generic interface component which directly sends it to the underlying protocol. This behavior is explained by the fact that the interface does not need to maintain any information about the operation request, as no return value is expected. This is the case of operations like `Join` or `Leave` (on the Membership Management service), `Disseminate` (on the Dissemination service), and `PutResource` or `RemovedResource` (on the Resource Storage service).

On the other side, if an operation is expected to return a result, the interface component acting as abstraction layer needs to maintain some information about the request in order to return the appropriate result. To explain in detail the mechanisms implemented to deal with these situations we first describe how an operation request is performed and, then, how the reply is handled.

When an operation request is performed, through the methods exposed by the generic interface component related with a service, and a return value is expected, a new identifier (implemented as an UUID) will be generated for the request. Then, the request information is stored on a (concurrent) Map structure with the generated identifier as key and an object storing the information as value. After that, the external request is mapped into an internal request and sent to the underlying protocol through the event-driven mechanisms of Babel because, as mentioned before, the interaction between interfaces and protocols is performed through Babel. The internal request contains all the required information that must be sent to the protocol, regarding the operation, plus the identifier generated earlier, which the protocol is required to return when replying with the result of the operation.

The behavior described above is the same, independently of the mechanism through which the operation was requested, although, in some situations additional logic is necessary. As an example, when relying on the Future-based mechanism to perform an operation, a Future object needs to be created, stored on the object that maintains information about the request for later completion, and returned to the application. Additionally, when requiring an operation to be performed by providing a handler to be executed when the result is available, the Java Consumer that represents the handler also needs to be stored. Finally, if the operation request was performed relying on the event-driven mechanisms provided by Babel to perform the application-interface interaction, the identifier of the Babel protocol which acted as the source of the request needs be stored for allowing later reply. In summary, on every situation described before, additional logic is required, and the additional data should be stored on the object responsible for maintaining the request information.

When receiving, from the underlying protocol, a reply representing the result of an operation, the Babel protocol implementing the service interface should obtain, from the Map maintaining the information of the requests, the data stored about the request. This data will then allow the adequate handling of the reply, e.g., by completing the Java Future associated with the request or executing the Java Consumer registered as handler, if applicable. Moreover, if the request was performed through the Babel mechanisms, the reply will also be converted, from the internal reply received, to an external Babel reply and sent to the protocol who originally performed the request.

A final note should be made regarding a specific parameter received by some operations:

the `requestId`. As described in Section 3.2, when performing operations where a return value is expected, applications relying on our solution can provide a `requestId` which is guaranteed to be delivered when returning the result, in order to allow the correlation between an operation and the respective result, on asynchronous interaction. This value could be used as internal request identifier, instead of the generated UUID described earlier, however, this would result in requiring applications to always provide a unique identifier as `requestId`, making the correct behavior of our solution dependent on the data provided by the applications. This is why we settled with the generation of a unique identifier, effectively managed by the programming interface layer, that acts as a request identifier. As a result, if an application sends a `requestId` when performing an operation, it will be stored on the object that maintains all the information about the operation request and later returned, on the reply.

4.2.2.3 Dealing with notifications

As described in Section 3.2, notifications were also considered in our solution to allow protocols to notify applications about events that are not a direct result of any operation requested by the application, e.g., receiving data disseminated through a dissemination protocol.

The Babel framework already provides abstractions called notifications for inter-protocol communication, however when a Babel notification is triggered by a protocol, it will be delivered to all Babel protocols that registered interest in that type of notification, i.e., a protocol is not able to send a Babel notification to another specific protocol as it will be received by all protocols interested in that type of notifications. The above behavior became a challenge in our implementation if we consider, as an example, the existence of two protocols $protocolA$ and $protocolB$, both providing the Membership Management service, respectively, through the interfaces $interfaceA$ and $interfaceB$ and sending `NewNeighbor` notifications. As both interfaces are implemented as Babel protocols, both will receive this type of notifications from protocols, which will result in one interface receiving notifications intended to the other, and vice-versa.

To deal with the previous challenge, the interfaces were implemented in a way that, when receiving a notification, a verification is performed on the identifier of the origin protocol, i.e., the identifier of the Babel protocol that triggered the notification. This leads to interface components only considering notifications whose identifier of the origin protocol is equal to identifier of the protocol to which the interface was instantiated. This is why, in our implementation, we considered two types of notifications: *i)* internal Babel notifications, representing the notification objects sent from protocols to interfaces, and *ii)* external notifications, representing the notification objects that are exposed to applications.

Applications can register, using interface methods, handlers for each notification provided by it. The handlers should be implemented as Java Consumers, i.e., references for methods that receive parameters (in this case the object representing the notification) and do not return any value. It is important to note that we do not provide any guarantees of thread safety when executing the handlers and, consequently, programmers should be cautious when implementing handlers.

Registering Consumers is performed by passing them as parameters of the methods responsible for the registration. Then, they are registered in a Map data structure defined on the `GenericAPI` superclass which maintains the list of handlers to be executed for each different type of notification. When a notification arrives at an interface component the internal notification (from the protocol) is mapped to the external notification (returned to applications) and the handlers registered for the corresponding type of notification are executed. Additionally, the external notification is also retransmitted, via the mechanisms provided by Babel, to applications that may rely on the generic abstraction devised, but interact with them through the event-driven mechanisms exposed by Babel.

We clarify here that the handlers registered, by the client applications, on a given interface to be executed when a notification (or an operation result) is available, have nothing in common with the handlers that are registered, on Babel protocols, to deal with events. Whereas, in Babel, a protocol can register a method to be executed when a Babel reply or notification arrives, in our interfaces all registered handlers are Java Consumers that will be executed relying on a thread pool when required, i.e., when an operation is completed by the underlying protocol and the result is available, or a notification is delivered. A detailed description of the thread pool mechanism is available in Section 4.2.2.4.

4.2.2.4 Thread pool mechanism

As stated before, Java Consumers can be registered as handlers both to deal with notifications and return values of operations. In the first case, the Consumers are stored, as explained before, in a Map structure with the type of notification as key, for efficient retrieval when a notification of a given type needs to be delivered. In the second case, the Consumers are kept in the object that maintains information about a given operation request, to be executed when the respective return value is available.

For executing the Consumers we leverage, in both cases, on the use of a thread pool mechanism. The thread pool is global, and all interface components rely on the same pool instance to execute the necessary operations. The pool is implemented by relying on a Java fixed size thread pool, initialized with a specific number of threads that are always available. Consumers will be executed immediately if a thread is available in the pool to handle them, otherwise they will need to wait, in a queue, until a new thread becomes available. Concurrency issues related with the simultaneous execution of Consumers, e.g., if two Consumers use the same data structure concurrently, need to be handled by the programmer during implementation.

In each process running our solution, the thread pool size can be configured by

Protocols	Services			
	Membership Management	Dissemination	Routing	Resource Storage
Kademlia	X		X	X
HyParView	X			
Cyclon	X			
Plumtree	X	X		
FloodGossip	X	X		

Table 4.1: Implemented protocols and services

setting the `system.threadpoolsize` global property with the required value. Adjusting this property might be important when the registered handlers are expected to be more complex and take significant time to execute (when compared with the expected interval between notifications or results received). Consequently, applications can benefit from the increase on the number of threads available to attenuate the increased complexity.

4.2.3 Decentralized Protocols

The decentralized protocols are the components responsible for effectively providing the services that can be leveraged by applications relying on a decentralized approach. In this work, we considered four services, described in detail in Section 3.2: *Membership Management*, *Dissemination*, *Resource Storage*, and *Routing*. There are multiple existent protocols able to provide each service considered, some of which already presented before in this work. For implementation purposes we choose a set of decentralized protocols that covers all the considered services, with some of them even being provided by more than one implemented protocol. Table 4.1 presents each protocol implemented, as well as the services provided by each one. Kademlia [42] was implemented from scratch, whereas the remaining were adapted to our solution based on the ones already implemented in Babel [12, 13].

It is noteworthy that the FloodGossip protocol is a simple gossip based dissemination protocol where nodes flood messages throughout the entire network by sending them to all neighbors. In this protocol the nodes should verify if a message was already received (using an identifier) and, if not, send it to all neighbors, obtained by relying on an underlying peer sampling service, which then perform the same operation until the message eventually reaches all network nodes.

Our solution can be extended with more protocols than the ones presented here just by implementing the Java interfaces related with the services provided. An example of these interfaces is the one presented in Listing 4.6 on the previous section. By implementing the interfaces, the protocol is forced to implement the methods responsible for receiving, from the interface components described in Section 4.2.2, the internal events related with the operations of the service. As protocols are also implemented on the Babel framework, these methods should then be registered as Babel handlers for the events. Moreover, protocols should trigger service-specific notifications and replies that will be received by the upper component which already implements the Babel handlers for those events. Lastly, each protocol should be registered on the `AvailableProtocol` Java enumerate which stores information about all protocols available on the system.

As discussed before, protocols can define a set of properties, related with the services provided, to allow applications to choose the most appropriate one, among all implementing a service. In our implementation we define these properties by storing them on a `.properties` file related with each protocol, whose path should be provided on the `AvailableProtocol` Java enumerate. As protocols can provide multiple services, the properties keys are defined with a `[service].[property]` notation. When requesting a new protocol, the Protocol Manager will consider the files containing the properties of the protocol to perform the required verifications.

As an example, if one wants to add support for the Chord protocol in our architecture, a Babel protocol needs to be implemented with the logic regarding the protocol. The protocol class should implement the Java interfaces related with the Membership Management and Routing services (and eventually the Resource Storage service) and register the adequate Babel handlers, as well as trigger the respective events. Finally, the protocol needs to be registered on the `AvailableProtocol` enumerate, as well as the path of the respective properties file. Performing these steps makes the protocol available to any application relying on our solution. Furthermore, if one wants to use a protocol, developed to provide a service to one application, in another application, it only needs to add the protocol files to the second one and register them on the `AvailableProtocol` enumerate.

Protocols should only send and receive events to and from the upper interface components, therefore not interacting directly with the applications. In addition, protocols must send the event to the appropriate service interface, if more than one service is provided, e.g., if a protocol provides the Membership Management and Dissemination services, each event needs to be sent to the respective interface.

The following sections describe some noteworthy challenges that arise when developing the decentralized protocols for the reference implementation.

4.2.3.1 Protocols as abstractions clients

As discussed in Section 3.3.4, not only applications can rely on our solution, but also protocols acting as clients of the programming interfaces provided. Examples of this situation can be found in the implementations of the Plumtree [28] and the FloodGossip dissemination protocols as both rely on an underlying peer sampling protocol for network management. In these situations the dissemination protocols request, from the Protocol Manager, an instance of the peer sampling protocol required and interact with it through the Membership Management programming interface, by calling the `GetNeighbors` operation to obtain a sample of peers and then listening for the `NewNeighbor` and `DropNeighbor` notifications in order to respond to membership changes. The upper dissemination protocols also expose the operations related with Membership Management, namely the `Join` and `Leave` operations, to allow applications to request their execution, however, the requests should be relayed to the underlying protocol, in charge of maintaining the network. In our implementation, the underlying peer sampling protocol can even be defined just by setting a property value, when requesting the instantiation of a dissemination protocol, as the exposed interface that allows the interaction between protocols remains the same.

A note should also be made regarding the use of channels exposed by Babel when coupling two protocols together, as described before. In Babel the communication between different processes is performed over the network, through abstractions called channels. Channels should be created by a host, on an IP address and port, to send and receive data from channels of other processes on the network by opening connections, allowing information to be exchanged. Then, when a connection to another host is no longer required it can be closed. In Babel, channels can be shared across multiple protocols, allowing the underlying protocol to share a channel with the upper protocol meaning that, as an example, only one port is required for the protocols to operate, simplifying the management of the network as only one channel is required and needs to be managed instead of multiple ones.

Although the previously discussed mechanism for sharing channels brings advantages, it also brought challenges during the implementation of our solution, in particular when considering situations where requests for closing connections were performed by both protocols. This is the case of Cyclon and the FloodGossip protocols when used in conjunction, respectively, as peer sampling and dissemination protocol. In this situation as the Cyclon protocol only maintains connections open during the exchange of information, e.g., when executing shuffle operations, closing them afterwards, this would cause messages to be dropped by the upper protocol if the connection to a node was closed by the underlying one during that period. To overcome this challenge a new channel, called `SharedTCPChannel` was developed for Babel and is already available for use on the GitHub repository [18, 19]. This channel relies on the same mechanisms of the pre-existent Babel `TCPChannel` but maintains information about which protocols opened a connection to a given host and, when executing the close connection operation, it is only effectively closed if no other protocol required that connection to be opened.

4.2.3.2 Leave network operation

Another challenge encountered when implementing the protocols for the solution presented in this document was related with the `Leave` operation exposed by the Membership Management service. As described in Section 3.2.1, this operation removes a node from the network and should leave the node on the same state as it would be if it was instantiated, and no `Join` operation was performed afterwards.

In general, performing this operation would require a node to stop sending messages and close all connections on the network maintained by the protocol which, in turn, would force all neighbors to consider that the node left the network and act accordingly. However, in Babel, two TCP connections are maintained to another host from the perspective of one channel: an outgoing one, created from a node to another, and an incoming one, on the opposite direction. Therefore, even if a node closes a connection this only applies to the outgoing, as the incoming one is kept open and cannot be closed. Another possibility could be discarding the channel and creating a new one, therefore closing all the TCP connections related with the channel and even freeing the port again, but this is not possible in the current version of Babel. To tackle this challenge, different mechanisms regarding the implementation of the `Leave` operation were put in place when implementing and adapting the protocols to our solution.

Our version of Kademlia was developed relying on TCP connections as these are the ones supported by Babel. Hence, connections are always maintained between nodes from the moment a node discovers another one, i.e., if a node A discovers a node B a connection from A to B is established and, as a result, when B detects that A opened a connection it also establishes the opposite connection. Information about existent connections is maintained in a separated data structure and a node only closes connections when *i)* it leaves the network or *ii)* a failure of a node is detected. Consequently, because Babel is able to trigger channel notifications when an incoming or outgoing connection is closed, if a node wants to leave the network it should close the connections to all other nodes to whom connections are maintained. This will result in nodes receiving the information about the closure of an incoming connection with the node that left, triggering the closure of the outgoing connections and acting as if the node crashed, effectively removing it from the network.

In Cyclon, as in our implementation TCP connections are only maintained during the exchange of information, the mechanisms for leaving the network are simpler. Therefore, when a node wants to leave it stops sending and replying to shuffle messages, leading its neighbors to eventually consider the node as crashed and removing it from their network views. Moreover, the connections that are active when the `Leave` operation is performed are also closed.

In HyParView relying on any of the solutions presented above was not sufficient as a node could still consider another one as being present in the network even if it closed all outgoing connections. This would happen because HyParView relies on TCP connections and the capability of establishing them as a failure detection mechanism. For that reason, even if the outgoing connections of the leaving node were closed, the presence of the node in the passive views of others this might lead to it being added again in the active views as incoming connections could still be established because the channel was effectively open (and cannot be fully closed due to the limitations discussed above). To handle this situation, when performing the `Leave` operation, the HyParView protocol not only stops all shuffles, closes all connections, and cleans all views, but also sends a special message indicating that the node is not available when it receives messages from any other node while out of the network. Other nodes should interpret those messages in the same way they would if the node crashed and therefore was unresponsive.

It is important to note that the solutions used in Cyclon and HyParView are not without a drawback: the `Join` operation needs to be explicitly performed even in the first network node (e.g., with an empty set of contact nodes) to allow the node to start answering requests and effectively be present in the network. Otherwise, it will not answer any request from other nodes (in Cyclon) or answer with the special not available message (in HyParView).

4.2.3.3 Kademlia as resource storage protocol

The Kademlia protocol was implemented according to the operation described in the original paper [42], in particular considering the XOR based node lookup mechanisms on which the protocol operation relies. However, regarding the resource storage mechanisms, we believe that some details of our implementation are noteworthy. First, the operation to store a new resource is performed as expected, i.e., by performing a lookup for the k-nearest nodes to the resource identifier and storing the resource on them.

Then, each node storing resources is responsible, as described in the Kademlia paper, for republishing the keys in a fixed (configurable) time interval in order to avoid losses of data due to node failures, as well as to guarantee the storage of resources in the nearest nodes, even if new nodes joined the system after the resource has been stored. Moreover, we also employed the optimization where each node only republishes a key if it was not republished by any other node, which can be verified by leveraging on the last update timestamp of each resource. If, when republishing a key, a node discovers that it is no longer on the nearest nodes set of the resource identifier, the resource is removed from the node. This is, as far as we know, a different solution when compared to the ones commonly employed when implementing Kademlia, such as the one in IPFS [4], where the keys are republished by the nodes who requested the resource to be stored, instead of the ones who effectively store the resource.

When obtaining a resource, all the nearest nodes to the resource identifier are queried and should return the resource matching the provided identifier, or the information about its absence. The first matching resource obtained from the nearest nodes is returned and the lookup finishes. On the other side, if all nearest nodes do not contain the requested resource or if a (configurable) lookup timeout is triggered, we consider that the resource does not exist.

Finally, the operation to remove a resource from the network was also implemented. This operation is not described in the Kademlia paper, and it was implemented based on a best-effort assumption to comply with our Resource Storage programming interface. Therefore, when a node wants to remove a resource from the network, a lookup for the resource identifier is performed, and a resource removal message is sent to all nearest nodes which should, in turn, perform the operation. It is expected that the removal is eventually performed when republishing the key even in nodes that are not on the nearest nodes set but still maintain the resource, through the mechanism described above. However, this operation does not guarantee the removal of a resource in all conditions. As an example, if a node A, storing a resource, is no longer included on the set of the nearest nodes of a resource, it will not receive the removal message, even though it can be added again later, e.g., due to a failure on a node B included in the set. This will lead to the resource being republished, on the next republish window of A, which makes it available again on the network even if previously removed.

4.2.4 Applications

The applications are the system components that interact with the decentralized services provided by the protocols, through the generic programming interfaces. Applications are expected to only interact with the Protocol Manager and the APIs provided, instead of directly interacting with the decentralized protocols. As explained before, this approach allows applications to change the protocol in which a given service relies on, without the need for additional changes.

In general, as seen in the simple dissemination application presented in Listing 4.1, applications will first call the `getInstance` method of the Protocol Manager to obtain a manager instance and then rely on it, by calling the `newProtocol` method, to instantiate the necessary protocols. Moreover, if an application needs to change the protocol in use, this will only require a change of the requested protocol, when considering the instantiation of a protocol by name, or the `requiredProperties`, when instantiating a protocol by the services provided.

To validate, test, and evaluate our solution four applications were developed: a peer sampling application, a routing application, a dissemination application, and a decentralized storage solution that triggers notifications when resources are changed. These applications entirely cover all aspects of our solution, by relying on all services available. Sections 4.2.4.1, 4.2.4.2, 4.2.4.3, and 4.2.4.4 describe each one, respectively. The performance evaluation results presented in Chapter 5 are based on both the routing and dissemination applications described here whereas, for the evaluation of code complexity, all applications were considered.

4.2.4.1 Peer Sampling application

The peer sampling application consists in a simple application that performs multiple requests to obtain a set of nodes from an underlying protocol. The application is therefore supported by a peer sampling protocol and interacts with it through the Membership Management interface using the `GetNeighbors` operation. This application is described in detail in Section 4.3 where it is used as an example when comparing between an implementation based on our solution and an implementation performed using only Babel.

The application operates by first requesting a `Join` operation from the Membership Management interface with the objective of joining the network maintained by the protocol. Then, the requests are performed in a closed-loop, using a configured time interval, by calling the blocking version of the `GetNeighbors` method exposed by the Membership Management interface to obtain the sample of nodes. The nodes retrieved from the protocol are presented on the console after each call to the method responsible for providing the sample.

4.2.4.2 Routing application

The routing application is responsible for performing query routing requests and expects to receive the set of nodes that match each query. This application is supported by the Routing interface, leveraging the `FindNodes` operation and an underlying protocol implementing it. Furthermore, it also interacts with the Membership Management interface to perform the `Join` operation.

We implemented two versions of the application, an interactive one, for manual testing, and an automated one, in order to perform the evaluation. In the manual version both the `Join` and `FindNodes` operations need to be performed manually, by interacting with the application through the insertion of commands and providing, respectively, the contact node and the data to be considered when routing the query. Moreover, the manual application also supports the execution of the `Leave` operation.

In the automated version, each node first performs a `Join` operation, waits for a predefined amount of time, and then issues multiple routing operations, using a randomly generated query, in a closed-loop, i.e., when it receives the result of one operation the next one is requested, during a predefined amount of time. For evaluation purposes the moment at which the routing operations start and end, as well as a cooldown period during which, although the node stays on the network it does not perform any routing requests, are configurable.

In both application versions, we request an operation and expect the result to be returned before the next one is performed. In the interactive version, the request is performed by relying on the synchronous `FindNodes` method, provided by our API, blocking until the return value is available. Regarding the automated version, the application relies on the Future-based API method to perform the requests. This way the automatic routing application calls the Future-based `FindNodes` method, obtains the returned future and blocks until the operation is completed.

The use of Futures, instead of the blocking operation exposed by the API, is explained by the necessity to define a timeout if no return value is available, which can be configured when calling the `Get` operation of the Java Futures. In fact, the same could be done through a configuration property of our solution, which sets the timeout for the blocking operations, but by setting the timeout on the Future it can be configured at the application level.

4.2.4.3 Dissemination application

The dissemination application performs the dissemination of data throughout the network. It is supported by the Dissemination interface, through the `Disseminate` operation and the underlying protocol implementing it. As in the previous application, it also relies on the Membership Management interface for performing the `Join` operation.

An underlying peer sampling service is often required for protocols providing a dissemination service, like Plumtree [28] or the implemented FloodGossip protocol, in order to obtain a subset of network peers. Therefore, in our implementation, the dissemination protocols that exhibit this behavior can request from the Protocol Manager an underlying protocol able to maintain an overlay network and provide a sample of peers. The interaction with this peer sampling protocol will be performed through the Membership Management interface by requesting the `GetNeighbors` operation. This behavior means that the Membership Management interface is, although indirectly, also extensively used by this application.

As before, we developed two distinct versions of the Dissemination application: an interactive one, where a user can manually request data to be disseminated through the network (as well as perform the `Join` and `Leave` operations), through the insertion of commands, and an automated one, for evaluation purposes.

The automated version of the dissemination application is implemented as follows: first, the node performs the `Join` operation with a preconfigured contact node, then, it starts a thread responsible for disseminating random data payloads with a given size at fixed intervals, until a time limit is reached. Both the dissemination interval and the payload size are configurable. For evaluation purposes, it is also possible to define the moment for starting and stopping the message dissemination, as well as a cooldown period during which messages disseminated by other nodes can still be received. We present the evaluation aspects in detail in Chapter 5.

In this application, as the `Disseminate` operation is intrinsically asynchronous we take advantage of the asynchronous mechanisms exposed by our programming interface through the registration of a handler responsible for receiving the incoming data.

4.2.4.4 Decentralized Storage application

A more complex application responsible for providing a decentralized storage service, with resource change notifications, was implemented. In addition to the Membership Management interface, this application relies on both the Resource Storage and Dissemination programming interfaces. On one hand, the Resource Storage interface is required to perform the resources-related operations (`PutResource`, `GetResource`, and `RemoveResource`), but on the other hand, the Dissemination interface is required to disseminate the notifications related with the modification of a resource. Below we describe in detail the operation of the application.

The mechanisms related with resource storage are implemented as expected, i.e, by relying on the operations exposed by the Resource Storage API and implemented by the underlying protocols. Consequently, the application requests, after the initialization, from the Protocol Manager the instantiation of a protocol able to provide the Resource Storage service and then leverages on the returned interfaces for interacting with it. The resources are defined by a name, considered the resource key, and an array of bytes describing the resource content. In our implementation, when interacting with the application, both the name and resource content can be provided as String values.

For the notification mechanisms two different approaches could be followed: *i)* an approach where only one instance of a dissemination protocol is executed in each node, with all nodes participating in the same network, where each node that performs a change on a resource is responsible for disseminating the notification, or *ii)* an approach where each node runs multiple instances of dissemination protocols, therefore being present on multiple networks, each one responsible for the dissemination of messages for a subset of resources stored in the system.

The first solution is more simple and less computationally expensive, however, the effort put on the network may be higher because messages need to be disseminated throughout the entire network, even if only a small set of nodes are interested. Moreover, each node will end up receiving numerous unnecessary messages that are not related with any resource on which it is interested in and that, consequently, will end up being discarded. This solution is suitable in environments where we expect that, in general, a significant part of all nodes on the network will be interested in being notified about any resource.

Conversely, the second solution is more complex and may be more computationally expensive due to the large number of protocol instances that might need to run on a single process. However, the effort that is put on each instance is expected to be lower because it will only be responsible for disseminating information about changes on a subset of resources, and only the nodes related with those resources will be present on the network. As a result, this solution is more suitable when it is expected that only a small subset of nodes is interested in receiving information about changes on a given resource.

In our implementation we opted for the second solution by leveraging on multiple instances of a protocol providing the Dissemination service, each responsible for disseminating the data related with a subset of resources. Consequently, based on a maximum number of dissemination instances that should be executed ($maxinstances$) the application will, for each resource, perform a hash function on the key resulting in an $hashvalue$ where $0 <= hashvalue < maxinstances$. The $hashvalue$ will then be used, by the system nodes, to define the protocol identifier and request, from the Protocol Manager, the instantiation of a dissemination protocol with that identifier. As a result, no more than $maxinstances$ dissemination protocols will be executed on each process.

We expect that two types of nodes need to participate on a dissemination network related with a subset of resources sharing a common $hashvalue$: *i)* nodes that are interested in receiving notifications about changes occurred on (at least) a resource whose $hashvalue$ matches the one of the dissemination network, and *ii)* nodes that, even though may not be interested in receiving notifications about any resource related with a given network, are effectively storing those resources, e.g., the nearest nodes to the resource identifier in protocols like Chord or Kademlia.

The first ones need to be present on the network in order to receive notifications about the resources in which they are interested, in our implementation this corresponds with the resources whose key was retrieved at some point by the node. The second ones need to be present because they will act as contact nodes for the network, allowing the application to use the nodes that effectively store a resource as contact nodes for joining the dissemination network of that resource. In fact, the application relies on the `NewResource` notification delivered through the Resource Storage interface to know when a new resource is stored locally on the node, by the protocol providing Resource Storage. When the notification is received the node should join the correspondent network, if not joined yet, through the instantiation of the required protocol. On the opposite side, when a `RemovedResource` notification arrives from the Resource Storage service, a node can verify if no longer needs to be present on the network related with that resource, i.e., if no other known resource is related with that network, and, if that is the case, trigger the `Leave` operation of the Membership Management service.

To implement the logic described above, our resource storage application maintains a data structure that stores, for each dissemination protocol maintaining a network for supporting notifications of resources related with a given hash value, the sets of resources *subscribed* and *tracked*. The subscribed resources are the ones that should be considered when receiving a new dissemination notification as they are the ones from which the application is interested in receiving notifications. The tracked resources consist on the previously mentioned resources plus the ones that, although not interested in receiving notifications, are effectively stored in the node, forcing it to participate on the networks (related with the hash values) to be used as contact.

When the application wants to store a new resource, or modify an existent one, it should perform a request, through the `PutResource` operation of the Resource Storage interface, to store it by relying on the underlying protocol providing the service. After that, it should verify if a dissemination protocol maintaining a network for the hash value of the resource is already running and, if not, instantiate a new one. Then, if the resource was not yet tracked by the node, it performs a `Join` operation on the dissemination protocol, considering the nodes that store the file as contacts. Finally, a notification should be disseminated through the dissemination protocol to inform other nodes about the resource change. We should note that, regarding the dissemination protocol, even if one protocol related with the hash value of the resource is running, but the resource is not in the set of tracked ones, the `Join` operation should be performed. This is required in order to possibly merge two dissemination networks that may be disjoint, i.e., have no peers in common, because they were created for different resources, but with the same hash value as the new resource.

To obtain a resource stored on the decentralized storage system the application should perform a `GetResource` operation, through the Resource Storage interface, to be executed on the protocol providing the service. Moreover, as we consider that when a node retrieves a resource it is interested in receiving notifications about future changes, it should also join the dissemination network, as described above when discussing the resource storage/modification logic.

Lastly, to remove a resource from the system, a node should call the `RemoveResource` operation of the Resource Storage interface which will trigger the necessary actions on the underlying protocol. Also, the node should verify if it needs to stay on the dissemination network related with the hash value of the resource or, otherwise, if it can leave. The same behavior should also be triggered on the nodes effectively storing the resource and where it was actually removed, e.g., the nearest nodes to the resource identifier.

It is noteworthy however, that even with this implementation the notifications received related with resource changes need to be filtered, by taking into consideration the subscribed resources, as collisions between hash values of resource keys are expected to happen. As an example, if we consider two resources A and B, a hash function `h`, and an application only interested in notifications about resource A, if $h(A) = h(B)$ then notifications about both resources will be disseminated throughout the same network, leading to the necessity of filtering to only consider the notifications related with A.

The decentralized storage application presented in this section shows the importance of relying on the implemented abstractions to simplify the development of more complex applications that rely on multiple services provided by decentralized protocols. In this example, a significant number of protocols are required to be executed simultaneously, e.g., due to the necessity of instantiating multiple dissemination protocols based on the hash value of the resources. Also, the interaction with various decentralized services is required when considering the Membership Management services, to perform the network management operations on both the protocols responsible for resource location and dissemination, as well as when interacting with the Resource Location and Dissemination services necessary for the operation of the application.

In this situation relying on common interfaces for interacting with the services provided by different protocols helps the developer when building the application and managing the various running protocols providing distinct services, even if a complex use of decentralized protocols is required.

4.3 Comparing applications implementation

In this section we present a comparison between the implementation of a simple application using our solution and relying solely on Babel. The application considered here is the peer sampling application discussed in Section 4.2.4.1 and presents the following behavior: relying on a peer sampling protocol it joins the network, waits for two minutes to allow other nodes to join and, then, performs consecutive requests to the peer sampling protocol in intervals of two minutes. The sample obtained as a result of each request is then presented in the console.

Listing 4.8 presents the implementation of the application considering our solution whereas in Listings 4.9 and 4.10 an implementation of the same application relying exclusively on Babel is provided.

Listing 4.8: Example of a peer sampling application developed with our solution

```java
public class PeerSamplingApp {
   private static final short PEER_SAMPLING_PROTOCOL_ID = 10000;
   private static final long REQUEST_TIME = Duration.ofMinutes(2).toMillis();

   public static void main(String[] args) throws Exception {
      Host contact = null;

      if (args.length > 0)
         contact = HostUtils.getHostFromString(args[0]);

      ProtocolManager.registerProps(args);
      ProtocolManager protocolManager = ProtocolManager.getInstance();

      Properties protocolProperties = new Properties();
      protocolProperties.setProperty(MembershipManagementProperties.PEER_SAMPLING_TYPE.
          ↪ getProperty(),
            MembershipManagementProperties.PeerSamplingType.STATIC.name());
      protocolProperties.setProperty(MembershipManagementProperties.OVERLAY_STRUCTURE.
          ↪ getProperty(),
            MembershipManagementProperties.OverlayStructure.UNSTRUCTURED.name());

      ProtocolInformation protocolInformation =
            protocolManager.newProtocol(Collections.singleton(AvailableAPI.
                ↪ MEMBERSHIP_MANAGEMENT), PEER_SAMPLING_PROTOCOL_ID,
                  true, false, null, protocolProperties, null);

      MembershipManagementAPI membershipManagementAPI = protocolInformation.getAPI(
          ↪ AvailableAPI.MEMBERSHIP_MANAGEMENT);

      Set<Host> contactNodes = new HashSet<>();

      if (contact != null)
         contactNodes.add(contact);

      membershipManagementAPI.joinRequest(new JoinRequest(contactNodes));

      while (true) {
         Thread.sleep(REQUEST_TIME);
         GetNeighborsReply getNeighborsReply = membershipManagementAPI.
             ↪ getNeighborsRequestBlocking(new GetNeighborsRequest());
         System.out.println("Node␣neighbors:␣" + getNeighborsReply.getNeighbors());
      }
   }
}
```

By examining Listing 4.8 it is possible to understand how the proposed solution can be used when developing applications taking advantage of a decentralized approach.

First, an instance of the Protocol Manager is requested which leads to its creation, if no instance was already created. After that, between lines 20 and 22, a protocol able to provide the Membership Management service is requested by providing as first parameter a set containing a single element representing the required service. In this example the protocol is requested by specifying the set of services that should be provided, instead of explicitly specifying which protocol should be instantiated. The remaining parameters define, respectively, the protocol identifier, a boolean indicating if the protocol should be initialized, a boolean indicating if a currently running instance should be returned, the Host object representing the address and port of the protocol, the required properties from the protocol, and the properties related with the running configuration. Section 4.2.1 provides a detailed description about the parameters required when requesting a protocol instantiation.

When the request for instantiating the protocol is performed it is possible to observe that, not only the set of services is provided, but also a Properties object containing the required properties. This object is populated above and guarantees that the protocol returned addresses two properties: it is based on an unstructured overlay network and provides a static peer sampling service. These service-related properties are discussed in Section 3.2, where each service is described in detail.

A closer look at the code responsible for instantiating a protocol reveals other particular aspect: both the parameter related with the address and port of the protocol and the one related with the running configurations are defined as null. This is explained by the fact that, in our solution, the network location may be specified in the configuration file (or as command-line argument), allowing the Protocol Manager to know the network location of a protocol when instantiating it. Moreover, the running configurations of a protocol might also not be provided and, in that situation, the default ones defined in the configuration file are used. As discussed before, the required properties from a protocol should not be confused with the running configurations of the protocol instance. Whereas the first are related with the protocol properties regarding the services provided (e.g., if the peer sampling service is static or dynamic), the second are related with the configuration of the protocol (e.g., the size of the views or the timeouts).

After the instantiation of the protocol, the interface for requesting the operations related with Membership Management is obtained from the `ProtocolInformation` object and a `Join` request is performed with the contact node provided as parameter. If the request is performed with an empty set the node will act as the first node in the network. The application then starts issuing requests to the peer sampling protocol through the Membership Management interface in two minutes intervals, also waiting two minutes before the first request. It is important to note that, in this example, the application relies on the blocking interaction mechanism to request the neighbors of the node, therefore the call to the `getNeighborsRequestBlocking` method blocks the execution of the thread until the results are available. After that, the results are presented in the console, and the next request is performed.

Listing 4.9: Peer sampling application developed exclusively with Babel

```java
public class PeerSamplingAppBabel extends GenericProtocol {
   public static final short APPLICATION_ID = 100;
   public static final String APPLICATION_NAME = "Peer␣Sampling␣Application";
   private static final long REQUEST_TIME = Duration.ofMinutes(2).toMillis();

   private final Host contact;
   private final short peerSamplingProtocolId;

   public PeerSamplingAppBabel(Host contact, short peerSamplingProtocolId)
         throws HandlerRegistrationException {
      super(APPLICATION_NAME, APPLICATION_ID);

      this.contact = contact;
      this.peerSamplingProtocolId = peerSamplingProtocolId;

      registerReplyHandler(GetNeighborsReply.REPLY_ID, this::uponGetNeighborsReply);
      registerTimerHandler(PeerSamplingTimer.TIMER_ID, this::uponPeerSamplingTimer);
   }

   @Override
   public void init(Properties properties) throws HandlerRegistrationException,
       ↪ IOException {
      Set<Host> contacts = new HashSet<>();

      if (contact != null)
         contacts.add(contact);

      sendRequest(new JoinRequest(contacts), peerSamplingProtocolId);
      setupTimer(new PeerSamplingTimer(), Duration.ofMinutes(2).toMillis());
   }

   private void uponPeerSamplingTimer(PeerSamplingTimer timer, long timerId) {
      sendRequest(new GetNeighborsRequest(null), peerSamplingProtocolId);
   }

   private void uponGetNeighborsReply(GetNeighborsReply reply, short sourceProto) {
      System.out.println("Received␣neighbors:␣" + reply.getNeighbors());
      setupTimer(new PeerSamplingTimer(), REQUEST_TIME);
   }
}
```

Listings 4.9 and 4.10 present the implementation of the same peer sampling application discussed before, but based only on Babel. In this case the application needs to be developed itself as a Babel protocol to be able to communicate, using requests and replies provided by Babel, with the underlying peer sampling protocol. This is what explains the need for two classes: the class with the application logic implemented as a Babel protocol and a class containing the Java Main method responsible for instantiating Babel, instantiate and register the required protocols (the application and the HyParView peer sampling protocol), initialize them, and start Babel.

The implementation presented in Listing 4.9, has the same behavior as the one presented in Listing 4.8. To implement the required behavior, the application relies on

Listing 4.10: Peer sampling application developed exclusively with Babel (Main class)

```java
public class PeerSamplingApp {
   private static final String CONFIG_FILE = "config.properties";
   private static final short PEER_SAMPLING_PROTOCOL_ID = 10000;

   public static void main(String[] args) throws Exception {
      Host myself = HostUtils.getHostFromString(args[0]);
      Host contact = null;

      if (args.length > 1)
         contact = HostUtils.getHostFromString(args[1]);

      String[] babelArgs;
      //Removes the first arguments witch are not Babel configs
      if (contact != null)
         babelArgs = Arrays.stream(args).skip(2).toArray(String[]::new);
      else
         babelArgs = Arrays.stream(args).skip(1).toArray(String[]::new);

      Babel babel = Babel.getInstance();
      Properties properties = Babel.loadConfig(babelArgs, CONFIG_FILE);

      HyparView hyparView = new HyparView(properties, PEER_SAMPLING_PROTOCOL_ID, myself);
      PeerSamplingAppBabel peerSamplingApp = new PeerSamplingAppBabel(contact,
          ↪ PEER_SAMPLING_PROTOCOL_ID);

      babel.registerProtocol(hyparView);
      babel.registerProtocol(peerSamplingApp);
      hyparView.init(properties);
      peerSamplingApp.init(properties);
      babel.start();
   }
}
```

defining handlers for *i)* the trigger of a timer responsible for sending the peer sampling requests, and *ii)* the replies received from the peer sampling protocol in response to the requests performed. It is important the note that, in addition to the ones presented here, a Java class representing the timer used to trigger the execution of a new request was also required but, due to space constraints and simplicity, it is not presented here.

Regarding the constructor of the application, it not only registers the required handlers but also receives the identifier of the underlying peer sampling protocol to use as well as the contact node to be used when joining the network. The application initialization is then performed by issuing a request to join the network and setting a timer for scheduling the first peer sampling request.

Concerning the submission of requests to the underlying protocol to obtain a sample of peers, this logic is implemented in the handler executed when the timer is triggered. Therefore, when the timer is triggered by Babel, the request is sent using the Babel `sendRequest` method. Finally, when receiving a reply from the underlying protocol to the previously made request, the results are presented in the console and a timer is set again to schedule the next request.

By comparing both implementations of the application some aspects are worth discussing. First, the code complexity of the version relying solely on Babel is higher that the one implemented with our solution. This is explained not only by the higher number of classes and lines of code (even in this simple example), but also by the need of understanding more complex aspects like asynchronous operations, handlers registration, and the operation of Babel. The necessity of developing the application as a Babel protocol also contributes for this increase in complexity, specially for programmers that are only interested in using existent decentralized protocols in applications in opposition to developing new ones.

Another relevant aspect is the capability of choosing a decentralized protocol by the services and properties required, instead of explicitly defining what protocol should be instantiated. This is presented in the example using our solution but, in the example relying on Babel, the HyParView protocol needed to be explicitly specified. Moreover, with our solution, even if the protocol was explicitly specified (or if the properties required changed, leading to the choice of another peer sampling protocol) the application would not suffer any additional changes as the interfaces remained the same.

Furthermore, our solution provides many distinct approaches for interacting with the operations exposed by the interfaces of the services. These approaches include blocking operations (as the one employed here), a Future-based approach, and an asynchronous approach through the specification of a handler to be executed when the results of a specific operation are available. Additionally, an asynchronous approach based on requests and replies (registered for each type of reply) provided by Babel is also available. By comparison, the implementation relying exclusively on Babel only provides the last model of interaction.

As an example, in this situation relying on a blocking approach for interacting with the peer sampling service simplifies the development of the application and the resulting code as the application does not require any asynchronous mechanism. In fact, this is the case in many applications leaving developers with the burden of implementing more complex, time-consuming, and error-prone asynchronous logic, where a synchronous mechanism would be more simple and easy to use. Finally, by relying on the approach exposed by our solution a timeout can be defined for blocking operations, through the configuration file, protecting applications of waiting indefinitely for the results to arrive. To implement the same behavior on the second application, more logic would need to be implemented directly in the application, namely by implementing more timers.

4.4 Summary

In this chapter we have presented a reference implementation for the solution proposed in Chapter 3 based on the Java programming language and the Babel framework. Our implementation aims at allowing applications to easily instantiate, interact, and manage protocols capable of providing decentralized services, by relying on a set of generic abstractions devised by service, instead of relying on protocol specific interfaces and mechanisms.

It is worth noting that in this reference implementation we implemented all interaction models (synchronous and asynchronous) between applications and protocols considered in the solution architecture. The implementation presented here can also be provided as a standalone Java package, e.g., available in a Maven [41] repository, to be used by applications who wish to leverage on the abstractions and mechanisms implemented.

With this implementation we strived to develop an easy-to-use solution for applications that rely on decentralized services for their operation, e.g., in order to take advantage of the edge computing paradigm. We also expect this solution to be easily extended with both new decentralized services and protocols, in addition to those already devised and implemented.

The implementation of each component was presented in detail, including details on the applications developed leveraging on our solution as well as available protocols. As explained before, although in this chapter we have presented a reference implementation based on a concrete programming language and framework we believe that our solution is sufficiently generic to be implemented in a variety of languages and frameworks.

Finally, we completed the chapter with a comparison between the implementation of an application using our solution *versus* relying solely on mechanisms provided by the Babel framework where it was possible to observe the main advantages of the architecture proposed.

In the next chapter we present an experimental evaluation of the solution, based on the reference implementation and applications whose development was described here.

5 Evaluation

In this chapter the experimental evaluation of our middleware solution is presented. This evaluation focus both on the impact on performance and code complexity of the use of our solution. Our objective is to study, on one hand, the performance impact of the abstractions proposed in this document and, on the other hand, its influence on the complexity of the code. This way we are able to compare the effect of our solution on the performance of the applications *versus* the benefits it brings for programmers when developing applications that rely on services provided by decentralized protocols.

The chapter is structured as follows: in Section 5.1 we describe the experimental settings in which the evaluation took place. Then, in sections 5.2 and 5.3 we present and discuss the results obtained in terms of performance and code complexity, respectively. The chapter ends with a summary in Section 5.4.

5.1 Experimental Settings

The evaluation of our solution was performed based on the reference implementation proposed in Chapter 4, in particular considering the developed applications that take advantage of our architecture for interacting with the services provided by decentralized protocols. All performance tests were executed on the DI/NOVA-LINCS Research Cluster, using machines with the following specifications: 2 x AMD EPYC 7343 CPUs with a total of 32 cores and 64 threads, 128 GB of DDR4 memory at 3200 MHz, 450 GB of internal SSD storage, and a 2x10 Gbps network connection. All cluster machines used for the experiments were co-located.

For performing the evaluation, we relied on the applications whose implementation is discussed in Section 4.2.4. For the performance evaluation the routing and dissemination applications were considered whereas for the evaluation of code complexity all applications were taken into account. To compare both performance and code complexity, a second version of the applications was developed relying only on Babel [19] mechanisms, instead of the programming interfaces provided by our solution. As discussed before when presenting the applications in Section 4.2.4, for the routing and dissemination applications two distinct implementations were developed: an interactive one and an automated one. Hence, for these applications only the automated implementations were considered for evaluation and, as so, only those were also developed without relying on the proposed abstractions.

For the version without abstractions, applications were developed as Babel protocols as they needed access to the mechanisms for an asynchronous and event-based interaction exposed by Babel to interact directly with the protocols through requests and replies. No Protocol Manager or programming interfaces are available, and the applications, as well as the protocols, were manually instantiated and registered in the Babel framework before starting their operation. No mechanisms related with the instantiation of a protocol by providing the set of services and properties are available in these applications. Figures 4.9 and 4.10, discussed in detail in Section 4.3, provide an example of how the applications relying only on mechanisms exposed by Babel were implemented.

Regarding the applications considered for the performance evaluation, both versions of the routing and dissemination applications perform a series of requests to execute an operation on a decentralized protocol. The first one performs requests, in a closed-loop, for routing a (randomly generated) query through the network and expects a set of nodes matching the query to be returned. The second one requests the dissemination of messages, of a given size, throughout the network in fixed time intervals.

The Kademlia [42] protocol was used as routing protocol for the routing application, whereas the Plumtree [28] protocol (with HyParView [30] as peer sampling protocol) was used as dissemination protocol for the dissemination application. The interaction between Plumtree and HyParView, when relying on abstractions, is performed as described in Section 4.2.3.1. It is worth noting that we decided to use HyParView as the peer sampling protocol for Plumtree due to its capability of providing a *static* peer sampling service, as discussed in Section 3.2.1.3. The Plumtree protocol benefits from relying on a peer sampling service with these characteristics as discussed in [28].

Both the routing and dissemination applications were initialized similarly. First, the mechanisms responsible for launching applications on each machine were executed in serial with a delay of 5 seconds between each execution. Then, on each machine, all the nodes were also started in serial with a delay of 1 second between each process launch. Regarding the definition of the contact nodes, necessary for the operation of decentralized protocols, it was performed as described next. The first process launched on each machine received the first process launched on the previous machine as contact while, in the following ones, the previously (locally) started process was defined as contact. A special case happens when launching the first process on the first machine as this is the first node on the network and, therefore, no contact node was defined.

5.2 Performance Evaluation

In this section we present the performance evaluation of our approach by comparing both routing and dissemination applications against their respective versions that only rely on Babel. To evaluate the performance of our solution each application was executed for a predefined amount of time and a set of metrics, described in detail in each of the following sections, were considered. The results presented for each of the test conditions were, in all cases, obtained from an average of multiple test executions and, when deemed necessary, the confidence intervals (with a level of confidence of 95%) are also presented. Also, the retrieval of the metrics for each test was performed by parsing the logs obtained from each execution, after the test has been completed, in order to not contaminate the execution of the tests with the load of computing or storing metric-related data.

A comparison between initialization times was also performed to assess the impact of the Protocol Manager and the abstraction layer composed by the programming interfaces. We consider the initialization time as the period between the request for the instantiation of a protocol, both through the Protocol Manager or directly, and the moment when the protocol is available to be used by an application.

In the following we describe the evaluation performed for each one of the routing and dissemination applications. The metrics considered in each case are presented, as well as the results obtained after executing the evaluation. A discussion of the results is also provided.

5.2.1 Routing application

As described before, the routing application is responsible for performing routing requests of randomly generated queries through the network, by relying on a routing protocol like Kademlia. The application expects to receive the set of nodes matching the query provided. In our evaluation the Kademlia protocol was executed with the following configurations: $kvalue = 20$, $alpha = 3$, and the timeout for node lookups set for 10 seconds. The routing requests were carried out by the application, in a closed-loop, for 4 minutes with a start and cooldown period of 2 minutes each.

5.2.1.1 Evaluation metrics

When evaluating the performance of the routing application, the following metrics were considered.

Number of requests sent The number of requests sent by the application for performing routing operations, each with a different randomly generated query.

Number of results received The number of results received, from the routing protocol, containing the set of nodes that result from the queries provided.

Application	Nodes	Requests Sent	Responses Received	Latency (ms)	Throughput ($responses/s$)	Recall
With middleware	192	353151.3	353151.3	129.267 ± 0.834	1128.3	1.0
	384	517837.7	517837.7	176.848 ± 1.650	1580.4	1.0
	576	617627.3	617627.3	222.793 ± 0.370	1800.7	1.0
Without middleware	192	357782.3	357782.3	127.686 ± 0.723	1143.1	1.0
	384	522143.3	522143.3	175.432 ± 0.334	1591.9	1.0
	576	620851	620851	221.686 ± 2.901	1810.1	1.0

Table 5.1: Routing application performance results

Average Latency The average latency, in milliseconds, until receiving the routing results. The latency was obtained, for each request, by calculating the difference between the moment when the request was performed and the moment when the respective result arrived. Then, the average was calculated considering the latencies of all requests.

Throughput The throughput is presented in responses per second and represents the rate of routing responses arriving in a given time interval (1 second in this case). The throughput was obtained by dividing the number of results received by the period of testing (in seconds).

Recall The recall metric represents the fraction of values that are considered correct between all returned by the routing protocol. This metric was calculated by verifying, for each result obtained, if the returned set of nodes was correct and, then, dividing the number of correct results by the total number of routing results received. To verify each result, our parser considered all the nodes present on the network and compared the results obtained with the expected ones, considering the nearest nodes to the query provided, based on the XOR metric.

5.2.1.2 Discussion of results

The tests with the routing application were performed, in the conditions described above, with 192, 384, and 576 nodes split, respectively, by 3, 6 and 9 machines running 64 nodes each. Each test was performed three times and the presented results are the average of the executions. The comparison of latencies between the versions of the routing application, with and without relying on the developed abstractions, are provided in Figure 5.1. The results regarding the remaining metrics are presented in Table 5.1. Furthermore, Figure 5.2 provides a comparison between the initialization time of both application versions. The initialization time was obtained by calculating an average of the initialization times considering 192 nodes split into three machines.

Through the analysis of the results presented in Table 5.1 and Figure 5.1 it is possible to conclude that, although a slight increase in latency and, therefore, a reduction in throughput, can be observed in the application version that relies on our solution, the

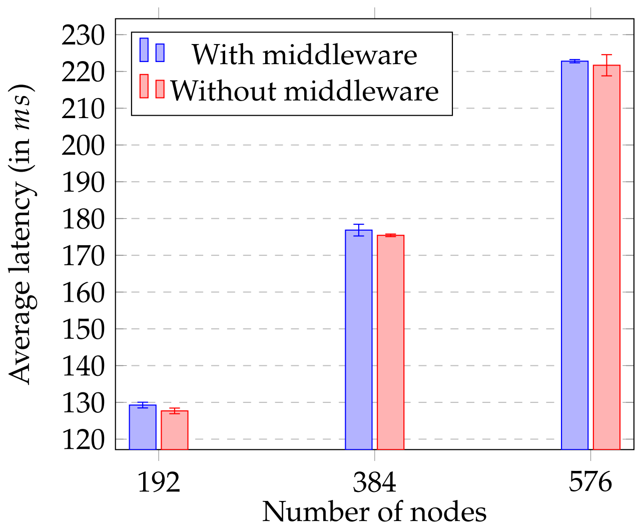


Figure 5.1: Comparison between the latency results of the routing application

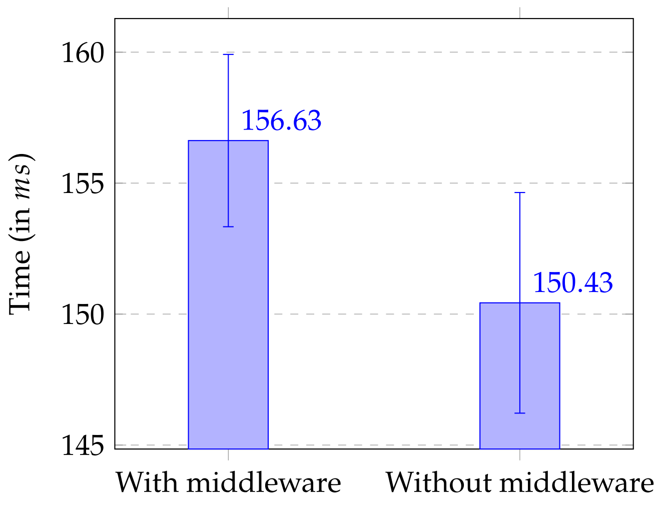


Figure 5.2: Comparison between the initialization times of the routing protocol

impact of the abstraction components is minimal. In fact, when considering the confidence interval, both solutions might even be considered equivalent in terms of performance. This result is in line with the expected behavior as we believe that the most impact on latency is on the network side, and not on the computations performed locally. Therefore, the impact that our abstractions could pose on latency, when sending requests or receiving results, due to the additional computations, is mitigated by the impact of communicating over the network. Additionally, the recall value of 1.0, obtained for all tests performed confirms that both the implemented decentralized protocols and all other components of our solution are behaving as expected, with all operation requests being correctly handled.

Regarding the time required for protocol initialization, the results show that the time to initialize a protocol relying on the abstraction mechanisms is slightly higher than its counterpart, even though when considering the confidence intervals the results can be, to some extent, deemed equivalent. This is the expected result as the mechanisms for protocol instantiation are more complex in our solution than just calling the constructor of a Java class and performing a protocol registration on Babel. However, we believe that this result does not pose a significant performance problem because *i)* although higher, we do not consider the initialization time when relying on the developed abstractions, to be very significant, and *ii)* in most situations the instantiation of protocols in only performed a few times during the execution of the applications (or even just at the start).

5.2.2 Dissemination application

Our solution was also evaluated through the use of a dissemination application. As described before, this application requests the dissemination of messages with a configurable size, throughout the network, over a fixed period of time. The application generates an identifier for each message (an UUID) and logs the moment when the dissemination request was performed. Then, when a message is received, the identifier is obtained and logged together with a timestamp.

For this application we leveraged on Plumtree as dissemination protocol and on HyParView as the underlying peer sampling protocol. The Plumtree timeout, for considering a connection as failed was configured to 7 seconds, while the timeout for awaiting a GRAFT response was defined to 3.5 seconds. The HyParView protocol relied on active and passive views of size 5 and 12, respectively. The application performs the dissemination of a message every 30 seconds, with a size of 100 KB, during a period of 4 minutes with a start and cooldown period of 2 minutes each.

5.2.2.1 Evaluation metrics

When evaluating the performance of the dissemination application, the metrics described below were considered.

Number of messages sent The total number of messages requested by the application to be disseminated throughout the network, relying on the underlying protocols. For each request, a payload with the pre-configured size was disseminated.

Total delivered messages The total number of messages delivered on the various instances of the dissemination application. As an example, if a message is disseminated by a node on a network containing 8 nodes, and all of them receive the message, we consider the total delivered messages to be 8.

Average Latency The average latency was considered as the average of the maximum latencies obtained for each disseminated message. To retrieve this metric the maximum latency for each message was obtained by calculating the difference between the moment when the message was last delivered and the moment when the dissemination request was made. Then, an average of the maximum latencies was performed to obtain the final result.

Application	Nodes	Messages Sent	Total Delivered Messages	Latency (ms)	Throughput ($messages/s$)	Reliability
With middleware	192	1536	294912	26.445 ± 0.488	614.4	1.0
	384	3072	1179648	33.821 ± 1.497	2457.6	1.0
	448	3584	1605632	40.273 ± 3.052	3345.1	1.0
Without middleware	192	1618.7	310732.8	26.624 ± 0.946	647.4	1.0
	384	3221.7	1237132.8	33.410 ± 1.088	2577.4	1.0
	448	3757.6	1683404.8	35.962 ± 2.107	3507.1	1.0

Table 5.2: Dissemination application performance results

Throughput The throughput is presented in messages per second and represents the rate of messages delivered on the system in a given time interval (1 second in this case). The throughput was obtained by dividing the number of total delivered messages by the period of testing (in seconds).

Reliability The reliability represents the fraction of disseminated messages that are correctly delivered to nodes present in the network. In our evaluation we calculated the reliability for each message by dividing the number of nodes on which the message was delivered by the total number of network nodes. An average of the reliability values was then performed to obtain the final result.

5.2.2.2 Discussion of results

The evaluation of the dissemination application was performed by executing 192, 384, and 448 nodes on 3, 6, and 7 machines respectively. Each test was repeated ten times and the results presented here, for each test condition, are the average of the executions. The higher number of tests performed, in comparison with the routing application, is explained by the higher variation in the values of metrics that is expected from this application, thus leading to the necessity of more tests to obtain comprehensive results. In Figure 5.3 we present the comparison between the latency values of both implementations of the dissemination application, with and without relying on our solution. The complete results are presented in Table 5.2. Additionally, Figure 5.4 presents the comparison between the initialization time of protocols on both implemented versions. As before, the evaluation of the initialization time was performed by calculating the average of all initialization times, considering 192 nodes split into three machines.

Considering the results on Table 5.2 and Figure 5.3 it is possible to observe similar results to the ones previously discussed for the routing application. Hence, a significant difference in performance is not observable when comparing both implementations of the dissemination application, as the results obtained can be deemed equivalent when considering the confidence intervals. As discussed before, this is the expected result as we believe that the influence of the network communication on performance, in particular when taking into account the latency values, is more significant than the influence of local computations. As a result, the impact of our solution on the overall performance is

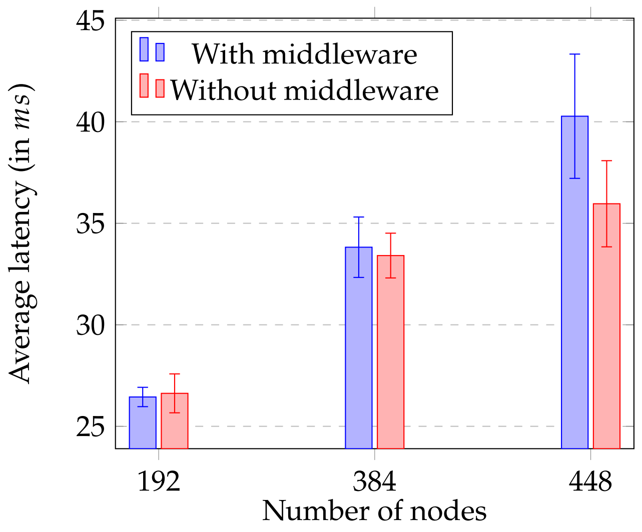


Figure 5.3: Comparison between the latency results of the dissemination application

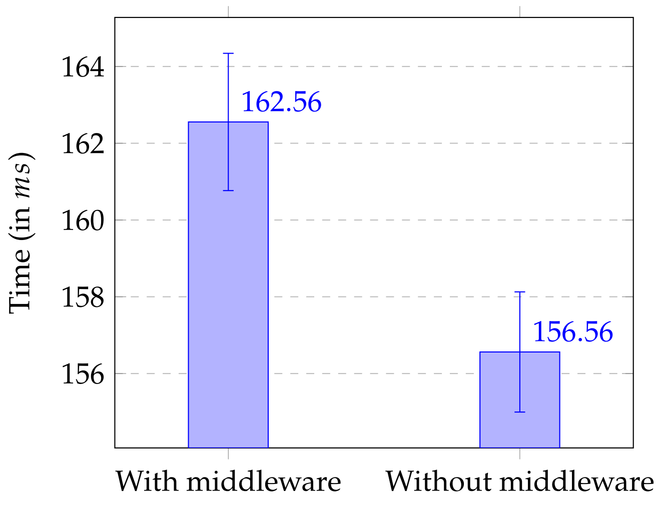


Figure 5.4: Comparison between the initialization times of the dissemination protocol

reduced. Furthermore, the reliability values of 1.0 also support the conclusion that our solution is working as expected, with all dissemination requests being correctly handled, as well as the deliveries of the messages to the applications.

A note should be made regarding the results for the messages sent and total delivered messages metrics presented in Table 5.2. In the implementation of the application without relying on the developed middleware, the results regarding those metrics present slight variations between tests. This is explained because in this implementation we relied on Babel timers to trigger the dissemination of messages, unlike its counterpart where due to the use of our programming interfaces a thread was launched for sending the requests as the application was not required to be implemented as a Babel protocol. This way, the timers responsible for the dissemination of messages may be put in the queue of events before the timer responsible for stopping the dissemination. The order of both events in the queue matters in this situation, as the last trigger of the dissemination timer is expected to happen at the same time as the trigger of the timer for canceling the dissemination. Consequently, each node can issue, at maximum, one more message than initially expected. The results presented are an average of the messages sent and delivered on each test performed.

When considering the results regarding the initialization time, presented in Figure 5.4 similar results to the ones discussed before are also observable. By comparing the initialization times of both implementations of the dissemination application it is possible to conclude that the initialization of a new protocol on the implementation relying on our middleware takes slightly longer that its counterpart. This is the expected result as more operations need to be performed in our solution, for instantiating a protocol, when compared to calling the Java constructor of a protocol and registering the instance on Babel.

The additional operations include verifications performed by the Protocol Manager (e.g., related with the identifier of the protocol) and the instantiation of the generic interfaces allowing the interaction with the services provided by the protocol. Additionally, the Protocol Manager is also required to retrieve the running configuration and store information about the execution of the new protocol. Moreover, as discussed in Section 4.2.1 the instantiation of the protocols in our solution is performed leveraging on the Java reflection mechanisms, which are known to be slower when compared to directly calling a constructor [60, 16].

5.3 Code complexity evaluation

For comparing the code complexity between the application versions with and without relying on the programming interfaces and management mechanisms developed, we performed an analysis on the number of lines of code required for implementing each version. All applications described in Section 4.2.4 were considered, by comparing the complexity of the version developed using our solution with the equivalent one relying exclusively on the Babel framework.

When counting the lines of code required to implement each version of the applications the blank lines and comments were not considered as well as the lines related with imports and package definitions in Java classes. In the comparison between both versions of the decentralized storage application, the lines related with the user interaction for performing the operations were also not taken into account.

The results regarding the code complexity of each application are presented in Table 5.3. The table presents the number of lines of code required to implement both versions of each application and the percentage of reduction. A positive percentage means that the implementation of the version relying on the abstractions developed is smaller, by the factor expressed in the percentage, when compared to the one that relies directly on Babel.

Through the analysis of Table 5.3, that presents the comparison in terms of code complexity between the different implementations of each application, it is possible to

Applications	Lines of Code		% of reduction
	With middleware	Without middleware	
Peer Sampling	29	64	54.7 %
Dissemination	70	132	46.9 %
Routing	66	120	45 %
Resource Storage	202	360	43.9 %

Table 5.3: Comparison between the complexity of each application

notice a significant improvement when leveraging on our solution for interacting with decentralized services. This improvement is explained by the simple mechanisms for protocol instantiation as well as the common programming interfaces exposed, providing multiple synchronous and asynchronous interaction mechanisms between protocols and applications.

As an example, when considering the Routing application, the version relying on the abstractions developed can launch a thread, active on the period during which the routing requests should be performed, to issue `FindNodes` operations in a closed-loop. The code leverages on the Futures-based interaction mechanism, requesting the operation and blocking on the returned future until the return value is available or a timeout expires. The simple instantiation of a protocol just by calling an operation exposed by the Protocol Manager, therefore obtaining the programming interfaces to interact with it and perform the necessary operations is also an advantage. Additionally, being able to implement the application without having to interact with the Babel mechanisms simplifies the development both by reducing the learning curve for the programmer and the complexity of code, as no initialize methods, specific constructors, nor event handlers are required because the application does not need to be implemented as a Babel protocol.

Conversely, the version of the application that does not rely on the abstraction layer, due to the necessity of development as a Babel protocol, needs to implement all logic based on the asynchronous mechanisms, exposed by Babel, through the implementation of handlers responsible for dealing with the replies from the `FindNodes` operation. The application is also required to register those handlers in Babel. To implement the same closed-loop behavior, the handler should send the next `FindNodes` request to the protocol when the last one is received. Moreover, the start and stop of the routing requests needs to be managed by a Babel timer, which requires the development of one more Java class and the respective timer handler.

When considering more complex examples, like the Resource Storage application, the advantages of our solution become even more evident as not only the number of lines of code is reduced, but also the implementation of the application becomes easier for the developer. As an example, when a node wants to add a new file a set of operations might need to be performed, as discussed in Section 4.2.4.4. These operations include instantiating a protocol to disseminate the notifications related with the file, performing a `Join` operation in the dissemination protocol using the nearest nodes to the file identifier as contact nodes, and disseminating the notification throughout the network.

Implementing the logic described before requires coordination between distinct protocols and operations. An example of coordination is obtaining of the nearest nodes to the file identifier, which needs to be done before requesting the `Join` operation from the dissemination protocol, as the nodes retrieved will be used as contacts. The coordination between different operations leverages on the blocking operations exposed by our solution instead of dealing with an asynchronous interaction model that requires more complex logic and is more error-prone.

Additionally, in more complex applications that require multiple decentralized protocols to work properly, having a single component for managing all of them (in this case the Protocol Manager) is also an advantage. Moreover, relying on generic interfaces for interacting with the services provided by the protocols not only simplifies the development, but also contributes to the maintainability by allowing the change from one protocol to another without profound application changes.

Another advantage of using the solution proposed in this dissertation is related with the distribution of logic between different classes. In the implementation of the Resource Storage application leveraging on our solution the application logic is distributed between two classes: one responsible for handling the input from the user, and another responsible for the application logic itself, e.g., instantiating the protocols, calling the operations from the services-based interfaces, or handling the notifications. Implementing the same logic in the version relying exclusively on Babel is more complex as all classes need to be implemented as Babel protocols. Consequently, two Babel protocols needed to be implemented both for user interaction and application logic, with the communication between them being performed through asynchronous mechanisms like requests, replies, and notifications. This not only translates into more complex logic but also requires the development of more classes, e.g., the version implemented with our solution required the development of three Java classes `versus` the nine implemented using only Babel.

In Section 4.3, we further discussed the differences between using the solution proposed in this dissertation and relying only on mechanisms exposed by the Babel framework through a comparison between both implementations of the peer sampling application.

5.4 Summary

The evaluation presented in this chapter allow us to study the impacts of our solution for decentralized abstractions both in terms of performance and code complexity. By analyzing the results it is possible to conclude that, although a slight decrease in performance may occur due to the existence of a middleware layer between applications and protocols (even though the results can be considered equivalent when taking into consideration the confidence intervals), it is compensated by the simplified service-based model of interaction. Therefore, we believe that our solution significantly simplifies the development process of applications that require the use of decentralized protocols, e.g., for relying on an edge computing approach, by leveraging on a set of standardized components that make applications easier to develop and maintain, even if changes are needed in the future.

The comparison performed on the code complexity, considering both implementations of each application, shows that the proposed solution allows a simple development of applications as the evaluation results show a significant decrease in the number of lines of code that are required. Furthermore, programmers also benefit from having access to generic (service-based) interfaces that increase maintainability and provide multiple interaction models, both synchronous and asynchronous, allowing the usage of the most appropriate approach. Also, relying on protocol management mechanisms can be beneficial, in particular when considering more complex applications that require multiple decentralized protocols.

In summary, we believe that the upsides of using the middleware solution presented in this dissertation, for developing and maintaining decentralized applications, largely exceed the possible downsides of a slight decrease in performance. In the next chapter we present the conclusions and future work regarding this dissertation.

6 Conclusions and Future Work

Nowadays, many applications rely on decentralized approaches to improve reliability, fault-tolerance, and scalability due to the lack of a single point of failure as well as the ability to distribute computations through multiple machines. In some cases, decentralized architectures even provide useful privacy guarantees. These applications leverage on the use of protocols, many of which developed in the context of peer-to-peer (P2P) systems, that provide a set of decentralized services related with membership management and communication. In fact, some applications use these services for relying on an edge computing approach, allowing computations to be performed closer to clients, therefore reducing latency and distributing the load on the system.

Although a significant number of protocols already exist for providing the required services, some of which were discussed before in this dissertation, each one exposes a specific interface to interact with it. This happens, not only when comparing protocols providing similar services, but even when considering distinct implementations of the same protocol. These protocol-based interfaces difficult the development of decentralized applications and reduce their maintainability, e.g., by leading a protocol change, even for another one providing the same set of services, to trigger extensive modifications on an application. This gains even more importance when considering applications that rely on multiple decentralized protocols for their operation. Furthermore, the existence of generic mechanisms to simplify, not only the interaction between applications and protocols, but also the instantiation and management of the decentralized protocols running on a process, independently of their operation or services provided, would be beneficial.

In this dissertation we performed a study on a set of decentralized protocols, regarding the services provided, operations exposed, as well as their properties. Then, we developed a set of generic abstractions (or programming interfaces) that can be leveraged to interact with them, by relying on a service-based approach in opposition to a protocol-based one. Our solution provides multiple interaction mechanisms for requesting operations from decentralized protocols, employing both synchronous and asynchronous approaches. Programmers can also request a decentralized service to be provided even without knowledge of the specific protocols providing it, as the instantiation of protocols can be requested just by defining the service(s) and (optionally) the properties required. In summary, by combining the devised interfaces with the mechanisms developed for managing decentralized protocols running on a system, a middleware solution based on multiple components was developed.

We provided a reference implementation of the solution proposed, developed in Java, based on the Babel framework. Finally, an evaluation was performed, leveraging on the implementation mentioned before, to assess the impact of the solution on the performance and code complexity of applications. The results showed that the improvements in terms of code complexity are noticeable without a relevant impact on key performance indicators of applications.

6.1 Future Work

In this section we present the work that might be developed in the future based on the solution proposed in this dissertation.

Definition of more services and respective generic interfaces More abstractions could be devised in order to accommodate protocols providing different services to applications. This would require a study of more protocols, providing services that are distinct from the ones presented in this document, and the definition of the respective programming interfaces. The extension mechanisms, already in place on the solution proposed here, could be leveraged to implement those interfaces. Moreover, the currently devised service-based interfaces can be extended if required, as an example, by adding new operations and/or more parameters to the current ones.

Integration of more protocols The development of more protocols, providing the services presented in this dissertation or new ones, could also be performed. As in the above case, the existing extension mechanisms to simplify the registration of new protocols in our solution could be leveraged to adapt current implementations of decentralized protocols to our solution.

Development of a distributed mechanism for protocol management The proposal, in this dissertation, of generic mechanisms for instantiating decentralized protocols also opens the possibility of developing a decentralized control mechanism for remotely managing the protocols running on each node. This mechanism might take the form of a special protocol, running on nodes leveraging on our solution, that allows performing requests for the remote instantiation of decentralized protocols, providing the required services. With the remote management mechanism, a node can not only request the local instantiation of a protocol using the generic abstractions presented in this work, but also request the instantiation of the protocol on another node. However, this approach might lead to security challenges, namely the ones related with the possibility of performing attacks targeting the availability of nodes, e.g., by forcing the execution of a significant number of protocols on a node.

Development of mechanisms for dynamic adaptation of protocols The development of generic abstractions for interacting with decentralized protocols, through common service-based interfaces, also paves the way for developing dynamic adaptation mechanisms. These mechanisms would be capable of adapting the protocol in use in response to changes in the operational conditions, e.g., if a network becomes more unstable. This could lead to a protocol being changed to another one, providing the same set of services, but more suitable to the new conditions without any impact on the operation of the upper application, as the interface for interaction remains the same. In fact, we can even consider situations where parts of a network are maintained relying on a given protocol while others are maintained by a different one, if the two are capable of being coupled together, while the abstractions allowing applications to interact with those protocols remain the same.

Bibliography

[1] D. Anderson et al. “SETI@home: An Experiment in Public-Resource Computing”. In: *Commun. ACM* 45 (2002-11), pp. 56–61. DOI: 10.1145/581571.581573 (cit. on pp. 8, 21).

[2] T. Arnold et al. “(How Much) Does a Private WAN Improve Cloud Performance?” In: *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*. 2020, pp. 79–88. DOI: 10.1109/INFOCOM41043.2020.9155428 (cit. on p. 2).

[3] I. Baumgart et al. “OverArch: A common architecture for structured and unstructured overlay networks”. In: 2012-03, pp. 2534–2539. ISBN: 9781467310178. DOI: 10.1109/INFCOMW.2012.6193490 (cit. on pp. 4, 11, 28, 34).

[4] J. Benet. “IPFS - Content Addressed, Versioned, P2P File System”. In: (2014-07) (cit. on p. 72).

[5] B. Beverly Yang and H. Garcia-Molina. “Designing a super-peer network”. In: *Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405)*. 2003, pp. 49–60. DOI: 10.1109/ICDE.2003.1260781 (cit. on p. 16).

[6] J. Cao and M. Yang. “Energy Internet – Towards Smart Grid 2.0”. In: *2013 Fourth International Conference on Networking and Distributed Computing*. 2013, pp. 105–110. DOI: 10.1109/ICNDC.2013.10 (cit. on p. 2).

[7] B. Carlsson and R. Gustavsson. “The Rise and Fall of Napster - An Evolutionary Approach”. In: *Active Media Technology*. Ed. by J. Liu et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 347–354. ISBN: 978-3-540-45336-9 (cit. on p. 10).

[8] Y. Chawathe et al. “Making Gnutella-like P2P Systems Scalable”. In: *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. SIGCOMM ’03. Karlsruhe, Germany: Association for Computing Machinery, 2003, pp. 407–418. ISBN: 1581137354. DOI: 10.1145/863955.864000. URL: https://doi.org/10.1145/863955.864000 (cit. on pp. 3, 10, 11, 13, 16–19, 23, 34, 38).

[9] Cisco. *Fog Computing and the Internet of Things: Extend the Cloud to Where the Things Are*. Tech. rep. 2015 (cit. on pp. 3, 7, 8, 10).

[10] I. Clarke et al. "Freenet: A Distributed Anonymous Information Storage and Retrieval System". In: *Lecture Notes in Computer Science* 2009 (2001-03). DOI: 10.1007/3-540-44702-4_4 (cit. on pp. 1, 3, 9, 10, 13, 19, 25, 34, 40).

[11] M. Conoscenti, A. Vetrò, and J. C. De Martin. "Peer to Peer for Privacy and Decentralization in the Internet of Things". In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 2017, pp. 288–290. DOI: 10.1109/ICSE-C.2017.60 (cit. on pp. 1, 2, 9).

[12] P. Á. Costa, P. Fouto, and J. Leitão. *Edge Overlays - Overlay Protocols for Edge Computing Management*. https://github.com/pedroAkos/EdgeOverlayNetworks (cit. on p. 68).

[13] P. Á. Costa, P. Fouto, and J. Leitão. "Overlay Networks for Edge Management". In: *2020 IEEE 19th International Symposium on Network Computing and Applications (NCA)*. 2020, pp. 1–10. DOI: 10.1109/NCA51143.2020.9306716 (cit. on p. 68).

[14] F. Dabek et al. "Towards a Common API for Structured Peer-to-Peer Overlays". In: vol. 2003. 2003-11. ISBN: 978-3-540-40724-9. DOI: 10.1007/978-3-540-45172-3_3 (cit. on pp. 4, 11, 28, 29, 34).

[15] P. T. Eugster et al. "The Many Faces of Publish/Subscribe". In: *ACM Comput. Surv.* 35.2 (2003-06), pp. 114–131. ISSN: 0360-0300. DOI: 10.1145/857076.857078. URL: https://doi.org/10.1145/857076.857078 (cit. on p. 20).

[16] B. Evans. *The performance implications of Java reflection*. 2023-04. URL: https://blogs.oracle.com/javamagazine/post/java-reflection-performance (cit. on pp. 59, 93).

[17] I. Foster and A. Iamnitchi. "On Death, Taxes, and the Convergence of Peer-to-Peer and Grid Computing". In: *Peer-to-Peer Systems II*. Ed. by M. F. Kaashoek and I. Stoica. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 118–128. ISBN: 978-3-540-45172-3 (cit. on p. 21).

[18] P. Fouto et al. *Babel*. https://github.com/pfouto/babel-core (cit. on p. 70).

[19] P. Fouto et al. "Babel: A Framework for Developing Performant and Dependable Distributed Protocols". In: *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. Los Alamitos, CA, USA: IEEE Computer Society, 2022-09, pp. 146–155. DOI: 10.1109/SRDS55811.2022.00022. URL: https://doi.ieeecomputersociety.org/10.1109/SRDS55811.2022.00022 (cit. on pp. 25–27, 52, 53, 70, 85).

[20] *Future*. 2022-10. URL: https://learn.microsoft.com/en-us/cpp/standard-library/future?view=msvc-170 (cit. on p. 47).

[21] *Future*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html (cit. on p. 47).

[22] *Futures*. URL: https://doc.rust-lang.org/std/future/trait.Future.html (cit. on p. 47).

[23] I. Gupta et al. "Kelips: Building an Efficient and Stable P2P DHT through Increased Memory and Background Overhead". In: *Peer-to-Peer Systems II*. Ed. by M. F. Kaashoek and I. Stoica. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 160–169. ISBN: 978-3-540-45172-3 (cit. on pp. 3, 10, 23, 34).

[24] P. Haller et al. *Futures and promises*. URL: https://docs.scala-lang.org/overviews/core/futures.html (cit. on p. 47).

[25] D. Hughes, G. Coulson, and J. Walkerdine. "Free riding on Gnutella revisited: the bell tolls?" In: *IEEE Distributed Systems Online* 6.6 (2005). DOI: 10.1109/MDSO.2005.31 (cit. on pp. 16, 21, 23, 38).

[26] M. Jelasity, A. Montresor, and O. Babaoglu. "T-Man: Gossip-based fast overlay topology construction". In: *Computer Networks* 53.13 (2009). Gossiping in Distributed Systems, pp. 2321–2339. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2009.03.013. URL: https://www.sciencedirect.com/science/article/pii/S1389128609001224 (cit. on pp. 4, 11).

[27] Y.-J. Kim et al. "A secure decentralized data-centric information infrastructure for smart grid". In: *IEEE Communications Magazine* 48.11 (2010), pp. 58–65. DOI: 10.1109/MCOM.2010.5621968 (cit. on p. 2).

[28] J. Leitao, J. Pereira, and L. Rodrigues. "Epidemic Broadcast Trees". In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. 2007, pp. 301–310. DOI: 10.1109/SRDS.2007.27 (cit. on pp. 3, 10, 20, 24, 34, 42, 69, 75, 86).

[29] J. Leitão. "Topology Management for Unstructured Overlay Networks". PhD thesis. Technical University of Lisbon, 2012-09 (cit. on pp. 1–4, 9–20, 23, 24).

[30] J. Leitão, J. Pereira, and L. Rodrigues. "HyParView: a membership protocol for reliable gossip-based broadcast". In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. Edinburgh, UK, 2007-06, pp. 419–429 (cit. on pp. 3, 10, 11, 13–16, 20, 24, 32, 34, 57, 86).

[31] J. Leitão, J. Pereira, and L. Rodrigues. "Gossip-Based Broadcast". In: 2010-10, pp. 831–860. ISBN: 978-0-387-09750-3. DOI: 10.1007/978-0-387-09751-0_29 (cit. on pp. 11, 24, 34, 37).

[32] J. Leitão et al. "On Adding Structure to Unstructured Overlay Networks". In: *Handbook of Peer-to-Peer Networking*. Ed. by X. Shen et al. Boston, MA: Springer US, 2010, pp. 327–365. ISBN: 978-0-387-09751-0. DOI: 10.1007/978-0-387-09751-0_13. URL: https://doi.org/10.1007/978-0-387-09751-0_13 (cit. on pp. 11, 13).

[33] J. Leitão et al. *Towards Enabling Novel Edge-Enabled Applications*. 2018. DOI: 10.48550/ARXIV.1805.06989. URL: https://arxiv.org/abs/1805.06989 (cit. on pp. 3, 8–10).

[34] J. C. A. Leitão and L. E. T. Rodrigues. "Overnesia: A Resilient Overlay Network for Virtual Super-Peers". In: *2014 IEEE 33rd International Symposium on Reliable Distributed Systems*. 2014, pp. 281–290. DOI: 10.1109/SRDS.2014.40 (cit. on pp. 3, 11, 13, 15, 16).

[35] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cit. on p. ii).

[36] D. Lucke, C. Constantinescu, and E. Westkämper. "Smart Factory - A Step towards the Next Generation of Manufacturing". In: 2008-01, pp. 115–118. ISBN: 978-1-84800-266-1. DOI: 10.1007/978-1-84800-267-8_23 (cit. on p. 2).

[37] A. S. T. Maarten van Steen. *Distributed Systems*. 4th ed. distributed-systems.net, 2023. Chap. 1 (cit. on pp. 1, 2).

[38] A. S. T. Maarten van Steen. *Distributed Systems*. 4th ed. distributed-systems.net, 2023. Chap. 2 (cit. on pp. 3, 4, 8, 9, 11–13, 15, 19–21, 24).

[39] A. S. T. Maarten van Steen. *Distributed Systems*. 4th ed. distributed-systems.net, 2023. Chap. 4 (cit. on pp. 19, 20).

[40] R. Mahesa. *How cloud, fog, and mist computing can work together*. 2018-03. URL: https://developer.ibm.com/articles/how-cloud-fog-and-mist-computing-can-work-together/ (cit. on pp. 8–10).

[41] *Maven*. URL: https://maven.apache.org (cit. on p. 84).

[42] P. Maymounkov and D. Mazières. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *Peer-to-Peer Systems*. Ed. by P. Druschel, F. Kaashoek, and A. Rowstron. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 53–65. ISBN: 978-3-540-45748-0 (cit. on pp. 3, 10, 13, 15, 17, 22, 32, 34, 40, 68, 72, 86).

[43] R. Melamed and I. Keidar. "Araneola: a scalable reliable multicast system for dynamic environments". In: *Third IEEE International Symposium on Network Computing and Applications, 2004. (NCA 2004). Proceedings*. 2004, pp. 5–14. DOI: 10.1109/NCA.2004.1347755 (cit. on pp. 10, 34, 42).

[44] H. Miranda, A. Pinto, and L. Rodrigues. "Appia, a flexible protocol kernel supporting multiple coordinated channels". In: *Proceedings 21st International Conference on Distributed Computing Systems*. 2001, pp. 707–710. DOI: 10.1109/ICDSC.2001.919005 (cit. on pp. 25, 26).

[45] E. Patti et al. "Distributed Software Infrastructure for General Purpose Services in Smart Grid". In: *IEEE Transactions on Smart Grid* 7.2 (2016), pp. 1156–1163. DOI: 10.1109/TSG.2014.2375197 (cit. on p. 2).

[46] P. Poonpakdee, J. Koiwanit, and C. Yuangyai. “Decentralized Network Building Change in Large Manufacturing Companies towards Industry 4.0”. In: *Procedia Computer Science* 110 (2017). 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops, pp. 46–53. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2017.06.113. URL: https://www.sciencedirect.com/science/article/pii/S1877050917312929 (cit. on p. 2).

[47] J. Pouwelse et al. “The Bittorrent P2P File-Sharing System: Measurements and Analysis”. In: vol. 3640. 2005-02, pp. 205–216. ISBN: 978-3-540-29068-1. DOI: 10.1007/11558989_19 (cit. on pp. 2, 9, 10, 21).

[48] G. Press. *The State Of Data, December 2020*. 2020. URL: https://www.forbes.com/sites/gilpress/2021/12/27/the-state-of-data-december-2020/?sh=6d2fbc333462 (cit. on pp. 2, 7).

[49] Protocol Labs. *Libp2p*. URL: https://libp2p.io/ (cit. on pp. 25, 26).

[50] Protocol Labs. *Libp2p Connectivity*. URL: https://connectivity.libp2p.io/ (cit. on p. 26).

[51] Protocol Labs. *Libp2p Implementations*. URL: https://libp2p.io/implementations/ (cit. on p. 26).

[52] J. Qin, Y. Liu, and R. Grosvenor. “A Categorical Framework of Manufacturing for Industry 4.0 and Beyond”. In: *Procedia CIRP* 52 (2016). The Sixth International Conference on Changeable, Agile, Reconfigurable and Virtual Production (CARV2016), pp. 173–178. ISSN: 2212-8271. DOI: https://doi.org/10.1016/j.procir.2016.08.005. URL: https://www.sciencedirect.com/science/article/pii/S221282711630854X (cit. on p. 2).

[53] M. Ripeanu. “Peer-to-peer architecture case study: Gnutella network”. In: *Proceedings First International Conference on Peer-to-Peer Computing*. 2001, pp. 99–100. DOI: 10.1109/P2P.2001.990433 (cit. on pp. 23, 38).

[54] S. K. Routray et al. “Satellite Based IoT for Mission Critical Applications”. In: *2019 International Conference on Data Science and Communication (IconDSC)*. 2019, pp. 1–6. DOI: 10.1109/IconDSC.2019.8817030 (cit. on p. 2).

[55] A. Rowstron and P. Druschel. “Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems”. In: *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*. Vol. 2218. Springer Berlin Heidelberg, 2001-11. Chap. Middleware 2001, pp. 329–350. ISBN: 978-3-540-45518-9. URL: https://www.microsoft.com/en-us/research/publication/pastry-scalable-distributed-object-location-and-routing-for-large-scale-peer-to-peer-systems/ (cit. on p. 20).

[56] A. Rowstron et al. "Scribe: The Design of a Large-Scale Event Notification Infrastructure". In: *Networked Group Communication*. Ed. by J. Crowcroft and M. Hofmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 30–43. ISBN: 978-3-540-45546-2 (cit. on p. 20).

[57] I. Stoica et al. "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications". In: *SIGCOMM Comput. Commun. Rev.* 31.4 (2001-08), pp. 149–160. ISSN: 0146-4833. DOI: 10.1145/964723.383071. URL: https://doi.org/10.1145/964723.383071 (cit. on pp. 3, 10, 13, 15, 17, 19, 22, 32, 34, 40).

[58] C. Tang, R. Chang, and C. Ward. "GoCast: gossip-enhanced overlay multicast for fast and dependable group communication". In: *2005 International Conference on Dependable Systems and Networks (DSN'05)*. 2005, pp. 140–149. DOI: 10.1109/DSN.2005.52 (cit. on pp. 10, 34, 42).

[59] *TaRDIS: Trustworthy and Resilient Decentralised Intelligence for Edge Systems*. URL: https://cordis.europa.eu/project/id/101093006 (visited on 2023-02-05) (cit. on pp. 2, 5).

[60] C. Tudose, C. Odubăşteanu, and S. Radu. "Java Reflection Performance Analysis Using Different Java Development". In: 187 (2013-01), pp. 439–452. DOI: 10.1007/978-3-642-32548-9-31 (cit. on pp. 59, 93).

[61] J. Verbeke et al. "Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment". In: *Grid Computing — GRID 2002*. Ed. by M. Parashar. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 1–12. ISBN: 978-3-540-36133-6 (cit. on p. 21).

[62] S. Voulgaris, D. Gavidia, and M. van Steen. "CYCLON: Inexpensive Membership Management for Unstructured P2P Overlays". In: *Journal of Network and Systems Management* 13.2 (2005), pp. 197–217. DOI: 10.1007/s10922-005-4441-x. URL: https://doi.org/10.1007/s10922-005-4441-x (cit. on pp. 3, 10, 16, 24, 32, 34, 57).

[63] *What is Pub/Sub Messaging?* URL: https://aws.amazon.com/pt/pub-sub-messaging/ (cit. on p. 20).

[64] *What is pub/sub?* URL: https://cloud.google.com/pubsub/docs/overview (cit. on p. 20).

2023 Generic Decentralized Membership and Communication Abstractions for Edge Systems Diogo Barreto

NOVA SCHOOL OF SCIENCE & TECHNOLOGY