TOMÁS FILIPE PINA GABRIEL

BSc in Computer Science

OVERLORD: CENTRALIZED CONTROL IN HIGHLY DECENTRALIZED SYSTEMS

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2025

DEPARTMENT OF
COMPUTER SCIENCE

OVERLORD: CENTRALIZED CONTROL IN HIGHLY DECENTRALIZED SYSTEMS

TOMÁS FILIPE PINA GABRIEL

BSc in Computer Science

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

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
February, 2025

Abstract

Decentralized systems are gaining popularity once more, not only because of the success of blockchains in the context of cryptocurrency, but the Web3 movement is currently pushing for a more decentralized (and democratic) architecture for the Internet. Increasing the level of abstraction, Swarm Computing, where highly heterogeneous devices can cooperate directly and in a mostly autonomous fashion, demands for highly flexible and robust decentralized solutions, including distributed and decentralized membership, as well as communication protocols.

The peer-to-peer literature is full of different proposals for such protocols. However, the largest majority assumes a static configuration, meaning that the parameters that govern the operation of the system are defined at bootstrap time, based on expectations on the operational conditions of the system throughout its entire lifetime, and remain unchanged even if these conditions change significantly, making the system potentially inefficient or even unavailable. The autonomic computing paradigm has been proposed many years ago as a way to achieve a generic architecture to build systems that monitor themselves and their environment to make decisions on how to reconfigure themselves, and apply those reconfiguration actions. Solutions focused on autonomic computing are usually fully decentralized, or fully centralized, leaving space for the study of hybrid solutions.

In this work we aim to design a centralized (or partially centralized) component that can make management decisions for a large-scale decentralized system, which we named Overlord. The objective is to explore the idea of autonomic computing to govern both membership and communication peer-to-peer protocols. To this end, we propose to devise a design for the solution, implement and evaluate it using different application case studies.

Keywords: Decentralized Systems, Autonomic Computing, Peer-To-Peer, Self-Adaptive Systems

Resumo

Sistemas descentralizados têm vindo a ganhar popularidade de novo, não apenas devido ao sucesso das *blockchains* no contexto das criptomoedas, mas também pelo movimento *Web3* que promove uma arquitetura mais descentralizada (e democrática) para a internet. Elevando o nível de abstração, *Swarm Computing*, na qual dispositivos altamente heterogéneos podem cooperar diretamente de forma maioritariamente autónoma, exige soluções descentralizadas flexíveis e robustas, nomeadamente protocolos distribuidos e descentralizados de gestão de membros e comunicação.

A literatura de *peer-to-peer* propõe diversas abordagens para tais protocolos. No entanto, a grande maioria assume uma configuração estática, o que significa que os parâmetros que gerem a operação do sistema são definidos no momento do seu arranque e baseiam-se em expectativas sobre as condições do sistema ao longo do seu tempo de vida. Isto pode resultar num funcionamento ineficiente ou até na indisponibilidade do sistema. O paradigma da computação autónoma foi proposto há bastante tempo como um padrão de arquitetura para a construção de sistemas que se monitorizam a si próprios e ao seu ambiente para tomarem decisões sobre como se reconfigurarem e as aplicam autonomamente. Soluções baseadas em computação autónoma tendem a ser altamente descentralizadas ou altamente centralizadas, o que abre espaço para o estudo de uma solução híbrida.

O principal foco é a conceção de um componente centralizado (ou parcialmente centralizado) capaz de tomar decisões num sistema descentralizado em larga escala, ao qual nomeámos *Overlord*. O principal objetivo será a exploração do conceito de computação autónoma para a gestão de protocolos *peer-to-peer* de gestão de membros e comunicação. Para este fim, propomos o desenho da solução, a implementação e a avaliação através de casos de estudo.

Palavras-chave: Sistemas Descentralizados, Computação Autónoma, Peer-To-Peer, Sistemas Auto-Adaptativos

Contents

List of Figures **v**

Acronyms **vi**

1 Introduction **1**

2 Related Work **5**

2.1 Decentralized Systems 5

2.1.1 Peer-To-Peer Systems 5

2.1.2 Membership 7

2.1.3 Overlay Networks 7

2.1.4 Gossip Protocols 8

2.1.5 Resource Location 10

2.1.6 Replication Mechanisms 10

2.1.7 Case Studies of Decentralized Applications 12

2.2 Autonomic Computing 16

2.2.1 MAPE-K 17

2.2.2 Monitoring 18

2.2.3 Planing & Analyzing 20

2.2.4 Execution 23

2.2.5 Summary 24

2.3 Frameworks 25

2.3.1 Cactus Framework 25

2.3.2 Appia Framework 26

2.3.3 R-Appia Framework 26

2.3.4 Rainbow Framework 28

2.3.5 Babel Framework 29

3 Work Plan **31**

3.1 Problem Description 31

3.2 Proposed Solution 31
3.3 Evaluation 33
3.4 Planning and Scheduling 34

Bibliography **36**

List of Figures

2.1 General Gossip-Based Communication Between Nodes 9
2.2 Single-Master vs. Multi-Master Replication. 12
2.3 IBM's MAPE-K (Adapted from [23]) 17
2.4 Babel Framework Architecture 29

3.1 Proposed Communication With the *Overlord* 32
3.2 Gantt Chart Of Tasks and Their Expected Duration 34

Acronyms

CAN	Content Addressable Network *(pp. 14, 15)*
DHT	Distributed Hash Table *(pp. 8, 10, 33)*
ECA	Event Condition Action *(pp. 20–24, 33)*
INA	In-Network Aggregation *(pp. 19, 20, 24)*
P2P	Peer-To-Peer *(pp. 5–8, 10, 13–16, 32)*
QoS	Quality of Service *(p. 26)*

1 Introduction

This chapter discusses the motivation and goals of this work. First, we elaborate on the motivations and the problem our work aims to solve. Then we focus on the main objectives and the expected contributions. Finally, we provide a brief explanation of the remainder of the document's structure.

Motivation

In today's age, most services and applications on the internet are supported by cloud infrastructures such as Google, Amazon, Microsoft and Cloudflare, which host a variety of services for developers to host their applications, and also serve users directly. This paradigm is highly centralized and sits on the premise that one trusts and relies on these services and companies, while they dictate the terms of service with little competition. **Web3** emerged to address these issues, focussing on a user-centric Internet with a more democratic and cooperative view of the Internet, ensuring that big companies do not hold all the power. Web3, ultimately aims to promote decentralization, security, and privacy. Web3 is still in its early stages and has not yet become competitive, but has revived the appeal for research in decentralized systems [10]. This resurgence is mainly motivated by the success of the blockchain in the world of cryptocurrencies.

Blockchain is, as its name implies, a chain where each block contains a cryptographic hash of the previous block [38]. Storing the hash in the next block prevents any kind of tampering. Blockchain technology leveraged peer-to-peer networks and consensus models to provide infrastructure for decentralized systems [6]. This was the main catalyst for the resurgence of decentralized systems.

Swarm computing is a paradigm inspired by swarm intelligence, in which members of a system cooperate and coordinate to achieve self-organization, leading to a global behavior denominated as swarm behavior [25]. Swarm computing refers to computer systems that follow this paradigm. At its core, nodes of the system need to communicate in order to achieve cooperation. Therefore, communication and membership protocols are essential to achieve cooperation between nodes. A vast literature of protocols have been studied and developed, but most assume a static configuration, i.e. parameters of these protocols are entered at the start of the system, based on expectations of how the system will perform throughout its lifespan.

Complementarily, autonomic systems were initially proposed long ago with the purpose of providing systems the ability of self optimization without intervention from a user. This is a very important concept that should play a part in the resurgence of decentralized systems, since it allows systems to manage themselves according to environment conditions, eliminating the need for human intervention.

Moreover, solutions of autonomic computing are either fully centralized, in the sense that a central autonomic component manages a single system, or are highly decentralized, meaning that nodes of a system either manage themselves independently or have to coordinate for executing changes. This is not ideal, as these solutions are very complex and can usually cause a lot of network traffic.

Problem Statement

As previously discussed, existing approaches can be categorized into two types: centralized or decentralized. Centralized approaches, in this context, are undesirable since they contradict the idea of decentralized systems directly. Decentralized approaches, involve each component managing itself or communicating with the remaining nodes for deciding and executing changes. These solutions are usually very complex and also very network intensive, since they resort to consensus protocols, which share these issues. Some adaptations, like changing the stack of protocols being used, require consensus, but less complex reconfiguration actions such as changing parameters of protocols, do not usually require strong coordination among nodes.

Some proposed solutions are limited due to providing adaptation of protocol stacks and other complex adaptations, which usually demand coordination of nodes. If we consider only adaptation of protocol parameters, the need for consensus protocols is no longer a concern, freeing the network of intensive load that they produce. Also, already proposed solutions are not scalable. This presents us with a promising research direction, in large-scale decentralized systems, which are managed by a centralized component.

Objectives

With this work, we aim to explore the potential of a system in which a central autonomic component manages a large-scale decentralized system. We aim to develop a solution that implements this model, where the autonomic manager will receive information on the system's overall performance, make a decision on weather any of the protocols require any sort of parameter adjustment and disseminate it to all nodes. Our work will also have to take into consideration the scalability of the system.

Additionally, we will evaluate the solution using application case studies. This consists of running an application using a set of protocols and the developed component, and comparing its performance with another application that employs the same protocols, but remain static throughout the system's lifespan. Both systems will be subjected to the same conditions.

Our solution will be developed using the Babel-Swarm java framework [17], which already features some mechanisms for runtime management of decentralized protocols and systems, while also featuring a library with several such protocols.

Expected Contributions

The main contributions this thesis aims to accomplish are:

1. A new approach of development of self-adaptive large-scale decentralized systems, where a centralized component oversees system adaptation, while its operation remains decentralized.

2. An implementation following this approach with the Babel-Swarm framework.

3. Its evaluation, with case studies, including comparative analysis against equivalent static protocols under identical environmental conditions.

Research Context

This work is being developed in the context of the **TaRDIS** (Trustworthy and Resilient Decentralized Intelligence for Edge Systems) European project, using the Babel-Swarm framework.

"The project TaRDIS focuses on 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" [51].

Document Structure

The remainder of this document will be structured as follows:

Chapter 2 This chapter will present three main areas that are core to this thesis. First we will discuss decentralized systems, which will touch on subjects such as peer-to-peer systems, communication protocols and relevant work developed. Secondly, we will explore the concept of autonomic computing, while discussing the different components such as system monitoring, planning, and execution, which are key to a system capable of self-adaptation. Finally, we will introduce relevant frameworks that are either a useful foundation to build upon, or have themselves made contributions to the world of autonomic computing.

Chapter 3 This chapter will describe the problem to be solved, a proposed solution, application case studies to be conducted during evaluation, and an initial scheduling of tasks that will be part of the elaboration of the remainder of this work.

2

Related Work

In this chapter, we will examine the work conducted in this field and highlight the most relevant contributions. In Section 2.1, we will examine the key concepts of distributed and decentralized systems. Section 2.2 will analyze the concept of autonomic computing, including the many components required for a system to manage itself. Finally, in Section 2.3, we will review relevant frameworks by discussing their contributions and identifying areas of improvement in light of this thesis's promises, as well as explaining the Babel framework and why it was chosen instead of other options.

2.1 Decentralized Systems

In a decentralized system, workloads and responsibilities are distributed among participants of the system. Instead of relying on central points that provide a specific service, all users have to cooperate to serve other users and be served by other users. This approach ensures a balanced workload and also ensures fault tolerance. To understand the concept of decentralized systems, it is essential to understand the principles of peer-to-peer systems and its associated concepts.

2.1.1 Peer-To-Peer Systems

A Peer-To-Peer (P2P) system (can also be called a P2P network) consists of nodes that can directly communicate and interact with each other, making them capable of sharing information and services without the relying on centralized servers. A very important feature that defines these systems is the fact that a node serves as both a resource or service provider, and a requester [47]. This models human interaction–whenever we need something, we ask our friends or relatives, and if they can't help, they might connect us with their peers, which in turn might be able to help us [54]. P2P systems have several characteristics that distinguish them from more traditional approaches to distributed systems, such as:

1. **Symmetric role:** As mentioned before, a node has the role of both server and client;

2. **Scalability:** P2P systems can scale to have thousands of nodes, harnessing the power of many computers instead of just one computer;

3. **Heterogeneity:** P2P systems do not impose any hardware capacity restrictions, i.e. a node can be very hardware limited and another can be very hardware capable.

4. **Distributed Control** In its most pure form, P2P requires load and responsibility distribution among nodes, resulting in a fully decentralized solution, where any random node could be removed, and the system would still be able to perform [47].

5. **Dynamism:** In a P2P system, nodes may leave and join the system at any time, meaning that its topology will constantly be dynamic.

P2P Taxonomy Even though a clear definition of P2P systems was given in the previous paragraph, the actual classification of a P2P system can vary. At the surface level, a P2P system can be decentralized or hybrid, where hybrid P2P systems can be closer to a traditional centralized architecture.

2.1.1.1 Pure Decentralized P2P Systems

A pure P2P system has absolutely no centralized point, i.e. all nodes must have equivalent functionality. This means that there are no centralized nodes or super-nodes (nodes that act as a centralized server to a subset of clients [56]) [42]. Without a central point of failure, upon failure of a node, another one can simply take over. However, this decentralized nature often results in slower location of resources and can introduce challenges such as unpredictability of the system.

2.1.1.2 Hybrid P2P Systems

Hybrid P2P Systems combine strengths of both centralized and pure P2P architectures. These systems feature a central server, but do not remove the feature of peer cooperation. Hybrid systems can be categorized in two main types [42]:

Centralized The system contains a central server that provides some service to others, or has more responsibility than other nodes, and peers maintain a connection to this central server. This approach sacrifices some benefits of pure P2P systems, such as being more vulnerable to attacks and having a single point of failure, but can also provide better coordination among nodes, and optimized routing (discussed further in Section 2.1.5).

Decentralized Instead of centralization in one node, some nodes of the system have a more important role. These are called "super-nodes" and act as a centralized server to peers. Like the centralized approach, they can offer benefits such as optimized routing and coordination, but removing the single point of failure. If a super-node fails, peers can contact another super-node, or, in case they all fail, ordinary peers can assume the role of super-peers.

Overall, the kind of system we are proposing, in its normal functionality will ideally be a pure P2P system, with its autonomic component as a hybrid system. In terms of the hybrid approach, we will either have a central server that coordinates adaptation of all nodes, or have a selection of super-nodes that are coordinated by the *Overlord* component. These approaches are further explored in Chapter 3, under Section 3.2.

2.1.2 Membership

Before exploring the concept of overlay networks, an important concept is membership. This is a core aspect of a P2P network. Most systems assume global membership (i.e. every node knows all nodes in the system), but in decentralized systems, especially large-scale systems, this is not the case. In large systems, the topology is always being changed, with nodes joining and leaving the system, and keeping this information updated for all nodes is not a viable option. For this, protocols rely on *partial views*, which are sets of (contact information for) subsets of nodes (typically logarithmic [30]) in the system [28]. Typically, a node only needs to know a logarithmic amount of nodes for message routing to be done with $O(\log(n))$ complexity, where n is the number of peers in the system [46]

2.1.3 Overlay Networks

An overlay network is a network built on top of an existing network (referred to as the underlay network). The overlay relies on it for basic network communication like routing and forwarding. Nowadays, most overlay networks are built on top of the TCP/IP protocol [52]. The nodes of an overlay network have logical connections that can span many physical connections. This concept is the foundation for P2P systems (and for decentralized systems as a consequence). A P2P system is an overlay network that establishes a (logical) connection between its peers. Overlay networks are usually divided in two categories:

2.1.3.1 Unstructured Overlay Networks

Unstructured overlay networks are also known as random overlay networks, and this is due to their topology. Each node has a selection of random neighbors. It is important to note that, even though not all peers are directly connected, communication between them is still possible through gossip-based protocols (discussed further in Section 2.1.4). One thing to consider is the number of neighbors each peer has. Every peer will have a subset of nodes as their neighbor set, which are usually called a *partial view* (as mentioned in Section 2.1.2). Defining the number of neighbors of each node is a topic that matters most to the protocol being used, and will be discussed in Section 2.1.4. Overall, unstructured overlay networks are very fitting for systems where nodes are joining and leaving at a high rate. However, this comes at a cost since, due to not having any structure, routing can be very challenging [13].

2.1.3.2 Structured Overlay Networks

Structured Overlay Networks are usually based on the concept of consistent hashing (a technique used to deterministically distribute keys across nodes based on their unique identifier), which is fundamental to Distributed Hash Tables (DHTs). DHTs are useful because they allow nodes to find data efficiently, as opposed to having to search all nodes for it, for instance, if a node is looking for a specific resource, they can calculate its hash and know its location (further discussed in Section 2.1.5). Some notable examples of DHT protocols are Chord [49] and Kademlia [34] "a peer-to-peer distributed hash table with provable consistency and performance in a fault-prone environment". Additionally, DHTs can offer this while maintaining little information on the nodes of the system. However, when a node detects a failure by one of its peers, it cannot simply replace it for another random peer, meaning it has to resort to a structured way of finding a suitable replacement [30]. It is also important to note that, a resource can also be a specific peer, instead of data.

2.1.4 Gossip Protocols

After establishing a P2P network, we must then establish a way for peers to communicate.

Gossip protocols are a viable way for nodes of a distributed system to communicate, taking advantage of the cooperation that P2P provides. The general idea of gossip consists of, when a node disseminates a message (to another node or a group of nodes), instead of directly transmitting the message individually to every node, it will send it to its direct neighbors, and their neighbors will send it to their neighbors, until all the intended recipients receive the message. A good way to reason about gossiping protocols, is that messages spread like a contagious disease in epidemic situations [3]. Interestingly, we can also infer the number of neighbors (to which a node has to send a message) necessary for the message to spread from the theory of epidemics [14]. This value is ideally logarithmic in relation to the number of nodes in the system [30]. This parameter is referred to as *fanout*. High values of *fanout* leads to more redundancy, but overall more reliability, as the probability of every node receiving the message gets closer to 1. Lower values of *fanout* reduce redundancy, but also reduce reliability, as not every node may get the message. One variant of this protocol is *flood*, in which a peer sends the message to **all** of its neighbors. Another approach is using a *hop-count*, where a message carries a variable, and each time its transmitted, its *hop-count* increases. When the *hop-count* reaches a certain value, peers stop transmitting it along the network. This reduces redundancy, but this also reduces reliability, as not everyone may get the message.

One common use for gossip protocols is to broadcast a message to every peer in the network. Assume that a node wants to send a message to every peer. The most direct approach, is for that node to send the message directly to each node. However, this is

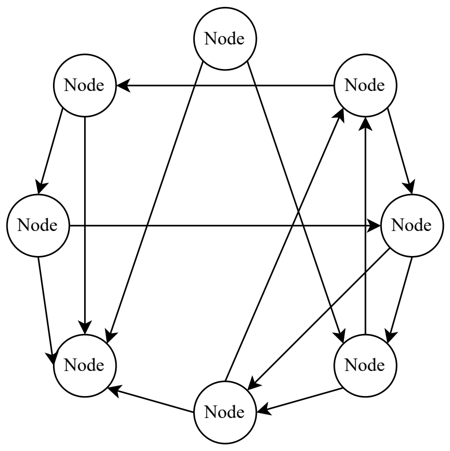


Figure 2.1: General Gossip-Based Communication Between Nodes

unbalanced, as one single node is responsible for dissemination of the message to everyone, while the other nodes just have to receive it. A much more reasonable approach, would be for other nodes to cooperate and help deliver the message to their neighbors as it is passed through the network (as a gossip approach suggests).

Figure 2.1 provides a general illustration of how a gossip protocol works, in which each node, when it receives the message, chooses two random nodes to send the message. Now we discuss the three main approaches to gossip [30]:

Eager Push Gossip Consists of the most basic model. When a peer receives a message, it transmits it to a random selection of its neighbors (can be all of them). This approach is initiated by the sender. This is the variant represented in Figure 2.1

Pull Gossip This approach consists of nodes informing their peers of the data they have periodically (e.g. sending the hash of the data). When a node receives a hash they do not yet have, they can request it directly from peers. This is a good option if messages are big, because messages are sent only if they are needed. However, for each message we have two interactions (*request* + *message*), so if messages are small, we might be putting the network under more stress.

Lazy Pull Gossip In essence, when a peer receives a message, it sends to a random selection of peers its identifier (e.g. a hash). If those peers do not have the message, they can request it. This approach removes necessity for periodically checking in with each peer, but makes it so that each message needs three interactions (*message ID* + *message request* + *message*).

Overall, the choice of which approach to use comes down to a trade-off between redundancy and latency. The eager push approach produces more redundant traffic, and the two other approaches produce more network traffic but less redundancy.

2.1.5 Resource Location

Resource location is a very relevant aspect of P2P systems. It consists of allowing peers to find the owners of a desired resource within the network. If a node wants to find a resource, there are two ways of doing so. It either knows exactly what resource it is looking for (i.e. *exact match* variant of resource location), or it only knows properties of that resource, and in that case, the peer will have to execute a query with the properties of the resource its looking for.

This service became popular with the appearance of Napster [37], and after Napster, a few others have shown up, like Gnutella [5], PAST [12] and KaZaA [19]. Now we look at the different types of architectures implemented by these services [30]:

Centralized Consists of having a centralized server or group of servers containing an index of the resources available and their location. When a peer wishes to locate a resource, it sends the query directly to the central server, which responds with the location of the file. The node then contacts the peer that has the file, so they can share it, maintaining the P2P aspect. Having a centralized server is good as it simplifies the design of the system. However, the system has a single point of failure, which is undesirable.

Distributed Over DHT In this approach, nodes use a DHT in order to locate resources. Each peer registers its resources in the DHT using its unique identifier. When a peer wishes to locate a resource, it searches for that unique identifier in order to know which peer has it. It is important to note that this architecture is only possible if peers know the exact resource they are searching for, so queries based on properties of a resource are not possible.

Distributed Over Unstructured Overlay Network This is an alternative that makes use of unstructured overlay networks. Each peer will maintain an index of the files it has. When a node wishes to locate a resource, it will disseminate a request to the network (e.g. using a gossip broadcast protocol). The peer that has the resource, when it receives this request, will send it to the peer who requested it. In systems where nodes execute a lot of queries this can become a problem, as the network will constantly be filled with a lot of queries being passed around from node to node.

2.1.6 Replication Mechanisms

Data replication, consists of a fault-tolerance mechanism based on the idea of having redundancy of information stored in different machines. In case one of them fails, another can simply be used, and the system keeps working as intended. If a system is reliant on a specific piece of data which is stored in one machine, if it fails, the availability of the system will be compromised. Since that piece of data is only located in one machine, we have a single point of failure, which is undesirable. Replication enables distribution of data along many locations, and if one of them fails, others can simply take over, ensuring availability and fault-tolerance. It might also be useful in reducing communication overhead, and can improve the scalability of the system [33]. This might be of relevance to our work in the sense that it provides a way to increase the availability of data required by a reconfiguration manager to operate correctly. But there are many factors that must be considered when replicating data across multiple systems.

Single-Master vs. Multi-Master Replication This aspect concerns the way replicas interact, and which replicas are effectively executing writes and reads, and which only receive the updates from other replicas.

In **Single-Master** replication, any replica can be contacted, but only one replica can effectively write. When it writes, it propagates the effects of that write to other replicas. This approach has no limitations concerning the order of operations and consistency of the data. On the other hand, it brings problems related to scalability and availability due to only having one replica executing writes. All clients that need to execute a write operation, must contact that one replica, which can become a significant bottleneck if the system has too many write requests at the same time.

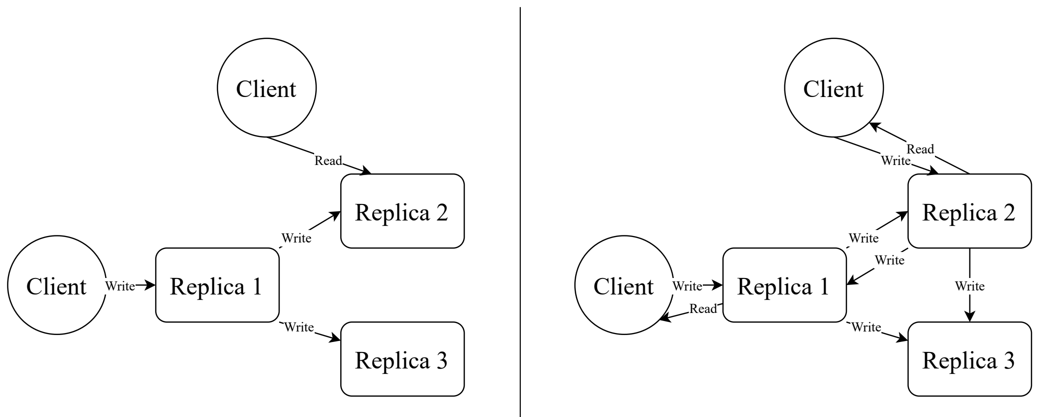
With **Multi-Master**, any replica can write, and is tasked with the responsibility of having to propagate the effects of writes to other replicas. This approach removes the bottleneck by allowing multiple replicas to execute writes providing a better load balance throughout the system. This can create some concerns with respect to the order of operations, so solutions are usually quite complex and expensive in terms of communication.

Figure 2.2 illustrates the difference between these approaches. In (a), only one replica writes, while others can only read. In (b), any replica can either read or write.

Full vs. Partial Replication This factor concerns the location of data among replicas.

With a **Full Replication** approach, all replicas contain a copy of all data in the system. This approach provides good load balancing since all replicas have to store updates, and also provides maximal availability as, in case all replicas fail, the remaining one will still be able to provide all data.

In a **Partial Replication** approach, all systems contain a copy of a subset of the data. This means that the data in one replica can be different from the data in other replicas. This approach limits load balancing, due to the fact that not all replicas will be able to execute operations, so if one object is accessed frequently, load will be unbalanced. It may also cause unavailability if all replicas that have a copy of an object fail.



(a) Single-Master Replication

(b) Multi-Master Replication

Figure 2.2: Single-Master vs. Multi-Master Replication.

Synchronous vs Asynchronous Replication This aspect concerns the point in which an operation is completed, i.e. when the replica (that the request was made to) responds to the client.

In a **Synchronous Replication** system, a replica only responds to the client when all or a certain subset of replicas have stored the update. This means that, a replica will first send the update to all other replicas, and only when it receives enough acknowledgments, will it respond to the client. This provides consistency, since a replica will only respond when it is certain that the update is actually stored in the system. However, it also introduces more latency, due to the client having to wait for the replica to receive acknowledgments from others.

With **Asynchronous Replication**, the replica that received the request, responds to the client when it has stored the update, and only then will it propagate the update to the other replicas. The client will have a much faster response time, but might experience inconsistencies. If a replica sends the response to the client and immediately fails before sending the update to other replicas, the operation will be lost.

In summary, replication and an appropriate choice of replication are a core aspect to a decentralized system. In terms of relation to our work, depending on the approach to the centralized component, it might be useful to replicate telemetry data (subject discussed further in Section 2.2.2) across other locations to provide availability. If the central node fails, the autonomic service would become unavailable. For another node to replace it, monitored data would either have to be collected from scratch, or already be replicated in the node that replaces the failed component.

2.1.7 Case Studies of Decentralized Applications

For evaluation, we will use case study applications that employ protocols and concepts mentioned in the previous sections. These can be good candidates for applications of case studies, because in their essence, they all use these base concepts, which in turn help define decentralized systems (which is our main focus). In this section, we go over how these applications and protocols work and how they employ the concepts discussed in the last section. We will also refer relevant parameters that could be a good fit for adaptation.

2.1.7.1 Gnutella

Gnutella [5] is a file-sharing protocol used for data storage and searching in P2P distributed systems. In Gnutella, all gnodes (nodes of Gnutella) can share and receive files. Its topology does not enforce any type of hierarchy, meaning Gnutella is a real example of a pure P2P system, due to all nodes having the same functionality and responsibility. Gnutella also uses a gossip-based protocol for the communication between nodes, which means that a node that enters the network, only needs one address (known as a contact) in order to be connected to it.

Gnutella has 5 functions, which are:

Ping Function Used by nodes to announce their presence in the network.

Pong Function A sort of acknowledge to the Ping function. This function also carries the neighbors of that node. This is to account for the following scenario: Assume we have 2 nodes *node-1* and *node-2*. If a *node-3* joins connecting to *node-2*, and *node-2* crashes, *node-3* will be disconnected from the network. If *node-2* sends its neighbors to *node-3* and then fails, *node-3* will still be able to participate in the network because it has access to *node-2*'s neighbors, i.e. in this case *node-1*.

Query Function Used by nodes that want to find a certain file. Nodes send a query for the file wanted to their neighbors, and the neighbors are responsible for sending that query to their neighbors until all nodes in the network have received the query.

Query Hit Function Consists of a response to the query. In this message, the node identifier is also included in case the receiving node cannot download the file directly from an IP, and has to resort to a push function.

Push Function This function is used by nodes that want a certain file, and cannot download it directly from the IP. Nodes will send a message to a peer that has that file.

Moreover, some relevant parameters that could be worth exploring in terms of adaptation could be the *simultaneous connections* which represents the number of simultaneous connections each peer tries to maintain and the *time-to-live* of a message (which works as the *hop-count* mentioned in Section 2.1.4).

Overall, Gnutella used a flooding approach, and so this led to a lot of network traffic and redundancy, which eventually killed its usage, but the protocol was studied to improve its efficiency and functionality. However, its approach could be a good candidate for academic evaluation purposes.

2.1.7.2 Freenet

Freenet [8] is an adaptive P2P system that allows for publication, replication and retrieval of data, while also providing anonymity. Files are mentioned without referring its location, and a file can be replicated in many locations.

Each node knows their upstream and downstream neighbors to provide privacy. A request, when its sent, contains a *hop-count* like mentioned in Section 2.1.4 to help prevent infinite chains, and also contains an identifier, so nodes can know if they are seeing a duplicate.

In order to obtain a file, a node must first calculate or obtain the key of the file. It then sends a request with that same key, an identifier and a *hops-to-live* value to itself. When a node receives a request, it will check its own data to look for the file and if its found, it will send the file back down the stream until it reaches the requestor. If it does not find the file, it will search for the nearest key in its table and forward the request to that node. When this node eventually receives the file requested, it will pass it down the stream until it reaches the nodes that requested it in the first place. A possible target of an adaptive solution could be the *hops-to-live* parameter, which could be dynamically adapted depending on the system's topology.

In general, the protocol does provide the decentralized approach, being another example of a pure P2P system due to all nodes having the same responsibility. This makes it a candidate approach for a case study, as it is a system fitting the decentralized approach we aim for. An aspect that might be considered, is the fact that once a file is found, it could be directly transmitted to the requestor, but is instead passed down the stream. Essentially this is to preserve privacy, meaning it comes down to a trade-off between efficiency and privacy.

2.1.7.3 Content Addressable Network - CAN

Content Addressable Network (CAN) [43] is a distributed infrastructure that provides the functionality of hash-tables (where content is mapped by a hash of its key) but through a decentralized system.

CAN was designed to be a more scalable alternative to Napster and Gnutella. Napster, has the index of its files in a single central server, meaning there is a single point of failure. Gnutella, on the other hand, does not centralize the index of its files, but floods the entire network, which makes it not scalable. CAN proposes a solution to these challenges, in order to have a pure scalable P2P system.

The process of file transferring between peers is inherently scalable, so the authors affirm that the issue is locating the file (Section 2.1.5). CAN proposes that nodes are logically organized in a dimensional space. Nodes dynamically organize themselves into a structured overlay network that represents a virtual coordinate space. Each node is responsible for a specific region of this space.

To communicate effectively, each node maintains a record of the IP addresses of its neighboring nodes that oversee adjacent coordinate zones. This group of direct neighbors acts as a routing table, enabling the system to relay messages between any two locations in the network through the virtual coordinate system.

CAN has several parameters that could be a target of adaptation, some of which are *dimensions:* d which is the number of dimensions in the coordinate space, *realities:* r which refers to a replication factor, and parameter p which is the number of peers per zone. Different values of these parameters can provide different effects on the system, mainly affecting path length, latency and routing fault tolerance.

In general, CAN manages to suggest an approach to a scalable pure P2P solution (unlike other previously mentioned solutions). This is a good aspect to be considered since, one of our main focus points is scalability of the system, and other approaches might introduce restrictions concerning this aspect.

2.1.7.4 Pastry

Pastry [46] is another alternative that addresses the resource location problem. It provides routing at the application level and resource location in a P2P system. When a node receives a message, and a key, it re-routes the message to the node that has an identifier closest to the key. Like the other examples, Pastry is a completely decentralized and scalable solution. Assuming we have N nodes, pastry can route to numerically the closest node to a key in $\log N$ steps.

Pastry has some parameters that may be relevant candidates for adaptation, which are: b the number of bits of entries in the routing table, L which is the set of the numerically closest nodes, and M which is the set of the physically closest nodes.

Pastry does provide what it promises, and has been used by many as a building block. One of these was PAST [12], which will be discussed in the next section. Pastry provides efficient routing in a decentralized manner, being a good solution to the resource location problem, and a good fit for a decentralized application.

2.1.7.5 PAST

As mentioned previously, PAST [12] was built on top of Pastry as a storage system providing scalability, high availability, persistence and security. PAST nodes form an overlay network capable of reorganizing itself. PAST also provides high availability by replicating files at multiple nodes. To ensure balance of storage, PAST has a quota system, in which users are assigned fixed quotas, or are allowed to use as much storage as they contribute. To provide privacy, users may use encryption.

The aim of PAST is to combine the scalability and self-organization of systems like Freenet (Section 2.1.7.2) with the reliability and persistence expected from storage systems.

Summary In summary, decentralized systems are a great solution that provides not only decentralization, but also scalability, availability, and fault-tolerance. They can also introduce challenges relatively to coordination, overhead, and in general, preventing different nodes from thinking that the system is in different states is usually complex and challenging. In general, to accomplish our solution, normal system behavior should be the closest to a pure P2P system. The autonomic component of the system will be centralized, meaning it will be closer to a hybrid P2P system. To provide scalability, messaging the centralized component should be done using a gossip-based protocol to relive the component from having to manage all nodes directly.

2.2 Autonomic Computing

Due to rapid advancements made on the field of computing, applications have not only become increasingly complex, but also heterogeneous and dynamic. Distributed systems did not help on this matter, since even the most simple distributed system can succumb to the problems of network communication. In 2001, IBM suggested the concept of autonomic computing [22]. It emerges as a different paradigm for systems and application design, which seeks to strengthen computer systems while also reducing human involvement. IBM, along with this new paradigm, also formulated the four properties of an autonomic system, usually referred to as the **self-* properties** [23]:

Self-Configuration Consists of the capability for a system to reconfigure itself according to high-level goals, i.e. through specifying an objective and not necessarily how to achieve it.

Self-Optimization A system that is capable of optimizing the use of its resources under changes within itself or its environment.

Self-Healing A system that autonomously detects and determines issues within itself. Issues can go from hardware related problems to software errors [41]. It is necessary that, through the self-healing process, the system is not further harmed by, for example, new bugs introduced. Fault-tolerance is a crucial aspect.

Self-Protecting The ability of a system to protect itself from attacks. This means that the system should be able to provide security, privacy and integrity of data.

As mentioned before, a self-adaptive system should be able to provide these four properties. However, the main properties this thesis will have its focus on are self-configuration and self-optimization (with less emphasis on the two other mentioned).

The autonomic computing paradigm is inspired by the human nervous system, the most sophisticated example of autonomic behavior existing in nature today. It possesses the ability of detecting issues within itself, and, as an unconscious reflex, respond with

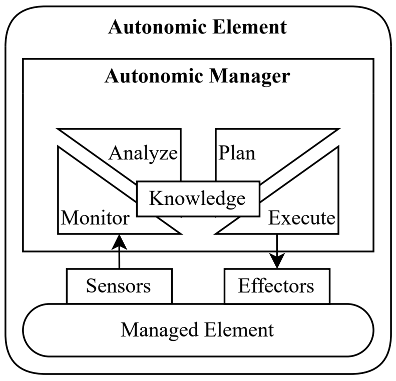


Figure 2.3: IBM's MAPE-K (Adapted from [23])

corrective actions. For instance, when a human is exposed to low temperatures, blood vessels constrict to minimize heat loss, or when the eye is exposed to light, its pupil contracts.

The goal of the autonomic paradigm is to enable a system to analyze itself and, based on high level goals, respond to changes ensuring its continuous and correct operation [40]. In essence, the objective is to create a system that works similarly to a human nervous system, quickly reacting to anomalies within itself or adverse external conditions, and carrying out measures to counter them, bringing back the system to its normal and correct state.

Adaptation Throughout this document, the term adaptation will be used quite frequently, so it is important to understand its meaning in this context. Adaptation refers to the action of changing a certain behavior of a system based on when a condition is met [45]. For example, suppose we have a system that needs to broadcast information from one node to all others. If communication is made through a gossip based approach, one might simply adapt to have a bigger or smaller *fanout* depending on the number of nodes in the system. Or even change the wait time before a *timeout* is triggered.

2.2.1 MAPE-K

MAPE-K, originally proposed by IBM, is a control loop (a loop that collects information on the system, acts upon it and feeds it back into the loop [48]) and a well-recognized engineering approach to achieve self-adaptive systems [2]. It consists of four computational steps: *Monitor*, *Analyze*, *Plan*, *Execute* over a *Knowledge* base, as illustrated in Figure 2.3.

An autonomic element can be any type of computing resource such as an application server, storage device, database, among others [26]. The autonomic element consists of a *managed element*, which represents a software or hardware resource that achieves autonomic behavior through integration with an *autonomic manager*. *Sensors* (also called *probes*) are responsible for collecting information about the performance of the *managed element* and its environment, that is considered relevant for making a decision. *Effectors* implement modifications on the *managed element*. The *autonomic manager*, with the data collected by the *sensors*, can *monitor* the *managed element* and *execute* reconfiguration actions through the *effectors* [23].

A component *K* (*Knowledge*) keeps data about system (including its environment) and data concerning its adaptation goals. This data can come from human experts and/or gathered data from the *sensors*.

The component *M* (*Monitor*) gathers particular data from the execution environment, the system itself, and other needed information, storing it in the *K* component. The component *A* (*Analyze*) is responsible for interpreting the collected data and detecting relevant changes from the expected behavior of the system. If a change is detected, this component is also responsible for triggering the *P* (*Plan*) component, which uses the provided information to orchestrate a workflow of reconfiguration (i.e. adaptation) actions necessary to achieve the system's goals. Finally, the *E* (*Execute*) component carries out these actions in order to bring the system back to a normal behavior.

This separation into multiple components helps the designer focus on one activity at a time, and implement and modify each one of them as a separate module of a bigger system.

As stated in [40], 'Sensing'; 'Analyzing'; 'Planning'; 'Knowledge' and 'Execution' are keywords to identify an autonomic system, and although the MAPE-K model doesn't define the entire field of autonomic computing, the objectives of each of the components are the main areas of concern. This being said, each component has a subsection, in which the advancements made on the thematic will be detailed independently of MAPE-K.

2.2.2 Monitoring

As stated in Section 2.2.1, monitoring consists of gathering information about a system and its environment. The collected data should be useful in order to help the other components make a decision.

For instance, assume we have a system composed of nodes that send messages to each other. Relevant information can include latency, packet loss, throughput, CPU usage, among others. Essentially, information that can then be used to have a sense of the current status and behavior of the system.

According to [23], there are two types of monitoring in self-adaptive systems:

Passive Monitoring Consists of observation and collection of data without actually interfering with the operation of the system. For instance, operating systems provide tools for observing the CPU and memory usage, runtime information of the system, devices mounted, per-process information, among others.

Active Monitoring Involves engineering the software, adding code to its implementation specifically tailored to collect the specific required data.

2.2.2.1 In-Network Aggregation

If we think of a distributed system with a high number of communicating nodes that need to share information about their performance, network traffic can become a limiting factor quite fast. As stated in [39], there is a need for new techniques that reduce the monitoring overhead. Other applications with similar problems prompted research in solutions for In-Network Aggregation (INA). Essentially, it consists of aggregating data in advance along the network path, i.e. combining or summarizing data as it travels through the communicating nodes [16]. This is achieved through a distributed aggregation protocol [9], but since, in this context, monitored information changes over time, using a continuous aggregation protocol is the best approach. There are two approaches for in-network aggregation [15]:

In-Network Aggregation with size reduction: Consists of combining and compressing information from distinct sources before sending it to the next node. For example, in a distributed system, a node receives the latency of other nodes. In order to reduce traffic over the network, one could pass along the average of the latencies instead of their isolated values.

In-Network Aggregation without size reduction: Consists of forwarding the data along the network without any kind of reduction. In a distributed system, when a node receives the latency from other nodes, it would simply combine them in one message and forward it to the next node.

Choosing which approach to take will depend on the needs of the application, but above all, will depend on the need for accuracy of the information. For the purposes of this document, the main objective is of monitoring the overall performance of the system, meaning that the first approach might be the most adequate one, although the version without size reduction might be useful (e.g. detecting faulty nodes and outliers, which can only be done with lossless functions).

Concerning aggregation functions, these are responsible for aggregating the data, meaning they take (collections or several entries of) data as an input and output the aggregated data. There are two main aspects that concern aggregation functions [15]:

Lossy and Lossless Functions that compress data in either a lossy or lossless manner. With lossy functions, the original values are not recoverable and a certain degree of precision is lost. With lossless functions, the original values can fully be reconstructed after the function has been applied.

Duplicate Sensitive and Insensitive Essentially, if the number of times the same information is considered affects the output, we have a duplicate sensitive function. If the output is the same no matter how many times the same information is considered, then it is a duplicate insensitive function. For example, an average function is duplicate sensitive, while a max value function is duplicate insensitive.

Another aspect to consider in this context, is that these functions should not be resource intensive. If too much time and computing resources is spent on executing these functions, we lose efficiency, which is the main issue being tackled.

Overall, monitoring is very dependent on the application, and for a general solution, it is important for mechanisms to be in place that can efficiently convey the current status of the system's performance, and INA solutions might be important in the efficiency part. Since nodes will be sending their information to other nodes (as in gossip-based communication, Section 2.1.4), a node will receive information from distinct nodes. An issue that may arise is the consideration of the same node more than once. Assume four nodes: *node-1*, *node-2*, *node-3* and *node-4*. If both *node-2* and *node-3* individually send information to *node-4* that considers *node-1*, *node-4* might aggregate their information, considering the information of *node-1* twice. To tackle this, we will make use of duplicate insensitive aggregation functions (although probably not as trivial as a *max value* function). This will allow for information of the same node to produce the same result despite how many times it is considered.

2.2.3 Planing & Analyzing

Once a system has sufficient information about its state, it must then evaluate whether it is operating as intended or if it requires adaptation. If reconfiguration actions are indeed required, the system should determine the appropriate corrective measures necessary for it to return to the normal expected behavior. 'Analyzing' was also joined with this section due to it being very application dependent and also, usually being co-joined with the planning phase. Although there are many models (e.g. graph-based, process-based and state-based), according to [45], policy-based solutions are considered to be the best approach. This is due to their declarative nature, which results in a better understanding by an operator, of the system and when it decides to adapt. Policy models can range from more simple solutions (like if-else behavior) to more complex solutions (e.g. policies that achieve a high-level goal).

2.2.3.1 Policy-Based Planning

Event Condition Action Rules Event Condition Action (ECA) rules are located on the simpler side of policy-based models. ECA rules consist of a set of rules that specify what actions are taken by the system when certain events occur. When an **event** takes place, the **condition** is evaluated, and if it is met, an associated **action** is executed [35]. The designer of the system can specify what events should trigger a change, and what changes are triggered by each event.

For example, consider a distributed system composed of nodes that communicate with each other using an epidemic broadcast strategy. If the delivery rate (i.e. rate at which messages are successfully delivered) decreases, it might be beneficial to increase the *fanout* parameter. For this purpose, we might have an ECA rule specifying that, when the message delivery rate reaches a number less than x, increase *fanout* to y.

Although the idea of ECA rules was initially conceived in the context of databases, their concept fits perfectly in the context of self-adaptive systems and has widely been used in research done in its field. This is due to them providing a direct way of mapping specific events (e.g. when delivery-rate reaches a defined critical level) to actions (e.g. increase *fanout*). So assuming a system is capable of collecting information about itself and of making changes to itself, implementing ECA rules would complete the system, enabling it to manage itself. However, a challenge associated to ECA rules, can be the extensive knowledge about the system that administrators must have. Since the whole concept is for events to be directly connected with actions, this means that, to conceive a policy, one must have a deep understanding of the system including potentially aspects related with their low-level nature. Another possible issue, might be the scalability of these rules. If a system grows to have more and more rules, it may become impossible to foresee what could occur in the event of multiple policies in place. These conflicts need to be resolved, and more research on this matter has emerged. Primarily, [32] proposes precedence relationships between policies, enabling the coexistence of inconsistent policies. Also, [24] researched conflicting authorization policies.

Goal Policies Goal policies come as a more high-level approach to ECA rules, in the sense that, instead of defining a specific action, one would define a desired state, and let the system find a way to achieve that state [23]. For instance, assume a distributed system composed of communicating nodes. We could specify that the communication latency between nodes in the system should remain under 50 ms. The autonomic manager (remember Section 2.2.1) uses internal rules (i.e., knowledge) to adjust resources or configurations as needed to reach the desired state. An approach to goal-oriented policies was presented in [4], where the general objective is to refine high-level goals into low-level operations that can subsequently be mapped onto ECA-based policies. In approaches like [50], goals are expressed in a temporal logic (a formalism used to describe how propositions or conditions evolve over time), and plans are generated to achieve a particular goal in a given environment.

Goal policies require planning from the autonomic manager, making them more resource-heavy than ECA rules defined in the previous section. These policies also suffer from the fact that each state has to be described as desirable or not desirable, and if a desirable state cannot be reached, the system will not know which will be less desirable than the other.

Utility Functions Utility functions are a way to overcome the main problem of goal policies. Instead of states being classified as desirable or undesirable, states will have a scalar value of desirability. A utility function is a function that takes an expected effect of a configuration as input, and a value as output, which represents its level of desirability. Utility functions enable on-the-fly determination of an (expected) optimal feasible state, whereas goal policies guide the system toward any state that meets both feasible and acceptable criteria [55]. Usually, utility functions are used to compare alternative configurations, where one applies it to the different expected effects of a given configuration, and based on the output, choose the best one.

The greater problem of utility functions is its definition. Every aspect that influences the decision must be quantified, although research concerning this question is being done. A contribution in the field of utility functions was presented in [55] where utility functions are employed to dynamically allocate resources within a realistic autonomic computing system.

Policy Languages Policy languages are a form of specifying policies in a system, i.e. specify policies in systems employing policy-based planning. Policy configurations are often described in XML format to ensure accuracy and compatibility across different platforms. Current methodologies provide only definition and configuration of policies prior to initialization of the system, meaning the policies defined remain static throughout the system's runtime. This is not ideal, as policies should be capable of self-adaptation to better align with the goals of the system.

For instance, consider a policy that adjusts the monitoring rate based on the level of dynamism in the monitored object's behavior. If the rate is too high, the associated processing and communication costs become excessive. However, if the rate is too low, the system might fail to respond to a sudden change [1].

The number of contributions to this field is extensive, but some notable ones include Ponder [11] and PDL [31], which provide an ECA rule-based implementation, where Ponder was devised both for distributed system's and security management. Another contribution, specifically on the field of policy conflict detection and resolution (specifically ECA-based policies) is [7]. Finally, a language specification that supports dynamic self-adaptation of policies [1].

2.2.3.2 Architectural Model Based Planning

This approach consists of creating a model of the system (the *architectural model*). This model reflects the system's behavior, outlines its requirements, and its objective. The model may also represent aspects of the system's environment. This model is then used to help reason about the impact of certain adaptations. The great benefit of this approach is that, assuming the model correctly represents the system's behavior, it is possible to observe how it will react to adaptations, and if it continues to operate smoothly. This is achieved by applying adaptations to the model first, observing the resulting state from the adaptations, and if the results are positive, apply them to the system.

The use of this approach is compatible with the use of policy-based planning. The adaptations on the model can be specified with ECA rules. For instance, if the model starts operating outside the expected behavior, the correcting measures that return the model to a stable state might be specified by means of ECA rules.

A negative aspect of this approach is the fact that, there will always be a delay between an event happening in the system, and the model being updated to reflect that event. If the delay is significant, and the system changes its state frequently, a specific planning might be carried out when the model assumes the system is under a state that in fact has already changed to another [23]. For example, a distributed system where nodes communicate with each other. Assume that for a moment, latency increases drastically, but then returns to its expected value. Even though latency has already returned to its normal value, the system might already be deciding to increase the *timeout* value as a counter-measure to an outdated state of the system.

2.2.4 Execution

After planning completes, the system needs a way to carry out the selected adaptation plan successfully. Depending on the type of adaptation decided, different approaches by the same system may be taken, and the system might be disrupted in different levels.

When we have nodes communicating with each other, one cannot simply decide to adapt on its own. For example, a node that decides to adapt while it is in the middle of communication with other nodes. Messages sent to a reconfiguring node will either be completely lost, or have to wait for the node to finish reconfiguration, leading other nodes to *time out* and think that it failed. Moreover, if nodes have to switch protocols, and their messages are incompatible, those messages might be lost.

An approach to ensure the system operates as expected, is to make a node *quiescent* before adapting. A node being in a *quiescent* state, implies that it is not currently engaged in communication, either initiated by itself or any other node, i.e. before becoming *quiescent*, a node has to complete any communication it is currently engaged in [27]. Before a node becomes *quiescent*, it will usually become a *passive* node, i.e. it will not initiate any communication, but will still answer to communication from other nodes. Another aspect of this approach, is that other nodes around the target, will have to become *passive*, or they might initiate communication with it. To achieve this kind of coordination, some form of a configuration manager is needed to identify these surrounding nodes and change them to a *passive* state.

The work described in [20] suggests two algorithms. The first algorithm suggests letting ongoing communications finish, and then block all nodes of the system before adapting. This approach is disruptive as it stops the whole system, even though some nodes might not even be affected by the adaptation taking place. The second algorithm suggests only blocking the target nodes, and any others that are in current communications with a target. All requests made to any of those nodes will have to be postponed until after adaptation has taken place, but other nodes can continue operating unaffected.

Offering a less disruptive alternative, the work in [53] proposes a solution that only requires stopping the target node, and executing adaptation only when all other nodes assure they will not initiate any interaction with it.

2.2.4.1 Switching Protocols

Distributed applications run several protocols. Depending on the conditions of the system and its environment, one might choose different protocols for different situations, which may require switching between them. Some protocols also need to maintain a state.

Switching protocols is something that can become complex very rapidly, specially if the state of one protocol also has to be transferred to the new protocol. The work in [45] describes an example using two distinct **total order broadcast** algorithms. Total order broadcast, a frequently used as a case study, ensures that all nodes receive messages in the same order. Assume two general protocols $TO1$ and $TO2$. Switching between two of these is not trivial, especially due to the system's distributed nature. If messages between the two are incompatible, and a node broadcasts a $TO2$ message, and another node is still executing $TO1$, that message might be lost. Moreover, if a node is executing $TO1$ and switches to $TO2$ without properly transferring its sate, messages can also be fully lost. To solve these issues, the general approach is to place nodes in a *quiescent* state, and use consensus protocols to make them agree on when to change, and when communication can be resumed.

However, consensus protocols are quite complex and usually consume a lot of network traffic. To avoid the restrictive behavior that strong coordination among nodes is associated with, our work will focus more on the adaptation of protocol parameters, instead of changing the protocols being used altogether.

2.2.5 Summary

Overall, the contents discussed in this section will prove important in the elaboration phase of our work. Since we are in the context of distributed systems, INA will provide less network traffic and more efficient communication. Policy-based planning will be the main interest since they provide a clear relation between expected events and any actions of adaptation taken by the system. Specifically, we think ECA rules might be the best fitting option. Since we are mostly concerned with specific parameters of protocols, low-level knowledge of the protocol is needed. Finally, since this work will focus more on parameter adaptation of distributed protocols, complex solutions of executing adaptation might not be (strictly) necessary. Instead, solutions that cause less disruption in the system will be preferred (discussed further in Section 3.2).

2.3 Frameworks

In this section, we will examine relevant frameworks, for supporting distributed systems, these being Cactus which enables a survivable and self-adaptive system, Appia which enables customizable communication through the use of protocol stacks, R-Appia which adds support for adaptability to Appia, Rainbow which provides monitoring and adaptation for a system, and lastly Babel, a framework for development of distributed protocols. These are either a good base for implementing self-adaptation, or have themselves made contributions to research of self-adaptive systems. In this chapter, we will be focusing on their primary objectives, their advantages and disadvantages, and, where applicable, how their autonomic capabilities compare to the work we are developing.

2.3.1 Cactus Framework

The main objective of Cactus [21] is to provide a system capable of survivability, i.e. tolerable to attacks and failures. The authors propose fine-grained customization which allows detailing many aspects of the system, enabling choice between trade-offs of different guarantees. They also propose dynamic adaptation allowing services to change their behavior at runtime as a reaction to intrusions or failures.

Cactus addresses functional properties and quality of service attributes (e.g. reliability, performance, security). In order to provide a high degree of customizability, different properties and functions constitute different modules which interact trough events. A service is implemented as a composite protocol and each variant is implemented as a micro-protocol. A micro-protocol is in essence, a collection of event handlers, executed when a given event that they are bound to, occurs. An example of an event can be a message arrival.

When it comes to adaptation, Cactus, due to its event-driven nature, allows for changing out old micro-protocols for new ones by simply binding and unbinding respective events. Cactus allows different approaches of adaptation, such as modifying parameter values of already in place protocols, and changing protocols in place for others. This second approach has two variants which are event-handler rebinding (discussed above), and dynamic code loading. Dynamic code loading implies that new micro-protocols are loaded into a running composite protocol.

However, this approach of dynamic code loading demands coordination among peers, that is achieved through consensus protocols, which in turn are known to be very complex and consume a lot of network traffic. Moreover, adaptation is triggered by means of events, and since there is no centralized component, if too many adaptations occur at the same time starting in different nodes, some inconsistencies might occur. Another issue is related to the fact that the adaptation logic is joined with the adaptation execution, which may raise challenges in changing the adaptive behavior. Finally, Cactus does not offer concurrency management.

2.3.2 Appia Framework

Appia [36] is a framework that allows for building and composing network protocols, providing communication channels that use said protocol composition. Protocol compositions are referred to as protocol stacks, and can be reused to create different communication channels that comprise the same protocols. Appia is referred to as a protocol kernel, which is a framework that supports composition of micro-protocols. These micro-protocols are then organized in a stack. Appia also supports coordination between channels and a way to specify inter-channel constraints (e.g. channels that provide information about node failures).

An Appia module is called a *layer* (a micro-protocol) that provides a service. Composing these layers on top of one another gives us a protocol stack that offers a given Quality of Service (QoS). Once a stack is defined, multiple channels can be created implementing that same stack, which will have its associated QoS. A channel with a QoS has also a stack of *sessions*, which maintain state variables for each protocol. *Sessions* interact through events.

While Appia lacks built-in adaptive capabilities, it provides a solid foundation for their implementation. Its protocol composition enables fine-grained configuration, allowing a hypothetical adaptive component to dynamically switch between protocol stacks when system performance declines. This was achieved with R-Appia, which will be discussed in detail in the following section (Section 2.3.3).

2.3.3 R-Appia Framework

R-Appia [44] is built on top of Appia, presented previously, and provides dynamic reconfiguration of protocols.

Appia has some issues that prevent its support for adaptation, such as:

- Does not include a mechanism for monitoring the system.
- Only supports static protocol composition, meaning that protocols cannot be modified at runtime without stopping the system.
- Does not have a mechanism that enables coordination among multiple nodes for reconfiguration.
- Does not have a mechanism that forces channels to become *quiescent*
- There is no mechanism that allows carrying over a state from one protocol stack to another.

R-Appia, comprises add-ons (additions to the Appia kernel without modifying it) that mitigated these issues, but also comprises changes to the Appia kernel.

First, the authors address **context sensors** (that enable monitoring of system performance). To enable system monitoring, The authors created new event types, as well as a

GenericSensor that can be added to any protocol stack. R-Appia also provides a *Channel Reconfigurator*, a technique that forces channels to become *quiescent* by circumventing an optimization of event flow that restricted this behavior. R-Appia also introduces a **buffering layer**, which hides the reconfiguration from the application. Since the application does not know that a reconfiguration is taking place, it will keep sending messages, and the **buffering layer** is responsible for storing those messages until communication is restored. Also, the protocols developed for Appia also need some adjustments.

R-Appia also presents a reconfiguration strategy to account for different types of reconfiguration. Some adaptations can be carried out without any coordination of nodes and some require coordination between them. The elements of each combination include **orchestration**, which determines whether nodes require coordination, and **local reconfiguration mode**, which specifies whether the node must first become *quiescent* or if the reconfiguration can be applied directly.

Zero Quiescence Reconfigurations that involve changing configuration parameters of protocols. This kind of reconfiguration does not require any coordination between nodes, i.e. orchestration is uncoordinated. Locally, the adaptation can be applied directly without need for the channel to become *quiescent*.

Local Quiescence When the reconfiguration action does not involve changing the structure of messages that pass through the protocol stack. Nodes do not require coordination, but the protocol composition requires *quiescence*. It also means that part of the state from the previous stack needs to be transferred to the new one.

Global Quiescence Nodes will need strong coordination between them. Protocol compositions will also need to be put in a *quiescent* state just as in local *quiescence*. This is highly intrusive, although necessary due to certain adaptations needing this intrusiveness.

The mentioned features constitute add-ons to Appia, meaning that they do not involve modifications to the Appia kernel. These next modifications enact changes to the Appia kernel:

1. **Pool of Headers:** A message in Appia contains a stack of headers, which every protocol pushes when the message passes through the respective protocol. This means that, when another node receives the message, each protocol will have to pop its corresponding header, which implies strong coordination among nodes. R-Appia, instead of a stack, proposes a pool of headers, where each protocol can check if its header exists and remove it from the message. This does not remove stacking of protocols, just headers in messages.

2. **Dynamic Update of Event Routes:** Appia only provides support for static composition of protocols due to an optimization regarding the pre-computing of routes for all events in a communication channel. The solution discussed in the last paragraph

suggested putting the channel in a *quiescent* state, but since we now are changing the kernel, a new solution can be implemented, which consists of storing the pre-calculated routes, and updating them relatively to the new protocol stack.

R-Appia is a relevant approach to tackle decentralized adaptation approach, providing dynamic reconfiguration to the Appia kernel, either with add-ons or by modifying the kernel. Its main limitations, come from Appia, mainly the stacking of protocols, that is not ideal as it restricts communication, and also the fact that its execution is single-threaded. Moreover, it is not a scalable solution, as all nodes contact directly with the autonomic manager. This means that, if we have a high number of nodes in the system, all of them will be communicating their status with the monitor in the autonomic component, leading to scalability issues, possibly dragging the whole system to a halt.

2.3.4 Rainbow Framework

The Rainbow Framework [18], is a proposal on the architectural model based planning (referred in Section 2.2.3.2). Rainbow makes use of architectural models to dynamically monitor and adapt a system, while also focussing on reutilization of adaptation strategies and infrastructure across different systems.

To achieve the goal of reusability, Rainbow splits the framework in two aspects. The *adaptation infrastructure*, which concerns the reusable aspect, and the *system specific adaption knowledge* which, as its name indicates, concerns more system specific aspects. Rainbow is also divided into three infrastructures:

System Layer Infrastructure An infrastructure that represents communication with the actual system, containing probes and effectors to enact changes in the running system.

Architecture Layer Infrastructure Responsible for the adaptation decision and execution. This layer contains *gauges* that aggregate information received by the *probes*, a *model manager* that provides access to the architectural model, a *constraint evaluator* that is responsible for detecting issues and triggering adaptations if needed. Finally, the *adaptation engine* that determines how to correct the system, and then carries out the corrections.

Translation infrastructures Essentially a translator between the system layer and the architecture layer, that mediates mapping information across the abstraction gap between these two.

Rainbow monitors the system (that runs in the system layer) through the *model-manager*, while the *constraint evaluator* evaluates the model for constraint violations, triggering the *adaptation engine* if needed. The *adaptation engine*, is then responsible for effecting changes on the running system.

The adaptation knowledge is application specific. It consists of behavioral constraints, strategies and properties of components. Adaptation knowledge can be reused through

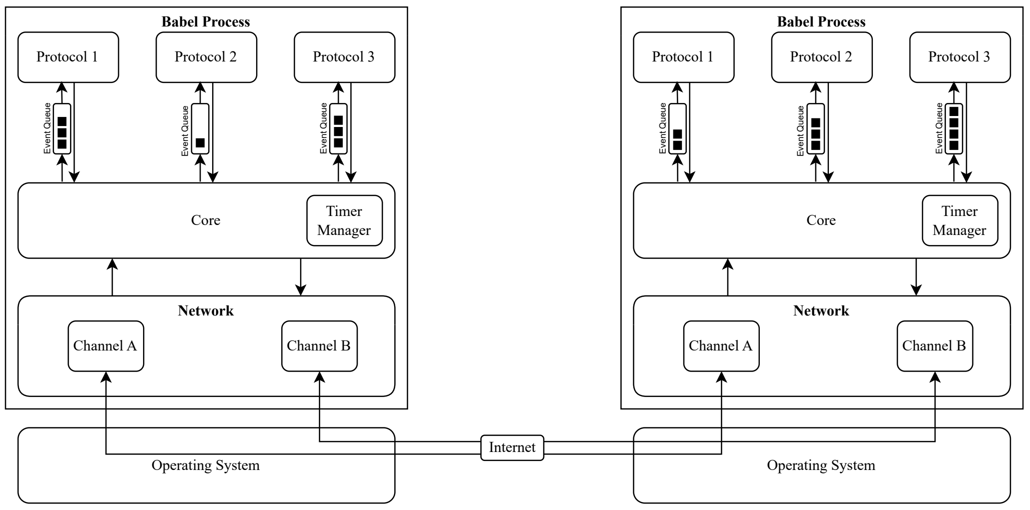


Figure 2.4: Babel Framework Architecture

the notion of *architectural styles*, which can be used by systems that share semantic and structural properties between themselves.

Rainbow separates the autonomic component fully from the system, thus being external to the system. Since it is based on architectural model planning, it might share their benefits, but also their negatives such as a delay between the model and the system, causing planning based on an outdated state of the system. It should be noted that the main focus of Rainbow is not on distributed systems.

2.3.5 Babel Framework

Babel [17] is a framework that proposes hiding the tedious and time-consuming tasks of dealing with low-level aspects of designing distributed systems and protocols such as communication channels, timeouts, or tasks associated with dealing with concurrency issues. Babel makes an abstraction of the low-level network aspects and timer aspects, leaving the programmer to design the system/protocol without needing to spend much time on these concerns. Babel provides a programming environment for protocols and distributed systems, promoting event driven programming.

Figure 2.4 illustrates the architecture of Babel. The framework has three main components, which are:

Protocols Protocols are programmed by developers using Babel. A protocol is modeled as a state machine that evolves through receiving events which are queued in the *event queue*. Events can be timer triggered, notifications, or events created specifically for the protocol being developed. Execution is also multithreaded as protocols have specific threads for handling events and a thread for themselves (i.e. protocols can execute concurrently, although without having any shared state).

Core The Babel core is the principal component responsible for mediating interaction between protocols (inter and intra process). When a process wishes to communicate with another one, it is the core that delivers events to connect them. When a process wants to communicate with another process, the core delivers the message to the appropriate network channel, which in turn sends the message to the appropriate network address. The core also handles receiving the message at the other process' end. Apart from this, the core also keeps track of timers and delivering events when the corresponding timer triggers.

Network The network is abstracted by Babel using channels. Instead of dealing with the challenges of handling low level aspects of network communication and error handling, Babel provides simple primitives for protocols to use for interacting with one another: `openConnection`, `sendMessage` and `closeConnection`. Channels generate specific events related with channels such as when an *outgoing connection* is established or disconnected. Babel also provides a way for developers to create their own channels to help deal with network specific enforcement.

Like Appia, Babel does not currently have any mechanisms to support adaptation. Moreover, Babel is a great foundation for building distributed systems, providing efficiency and removing the error-prone and time-consuming tasks of network communication.

2.3.5.1 Summary

After analyzing all frameworks presented in this section, Babel was chosen as the base framework, on which *Overlord* will be built on top of. Apart from being developed "in-house", it provides some relevant benefits that other frameworks lack. For instance, Appia executes in a single thread across all protocols, and requires stacking them, leading to restrictions in interactions between them, whereas Babel provides a thread for each protocol and does not require stacking. Cactus doesn't offer concurrency management, while Babel shields the developer from issues related with concurrency by avoiding (mutable) shared state among protocols.

3 Work Plan

3.1 Problem Description

As stated previously in Chapter 1, this work aims to devise a centralized autonomic component that manages a pure decentralized system. Current autonomic solutions are either fully centralized, not considering distributed systems at all, or fully decentralized, where nodes manage themselves, often requiring coordination. This coordination, although for some adaptive actions is needed, is not ideal as it puts a system under stress with a lot of network traffic for coordination. If nodes also need to share telemetry information with one another, this can also put a heavy load on the network. Moreover, scalability is a key factor, as a decentralized system must consider a very dynamic topology, with numerous nodes constantly leaving and joining its network.

This means that there is scope for the study of an approach that is scalable and consumes less network traffic.

3.2 Proposed Solution

The proposed solution, is to develop a centralized component that manages a decentralized large scale system, based on its telemetry information collaboratively gathered by other processes. This component will receive information on the current performance of the system, and will be tasked with evaluating the presence of any issues, choosing a course of action and disseminating it to the rest of the nodes. It is important to note that, our solution will focus more on adjustment of protocol parameters. Another caveat of this solution is that it has to be scalable, i.e. performance should be unaffected independently of the total number of nodes of the system.

To this end, we aim to develop our solution on top of the Babel-Swarm framework, which has some features of interest, mainly a vast library of already implemented protocols and some mechanisms for adapting a protocol at runtime. We have named our solution *Overlord*.

Some design aspects that are to be considered include:

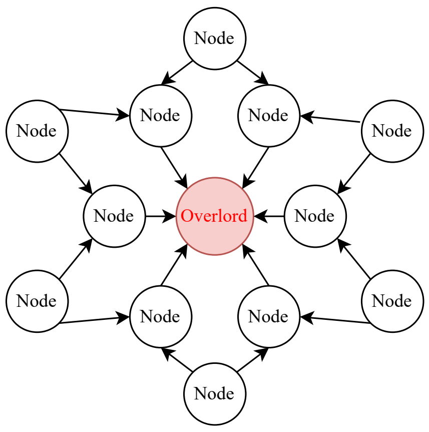


Figure 3.1: Proposed Communication With the *Overlord*

Centralized or Partially Centralized Component The core premise of our work is that the autonomic manager is centralized. This centralization can consist of just one node that is fully responsible for autonomic management of the whole system. However, another suggested alternative that is to be considered, is a hierarchy of nodes. A system could have one central node (denominated *Overlord*) which has in its *partial view*, a number of nodes which are responsible for disseminating the *Overlord*'s decisions and aggregating the system's performance data before sending it back to the *Overlord*. Depending on its implementation, this second approach could improve scalability of the system, but also raises some questions, mainly, how to elect these intermediary nodes, how many should there be relatively to the number of nodes of the system, and how to recover if one of them fails.

Communication with the *Overlord* As stated previously, the problem with existing hybrid solutions, is that they are not scalable due to all nodes of the system having to communicate with the centralized *Overlord* node for it to assess current performance of the system. Considering a decentralized large scale system, this is highly restrictive because one node will be constantly receiving updates by a large number of nodes. An approach to solve this, is to use a gossip-based protocol. Instead of all nodes communicating with the central node, nodes will send their performance to their neighbors, which will in turn aggregate their performance with the ones they receive, until it reaches the *Overlord*. This distributes the load much more efficiently and fairly, and consequentially, the central node will also receive already aggregated information, ready for processing. Increasing the number of nodes will not affect the central node, because those nodes will leverage the benefits of P2P systems, cooperating to aggregate all information for the central node. Figure 3.1 provides a visual representation of how communication with the *Overlord* should be conducted.

Dissemination of Decisions Once the *Overlord* makes a decision of adapting the system, it has to disseminate it. One option is for the *Overlord* to directly contact each node in the managed system to instruct it of the changes that are to be carried out. However, this is not really balanced, as it puts the complete load on the *Overlord*. A more reasonable way of disseminating decisions is to also use gossip-based communication for dissemination of decided changes, improving system load balance.

Planning Since our focus will be on parameter adjustment, it is imperative that an operator has low-level knowledge of the protocol they are developing. With that in mind, our approach will lean more towards be ECA rule based planning, providing direct relation from events to action, which is ideal for a developer of a protocol.

For our initial solution, we will be choosing a mix of the centralized and partially centralized approach. Some nodes will be responsible for aggregating the telemetry data from their neighbors, and only these will deliver the aggregated data to the *Overlord*. Dissemination of adaptation instructions will be done in an epidemic broadcast fashion. During the elaboration of our work, we might return to these relevant aspects and, based on results from the first evaluation, reason if any other alternative should be taken.

3.3 Evaluation

To correctly evaluate the developed component, a series of tests will be conducted. These tests will compare systems running already established and tested protocols, which will be our case studies (discussed further ahead). For each test, we will compare a static version of the protocol, with a protocol running the developed component. Additionally, we subject both systems to the same conditions and will analyze how both systems behave in adverse conditions, such as adding a high number of nodes, removing nodes, among others.

As mentioned, we will use applications using already tested protocols as case studies, such as:

Broadcast Application A simple application for broadcasting a message to all peers in a network.

File Sharing Application Similar to applications such as Napster [37] and Gnutella [5]. Additionally, we will make use of a DHT to distribute files and to locate files (similarly to Pastry, in Section 2.1.7.4), making use of protocols like Chord [49] or Kademlia [34].

Both these applications will make use of some protocols, specifically for communication and membership of the system. Protocols that will be relevant in these applications constitute:

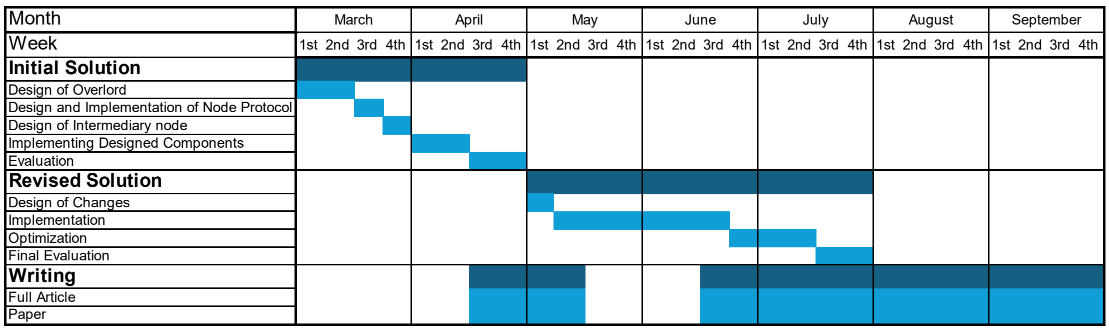


Figure 3.2: Gantt Chart Of Tasks and Their Expected Duration

Gossip-Based Protocols The epidemic broadcast approach. In our case, we will compare a system that runs a static *fanout* compared to a system that changes the *fanout* based on gathered performance.

HyParView Protocol [28] A membership protocol that ensures high levels of reliability in the presence of a high rate of failures. This protocol proposes two distinct views at each node, a small *active view* and a larger *passive view*, both of which can be tweaked considering the number of nodes in the system. Apart from these, there are other parameters such as a *shuffle period* and *check connectivity period*.

X-BOT Protocol [29] A protocol that allows influencing the way nodes are connected in a gossip overlay network to provide efficiency. Some protocol parameters that might be of interest are *Passive Scan Length*, which is the number of nodes scanned in each set of optimization rounds, and *Unbiased Neighbors and simply denoted*, which represents the number of "high-cost" neighbors each node keeps.

3.4 Planning and Scheduling

An estimation of the schedule of work is presented in Figure 3.2. The work was planned around a set of high level tasks described bellow:

Initial Solution Consists of first exploring the already implemented mechanisms in the Babel-Swarm framework for adaptation. Then, designing the *Overlord* component, in a way that it can integrate with the already existing Babel features. After this, we will be designing a protocol that will run in each node of the system, which will be responsible for providing adaptive capabilities for each node. Mainly, enabling monitoring of the node itself and its environment, as well as being prepared to receive adaptation instructions and enact those changes to the corresponding protocols. Finally, the design and implementation of the intermediary node, which will be responsible for aggregating data from its neighbors and sending it to the *Overlord*. This section also includes implementation of the defined case studies (Section 3.3), including testing them in adverse environment conditions and stress testing, to test robustness.

Refined Solution After the initial evaluation, we design and implement changes to the initial solution, resorting to the information retrieved from the first proposed solution. If any other architecture alternative could prove more viable, it will be considered during the re-designing phase. This phase also comprises the implementation of these changes and optimizations, as well as evaluation using the same case studies and environment conditions.

Writing It comprises writing of the dissertation elaboration document, and writing a paper, that will be submitted at a conference.

Bibliography

[1] R. Anthony. "A Policy-Definition Language and Prototype Implementation Library for Policy-based Autonomic Systems". In: *2006 IEEE International Conference on Autonomic Computing*. 2006 IEEE International Conference on Autonomic Computing. Dublin, Ireland: IEEE, 2006, pp. 265–276. ISBN: 978-1-4244-0175-8. DOI: 10.1109/ICAC.2006.1662407. URL: http://ieeexplore.ieee.org/document/1662407/ (visited on 2025-01-22) (cit. on p. 22).

[2] P. Arcaini, E. Riccobene, and P. Scandurra. "Modeling and Analyzing MAPE-K Feedback Loops for Self-Adaptation". In: *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*. 2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS). Florence, Italy: IEEE, 2015-05, pp. 13–23. ISBN: 978-0-7695-5567-6. DOI: 10.1109/SEAMS.2015.10. URL: http://ieeexplore.ieee.org/document/7194653/ (visited on 2025-01-22) (cit. on p. 17).

[3] R. Bakhshi et al. "Formal Analysis Techniques for Gossiping Protocols". In: *ACM SIGOPS Operating Systems Review* 41.5 (2007-10), pp. 28–36. ISSN: 0163-5980. DOI: 10.1145/1317379.1317385. URL: https://dl.acm.org/doi/10.1145/1317379.1317385 (visited on 2025-01-24) (cit. on p. 8).

[4] A. Bandara et al. "A Goal-Based Approach to Policy Refinement". In: *Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004.* Proceedings. Fifth IEEE International Workshop on Policies for Distributed Systems and Networks, 2004. POLICY 2004. Yorktown Heights, NY, USA: IEEE, 2004, pp. 229–239. ISBN: 978-0-7695-2141-1. DOI: 10.1109/POLICY.2004.1309175. URL: http://ieeexplore.ieee.org/document/1309175/ (visited on 2025-01-22) (cit. on p. 21).

[5] F. Bordignon and G. Tolosa. *Gnutella: Distributed System for Information Storage and Searching Model Description* (cit. on pp. 10, 13, 33).

[6] W. Cai et al. "Decentralized Applications: The Blockchain-Empowered Software System". In: *IEEE Access* 6 (2018), pp. 53019–53033. ISSN: 2169-3536. DOI: 10.1109

/ACCESS.2018.2870644. URL: https://ieeexplore.ieee.org/document/8466786/ (visited on 2025-02-04) (cit. on p. 1).

[7] J. Chomicki and J. Lobo. "Monitors for History-Based Policies". In: *Policies for Distributed Systems and Networks*. Ed. by M. Sloman, E. C. Lupu, and J. Lobo. Red. by G. Goos, J. Hartmanis, and J. Van Leeuwen. Vol. 1995. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 57–72. ISBN: 978-3-540-41610-4 978-3-540-44569-2. DOI: 10.1007/3-540-44569-2_4. URL: http://link.springer.com/10.1007/3-540-44569-2_4 (visited on 2025-01-23) (cit. on p. 22).

[8] I. Clarke et al. "Freenet: A Distributed Anonymous Information Storage and Retrieval System". In: *Designing Privacy Enhancing Technologies*. Ed. by H. Federrath. Red. by G. Goos, J. Hartmanis, and J. Van Leeuwen. Vol. 2009. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 46–66. ISBN: 978-3-540-41724-8 978-3-540-44702-3. DOI: 10.1007/3-540-44702-4_4. URL: http://link.springer.com/10.1007/3-540-44702-4_4 (visited on 2025-01-25) (cit. on p. 14).

[9] P. A. Costa and J. Leitao. "Practical Continuous Aggregation in Wireless Edge Environments". In: *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. 2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS). Salvador, Brazil: IEEE, 2018-10, pp. 41–50. ISBN: 978-1-5386-8301-9. DOI: 10.1109/SRDS.2018.00015. URL: https://ieeexplore.ieee.org/document/8613952/ (visited on 2025-01-22) (cit. on p. 19).

[10] P. A. Costa, J. Leitao, and Y. Psaras. "Studying the Workload of a Fully Decentralized Web3 System: IPFS". In: *Distributed Applications and Interoperable Systems*. Ed. by M. Patino-Martinez and J. Paulo. Vol. 13909. Cham: Springer Nature Switzerland, 2023, pp. 20–36. ISBN: 978-3-031-35259-1 978-3-031-35260-7. DOI: 10.1007/978-3-031-35260-7_2. URL: https://link.springer.com/10.1007/978-3-031-35260-7_2 (visited on 2025-02-04) (cit. on p. 1).

[11] N. Damianou et al. "The Ponder Policy Specification Language". In: *Policies for Distributed Systems and Networks*. Ed. by M. Sloman, E. C. Lupu, and J. Lobo. Red. by G. Goos, J. Hartmanis, and J. Van Leeuwen. Vol. 1995. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 18–38. ISBN: 978-3-540-41610-4 978-3-540-44569-2. DOI: 10.1007/3-540-44569-2_2. URL: http://link.springer.com/10.1007/3-540-44569-2_2 (visited on 2025-01-22) (cit. on p. 22).

[12] P. Druschel and A. Rowstron. "PAST: A Large-Scale, Persistent Peer-to-Peer Storage Utility". In: *Proceedings Eighth Workshop on Hot Topics in Operating Systems*. 8th Workshop on Hot Topic in Operating Systems. Elmau, Germany: IEEE Comput. Soc, 2001, pp. 75–80. ISBN: 978-0-7695-1040-8. DOI: 10.1109/HOTOS.2001.990064. URL: http://ieeexplore.ieee.org/document/990064/ (visited on 2025-01-25) (cit. on pp. 10, 15).

[13] Eng Keong Lua et al. “A Survey and Comparison of Peer-to-Peer Overlay Network Schemes”. In: *IEEE Communications Surveys & Tutorials* 7.2 (2005), pp. 72–93. ISSN: 1553-877X, 2373-745X. DOI: 10.1109/COMST.2005.1610546. URL: https://ieeexplore.ieee.org/document/1610546/ (visited on 2025-02-11) (cit. on p. 8).

[14] P. T. Eugster et al. “From Epidemics to Distributed Computing”. In: *IEEE Computer* 37 (2004), pp. 60–67. URL: https://api.semanticscholar.org/CorpusID:1489599 (cit. on p. 8).

[15] E. Fasolo et al. “In-Network Aggregation Techniques for Wireless Sensor Networks: A Survey”. In: *IEEE Wireless Communications* 14.2 (2007-04), pp. 70–87. ISSN: 1536-1284. DOI: 10.1109/MWC.2007.358967. URL: http://ieeexplore.ieee.org/document/4198169/ (visited on 2025-01-22) (cit. on p. 19).

[16] A. Feng et al. “In-Network Aggregation for Data Center Networks: A Survey”. In: *Computer Communications* 198 (2023-01), pp. 63–76. ISSN: 01403664. DOI: 10.1016/j.comcom.2022.11.004. URL: https://linkinghub.elsevier.com/retrieve/pii/S0140366422004273 (visited on 2025-01-22) (cit. on p. 19).

[17] P. Fouto et al. “Babel: A Framework for Developing Performant and Dependable Distributed Protocols”. In: *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. 2022 41st International Symposium on Reliable Distributed Systems (SRDS). Vienna, Austria: IEEE, 2022-09, pp. 146–155. ISBN: 978-1-6654-9753-4. DOI: 10.1109/SRDS55811.2022.00022. URL: https://ieeexplore.ieee.org/document/9996836/ (visited on 2025-01-22) (cit. on pp. 3, 29).

[18] D. Garlan et al. “Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure”. In: *Computer* 37.10 (2004-10), pp. 46–54. ISSN: 0018-9162. DOI: 10.1109/MC.2004.175. URL: http://ieeexplore.ieee.org/document/1350726/ (visited on 2025-02-11) (cit. on p. 28).

[19] N. S. Good and A. Krekelberg. “Usability and Privacy: A Study of Kazaa P2P File-Sharing”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI03: Human Factors in Computing Systems. Ft. Lauderdale Florida USA: ACM, 2003-04-05, pp. 137–144. ISBN: 978-1-58113-630-2. DOI: 10.1145/642611.642636. URL: https://dl.acm.org/doi/10.1145/642611.642636 (visited on 2025-01-25) (cit. on p. 10).

[20] K. Goudarzi and J. Kramer. “Maintaining Node Consistency in the Face of Dynamic Change”. In: *Proceedings of International Conference on Configurable Distributed Systems*. International Conference on Configurable Distributed Systems. Annapolis, MD, USA: IEEE Comput. Soc. Press, 1996, pp. 62–69. ISBN: 978-0-8186-7395-5. DOI: 10.1109/CDS.1996.509347. URL: http://ieeexplore.ieee.org/document/509347/ (visited on 2025-01-23) (cit. on p. 23).

[21] M. Hiltunen et al. "Survivability through Customization and Adaptability: The Cactus Approach". In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. DARPA Information Survivability Conference and Exposition. DISCEX'00. Vol. 1. Hilton Head, SC, USA: IEEE Comput. Soc, 1999, pp. 294–307. ISBN: 978-0-7695-0490-2. DOI: 10.1109/DISCEX.2000.825033. URL: http://ieeexplore.ieee.org/document/825033/ (visited on 2025-01-31) (cit. on p. 25).

[22] P. Horn. *Autonomic Computing: IBM's Perspective on the State of Information Technology*. New York, 2001 (cit. on p. 16).

[23] M. C. Huebscher and J. A. McCann. "A Survey of Autonomic Computing—Degrees, Models, and Applications". In: *ACM Computing Surveys* 40.3 (2008-08), pp. 1–28. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/1380584.1380585. URL: https://dl.acm.org/doi/10.1145/1380584.1380585 (visited on 2025-01-22) (cit. on pp. 16–18, 21, 23).

[24] H. Kamoda et al. *Policy Conflict Analysis Using Free Variable Tableaux for Access Control in Web Services Environments*. 2005-01 (cit. on p. 21).

[25] K. Kaur and Y. Kumar. "Swarm Intelligence and Its Applications towards Various Computing: A Systematic Review". In: *2020 International Conference on Intelligent Engineering and Management (ICIEM)*. 2020 International Conference on Intelligent Engineering and Management (ICIEM). London, United Kingdom: IEEE, 2020-06, pp. 57–62. ISBN: 978-1-7281-4097-1. DOI: 10.1109/ICIEM48762.2020.9160177. URL: https://ieeexplore.ieee.org/document/9160177/ (visited on 2025-02-07) (cit. on p. 1).

[26] J. Kephart. "Research Challenges of Autonomic Computing". In: *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* 27th International Conference on Software Engineering, 2005. ICSE 2005. St. Louis, MO, USA: IEEe, 2005, pp. 15–22. DOI: 10.1109/ICSE.2005.1553533. URL: http://ieeexplore.ieee.org/document/1553533/ (visited on 2025-01-23) (cit. on p. 17).

[27] J. Kramer and J. Magee. "Analysing Dynamic Change in Software Architectures: A Case Study". In: *Proceedings. Fourth International Conference on Configurable Distributed Systems (Cat. No.98EX159)*. Proceedings. Fourth International Conference on Configurable Distributed Systems (Cat. No.98EX159). Annapolis, MA, USA: IEEE Comput. Soc, 1998, pp. 91–100. ISBN: 978-0-8186-8451-7. DOI: 10.1109/CDS.1998.675762. URL: http://ieeexplore.ieee.org/document/675762/ (visited on 2025-01-23) (cit. on p. 23).

[28] J. Leitao, J. Pereira, and L. Rodrigues. "HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast". In: *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07). Edinburgh, UK: IEEE,

2007-06, pp. 419–429. ISBN: 978-0-7695-2855-7. DOI: 10.1109/DSN.2007.56. URL: http://ieeexplore.ieee.org/document/4272993/ (visited on 2025-02-06) (cit. on pp. 7, 34).

[29] J. C. A. Leitao et al. "X-BOT: A Protocol for Resilient Optimization of Unstructured Overlays". In: *2009 28th IEEE International Symposium on Reliable Distributed Systems*. 2009 28th IEEE International Symposium on Reliable Distributed Systems (SRDS). Niagara Falls, New York, USA: IEEE, 2009-09, pp. 236–245. ISBN: 978-0-7695-3826-6. DOI: 10.1109/SRDS.2009.20. URL: http://ieeexplore.ieee.org/document/5283246/ (visited on 2025-02-06) (cit. on p. 34).

[30] J. C. A. Leitão. "Topology Management for Unstructured Overlay Networks". 2010 (cit. on pp. 7–10).

[31] J. Lobo, R. Bhatia, and S. A. Naqvi. "A Policy Description Language". In: *AAAI/IAAI*. 1999. URL: https://api.semanticscholar.org/CorpusID:962745 (cit. on p. 22).

[32] E. Lupu and M. Sloman. "Conflicts in Policy-Based Distributed Systems Management". In: *IEEE Transactions on Software Engineering* 25.6 (Nov.-Dec./1999), pp. 852–869. ISSN: 00985589. DOI: 10.1109/32.824414. URL: http://ieeexplore.ieee.org/document/824414/ (visited on 2025-01-22) (cit. on p. 21).

[33] V. Martins, E. Pacitti, and P. Valduriez. "Survey of Data Replication in P2P Systems". INRIA, 2006 (cit. on p. 11).

[34] P. Maymounkov and D. Mazieres. "Kademlia: A Peer-to-Peer Information System Based on the XOR Metric". In: *Revised Papers from the First International Workshop on Peer-to-Peer Systems*. IPTPS '01. Berlin, Heidelberg: Springer-Verlag, 2002, pp. 53–65. ISBN: 3540441794 (cit. on pp. 8, 33).

[35] D. McCarthy and U. Dayal. "The Architecture of an Active Database Management System". In: *ACM SIGMOD Record* 18.2 (1989-06), pp. 215–224. ISSN: 0163-5808. DOI: 10.1145/66926.66946. URL: https://dl.acm.org/doi/10.1145/66926.66946 (visited on 2025-02-11) (cit. on p. 20).

[36] 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*. 21st International Conference on Distributed Computing Systems. Mesa, AZ, USA: IEEE Comput. Soc, 2001, pp. 707–710. ISBN: 978-0-7695-1077-4. DOI: 10.1109/ICDSC.2001.919005. URL: http://ieeexplore.ieee.org/document/919005/ (visited on 2025-01-22) (cit. on p. 26).

[37] *Napster*. URL: http://www.napster.com. (cit. on pp. 10, 33).

[38] M. Nofer et al. "Blockchain". In: *Business & Information Systems Engineering* 59.3 (2017-06), pp. 183–187. ISSN: 2363-7005, 1867-0202. DOI: 10.1007/s12599-017-0467-3. URL: http://link.springer.com/10.1007/s12599-017-0467-3 (visited on 2025-02-05) (cit. on p. 1).

[39] P. Oreizy et al. “An Architecture-Based Approach to Self-Adaptive Software”. In: *IEEE Intelligent Systems* 14.3 (1999-05), pp. 54–62. ISSN: 1094-7167. DOI: 10.1109/5254.769885. URL: http://ieeexplore.ieee.org/document/769885/ (visited on 2025-01-22) (cit. on p. 19).

[40] M. Parashar and S. Hariri. “Autonomic Computing: An Overview”. In: *Unconventional Programming Paradigms*. Ed. by J.-P. Banâtre et al. Red. by D. Hutchison et al. Vol. 3566. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 257–269. ISBN: 978-3-540-27884-9 978-3-540-31482-0. DOI: 10.1007/11527800_20. URL: http://link.springer.com/10.1007/11527800_20 (visited on 2025-01-22) (cit. on pp. 17, 18).

[41] L. D. Paulson. “Computer System, Heal Thyself.” In: *Computer* 35.8 (2002), pp. 20–22 (cit. on p. 16).

[42] B Pourebrahimi, K Bertels, and S Vassiliadis. “A survey of peer-to-peer networks”. In: *Proceedings of the 16th annual workshop on Circuits, Systems and Signal Processing*. 2005, pp. 570–577 (cit. on p. 6).

[43] S. Ratnasamy et al. “A Scalable Content-Addressable Network”. In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001-10), pp. 161–172. ISSN: 0146-4833. DOI: 10.1145/964723.383072. URL: https://dl.acm.org/doi/10.1145/964723.383072 (visited on 2025-01-25) (cit. on p. 14).

[44] L. Rosa, L. Rodrigues, and A. Lopes. *Appia to R-Appia: Refactoring a Protocol Composition Framework for Dynamic Reconfiguration*. di-fcul-tr-07-4. Department of Informatics, University of Lisbon, 2007-03. URL: http://hdl.handle.net/10455/2907 (cit. on p. 26).

[45] L. W. F. Rosa. “Self-Management of Systems Built from Adaptable Components”. PhD thesis. Instituto Superior Técnico, 2012 (cit. on pp. 17, 20, 24).

[46] A. Rowstron and P. Druschel. “Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems”. In: *Middleware 2001*. Ed. by R. Guerraoui. Red. by G. Goos, J. Hartmanis, and J. Van Leeuwen. Vol. 2218. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 329–350. ISBN: 978-3-540-42800-8 978-3-540-45518-9. DOI: 10.1007/3-540-45518-3_18. URL: http://link.springer.com/10.1007/3-540-45518-3_18 (visited on 2025-01-24) (cit. on pp. 7, 15).

[47] R. Schollmeier. “A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications”. In: *Proceedings First International Conference on Peer-to-Peer Computing*. First International Conference on Peer-to-Peer Computing. Linkoping, Sweden: IEEE Comput. Soc, 2002, pp. 101–102. ISBN: 978-0-7695-1503-8. DOI: 10.1109/P2P.2001.990434. URL: http://ieeexplore.ieee.org/document/990434/ (visited on 2025-01-24) (cit. on pp. 5, 6).

[48] D. E. Seborg. *Process Dynamics and Control*. Fourth edition. Hoboken, NJ: Wiley, 2017. 502 pp. ISBN: 978-1-119-28591-5 (cit. on p. 17).

[49] I. Stoica et al. “Chord: a scalable peer-to-peer lookup protocol for Internet applications”. In: *IEEE/ACM Transactions on Networking* 11.1 (2003), pp. 17–32. DOI: 10.1109/TNET.2002.808407 (cit. on pp. 8, 33).

[50] D. Sykes et al. “From Goals to Components: A Combined Approach to Self-Management”. In: *Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems*. ICSE ’08: International Conference on Software Engineering. Leipzig Germany: ACM, 2008-05-12, pp. 1–8. ISBN: 978-1-60558-037-1. DOI: 10.1145/1370018.1370020. URL: https://dl.acm.org/doi/10.1145/1370018.1370020 (visited on 2025-01-22) (cit. on p. 21).

[51] *TaRDIS: Trustworthy and Resilient Decentralised Intelligence for Edge Systems*. 2023. URL: https://cordis.europa.eu/project/id/101093006 (cit. on p. 3).

[52] S. Tarkoma. *Overlay Networks*. 0th ed. Auerbach Publications, 2010-02-09. ISBN: 978-1-4398-1373-7. DOI: 10.1201/9781439813737. URL: https://www.taylorfrancis.com/books/9781439813737 (visited on 2025-01-26) (cit. on p. 7).

[53] Y. Vandewoude et al. “Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates”. In: *IEEE Transactions on Software Engineering* 33.12 (2007-12), pp. 856–868. ISSN: 0098-5589. DOI: 10.1109/TSE.2007.70733. URL: http://ieeexplore.ieee.org/document/4359466/ (visited on 2025-01-23) (cit. on p. 24).

[54] Q. H. Vu, M. Lupu, and B. C. Ooi. *Peer-to-Peer Computing: Principles and Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010. ISBN: 978-3-642-03513-5 978-3-642-03514-2. DOI: 10.1007/978-3-642-03514-2. URL: https://link.springer.com/10.1007/978-3-642-03514-2 (visited on 2025-01-24) (cit. on p. 5).

[55] W. Walsh et al. “Utility Functions in Autonomic Systems”. In: *International Conference on Autonomic Computing, 2004. Proceedings.* International Conference on Autonomic Computing, 2004. New York, NY, USA: IEEE, 2004, pp. 70–77. ISBN: 978-0-7695-2114-5. DOI: 10.1109/ICAC.2004.1301349. URL: http://ieeexplore.ieee.org/document/1301349/ (visited on 2025-01-22) (cit. on p. 22).

[56] B. B. Yang and H. Garcia-Molina. *Designing a Super-Peer Network*. IEEE, 2003 (cit. on p. 6).