NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF DEPARTMENT OF COMPUTER SCIENCE

RAFAEL DOMINGUES MATOS

BSc in Computer Science

BABEL 2: A UNIFIED AND ADAPTIVE RUNTIME FOR DECENTRALIZED PROTOCOLS ACROSS DYNAMIC AND HETEROGENEOUS ENVIRONMENTS

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING
SPECIALIZATION IN DISTRIBUTED AND PARALLEL SYSTEMS

NOVA University Lisbon
February, 2026

NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF DEPARTMENT OF COMPUTER SCIENCE

BABEL 2: A UNIFIED AND ADAPTIVE RUNTIME FOR DECENTRALIZED PROTOCOLS ACROSS DYNAMIC AND HETEROGENEOUS ENVIRONMENTS

RAFAEL DOMINGUES MATOS

BSc in Computer Science

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

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING
SPECIALIZATION IN DISTRIBUTED AND PARALLEL SYSTEMS

NOVA University Lisbon
February, 2026

Abstract

Distributed systems now permeate all aspects of daily life and have evolved to exhibit many different architectures. Recent and emerging paradigms leverage peer-to-peer connections between mobile, power-constrained, and intermittently connected devices, and classic desktop devices with stable connections. This creates heterogeneous "swarm" environments where every device is working towards a common goal. The emergence of modular frameworks like Babel has somewhat streamlined distributed systems development by enabling modular composition of distributed protocols. However, there is a significant gap in their ability to handle real-world network complexities. Current solutions largely ignore the prevalence of restrictive NAT middleboxes controlling the network, and seamless transition across multiple network interfaces during the lifetime of an application. Additionally, existing frameworks fail to accommodate mobile device lifecycles where process suspension and energy constraints are a fundamental aspect of the application. Thus, developing resilient, leaderless decentralized systems across edge and mobile platforms remains a complex, time-consuming, and error-prone process.

In this work, we propose Babel 2, a unified and adaptive runtime designed to bridge these gaps by introducing transport-agnostic communication and automated lifecycle management. This new framework will integrate network interface migration at runtime and native NAT traversal, ensuring connectivity in restricted and dynamic settings. The framework will also introduce a family-based protocol registration system to reduce boilerplate code that compromises the modularity of the original Babel framework. The solution will be evaluated through implementation of a State Machine Replication and a gossip-based application leveraging different distributed protocol stacks across Linux desktops, Android devices, and Raspberry Pi Zero platforms. Experimental results are expected to demonstrate that Babel 2 achieves superior adaptability and reduced code complexity without compromising overall performance and reliability of applications.

Keywords: Distributed Systems Frameworks · NAT Traversal · Swarm Systems · Transport-Agnostic Communication

Resumo

Os sistemas distribuídos atualmente fazem parte de todos os aspetos da vida quotidiana e evoluíram para diversas arquiteturas. Novos paradigmas emergentes envolvem a exploração de ligações *peer-to-peer* entre dispositivos móveis, com restrições de energia e conectividade intermitente, e dispositivos fixos com ligações estáveis. Isto cria ambientes de *"swarm"* heterogéneos, onde todos os dispositivos trabalham para um objetivo comum. O aparecimento de *frameworks* modulares como o Babel simplificou o desenvolvimento de sistemas distribuídos, permitindo a composição modular de protocolos distribuídos. No entanto, existe uma falha significativa na sua capacidade de lidar com as complexidades das redes no mundo real. As soluções atuais ignoram largamente a prevalência de NAT *middleboxes* restritivas que controlam a rede, bem como a transição transparente entre múltiplas interfaces de rede durante o tempo de execução de uma aplicação. Adicionalmente, as *frameworks* existentes não conseguem acomodar os ciclos de vida de dispositivos móveis, onde a suspensão de processos e as restrições de energia são aspetos fundamentais. Assim, o desenvolvimento de sistemas descentralizados resilientes e sem líder em plataformas móveis e na *edge* continua a ser complexo, demorado e propenso a erros.

Neste trabalho, propomos o Babel 2, um ambiente de execução (runtime) unificado e adaptativo concebido para colmatar estas falhas através da introdução de comunicação agnóstica ao transporte e gestão automática do ciclo de vida. Esta nova *framework* integrará a migração de interfaces em tempo de execução e a travessia nativa de NAT, garantindo a conectividade em redes restritivas e ambientes dinâmicos. A *framework* introduzirá também um sistema de registo de famílias de protocolos para reduzir o código repetitivo que dificulta a modularidade da *framework* Babel original. A solução será avaliada através da implementação de uma aplicação baseada em replicação de máquina de estados e *gossip*, tirando partido de composições de protocolos distribuídos em ambientes de *desktop* Linux, Android e Raspberry Pi Zero. Espera-se que os resultados experimentais demonstrem que o Babel 2 consiga uma adaptabilidade superior e uma menor complexidade de código, sem comprometer o desempenho global e a fiabilidade das aplicações.

Palavras-chave: *Frameworks* de Sistemas Distribuídos · Travessia de NAT · Sistemas *swarm*

· Comunicação agnóstica ao transporte

Disclosure of Usage of Generative Artificial Intelligence

The author, Rafael Domingues Matos, acknowledges the use of the following generative artificial intelligence tools in the development of this thesis. These technologies were utilized under strict supervision, and all AI-generated outputs were critically reviewed and verified for accuracy. The author accepts full responsibility for the validity and originality of the content, confirming that the use of these tools aligns with all institutional policies and standards of academic integrity.[1]

Generative AI tools used: Gemini 3 Pro.

Activities supported by Generative Artificial Intelligence Tools:[2]

Conceptualization

- ☐ Idea generation
- ☐ Defining the research objective
- ☐ Formulating research questions and hypotheses
- ☐ Feasibility assessment and risk evaluation
- ☐ Hypothesis viability assessment
- ☐ Simulated debate and argument testing

Literature Review

- ☐ Search and Discovery
- ☐ Literature summarization and synthesis
- ☐ Concept Mapping and Systematization
- ☐ Market/patent landscape analysis
- ☐ Gap Identification and Novelty Evaluation
- ☐ Cross-lingual literature comprehension

Methodology

- ☐ Experimental Design Optimization
- ☐ Development of experimental or research protocols
- ☐ Methodological Instrument Selection

[1] This declaration was generated using XeLaTeX [44] and the LaTeX package `aidisclose` [31].
[2] This list extends the GAIDeT taxonomy [42].

Software Development and Automation

- [ ] Code Generation
- [ ] Debugging and Repair
- [ ] Algorithm design
- [ ] Refactoring and Optimization
- [ ] Process automation
- [ ] Code documentation and comment generation

Data Management

- [ ] Data collection
- [ ] Data analysis
- [ ] Synthetic data generation
- [ ] Validation
- [ ] Quantitative Plotting and Charting
- [ ] De-identification and anonymization support
- [ ] Data cleaning
- [ ] Reproducibility and rerun checks
- [ ] Audio-to-text transcription and diarization
- [ ] Data curation and organization
- [ ] Data labeling and annotation assistance

Visuals and Multimedia

- [ ] Synthetic Asset Generation
- [ ] Image Enhancement and Editing
- [ ] Diagrammatic and Schematic Design

Writing and Editing

- [ ] Drafting Text
- [ ] Formulation of conclusions
- [ ] Citation formatting and bibliography management
- [x] **Polishing and Editing**
- [ ] Tone Adjustment
- [ ] Press releases and outreach materials
- [ ] Abstract and Executive Summary Generation
- [x] **Translation**
- [ ] Title Generation

Ethics Review

- [ ] Bias analysis and discrimination assessment
- [ ] Ethical risk analysis
- [ ] Data confidentiality monitoring
- [ ] Monitoring compliance with ethical standards

Quality Assurance

- [ ] Simulated Peer Review
- [ ] Identification of limitations
- [ ] Publication support
- [ ] Consistency checking against field trends
- [ ] Publication strategy and journal selection

Additional comment #1: Generative AI tools were used to find and fix typos and grammar issues in the entire text. They were also used to translate the abstract to portuguese.

Contents

Abstract ii

Resumo iii

Disclosure of Usage of Generative Artificial Intelligence v

Contents vii

List of Figures ix

1 Introduction 1

1.1 Context . . . 1

1.2 Motivation . . . 2

1.3 Expected Contributions . . . 2

1.4 Document Organization . . . 3

2 Related Work 4

2.1 Relevant Technologies . . . 4

2.1.1 QUIC . . . 4

2.1.2 NAT Traversal . . . 5

2.1.3 Netty . . . 8

2.1.4 Identity Management . . . 8

2.2 Library Based Solutions . . . 9

2.2.1 x-kernel . . . 9

2.2.2 Cactus . . . 10

2.2.3 Horus . . . 11

2.2.4 Appia . . . 12

2.2.5 Babel . . . 12

2.2.6 libp2p . . . 14

2.2.7 Iroh . . . 14

2.2.8 Akka . . . 15

2.2.9 Summary . . . 15

2.3 Non-Library Based Solutions . . . 16

2.3.1 Domain Specific Languages 16
2.3.2 BFT-SMaRt 17
2.3.3 Summary 18
2.4 Discussion 18
2.5 Summary 19

3 Work Plan **20**
3.1 Proposed Solution 20
3.1.1 Network 21
3.1.2 Core 22
3.1.3 Protocols 23
3.2 Experimental Evaluation 24
3.3 Schedule 25
3.4 Summary 26

Bibliography **28**

List of Figures

2.1 Graph formed by the x-kernel protocol and session interactions. Figure extracted from [18]. 9
2.2 Cactus Composition. Figure extracted from [34]. 11
2.3 Appia Composition. Figure extracted from [34] 12

3.1 Babel Architecture. Figure extracted from [11]. 21
3.2 Thesis Schedule (March 2026 – September 2026) 27

1 Introduction

1.1 Context

Distributed systems are increasingly relevant and present in our everyday lives. The Internet is becoming more present in each aspect of human life which leads to higher demands for reliability, security, and computing power. This growing demand has led to more time devoted to developing, validating, and maintaining these systems [47]. Due to this, development of these systems has grown more complex requiring more development time, effort, and leading to more complex and harder to identify bugs. Chandra et. al [5] have noted that implementing a performant and correct Paxos [24] implementation from zero is a non-trivial task.

Additionally, distributed systems need to adapt to more scenarios than before. One such case is the Internet of Things (IoT) and Edge Computing which led to more devices connected, additional heterogeneity, and more failure scenarios. Networks have evolved to accommodate these devices with new network technologies such as Bluetooth Low Energy (BLE), and LoRa. These new devices are also constrained in computational power, energy, and communication capacity. With more devices connected, there are more possibilities for attack vectors. Simple WiFi equipped microcontrollers can carry out attacks on IoT devices [45]. Developers must take into account all these aspects - heterogeneity, resource and communication constraints, and security and privacy - into new distributed systems that integrate a diverse set of devices. One such example are swarm systems, where various and heterogeneous devices interact between each other without a leader towards a common goal. Swarm systems can be composed of IoT devices - e.g. multiple sensors in a forest could help detect fires earlier [32].

Alongside these new architectural challenges, developing these systems involves dealing with low level implementation details - sockets, threads, memory management - as well as the design of the protocols that compose and support the operation of these systems. These protocols are described in pseudocode or natural language in their original papers. One example is the Lamport's distributed mutual exclusion algorithm which was first described in five simple rules in natural language [25]. Mapping these abstract pseudocode or natural language descriptions to concrete implementations for new IoT and

Edge computing scenarios is non-trivial. One must take into account things that were a given in desktop and server environments - e.g. energy is not a problem in these environments, but is limited in mobile environments.

1.2 Motivation

To accommodate these challenges, middleware, frameworks, domain specific languages (DSLs), and very specialized solutions have been explored and developed to streamline distributed systems development [40]. We will call them generic distributed protocol frameworks, just frameworks from now on. They abstract away the communication details, have a close relationship to the original pseudocode, and accommodate a wide range of distributed system necessities to improve development time and reduce faulty behavior.

Existing frameworks take very different approaches and focus on different ways to improve development iteration speed. These different approaches have many advantages and disadvantages depending on their goal, and many have compared frameworks [34]. One clear advantage is their modularity. As previously stated, this is a common aspect among these tools. Distributed systems can take many forms, and their supporting protocols need to be easily implemented with these tools. They are then composed to form a stack that provides the necessary services. However, one disadvantage is the execution environments.

Existing frameworks have largely ignored mobile environments only focusing on desktop and server environments. If one wishes to develop a distributed system incorporating a diverse set of devices, little to no support would be provided by these frameworks, as we will discuss in later sections. One solution could be using various frameworks to support all these devices. However, this could easily become cumbersome if these frameworks don't interoperate seamlessly. Despite their modularity, frameworks often have a rigid static configuration that cannot be changed during runtime.

Network conditions vary across a diverse set of devices. The protocols that base these distributed systems often ignore the network conditions and assume a direct connection between nodes of the system. However, this is often not the case in a heterogeneous environments. Devices can be connected through a myriad of means other than traditional Ethernet, such as WiFi, cellular network, etc. Alongside the connection method, other constraints such as firewalls and Network Address Translator (NAT) boxes limit the capacity for establishing a direct connection, as we will discuss in subsection 2.1.2.

1.3 Expected Contributions

This thesis proposes tackling these challenges by expanding the existing Babel framework [11] and its variants [13], all of them developed in NOVA LINCS and described in subsection 2.2.5. This will culminate in a new unified version capable of managing

multiple environments. The new version of Babel to be designed and developed should support:

- A new transport-agnostic communication system capable of switching interfaces during runtime and establishing direct connections even behind restrictive firewalls and NATs.
- A new runtime management system to adapt to various constrained systems. This new runtime management should be able to be adapted to mobile and IoT systems. Intermittent connections, possible process suspensions and energy constraints are also consider in the design.
- Expand upon the main abstractions available in Babel. A more powerful feature set should make distributed system development easier, less error prone. Babel still necessitates boilerplate for common tasks in distributed algorithms - e.g. broadcasting to all known peers.
- A more robust and easy to use set of security primitives. A new and improved security library must be ready to integrate more dynamic transport and runtime environments. Current works have a clear emphasis on privacy and security, while Babel has clear gaps in this area.

1.4 Document Organization

The remainder of this document is structured as follows:

Chapter 2 presents the previous work done related to our objectives. This chapter will cover relevant technologies to tackle the communication problems, and runtime management; as well as existing solutions that are conceptually close to Babel, and more distinct solutions.

Chapter 3 presents the proposed work plan to implement the proposed solution and evaluate it.

2

Related Work

In this chapter, we discuss many frameworks that try to make the development of distributed algorithms and systems easier. Before discussing those frameworks, we will cover some technologies that make these frameworks possible.

Many approaches have been explored throughout the years. As such, we will split the frameworks in three categories: domain specific languages, library-based solutions, and non-generic solutions.

2.1 Relevant Technologies

In this section, we will cover relevant technologies that are essential to understand the discussion about frameworks later in this chapter, as well as our own proposal in chapter 2.

2.1.1 QUIC

QUIC is a transport protocol built by Google that operates on top of UDP. According to the original 2017 paper, 7% of the Internet traffic is QUIC [26]. In 2023, Cloudflare reported that about 30% of the HTTP requests were executed by relying on QUIC (HTTP/3) [10]. QUIC tries to overcome some of the limitations of the TLS/TCP ecosystem.

The first limitation is protocol entrenchment. TCP has become the default transport protocol in many scenarios. The middleboxes that sit at key control points of the Internet - such as NATs and firewalls - tend to rewrite transport headers and block unusual traffic, respectively. Innovation in this area is very slow due to the way TCP is expected to work in these middleboxes.

The second limitation is implementation entrenchment. TCP is implemented at the operating system (OS) kernel level. With the constant stream of attacks on Internet infrastructure, the TCP stack on the kernel may need to be updated. OS updates are slow due to the system-wide implications they bring. This limits iteration velocity for simple networking changes.

The third limitation is the handshake delay. TCP has the well known three way handshake to establish connections. On top of that, most of the Internet connections are done

with TLS for added encryption and security. TLS adds two round trips to the TCP handshake. All these additional communication steps pile up, specially for short transfers and interactions over HTTPS.

The fourth limitation is head-of-line blocking. TCP guarantees delivery and the order of packets between sender and receiver. If old packets need to be retransmitted, that can incur delays on newer packets and affect concurrent requests.

QUIC implements several features to mitigate the mentioned limitations of TCP. QUIC features a combination of both cryptographic and transport handshakes to reduce the number of round trips compared to TCP/TLS. It features a multiplexing feature providing each message a unique stream so that no message can block another for being late and a flow control algorithm so that no receiver buffer exhausts.

Since QUIC is implemented in user land, it can be quickly iterated and deployed to all clients. QUIC is also mostly encrypted which limits tampering by middleboxes and avoids protocol development stagnation. Most importantly, QUIC features a Connection ID on its header to identify connections between nodes. This effectively decouples the IP layer details from the connection identity, allowing a fast and transparent transition between addresses by both nodes without a new handshake. New network conditions can happen in multiple scenarios: clients can change the IP address when changing interfaces (e.g. WiFi to Ethernet or WiFi to 4G), using a new address due to the DHCP lease ending or NAT port rebinding. As stated in chapter 1, this is going to be important for the implementation of our solution.

The original paper shows that a latency reduction was possible in most latency scenarios in both web search and video playback with higher gains in high latency scenarios.

Babel and its variants, described further in subsection 2.2.5, do not currently support QUIC, which is a significant limitation in multiple domains. Some of the more modern solutions in the next section use QUIC for their transport needs.

2.1.2 NAT Traversal

Network Address Translation (NAT) [8] is a method to map one address group to another. This method is used to preserve the number of IPv4 addresses since all the public IPv4 address space has exhausted [12]. Organizations now rely on IPv6 larger address space or NAT boxes that sit between the private network and the public Internet to connect a large number of devices [48].

Despite continued efforts to migrate the Internet to the newer IP standard - IPv6 - most of the Internet traffic is still made through IPv4 and, consequently, behind NAT boxes. Google reports that during the second half of 2025, only 45% to 50% of the traffic to their websites was made through IPv6 [20], thus taking NATs into account is still a necessity.

Due to the way NATs work, many devices can share a single public IP address, making establishing a direct connection to any of these devices hard. According to RFC 4787 [21], there are many types of NATs that we must take into account:

Endpoint-Independent Mapping The NAT reuses the same mapping for all packets sent from the same private address and port to any external address and port.

Address-Dependent Mapping The NAT reuses the same mapping for all packets sent from the same private address and port to the same external address regardless of port.

Address and Port-Dependent Mapping The NAT reuses the same mapping for all packets sent from the same private address and port to the same address and port while the mapping is active.

This is the terminology we will use further on as this is the most recent RFC. Other literature refers to these using different terms. Different strategies are employed to circumvent each type of NAT. Currently, Babel has no native support to deal with NAT boxes limiting connectivity between processes that have a public or accessible (within a local network) address in the machine where it executes. This limits significantly the ability to deploy distributed applications and systems in practice.

We now discuss common techniques to deal with NAT traversal (sometimes called hole punching).

2.1.2.1 STUN

Session Traversal Utilities for NAT (STUN) [37] is client-server protocol to determine the address and port allocated by a NAT that maps to a private address. STUN is used alongside other protocols to assist in NAT traversal.

STUN has two possible transactions, a request/response transaction and an indication. The most crucial mechanism is the Binding method. A STUN client (that sits behind a NAT) sends a Binding request to a STUN server. That request will pass through a NAT that will change the address and port header in the packet. Once it arrives at the STUN server, it will send a response back in the packet body as a `XOR-MAPPED-ADDRESS` attribute. The response will pass through the NAT and keep the body intact. The client will then know what is its address and port available through the NAT. STUN also provides a keep alive mechanism to keep the NAT binding up. STUN can effectively be used with Endpoint-Independent Mapping and Address-Dependent Mapping because the client will always present itself with the same address to the server. There are no such guarantees when the NAT uses an Address and Port-Dependent Mapping. More tools need to be deployed to make a full NAT traversal solution.

2.1.2.2 TURN

Traversal Using Relays around NAT (TURN) [38] is a protocol that allows two nodes behind restrictive NATs to communicate. In Address and Port-Dependent NATs, relying only on STUN to make a binding in the NAT is impossible. Every time any of the peers

try to communicate with each other, a new address and port is allocated on the NAT box, making it unfeasible to establish a stable connection.

TURN allows two nodes behind NATs to use a relay server on the public Internet to communicate between each other. One node may ask for a relayed address on the TURN server with a permission lifetime. Any other node that wishes to communicate with this one, can send their information to the relayed address on the TURN server as long as the set permission allows. This allows nodes behind very restrictive NATs to communicate through a middleman.

There are some trade-offs to using a relay server. It introduces some latency between nodes. Bandwidth and cost constraints can also make relay servers not feasible. Usually, TURN and relay servers should be used as a last resort, however, in certain scenarios with very restrictive NATs - e.g. devices connected through mobile data and devices in an enterprise network - TURN is the only pragmatic solution.

2.1.2.3 ICE

Interactive Connectivity Establishment (ICE) [22] is a protocol for finding communication paths between nodes. Initially, the nodes are unaware of their network topology - they may or may not be behind one or several NAT boxes. ICE is not a signaling protocol and is not concerned with how the nodes are aware of each other or how they send each other information before establishing the connection.

Firstly, the ICE protocol will gather all the possible candidates for an address. Any node can have multiple candidates for connection: an address associated to an attached network interface, a public available mapping on the NAT box - which can be found through STUN -, or a relayed address available in a TURN server. Once the gathering finishes, both nodes will send each other the candidates through the signaling protocol.

Once they are aware of the possible candidates, each node at the time will send STUN requests to each candidate ordered by a priority order. Usually, nodes prioritize candidates without NATs or relays. Once a STUN request/response concludes successfully in both ways, the candidate addresses are added to a valid list. Once all the candidates are checked, the handshake finishes.

ICE Lite implementations exist for when a node has direct access to a public IP address. These nodes do not run connectivity checks but need to answer any incoming check. Of course, this is only feasible if one of the nodes has direct access to a public address.

2.1.2.4 Summary

These tools and techniques are used in well known technologies that rely on P2P connectivity, such as WebRTC [41]. This is, of course, important to consider when creating a new framework focused on distributed protocols and systems developments as we will need to accomodate a myriad of different connectivity environments. In the next section, some of the solutions we will discuss implement some form of NAT traversal.

2.1.3 Netty

Netty [36] is the industry standard for asynchronous event-driven network application framework in Java. It boasts powerful abstractions to simplify newtork communication with native support for well established protocols such as WebSockets, TLS and HTTP, and decouple application logic from the network logic.

Channels are the main way an application can interact with the network. They are abstractions over the classic network sockets and are fully non blocking - read and write operations happen on different threads other than the main application threads. Developers can assign an `EventLoop` that will handle all I/O events from the channels. These `EventLoops` can be a single thread that will process all the events in a sequential fashion or be a multi thread pool that can handle multiple requests at the same time. Each channel is equipped with a pipeline that will treat the incoming or outcoming data. The pipeline consists of a series of `ChannelHandlers` that can modularize logic - e.g. one handler can treat the protocol-specific messages serialization while another handler signs and encrypt messages.

Of course, this is just an abstraction for the network logic. Netty does not provide a quick and easy way of switching interfaces on the fly. Each channel is bound to a single interface or all of them (similiarly to what happens with sockets). Dynamic reconfigurations need to be built on top of Netty by developers which also requires signalling applications.

In the next section, some of the solutions presented rely on Netty for the network logic.

2.1.4 Identity Management

In distributed systems, we need to know the identity of whoever we are communicating with. This can be done through certificates. If two nodes exchange certificates, they can establish encrypted connections. However, one must assertain if the certificates received are valid or not. We already established that a new framework must integrate QUIC in its transport protocols, and, since QUIC must be encrypted, we must find away to manage identities. Although there are many ways to do this, as explored by Li et al. in [27] and more recently by Mazzoca et al. in [33], we will only cover two models that we believe must be integrated in a new framework:

Central Authority (CA) In this model, a central trusted authority issues certificates that every node will trust. They can be easily revoked by the autority.

Self-Certified Identities In this model, every node can generate its own identity with a public/private key pair. Every time each node pair begins communicating, they ensure that the other node is in possession of the public key they announce they have.

For static clusters with a stable conncetion, a CA can be employed to ensure no malicious nodes are present. However, that is not possible on a heterogeneous environment

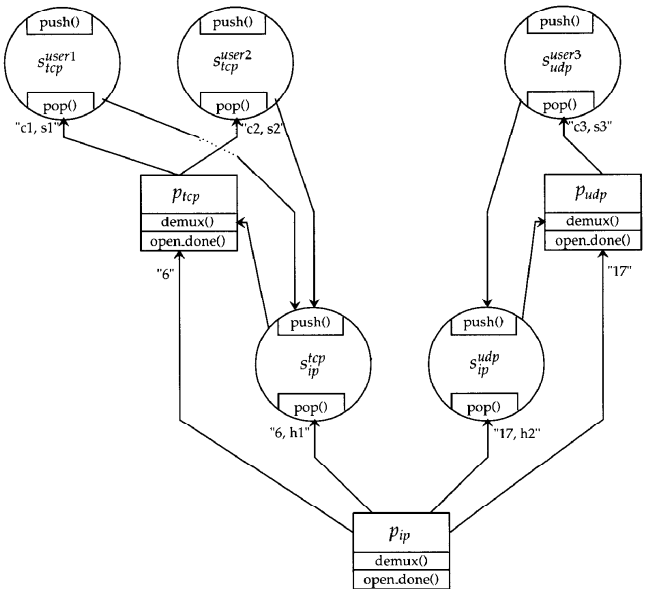

Figure 2.1: Graph formed by the x-kernel protocol and session interactions. Figure extracted from [18].

with intermittent connection. Some of the solutions in the next sections employ a Self-Certified solution. Since we are striving for a new unified framework, we will accept both forms of authentication, with a clear emphasis on Self-Certified identities.

2.2 Library Based Solutions

In this section we take a look at all solutions to support development of distributed protocols, applications, and systems which are provided to developers in some form of sofware packages. In other words, some primitives (classes, functions, data types...) are provided to be used in one specific general purpose programming language or many.

2.2.1 x-kernel

x-kernel [18] is one of the foundational works for this field. x-kernel was developed in C at the University of Arizona. It introduced a modular architecture that lets developers stack different protocols instead of working with a monolith. Interactions between these protocols are well defined through a common interface all protocols must implement.

x-kernel presents three main abstractions:

Protocols Defines the relationship between each protocol, is defined at boot time.

Sessions They are instances of protocol objects. They contain the protocol code and all the necessary data structures.

Messages Active objects that move through the protocols.

Each protocol can instantiate multiple sessions according to a specific implementation. For example, an IP implementation might need to demux to know where to send the message, either to an UDP or TCP session. Sending the information up and down the stack can be done through a `pop` and a `push` commands, respectively. This common interface allows any protocol to communicate with any protocol, although there's no guarantees it will work. This is called the Uniform Protocol Interface (UPI). All these interactions between protocols and sessions form a graph like the one in Figure 2.1. Each protocol is indicated as a square and has a `demux` operation. That demux operation should direct an incoming message to the correct session for that protocol, indicated as circles. In Figure 2.1, we can see an example with IP having two possible sessions to demux: TCP and UDP. Each of these has a protocol to do the demuxing to know which of the user implementations to forward.

This Lego-like stacking of protocols is a pattern we will see throughout all the solutions in this section. However, execution models differ. x-kernel has a unique execution model as it makes a process for evey message that arrives at a node. This will differ significantly from the next solutions.

Due to its tight integration with the OS kernel, a lot of code relies on running in privileged mode. This makes implementation and debugging more complex, as the original paper's authors state. Faulty or malicious code running in privileged mode can be disastrous for system stability and a big risk for security.

2.2.2 Cactus

Cactus [17] is a framework developed at the University of Arizona with many implementations available in multiple languages. In the paper, the authors argue that survivability, the ability of a system to tolerate attacks and failures, can be improved with fine-grain customization and dynamic adaptation. As such, Cactus provides support for both and allows developers to create protocol stacks that are both customized and adaptable during runtime.

Cactus presents us with the same idea of a stack of protocols as x-kernel with some key differences, namely the departure from priveliged mode and more granular protocol composition. There are now two categories for protocols:

Micro-Protocol A collection of event handlers. They implement a small subset of interactions for a desired service in an event-driven way.

Composite Protocol A collection of micro-protocols, shared memory and events. Micro-protocols interact between each other inside a composite protocol. Within a composite protocol, micro-protocols interact by raising events - which are handled atomically - or by readind and writing to the composite protocol's memory.

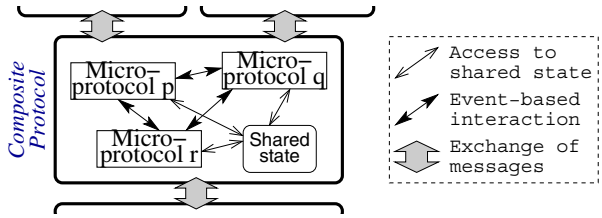

Figure 2.2: Cactus Composition. Figure extracted from [34].

Cactus also provides a message abstraction with a header and a body composed of attributes. Those attributes can have a local (only available inside the composite protocol), stack (available throughout the stack) or peer scopes (available to both the message sender and receiver nodes). Micro-protocols can add, retrieve and delete these attributes independently, facilitating protocol composition.

This framework doesn't dictate how composite protocols should interact, but raises the possibility of interacting much like the UPI in x-kernel and other mechanisms. In [34], a message passing solution is implemented. In Figure 2.2, we can see all the possible interactions in Cactus.

Since events can be raised and handled by any micro-protocol, adaptability is easily achieved with the handlers during runtime. The original authors also present other adaptability scenarios out of the scope for this work.

2.2.3 Horus

Horus [46] is an object oriented protocol composition framework created at Cornell University. It deviates from the low level aspects of x-kernel and the complex protocol compositon of Cactus. Instead, Horus leverages the object-oriented concepts by treating a protocol as an abstract data type. Other protocols or applications sit atop or below in a hiearchical compositon (or stack like).

Horus provides abstractions for common needs, such as endpoints, group objects, messages, and threads. Endpoints can receive and send messages and have addresses for membership purposes. Each stack of protocols has an endpoint. Group objects maintain protocol state. Message objects keep data structures added by each protocol as headers. Usually, message objects contain the same data as the messages sent in the network.

Besides these abstractions, Horus has special protocol types ready to be stacked and used for common things such as membership protocols or message signing. A common protocol interface is also present to allow the protocols to be stacked in any way the developer wants.

While this proves that Horus is very extensible, little work is done on protocol lifecycle and dynamic configuration with no clear ability to handle interruptions. Its network capabilities are closely tied to traditional IP networks. Despite Horus being a multithreaded

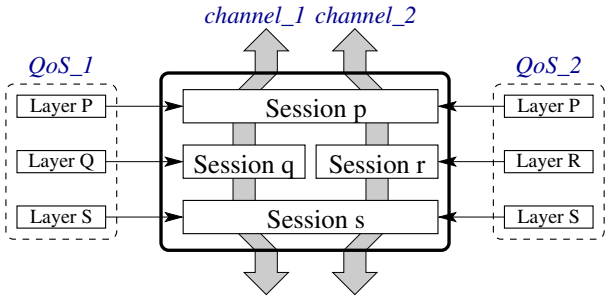

Figure 2.3: Appia Composition. Figure extracted from [34]

solution, threads seem to still be a burden on the developer end as synchronization needs to be manually performed by the developer.

2.2.4 Appia

Appia [35] improved on the modular nature of Horus in a more modern setting. Appia is a framework specialized in protocol composition developed in Java at University of Lisbon.

Appia diverges from Horus in the provided abstractions. Protocols and sessions behave in the same way, as protocols are a specification and sessions are an instantiation of the protocol. Since Appia was developed in Java, sessions are just object instances of a class/protocol. Its new feature, compared to its predecessors, is that it allows for coordinated channels. That is, it allows two independent channels to share sessions across layers, as depicted in Figure 2.3.

Appia doesn't follow the same graph structure as x-kernel, instead opting for a stack paradigm. A stack is composed of protocols in its layers. Unlike Horus, each layer can have multiple sessions. For instance, a broadcast protocol can have multiple sessions for different levels of reliability. The stack is referred to as a Quality of Service (QoS). This provides extensibility during configuration and before runtime, but no work is done on adaptability during runtime. Another limitation to this approach, as with Horus, is that inter-layers communication is limited to the layer up and below in the stack.

Unlike Horus, the whole stack operates under a single thread, reducing implementation complexity, but limiting potential performance. In Figure 2.3, we can see two sessions in a single layer. Despite working independently, no concurrency is implemented to let sessions on the same layer to execute simultaneously.

2.2.5 Babel

Babel [11] is a framework developed in Java at NOVA University of Lisbon and NOVA LINCS which aims to simplify the development process of distributed protocols and algorithms. It follows the same architecture as the one present in Horus with object oriented and event-driven programming.

Babel is composed of three main components: protocols, core, and network.

The protocols extend the `GenericProtocol` abstract class. Each protocol has its dedicated thread and event queue. This allows multiple protocols to easily run concurrently, shielding the developer from complex synchronization logic. The core coordinates the interactions and events by delivering them to the interested protocols. It also handles timers and their triggers. Lastly, the network layer is responsible for network communication and is built on top of the previously discussed Netty. These layers abstracts networking into channels which developers can choose the ones needed for their needs (e.g. bi-directional `TCPChannel`, client/server) or implement their own.

All these components can be interacted with through a simple API to register handles for the events created by the core (e.g. timers triggering), the network (e.g. channels going down or up) or the custom made events for the protocols. The custom made events can be delivered to only one protocol (called requests and replies) or many protocols (called notifications).

With these abstractions in place, the authors were able to reduce the lines of code needed to implement a State Machine Replication (SMR) using the Paxos protocol [24] compared to specialized and naive implementation. The Babel implementation achieved better performance compared to naive Paxos implementations and comparable performance to the specialized implementation.

2.2.5.1 Babel Swarm

Babel Swarm [13] is an extension of the Babel framework that was developed in the context of the EU project TaRDIS [43] tailored for intelligent swarm systems. It features extensions for automatic configuration, security and autonomic management.

Automatic configuration is composed of two mechanisms: contact discovery and parameter configuration. Contact discovery is done through a custom discovery protocol that multicasts messages through the network to find other nodes with the necessary services. Parameter configuration can be done through DNS TXT record queries or by copying configurations from other peers. The found parameters are injected through Java reflection and annotations.

Security was added on top of Babel's existing network layer. It provides different types of secure channels with different guarantees. The authentication done by these channels is managed by a new identity manager that can be configured with multiple security policies (e.g. blindly trusting any certificate presented by other nodes or waiting for enough neighbours to attest its authenticity). Encryption is exposed through an API built on top of Java's Cryptography Architecture and Bouncy Castle.

Autonomic management is responsible to adapting the protocols parameters to the conditions of the network. In the Babel Swarm paper, the authors implement an autonomic controller that has a Random Tour protocol to estimate the network size and adapat a gossip protocol with parameters more suited to the estimated network size.

The evaluation performed over this variant (when compared with the original Babel)

showed that with the addition of these features, adapting existing protocols to use these features wouldn't ask for a significant effort by the developers. Adding auto-configuration required 180 additional lines of code, security required 63 lines, and autonomic features required 35 lines. No significant impacts were found in resource usage and latency and throughput stayed on par without the extensions.

Despite its name, work can still be done in the swarm front. No work was done in this version for dynamic interface swapping and network adaptability.

2.2.6 libp2p

libp2p [28] is a highly modular library that facilitates the development of peer-to-peer (P2P) applications developed by Protocol Labs. It is used in some very well known P2P applications such as the InterPlanetary File System (IPFS) [2] and Ethereum. libp2p was first developed as a networking layer for IPFS by Protocol Labs, and as since then, evolved to be an independent project.

This library has multiple modules for common needs in P2P application such as service discovery, NAT traversal, secure communication, among others. One of the more innovative features, was the use of a new addressing scheme called multiaddress. Instead of only relying on common IP addresses, multiaddress adds more layers in a human-readable way while being easily parsable by a machine. This ensures that libp2p can work in a variety of networks independent of the finer details of the addressing scheme. One might use the conventional IP addresses or a known public identity and contact the node over all the known IP addresses for that node. More information can be conveyed in this multiaddress format, such as what transport protocol we want to use (TCP, UDP, QUIC, TLS, etc.), if we are using a relay to contact the node or what kind of encryption we will use on the communication.

libp2p is widely used in production-grade applications. However, this theoritically modular design often delivers a pre-bundled and rigid stack of protocols, since libp2p already provides a variety of them. There is a lack of user implemented protocols. Furthermore, some reaserchers note a time consuming setup for some functionalities who implemented protocols in libp2p [14]. They state that the setup steps for peer discovery are time consuming.

2.2.7 Iroh

Iroh [19] is a framework written in Rust and developed by `n0` company for facilitating P2P communication. It is built on top of a QUIC. Iroh lets nodes connect by their endpoint ID, which is a public key. Communication between nodes can be established through multiple ways. In case nodes sit behind a NAT or a firewall, holepunching is employed to ensure a direct connection between them. Besides this, encryption is integrated and easily deployable with QUIC and the public keys.

Despite being a library used for P2P communication, relay servers are needed to effectively traverse NATs and firewalls. There are relay servers operated by `n0` themselves and the relay server's code is open source. Iroh is also on path to adapt QUIC's multipath extension which essentially lets two nodes exchange information over one connection through multiple paths and interfaces.

Similiar to libp2p, Iroh has many pre-made protocols available for use like gossip protocols and can be extended with our own protocol implementations - which is just a small section in the documentation page. However, since Iroh is implemented in Rust, it limits its reachability in various devices. Some low level adaptations may need to be done to accommodate the Rust ecosystem in a new device.

2.2.8 Akka

Akka [9] is a framework and runtime for building distributed message-driven applications on the JVM - specially Java and Scala. It's main concept is the implementation of the Actor Model which was first introduced in 1978 [16]. Each actor has a message queue. Whenever an actor receives a message in the queue, they will execute some code and may send messages to other actors.

When designing the actors and their interactions, developers don't need to know where an actor "lives". They may be available locally or in a remote node. Actors are supervised by a parent that decides whether to resume, stop, or restart a child [39]. This provides resilience to the system.

While Akka provides a mature actor model implementation, its cluster management relies on a single leader, which is unsuitable for leaderless swarm systems. Additionally, all nodes - that accommodate various actors - need to agree on the cluster state for the leader to add or remove members, an act called gossip convergence [6]. This can't happen if any of the nodes become unreachable, a common ocorrence with intermittent connections in mobile environments. The entire network could become unresponsive due to some nodes becoming unreachable.

2.2.9 Summary

In this section, we covered relevant library based solutions, from foundational work to modern approaches. We discussed the modularity provided by all the solutions and the advantages and disadvantages of them. These solutions take a wide spectrum of appraoches regarding protocol composition with some being more liberal, like x-kernel, while others are more restrictive, like libp2p.

Additionally, we looked over their adaptability, namely dynamic reconfiguration and network adaptability. We discussed how centralized operation in Akka is not suitable for swarm environments, and how some solutions do not provide a full suite capable of traversing network complexities like Babel.

The key takeway from this section is that a new solution must be flexible enough in protocol composition and be adaptable to a wide variety of network scenarions.

2.3 Non-Library Based Solutions

In this section we will cover non-library based solutions, specifically domain specific languages (DSLs) in subsection 2.3.1 and a modular framework for SMR protocols in subsection 2.3.2. In other words, this section we cover languages specifically built to be used in distributed protocol development.

2.3.1 Domain Specific Languages

DSLs have been used to specify, reason about, and verify distributed systems in a more systematic way. Many DSLs exist for that end, but, most notably, Lamport created the TLA+ [23] which is widely used for distributed systems verification. However, these verification DSLs do not provide us with a execution environment to actually put them in practice.

Work has been conducted towards creating transpilers to convert these formal languages to actual executable implementations [15], execution environments that execute a DSL on the fly [30], or even creating a DSL from the ground up specifically to create an executable. We will focus on a single DSL that was built from the ground up to build executable implementations, DistAlgo.

2.3.1.1 DistAlgo

DistAlgo [29] is a very-high level language developed at State University of New York. It provides many high-level control flows easily expressed through high-level queries. The code is transpiled and optimized to Python. The authors claim that distributed algorithms development has relied on hard to understand formal languages or imprecise pseudocode in English. Furthermore, they claim distributed systems development has relied on complex libraries, such as the ones seen in section 2.2.

DistAlgo adds four concepts to aid development: processes and sending of messages, control flows and handling of received messages, synchronization conditions using high-level queries, and configuration.

Processes are classes that extend a special class Process. They function as threads but cannot share memory and their only way of communication is through message passing. Messages are sent through a simple send statement.

Specific yield points may be added to specify where message handling can be done throughout the process lifecycle. Handling received messages needs a receive definition. For that definition, we need to provide which messages and from whom we want to handle as well as all the yield points where we can start handling the message. Additionally,

synchronization can be done with await statements and specifying a boolean expression, as well as a timeout.

More complex expressions can be done by using this language's high level queries. Often times, distributed algorithms will express complex conditionals for certain statements. In SMR algorithms, it is common to wait for the set of received messages to be at least a certain size. As such, DistAlgo provides quantifications (each, some), comprehensions and aggregates (min, max, sum, avg). The authors note that quantification programming can be hard, error-prone, and performance intensive. For that reason, they also provide all the conversions DistAlgo does from quantifications to common aggregate functions which they refer to as incrementalization.

Configurations use simple use statements to declare, for example, what kind of channel we want (reliable or unreliable). These configurations are then used by the compiler to generate the underlying socket code.

These high level constructs yield very small source code for common distributed algorithms. For instance, Paxos can be implemented in as little as 43 lines.

However, other common features we see in library based solutions are missing in DistAlgo and other solutions mentioned before. There is no support to composing these algorithms together. We would need to look at the transpiled code, if the DSL allows. Some DSLs do make an effort in adding signed messages [30], but further support for cryptography is missing. Additionally, no support for dynamic reconfiguration is supported in any of the DSLs. They provide very useful abstractions for specifying and verifying distribued systems overall, but don't provide necessary proporties for new more dynamic systems.

2.3.2 BFT-SMaRt

BFT-SMaRt [4] is a middleware developed in Java. It is designed with Byzantine Fault Tolerant (BFT) SMR in mind. It distinguishes itself from other similar works focused on SMR by having improved modularity, multicore-awareness and reconfiguration support. BFT-SMaRt achieves its modularity by having different modules containing the core functionality. There are three of these: communication system, SMR and state management.

The communication system is comprised of three main modules that deal with client communication and server communication. The client communication modules are implemented on top of the Netty framework while the server communication modules are implemented on top of normal Java threads and sockets. This system is charged with sending and receiving requests from and to clients in an open-group model, and sending and receiving requests from and to servers in a closed-group model; channel recovery after a failure, verifying client signatures and sequence numbers to avoid replay attacks.

The SMR system is comprised of six main modules. These modules are responsible for proposing and accepting values in each consensus instance, managing who is the leader, total order multicast and managing what replicas are in the system as well as the number

of tolerated faults. This system is built on top of the communication system.

The last system - state management - is implemented by the developer over a very simple API both for the server and client.

BFT-SMaRt incorporates throughout its entire design multithreading. Besides the Netty pool of threads to handle client communication, there are threads for the sockets for server communication, a message processor thread, and a proposer thread. Threads communicate through message queues akin to a consumer producer model. Over the years, BFT-SMaRt has matured and has seen a variety of adaptations to various scenarios. IBM has used BFT-SMaRt in Hyperledger Fabric to make a distributed ledger platform [1]. However, the core stays the same and is mainly for BFT SMR.

2.3.3 Summary

In this section, we discussed non library based solutions that take a fundamental different approach that we will take in our proposed solution. We discussed DSLs, specifically DistAlgo, and how they provide more abstractions to deliver very small implementations for known distributed protocols. However, they lack features present in library based solutions like better protocol composition. We also discussed BTF-SMaRt in the context of its architectural design. It features a robust multi threaded architecture with a tight integration with Netty.

A new library based framework can close the gap in small implementation footprints compared to DSLs and integrate a better architectural design, like BFT-SMaRt, that doesn't compromise on flexibility.

2.4 Discussion

In section 2.1, we presented some relevant technologies that are used extensively in some of the solutions. While a direct implementation of QUIC will be necessary to any new solution, some thought must be put into what form the NAT traversal will take. Netty is the industry standard of network communication. However, an abstraction layer on top of Netty needs to be implemented to support dynamic reconfiguration - e.g. changing network interfaces for communication with another node during runtime. DistAlgo, described in subsection 2.3.1, abstracts away the network communication details by just asking from the developer what guarantees they want for the connection between nodes.

Execution environments saw little to no regard in most of the earlier solutions. Modern ones see real world implementations in different environments other than desktop and server hardware with limited reach. Two chat apps have been developed, namely Berty [3] in libp2p and Delta Chat [7] in Iroh. We want to make the transition between platforms seamless.

Modularity is a common aspect throughout all the solutions in section 2.2. Particularly, some of the earliest work presented - like x-kernel and Horus - a broad enough

interface to compose protocols in a stack in any way one wishes, even if the frameworks don't make any guarantees that the stack will work. This lack of guarantees may lead to errors later on that we want to avoid. In subsection 2.3.2, we presented the BFT-SMaRt's modular design, which doesn't compromise the execution guarantees. However, it is limited to SMR protocols. Babel, presented in subsection 2.2.5, also has a modular design but the developer needs to know too much to direct the request or notification to the correct implementation. There's a clear void in a correct, easy-to-use modular design.

2.5 Summary

In this chapter, we explored relevant technologies for the development of a general distributed protcol framework and the state of the art in that regard. We presented the foundational works in library based solutions, like x-kernel, and modern P2P solutions, like libp2p. Classic frameworks focus on static clusters of nodes while modern solutions are adaptaded for global P2P solutions. There is a void in the middle for swarms of mobile devices.

3

Work Plan

In this chapter, we will discuss our proposed solution and how we plan to experimentally evaluate it. In section 3.1, we propose a new framework based on Babel with new features to overcome the identified gaps in the previous chapter. In section 3.2, we present the metrics we will use to evaluate how the new framework improves and overcomes the challenges presented. In section 3.3, we present the schedule that will guide us to accomplish the goals we set forth.

3.1 Proposed Solution

As proposed in chapter 1, our goal is to build a new framework based on Babel (described and discussed in subsection 2.2.5). Previously, in section 2.1, we presented relevant technologies that must integrate a new solution. Alongside these technologies, the abstractions we build upon them must be general enough to accommodate a wide range of protocols in distributed systems exposing programming interfaces that avoid to overwhelm the developer.

Modular design is common in all the solutions covered in section 2.2. Some take a liberal approach while not providing many guarantees as to what will happen when things are composed like in x-kernel and Horus (described in subsection 2.2.1 and subsection 2.2.3, respectively). Modern P2P approaches such as libp2p and Iroh (described in subsection 2.2.6 and subsection 2.2.7, respectively) provide plenty of pre-definied parts for various features that can be composed. However, developing your own protocols is not encouraged or is only left as a footnote not being a key feature supported by these solutions. Developers can also spend too much time setting up the necessary services, as stated by Guidi et al [14]. A new framework must find a sweet spot of modular design with enough guard rails, pre-packaged parts and setup lines for rapid development without sacrificing flexibility.

Finally, we must take into account the clear gap in heterogeneous distributed systems. The new solution must take into account a variety of execution environments - static cluster of nodes, a global P2P network, and a leaderless swarm system. This versatility ensures

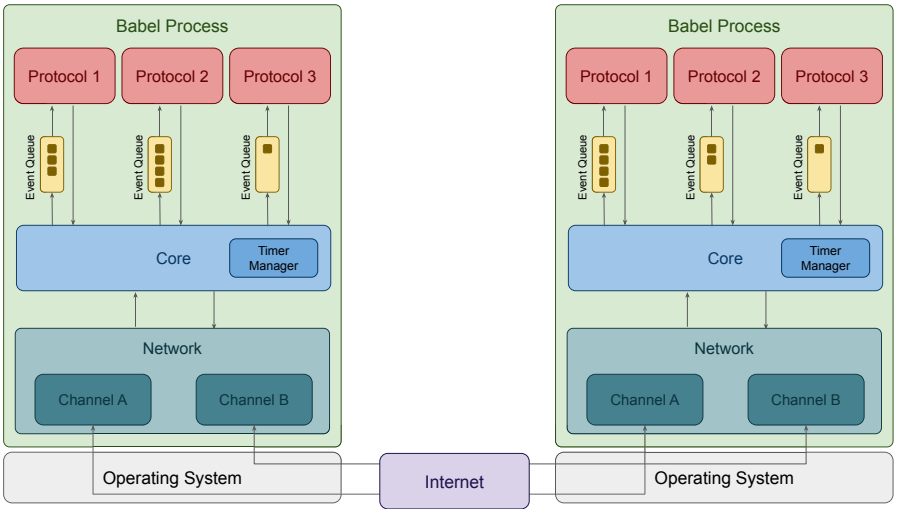

Figure 3.1: Babel Architecture. Figure extracted from [11].

that the framework remains resilient and performant across a wide spectrum of devices in modern distributed systems.

In the following sections, we will look at each component of the Babel Runtime, as seen in Figure 3.1 (Network, Core, and Protocols), and detail our proposed solution.

3.1.1 Network

Firstly, we will redesign the network layer.

As noted in subsection 2.1.4, we will need to manage identities for cryptography. We will use an idea similiar to libp2p and use an ID derived from the public key as the primary means to identify nodes in the system. This will allow to decouple node identifiers from their network address as is the current practice in the Babel framework. Here, in the nework layer, we will manage all known identities and all the contactable addresses. This move will make a new framework more secure avoiding eavesdropping attacks.

Additionally, this approach will make NAT traversal easier since, as we noted in subsection 2.1.2, one node might present itself with multiple addresses. NAT traversal will be implemented with a combination of STUN, TURN and ICE. A signaling protocol will be done in a server alongside a relay. Firstly, this server will serve as an overall ICE agreement server, supporting STUN without a relay server after the connection is established. Later on, any known public facing node connected with Babel could serve as an ICE agreement server. Node IDs can be resolved either through multicast (already present in Babel Swarm) or through the signaling server. We will later assess whether this addressing scheme is sufficient or if a more sophisticated peer discovery scheme will be employed.

Connectivity in the devices might change at runtime and, therefore, change the network interface used to interact with the system. We will implement the dynamic network interface changes in the network layer. In current Babel versions, there's no recovery in case a network interface fails or loses connections. Other frameworks suffer from this issue

too, specifically older ones, but new ones still require signficant effort from the developer to deal with such scenario. Ideally, the network interfaces could fail or lose connectivity in a transient way, but recover automatically and resume operation as normal. We will integrate QUIC to leverage Connection IDs for interface migration. On a more technical side, and as we stated from the beginning, we also want to unify all known Java Babel versions. Besides Babel Swarm - running on Java 21 - there is a Babel Android that was backported to Java 17 to support the Android Runtime. Its main difference is how it interacts with Android's network stack. If we are to keep Android support, we will keep the framework based on Java 17.

As detailed in section 2.2, many solutions provide a more modern and robust communication solution while Babel falls short by only providing communication channels in TCP. A modern solution must provide abstractions to easily communicate using a variety of transport protocols with various security guarantees. In Babel Swarm, new abstractions to use local network multicast were done using classic Java sockets inside the protocol logic due to the shortcomings of the current network abstraction. This of course, goes against the principle of separating network logic and protocol logic. As a first step, we will only add support for IP and all its transport protocols (TCP, UDP, and QUIC), and later add support for Bluetooth for smaller devices. The API will support stream-based protocols (TCP, QUIC) and datagram-based ones (UDP/Bluetooth).

Additionally, we want to remove most of the burden for setting up the communication layer. In all the Babel versions, one must set up the channels they want to use. Setting up these channels isn't so different from using the classic Java sockets. The only added feature on top of sockets is asynchronous I/O from Netty. Ideally, the developer would just use a `sendMessage` primitve with the a node ID to abstract away network addressing. In the new version, this node ID will be, in a first phase, a public key. This public key will be solved in a simple manner using a combination of network level multicast and DNS.

In the new version, developers will be able to specify the communication channel characteristics they want to have in their protocols, akin to what DistAlgo did, as described in subsection 2.3.1 - e.g. send and receive messages in FIFO order, reliable or unreliable delivery, stream-based or message-based. When setting up the protocol, the developer would could choose a combination of the guarantees for the communication channels the protocol needs.

3.1.2 Core

This is where most of our work for modularity will be implemented. In the original Babel paper [11], one of the future work goals was to move the actual implementation closer to the pseudocode and descriptions present in the protocol presentation. However, as discussed in section 2.3, Babel clearly does not excel in this department with DSLs easily implementing Paxos in very few lines of code compared to what Babel did. New abstractions will need to be implemented here to move the new Babel closer to the pseudocode.

We won't be able to directly compare to DSLs since we are still constrained to what Java provides us. However, we can still abstract away some of the more common features present in widely used in distributed algorithms and protocols. One of our proposals is to make a system to register protocols into known families to create ease to use primitives for other protocols. Examples include:

- A broadcast protocol must have a simple broadcast primitive to announce something to the entire network. This should be effective across multiple membership solutions.
- A consensus protocol can have a simple propose primitive to propose a value and an event to announce that a value has been decided. This should be effective regardless of the SMR protocol used on top.
- A network membership protocol can have primitives to enter and exit the group network and have a list of known neighbours.

This main idea to be employed stems directly from the system implemented in Horus. In it, protocols can be classified into classes that can be composed, as described in subsection 2.2.3. Of course, this new system would be more powerful since this would permit stacking protocols in any way the developer wants - e.g. pick any broadcast protocol for the consensus protocol- and avoid the problems that plague the "stack in any way you want" frameworks, like Horus itself. This would be implemented by creating interfaces and then letting developers register their protocols in the Babel Runtime. Two approaches could then be done: have method stubs for all the common methods or have the registry available to all the protocols to choose what suits their needs best. Such feature would move Babel closer to what the DSLs do while still keeping the advantages of being built on top of a general purpose language - developers don't need to learn a language from scratch and we have access to a better library support.

As stated in the previous section, there it exists a Babel Android version to accommodate some of the peculiarities of the Android Runtime. It still lacks on the lifecycle concerns discussed in chapter 1. Support for a more robust protocol lifecycle management must be added while keeping it simple to not over burden the developer. Ideally, Babel would react to events from the operating system to suspend the process gracefully, picking it back up once the operating system sends the event, without much developer effort. For the protocol interface, we will add two event handlers for pausing and resuming execution. Specific adapters for each supported OS will be implemented to handle the different signals available in each OS. All the adapters will share the same outwards facing API. This of course, must not require a significant effort from the developer.

3.1.3 Protocols

Due to many of the proposed evolutions described above, protocols will change their interaction with the framework as a whole. We will now require developers to categorize the

protocols they develop to fit in the new families we propose. Of course, if one developer wishes to add a protocol to the broadcast family, it must implement the broadcast interface and provide the necessary implementations for the stubs. Additionally, we would also like to remove the application logic from the Babel protocols. Right now, for the application logic - e.g. a chat app - there must be a protocol with only the application logic that only interacts with other protocols that actually do network communications. Ideally, the application logic would only receive a known set of events from the Babel runtime and interact with it through a known set of events, without ever being part of the protocols registerd in Babel.

3.2 Experimental Evaluation

In this section, we explore how we plan to evaluate our solution.

With the new framework we must effectively validate and measure the cost of all the new features as well as understand how they impact the overall development of the protocols, alongisde the new environments they can now execute on. We will work on implementing two types of protocols on this new framework: an SMR protocol and a membership/gossip protocol. Fault tolerant distributed systems are often implemented using an SMR protocol and a membership/gossip protocol is a great way to test swarm systems to the churn[1] they can withstand, hence our choices. Both of these need to be tested in new platform environments and network environments. All these implementations will be adapted into 3 platforms: a standard Linux desktop environment, Android and a Raspberry Pi Zero. On top of the SMR protocol and membership protocol, applications will be built to take advantage of the protocols and the new architectural changes. A distributed replicated key store will be implemented on top of the SMR protocol. A decentalized chat app will be implemented on top of the membership/gossip protocol.

Lines of code We will compare implementations of the protocols we mentioned before in Babel and compare them to the new version. Additionally, we will compare speciallized implementations. We will look at the lines of code needed to implement the protocols in these 3 scenarios. We expect to achieve a smaller implementation compared to the original Babel.

Network performance and resource usage We will compare various network metrics in the various implementations. We can easily measure throughput and latency in these solutions. Additionally, we must also measure the added overhead for cryptography in QUIC. Resource usage in the nodes must also be measured in all the platforms. We expect to achieve better result in all these metrics, even with the added cryptography overhead.

[1] Rate of entry and exit of nodes from the distributed system

New platform and network environments Alongside the network performance, we will measure how well our new version behaves under new network scencarios. We will assess the success rate of NAT traversal of different types of NAT and firewalls, discussed in section 2.1, and how this impacts overall network performance with the metrics stated before. If we are able to establish a direct connection, we expect no change in overall performance. However, if a relay server is used, we expect the overall performance to decrease. We will try to dimish the added overhead of usign a relay server as much as possible. Furthermore, we will measure the connectivity handover changes (e.g. WiFi to 4G), specifically connection handover latency. The implementations will be the same across all platforms.

3.3 Schedule

In this section we present the schedule for the tasks planned. We separated the tasks into 5 main groups:

Design the new network layer In this phase, we will first implement QUIC as this will be the basis for other important tasks. After this, work can be started on porting implementation done in Babel for an SMR protocol and Gossip protocols to the new version. After QUIC is implemented, we move on to NAT traversal and testing the success rate of traversing different types of NATs. We finish this group with the dynamic interface migration.

Design the new modular core In this phase, we will implement the protocol categorization system, and dyanmic lifecycle suupport for different OSes. We will continuosuly integrate these features in our porting efforts.

Protocol porting We finish the final details to port the implementations to the new versions. We don't expect the final touches to take much time since this will be an ongoing effort. The supporting applications will be implemented on top of the ported protocols. We also remove the application logic from the Babel lifecycle, implementing an event system to communicate with the application layer.

Experimental Evaluation In this phase, we will test these protocols in a Linux, Android, and Raspberry Pi Zero platforms. In each, we will gather information for overall performance and resource usage. Compare it to the original Babel implementations. In the Android and Raspberry Pi platforms, we will also measure handover latency for the interface migration. This will be done after the new network layer is finished. We will compare our implementations code complexity to the original Babel versions after the new modular core is finised.

Thesis Writing Finally, we gather our findings and write the thesis and a paper for the Inforum conference.

The planned tasks schedule can be seen in a Gantt Chart in Figure 3.2.

3.4 Summary

In this chapter we discussed the new framework to replace all the existing Babel versions. We discussed its new features to fill the void identifed in the current state of the art. We will validate our solution according to the presented experimental evaluation plan. We conclude with the schedule that will serve as a guideline for the remainder of the work.

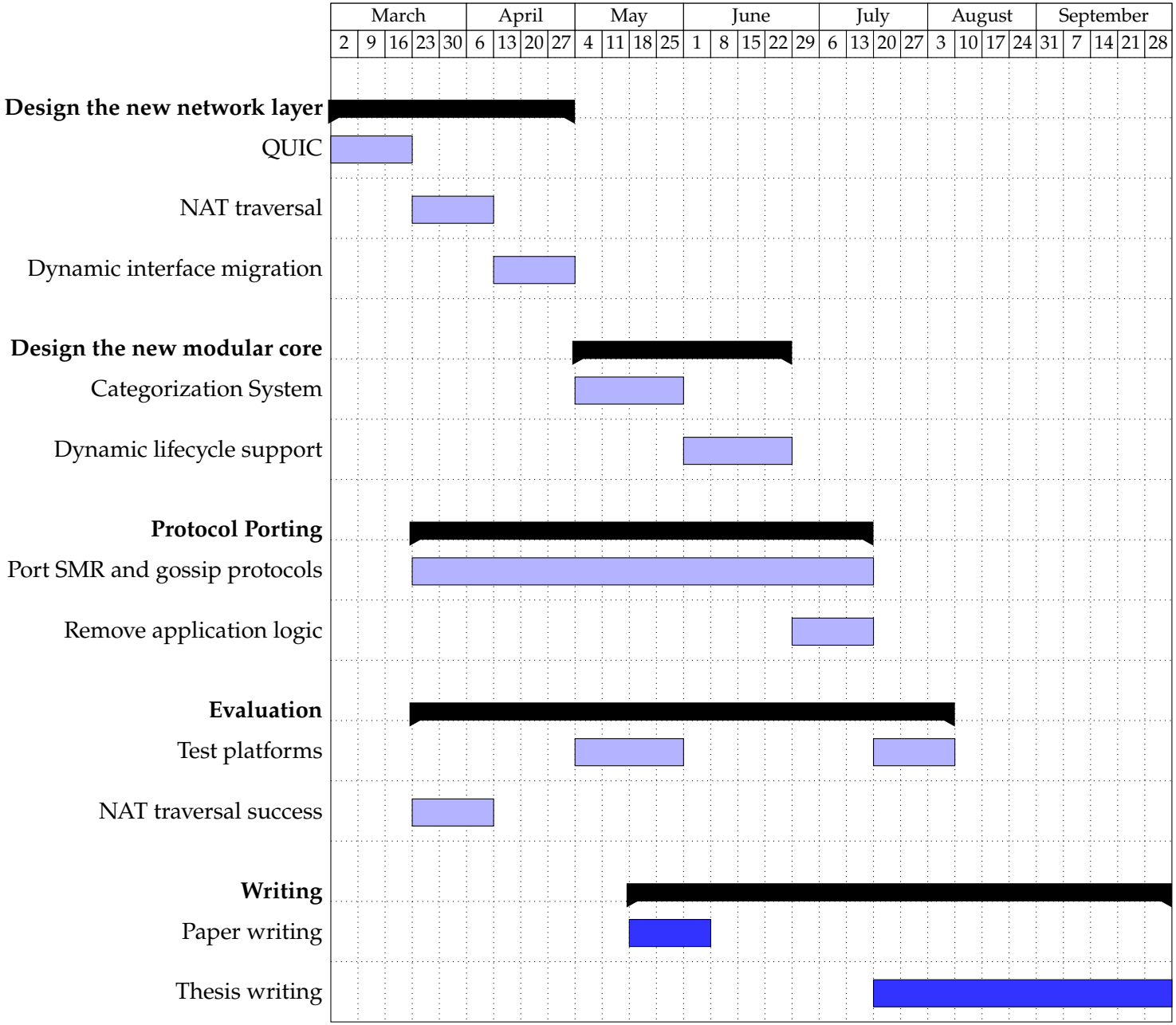

Figure 3.2: Thesis Schedule (March 2026 – September 2026)

Bibliography

[1] A. Barger et al. "A Byzantine Fault-Tolerant Consensus Library for Hyperledger Fabric". In: *2021 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 2021, pp. 1–9. DOI: 10.1109/ICBC51069.2021.9461099 (cit. on p. 18).

[2] J. Benet. "IPFS - Content Addressed, Versioned, P2P File System". In: *arXiv* (2014-07). DOI: 10.48550/arXiv.1407.3561 (cit. on p. 14).

[3] Berty Technologies. *Berty Technologies*. [Online; accessed 19. Jan. 2026]. 2026-01. URL: https://berty.tech (cit. on p. 18).

[4] A. Bessani, J. Sousa, and E. E. P. Alchieri. "State Machine Replication for the Masses with BFT-SMART". In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, pp. 23–26. ISBN: 978-1-4799-2233-8. DOI: 10.1109/DSN.2014.43 (cit. on p. 17).

[5] T. D. Chandra, R. Griesemer, and J. Redstone. "Paxos made live: an engineering perspective". In: *ACM Conferences*. 2007-08, pp. 398–407. DOI: 10.1145/1281100.1281103 (cit. on p. 1).

[6] *Cluster Specification • Akka core*. [Online; accessed 27. Jan. 2026]. 2026-01. URL: https://doc.akka.io/libraries/akka-core/current/typed/cluster-concepts.html#gossip-convergence (cit. on p. 15).

[7] *Delta Chat: Delta Chat, decentralized secure messenger*. [Online; accessed 19. Jan. 2026]. 2026-01. URL: https://delta.chat/pt (cit. on p. 18).

[8] K. B. Egevang and P. Srisuresh. *Traditional IP Network Address Translator (Traditional NAT)*. RFC 3022. 2001-01. DOI: 10.17487/RFC3022 (cit. on p. 5).

[9] *Enterprise Agentic AI*. [Online; accessed 19. Jan. 2026]. 2026-01. URL: https://akka.io (cit. on p. 15).

[10] *Examining HTTP/3 usage one year on*. [Online; accessed 15. Jan. 2026]. 2025-10. URL: https://blog.cloudflare.com/http3-usage-one-year-on (cit. on p. 4).

[11] P. Fouto et al. "Babel: A Framework for Developing Performant and Dependable Distributed Protocols". In: *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. IEEE, pp. 19–22. DOI: 10.1109/SRDS55811.2022.00022 (cit. on pp. 2, 12, 21, 22).

[12] *Free Pool of IPv4 Address Space Depleted | The Number Resource Organization*. [Online; accessed 15. Jan. 2026]. 2026-01. URL: https://www.nro.net/ipv4-free-pool-depleted (cit. on p. 5).

[13] T. Galvão et al. "Suporte ao Desenvolvimento de Novas Aplicações Descentralizadas". In: () (cit. on pp. 2, 13).

[14] B. Guidi, A. Michienzi, and L. Ricci. "A libP2P Implementation of the Bitcoin Block Exchange Protocol". In: *Proceedings of the 2nd International Workshop on Distributed Infrastructure for Common Good*. DICG'21. Virtual Event, Canada: Association for Computing Machinery, 2021, pp. 1–4. ISBN: 9781450391696. DOI: 10.1145/3493426.3493822 (cit. on pp. 14, 20).

[15] F. Hackett et al. "Compiling Distributed System Models with PGo". In: *Compiling Distributed System Models with PGo [evaluation]*. 2023-01, pp. 159–175. DOI: 10.1145/3575693.3575695 (cit. on p. 16).

[16] C. Hewitt, P. Bishop, and R. Steiger. "Session 8 formalisms for artificial intelligence a universal modular actor formalism for artificial intelligence". In: *Advance papers of the conference*. Vol. 3. Stanford Research Institute Menlo Park, CA. 1973, p. 235 (cit. on p. 15).

[17] M. A. Hiltunen et al. "Survivability through customization and adaptability: the Cactus approach". In: *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*. IEEE, pp. 25–27. ISBN: 978-0-7695-0490. DOI: 10.1109/DISCEX.2000.825033 (cit. on p. 10).

[18] N. C. Hutchinson and L. L. Peterson. "The x-Kernel: An Architecture for Implementing Network Protocols". English. In: *IEEE Transactions on Software Engineering* 17.1 (1991-01). Copyright - Copyright Institute of Electrical and Electronics Engineers, Inc. (IEEE) Jan 1991; Última atualização em - 2024-12-02; CODEN - IESEDJ, pp. 64–76. URL: https://www.proquest.com/scholarly-journals/x-kernel-architecture-implementing-network/docview/195592126/se-2 (cit. on p. 9).

[19] *Introduction - iroh*. [Online; accessed 19. Jan. 2026]. 2026-01. URL: https://docs.iroh.computer (cit. on p. 14).

[20] *IPv6 – Google*. [Online; accessed 15. Jan. 2026]. 2024-09. URL: https://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption (cit. on p. 5).

[21] C. F. Jennings and F. Audet. *Network Address Translation (NAT) Behavioral Requirements for Unicast UDP*. RFC 4787. 2007-01. DOI: 10.17487/RFC4787 (cit. on p. 5).

[22] A. Keränen, C. Holmberg, and J. Rosenberg. *Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal*. RFC 8445. 2018-07. DOI: 10.17487/RFC8445 (cit. on p. 7).

[23] L. Lamport. *Specifying systems*. Vol. 388. Addison-Wesley Boston, 2002 (cit. on p. 16).

[24] L. Lamport. "The part-time parliament". In: *ACM Trans. Comput. Syst.* 16.2 (1998-05), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229 (cit. on pp. 1, 13).

[25] L. Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Commun. ACM* 21.7 (1978-07), pp. 558–565. ISSN: 0001-0782. DOI: 10.1145/359545.359563 (cit. on p. 1).

[26] A. Langley et al. "The QUIC Transport Protocol: Design and Internet-Scale Deployment". In: *ACM Conferences*. New York, NY, USA: Association for Computing Machinery, 2017-08, pp. 183–196. DOI: 10.1145/3098822.3098842 (cit. on p. 4).

[27] Z. Li et al. "Authentication in Peer-to-Peer Network: Survey and Research Directions". In: *2009 Third International Conference on Network and System Security*. IEEE, pp. 19–21. DOI: 10.1109/NSS.2009.30 (cit. on p. 8).

[28] *libp2p - libp2p Documentation Portal*. [Online; accessed 19. Jan. 2026]. 2026-01. URL: https://docs.libp2p.io (cit. on p. 14).

[29] Y. A. Liu et al. "From clarity to efficiency for distributed algorithms". In: *SIGPLAN Not.* 47.10 (2012-10), pp. 395–410. ISSN: 0362-1340. DOI: 10.1145/2398857.2384645 (cit. on p. 16).

[30] N. C. P. Lopes. "Practical Executable Specifications for Distributed Systems". In: (2009) (cit. on pp. 16, 17).

[31] J. M. Lourenço. *The `aidisclose` package: Generative AI disclosure checklist and statements*. Version 1.11.0. 2025. URL: https://ctan.org/pkg/aidisclose/aidisclose-doc.pdf (cit. on p. v).

[32] R. Mahaveerakannan et al. "An IoT based forest fire detection system using integration of cat swarm with LSTM model". In: *Comput. Commun.* 211 (2023-11), pp. 37–45. ISSN: 0140-3664. DOI: 10.1016/j.comcom.2023.08.020 (cit. on p. 1).

[33] C. Mazzocca et al. "A Survey on Decentralized Identifiers and Verifiable Credentials". In: *IEEE Commun. Surv. Tutorials* 27.6 (2025-02), pp. 3641–3671. DOI: 10.1109/COMST.2025.3543197 (cit. on p. 8).

[34] S. Mena et al. "Appia vs. Cactus: comparing protocol composition frameworks". In: *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.* IEEE, pp. 06–08. ISBN: 978-0-7695-1955. DOI: 10.1109/RELDIS.2003.1238068 (cit. on pp. 2, 11, 12).

[35] 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*. IEEE, pp. 16–19. ISBN: 978-0-7695-1077. DOI: 10.1109/ICDSC.2001.919005 (cit. on p. 12).

[36] *Netty: Home*. [Online; accessed 20. Jan. 2026]. 2025-12. URL: https://netty.io (cit. on p. 8).

[37] M. Petit-Huguenin et al. *Session Traversal Utilities for NAT (STUN)*. RFC 8489. 2020-02. DOI: 10.17487/RFC8489 (cit. on p. 6).

[38] T. Reddy.K et al. *Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN)*. RFC 8656. 2020-02. DOI: 10.17487/RFC8656 (cit. on p. 6).

[39] R. Roestenburg, R. Williams, and R. Bakker. *Akka in action*. Simon and Schuster, 2016 (cit. on p. 15).

[40] R. E. Schantz and D. C. Schmidt. *Middleware for Distributed Systems*. 2008 (cit. on p. 2).

[41] B. Sredojev, D. Samardzija, and D. Posarac. "WebRTC technology overview and signaling solution design and implementation". In: *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. IEEE, pp. 25–29. DOI: 10.1109/MIPRO.2015.7160422 (cit. on p. 7).

[42] Y. Suchikova et al. "GAIDeT (Generative AI Delegation Taxonomy): A Taxonomy for Humans to Delegate Tasks to Generative Artificial Intelligence in Scientific Research and Publishing". In: *Accountability in Research* (2025), pp. 1–27. DOI: 10.1080/08989621.2025.2544331 (cit. on p. v).

[43] *TaRDIS project*. [Online; accessed 10. Feb. 2026]. 2025-11. URL: https://project-tardis.eu (cit. on p. 13).

[44] The LaTeX Project team. *LaTeX - A document preparation system*. 1985. URL: https://www.latex-project.org (visited on 2026-01-14) (cit. on p. v).

[45] I. Vaccari et al. "Perpetrate cyber-attacks using IoT devices as attack vector: The ESP8266 use case". In: *CEUR Workshop Proceedings*. Vol. 2940. 2021, pp. 35–46 (cit. on p. 1).

[46] R. Van Renesse and K. P. Birman. *Protocol composition in Horus*. Tech. rep. Cornell University, 1995 (cit. on p. 11).

[47] M. Van Steen and A. S. Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017 (cit. on p. 1).

[48] D. Wing. "Network Address Translation: Extending the Internet Address Space". In: *IEEE Internet Comput.* 14.4 (2010-06), pp. 66–70. DOI: 10.1109/MIC.2010.96 (cit. on p. 5).