NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

DANIEL DE ALMEIDA JOÃO

BSc Computer Engineering

NETWORK ABSTRACTIONS AND EMULATION FOR DISTRIBUTED SYSTEMS

Dissertation Plan
MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
February, 2023

NETWORK ABSTRACTIONS AND EMULATION FOR DISTRIBUTED SYSTEMS

DANIEL DE ALMEIDA JOÃO

BSc Computer Engineering

Adviser: João Carles Antunes Leitão
Assitent Professor, NOVA University of Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
February, 2023

Abstract

Babel is a Java framework which simplifies the development of distributed applications. Babel offers a network abstraction called channels, which allow its protocols to send and receive messages over the network. However, current implementations of the channels are not very flexible, as they only support TCP-based type of communication, and do not allow Babel to be integrated with any other external systems, such as Kafka, RabbitMQ, ZooKeeper, etc.

Nowadays, it is becoming extremely important to have frameworks that offer a wide range of transport protocols, and can easily be integrated with other frameworks, or distributed services. This feature of extensibility is very important in various distinct areas, such as education, health care systems, telecommunications, military, and others, as they require different applications to exchange data. This feature allows for developers to easily integrate new aspects to their systems, in a rapid way, without having to deal with the hassle of implementing them from scratch, or altering the design of the existing system. Moreover, it is an important feature to achieve because it reduces the costs of software development as it requires fewer resources and maintenance; it improves quality of data as more sources can be grouped together; it increases efficiency, as it minimizes the time needed to process data. However, implementing and designing flexible frameworks have some challenges that need to be resolved, such as data privacy, security, and different data representation different systems might have.

In this work, we intend to explore technologies with features supported by Babel, as well as technologies with features Babel does not support. We explore the concepts of TCP, UDP, and QUIC; as well as distributed libraries and applications. In the end, we present a library that will offer an API to extend the networking channels of Babel; the API will allow Babel to support transport protocols such as UDP and QUIC; and will also allow Babel to be integrated with systems such as Kafka, RabbitMQ, and ZooKeeper.

Keywords: distributed services, network abstraction, transport protocols, distributed applications, communication libraries

Resumo

Babel é uma framework de Java que simplifica o desenvolvimento de aplicações distribuídas. Babel oferece uma abstração de rede chamada *channels*, que permite que os seus protocolos enviem e recebam mensagens pela rede. No entanto, as implementações atuais dos canais não são muito flexíveis, pois suportam apenas o tipo de comunicação baseada em TCP e não permitem que o Babel seja integrado a nenhum outro sistema externo, como o Kafka, o RabbitMQ, o ZooKeeper, etc.

Hoje em dia, torna-se extremamente importante ter frameworks que ofereçam uma ampla gama de protocolos de transporte e possam ser facilmente integrados a outras frameworks ou serviços distribuídos. Esta característica de extensibilidade é muito importante em várias áreas distintas, como educação, sistemas de saúde, telecomunicações, militares, entre outros, pois requerem diferentes aplicações para troca de dados. Esse recurso permite que os desenvolvedores integrem facilmente novos aspetos aos seus sistemas, de forma rápida, sem ter que lidar com o trabalho de implementá-los do zero ou alterar o design do sistema existente. Além disso, é um recurso importante a ser alcançado porque reduz os custos de desenvolvimento de software, pois requer menos recursos e manutenção; melhora a qualidade dos dados, pois mais fontes podem ser agrupadas; aumenta a eficiência, pois minimiza o tempo necessário para processar os dados. No entanto, implementar e projetar estruturas flexíveis tem alguns desafios que têm de ser resolvidos, como privacidade de dados, segurança e diferentes representações de dados que diferentes sistemas podem ter.

Neste trabalho pretendemos explorar tecnologias com funcionalidades suportadas pelo Babel, bem como tecnologias com funcionalidades que o Babel não suporta. Exploramos os conceitos de TCP, UDP e QUIC; bem como bibliotecas e aplicações distribuídas. No final, apresentamos uma biblioteca que oferecerá uma API para ampliar os canais de redes do Babel; a API permitirá que o Babel suporte protocolos de transporte como UDP e QUIC; e também permitirá que o Babel seja integrado a sistemas como Kafka, RabbitMQ e ZooKeeper.

Palavras-chave: serviços distribuídas, abstração de redes, protocolos de transporte, aplicações distribuídas, bibliotecas de comunicação

Contents

List of Figures

1 Introduction

1.1 Context

As the Internet continues to grow and more data is produced, it is becoming increasingly relevant to enable developers to create high-performance distributed applications within a short period of time; and these applications must also be able to interact with other systems [113, 96]. The integration of software applications in our daily lives has accelerated the pace at which we perform our tasks, so it is very crucial that these applications take less development time. This has a significant impact for researchers that want to build prototypes for conducting experimental evaluation; practitioners that want to compare different design alternatives or solutions; and even for practical teaching activities on distributed algorithms courses [96].

One thing developers can do to reduce the time it takes to develop distributed applications is to focus less on time-consuming, networking aspects by relying on available, efficient, and trustable network abstractions and programming frameworks that abstract these common aspects. This way, they can devote all of their attention to data processing tasks or distributed protocol logics. However, networking programming is not as simple as using network abstractions; we need to think about a set of rules that must be followed when data is transmitted, the heterogeneity of the devices in the network, security constraints, the rise of network partitions, network configurations, data integrity, packet loss, and many other aspects [6].

Babel [79, 80] is a Java framework developed at NOVA with the aim of simplifying the development of distributed protocols and systems by providing abstractions that handle repetitive and time-consuming aspects of development, namely network connections. However, the current version of Babel offers only simple network abstractions, such as TCP connections through an abstraction named channels, which is limitative in some cases. Moreover, Babel has no channel supports to allow it to interact with distributed services such as Apache Kafka [65], RabbitMQ [74], Apache ZooKeep [64], and other distributed messaging services.

In this work, we aim to address network abstractions available to developers in the

Babel[79] Framework, and in other frameworks and distributed services; we intend to study other networking features not supported in Babel, with the goal of developing a solution to integrate them in Babel. We will revisit the channel abstraction of Babel[79] and enrich its transport protocols (which offers TCP) to include support for other protocols, such as UDP [108] and QUIC [91]. These are some of the most used protocols in network abstractions due to their properties, such as: reliability, integrity, performance, flexibility and evolvability. These supports will be provided by extending the channel abstraction, and consequently, they will allow several use-cases implementations, making Babel more flexible, and able to provide different guarantees [91, 108, 107].

1.2 Objective

With this work, we intend to study and evaluate currently available network abstractions, transport protocols and communication services, their implementation details, and their uses in order to be able to develop new network abstractions that will extend the Babel framework. We aim to understand how these existing networking abstractions handle network communications, in order to gain an insight on what to do to allow Babel to support a wider range of distributed protocols and execution environments.

1.3 Contributions

The main expected contribution of this work include:

- Provide a framework that will allow Babel [79] to support new network abstractions, a wider range of distributed protocols and execution environments;
- Future developers wanting to add more features to Babel, and integration with different systems, will guide themselves on the concepts presented in this work;

1.4 Document Structure

The remainder of the document is structured as follows:

- Chapter 2 starts by defining the concept of network abstraction in the context of this dissertation, and continue by discussing available network abstractions. It discusses transport protocols, communication libraries, distributed frameworks, networking services, and distributed applications.
- Chapter 3 presents a proposed solution of a network abstraction that will provide additional flexibility to Babel[79, 80]. This chapter also discusses the main challenges, and how these will be tackled, as well as different tests to be conducted throughout the work. Finally, it presents a proposal for scheduling of the activities.

2 Related Work

The goal of this chapter is to highlight some technologies that share common features with Babel [79, 80] as well as some aspects that Babel does not support, and would be of our interest to have Babel supporting them. Babel is a novel framework, created by NOVA LINCS [98] at NOVA FCT [97], to develop, implement, and execute distributed protocols and systems. It employs a network abstraction called channels; they abstract all the complexity of dealing with networking and allow for network communications between processes [79].

In this chapter, we start by briefly defining network abstractions in Section 2.1 and talking about their relevance as well as their challenges. Next, we briefly introduce the OSI Model in Section 2.2, a model that describes how data moves from an application in one computer to an application in another computer [23]; we'll delve into the transport layer protocols in Section 2.3, which are the basis for implementing network abstractions; messaging patterns in Section 2.4, they describe the ways applications communicate with each other within the network [55]; communication libraries, Section 2.5, offer a set of methods and functions to implement communication abstractions for the network; we'll talk about some frameworks, Section 2.6, which provide structures for building distributed applications; networking services, Section 2.7, that tend to offer more complex and high-level abstractions used by other applications; distributed applications, Section 2.8, applications that run on multiple machines and share messages over the network to achieve goals such as fault-tolerance and scalability [25, 99].

2.1 Network Abstractions

The main purpose of a network abstraction is to hide the complex mechanisms and details of the network [114], by offering additional flexibility [81] and a very generalized interface through which the developer delegates the task of exchanging data between peers. This gives the developer more time to focus on other pertinent details. For example, if a developer needs to send data from computer A to computer B, they will not need to fully understand the network stack, nor the full operation of the Internet to be able to send and receive data over the network; depending on their needs, they can simply use the most convenient existing network abstractions to complete the work.

However, implementing a network abstraction is not a simple task. Because the network as we know it today is constantly changing and expanding very rapidly [105], with a vast variety of devices joining the internet every single day, we need to implement abstractions that will take this challenge into consideration. Moreover, the difficulty is also exacerbated when thinking about other sets of guarantees a network abstraction must provide, namely privacy, security, data integrity, reliability, interoperability, recovery, and others [5].

2.2 The OSI Model

The Open Systems Interconnection (OSI) model is a conceptual model that provides a standard for different computer systems to be able to communicate with each other. It can be seen as a universal language for computer networking; it splits up a communication system into seven abstract layers: the application layer, presentation layer, session layer, transport layer, network layer, datalink layer, and physical layer [89].

Application Layer This is the layer that supports end-user software applications and processes; this is where application processes access the network services; web browsers and email clients rely on this layer for communication [89, 23].

Presentation Layer This layer formats the data from the sending layer (application or session layer) to be understood by the next layer [89, 23].

Session Layer It is responsible for establishing connection between the two interested parties [89, 23].

Transport Layer It is the layer that provides the rules for exchanging variable-length data sequences from a source to a destination host across a network. Furthermore, it is where the negotiation of quality and type of services takes place, the user and the transport protocols negotiate the quality of service to be provided; it also guarantees services, as it may provide reliable service over an unreliable network layer; this is also where we find TCP, UDP and QUIC [89, 23, 107, 108, 91]. For the purpose of this document, we are mostly interested in the type of abstraction offered by this layer.

Network Layer This layer breaks up the segments into smaller units called packets on the sending side and reassembles them on the receiving side. It is also responsible for addressing and for finding the best physical path for the packets [89, 23].

Data Link Layer Performs error detection and control on the data [89, 23].

Physical Layer This layer is made of the physical equipments (cables and switches) responsible for transferring the data to another device. This is where data is converted into zeros and ones, and where the involved parties negotiate on the type of signal to codify these bits.

The following subsections provide overviews of the most well-known transport protocols, TCP [107], UDP [108], and QUIC [117]. However, we also consider to be important to give an overview about sockets and networking medium, as they also play a key role in the realm of protocols.

2.3 Transport Protocols

These are the Transport Layer protocols that provide procedural methods on how to exchange data over the network [57, 30]. These protocols are differentiated by the features they provide, such as reliable delivery, ordered delivery, content privacy, and integrity protection [30]. Moreover, there are a variety of protocols [30, 58], which include Transmission Control Protocol (TCP), User Datagram Protocol (UDP), QUIC, Stream Control Transmission Protocol (SCTCP), Datagram Congestion Control Protocol (DCCP), Multipath TCP (MPTCP), UDP-Lite and others. In this dissertation, we are only interested in TCP and UDP, which are the most popular ones, and QUIC [8] which is coming to replace TCP.

2.3.1 Internet Protocol

It is a protocol designed for interconnected systems of packet-switched computer communication networks. It is responsible for addressing, fragmenting and defragmenting data (if necessary for small packet networks), and for transmitting blocks of data (called datagrams) from sources to destinations. This is the protocol called on by host-to-host protocols in an internet environment to carry an internet datagram to the next gateway or destination host. For example, a TCP module would call on the internet module to take a TCP segment (including the TCP header with addresses and other parameters, and user data) as the data portion of an internet datagram; the internet module would then create an internet datagram out of the TCP segment and call on the local network interface to transmit the internet datagram. Figure 2.1 shows its relation with high-level protocols [51].

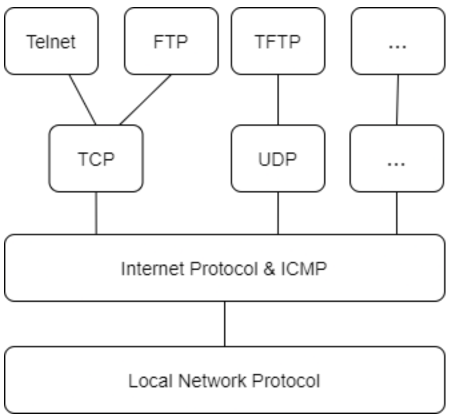


Figure 2.1: Protocol Relationships. Extracted from [51]. Internet Control Message Protocol(ICMP) is used by the IP protocol when a peer needs to report an error in datagram processing to another peer [50]. In this context a "local network" may be a small network in a building or a large network such as the ARPANET [51].

2.3.2 Network Sockets

A socket is a software structure that identifies a process running in a network node, which at the same time serves for receiving and sending data across the network. It also serves for inter-process communication. A socket has the same lifetime as the process which created it. The basic operations of sockets include opening a connection to a remote machine, sending and receiving data, and closing the connection [116, 88]. Moreover, the term network socket is most commonly used in the context of the Internet protocol suite, and is therefore often also referred to as internet socket. In this context, a socket is externally identified to other hosts by the triad of transport protocol, IP address, and port number. This identification is called the socket address and allows the Internet sockets to deliver the incoming data to the right process [107, 116, 88].

2.3.3 TCP

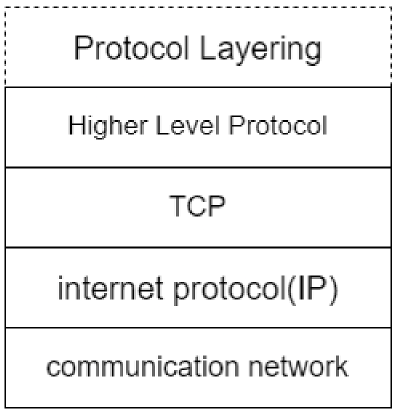


Figure 2.2: Protocol Layering [107].

TCP is a transport layer protocol that aims to provide a reliable process-to-process communication service in a multi-network environment [107]. TCP works on top of the IP internet protocol, which is responsible for TCP sending and receiving variable-length segments of information enclosed in internet datagram "envelopes".

As mentioned above, TCP focuses on providing highly reliable, secure, and oriented connections between peers. It does so by ensuring the following properties: basic data transfer, reliability, flow control, multiplexing, connections and precedence, and security.

Basic Data Transfer: TCP fragments a continuous stream of bytes into segments that are sent through the internet system in each direction between endpoints [107, 30]. To ensure the users that all the data they submitted are actually transmitted, TCP defines a push function. When the sending user triggers the function, it causes the TCPs to promptly forward and deliver data up to that point to the receiver [107].

Reliability: TCP has to ensure that the receiver always gets the original data from the sender once; the data must be neither damaged nor duplicated. To make this work, TCP assigns a sequence number to each packet transmitted, and requires a positive acknowledgment (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. At the receiver, the sequence numbers are used to correctly order the segments that may be received out of order, and also to eliminate duplicates. Damage is handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding corrupted segments. The reliability offered by TCP ensures that, as long as the Internet system does not become completely partitioned, no transmission errors will affect the correct delivery of data. TCP also offers mechanisms for recovering from internet communication system errors [107, 30].

Control Flow: TCP offers congestion control mechanisms to prevent the sender from overflowing the network, and to reduce packet loss. The receiver sends a window with every ACK indicating the maximum number of bytes it has available in its buffer; and the sender can not send more bytes than that. If the sender receives a window of zero, it temporarily stops sending data, because it means that the buffer of the receiver is full. In general, more occurrences of window sizes of zero, result in slower data transmission across the network [107, 30, 27].

Multiplexing: In order to have many processes within a host using TCP simultaneously, TCP defines a set of ports within each host. Processes within a host are identified by a unique port number. A port number combined with the network host address is defines a socket [107, 30].

Connections The reliability and flow control mechanisms described above require that TCPs initialize and maintain certain status information for each data stream, this is called a connection. A connection is the result of combining all of this data, including sockets, sequence numbers, and window sizes. Each connection is uniquely specified by a pair of sockets identifying its two endpoints. [107]. TCPs relies on the three-way handshake (Figure 2.3) agreement to correctly initiate a connection when two processes want to communicate; and once the communication is complete, the connection must be closed to release resources for other processes.

In Figure 2.3, TCP A sends a synchronization segment to TCP B with its sequence number, starting at 100; TCP B answers with a SYN segment + an acknowledgment to the SYN of A, saying that it will use sequence numbers starting at 300, and is now expecting sequence 101 from TCP A; TCP A answers with an ACK, which contains its sequence number and the sequence number it expects from TCP B; after ACKing the SYN of TCP B, TCP A sends some data with the same sequence it used previously, this is because ACKs do not occupy sequence number space, in order to prevent peers ACKing ACKs [107, 30].

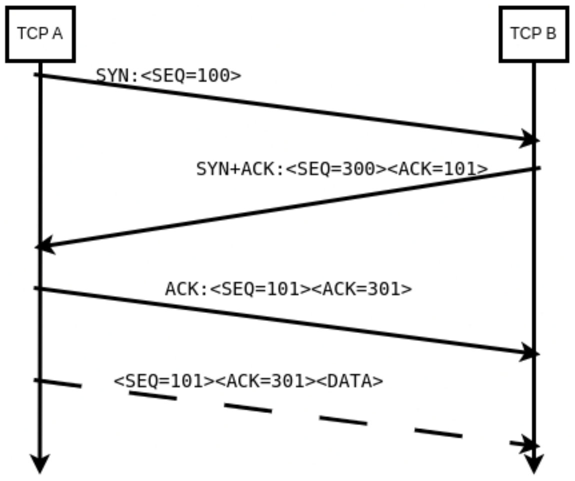


Figure 2.3: Basic 3-Way Handshake for Connection Synchronization [2] [107].

Precedence and Security: TCP allows its users to indicate the security and precedence configurations of their communication. However, default values are used if they do not specify any configurations.

2.3.4 UDP

The User Datagram Protocol, is a communication protocol designed for fast transmission of data. It does not offer a strong guarantee of reliability as TCP does; it can not guarantee delivery and duplicate protection, for example. UDP is transaction oriented, its communication is straight forward, its users do not need to establish a connection, the sender simply sends data without checking the order or whether they arrived at the destination [108, 30]
.

Protocol Description UDP is a connectionless protocol that maintains message boundaries[1], with no connection setup or feature negotiation. The protocol uses independent messages, ordinarily called "datagrams". It provides detection of payload errors and misdelivery of packets to an unintended endpoint, both of which result in discard of received datagrams, with no indication to the user of the service [30].

UDP provides neither reliability nor retransmission of packets. Its messages may be out of order, lost, or duplicated. It is also important to stress that UDP offers a relatively weak form of checksum, and applications that require these mechanisms are recommended to use a stronger integrity check of their payload [30].

UDP does not offer a flow control procedure, which can make a slow receiving application lose packets. Its lack of congestion control mechanism implies that UDP traffic may lose packets when using an overloaded path, and may also cause other protocols sharing the network paths to lose messages as well [30].

Moreover, UDP encapsulates each datagram into a single IP packet or several IP packet fragments. This feature allows a datagram to be larger than the effective path MTU (maximum unit of transmission). These fragments are later reassembled on the receiver side by the UDP receiver before being delivered to the application.

Despite all the features above-mentioned, UDP on its own does not support segmentation, receiver flow control, PMTUD[2] or explicit congestion notification(ECN). It becomes the application's task to ensure these features when using UDP [30]..

2.3.5 QUIC

QUIC is a multiplexed transport protocol built to take advantage of the UDP protocol. It was initially designed to replace TCP+TLS+HTTP/2, with the goal of improving user experience, particularly web page loading time [8, 117]. QUIC enhances the performance of TCP-based connection-oriented web applications; it does this by establishing a number of multiplex connections between endpoints using User Data Protocol (UDP), and is designed to obsolete TCP at the transport layer for many applications, thus earning the protocol the occasional nickname of "TCP/2" [115].

The main advantages of QUIC over TCP include: low connection establishment latency, multiplexing without head-of-line blocking, authenticated and encrypted payload, stream and connection flow control, connection migration, and transport extensibility.

Connection Establishment QUIC uses an encrypted transport handshake to establish a secure transport connection. On a successful handshake, a client stores information about the server (host name, port, source-address token, etc.), so that on a subsequent connection to the same server, the client can establish an encrypted connection without additional

[1] is the separation between two messages being sent over a protocol.

[2] "It stands for Path Maximum Transmission Unit, it is a technique to determine an acceptable MTU between two IP network connections. The goal of this technique is to discover the largest size datagram that does not require fragmentation anywhere along the path between the source and destination" [28]

round trips (0-RTT), and data can be sent immediately following the client handshake packet without waiting for a reply from the server. To initiate the first connection with the server, the client sends an *inchoate client hello* (CHLO) to the server, to make the server send a *reject* (REJ) message. The *REJ* message contains the server configurations , cryptographic specifications, and a *source-address token* (an authenticated-encryption block with the client IP address and a timestamp by the server) which the client will use in subsequent connections to authenticate itself. Once the client has received the *REJ* message, it authenticates the server and then sends a *complete CHLO*, containing the ephemeral Diffie-Hellman [106] public value (in short, it will be used by the server to encrypt the data). After sending the complete CHLO, the client has everything to encrypt and decrypt the data exchanged between them, and can start sending application data immediately without waiting for the *server hello* (SHLO) message. In subsequent connections, the client starts by sending the complete CHLO message to the server and the encrypted data; the server answers either with an SHLO or a REJ; SHLO if the server accepts the token and is able to authenticate the client; REJ indicates that the server did not authenticate the client, so the client must resend the complete CHLO with the information on the new REJ. [91] Figure 2.4 shows how the connection takes place.

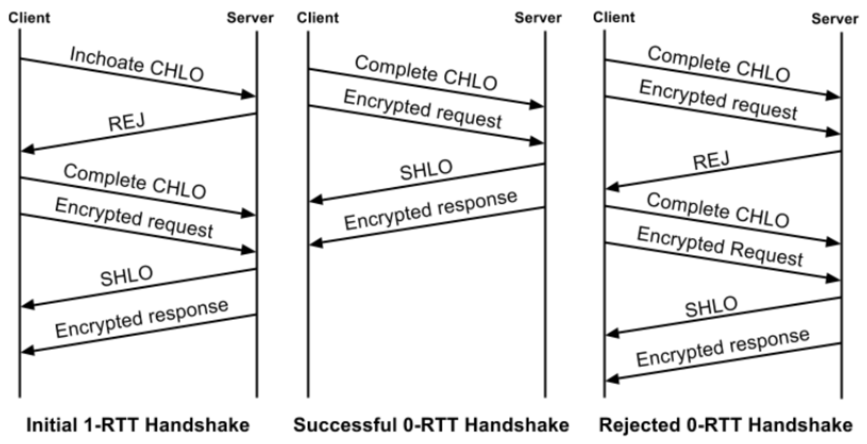


Figure 2.4: QUIC connection establishment. Extracted from [91].

Stream multiplexing Applications commonly multiplex data within the single-bytestream abstraction of TCP, which blocks the whole stream until retransmission if a packet is lost (head-of-line blocking, the first packet in a queue can not be delivered because there are packets missing, the packets behind will also be blocked [92]). Therefore, QUIC uses multiple streams to prevent this issue. In QUIC a stream is an abstraction for a unidirectional or bidirectional channel within a connection. They can be created by sending data, and are identified within a connection by a numeric value, referred to as the stream ID. Streams have frames (STREAM frames) which encapsulate data sent by an application. Moreover, an endpoint uses the Stream ID and Offset fields in STREAM frames to indicate the stream the frames belongs to and to order the data; and a QUIC packet is composed of one or more stream frames plus a header. QUIC achieves Stream multiplexing by placing

STREAM frames from multiple streams into one or more QUIC packets. A single QUIC packet can include multiple STREAM frames from one or more streams. When a packet loss occurs, only streams with data in that packet are blocked waiting for a retransmission to be received, while other streams can continue making progress. Implementations are advised to include as few different streams as necessary in an outgoing packet to prevent multiple streams blocking when the packet is lost [91, 87].

Stream and connection flow control To prevent a fast sender from overwhelming a slow receiver, and to stop a malicious sender from using up a lot of memory at a receiver, it is important to limit the amount of data that a receiver could buffer. Therefore, streams are flow controlled both individually (stream-level flow control, the receiver indicates the available space it has for each stream buffer) and collectively (connection-level flow control, the receiver indicates the total available byte space it has to buffer all the frames) to allow a receiver to restrict memory commitment to a connection and put pressure back on the sender. A QUIC receiver controls the maximum amount of data the sender can send on a stream at any time, and also to limit concurrency, a QUIC endpoint controls the maximum cumulative number of streams that its peer can initiate.

Transport extensibility QUIC is implemented outside the operating system kernel, in the application space. This is something advantageous because it circumvents the hardships of evolving TCP, that requires updates to kernels. This way, QUIC can evolve and be deployed faster and with minimal effort. [8, 91].

Authenticated and encrypted payload With QUIC, overhead connection setup is greatly reduced compared to TCP. Because most of the HTTP connections will demand TLS, in QUIC, exchanging setup keys and supported protocols are part of its initial handshake process. The response packet got by the client when they open a connection include the data needed to encrypt future packets, in this way, there is no need to set up a connection then negotiate the security protocol via additional packets. QUIC packets are fully authenticated and application data are always encrypted [117].

Connection migration QUIC endpoints identify their connections using a 64-bit Connection ID, encoded in every packet they exchange. The connection ID allows a connection to survive when the port number and the IP address of the client changes; the client does not need to open a new connection to resume exchanging application data with the server. These changes can be caused by the client changing the network or by a NAT[3] rebinding (When a connection is idle for a long time, the NAT may guess it has terminated and assign the client port to a new connection). When the client sends packets with a new address to the server with the same connection ID, the server recognizes that the client

[3]NAT stands for network address translation. It's a way to map multiple local private addresses to a public one before transferring the information. [24]

has changed its address, and starts sending data to the client using the new address [91, 100, 87].

2.3.6 Discussion

We started off this section by introducing the concept of the OSI model and its relevance when talking about network abstractions. We also defined the concepts of network sockets, and the widely famous Internet Protocol. We then went on discussing the two major transport protocols, TCP and UDP, which are also the basic for network abstractions, and an emerging one called QUIC. We saw that TCP [107] is a protocol widely used for its reliability guarantees, while UDP [108] is used mainly when great performance is the goal. On the other hand, QUIC [8, 117] is a UDP-based protocol that was initially designed to replace TCP+TLS+HTTP/2 in order to improve web browsers response times; it has now been gaining relevance as an alternative to TCP as it can both guarantee reliability and performance, and is much easier to evolve because it is implemented in the user space.

In the next sections, we show the importance of these protocols by discussing frameworks, distributed services, and applications that rely on them to exchange data over the network.

2.4 Messaging Patterns

The transport protocols (UDP, TCP, and QUIC) we saw above, are widely used for implementing various types of communications among peers on the Internet. These types of communications are called messaging patterns. Messaging patterns describe how different parts of an application, or different systems, connect and communicate with each other [55].

There are various types of messaging patterns; in this section, we briefly introduce those that are going to be referenced throughout the document. They are Publish/Subscribe, Message queue, Request/Reply, and Push/Pull.

Publish/Subscribe (Pub/Sub) It is an asynchronous way of communication in which a client sends messages to a topic queue in a broker (server). The broker in turn sends copies of the messages to all clients that showed interest to that topic. A topic queue is an abstraction of a queue where all the messages with a particular topic go. [21] The broker is a server which filters the messages sent by the publishers and distribute them to all subscribers. The broker decouples the publishers from the subscribers in three ways: the publishers and the subscribers are not aware of the location of one another and do not exchange address information (space decoupling); the publishers and the subscribers do not need to be active during the same time (time decoupling), and a subscriber does not have to wait for the publisher to send a message (Synchronization decoupling) [19].

However, there can be a publish/subscribe system without a broker [104], the clients coordinate among themselves how to forward the messages to the interested peers.

Message Queue In this pattern, producers send messages to queues and consumers read the messages. When a consumer reads a message from the queue, the message becomes unavailable for other consumers [18].

Push/Pull In this pattern, client publishers push data to the server and client consumers pull the data, consumers explicitly ask for the data [7].

Request/Reply This is a distributed communication pattern in which a peer (client) sends a request to another peer (server) expecting a reply from it. The client either waits for a reply with a timeout, or receives a reply asynchronously [20].

2.4.1 Discussion

In this Section, we briefly discussed some of the messaging patterns that will be mentioned throughout the document. These patterns are Publish/Subscribe, Message Queue, Push/Pull, and Request/Reply. These patterns describe how different peers communicate on the Internet, and rely on the widely used transport protocols such as UDP, TCP, and QUIC, for communication over the network.

2.5 Communication Libraries

A library is a set of reusable pre-written block of code that solve specific problems [29]. Netty [39], Libp2p [35] and ZeroMQ [59] are communication libraries that offer a set of functions to allow communication between peers using the commonly well known protocols such as TCP, UDP, QUIC, and others. In the following sections, we discuss the network abstractions offered by each of them.

2.5.1 Netty

Netty is a java library for building blocking and non-blocking networking applications. It supports the implementations of core transport protocols such as UDP and TCP [39, 101].

Its top performance comes from the fact that Netty uses native java socket libraries (that have non-blocking calls), and the class java.nio.channels.Selector to provide asynchronous non-blocking input/output (NIO) operations. The selector class uses the event notification API to indicate which non-blocking sockets are ready for I/O operations. The Selector can use a single thread to monitor multiple connections, because the sockets read/write operation completion status can be checked at any time [101].

Netty has four components that allow the applications to access the network, and the data that flows through it. These components represent the resources, logic and notifications. They are Channels, Callbacks, Futures and Events and Handlers [101].

Channel Is an abstraction that represents a connection to an entity such as a device, a network socket, or a file that is able to perform distinct input/output operations. It can be opened or closed for sending and receiving data [101].

Callback Is a method provided to another method as an argument reference to be called later at an appropriate time. They represent a way to notify interested entities that an operation has completed. It is used by Netty to handle events such as the arrival of data from the network and the expiration of a timer [101].

Futures A Future is another way of notifying an application that an operation has completed. The difference between this and call backs is that Futures are objects that contain the result data of the asynchronous operations. Netty provides its own implementations of Futures (ChannelFuture), because the implementation offered by JDK is synchronous. Each outbound I/O operations in Netty returns a Future [101].

Events and Handlers They are used to notify the application about the status of the performed operations. Since Netty is a networking framework, the events can be active or inactive connections, data reads, or error events. These events can be sent to a user-implemented method, and allow the application to perform actions such as logging, data transformation, and application logics [101].

Transports They refer to the way data is moved around the network. Natty offers several ready APIs to use transports, such as: NIO (non-blocking I/O) and OIO (old blocking I/O) transports for asynchronous and synchronous networking communication respectively; Epoll is a native non-blocking transport for Linux to monitor file descriptors (identifier of an open file) to determine whether I/O operations are possible; Local, a transport for asynchronous communication between clients and servers in the same Java Virtual Machine (JVM). Some of these transports (NIO, Epoll, and OIO) support core protocols TCP and UDP [101].

2.5.2 Libp2p

Libp2p is a peer-to-peer (P2P) networking library that facilitates the development of P2P applications [35]. It is a collection of protocols, libraries, and specifications that make it easier to establish P2P communication between network participants [35]. It currently offers usable implementations for languages such as GO, Rust, JavaScript, JVM, Nim, $C++$, Swift, Python, and Erlang [32].

Libp2p handles a lot of networking tasks in a decentralized system, freeing the programmer to deal with and focus on their application logic development. Moreover, Libp2p is highly customizable regarding transport, identity and security; It tries to solve problems in areas such as transports, and others [31].

Transport Transports refer to the protocols that are used to move data over the network. In Libp2p, transports (or connections) are defined in terms of *listening* and *dialing*. Listening for accepting incoming connections from other peers. Dialing is for opening an outgoing connection to a listening peer. One of Libp2p design goals is to be transport agnostic, which means that the developer of an application decides which transport protocol to employ; this gives the developer the opportunity to use many different transport protocols at the same time. By design, Libp2p supports many different transports such as TCP, QUIC, WebSocket, WebTransport, etc. It also offers support for WebRTC, a framework used to establish real-time communication between browser and server, and browser to browser [35, 31, 33] Libp2p also offers stream multiplexing [37], which allows different processes to share a connection by using unique port numbers to distinguish the streams [37]; In the case of TCP connections [109], it uses stream multiplexers such as yamux [82, 36] and mplex [38]; for QUIC [109], it simply uses QUIC streams.

WebSocket It is a protocol that provides a two-way communication between client and server; both the client and the server can send messages without have been asked for in the form of requests. It does this by providing a single TCP connection for traffic in both directions. It allows for servers to send web browsers event notifications without the requests of the clients; it is an alternative to HTTP polling (HTTP clients sending requests to servers asking for data). Moreover, the protocol consists of a handshake and a data transfer part. In the handshake part, both the client and the server send their handshakes to each other. The handshake of the client is an HTTP Upgrade request, so that a single port can be shared by HTTP and WebSocket clients to talk to the server. After successful handshakes, clients and servers become enabled to exchange data back and forth [53].

WebTransport HTTP/3 uses QUIC to transport data over the network, and it does not offer a bidirectional communication channel between HTTP/3 clients and servers. WebTransport is an alternative to HTTP/3 unidirectional communication model (servers send data only after receiving a request from the client). It can be seen as WebSocket over QUIC; WebSocket is built on top of TCP, and is an alternative to TCP based HTTP versions; WebTransport is built on top of QUIC, and is an alternative to HTTP/3. Therefore, WebSockets benefit from all the advantages QUIC has over TCP [34].

2.5.3 ZeroMq

ZeroMQ is an asynchronous, high-performance messaging library used for building distributed and concurrent applications. It provides a message queue that can run without a dedicated message broker, unlike message-oriented middleware. ZeroMQ offers supports for common messaging patterns (pub/sub, request/reply, pipelining, and others) over a diverse set of transports like TCP. It makes inter-process messaging as simple as inter-thread messaging, keeping code clear, modular, and easy to scale. Additionally, it has supports for many languages such as C, $C++$, C#, Dart, Erlang, F#, GO, Haskell, Java, Node.js, Perl, Python, Ruby, Rust, and others. [59] [60]

It is asynchronous because it uses multiple worker threads to handle asynchronous tasks such as reading data from the network, enqueueing messages, accepting incoming connections, and others. Furthermore, its concurrency model consists of avoiding locks in order to let the threads run at full speed; the threads use the actor model for communication, they communicate asynchronously by passing event messages between them. Moreover, it launches one worker thread per CPU core to prevent unnecessary context switches [111].

ZeroMQ uses the ZMTP protocol for sending and receiving data over the network. ZMTP stands for ZeroMQ Message Transport Protocol, it is a transport layer protocol for exchanging messages between two peers over a connected transport layer such as TCP. The protocol aims at solving problems that arise when using TCP to exchange data over the network; problems such as TCP not preserving message boundaries (sending two messages as byte streams, TCP does not know where the first finishes and the second starts), TCP carries redundant metadata on each frame (such as, whether the frame is part of a multipart message, or not), TCP peers can not interact with older versions, and others [3, 4].

2.5.3.1 ZMTP

The ZMTP protocol is made of three layers: the framing, the connection, and the content layer [3, 4].

Framing Layer Framing consists of creating structured messages instead of breaking them into fragments. For example, the streams provided by TCP is turned into a series of length-specified frames. The frames are length-specified so that peers can safely reject them if they are oversized. A frame is like a container for the data; it is made of a length field, followed by a **flags** field, and a frame body of size length-1 bytes. The *flags* field consists of a single byte; the first bit indicates whether more frames will follow (if one) or not (if zero); bits 1 to 7 are reserved for future use and should be zero. A ZMTP message is made of one or more frames which are sent and delivered atomically (all of them is sent or none is sent); the sender stores all frames of a message in a queue until the last frame is sent [3, 4].

Connection Layer Allows for peers to establish a link. ZMTP offers bidirectional and asynchronous connections, at any time, either peer may send a message to the other. The connection is equivalent to a TCP connection, and each side of the connection sends a greeting followed with zero or more contents. A greeting followed by zero content (empty string) indicates that the peer is anonymous (the connection is not durable, all the related resources are deleted when the connection ends); a greeting followed by more contents (a unique identifier string) tells the receiving peer to indefinitely map the ID to the connection resources of the sender [3, 4].

Content Layer It defines the semantics and the format of the ZMTP content messages exchanged across a connection. The protocol defines 3 types of contents: the broadcast type is used by publishers and subscribers; the addressed type is used between peers in a request-reply pattern, and also between peers that require routing; and the neutral content type is used between peers that do not require routing [3, 4].

Interoperability . When version detection is enabled, the peers must send their versions after the greeting [3, 4].

2.5.3.2 Publish-Subscribe (Pub/Sub)

It is a messaging pattern in which a sender (publisher) sends a message to a set of interested consumers called subscribers. ZeroMQ provides 2 (sockets endpoint interfaces) to handle this type of communication: PUB sockets for publishers and SUB sockets for subscribers [83].

The PUB Socket It is used only by the publishers, and serves for sending messages to SUB socket subscribers, and may receive subscribe and unsubscribe messages. It creates an outgoing message queue for each subscribed client; messages are silently discarded from the queue after it has been sent; messages are only sent to subscribers with matching subscription; a queue is deleted (with its messages) if the subscriber disconnects; moreover, queues have configurable sizes, and new messages are discarded if they are full [83].

The SUB Socket It listens for incoming messages from the publishers. It creates a message queue for each connected publisher; if the publisher disconnects, it destroys the respective queue and discards the messages; queues have configurable sizes and new messages are discarded if the queue is full; it receives incoming messages from the publishers in a round-robin way (a time slot is given to each queue; it processes the messages in a queue within its time slot; when the time expires, it moves on to another queue); filters messages according to subscriptions and delivers them to the application [83].

2.5.3.3 Request-Reply (REQ/REP)

The REQ/REP pattern is used for request-reply type of communication, in which a peer sends a message to another peer and waits for a respective response. ZeroMQ offers two interfaces for this type of communication: REQ and REP sockets [84].

REQ Socket It is the client interface for sending request messages and receiving the respective replies. It is connected to any number of peers. It sends and receives exactly one message at a time; Its other aspects include blocking when sending, or returns an error if no peer is connected; accepts an incoming message only from the last peer to which it sent a request, the others are discarded [84].

REP Socket It is an interface that acts as a server for receiving requests and sending responses. It is connected to any number of peers; it receives and then sends exactly one message at a time. It receives the messages from the clients in a round-robin way, and delivers the data frames to the application. Unlike the REQ socket, it does not block on sending. It discards the reply if the peer which sent the request disconnects [84].

Pipeline It is a task distribution pattern where nodes push work to many workers, which in turn push the results to one or a few collectors. ZeroMQ offers two socket interfaces to implement this type of pattern: PUSH and PULL sockets. The PUSH socket is used for sending messages to workers, and is very similar with the PUB socket, with the difference being that PUSH blocks on sending, and sends to all worker nodes (instead of filtering by subscription) in a round-robin fashion; PUSH also differs from PUB in the manner that PUSH does not discard messages when queues are full. The PULL socket is very similar to the SUB socket, differing in the sense that PULL sockets do not use subscriptions, they simply receive from all peers they are connected to [85].

2.5.4 Discussion

In this section, we saw some of the frameworks used to build network abstractions. Netty is a Java framework for networking programming, it makes it easier to develop TCP and UDP applications; Libp2p is a library that facilitates the development of peer-to-peer applications, offers supports for TCP, UDP, and QUIC protocols, as well support for many language implementations; ZeroMQ is a zero broker messaging library for distributed applications, offering implementations for various languages.

Comparing them all to each other, Libp2p and ZeroMQ offer implementations for various programming languages and many more protocols. Relating to protocol implementations, Netty and Libp2p have support for QUIC [40, 35], while ZeroMQ does not. Moreover, ZeroMQ offers ready-to-use messaging patterns. These libraries could be combined in order to enrich the channels of Babel [79, 80] into supporting UDP, QUIC, and the messaging patterns offered by ZeroMQ.

2.6 Frameworks

This section discusses protocol composition frameworks and their network abstractions. These are frameworks that allow programmers to build complex protocols using a collection of pre-made building blocks [102].

2.6.1 Babel

Babel [79, 80] is a framework to develop, implement, and execute distributed protocols and systems. Babel offers an event driven programming model and execution that aim to simplify the task of translating pseudo-code specification into real working code, while allowing the programmer to focus on the relevant implementations of the desired application [79, 80].

In Babel, protocols describe the behavior of a distributed system, and they are represented by a Java class which contains the methods with the logic of the protocol in question, plus the methods for sending and receiving data over the network. A Babel process can have more than one protocol running at the same time; each protocol runs in a single thread; and the protocols in the same process communicate through events, such as notifications, requests, and replies. Each protocol has its own event queue from which consumes the events serially. Babel multithreaded execution model avoids concurrency issues [79, 80].

Network In Babel, communication between processes happen through *channels*. A *channel* is a network abstraction that focus on hiding all the complex implementation details associated with networking programming. The channels are implemented with Netty [39] and can be extended/modified by the developer. The protocols interact with channels using openConnection, sendMessage, and closeConnection primitives; and also for receiving events such as messages from the network; a channel can be used by more than one protocol inside the same process, and a protocol can also use more than one channel. The framework provides a TCP channel which is intended for TCP communication between babel processes. The framework is also flexible, as it allows the developers to design their own channels [79].

Babel experiments show that protocols implemented with Babel tend to show performances close to the optimized versions of competing implementations of those protocols [79].

2.6.2 Appia

Appia [103, 79] is a framework for protocol development that benefits from the Java inheritance model. Appia presents the concepts of sessions and channels. A session is an instance of a micro-protocol [103], and a channel is a sequence of session instances [102]. These elements give the developers more control over the binding of protocols to develop different types of services within a single application. However, this requires the developers to stack protocols, which makes it difficult for different protocols to interact. In Appia, all distributed system protocols are executed in a single execution thread. This causes undesired performance bottlenecks.

In Appia, communication with the outside world is made by inserting events occurrences into channels to notify about external happenings, these events can be the arrival and the sending of packets to the network [102]. The class **SendableEvent** is a common interface to all the events with data to be sent and received from the network [103].

2.6.3 Yggdrasil

Yggdrasil is a C framework and middleware for the development and execution of distributed applications and protocols that run on commodity devices in wireless ad hoc networks. Yggdrasil combines an event-driven programming model with a multithreaded execution environment that releases the developer from concurrency issues [77].

In Yggdrasil, each protocol (and application) is executed by an independent execution thread, enabling the evolution of multiple protocols running in parallel in the same process. Protocols are also provided with an event queue from which they will consume their events [77].

Yggdrasil has four components. *Yggdrasil Runtime* configures the radio device for network communication and handles the execution of the protocols. *Dispatcher Protocol* deals with network communication. *Timer Management Protocol* monitors all timer events. Finally, *Protocol Executor* allows some protocols to share a single execution thread [77].

When a Yggdrasil application starts, the Runtime is immediately initialized with network configurations which include the radio mode, the frequency of the radio, and the network name to be created or joined. The Runtime configures the radio device through the Low Level Yggdrasil Library, which invokes system calls to manipulate the radio interface. The Low Level Yggdrasil also provides the Runtime an abstraction channel through which network messages are directly exchanged at the MAC layer (layer 2 of the Network Stack). The channel is equipped with a kernel packet filter for filtering unwanted network messages from sources other than Yggdrasil processes. Additionally, the Runtime gives the Dispatcher access to the previously created channel, to handle network communications [77].

However, Yggdrasil only supports one network abstraction, it can only implement protocols that run on wireless ad hoc [77] or wired IP networking [79]. It is also developed in C, which requires technically disciplined developers as to avoid frequent memory issues that could halt the development task [77, 79].

2.6.4 Cactus

Cactus is a C, C++ and Java framework for developing configurable protocols and services, in which each service property or functional component is implemented as a separate module [78].

In Cactus there are two ways of protocol composition. Coarse grain and fine grain. [78] The fine grain protocols are micro-protocols that can exist on their own [78].

Coarse grain, composite protocols, are composed by forming a hierarchical graph of a set of protocols. The composite protocols group a set of finer grain micro-protocols arranged in a cooperative way (with no hierarchy), which interact via event triggers and data sharing. In the hierarchical graph, micro-protocols can not exist on their own. The composite protocols are linked by edges, and only linked protocols can communicate [78].

Micro-protocols can be executed asynchronously, however, it is the programmer's task to enforce synchronization, as Cactus does not offer it [78].

To communicate with the outside world, the micro-protocols at the lower level may open sockets or can be placed on top of an x-kernel [86] protocol stack, since Cactus is an extension of the x-kernel [102] framework. The x-kernel protocol stack provides three communication object primitives: protocols, sessions, and messages. A protocol corresponds to a conventional network protocol such as IP, UDP and TCP; a session represents an instance of a protocol; they are created at runtime with **open** or **openDone** operations, and correspond to the end-point of a network connection; and messages are objects that move through the session and the protocol objects, they contain the protocol headers and the user data [86].

2.6.5 Discussion

In this section, we discussed framework for protocol implementations such as Babel [79], Appia [103], Yggdrasil [77] and Cactus [78]. However, all of them have limiting features. Appia [103] has a single thread execution model that causes undesirable performance bottlenecks; Yggdrasil [77] supports only protocols for ad hoc networks; Cactus [78] does not ensure synchronization mechanisms, it is the task of the developer to do so; Babel [79] has networking limitations, as it currently supports only TCP based communications. Moreover, Babel does not support communication with devices running on ad hoc networks; integrating Babel with Yggdrasil would definitely expand its reach.

2.7 Networking Services

A network service in computer networking is an application running at the network application or higher layer that offers data storage, manipulation, presentation, communication, and other functionalities to other applications in the network. These services are often implemented using a client-server or a peer-to-peer architecture based on application layer network protocols. [54] Each service is typically offered by a server. Client components operating on other devices access it through a network. However, it is also possible that server and client components operate on the same machine [56].

In this section we discuss RabbitMQ [74], Apache Kafka [65], and ActiveMQ [9]. These services provide communication abstractions through a centralized logic components that expose services to the network.

2.7.1 RabbitMQ

RabbitMQ is a lightweight and easy to deploy open source message broker. Just like message brokers, RabbitMQ receives messages from application publishers and route them to application consumers [74].

Client applications interact with RabbitMQ using client libraries of the protocol they want to use. They first open a TCP connection, then the client and the server negotiate the specific messaging protocol to be used. All the protocols are TCP-based, and assume long-lived connections (connections are not opened for each protocol operation) for efficiency; a client library connection uses just a single TCP connection. After the connection, clients become able to publish and consume messages, and perform other operations provided by the protocol. Furthermore, since connections are designed to be long-lived, clients usually register a subscription in order to be able to consume the messages, and the messages are delivered (pushed) to them instead of polling (client asking RabbitMQ if they have messages for them). A special proxy server [71] can be used for clients that cannot keep long-lived connections. The client makes a connection to the proxy server, and the proxy opens a connection to the RabbitMQ server using the credentials provided by the client; the proxy forwards the traffic from the client to RabbitMQ, but when the client disconnects, the proxy intercepts the call and keeps the connection open to be reused later when the client connects again [72].

Moreover, RabbitMQ supports several messaging protocols, all TCP-based, either directly or through the use of plugins, among which include: AMQP 0-9-1 and extensions, STOMP, MQTT, AMQP 1.0, HTTP and WebSockets [75, 72].

2.7.1.1 AMQP 0-9-1

AMQP stands for Advanced Message Queuing Protocol. It is a messaging protocol that enables client applications to communicate with messaging middleware brokers. It is an application level protocol that relies on TCP for reliable delivery. It typically has long-lived connections. It uses authenticated connections that can be protected using Transport Layer Security(TLS) [52] [73]

In the protocol, messages are published to *exchanges* (often compared to post offices or mailboxes). Exchanges then distribute message copies to *queues* using rules called *bindings*. Then the broker either deliver messages to consumers subscribed to queues, or consumers fetch/pull messages from queues on demand. When a message is delivered to a consumer, the consumer notifies the broker, either automatically or when the consumer chooses to do so. When message acknowledgements are in use, the broker only deletes a message after it has received all the expected acknowledgments [73].

In certain situations, for example when the message cannot be routed, the message is dropped or delivered to the publisher, which chooses what to do with the message [73].

AMQP 0-9-1 is a programmable protocol, meaning that queues, exchanges, bindings and routing schemes are primarily defined by the application developers, not by a broker administrator [73].

Because some applications need to have multiple connections to the broker, AMQP 0-9-1 connections are multiplexed with *channels*, which can be seen as lightweight connections that share a single TCP connection. Every protocol operation performed by a client happens on a channel, and communication on a particular channel is completely independent and separate from others. Therefore, every protocol operation carries with it a channel ID, to identify the channel the operation is for. It is also important to note that channels only exist in the context of a connection, and never on its own. When a connection is closed, so are all channels [73].

2.7.1.2 AMQP 1.0

It is a more complex protocol than AMQP 0-9-1, and despite the name, it is nothing like AMQP 0-9-1 / 0-9 / 0-8. They are totally different. AMQP 1.0 imposes much less semantic requirements, which makes it easier for brokers to support it [75].

AMQP is a transfer protocol that provides structure for binary data streams (like serialized data types) that flow in either direction over the network; the structure provides for delineation (boundary) for distinct blocks of data (called frames), so that peers can understand when frames can be transfered and the transfers can stop. The protocol can be used for peer-to-peer communication, for interaction with message brokers that support queues and publish/subscribe entities, and also for interaction between different systems [22, 110].

AMQP network consists of connected nodes. The nodes are entities inside containers responsible for the safe storage and/or delivery of messages. Examples of containers can be brokers and client applications. Examples of AMQP nodes can be producers, consumers and queues. AMQP allows multiplexing, a single connection can be used for many communication paths between nodes [22, 110].

The network connection is initiated by the client container by making an outbound TCP socket connection to a receiver container, a container that listens and accepts inbound TCP connections. The connection handshake includes the negotiation of the protocol version and the security specifications [22, 110].

After the establishment of the connection, the containers declare the maximum frame size they are able to handle, and after an idle timeout they will both disconnect if there is no activity. They also declare the number of concurrent channels that are supported. A channel is a unidirectional, outbound abstraction path on top of the connection. In AMQP, a session takes a channel from each of the interconnected containers to form a bidirectional communication path. It allows multiple sessions so that high priority traffic can get past normal traffics [22, 110].

The protocol offers a window-based flow control similar to the one presented by TCP; when a session is created, each peer declares the number of frames he/she will be willing to accept into its receiving window. During the exchanging of frames, transfers stop if the windows fill in until the window is reset or expanded [22, 110].

2.7.1.3 STOMP

Simple (or streaming) Text Oriented Message Protocol (STOMP), is a simple text-based protocol for sending data across applications. Compared to AMQP, it is a simpler and less complex protocol, being more similar to HTTP. STOMP clients can communicate with almost every available STOMP message broker. This allows for simple and widespread messaging interoperability across many different platforms, languages and brokers. For example, it is possible to connect to a STOMP broker using a telnet client [76, 75].

RabbitMQ supports all current versions of STOMP through the Stomp plugin. Moreover, STOMP does not deal with neither queues nor topics. It uses a SEND semantic with a destination string. The messages are then mapped to topics, queues, or exchanges by RabbitMQ (other brokers might have a different strategy). Then the consumers subscribe to those destinations. Therefore, it advised if one is planning to construct a simple message queuing application that won't place sophisticated demands on a combination of exchanges and queues [76, 75].

2.7.1.4 Message Queuing Telemetry Transport (MQTT)

MQTT is a messaging protocol (set of rules) used for machine-to-machine communication, used by smart sensors, wearables, and other Internet of Things (IoT) devices that typically have to exchange data over a resource-constrained network, with limited bandwidth [70, 19].

The protocol uses the publish/subscribe model to decouple the message sender (publisher) from the message receiver (subscriber). It does so by using a third component, called a broker, that acts as an intermediator between the publisher and the consumer. The broker has the goal of filtering the messages from the publishers to distribute them to the correct consumers (subscribers) [70, 19].

MQTT publish/subscriber model defines the clients (publishers and subscribers) and the brokers as: an MQTT client is any device (from a server to a microcontroller) that uses the MQTT library; the client can send and receive messages; the MQTT broker is the server which coordinates the messages between different clients, it receives the messages from the clients, filters them by topic, identify the subscribed clients of each topic, and sends each message to its subscriber(s). The broker also authorizes and authenticates the clients; passes messages to other systems for further analysis, and handles client sessions [70, 19].

Clients and brokers start communication by establishing an MQTT connection. Clients initiate the connection by sending a CONNECT message to the MQTT broker. The broker confirms and responds with a CONNACK message. The communication is based on the TCP/IP protocol. Clients never connect to each other, just with the broker. Once the connection is established, the client either send messages to the broker, receive messages from the broker, or do both. The messages sent by the clients contain the topic identifier and the data itself. When the broker gets the message, it forwards it to the respective subscribers; if the consumers lost connection, the messages can be stored or discarded according to the configuration. The broker can also retain the messages and send them to new subscribers [70, 19].

HTTP [1] Hypertext Transfer Protocol (HTTP) is used to load webpages using hypertext links, texts that point to a specific remote or local element. It is an application layer protocol (the OSI model layer used by end-user software such as web browsers) that runs on top of other protocols such as TCP and UDP. Each message is either a request or a response, and the communication is unidirectional (servers can not send messages to the clients without receiving requests). A client constructs request messages that communicate its intentions, and routes them to an origin server; the request is made by a method type (GET for fetching and PUT for uploading data), URL (Uniform Resource Locator, the address of the data), HTTP headers (to give the recipient additional information, for example, send the response in French) and an optional body containing some data (if PUT is used). The server listens for requests, parses, interprets the message intentions, and responds to the client with a response message.

2.7.2 ActiveMQ

ActiveMQ is an open-source, multiprotocol, Java-based message broker (receives messages from application publishers and route them to application consumers). It provides high availability, performance, scalability, reliability, and security for enterprise messaging. Additionally, it supports multiple, industry standard protocols, and allows connections from clients written in different languages such as JavaScript, C, C++, Python, .Net, and more [9, 15].

Moreover, ActiveMQ offers message queuing, in which the broker (server) routes each message in the queue to one of the available consumers in a round-robin pattern; and publish/subscribe (pub/sub) messaging, in which the broker routes each message to every available subscribers [95].

Client-Server In ActiveMQ, clients, requiring Publish/Subscribe or queue based type of communications, connect with a message broker server using typically TCP or SSL (Secure Sockets Layer, enables secure connection between client and server). Moreover, clients can also use network discovery to find available brokers. [10]. Additionally, UDP is also used by clients to communicate with the servers [11, 12].

Peer-To-Peer In the absence of servers, clients can form a peer-to-peer network and use multicast (communication from one peer to many peers, but not to all) for discovery or TCP/SSL unicast for heavy-lifting because it is much faster for communication [10].

Slave-Master Replication Model The slave-master system used in ActiveMQ consists of a master broker and a set of slave brokers. The slave brokers serve only to duplicate the state of the master broker (messages, acknowledgments, and transactions). Clients use the failover transport [13, 14] that allows them to connect to the available master broker. If the master goes down, it is automatically replaced by another broker and the failover transport [13, 14] immediately connect the clients to the available master [13].

2.7.3 Apache Kafka

Apache Kafka is an open-source distributed event streaming platform. By event streaming platform, it means that Kafka offers a publish/subscribe system, persistent storage and a processing service for streams of events. Kafka provides all those functionalities in a distributed, highly scalable, elastic, fault-tolerant, and secure way. Kafka can be deployed on bare-metal hardware, virtual machines, containers, on-premises and as well as in the cloud [65].

The Kafka system consists of servers and clients that communicate thorough a high-performance TCP based protocol. Some servers form the storage layer (called brokers) and others use Kafka Connect to continuously import and export data as event streams to integrate Kafka with external applications. The clients allow users to write distributed services and applications that read, write and process streams of events [65].

Network . Kafka uses a binary protocol over TCP that defines all APIs as request/response message pairs.The clients initiate all the requests, which result in corresponding responses from the server, except for no answer requests [66].

The protocol works by having the client initiating a socket connection and then writing a sequence of request messages, after which reads back the corresponding responses. No handshake is required on connection or disconnection, the justification is “TCP is happier if you maintain persistent connections used for many requests to amortize the cost of the TCP handshake, but beyond this penalty connecting is pretty cheap” [66].

On a single TCP connection, the server guarantees that requests will be processed in the order they are sent and that the responses will follow the same order. Clients can send requests while waiting for responses for previous requests. In addition, messages are size delimited; any requests that are larger than the maximum configured request size will cause the socket to be disconnected by the server [66].

According to the official documentation [65], Kafka is better than traditional message brokers, it offers high throughput and strong durability guarantees "Kafka works well as a replacement for a more traditional message broker". However, in this domain, Kafka is comparable to messaging systems like ActiveMQ and RabbitMQ [65].

Pubish/Subscribe The producer sends data directly to the broker that is the leader, of the partition of the target topic. To know the leader brokers, the producer sends requests to Kafka nodes asking for metadata related to the leaders. The consumer can also send batch requests in order to improve efficiency. On the other hand, the consumers work by making fetch requests to the leader brokers of the topics they are interested in [69, 68].

Replication . Kafka relies on ZooKeeper for data replication. however, its high efficiency is based on the fact that the followers in Kafka handle read requests, whereas in ZooKeeper the leader handles both the write and read requests. [112, 67]. In Section 2.8.3 we describe how the ZooKeeper [64] replication model works.

2.7.4 Discussion

RabbitMQ [73, 75, 74, 76] and ActiveMQ [9] are both message brokers with support for many networking protocols. Their main protocols are TCP-based. On the other hand, Apache Kafka [65, 66] is a streaming platform that offers a variety of services, such as pubish/subscribe system, persistent storage, and event streams processing; it also uses a TCP-based protocol like the previous mentioned services. However, despite their differences, Apache Kafka, RabbitMQ and ActiveMQ are comparable to Kafka in terms of throughput delivery and durability guarantees [65]. Moreover, their support for a diverse set of transport protocols, and messaging patterns would contribute very much into making Babel [79, 80] able to support a wider range of additional features.

2.8 Distributed Applications

In this section, we are going to discuss distributed applications, these are applications that run on more than one computer and communicate through a network [99]. They offer distributed services like databases, cache, and group services. The applications we are going to overview are: Apache Cassandra [17, 16, 26], Redis [41, 48, 42], ZooKeeper [64, 63, 62]. Our main goal is to focus on how they offer their network abstractions.

2.8.1 Apache Cassandra

Cassandra is a highly scalable, available and fault-tolerant open-source NoSQL database. Cassandra is perfect for managing large amounts of structured, semi-structured, and unstructured data across multiple data centers and the cloud, while also offering linear scalability, maximum flexibility, great performance, fast response times and operational simplicity with no single point of failure [26].In benchmarks and real applications, Cassandra consistently outperforms popular NoSQL alternatives, primarily due to fundamental architectural choices [17].

Cassandra's capacity may be easily increased simply by adding new nodes online. For example, if 2 nodes can handle 100,000 transactions per second, 4 nodes will support 200,000 transactions/sec and 8 nodes will tackle 400,000 transactions/sec [26].

Network. Cassandra employs a peer-to-peer distributed system across homogeneous nodes to deal with failures [26]. The nodes form a ring topology and communicate with each other by frequently exchanging state information about themselves and other nodes across the cluster using a peer-to-peer gossip protocol [26] over TCP/IP [16]. The data is partitioned, and automatically distributed among all nodes in the cluster [26], and the node that gets the client request serves as the proxy between the client and the node in the ring that contains the requested data [26].

2.8.2 Redis

Redis is an open source in-memory data structure store, used as a distributed, in-memory-key-value database, cache, message broker, and streaming engine with optional durability. It offers data structures such as strings, hashes, lists, sets, sorted sets with range queries, bitmaps, hyper-loglogs, geospatial indexes, and streams. It also has built-in replication, LRU (Last Recently Used) cache eviction policy, transactions, publish/subscribe, different levels of on-disk persistence, and offers high availability. Moreover, its in-memory dataset is the core engine for its top performance achievement, it does not need to write every change to disk [41].

Replication . Redis offers an asynchronous primary-replica replication model. The primary replica receives all the write operations and is responsible to propagate the changes to the replicas. Additionally, the replicas are enabled to receive read operations. By asynchronous, it means that the clients performing write operations get an acknowledgment before their request is propagated to the replcas [43].

Caching Redis exploits the available memory on application servers to directly store some subset of the database information. It stores the reply of popular queries for later use [44].

Publish/Subscribe A client (publisher) pushes data to a specific channel and interested (subscribed) clients will be able to read the data [45].

Transactions Redis transactions allow the execution of a group of commands in a single, isolated step, without interference. The commands in a transaction executed in sequentially [46].

Additionally, Redis uses two different network abstractions, one for client to server communication and another for server to server communication [42, 49].

RESP, Client to Client Communication RESP is the protocol used by Redis clients to communicate with the Redis server. It stands for Redis serialization protocol. It is just a serialization protocol that supports data types such Simple Strings, Errors, Integers, Bulk Strings and Arrays. While it is non-TCP specific, it is only used with TCP connections (or equivalent stream-oriented connections like Unix sockets) [42].

RESP is a Request-Response model protocol. Requests in Redis are sent from the client to the Redis server as arrays of strings that represent the arguments of the command to execute. The server replies accordingly with one of the supported data types. [42] However, it can sometimes work as a *pipelining* and other times as a *Publish/Subscribe*. It works as a pipelining when clients send multiple commands at once (not waiting for individual replies), and later read the replies in a single step [47]; It works as a publish/subscribe when clients subscribe to a Publish/Subscribe channel, the protocol becomes a *push* protocol, and the server will automatically send new messages to the client [42].

RESP is binary-safe (does not corrupt binary data) and uses prefixed-length to transfer bulk data [42].

Server to Server communication In Redis, cluster nodes are connected using a TCP bus and a binary protocol called the Redis Cluster Bus [48], a node-to-node communication channel that uses a binary protocol to spread information to any connected nodes [49]. Every node is connected to every other node in the cluster using the cluster bus [48]. The nodes use a gossip protocol to exchange information about the cluster in order to discover new nodes, to send ping packets, and to signal specific conditions [48]. The cluster bus is also used by the nodes for Publish/Subscribe messages across the cluster and to orchestrate manual failovers (fault tolerance mechanism) [48].

2.8.3 ZooKeeper

ZooKeeper is a centralized service for maintaining configuration information, naming, offering distributed synchronization, and providing group services. ZooKeeper focuses on separating the essence of these different services into a very simple interface to a centralized coordination service. It is distributed, with high reliability and performance. [62]. Moreover, it uses the Atomic Broadcast protocol to keep all the servers synchronized [63].

Atomic Broadcast. It is an atomic messaging system that keeps all the servers synchronized. It guarantees reliable delivery, total order, and causal order. In reliable delivery, if a message is delivery by one server, it will eventually be delivered by all the servers. In total order, if message A is delivered before B by one server, then A will be delivered before B by all the servers. Finally, in causal order, if a sender delivers A then sends B, A must be ordered before B; and if a sender sends C after sending B, B must be ordered before C.

Additionally, in order to guarantee that point-to-point FIFO channels can be constructed between the servers, so that it can guarantee efficiency and reliability, the protocol relies on TCP for communication, mainly due to TCP's properties of *ordered delivery* (A message m is delivered only after all messages sent before it have been delivered) and *No message after close* (a FIFO channel cannot send or receive messages once it is closed) [63].

The protocol consists of two phases: The first one is the leader activation phase, the phase in which the leader is elected and activated; And then there is the active messaging phase, the phase in which the leader accepts requests and proposes to the follower servers. The active messaging is very similar to the classic two-phase commit, but it does not need to handle aborts [63].

Leader Activation This phase includes the leader election phase, in which, the leader ends up with the highest transaction ID(zxid, a 64-bit number); the zxid is important to ensure total ordering, as all proposals will be stamped with a unique zxid, obtained incrementally by the leader; ZooKeeper does not care which algorithm is used for leader election as long as the leader has seen the zxid of all the followers, and it is followed by a quorum of followers. However, the leader needs to become active to start receiving and issuing proposals, and that only happens when a quorum (n/2+1 servers) of followers (including the leader) has synchronized with the leader. When a server becomes the leader, each follower will connect with it and receive missing proposals (or a snapshot of the state of the leader if they are too far behind); and then the leader will propose `NEW_LEADER` proposal to the synchronized follower, which the follower will ACK if it is the only `NEW_LEADER` proposal it has seen; the new leader will commit the `NEW_LEADER` proposal only after a quorum of followers has ACKed it; after that, the leader becomes active, and the followers will commit any state they received from the leader. Furthermore, if there is an error, the leader and the followers will time out and will go back to the leader election phase [63].

Active Messaging This happens after the leader election phase. In this phase, the leader receives requests from the clients and proposes them to the followers by the order the leader has received them; the followers will process and ACK the messages respecting the same order; the leader will issue a COMMIT to all the followers once it gets ACKs from a quorum of followers; finally, the followers process the commits respecting the same order. [63] Figure 2.5 shows how the **Active Messaging** phase works.

Client-Server communication. Client applications use TCP to communicate with the ZooKeeper servers [64]. The clients send requests, send heart beats, and get responses to a single server (connect only with a single server). Moreover, if the connection breaks, they will try a different one [64].

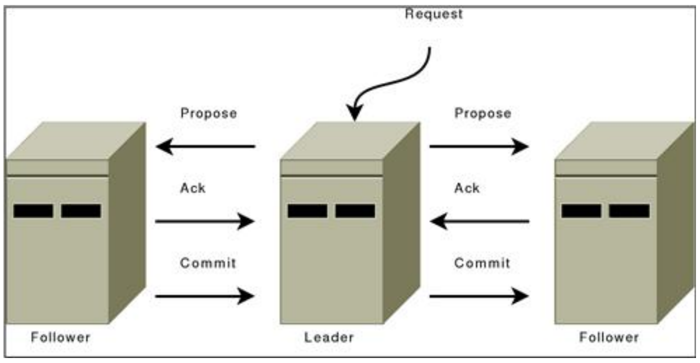


Figure 2.5: Active Messaging [63]. Extracted from [63].

2.8.4 Discussion

In this section we overviewed distributed applications such as Apache Cassandra, Redis, and ZooKeeper. Apache Cassandra is a highly scalable, available, and fault-tolerant NoSQL database that uses a TCP/IP gossip protocol network abstraction for communication between its nodes [17, 16, 26]. Redis is an in-memory data structure store that uses TCP-based network abstractions for communications between clients and servers, and servers to servers [41, 42, 48]. Zookeeper is a centralized service for maintaining configuration information, naming, offering distributed synchronization, and providing group services [64]; it uses TCP-based protocols for servers to servers and client to servers communications [63, 62]. However, despite their high performance and reliability, all the before-mentioned services do not offer transport flexibility, they only support TCP communication; they do not have other protocols such as UDP [108] and QUIC [8].

2.9 Summary

In this chapter, we overviewed technologies that are related to Babel [79]. In Section 2.1 we introduced the concept of network abstraction and its challenges; In Section 2.3 we talked about the basic transport protocols used in network abstractions (UDP, TCP and QUIC); In Section 2.5 we saw some libraries used to implement network abstractions (Netty [39], LibP2P [35] and ZeroMQ [59]); In Section 2.7 we overviewed technologies that offer distributed networking services (RabbitMQ [74], ActiveMQ [9] and Apache Kafka [65]), and the type of network abstraction they use; In Section 2.8 we discussed network applications (Apache Cassandra [17], Redis [41] and ZooKeeper [64]) and overviewed their network abstractions; and finally, in Section 2.6 we overviewed Babel [79] and similar frameworks (Appia [103], Yggdrasil [77] and Cactus [78]) for protocol implementations.

In the next chapter, we present our plan to implement a framework that will allow Babel [79] to support a wider range of protocols and execution environments. We also present the evaluations to be conducted, as well as the scheduling of the overall activities.

3 Future Work

As introduced in Chapter 1, this work consists of developing (and evaluating) new network abstractions that will allow Babel[79, 80] to support a wider range of distributed protocols and execution environments, paving the way to allow the interaction of protocols and systems developed using Babel to interact easily with other external systems such as Kafka, RabbitMQ and others. The main goals include creating the support, modeling, and implementing network stacks to allow the use of additional communication protocols (such as QUIC and UDP). This chapter aims at providing a proposal for the design of a library that will allow Babel to achieve these features.

Section 3.1 introduces the design of the proposed library which will be used to expand Babel; it discusses the technologies we intend to use to develop the library, and how Babel will be interacting with this library in order to interact with external systems. In Section 3.2 we discuss the evaluation methods and metrics we plan to use to evaluate the quality of the solution to be developed. Finally, in Section 3.3 we enumerate the tasks we will be doing to achieve our goal; we also include a gantt chart that shows when we intend to start and complete each task.

3.1 Proposed Solution

In this section, we present a solution for a library that will extend the communication channels of Babel[79, 80], allowing Babel to support more protocols, and integration with other distributed services, while keeping its architectural design model.

This library will support the development of new network abstractions that will allow Babel to support a broader range of communication protocols and including both transport and application specific protocols, which on one hand, allow systems and protocols developed in Babel to rely on a broader range of transport protocols , potentially some providing extra guarantees that are built at the level of the abstraction. The proposed solution consists of an event-driven library that will allow the specification of a variety of transport protocols with a diverse set of communication patterns for exchanging data over the network. The library will use ZeroMQ[59] and Netty[39] for the underlying

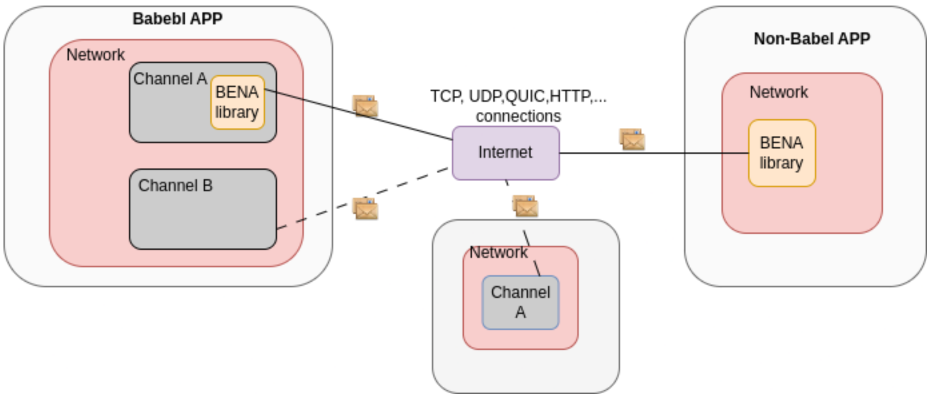


Figure 3.1: Architecure of BENA

network layer because they enable the easy development of distributed network systems and support a wide range of networking protocols[3, 61, 39, 40]. The goal is to provide a simple interface that empowers the developers to use these protocols without being concerned how these guarantees are indeed provided by this new Babel component; all the user will have to do is indicate the name of the protocols he or she wants to use; the user will also have the flexibility of implementing their user-level or application protocol. Assume a developer has a Babel application that needs to communicate with another application; the user will extend the proposed library in both applications and begin using them immediately after configuring the fundamentals, such as the address and the desired communication protocol; they will then invoke the respective API methods for sending and receiving messages. In Babel, the library will be exposed by a revised and improved Babel channel abstraction, this will allow Babel to keep its design pattern, as other components will not be aware of this change, they will continue interacting with channels in the same way they always did. The library will be called BENA, Babel Network Abstraction Extension. Figure 3.1 shows its architecture; in it, a Babel application (left side) is interacting with two other applications; it uses BENA to communicate with the non-Babel Application.

The API will be defined as we conduct our work; it will at least contain the following three methods: One for registering the protocol the user intends to use, as well as its properties; the methods for sending and receiving messages will internally use the implementations of the indicated protocol to perform the expected tasks.

3.2 Evaluation Method

In this section, we intend to present the methods and the metrics we will be using to assess the performance and quality of our proposed solution.

Our experiments will be run on a cluster scenario. The experiments will consist of implementing several case studies of protocols in Babel and in other different framework such as *Libp2p*, then having them interacting via the library we are proposing. The proposed protocols will exemplify several domains of relevance; they will include at least a peer-to-peer protocol for overlay management (e.g.: HyParView [94]), data dissemination (e.g.: Plumtree [93]), and a replicated system (e.g.: Paxos [90]).

In order to qualify the quality of our framework, we will be interested in observing the **reliability**, and the **latency** metrics, which we will then compare with the standalone implementations of the protocols in question. The reliability will refer to the fraction of the sent and delivered packets between the protocols; code quality will refer to the clarity of the code used to implement the networking layer between the protocols; and latency will be the amount of time between the moment a message is sent to the moment it is delivered by the destination.

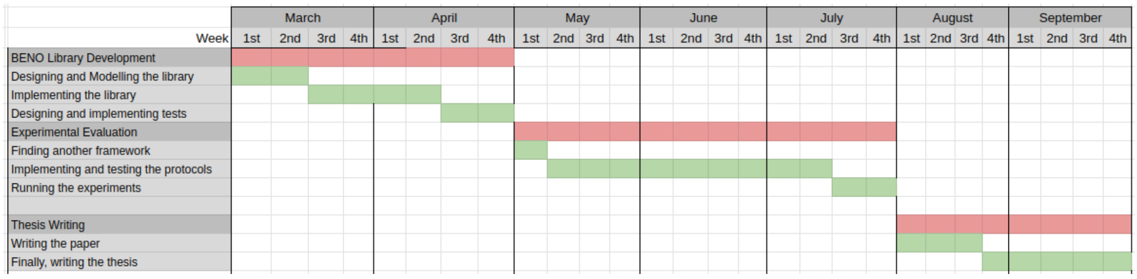
Regarding the results of the experiments, we expect that the results of our experiments obtains metric values that are very similar to the metric values of the standalone implementations of the protocols.

3.3 Scheduling

The tasks (as well as its gantt chart, Figure 3.2) that we will be doing in the next few months, during the development of this thesis are organized as follows:

1. BENO Library Development:
    a) Designing and Modelling the library;
    b) Implementing the library;
    c) Designing and implementing unit tests;
    d) Designing and implementing integration tests.
2. Experimental Evaluation:
    a) Implementing the protocols:
        i. Implementing a peer-to-peer overlay management protocol in Babel and in the found framework;
        ii. Implementing a peer-to-peer data diffusion protocol in Babel and in the found framework;
        iii. Implementing a peer-to-peer replicated system protocol in Babel and in the found framework;
        iv. Implementing a client-server application.
    b) Running the experiments and gathering metrics;
3. Thesis Writing:
    a) Writing the paper for the proposed solution to be submitted to the INForum 2023 symposium;

b) Finally, concluding the thesis.



	March				April				May				June				July				August				September			
Week	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th	1st	2nd	3rd	4th
BENO Library Development	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥																				
Designing and Modelling the library	🟩	🟩																										
Implementing the library			🟩	🟩	🟩	🟩																						
Designing and implementing tests							🟩	🟩																				
Experimental Evaluation									🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥								
Finding another framework									🟩																			
Implementing and testing the protocols										🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩	🟩										
Running the experiments																			🟩	🟩								
Thesis Writing																					🟥	🟥	🟥	🟥	🟥	🟥	🟥	🟥
Writing the paper																					🟩	🟩	🟩					
Finally, writing the thesis																								🟩	🟩	🟩	🟩	🟩

Figure 3.2: Work Plan.

Bibliography

[1] F. Adobe and G. community. “HTTP Semantics”. In: *https://www.rfc-editor.org/* (2022). DOI: https://www.rfc-editor.org/rfc/rfc9110.html (cit. on p. 25).

[2] A. AfterAcademy. “What is a TCP 3-way handshake process?” In: *https://afteracademy.com/blog* (2020). DOI: https://afteracademy.com/blog/what-is-a-tcp-3-way-handshake-process/ (cit. on p. 8).

[3] I. authors. “ZeroMQ Message Transport Protocol”. In: *https://rfc.zeromq.org/* (2023). DOI: https://rfc.zeromq.org/spec/23/ (cit. on pp. 16, 17, 33).

[4] I. authors. “ZeroMQ Message Transport Protocol”. In: *https://rfc.zeromq.org/* (2023). DOI: https://rfc.zeromq.org/spec/13/ (cit. on pp. 16, 17).

[5] M. T. Baldassarre et al. “Integrating security and privacy in software development”. In: *Software Quality Journal* 28 (2020), pp. 987–1018 (cit. on p. 4).

[6] S. Balne and G. Sindhu. “Network protocol Challenges of Internet of Things (IoT) Features-Review”. In: *International Journal of Innovative Research in Science, Engineering and Technology* 10.3 (2021), pp. 2305–2309 (cit. on p. 1).

[7] L. Bhatia. “Message Queues - A comprehensive overview”. In: *https://wiki.aalto.fi/* (2023). DOI: https://wiki.aalto.fi/download/attachments/116673303/message_queues.pdf?version=1&modificationDate=1479385770054&api=v2 (cit. on p. 13).

[8] chromium.org. “QUIC Protocol”. In: *https://www.chromium.org* (2023). DOI: https://www.chromium.org/quic/ (cit. on pp. 5, 9, 11, 12, 31).

[9] A. A. Community. “Apache ActiveMQ”. In: *https://activemq.apache.org/* (2023). DOI: https://activemq.apache.org/ (cit. on pp. 22, 25, 27, 31).

[10] A. A. Community. “Apache ActiveMQ”. In: *https://activemq.apache.org/* (2023). DOI: https://activemq.apache.org/topologies (cit. on p. 26).

[11] A. A. Community. “Apache ActiveMQ”. In: *https://activemq.apache.org/* (2023). DOI: https://activemq.apache.org/udp-transport-reference (cit. on p. 26).

[12] A. A. Community. "Apache ActiveMQ". In: *https://activemq.apache.org/* (2023). DOI: https://activemq.apache.org/uri-protocols (cit. on p. 26).

[13] A. A. Community. "Apache ActiveMQ". In: *https://activemq.apache.org/* (2023). DOI: https://activemq.apache.org/masterslave (cit. on p. 26).

[14] A. A. Community. "Apache ActiveMQ". In: *https://activemq.apache.org/* (2023). DOI: https://activemq.apache.org/failover-transport-reference (cit. on p. 26).

[15] A. A. Community. "Protocols". In: *https://activemq.apache.org/* (2023). DOI: https://activemq.apache.org/protocols (cit. on p. 25).

[16] A. C. Community. "Configurable Storage Ports and Why We Need Them". In: *https://cassandra.apache.org/$_/blog$* (2023). DOI: https://cassandra.apache.org/_/blog/Configurable-Storage-Ports-and-Why-We-Need-Them.html (cit. on pp. 27, 28, 31).

[17] A. C. Community. "What is Apache Cassandra?" In: *https://cassandra.apache.org* (2023). DOI: https://cassandra.apache.org/_/index.html (cit. on pp. 27, 28, 31).

[18] A. Community. "Message Queue Basics". In: *https://aws.amazon.com/* (2023). DOI: https://aws.amazon.com/message-queue/ (cit. on p. 13).

[19] A. Community. "MQTT". In: *https://aws.amazon.com/* (2023). DOI: https://aws.amazon.com/what-is/mqtt/ (cit. on pp. 12, 24, 25).

[20] A. Community. "Request-Reply". In: *https://docs.nats.io/nats-concepts/core-nats/* (2023). DOI: https://docs.nats.io/nats-concepts/core-nats/reqreply (cit. on p. 13).

[21] A. Community. "What is Pub/Sub Messaging?" In: *https://aws.amazon.com/* (2023). DOI: https://aws.amazon.com/pub-sub-messaging/ (cit. on p. 12).

[22] A. Community. "AMQP 1.0 in Azure Service Bus and Event Hubs protocol guide". In: *https://learn.microsoft.com/en-us/azure/* (2022). DOI: https://learn.microsoft.com/en-us/azure/service-bus-messaging/service-bus-amqp-protocol-guide (cit. on pp. 23, 24).

[23] C. Community. "What is the OSI Model". In: *https://www.cloudflare.com/learning* (2023). DOI: https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/ (cit. on pp. 3, 4).

[24] C. Community. "What Is NAT?" In: *https://www.comptia.org/* (2020). DOI: https://www.comptia.org/content/guides/what-is-network-address-translation (cit. on p. 11).

[25] C. Community. "distributed systems". In: *https://www.confluent.io/* (2022). DOI: https://www.confluent.io/learn/distributed-systems/ (cit. on p. 3).

[26] D. Community. “Architecture in brief”. In: *https://docs.datastax.com* (2023). DOI: https://docs.datastax.com/en/cassandra-oss/3.x/cassandra/architecture/archIntro.html (cit. on pp. 27, 28, 31).

[27] I. Community. “TCP flow control and the sliding window”. In: *https://www.ibm.com/docs/* (2013). DOI: https://www.ibm.com/docs/de/tsm/7.1.0?topic=tuning-tcp-flow-control-sliding-window (cit. on p. 7).

[28] I. Community. “What Is PMTUD?” In: *https://www.ibm.com/docs/* (2021). DOI: https://www.ibm.com/docs/fi/zvm/7.2?topic=terminology-what-is-pmtud (cit. on p. 9).

[29] I. Community. “What are libraries in programming?” In: *https://www.idtech.com/blog/* (2020). DOI: https://www.idtech.com/blog/what-are-libraries-in-coding (cit. on p. 13).

[30] I. Community. “Services Provided by IETF Transport Protocols and Congestion Control Mechanisms”. In: *https://www.rfc-editor.org/rfc/* (2017). DOI: https://www.rfc-editor.org/rfc/rfc8095.html#page-4 (cit. on pp. 5, 7–9).

[31] I. P. Community. “Libp2p”. In: *https://docs.ipfs.tech* (2022). DOI: https://docs.ipfs.tech/concepts/libp2p/ (cit. on p. 15).

[32] L. P. Community. “Implementations”. In: *https://libp2p.io/* (2022). DOI: https://libp2p.io/implementations/ (cit. on p. 14).

[33] L. P. Community. “WebRTC”. In: *https://docs.libp2p.io/* (2022). DOI: https://docs.libp2p.io/concepts/transports/webrtc/ (cit. on p. 15).

[34] L. P. Community. “WebTransport”. In: *https://docs.libp2p.io/* (0202). DOI: https://docs.libp2p.io/concepts/transports/webtransport/ (cit. on p. 15).

[35] L. P. Community. “What is libp2p”. In: *https://docs.libp2p.io/* (2022). DOI: https://docs.libp2p.io/concepts/introduction/overview/ (cit. on pp. 13–15, 18, 31).

[36] L. P. Community. “What is Stream Multiplexing”. In: *https://docs.libp2p.io/* (0202). DOI: https://docs.libp2p.io/concepts/multiplex/yamux/ (cit. on p. 15).

[37] L. P. Community. “What is Stream Multiplexing”. In: *https://docs.libp2p.io/* (2022). DOI: https://docs.libp2p.io/concepts/multiplex/overview/ (cit. on p. 15).

[38] L. P. Community. “What is Stream Multiplexing”. In: *https://docs.libp2p.io/* (2022). DOI: https://docs.libp2p.io/concepts/multiplex/mplex/ (cit. on p. 15).

[39] N. P. Community. “Netty”. In: *https://netty.io/* (2023-01). DOI: https://netty.io/ (cit. on pp. 13, 19, 31–33).

[40] N. P. Community. “Netty/Incubator/Codec/Quic 0.0.1.Final released”. In: *https://netty.io/* (2020). DOI: https://netty.io/news/2020/12/09/quic-0-0-1-Final.html (cit. on pp. 18, 33).

[41] R. Community. "Introduction to Redis". In: *https://redis.io/docs/* (2023). DOI: https://redis.io/docs/about/ (cit. on pp. 27, 28, 31).

[42] R. Community. "Introduction to Redis". In: *https://redis.io/docs/* (2023). DOI: https://redis.io/docs/reference/protocol-spec/ (cit. on pp. 27, 29, 31).

[43] R. Community. "Introduction to Redis". In: *https://redis.io/docs/* (2023). DOI: https://developer.redis.com/operate/redis-at-scale/high-availability/basic-replication/ (cit. on p. 28).

[44] R. Community. "Introduction to Redis". In: *https://redis.io/docs/* (2023). DOI: https://redis.io/docs/manual/client-side-caching/ (cit. on p. 28).

[45] R. Community. "Introduction to Redis". In: *https://redis.io/docs/* (2023). DOI: https://redis.io/docs/manual/pubsub/ (cit. on p. 28).

[46] R. Community. "Introduction to Redis". In: *https://redis.io/docs/* (2023). DOI: https://redis.io/docs/manual/transactions/ (cit. on p. 28).

[47] R. Community. "Introduction to Redis". In: *https://redis.io/docs/* (2023). DOI: https://redis.io/docs/manual/pipelining/ (cit. on p. 29).

[48] R. Community. "Redis cluster specification". In: *https://redis.io/docs/* (2023). DOI: https://redis.io/docs/reference/cluster-spec/ (cit. on pp. 27, 29, 31).

[49] R. Community. "Scaling with Redis Cluster". In: *https://redis.io/docs/* (2023). DOI: https://redis.io/docs/management/scaling/ (cit. on p. 29).

[50] R. Community. "INTERNET CONTROL MESSAGE PROTOCOL". In: *https://www.rfc-editor.org/* (1981). DOI: https://www.rfc-editor.org/rfc/rfc792 (cit. on p. 6).

[51] R. Community. "INTERNET PROTOCOL". In: *https://www.rfc-editor.org/* (1981). DOI: https://www.rfc-editor.org/rfc/rfc791 (cit. on pp. 5, 6).

[52] R. Community. "The Transport Layer Security (TLS) Protocol". In: *https://www.rfc-editor.org/* (1981). DOI: https://www.rfc-editor.org/rfc/rfc5246 (cit. on p. 22).

[53] R. community. "The WebSocket Protocol". In: *https://www.rfc-editor.org/* (2022). DOI: https://www.rfc-editor.org/rfc/rfc6455#page-5 (cit. on p. 15).

[54] S. Community. "Network service". In: *https://web.archive.org* (2015). DOI: https://web.archive.org/web/20150612134130/https://www.sdxcentral.com/term/network-service/ (cit. on p. 21).

[55] W. Community. "Messaging pattern". In: *https://en.wikipedia.org/* (2023). DOI: https://en.wikipedia.org/wiki/Messaging_pattern (cit. on pp. 3, 12).

[56] W. Community. "Network service". In: *https://en.wikipedia.org/* (2023). DOI: https://en.wikipedia.org/wiki/Network_service#cite_note-SDxCentral,_2015-1 (cit. on p. 21).

[57] W. Community. “OSI Model”. In: *https:https://en.wikipedia.org/* (2023). DOI: https://en.wikipedia.org/wiki/OSI_model#Layer_4:_Transport_layer (cit. on p. 5).

[58] W. Community. “Transport layer protocols”. In: *https://en.wikipedia.org/* (2016). DOI: https://en.wikipedia.org/wiki/Category:Transport_layer_protocols (cit. on p. 5).

[59] Z. P. Community. “Get Started”. In: *https://zeromq.org/* (2023). DOI: https://zeromq.org/get-started/ (cit. on pp. 13, 16, 31, 32).

[60] Z. P. Community. “ØMQ - The Guide”. In: *https://zguide.zeromq.org/* (2023). DOI: https://zguide.zeromq.org/docs/chapter3/ (cit. on p. 16).

[61] Z. P. Community. “ØMQ - The Guide”. In: *http://api.zeromq.org/* (2023). DOI: http://api.zeromq.org/master:zmq-udp (cit. on p. 33).

[62] Z. Community. “ZooKeeper Index”. In: *https://cwiki.apache.org/* (2023). DOI: https://cwiki.apache.org/confluence/display/ZOOKEEPER/Index (cit. on pp. 27, 29, 31).

[63] Z. Community. “ZooKeeper Internals”. In: *https://zookeeper.apache.org/* (2023). DOI: https://zookeeper.apache.org/doc/r3.4.13/zookeeperInternals.html#sc_logging (cit. on pp. 27, 29–31).

[64] Z. Community. “ZooKeeper Overview”. In: *https://zookeeper.apache.org/* (2023). DOI: https://zookeeper.apache.org/doc/current/zookeeperOver.html (cit. on pp. 1, 27, 30, 31).

[65] A. K. Communnity. “Documentation”. In: *https://kafka.apache.org/* (2023). DOI: https://kafka.apache.org/documentation/ (cit. on pp. 1, 22, 26, 27, 31).

[66] A. K. Communnity. “KAFKA PROTOCOL GUIDE”. In: *https://kafka.apache.org/* (2023). DOI: https://kafka.apache.org/protocol.html (cit. on pp. 26, 27).

[67] A. K. Communnity. “KAFKA PROTOCOL GUIDE”. In: *https://kafka.apache.org/* (2023). DOI: https://kafka.apache.org/documentation/#replication (cit. on p. 27).

[68] A. K. Communnity. “The consumer”. In: *https://kafka.apache.org/* (2023). DOI: https://kafka.apache.org/documentation/#theconsumer (cit. on p. 27).

[69] A. K. Communnity. “The producer”. In: *https://kafka.apache.org/* (2023). DOI: https://kafka.apache.org/documentation/#theproducer (cit. on p. 27).

[70] M. Communnity. “MQTT”. In: *https://mqtt.org/* (2023). DOI: https://mqtt.org/ (cit. on pp. 24, 25).

[71] R. Communnity. “AMQP 0-9-1 Model Explained”. In: *https://www.rabbitmq.com/* (2022). DOI: https://github.com/cloudamqp/amqproxy (cit. on p. 22).

[72] R. Communnity. "AMQP 0-9-1 Model Explained". In: *https://www.rabbitmq.com/* (2022). DOI: https://www.rabbitmq.com/connections.html (cit. on p. 22).

[73] R. Communnity. "AMQP 0-9-1 Model Explained". In: *https://www.rabbitmq.com/* (2022). DOI: https://www.rabbitmq.com/tutorials/amqp-concepts.html (cit. on pp. 22, 23, 27).

[74] R. Communnity. "RabbitMQ". In: *https://www.rabbitmq.com/* (2022). DOI: https://www.rabbitmq.com/ (cit. on pp. 1, 22, 27, 31).

[75] R. Communnity. "Which protocols does RabbitMQ support?" In: *https://www.rabbitmq.com/* (2022). DOI: https://www.rabbitmq.com/protocols.html (cit. on pp. 22–24, 27).

[76] S. Communnity. "STOMP". In: *https://www.cloudamqp.com/* (2022). DOI: https://www.cloudamqp.com/docs/stomp.html (cit. on pp. 24, 27).

[77] P. Á. Costa, A. Rosa, and J. Leitão. "Enabling wireless ad hoc edge systems with yggdrasil". In: (2020), pp. 2129–2136 (cit. on pp. 20, 21, 31).

[78] S. M. de la Cruz. "Protocol Composition Frameworks and Modular Group Communication: Models, Algorithms and Architectures. Chapeter 2: 2.3.2 Cactus and the x-kernel". In: (2006) (cit. on pp. 21, 31).

[79] P. Fouto et al. "Babel: A Framework for Developing Performant and Dependable Distributed Protocols". In: *arXiv preprint arXiv:2205.02106* (2022) (cit. on pp. 1–3, 18–21, 27, 31, 32).

[80] P. Fouto et al. "Babel: A Framework for Developing Performant and Dependable Distributed Protocols". In: *arXiv preprint arXiv:2205.02106* (2022) (cit. on pp. 1–3, 18, 19, 27, 32).

[81] J. Fruehe. "How do network virtualization and network abstraction compare?" In: *www.techtarget.com* (2019). DOI: https://www.techtarget.com/searchnetworking/answer/How-do-network-virtualization-and-network-abstraction-compare (cit. on p. 3).

[82] HashiCorp. "What is Stream Multiplexing". In: *https://docs.libp2p.io/* (2016). DOI: https://github.com/hashicorp/yamux/blob/master/spec.md (cit. on p. 15).

[83] P. Hintjens. "ZeroMQ Publish-Subscribe". In: *https://rfc.zeromq.org/* (2023). DOI: https://rfc.zeromq.org/spec/29/ (cit. on p. 17).

[84] P. Hintjens. "ZeroMQ Request-Reply". In: *https://rfc.zeromq.org/* (2023). DOI: https://rfc.zeromq.org/spec/28/ (cit. on p. 18).

[85] P. Hintjens. "ZeroMQ Request-Reply". In: *https://rfc.zeromq.org/* (2023). DOI: https://rfc.zeromq.org/spec/30/ (cit. on p. 18).

[86] N. Hutchinson and L. Peterson. "The x-Kernel: an architecture for implementing network protocols". In: *IEEE Transactions on Software Engineering* 17.1 (1991), pp. 64–76. DOI: 10.1109/32.67579 (cit. on p. 21).

[87] J. Iyengar and M. Thomson. *QUIC: A UDP-Based Multiplexed and Secure Transport*. RFC 9000. 2021-05. DOI: 10.17487/RFC9000. URL: https://www.rfc-editor.org/info/rfc9000 (cit. on pp. 11, 12).

[88] L. Kalita. "Socket programming". In: *International Journal of Computer Science and Information Technologies* 5.3 (2014), pp. 4802–4807 (cit. on p. 6).

[89] S. Kumar, S. Dalal, and V. Dixit. "The OSI model: Overview on the seven layers of computer networks". In: *International Journal of Computer Science and Information Technology Research* 2.3 (2014), pp. 461–466 (cit. on p. 4).

[90] L. Lamport. "Paxos made simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58 (cit. on p. 34).

[91] A. Langley et al. "The quic transport protocol: Design and internet-scale deployment". In: *Proceedings of the conference of the ACM special interest group on data communication*. 2017, pp. 183–196 (cit. on pp. 2, 4, 10–12).

[92] G. Lee. *Cloud networking: Understanding cloud-based data center networks*. Morgan Kaufmann, 2014 (cit. on p. 10).

[93] J. Leitao, J. Pereira, and L. Rodrigues. "Epidemic broadcast trees". In: *2007 26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*. IEEE. 2007, pp. 301–310 (cit. on p. 34).

[94] 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)*. IEEE. 2007, pp. 419–429 (cit. on p. 34).

[95] D. M. Lentz. "ActiveMQ architecture and key metrics". In: *https://www.datadoghq.com/blog/* (2021). DOI: https://www.datadoghq.com/blog/activemq-architecture-and-metrics/ (cit. on p. 25).

[96] S. Lewis. "Interoperability". In: *https://www.techtarget.com/* (2022). DOI: https://www.techtarget.com/searchapparchitecture/definition/interoperability (cit. on p. 1).

[97] N. Lincs. "Nova FCT home page". In: *https://www.fct.unl.pt/* (2023). DOI: https://www.fct.unl.pt/ (cit. on p. 3).

[98] N. Lincs. "Nova Lincs Home page". In: *https://nova-lincs.di.fct.unl.pt/* (2023). DOI: https://nova-lincs.di.fct.unl.pt/ (cit. on p. 3).

[99] B. Lutkevich. "distributed applications (distributed apps)". In: *https://www.techtarget.com/* (2022). DOI: https://www.techtarget.com/searchitoperations/definition/distributed-applications-distributed-apps (cit. on pp. 3, 27).

[100] I. M. Duke F5 Networks. "Network Address Translation Support for QUIC". In: *https://www.ietf.org/* (2020). DOI: https://www.ietf.org/archive/id/draft-duke-quic-natsupp-03.html (cit. on p. 12).

[101] N. Maurer and M. Wolfthal. *Netty in action*. Simon and Schuster, 2015 (cit. on pp. 13, 14).

[102] S. Mena et al. “Appia vs. Cactus: comparing protocol composition frameworks”. In: *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.* 2003, pp. 189–198. DOI: 10.1109/RELDIS.2003.1238068 (cit. on pp. 19–21).

[103] H. Miranda, A. Pinto, and L. Rodrigues. “Appia, a flexible protocol kernel supporting multiple coordinated channels”. In: *Proceedings 21st International Conference on Distributed Computing Systems*. 2001, pp. 707–710. DOI: 10.1109/ICDSC.2001.919005 (cit. on pp. 19–21, 31).

[104] P. Moll et al. “Resilient Brokerless Publish-Subscribe over NDN”. In: *MILCOM 2021-2021 IEEE Military Communications Conference (MILCOM)*. IEEE. 2021, pp. 438–444 (cit. on p. 13).

[105] J. Onisick. “Why We Need Network Abstraction”. In: *www.networkcomputing.com* (2012). DOI: https://www.networkcomputing.com/networking/why-we-need-network-abstraction (cit. on p. 4).

[106] E. Rescorla. *Diffie-hellman key agreement method*. Tech. rep. 1999 (cit. on p. 10).

[107] RFC. “TRANSMISSION CONTROL PROTOCOL”. In: *www.networkcomputing.com* (1981). DOI: https://www.ietf.org/rfc/rfc793.txt (cit. on pp. 2, 4–8, 12).

[108] J. P. RFC. “User Datagram Protocol”. In: *https://www.rfc-editor.org/* (2023). DOI: https://www.rfc-editor.org/rfc/rfc768 (cit. on pp. 2, 4, 5, 8, 12, 31).

[109] M. Seemann, M. Inden, and D. Vyzovitis. “Decentralized Hole Punching”. In: *2022 IEEE 42nd International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE. 2022, pp. 96–98 (cit. on p. 15).

[110] O. Standard. “Oasis advanced message queuing protocol (amqp) version 1.0”. In: *International Journal of Aerospace Engineering Hindawi www. hindawi. com* 2018 (2012) (cit. on pp. 23, 24).

[111] M. Sústrik et al. “ZeroMQ”. In: *Introduction Amy Brown and Greg Wilson* (2015) (cit. on p. 16).

[112] G. Wang et al. “Building a replicated logging system with Apache Kafka”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1654–1655 (cit. on p. 27).

[113] P. Wegner. “Interoperability”. In: *ACM Computing Surveys (CSUR)* 28.1 (1996), pp. 285–287 (cit. on p. 1).

[114] I. Wigmore. “How do network virtualization and network abstraction compare?” In: *www.techtarget.com* (2021). DOI: https://www.techtarget.com/whatis/definition/abstraction (cit. on p. 3).

[115] WIkipedia. “QUIC Protocol”. In: *https://en.wikipedia.org/* (2023). DOI: https://en.wikipedia.org/wiki/QUIC (cit. on p. 9).

[116] J. M. Winett. *Definition of a socket*. Tech. rep. 1971 (cit. on p. 6).

[117] J. Zhang et al. "Formal Analysis of QUIC Handshake Protocol Using Symbolic Model Checking". In: *IEEE Access* 9 (2021), pp. 14836–14848. DOI: 10.1109/ACCESS.2021.3052578 (cit. on pp. 5, 9, 11, 12).