DEPARTMENT OF
COMPUTER SCIENCE

DANIEL DE ALMEIDA JOÃO

BSc Computer Engineering

NETWORK ABSTRACTIONS AND EMULATION FOR DISTRIBUTED SYSTEMS

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2023

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

NETWORK ABSTRACTIONS AND EMULATION FOR DISTRIBUTED SYSTEMS

DANIEL DE ALMEIDA JOÃO

BSc Computer Engineering

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

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2023

Network abstractions and emulation for distributed systems

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.10) [64].

Acknowledgements

I would like to thank professor Joao Leitao for constantly helping me throughout the development of this thesis. I also want to thank Pedro Fouto, Pedro Ákos Costa, Diogo Paulico, David Antunes, and Diogo Barreto for their constant help in developing, testing and writing this final work.

Thanks to all the people who helped me reach the end.

"You cannot teach a man anything; you can only help him discover it in himself." (Galileo)

Abstract

Babel is a Java framework which simplifies the development of distributed applications. Babel is divided into two layers, the layer of the protocols and the network layer. The layer of the protocols contains the modules that encode the logic for application level protocols, such as data handling and data processing; and the network layer is the abstraction responsible for moderating the access to the network to the protocols, so that they can exchange data over the network. However, the network abstraction, characterized into channels, is not very flexible, as it only supports TCP-based type of communication. This is a limitative feature because there are more transport protocols which can be used depending on the requirements and specifications of the applications being built. We could want an application which requires neither the establishment of a connection nor reliability, such as the features offered by UDP; or an application that requires a network protocol such as QUIC that offers security and at the same time allows us to multiplex multiple data streams within a single connection, enabling concurrent data transfer.

In this work, we present a solution to mitigate this limitation of Babel, which consisted in developing three networking channels that support the TCP, UDP, and QUIC transport protocols. We also present the experimental evaluation and validation, obtained with case studies implementation of a peer-to-peer protocol for overlay management and data diffusion, a replicated system, and a simple file streaming application.

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

Resumo

Babel é uma framework para Java que simplifica o desenvolvimento de aplicações distribuídas. Babel é dividido em duas camadas, a camada dos protocolos e a camada de redes. A camada dos protocolos contém os módulos que codificam a lógica para protocolos do nível da aplicação, como manipulação e processamento de dados; e a camada de rede é a abstração responsável por moderar o acesso da rede aos protocolos, para que eles possam trocar dados pela rede. Porém, a abstração da rede, caracterizada por canais, não é muito flexível, pois suporta apenas o tipo de comunicação baseada em TCP. Esta é uma característica limitante porque há mais protocolos de transporte que podem ser usados dependendo dos requisitos e especificações das aplicações que estão a ser construídas. Poderíamos querer uma aplicação que não exija o estabelecimento de conexão nem confiabilidade, como as features oferecidas pelo UDP; ou uma aplicação que requer um protocolo de rede como o QUIC que oferece segurança e, ao mesmo tempo, nos permite multiplexar vários fluxos de dados numa única conexão, permitindo a transferência simultânea de dados.

Neste trabalho apresentamos uma solução para mitigar esta limitação do Babel, que consistiu no desenvolvimento de três canais de rede que suportam os protocolos de transporte como TCP, UDP e QUIC. Apresentamos também a avaliação e validação experimental, obtida com estudos de caso de implementação de um protocolo peer-to-peer para gerenciamento de sobreposição e difusão de dados, um sistema replicado e uma aplicação simples de streaming de ficheiros.

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

Contents

List of Figures x

1 Introduction 1
- 1.1 Problem Statement . . . 1
- 1.2 Contributions . . . 2
- 1.3 Document Structure . . . 2

2 Related Work 4
- 2.1 Network Abstractions . . . 4
- 2.2 The OSI Model . . . 5
- 2.3 Transport Protocols . . . 6
  - 2.3.1 Internet Protocol . . . 6
  - 2.3.2 Network Sockets . . . 7
  - 2.3.3 TCP . . . 8
  - 2.3.4 UDP . . . 10
  - 2.3.5 QUIC . . . 10
  - 2.3.6 Discussion . . . 13
- 2.4 Messaging Patterns . . . 13
  - 2.4.1 Discussion . . . 14
- 2.5 Communication Libraries . . . 14
  - 2.5.1 Netty . . . 15
  - 2.5.2 Libp2p . . . 16
  - 2.5.3 ZeroMq . . . 17
  - 2.5.4 Discussion . . . 20
- 2.6 Frameworks . . . 21
  - 2.6.1 Babel . . . 21
  - 2.6.2 Appia . . . 23
  - 2.6.3 Yggdrasil . . . 23
  - 2.6.4 Cactus . . . 24

2.6.5 Discussion . . . 25
2.7 Summary . . . 25

3 Distributed Networking Channels for Babel **26**
3.1 Context . . . 26
3.2 Connection-Oriented channels . . . 27
3.2.1 TCP Channels . . . 29
3.2.2 QUIC Channels . . . 31
3.2.3 Summary . . . 33
3.3 UDP Channel . . . 33
3.3.1 Summary . . . 35
3.4 Developed Channels . . . 35
3.4.1 Overview of the Channels and their roles . . . 35
3.4.2 How to create the Channels . . . 36
3.4.3 Interaction With Babel Protocols . . . 37
3.4.4 Operations of the Channels . . . 37
3.4.5 Stream Operations . . . 40
3.4.6 Metrics and Error handling . . . 41
3.4.7 Changes made to Babel protocols . . . 42
3.5 Summary . . . 43

4 Experimental Evaluation **45**
4.1 Experimental Settings . . . 45
4.2 Peer-to-Peer Overlay Management Experiments . . . 45
4.2.1 Dissemination rate of 1 message per second . . . 46
4.2.2 Dissemination rate of 1 message per half second . . . 47
4.2.3 Summary . . . 48
4.3 State Machine Replication Experiments (SMR) . . . 49
4.3.1 Experiments without Linux Traffic Control tool . . . 50
4.3.2 Experiments with Linux Traffic Control . . . 51
4.3.3 Summary . . . 52
4.4 Stream Diffusion Experiments . . . 52
4.4.1 Environment setup . . . 53
4.4.2 TCP Streams . . . 53
4.4.3 QUIC Streams . . . 54
4.4.4 Summary . . . 56
4.5 Performance and memory analysis . . . 56
4.5.1 Performance Analysis . . . 56
4.5.2 Memory Analysis with Jprofiler . . . 59
4.5.3 Summary . . . 61
4.6 Discussion . . . 61

4.7 Summary 62

5 Final Remarks **63**

5.1 Conclusions 63

5.2 Future Work 64

Bibliography **65**

List of Figures

2.1 Protocol Relationships. Extracted from [40]. 7
2.2 Protocol Layering [87]. 8
2.3 Basic 3-Way Handshake for Connection Synchronization [10] [87]. 9
2.4 QUIC connection establishment. Extracted from [50]. 11

3.1 New network abstractions for Babel 26

4.1 Overlay graphs with one sec dissemination 47
4.2 Overlay graphs with half-sec dissemination 48
4.3 Paxos Part 1 Experiments 51
4.4 Paxos Part 2 Experiments 52
4.5 Streams vs Messages with 6 and 10 clients 55
4.6 Streams vs Messages with 16 clients 56
4.7 sendMessage call trees 58

List of Listings

3.1 Registering the Channels 36
3.2 Registering metrics 42
3.3 Open Connection Failed 43

1 Introduction

As the Internet continues to grow and more data is produced, it is becoming increasingly relevant to enable developers to create high-performant distributed applications within a short period of time; and these applications must also be able to use a variety of transport protocols [94, 54]. 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 [54].

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 logic. 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 effects of network partitions, different network configurations, data integrity, packet loss, and many other aspects [7].

1.1 Problem Statement

Babel [26, 27] 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 handling low level aspects of communication over sockets. Babel has a network abstraction which allows the application running at the upper layers to exchange network messages over the network. However, its network abstraction only offers network communication through the use of a single network transport protocol, the Transmission Control Protocol (TCP). This is a

limitative feature because there are more networking protocols which can be used depending on the requirements and specifications of the networking application we are building. We could want an application which requires neither the establishment of a connection nor the reliability of data delivery, such as the features presented by the UDP transport protocol [88]; or an application that requires a network protocol such as QUIC [50] that offers security, delivery guarantees, and at the same time allows us to multiplex multiple streams within a single connection, enabling concurrent data transfer.

Even though TCP is a stream-oriented protocol, the channels of Babel do not support the sending and delivery of streams of bytes, only messages, which consists in sending the desired data together with the length of the data, so that the receiver can use the length to pile up the bytes into a message. They also require that two Babel processes use no more than two connections (one in each direction) to exchange data, adding a performance and scalability constraint for Babel applications that require each Babel protocol to use its own separated and independent TCP connection.

1.2 Contributions

To tackle the problems mentioned above, we developed network channels to enrich the current networking layers of Babel. These channels support TCP [87], UDP [88], and QUIC [44] transport protocols. The current channels of Babel only support TCP, but with several limitations, so we developed new TCP channels without the limitations that are present in the current channels of Babel. The TCP channels we develop allow streams, and multiple concurrent connections between two Babel applications. And the UDP channel supports the raw features of UDP such as connectionless communication, no message delivery guarantee, and broadcasting on local networks. However, we also implemented a form of message acknowledgement (ACK) to let the sender know that the message sent was indeed received; this consists in having a timer to resend a message if the respective ACK is not received before the expiration of the timer. Additionally, the QUIC channel offers the QUIC transport layer protocol, and it offers reliability, security, stream multiplexing and other features QUIC offers.

The main contribution of this work are:

- A framework that allows Babel [26, 27] to support new network abstractions, with a wider range of transport protocols.
- An experimental evaluation of this new family of network channels using different use cases.

1.3 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, and distributed frameworks. These are technologies which are similar to Babel, and can be used to extend the channels of Babel.
- Chapter 3 presents a proposed solution of network abstractions that provides additional flexibility to Babel[26, 27], allowing it to support more transport protocols.
- Chapter 4 presents the results of the validation and evaluation of the developed solutions.
- Chapter 5 concludes the dissertation with some final remarks and future works.

2

Related Work

The goal of this chapter is to highlight some technologies that share common features with Babel [26, 27] as well as some aspects that Babel does not support, and would be of our interest to have Babel supporting them. Babel is a recent framework, created by NOVA LINCS [79] at NOVA FCT [62], 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 [26].

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 [13]; we 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 [96]; communication libraries, Section 2.5, offer a set of methods and functions to implement communication abstractions for the network; we talk about some frameworks, in Section 2.6, which provide support for building distributed applications.

2.1 Network Abstractions

The main purpose of a network abstraction is to hide the complex mechanisms and details of the network [95], by offering additional flexibility [28] and a very generalized interface through which the developer delegates the task of exchanging data between different processes. 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 [80], 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 [6].

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 [49].

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 [49, 13].

Presentation Layer This layer formats the data from the sending layer (application or session layer) to be understood by the next layer [49, 13].

Session Layer It is responsible for establishing connection between the two interested parties [49, 13].

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 [49, 13, 87, 88, 50]. 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 [49, 13].

Data Link Layer Performs error detection and control on the data [49, 13].

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 [87], UDP [88], and QUIC [100]. 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 distributed protocol implementation and distributed systems programming.

2.3 Transport Protocols

These are the Transport Layer protocols that provide procedural methods on how to exchange data over the network [49, 20]. These protocols are differentiated by the features they provide, such as reliable delivery, ordered delivery, content privacy, and integrity protection [49, 20]. Moreover, there are a variety of protocols [49, 20], 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, among others. In this dissertation, we are only interested in TCP and UDP, which are the most popular ones, and QUIC [12] which is a more recent proposal that is effectively replacing TCP at least in the context of applications using HTTP.

We start by discussing the Internet Protocol (IP) over which these transport protocols usually operate.

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 [40].

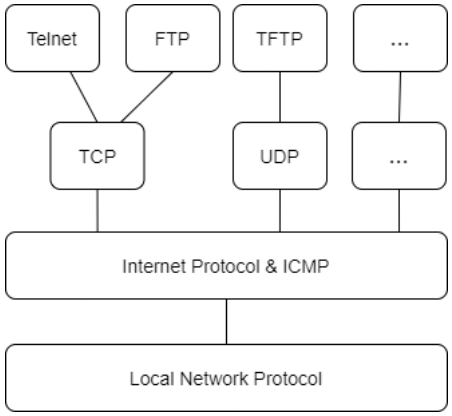


Figure 2.1: Protocol Relationships. Extracted from [40].

The Figure 2.1 shows the relationship among the many protocols that operate on a network, for example, the FTP protocol relies on TCP for reliable communication, and the TCP protocol also relies on the IP and the ICMP to send and receive data over the network. The 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 [43]. In this context, a "local network" may be a small network in a building or a large network such as the ARPANET [40].

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 [97, 48]. 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 [87, 97, 48].

2.3.3 TCP

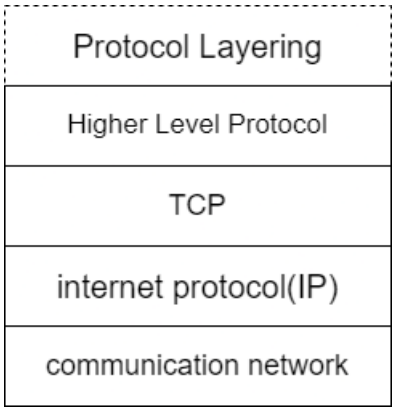


Figure 2.2: Protocol Layering [87].

TCP is a transport layer protocol that aims to provide a reliable process-to-process communication service in a multi-network environment [87]. 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 packets.

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 [87, 20]. 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 [87].

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 [87, 20].

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 [87, 20, 36].

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 [87, 20].

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. [87]. 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 [87, 20].

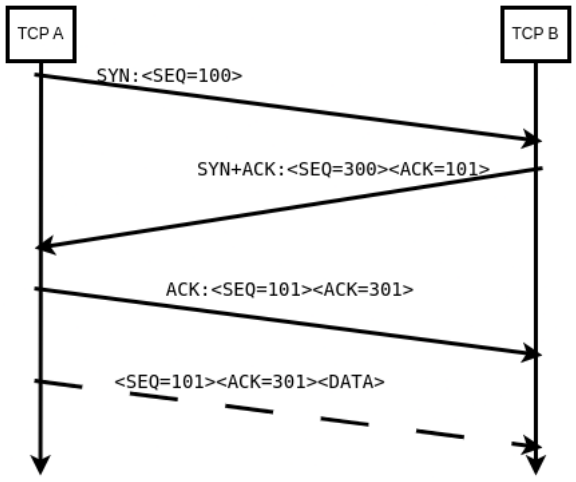


Figure 2.3: Basic 3-Way Handshake for Connection Synchronization [10] [87].

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 [88, 20] .

Protocol Description UDP is a connectionless protocol that maintains message boundaries[1], with no connection setup or feature negotiation. The protocol uses independent messages, usually 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 [20].

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 [20].

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 [20].

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 [20]..

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. QUIC enhances the performance of TCP-based connection-oriented web applications; it does this by establishing a number

[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" [37]

of multiplex connections between endpoints using User Data Protocol (UDP), and is designed to obsolete TCP at the transport layer for many applications [12, 100].

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 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 [86] 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. [50] Figure 2.4 shows how the connection takes place.

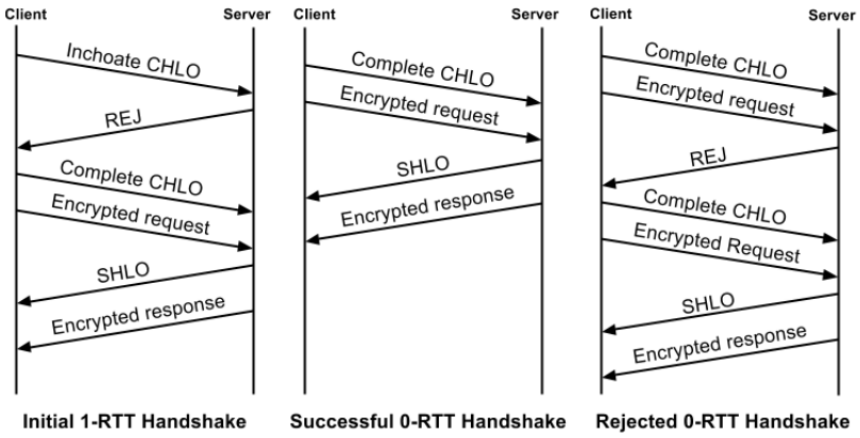


Figure 2.4: QUIC connection establishment. Extracted from [50].

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 [51]). 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 [50, 42].

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. [12, 50].

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 [100].

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 has changed its address, and starts sending data to the client using the new address [50, 65, 42].

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 [87] is a protocol widely used for its reliability guarantees, while UDP [88] is used mainly when great performance is the goal. On the other hand, QUIC [12, 100] 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 [96].

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.

[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. [14]

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. [5] 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) [3]. However, there can be a publish/subscribe system without a broker [72], 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 [2].

Push/Pull In this pattern, client publishers push data to the server and client consumers pull the data, consumers explicitly ask for the data [9].

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 [4].

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 [8]. Netty [77], Libp2p [58] and ZeroMQ [98] 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 ,QUIC, and TCP [77, 68, 22]. It is also the library that was used to develop the current TCP channels of Babel, and the library we also used to implement our solution.

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 [68].

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 [68].

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 [68].

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 [68].

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 [68].

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 [68].

Transports They refer to the way data is moved around the network. Natty offers several ready APIs to use transports, such as NIO, OIO, Epoll, and Local. NIO (non-blocking I/O) and OIO (old blocking I/O) are 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;

and Local is 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 [68].

2.5.1.1 How to set up a client-server application

It is very simple to set a client-server application in Netty, both the client and the server entity require two components to work: At least a *ChannelHandler* and a *Bootstrapping*. A *ChannelHandler* is a component that has the logics of how to handle the received data, it is the busisness logic. The *Bootstrapping* is the configuration of the server, and the client entities. They bind the port on which the server will listen for connection requests, and the port the client will use for establishing a connection; they also register the *ChannelHandler* elements[69].

ChannelHandler The developer must have a Java custom class that extends the various *ChannelHandler* implementation classes of Netty; when extending these classes, the methods that interest the most are the *channelActive*, *channelInactive*, *channelRead*, and *exceptionCaught*. THe channelActive method is the method triggered when a connection is established, this where the developer will implement the logics to deal with the establishment of new connections. The *channelInactive* method is called when a connection is closed. The *channelRead* is called whenever there is data to be read from the network, and it comes with a buffer containing the data bytes; this is where the developer implements the logics of how to handle the data. The *exceptionCaught* is the method triggered when there is an error, such as sending data to a closed connection. Moreover, the developer can register more than one *ChannelHandler*. The events are passed to all channel handlers in the order they were registered. They can also be deleted[69].

Bootstrapping Is the configuration of the server and the client entity. There are two types of bootstrapping instances, the server and the client instances, which can also be divided in terms of the transport protocols they use, such as TCP and UDP. They allow us to register the number of threads to use to handle network events, the local address, and the channel handlers[69].

2.5.2 Libp2p

Libp2p is a peer-to-peer (P2P) networking library that facilitates the development of P2P applications [58]. It is a collection of protocols, libraries, and specifications that make it easier to establish P2P communication between network participants [58]. It currently offers usable implementations for languages such as GO, Rust, JavaScript, JVM, Nim, *C* + +, Swift, Python, and Erlang [55].

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 [41].

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 [58, 41, 56] Libp2p also offers stream multiplexing [60], which allows different processes to share a connection by using unique port numbers to distinguish the streams [60]; In the case of TCP connections [90], it uses stream multiplexers such as yamux [30, 59] and mplex [61]; for QUIC [90], 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 [23].

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 [57].

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. [98] [99]

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 [92].

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 [38, 39].

2.5.3.1 ZMTP

The ZMTP protocol is made of three layers: the framing, the connection, and the content layer [38, 39].

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 [38, 39].

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 [38, 39].

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 [38, 39].

Interoperability . When version detection is enabled, the peers must send their versions after the greeting [38, 39].

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 [32].

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 [32].

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 [32].

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 [33].

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 [33].

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 [33].

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 [31].

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 [78, 58], 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 [26, 27] 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 [70].

2.6.1 Babel

Babel [26, 27] 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 [26, 27].

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. Moreover, the experiments show that the protocols implemented with Babel tend to show performances close to the optimized versions of competing implementations of those protocols [26].[26, 27].

2.6.1.1 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, they have the implementation for establishing network connections, and sending messages over a network. The channels use the network features offered by Netty [77] to achieve network connectivity. Moreover, the protocols interact with the channels using *openConnection*, *sendMessage*, and *closeConnection* operations. And the other interactions consist in receiving events, such as messages from the network. A channel can be used by more than one Babel protocol inside the same process, and a Babel protocol can also use more than one channel.

The framework of Babel provides three TCP channels which are intended for TCP communication between babel processes. The framework is also flexible, as it allows the developers to design their own channels [26].

The three channels of Babel are: *TCPChannel*, *SimpleClientChannel*, and *SimpleServerChannel*[24].

TCPChannel Is a peer-to-peer channel that acts both as a client and as a server simultaneously. It serves to initiate connections with other Babel processes while also accepting connection requests from them. Internally, this channel utilizes the client setup provided by Netty for configuring a client entity to establish connections. And to allow it to listen for incoming connections, it uses the server configurations provided by Netty.

This channel maintains a set of active connections, mapping them to the addresses of the remote Babel processes. This enables Babel protocols to exchange messages with multiple remote Babel processes. However, it has a limitation: It permits only a maximum of two connections between two Babel applications. It is important to note that, like other Babel channels, this channel supports messages only.

ClientChannel Is a channel with a network role of a client, it initiates connections, it does not wait for connection requests. The Babel protocols using this channel will always have to initiate interaction to be able to exchange messages with other Babel processes. This channel uses the client configurations offered by Netty. This channel only allows the protocols of Babel to exchange messages with one server at a time, it only keeps the connection of one server, it is the most recently established connection.

ServerChannel Is a channel that acts only as a server. It can not start connections, can only listen for incoming connection requests and accept them. This channel uses the server set up configurations offered by Netty. This channel keeps a set with all active client connections, allowing the protocols of Babel to exchange messages with various Babel clients at the same time, and concurrently.

Serializers and Deserializers These are class objects in Babel that allow the channels to write the data from the protocols of Babel to the network, and vice versa, reading the data from the network to the protocols of Babel. Every message in the protocols of Babel is a Java object which extends the class *ProtoMessage* and must have a field of type *ISerializer*, amongst other fields representing the effective data. The *ISerializer* class has two very important methods, the *serialize* and the *deserialize*. The *serialize* method is where the developer implements the logics of writing the fields of the Babel message object to the network buffer. When sending a message to the network, the channels first call the *serialize* method of the Babel message with the network buffer, the serialize function will write the fields of the Babel message into the network buffer, and then the channels will send the network buffer to the network sockets.

The *deserializer* does the inverse. When data from the network is delivered to the channels of Babel, the channels first read the headers of the data to know the exact *deserializer* to use, then they call the right *deserializer* method with the rest of the data in the buffer. The *deserializer* converts the array of bytes from the network buffer to Babel java objects. The internal of the *deserializer* consists in reading Java primitive data types from the network buffer, and use the primitive data types that were read to instantiate a new

Java Babel message object. However, this genial approach has a problem, a *deserializer* is given the network buffer with the data of the other *deserializers*. This can be bad when a developer by mistake implements a *deserializer* that reads more or less than it was supposed to, because the extra data that was read is discarded and will not be used to build the next message, and what was left to be read will be interpreted as a header in the next delivery.

InConnection and OutConnection when sending a message The *sendMessage* operation of the current channel of Babel requires that the user specifies a number indicating if the connection is *incoming* or *outgoing*. An *incoming* connection is a connection initiated by the remote Babel process and *outgoing* is a connection initiated by the local Babel process. This is problematic because it requires that, for each established connection, the developer keeps track of an integer indicating if the connection is incoming or outgoing. Moreover, it requires that the developer reads the implementation of the method to know exactly the number related with each connection type.

2.6.2 Appia

Appia [71, 26] 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 [71], and a channel is a sequence of session instances [70]. 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 [70]. The class **SendableEvent** is a common interface to all the events with data to be sent and received from the network [71].

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 [16].

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 [16].

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 [16].

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 [16].

However, Yggdrasil only supports one network abstraction, it can only implement protocols that run on wireless ad hoc [16] or wired IP networking [26]. It is also developed in C, which requires technically disciplined developers as to avoid frequent memory issues that could halt the development task [16, 26].

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 [17].

In Cactus there are two ways of protocol composition. Coarse grain and fine grain. [17] The fine grain protocols are micro-protocols that can exist on their own [17].

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 [17].

Micro-protocols can be executed asynchronously, however, it is the programmer's task to enforce synchronization, as Cactus does not offer it [17].

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 [35] protocol stack, since Cactus is an extension of the x-kernel [70] 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 [35].

2.6.5 Discussion

In this section, we discussed framework for protocol implementations such as Babel [26], Appia [71], Yggdrasil [16] and Cactus [17]. However, all of them have limiting features. Appia [71] has a single thread execution model that causes undesirable performance bottlenecks; Yggdrasil [16] supports only protocols for ad hoc networks; Cactus [17] does not ensure synchronization mechanisms, it is the task of the developer to do so; Babel [26] has networking limitations, one of them being that it currently supports only TCP based communications, the client channel shows the limitations of not keeping the established connections.

2.7 Summary

In this chapter, we overviewed technologies that are related to Babel [26]. 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 [77], LibP2P [58] and ZeroMQ [98]); and finally, in Section 2.6 we overviewed Babel [26] and similar frameworks (Appia [71], Yggdrasil [16] and Cactus [17]) for protocol implementations.

In the next chapter, we present our proposed solution, which consisted in developing networking channels that use the TCP, QUIC, and UDP transport protocols. The goal was to make the protocols of Babel [26] support a wider range of transport protocols.

3

Distributed Networking Channels for Babel

In this work, our goal was to develop a novel API and implementation to enrich the channel abstraction of the Babel framework. The solution (found at [45]) consisted of developing channels that support TCP, UDP, and QUIC as shown in figure 3.1. The choice of Netty[77] is justified by the fact that this library is widely adopted by developers of distributed applications, it features concise and organized documentation, and was used to develop the current network channels offered by the Babel framework today.

The figure 3.1 describes a case where two Babel applications are exchanging data using the network abstraction we developed. The two applications are exchanging data using three network transport protocols, TCP, QUIC, and UDP.

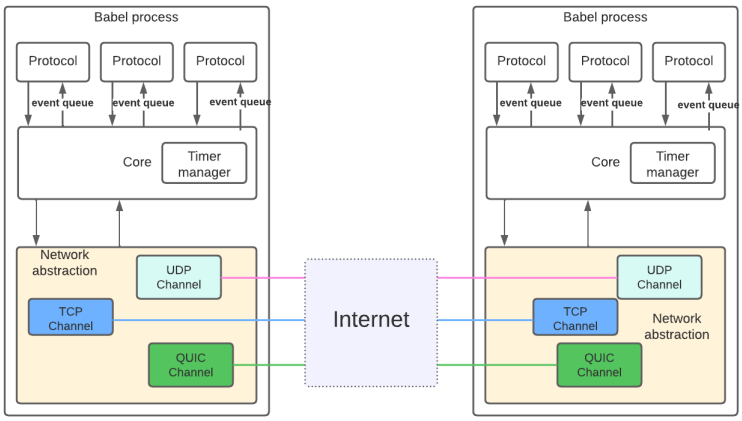


Figure 3.1: New network abstractions for Babel

3.1 Context

The current network channels of Babel support only one type of network communication protocol, TCP. However, they do not take full advantages of TCP, and other networking protocols. The TCP channels offered by Babel allow only two connections to a peer, one

in each direction; they support only the transmission of messages, they do not allow the use of streams.

At the network level, a message is a delimited set of bytes with a clear boundary. Whereas a stream is a continuous flow of bytes with no clear boundary. It is important for the channels of Babel to support streams because streams offer high performance for real time data processing, and transmitting huge amounts of data. Unlike traditional messages with fixed lengths, streams eliminate the need for receivers to wait until they receive the exact amount of data that the sender initially wrote to the network socket. Instead, they deliver data as it arrives, enabling faster handling and processing of information.

Additionally, application protocols sharing a channel can close connections being used by other protocols, which can lead to many messages not being sent; the client channel can only connect to one server, the client channel does not allow the protocols of Babel to connect to more than one server; and the application has no ways of shutting down a channel in order to stop receiving messages and release resources.

In sections below, we detail our solution, which consisted in developing channels supporting the TCP, QUIC, and UDP transport protocols. We also detail how our proposed solution addresses the limitations of Babel.

3.2 Connection-Oriented channels

These are the channels that use connection oriented transport protocol as the underlying protocol for exchanging data over a network. And we are supportting two, the TCP and QUIC transport protocols. The channels require that a connection must be established before starting exchanging network messages. Once the connection is established, the channels become able to send and receive data from the respective connections.

Each of these channels maintains a map for every active connection. Within the map, a sequentially generated ID serves as the key, while the corresponding value is represented by a Java entity linked to the connection. This entity encapsulates essential information, including the Netty connection object and the IP address that uniquely identifies the remote Babel process. Furthermore, we employ an additional map data structure to establish an association between the remote IP address of a Babel process and the collection of active connections associated with the respective remote Babel protocol. Moreover, the data structures are concurrent in order to prevent issues related with concurrency.

These two data structures play a pivotal role in facilitating the capability of Babel protocols to establish multiple connections with a single remote Babel application. These connections can be concurrently employed for the transmission and reception of data. The unique ID assigned to each connection provides a means for Babel protocols to select the specific connection they wish to utilize.

Furthermore, it is worth noting that the channels we have developed do not differentiate between incoming and outgoing connections, unlike the existing Babel channels. When

sending a message, these channels only need the message itself and either the destination IP address or the ID associated with the corresponding connection.

It is also important to note that the protocols of Babel do not have to establish multiple connections to the same remote Babel process whenever they try to open a connection to a remote Babel protocol that already has an active connection; they can opt for a single connection if they find it more suitable for their specific needs. When initiating a connection, the protocols of Babel have the flexibility to specify whether they intend to open a new connection or not if there is already a connection associated with the destination IP address. This level of control lets the protocols of Babel to manage the number of connections they employ, allowing them to effectively use the number of connections aligned with the requirements of the Babel application being built.

Stream support When opening a connection, the developer also chooses the type of data transmission connection to open. And the available types are message and stream connections.

In this context, a message is a delimited amount of bytes sent to the network in which the receiving channel only delivers to the protocols of Babel the same exact amount of bytes that were sent, and to do that the sender sends the data with its length so that the receiver later knows the amount of bytes to deliver at once. The receiver acts like this because TCP and QUIC are stream oriented protocols, and they treat sent messages as a continuous flow of bytes.

And a stream connection is one in which the senders do not send the data together with its length, just the data, and the receiving channels deliver the bytes to the protocols of Babel as they arrive. The stream connections in our channels also offer Netty implemented operations that facilitate and improve performance of sending data from any input stream (such as a file, a network stream) to the network; and also to read data from the network to any output stream (such as a file, or a network link). In Java, an input stream is a class that allows us to read data from a specific source such as a file and network connection [81]; and an output stream is a class that does the opposite, it allows the developer to write data to a specific destination, such a file, a network connection, or other channels [82].

The existing channels in Babel lack support for streams; they are primarily designed for sending messages. Streams, on the other hand, hold performance advantages, particularly in scenarios where large data volumes need to be transmitted. The advantage of streams lies in the fact that the receiving channel does not need to buffer the incoming data until receiving the exact amount of data sent, but can process it immediately.

Allocation of CPU Threads The channels offer developers the flexibility to manage the allocation of CPU threads for handling network events. This includes events like incoming requests to establish connections and events associated with the arrival of data.

We have exposed various configuration options from Netty to the developers of Babel. Within these options, developers have the option to specify the number of threads to be

allocated for handling events in outgoing and incoming connections. In which outgoing connections are those initiated locally, and incoming connections are those initiated by the remote Babel application.

Internal state of the channels Our channels also offer operations to inspect the internal state of the channels, such as the string IDs of the active connections, the IPs of the active remote Babel applications, and the transport protocol of the channel. Finally, a channel can be shut down when the protocols of Babel no longer need it.

Moreover, the channels were developed with the motivation of preventing the developers from doing repetitive and error-prone tasks. With that in mind, we made the channels keep track of all the protocols used by each connection, so that a connection could not be closed by one Babel protocol while being used by another Babel protocol. Additionally, the protocols of Babel can send messages without having to explicitly open connections, which is something that can be handled automatically by the channel.

3.2.1 TCP Channels

These are the channels that use the TCP transport protocol. In the next paragraphs, we talk with more details about the features we developed and how to enable them.

autoConnect Specifies whether the channel should open a connection to a peer when a message is sent and such a connection does not exist. This prevents the developer from always having to check if a peer is connected before sending a message.

connectTimeout This parameter defines the amount of time the client waits for the server to accept the connection before timeout.

Zoro-Copy The *Zero-Copy* feature in Netty provides an efficient means of rapidly transferring data from a file system to the network, all without copying the data from the kernel space to user space. This feature can yield significant performance improvements, particularly in protocols like FTP or HTTP. However, it is important to note that *Zero-Copy* is not universally supported across all operating systems. Specifically, it cannot be used with file systems that implement data encryption or compression; it is designed to work with the unaltered content of a file. [67]

By default, the TCP channels make use of *Zero-Copy* for sending files. Developers have the option to disable this feature by specifying the *notZeroCopy* option. When *notZeroCopy* is employed, the channel instead relies on an alternate approach. In this approach, Netty reads and sends the data of the file in manageable chunks to prevent the application from running out of memory [74]. The value for *notZeroCopy* can be any string. The default chunk size for this method is set at 8192 bytes, but developers can customize this chunk size by using the *chunkSize* property.

readInputStreamPeriod The channels we have developed come with the ability to periodically monitor input streams for accessible data and efficiently transmit this available data to the network. However, it is essential for developers to explicitly define the time interval at which the input stream should be inspected for available data. This interval, measured in seconds, can be configured using the *readInputStreamPeriod* parameter. To use this feature, the protocols of Babel need to send an input stream with a size parameter set to a value less than 1.

singleThreaded The threads of Netty serve exclusively to handle network events, such as accepting new connections, and delivering data from the network to the application, for example, to the protocols of Babel. However, Babel channels also rely on these threads to deserialize the array of bytes into Babel message objects, and also to deliver the Babel message objects to the application layer protocols. Deserializing the array of bytes has the cost of converting the array of bytes into multiple data types, such as integers, longs, strings, etc. And delivering the deserialized messages to the application protocols introduces the overhead of putting objects into a thread-safe blocking queue.

These costs can impact the performance of the application in scenarios where there are a lot of bytes to be consumed from the network, and the threads of Netty can not do that faster enough because they are busy doing the tasks described above. To solve this problem, the *singleThreaded* option lets the channel use an extra thread to release the network threads from having to deserialize and deliver the messages to the application of Babel. The network threads simply leave the reference of the array of bytes to this new thread. Inversely, the *useBabelThreadToSendM* parameter can be used so that the channels use the thread of the Babel protocols sending messages to serialize the messages sent into bytes, and write them to the network socket.

Threads It is very important to control the number of threads in multithreading networking applications, they allow us to distribute the work load and resources among the various components of our applications. For example, when having a Babel application which is not expected to receive a lot of network messages, it is very crucial not to let the channels rely on more threads than they really need, so that the rest could be allocated somewhere else, doing some data processing for example. And the options *serverThreads*, *clientThreads*, and *tcpServerBossThread* allow us to control that, in which *serverThreads* are the threads that will handle data from incoming connections, which are connections opened remotely; *clientThreads* will handle data for outgoing connections, the connections we started; and *tcpServerBossThread* is used to dedicate one single thread just for listening and handling new connections; it is mostly used in case where we expect to be receiving many connection requests. In other cases, the threads allocated to handle data for connections opened by the remote peer can also listen for incoming connection requests. These are Netty configurations which we are passing to the developers, so that they can control the amount of threads to use.

Metrics When dealing with network applications, it might be relevant to be able to collect performance indicators as for network usage and performance. They are essential to observe the health of the channels, efficiency, troubleshooting, improve quality of service, among others. This can be activated by using the flag *metrics*. And the metrics collected here refer to the number of received and sent messages in connections that are active, and also in connections that are closed.

3.2.2 QUIC Channels

These are the channels that use the QUIC transport protocol. In the next paragraphs, we talk with more details about the features we developed and how to enable them.

QUIC Stream Multiplexing In TCP channels, when a Babel protocol opens a connection to an already connected Babel remote application, the channel opens a new TCP connection. However, in the QUIC channels, it is different, when a Babel protocol tries to open a connection to an already connected Babel protocol, the channels open a new QUIC stream in the existing connection because it is most efficient than having to open a new QUIC connection. This does not make the QUIC channels using multiple QUIC streams less performant than the TCP channels using separate connections because the streams of QUIC are concurrent, and independent of each other.

QUIC is a general-purpose transport protocol and highly configurable. The following parameters were exported to the developers so that they can use them to modify the behavior of the developed QUIC channels. Some we implemented, and most of them were implemented by Netty.

Timeouts the *maxIdleTimeoutInSeconds* the maximum amount of time a connection can become idle, after which it is closed if *WITH_HEART_BEAT* is not enabled. *idleTimeoutPercentageHB* is a value between 1 and 100, indicating the percentage of the idle timeout value that should be used as a timeout to send heart beat messages to prevent the connection from becoming idle. If the value indicated is zero or negative, there will be no heart beat messages. The connection is closed if it becomes idle after a period of time, and 2 minutes is the default. Only message connections support this features, stream connections do not support heart beats, if the connection becomes idle, it is closed.

Authentication and Encryption QUIC clients and servers exchange cryptographic specs during the handshake process in order to establish a secure and authenticated connection[50]. And this is a feature that our QUIC channels support. The developer needs to enter cryptographic information for the clients and for the servers. *serverKeystoreFile*, *serverKeyStoreAlias*, and *serverKeyStorePassword* refer to the path of the Keystore file, the alias of the certificate in the Keystore, and the password of the Keystore of the server respectively. And *clientKeystoreFile*, *clientKeyStorePassword*, *clientKeyStoreAlias* these are for

the client. There needs to be two configurations for the server and for the client because we support QUIC channels with different network roles, such as a client, a server, and a peer-to-peer.

congestionControlAlgorithm QUIC lets us choose the congestion control algorithm to be used, and the options are RENO[91], CUBIC[29], and BBR[11]. CUBIC is the default.

RENO Is a congestion control algorithm which requires that an immediate acknowledgment (ACK) is received by the sender whenever the receiver receives the segment. This is so that lost packets are detected earlier, and it increases the window size (the amount of data the sender can send without an ACK) by one for every successful ACK received, and decreases by half when it detects loss; and it faces the problem of not performing well when there are multiple packet losses, because it can only detect a single packet loss at a time[91].

CUBIC Is a congestion control algorithm that uses a cubic function to increase the window size, its window growth is independent of the RTT which allows it to distribute the available bandwidth equally among data flow with different RTTs; It also increases the window size aggressively when the flow is far from the saturation point (maximum capacity of a network link), and slowly when it is close; this feature allows it to be very scalable in high speed network when the bandwidth and delay product (bandwidth * delay) is large[29].

BBR Is a new proposed congestion control algorithm. It does not use packet loss as a congestion signal, like many congestion controls do. It uses an estimate of the network link with the most bottleneck to determine its sending rate. It tries to provide high link utilization while avoiding to create queues in bottleneck buffers. And some publications show that it can deliver better performance compared to CUBIC TCP in some environments[34].

initialMaxData Is the maximum amount of unread data in bytes that will be buffered in the whole connection, which allows more data to be received as the buffer is consumed by the application. When the value is set to zero, there will not be any flow control and consequently there will be no data on any stream[84, 44].

initialMaxStreamDataBidirectionalLocal Similar as *initialMaxData*, but only applies for local bidirectional streams, streams initiated by the sender[84, 44].

initialMaxStreamDataBidiRemote Similar to *initialMaxData* but only for remote streams, streams initiated by the remote host[84, 44].

initialMaxStreamsBidirectional is the maximum amount of concurrent remotely-initiated bidirectional streams that will be allowed to be opened at any given time[84, 44].

maxAckDelay Is an integer value indicating the maximum amount of time in microseconds by which the endpoint will delay sending acknowledgments. This improves performance by avoiding excessive acking to every received packets. The QUIC receiver can delay sending ACKs so that the acks can be sent together when sending data packets[84, 44].

maxRecvUdpPayloadSize Is the maximum payload size that the endpoint is willing to receive, the default is 65527, the maximum permitted by UDP[84, 44].

maxSendUdpPayloadSize the maximum UDP outgoing payload size, default and minimum is 1200[84, 44]. *maxRecvUdpPayloadSize* and *maxSendUdpPayloadSize* act as constraint on datagram size in the same way as the path MTU (Maximum Transmission Unit, the maximum size of an IP packet that can be sent without fragmentation)[84, 44].

3.2.3 Summary

In this section, we provided an overview of the connection-oriented channels we have developed, including TCP and QUIC, highlighting the key features they offer. These features encompass support for streams; the capacity for a Babel process to establish multiple independent connections with the same remote Babel process; the developer can manage the number of threads to use for network event handling; not allowing a Babel protocol to use a connection while being used by other Babel protocols; and the provision for Babel protocols to inspect the internal state of the channels, including details such as the IP addresses of connected peers, among others.

3.3 UDP Channel

This particular channel operates on the UDP communication protocol, as defined in the UDP RFC[88], for data exchange over the network. Notably, UDP is a connectionless protocol, which means that no formal connection is established between Babel processes when utilizing this channel. Developers simply invoke the *sendMessage* method and provide it with the Babel message, at which point the network interface takes care of forwarding the packet.

It is crucial to acknowledge that UDP does not provide any delivery guarantees. Consequently, when using this transport protocol, the sender has no assurance regarding the successful delivery of the message. For enhanced flexibility, we have introduced an optional message acknowledgment feature in our UDP channel. This feature initiates a timeout upon message transmission and continues to resend the message until an

acknowledgment is received. This innovative approach allows our UDP channel to provide the speed and efficiency of UDP while also ensuring message delivery guarantees.

Furthermore, it is worth noting that the maximum UDP payload size over IP is constrained to 65,507 bytes[21]. Payload sizes exceeding this limit cannot be transmitted. To circumvent this limitation, our UDP channel has been engineered to address the issue by splitting the data into payload sizes that conform to the specifications of UDP. This enables the successful transmission of data while adhering to payload size constraints of UDP.

Retransmission of lost packets *retransmissionTimeout* and *maxRetransmissionTimeout* are the minimum and maximum timeout interval value in milliseconds. A timeout value between these two inputs is randomly generated every time a message sent, and if the message is not acknowledged before the timeout expires, the message is sent again and a new timeout is generated. If *retransmissionTimeout* is a negative number, then retransmission will be disabled. The data will only be resent *maxSendRetries* times until an ACK is received from the remote peer. We decided to have random numbers so that lost packets do not have to wait the same amount of time to be retransmitted, to reduce the number of packets retransmitted in a short interval of time. For example, if we send 100 consecutive datagram packets and none of them is acked, by introducing random delays, theoretically we can make that 50% of them are resent earlier while the other 50% wait, reducing congestion.

Moreover, if an ACK is not received, we consider that the remote peer is down, so we trigger a connection down event. The feature of acknowledging sent packets can be disabled by using a value less than one for *maxSendRetries*, and it is also disabled when using broadcasting.

Broadcasting It is one of the most used feature of UDP, in broadcasting a UDP datagram is sent to all the hosts of a network or subnet [77]. And in our UDP channel this can be activated with the flag *broadcast*.

Threads It allows the developer to allocate a specific number of threads to deal with network events, such as delivering data from the network. Specified with the flag *serverThreads*, similar to what happens in the TCP channels.

singleThreaded The same as in the TCP channels, it releases the network threads from having to deserialize network data to Babel message, and also from having to pass the Babel message to the application protocols of Babel. This feature is enabled by specifying the flag *singleThreaded*.

3.3.1 Summary

In this section, we presented an overview of the channel we have developed, which utilizes the UDP transport protocol for data exchange across the network. We also delved into additional features this channel supports, including broadcasting, a basic form of message acknowledgment to retransmit lost messages, and the capability to send messages with payloads that exceed the payload size constraints typically imposed by UDP.

3.4 Developed Channels

In this section, we discuss the exact channels we implemented and their operations. Before that, it is important to understand the concepts of network roles an application can play, such as peer-to-peer(P2P) and client-server.

3.4.1 Overview of the Channels and their roles

Peer-to-peer (P2P) and client-server applications are two fundamental architectures for networking applications, each with its own characteristics and use cases. A peer-to-peer application is one in which a participating node acts as a server and as a client at the same time, whereas, a client-server application is one in which the participating nodes can either act as a Server or act as a client but cannot embrace both capabilities[89]. And we made the channels to support these two systems, and they are: **BabelQUIC_P2P_Channel**, **BabelQUICClientChannel**, **BabelQUICServerChannel**, **BabelTCP_P2P_Channel**, **BabelTCPClientChannel**, **BabelTCPServerChannel**, and**BabelUDPChannel**.

BabelTCP_P2P_Channel and BabelQUIC_P2P_Channel They are TCP and QUIC peer-to-peer channels, which were developed to make Babel framework be used for designing and implementing peer-to-peer applications. The TCP channel is similar to the current peer-to-peer channel of Babel. Unlike the current channel of Babel, these channels have more features, such as: support of streams; they do not close connections that are being used by more than one Babel protocol; a process can open multiple connections to the same destination process; the Babel protocols have more operations to inspect the state of the channel; and the channel itself can be shutdown without terminating the application.

BabelTCPClientChannel and BabelQUICClientChannel These channels only implement the client side logic of, respectively, TCP and QUIC . They allow establishing connections to several destination hosts, but do not allow receiving incoming connections. They can only be used to connect to server and to peer-to-peer channels. They work just like the channels described above, apart from the fact that they do not listen for incoming connections. They are used for developing client applications using the Babel framework.

In the current client channel of Babel, every new connection established overrides the previously established connection, allowing only the existence of a single connection at a

time. Our solution to allow the client channel to exchange messages with more servers at the same time consists in having a map data structure, mapping the destination address of a server with a set of established connection to the same server; this allows a client channel to have multiple parallel connections to one single server, and at the same time having multiple connections to different servers. Plus, for each established connection a sequential ID is generated to be associated with the connection, so that the developer can choose which connections to use.

BabelTCPServerChannel and BabelQUICServerChannel TCP and QUIC server channels, respectively. They listen for incoming connections, they do not initiate them. They are used for server applications and are complementary to the client channels described previously. They work together with client and peer-to-peer channels. They support almost all the operations supported by **BabelTCP_P2P_Channel** channel, except for establishing connections.

BabelUDPChannel It is the channel that uses the UDP protocol. It is very simple because of the underlying transport protocol. It can be used as a server, client, or as a peer-to-peer channel because they do not require connections neither to be open, nor closed. The channel also offers some features that go beyond the semantics of UDP.

3.4.2 How to create the Channels

It is very easy to set up a channel so that they can be used by the protocols of Babel. The first step to take is to initialize the desired channel, as in the listings below. After initialization, it can be created the same way the current channels of Babel are created, as in listings below 3.1. It shows an example in which a channel is first initialized, then it is created.

Listing 3.1: Registering the Channels

```
babel.registerChannelInitializer(
    BabelQUIC_P2P_Channel.CHANNEL_NAME,
    new BabelQUICChannelInitializer(NetworkRole.P2P_CHANNEL)
);
...
Properties channelProps = NewChannelsFactoryUtils
    .quicChannelProperty(address,port);
channelId = createChannel(
    BabelQUIC_P2P_Channel.CHANNEL_NAME,
channelProps);
```

3.4.3 Interaction With Babel Protocols

There are two ways in which the protocols of Babel interact with the new channels, these interactions can be passive or active. The passive interaction is through events, events are sent to the protocols of Babel when a network event happens. And the active interaction is the direct invocation of the functions offered by the channels, the protocols directly call the functions and operations supported by the channels.

When initiating all the channels, they receive the following types of objects in the constructor: **BabelMessageSerializer**, **ChannelListener**, **Properties**, **Short**.

BabelMessageSerializer Is invoked when sending a Babel message to the network and when data is delivered from the network. It has the serializer and the deserializer of the message being sent and being delivered respectively. The serializer is used to convert the messages into bytes to be sent to the network, and the deserializer is used to convert data bytes from the network to the respective Babel message, which is a Java object.

ChannelListener Is invoked whenever a network event occurs, it is used to deliver events to the protocols of Babel; the events can be the establishment of a new connection, arrival of messages, or when a connection is closed. The existing implementation of the interface *ChannelListener* is the class *ChannelToProtoForwarder*. When there is an event to be delivered to a protocol, the *ChannelListener* puts the event into the queue of the respective protocol of Babel, and the thread of the protocol of Babel consumes the events linearly, one by one in the order they were inserted.

We decided to add concurrency for processing the events related to the data of stream connections. In the constructor of this class, we added an optional handler function as argument, which is executed directly by the network thread responsible for delivering the arrived data. The idea was to be faster in processing the data of the streams, instead of putting them in a queue to wait for their turn to be processed as described above. At the application level, this function can be passed when creating a new channel.

Property and short The property object holds the parameters to be used to set up the channel; *Short* is the data type of the identification number of the protocol of Babel that created the channel.

Despite the extensive internal differences of the network protocols which support each channel type, there was an effort to standardize the interface of the different channels. This was done with the help of Java Object-Oriented Programming features, for example, by having all the channels implementing the same **NewIChannel** interface.

3.4.4 Operations of the Channels

We changed the **IChannel** interface of Babel to a new one called **NewIChannel**. The **NewIChannel** interface is implemented by all the channels we developed, and it offers

the following operations that can be invoked by the protocols of Babel to interact with the channels:

openMessageConnection(host,protoId,alwaysConnect) Opens a connection to the specified address *host*; The name of the operation indicates that this connection will use messages, not streams. *protoId* is the ID of the protocol of Babel calling this operation. The operation is asynchronous, it means that the current execution thread of Babel does not wait until the operation of establishing a connection finishes. A string identifying the connection will be returned immediately. The flag **alwaysConnect** tells the channel to open one more connection, even if there is already a connection to the same host.

When the connection is established with success, the event *OnMessageConnectionUpEvent* will be sent to the protocol that called the operation. This event will have the connection ID, the remote host address, and other parameters. If the connection is not established, the event *OnOpenConnectionFailed* is sent with the error message.

openStreamConnection(host,protoId,destProto,alwaysConnect) this is very similar to the operation of openMessageConnection. The main differences being that it opens a connection for streaming bytes, and when the connection is established, a *OnStreamConnectionUpEvent* event is sent to the protocols of Babel. It is an independent operation, there is no need to have a previously established message connection in order to open a stream connection. The parameter *protoId* is the ID of the protocol of Babel that will receive the data of the created stream. The *destProto* is the ID of the protocol of Babel in the remote application that will receive the data sent in to the connection.

The main difference between *OnStreamConnectionUpEvent* and *OnMessageConnectionUpEvent* is that *OnStreamConnectionUpEvent* contains the *BabelInputStream* object, which is used by the protocols of Babel to stream bytes representing various primitive and non-primitive data types.

sendMessage(message,host,protoId) this operation is used to send messages over the network to the specified host. The parameter message is a BabelMessage object. Because there can be more than one connection to a single host, the message is sent to the first encountered message connection to the specified host. If there is not any connection to the specified address, the channel will open a connection if the parameter *autoConnect* was added in the set-up of the channel.

sendMessage(msg,connectionID,proto) This operation is used to send a message to a connection identified by the specified connectionID.

sendMessage(bytes,length,connectionID,sourceProtocol,destProtocol) The same as the previous one, the difference being that it uses the connection specified by *connectionID*.

getConnectionType(connectionId) returns the type of connection associated with this connectionId; It tells whether it is a message or a stream connection. If the connectionId does not exist, it returns null (undefined in Java).

registerChannelInterest(protocolId) This operation is called when a Babel protocol decides to use a channel created by another protocol. This is important because the channel will track the IDs of the protocols of Babel that are using the channel, so that when a Babel protocol tries to close a connection, the channel can inspect if other protocols are also using the respective connection being closed. The connection is not closed, if it is being used by other protocols of Babel.

closeConnection(connectionId,protocolId) It closes the connection associated by this *connectionId*; the protocolId is the protocol closing the connection. In the channels, a connection keeps a set of the protocols using it; when this operation is called, the protocol is removed from the set, and if the set becomes empty, the connection if effectively closed. However, the protocolId can also be a negative number, in this case, the connection is closed independently of the size of the set.

closeConnection(host,protoId) Similar to the previous one, the difference being that it closes all the connections to the specified *host*.

isConnected(host) Returns an indication if the channel is connected to the specified host.

isConnected(connectionId) Returns an indication if the channel has a connection identified by the specified *connectionId*.

getConnectionsIds Returns the IDs of all active connections.

getConnections Returns the IP addresses of all connected peers.

connectedPeers Returns the number of connected peers.

getChannelProto returns the ID of the Babel protocol that created the channel.

getNetWorkProtocol Returns a string that encodes the network transport protocol associated with this channel, the protocols can be TCP, UDP, or QUIC.

getNetworkRole Returns a string with the network role this channel is playing. The roles can be a Client, Server, or Peer-to-Peer.

shutDownChannel(protocolId) closes the channel and releases all the resources used by it.

activeConnectionsMetrics it returns an object with the metrics of the current active connections in connection-oriented channels like TCP and QUIC.

closedConnectionsMetrics the metrics of closed connections in connection-oriented channels like TCP and QUIC.

getUDPMetrics UDP metrics of the channel. Only works for channels that use UDP.

3.4.5 Stream Operations

By streams, we are referring to sets of bytes that are sent to the network without a clear boundary. For example, when sending messages with TCP, since it is a stream-oriented protocol, TCP treats all the messages sent as a continuous flow of bytes, there is no separation between the messages sent, and it is the receiver that receives the bytes and reconstructs them into protocol-specific discrete messages.

The operations of the streams are decoupled from the main interface of the channels. When a stream connection is opened, an event of the connection is delivered to the protocol that created the stream. This event contains a **BabelInputStream** instance which the protocol will use to write bytes to send to the over the network. When the receiving Babel protocol receives an event of stream delivery, this event will contain a **BabelOutputStream** instance that contains the received data, which can be read as raw bytes, or as primitive data types.

BabelInputStream Is used to stream bytes over the network, and it is associated to a single connection only, and the data is written to the connection it is associated with. The Babel protocol that wants to receive the data needs to register a handler function that will be called upon the arrival of data. It is registered with *registerStreamDataHandler(channelId,handlerFunction,sentHandler,failHandler)*.

handlerFunction Is the function that will be called when the protocol of Babel receives data from a stream connection. **sentHandler** Is called when data is sent. **failHandler** is called if the data failed to be sent. The following paragraphs show the operations supported by **BabelInputStream**.

Flushing It allows for data to be buffered before being written to the network sockets. It can increase performance by writing huge chunks of data to the network sockets instead of writing many small amounts. The operation *setFlushMode(trueOrFalse)* is used to control that, it tells the stream if the data should be sent to the network socket immediately (flushing) or should be buffered in the underlying network channel. The argument is a boolean flag, if set to true, the data is going to be flushed, else flushing is not happening.

After delaying flushing, the developer can also flush the buffered data by invoking the *flushStream()* operation.

Writing primitive data types There are also operations that allow the developers to write primitive data types to the network. The developer does not have to go through the struggles of converting them into bytes, they simply invoke the respective write operation that supports the data type have interest on. These primitive data types can be int, byte, short, long, float, double, boolean, and char.

Writing files and input streams The operations *writeFile* and *writeInputStream* are used to when the developer wishes to write data from a file or an input stream to the network. The *writeFile(file)* uses two different ways to stream the bytes of the file, it uses either zero-copy [68], which consists in efficiently moving data from a file system to the network without reading to the memory of the application; or chunk reading[74], which is reading the file chunk by chunk in a manageable way to prevent the application from running out of memory. And *writeInputStream(inputStream,length)* writes the specified length of data from a java input stream to the network; if the length is zero or below zero, a watcher is placed on the stream so that it periodically checks if there is new data added to it, then it is immediately written to the stream.

Application level buffering The other alternative to not flushing, is by allocating a Netty buffer at the application layer, in this case the developer simply writes the data to the allocated buffer, and send the buffer to the socket whenever they consider adequate. This is advantageous because every time we call a write operation we allocate a new buffer, and send it to the network layer; and by allocating only once we are reducing the overhead of buffer allocation. A Netty buffer is allocated by calling the operation **allocate**.

BabelOutputStream Is the object associated to a stream connection, and it carries the data that arrive from the stream it is associated with. It allows a Babel protocol to read the data from a stream connection. And we can read the data as various primitive data types, such as int, boolean, double, float, byte, short, and bytes. We can also read the data straight into an output stream without duplicating memory[76, 75]. However, the developer must be careful, because sometimes there are not enough bytes to read the data of a specific data type. For example, if there are only 3 bytes to read, and the developer tries to read an int, which requires four bytes, an error will happen.

3.4.6 Metrics and Error handling

In order to see the metrics of the channels, two properties must be added to the properties passed to the channels: **metrics** and **metricsInterval**. The value for **metrics** can be any, its presence indicates that the metrics must be calculated on the channel, and the value for **metricsInterval** must be an integer, it represents the time interval in seconds in which

the metrics event will be triggered with the metrics of the channel. **metricsInterval** only works if **metrics** is specified.

When using channels with QUIC or TCP protocols, there are metrics of the current active connections and the metrics of the old connections (closed connections). The metrics of a connection has the number of messages and the number of bytes received and sent by the applications that are using the channels. It also has the number of keep-alive messages received and sent, in the case of the channels using QUIC.

For UDP channels, the metrics are messages and bytes sent and received, as well as the sum of all the RTTs of the messages acknowledged. We use the sum so that the developers can later use this number to calculate the average RTTs of the delivered messages.

There are two ways to get the metrics, by directly invoking the functions *activeConnectionsMetrics*, *closedConnectionsMetrics*, and *getUDPMetrics*. These functions are present in the *NewIChannel* interface. And the other way to get the metrics is by registering metric event handlers in the protocol of Babel, as shown in the listing 3.2 below. The listing shows how to register the metrics handler functions in the protocols of Babel, and it also shows the signatures of the handler functions.

Listing 3.2: Registering metrics

```
registerChannelEventHandler(channelId,
    ConnectionProtocolChannelMetricsEvent.EVENT_ID,
    this::uponChannelMetrics);

registerChannelEventHandler(channelId,
    UDPMetricsEvent.EVENT_ID, this::uponUDPChannelMetrics);
...
void uponUDPChannelMetrics(UDPMetricsEvent event, int channelId) {}

void uponChannelMetrics(
    ConnectionProtocolChannelMetricsEvent event,
int channelId) {}
```

Error handling When it comes to errors that might unexpectedly arrive from the channels, the protocols of Babel can register the event **OnChannelError**, with a function that has as parameters **OnChannelError** and **channelId**. The object **OnChannelError** has the ID of the connection, the host address, and the error descriptor.

3.4.7 Changes made to Babel protocols

We were given the full liberty to make any changes we deemed necessary to the interface of the protocols of Babel. We made as few changes as possible to make the protocols of Babel fit with the new channels. The changes we made were very minimal because

we wanted an easy migration of existing Babel applications from the current channels of Babel to the new channels.

Our first change is in the method that is called when a message is delivered to the protocols of Babel; we added one more parameter to this method, the identification of the connection that received the message, because with the new channels, the developer can use the ID of a connection to send data to the respective connection.

Reading Data from stream connections We added the function **registerStreamDataHandler** to register the function that will be called when there are data to be read from a stream. The arguments of the function are: the ID of the channel; the function that will be called when there are data to be read from a stream, and it has only one argument, *BabelStreamDeliveryEvent*, which contains all the relevant information, such as the data.

We also replaced the **InConnectionUp** and **OutConnectionUp** events (that were called when incoming and outgoing connections were established) by **OnMessageConnectionUpEvent** for message connections, and *OnStreamConnectionUpEvent* for stream connections. To register the function to be called when a message connection is established, we use the function **registerChannelEventHandler**, in which the arguments are: the ID of the channel, the ID of the event, and the function to execute the event. The function to execute the event takes only two arguments, the ID of the channel, and the object *OnMessageConnectionUpEvent* for message connections, and *OnStreamConnectionUpEvent* for stream connections; These objects contain relevant information about the established connection.

Finally, to catch the error that occurs when a connection fails to be established, we register the event as depicted in the Listings 3.3.

Listing 3.3: Open Connection Failed

```
registerChannelEventHandler(channelId,
    OnOpenConnectionFailed.EVENT_ID,
    this::uponOpenConnectionFailed
);

void uponOpenConnectionFailed(OnOpenConnectionFailed
    ,channelId) {}
```

3.5 Summary

We started this chapter by introducing the transport protocols the channels we developed use. And continued with overviews of the features they use.

We also presented the 7 channels we effectively developed to expand, and enrich, the networking capabilities of Babel. The channels implemented are **BabelQUIC_P2P_Channel**,

BabelQUICClientChannel, **BabelQUICServerChannel**, **BabelTCP_P2P_Channel**, **BabelTCPClientChannel**, **BabelTCPServerChannel**, and**BabelUDPChannel**. These channels aim to make Babel support more transport protocols such as UDP and QUIC. These channels also benefited a lot from the current network channels of Babel, as they were used as initial guidelines.

Moreover, these channels also aimed to solve the problems and limitations found in the channels of Babel, such as the fact that two processes can only have two connections between them; the client channel can only connect to a single server process at a time; the protocols do not have ways to inspect the internal state of the channels; a protocol is able to close a connection that is being used by another protocol; a channel can not be destroyed once created; the protocols have to explicitly open a connection before sending a message; the channels do not allow streams despite using TCP, which is a streaming protocol; and they do not offer a ready to use interface to send files and input streams.

4

Experimental Evaluation

4.1 Experimental Settings

To evaluate our proposed and implemented channels for correctness, scalability, and performance, we took advantage of three different applications, which can be found at this repository [46]. Some of which were containerized using Docker [18] containers. We also leveraged tools and technologies like Jprofiler [47] for method execution time profiling, and the Linux Traffic Control [47] tool to simulate diverse network conditions.

4.2 Peer-to-Peer Overlay Management Experiments

Our first experiment relies on a simple peer-to-peer (P2P) application that disseminates messages over the network by leveraging two application level protocols in Babel, HyParView [53] and Plumtree [52]. HyParView [53] is a membership protocol that builds an unstructured overlay and provides nodes with stable neighboring nodes. And an unstructured overlay[83] is an abstract network topology in which the participating nodes do not follow a defined structure or hierarchy to establish connections, it is random. They are easier to maintain and use flooding or random walks (moving randomly) to discover data.

Plumtree[52] operates as a gossip protocol in which a node propagates messages to a random subset of nodes within the network, rather than broadcasting to all nodes indiscriminately. In the Plumtree protocol, a node dynamically categorizes its neighboring nodes into two distinct groups: active and passive neighbors. This classification is continually changing, with nodes transitioning between the active and passive states and vice versa. The classification changes the way in which nodes exchange messages between them.

The protocol functions as follows: when a node receives a message from one of its neighbors, it initially checks the 'time to live' counter field of the message. If this counter is less than zero, the node refrains from disseminating the message any further. Conversely, if the counter is greater than zero, the node decrements it by one and proceeds to forward

Name	CPU	total Cores/threads	Memory	Network interfaces	Disk
charmander-1	AMD EPYC 7281	16/32	128 GiB DDR4 2666 MHz	2 x 10 Gbps	1.8 TB HDD
charmander-2	AMD EPYC 7281	16/32	128 GiB DDR4 2666 MHz	2 x 10 Gbps	1.8 TB HDD
bulbasaur-1	2 x Intel Xeon E5-2609 v4	16/16	32 GiB DDR4 2400 MHz	2 x 1 Gbps	110 GB SSD
bulbasaur-2	2 x Intel Xeon E5-2609 v4	16/16	32 GiB DDR4 2400 MHz	2 x 1 Gbps	110 GB SSD

Table 4.1: Technical description of the servers

the message to its active set of neighbors.

To ensure that passive nodes also receive the messages, a node periodically sends the messages it has received to its passive set of neighbors.

These protocols had already been implemented in Babel as case studies, so we had to adapt them to make use of the newly developed channels.

Our experiment shows a P2P evaluation in a network composed of 100 nodes. In this experiment, we used the current TCP channel of Babel, the new TCP channel, and the QUIC channel we developed. For each channel, we measured the reliability of message delivery in two different scenarios: *i)* a fault-free scenario where none of the nodes fails, and *ii)* a fault scenario in which 24% of the nodes fail simultaneously.

We executed 100 Babel processes, distributed evenly across 4 servers. And the technical description of the servers are the ones shown in Table 4.1.

During the experiments, each node initiated message dissemination 15 seconds after the start, and continued until it has disseminated 300 messages. We employed two different dissemination rates: one message per second, as depicted in Figure 4.1, and one message every half a second, as illustrated in Figure 4.2.

4.2.1 Dissemination rate of 1 message per second

In this experiment, the nodes were sending a message every one second. And each experiment was executed for 7 minutes. In the end, we calculated the reliability of the messages disseminated. The reliability in this case being the number of all messages delivered divided by the number of all messages expected to be delivered by all nodes active, assuming that all active nodes are expected to receive all the messages disseminated.

Figure 4.1 presents the reliability results for all the channels. In Figures 4.1a and 4.1b, the y-axis represents the reliability, and the x-axis represents the time during which message dissemination occurred. In the third figure (Figure 4.1c) we show the time

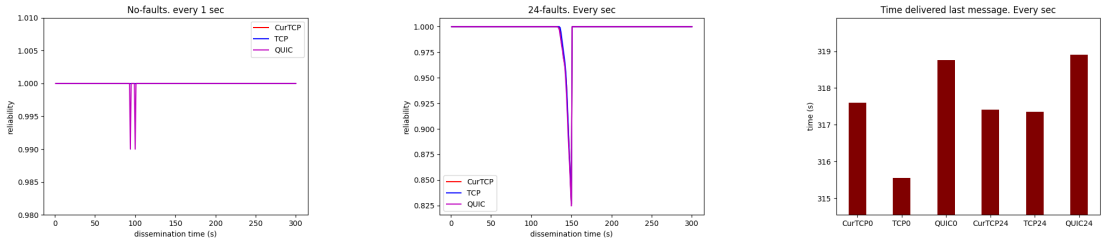
difference between the moment when the first message was disseminated in the network and the time when the last message was successfully delivered within the network. This time differential provides the total duration it took for all messages to be delivered.

The reduction observed in Figure 4.1a is attributed to the applications that utilized the QUIC channels. This drop occurred because a small subset of messages achieved a reliability rate of 99%. The reason behind this lies in the fact that in specific scenarios characterized by high network traffic, the implementation of the protocol we are using delivers messages with long delays. It is worth noting that the exact reasons for these delays are not yet fully understood, and we have been exchanging messages with the developers that implemented the Netty library we are using to investigate the causes of the problem.

Furthermore, only the current TCP channel of Babel and the new TCP channel that we developed managed to achieve 100% reliability. These two channels successfully delivered all the messages to every node in the network.

The reduction observed in 4.1b represent the moment faults happened, when 24 processes became unavailable due to crash we imposed. The reliability declines to about 76%, and after the stabilization of the overlay, the remaining nodes get to receive all the messages they sent, and the reliability reaches 100% in all channels. This is also because the overlay maintained by HyParView[53] is considered to be very stable and maintains high connectivity, nodes adjust quickly in case of failures [53, 26].

Regarding the time required to deliver all the messages, it is worthy noting that the channels exhibited delivery times ranging from 315 seconds to 319 seconds. Notably, the new TCP channel outperformed the others in scenarios where none of the nodes failed, as well as in cases with 24 node failures. This is shown in Figure 4.1c. Conversely, the QUIC channels displayed the slowest delivery times among the tested channels.



(a) reliability with zero faults (b) reliability with 24 faults (c) Time taken

Figure 4.1: Overlay graphs with one sec dissemination

4.2.2 Dissemination rate of 1 message per half second

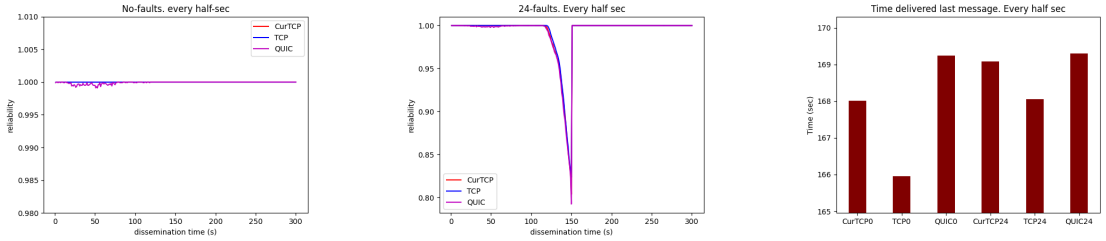
In this experiment, the nodes were sending a message every half second, every 500 milliseconds. And each experiment ran for 5 minutes.

In Figure 4.2a, the results are depicted for a scenario with no faults, where none of the nodes experienced failures. Notably, the current TCP channel of Babel and the new

TCP channel consistently achieved 100% reliability in message delivery. In contrast, the QUIC channel occasionally falls short of achieving 100% reliability because some messages exhibited a reliability rate of 99%. These instances of 99% reliability in the QUIC channel can be attributed to the issues described in Section 4.2.1.

Figure 4.2b shows the reliability in a scenario where 24 nodes failed simultaneously. Before the faults, all the TCP channels had always 100% of reliability, with the QUIC channel having reliability between 99% and 100%. And after the faults, all the channels achieved 100% reliability, all messages sent after the faults were delivered to every remained node.

Regarding the time required to deliver all the messages, it is worthy noting that the channels exhibited delivery times ranging from 165 seconds to 169 seconds. Similar to the results of the experiments where the nodes disseminated a message per second, the new TCP channel outperformed the others in scenarios where none of the nodes failed, as well as in cases with 24 node failures. This is evidenced in Figure 4.2c. Conversely, the QUIC channels displayed the slowest delivery times among the tested channels.



(a) reliability with zero faults (b) reliability with 24 faults (c) Time taken

Figure 4.2: Overlay graphs with half-sec dissemination

4.2.3 Summary

In this segment, we conducted a Peer-to-Peer experiment to assess the functionality and performance of the channels we have designed. The objective was to compare our new TCP channel with the existing TCP channel of Babel, with the expectation of achieving the same reliability with superior performance.

We calculated the reliability of messages delivered using two different dissemination ratios. We used two different dissemination ratios because we wanted to observe if the channels would be consistent on their results. And the Figures depict that the results of the channels are consistent with different dissemination rates we used, the reliability and performance results of the channels were consistent. The channels that achieved 100% reliability in all dissemination times with 1 message per second dissemination rate also achieved 100% reliability in all dissemination times with 1 message per half second in all dissemination rate. Furthermore, from one message per second rate to 1 message per half second rate, the time taken to deliver all messages also decreased by half.

The results we obtained with our new TCP channel showcased both improved speed and reliability when compared to the current TCP channel of Babel. This validation

demonstrates that our new TCP channel is not only reliable but also faster than the existing TCP channel within Babel.

Additionally, our experiment shed light on issues associated with the QUIC channel. Unfortunately, the QUIC channel failed to meet our expectations in terms of both reliability and speed.

4.3 State Machine Replication Experiments (SMR)

Our second experiment involves a state machine replication application. This application functions as an in-memory key-value store in which servers, each needing to maintain the same state, receive requests from clients. These servers then submit these requests to other server replicas to determine the order in which multiple requests should be executed across all replicas. To achieve consensus and facilitate this decision-making process, the application employs the MultiPaxos[19] consensus protocol.

In this experiment, we present an experimental comparison of a MultiPaxos implementation with four different network channels. The servers and the clients used the same channels, and we experimented with the four existing channels.

These experiments were conducted within a Docker environment, as outlined in the Docker documentation[18]. Our setup comprised four servers, each hosting a dedicated container. Within this setup, three of the servers were identical, with each running a replica, while the fourth server hosted the client application, which is a YSCB[15] benchmarking tool that we used to execute two hundred thousands (200,000) requests to the servers, with 50% inserts, and 50% reads, and payloads of 1KB. A detailed technical description of these servers is provided in Table 4.2.

Name	CPU	total Cores/threads	Memory	Network interfaces	Disk
charmander-1	AMD EPYC 7281	16/32	128 GiB DDR4 2666 MHz	2 x 10 Gbps	1.8 TB HDD
charmander-2	AMD EPYC 7281	16/32	128 GiB DDR4 2666 MHz	2 x 10 Gbps	1.8 TB HDD
charmander-3	AMD EPYC 7281	16/32	128 GiB DDR4 2666 MHz	2 x 10 Gbps	1.8 TB HDD
bulbasaur-1	2 x Intel Xeon E5-2609 v4	16/16	32 GiB DDR4 2400 MHz	2 x 1 Gbps	110 GB SSD

Table 4.2: Technical description of the servers

The experiments were divided in two parts: the first one consisted in running the containers without any emulated network restrictions, and the second part consisted in

running the containers in an emulated network with restrictions, where the network had increased latency and reduced bandwidth among the containers, which was done by using the Linux traffic control (tc) tool[63]. The goal of the artificial latency is to observe the impact of the different channels on the application on a realistic network scenario (i.e. with latencies that would be visible when replicas are not in the same local network).

In the part of the experiment that ran without the Linux traffic control tool to modify the network properties, the bandwidth was between 7Gbps to 9Gbps among all servers, with latencies of 0.240 ms. Whereas in the part of the experiment where we used Linux traffic control tool to modify the network properties, the bandwidth was 3Gbps among all servers, and the latency was 1s between some servers, and 5s between some other servers.

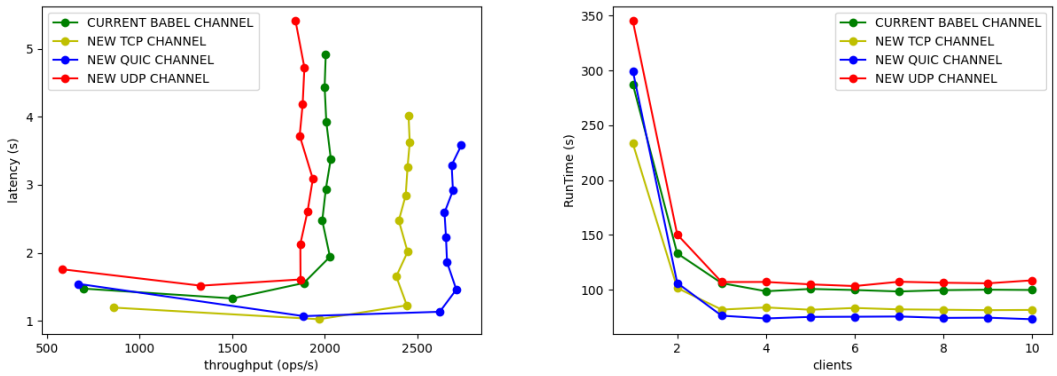
We executed the experiments with a number of client threads varying from 1 to 10. Each server application had two channels, a peer-to-peer channel to exchange messages with other replicas, and a server channel to receive requests from the clients. We repeated each execution three times. In the end, we calculated the average throughput, latencies, and runtime of each experiment so that we could easily compare the performance among the four channels involved. This experiment also shows a case of a Babel application interacting with a different application using both the channels we developed and the current channel of Babel.

4.3.1 Experiments without Linux Traffic Control tool

Figure 4.3a shows the average throughput (in the x-axis) and latency in the y-axis. We noted that our new TCP and QUIC channels have the highest throughputs, they were able to handle more requests per second than the other channels in the experiment, and with lower latencies. They reached a saturation throughput mark in between 2000–2500 messages per seconds.

In contrast, the current TCP channel of Babel and the UDP channel reached the lowest throughput numbers, and with higher latencies, they handled fewer requests per second. The weaker results obtained by the UDP channel can be explained by the lack of a control flow algorithm to adjust the amount of data to send in case of high network traffic. The current TCP channel did not perform as well as QUIC and the new TCP channel because of optimizations we made in the new channels, such as the removal of redundant code from the *sendMessage* operation as detailed in Section 4.5.1.1.

Moreover, Figure 4.3b shows the time it took the applications to respond to all the two hundred thousands (200,000) requests made by the clients; the y-axis represents the runtime in seconds, and the x-axis is the number of clients; we can see that the same applications with the highest throughputs also had the lowest execution times, they were faster to respond to the clients. Furthermore, 3 was the number of clients needed to saturate the system, because the throughput did go up as the number of clients increased. Finally, we validated the correctness of the solutions by inspecting the state of the replicated in-memory key-value store for each solution, in all cases, the replicas showed the same



(a) Latency and Throughput plot

(b) Runtime and Clients plot

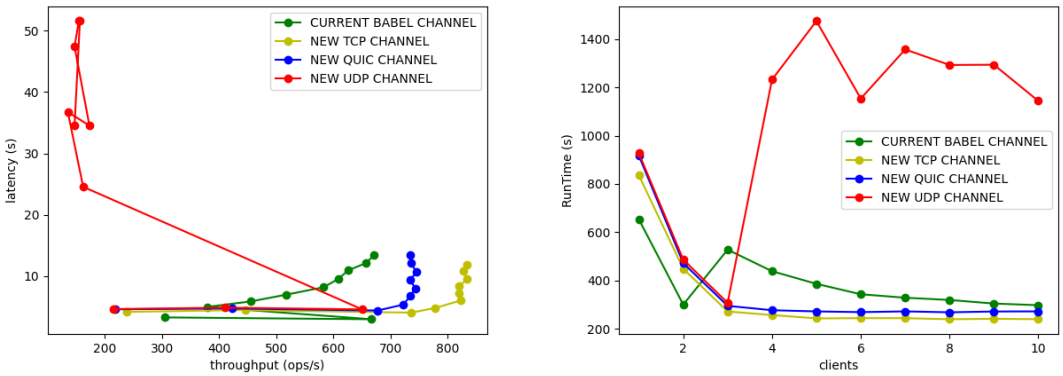
Figure 4.3: Paxos Part 1 Experiments

state.

4.3.2 Experiments with Linux Traffic Control

In the presence of a restricted network, as depicted in Figure 4.4, the results changed significantly, the maximum throughput value went from 2500 requests per second to 800 requests per second, and the new TCP channel overtook all the other channels. The current TCP channel of babel and the UDP channel fell behind, as in the experiments that did not use the Linux traffic control tool. Moreover, it is very hard to explain why the current TCP channel overtook the QUIC channel, for that we would need more testing beyond the application of the state machine replication we used.

Once again, UDP and the current TCP channel did worst than QUIC and the new TCP channel, with UDP suffering the most. And we believe it is for the same reasons we explained above, UDP does not use a proper congestion algorithm to adjust its flow according to the network conditions, and in the presence of high latencies, a lot of UDP packets ended up being dropped. And the reason why the current TCP fell behind might be due to various reasons, update in the library, optimizations we made in the code, and so on.



(a) Latency and Throughput plot

(b) Runtime and Clients plot

Figure 4.4: Paxos Part 2 Experiments

4.3.3 Summary

Through our Multipaxos experiments, as outlined in [19], our primary objectives were: Firstly, we aimed to confirm that our channels maintain the correctness of Babel applications, which were originally designed with the existing channels of Babel. Secondly, we sought to establish that Babel applications leveraging our channels exhibit significantly enhanced performance compared to the same applications utilizing current channels of Babel.

The outcomes of our experiments involving the TCP and QUIC channels, conducted under both normal and suboptimal network conditions, unequivocally showed the superiority of our channels over the existing channel of Babel. In terms of correctness, we observed that all the servers ended up with the same state.

Unfortunately, the UDP channel failed to meet our expectations in terms of speed.

4.4 Stream Diffusion Experiments

In addition to supporting message-based communication, our channels also offer the capability for stream-based communication. One of the primary distinctions between messages and streams within our channels lies in how data is handled during transmission. In messages, the receiver only delivers the data until receiving the exact number of bytes of the message sent. On the other hand, in streams, the receiver delivers any amount of bytes it gets from the network.

This distinction arises from the nature of stream protocols like QUIC and TCP. These protocols treat sent messages as a continuous, uninterrupted flow of bytes. For example, if the sender transmits 8 bytes, the receiver may initially receive 6 bytes and subsequently receive the remaining 2 bytes as part of the continuous stream.

This section provides an overview of the experiments conducted to assess the functionality and performance of message and stream connection types supported by our TCP

and QUIC channels.

4.4.1 Environment setup

We implemented a client-server application, in which the server sends a 1 GB file to each connected client. The process unfolds in the following manner.

- First, the clients simultaneously establish a message connection with the server.
- The server sends messages to all connected clients, specifying the length of the file. If the experiment is to use message connections, the server sends the bytes of the file shortly after a delay. But if the experiment is to use stream connections, the server waits for the establishment of stream connections initiated by the clients, and use the streams to send the file.

The experiments were executed in two equal servers with an AMD EPYC 7281 CPU, 16 cores with 32 threads, 128 GiB of RAM, and a network with two 10 Gbps interfaces.

We conducted a series of experiments, testing various methods of transmitting files using both message and stream connections. The experiments involving TCP were repeated ten times for each client count, while those involving QUIC were conducted five times. During each run, we calculated the average time it took for all connected clients to receive the file.

For each experiment, we generated plots representing the average time taken by clients across all runs.

4.4.2 TCP Streams

Figures 4.5a, 4.5b, and 4.6a display the outcomes of our experiments. The y-axis represents the average time required for all clients to receive the complete file. In these figures, we present the average time taken by all clients to receive the entire file. Additionally, the labels *S F* denote experiments where the channel was responsible for reading and transmitting the bytes of the file. The label *M* signifies cases in which the Babel protocols were responsible for reading the file and utilizing message-based connections to transmit the data, while *S M* indicates instances where the protocols of Babel itself read the file bytes and employed stream connections for transmission.

When comparing the results of experiments conducted exclusively using the TCP channel we developed, it becomes evident that *TCP S F* outperformed all other configurations. Following *TCP S F*, *TCP S M* exhibited better performance than *TCP M*. This performance hierarchy can be attributed to the unique characteristics of each configuration, such as the use of zero-copy with a stream connection, and the use of stream connection, and the use of message connection.

In *TCP S F*, the sender did not need to load the entire file into memory. Instead, it simply instructed the network manager to fetch the file from the file system and transmit, a concept referred to as 'zero-copy,' as detailed in the book of Netty [77].

Both *TCP M* and *TCP S M* employed a strategy of sending 65,536 bytes at a time. However, there was a key difference in how the receiver handled the data. In *TCP M*, the receiver delivered the bytes to the protocol of Babel only when the cumulative data received amounted to 65,536 bytes, due to its use of a message-based connection. On the other hand, *TCP S M*, much like *TCP S F*, promptly delivered any amount of received data to the protocol of Babel.

Additionally, it is worth noting that the receiver in *TCP M* had to perform a somewhat inefficient process. This involved reading bytes from the network, deserializing them into Babel messages (custom Java objects), and subsequently extracting these bytes from the Babel message objects to save them to the destination file. In contrast, stream connections provided a more efficient route by allowing data to be read directly from the network buffer and written directly to the destination file [76, 75]. In summary, the performance differences observed among the TCP configurations can be attributed to the following factors:

- *TCP S F* excelled due to the use of zero-copy by the sender.
- *TCP S M* achieved superior performance compared to *TCP M* because it leveraged stream connections. Stream connections ensure that any amount of received data is promptly delivered and directly written from the network buffer to the final destination, eliminating unnecessary memory copies.

In the context of our experiments, the designation *CurrTCP M* corresponds to instances where we utilized the existing TCP channel in Babel, which exclusively supports message-based communication. Notably, when we conducted experiments with six clients (Figure 4.5a), *CurrTCP M* exhibited suboptimal performance compared to the experiments employing the new TCP channels. This discrepancy can be attributed to several factors.

First, the new TCP channels not only offer support for streams but have also undergone enhancements designed to boost their overall performance. Consequently, with six clients, these new channels outperformed *CurrTCP M*.

However, the scenario is slightly different in the experiments represented by Figures 4.5b and 4.6a. In these instances, *CurrTCP M* demonstrated performance similar to that of *TCP M*, falling slightly behind the high-performing *TCP S F* and *TCP S M*, both of which leverage stream connections. This observation outlines the importance of stream connections for efficiently transmitting substantial volumes of data. In conclusion, our findings suggest that stream connections offer superior performance when dealing with large data transfers, as exemplified by the notable advantages observed in *TCP S F* and *TCP S M* configurations.

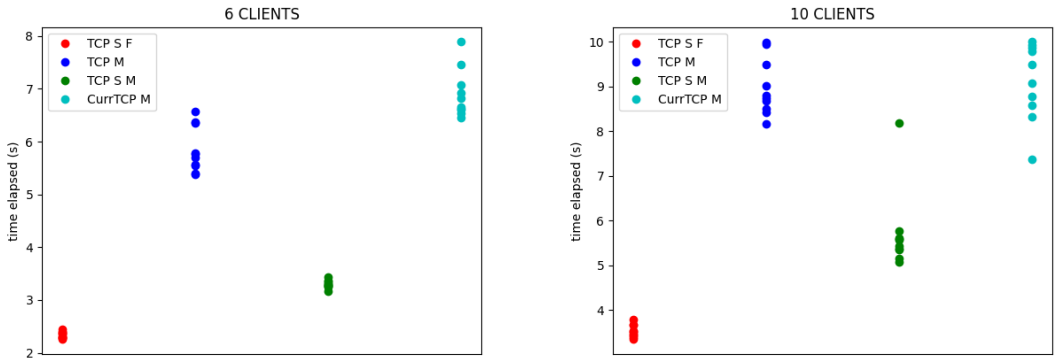
4.4.3 QUIC Streams

In our experiments, we also explored the use of QUIC. However, the outcomes were not as expected, and we encountered frequent occurrences where certain clients experienced

significant delays in receiving the complete file. The plot in Figure 4.6b clearly illustrates this disparity compared to the TCP channels, with QUIC clients taking anywhere from 100 seconds to 400 seconds to complete the file transfer.

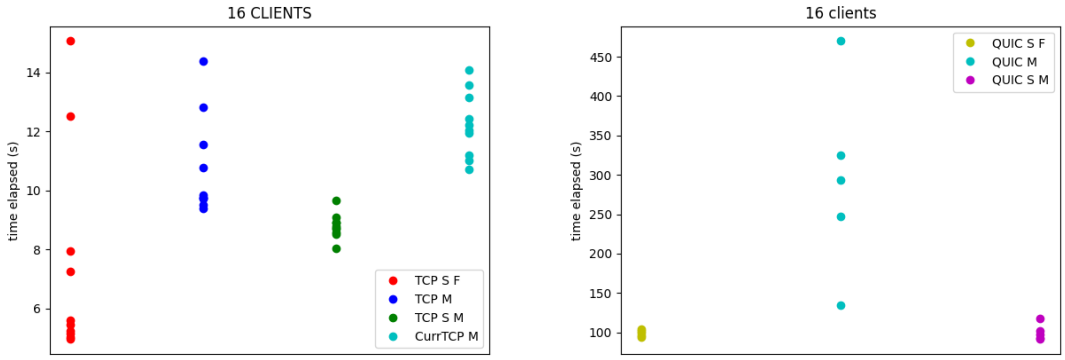
Notably, *QUIC M*, which relied on message-based connections, exhibited the least consistent performance among the various QUIC configurations. This inconsistency may be attributed to the challenges we previously discussed regarding the use of message connections for streaming bytes. Furthermore, *QUIC S F* employed an approach distinct from zero-copy. Instead, it leveraged a feature provided by Netty, which involved reading the file bytes in chunks and transmitting them individually. In this particular scenario, the chunk size was set at 8012.

The significant variations in performance observed with QUIC, as depicted in Figure 4.6b, emphasize the complexities and challenges associated with QUIC for our specific use case. Note however that QUIC is encrypting the contents being transmitted, which naturally have more overhead in relation to the other alternatives. Moreover, it is important to note that QUIC is an exceptionally configurable protocol. We firmly believe that comprehensive testing is imperative to identify the optimal parameter combinations that deliver consistent and exceptional results across a wide range of scenarios.



(a) 6 clients (b) 10 clients

Figure 4.5: Streams vs Messages with 6 and 10 clients



(a) 16 clients (b) QUIC 16 clients

Figure 4.6: Streams vs Messages with 16 clients

4.4.4 Summary

In this section, our objective was to study and highlight the benefits of utilizing streams for transmitting a substantial amount of data. The evaluation and experimentation presented here conclusively establish that, based on the performance of the current TCP channel we have developed, streams significantly outperform traditional message-based communication when dealing with large data payloads.

Unfortunately, the QUIC channels delivered unexpected results, falling significantly behind the TCP channels.

4.5 Performance and memory analysis

In this section, we analyze and explain the difference of performances among some channels, such as the current TCP channel of babel versus the new TCP channel, and the new TCP channel versus the QUIC channel.

This was analyzed in the context of the state machine replication application, in a single Babel server out of 3 servers, with a client that made 1 million requests. We gave 900MB of memory to each JVM server, and had 3 clients making 500000 requests, with 50% of which were writing and the other 50% were for reading, with messages of 1KB. and we ran this experiment in localhost in a personal laptop computer with x86_64 CPU architecture, 8 cores with 16 threads with 8GB of RAM. The goal of these experiments was to analyze the functions in the implementations of the channels that took the most time to be executed, and the memory allocation. We were not analyzing network performance.

4.5.1 Performance Analysis

Here we study and compare the methods of the channels that consume more processing time.

4.5.1.1 New TCP channel vs Current TCP channel

To understand why the current TCP channel of Babel had worst results compared to our TCP channel, we ran the JProfiler[47] to see the methods that were consuming the most time. And we found that the method *OutConnectionHandler.sendMessage* seemed to be the one responsible for it.

After diving deeper into the profiler, we found that the message is not immediately sent to the network, the call to send goes through the pipeline chain of handlers to the *MessageEncoder* handler, which has a method called by Netty to perform some custom logics (such as logging, and registering metrics) defined by the programmer before the data is effectively sent. And we also found that a call to *StringBuilder.append* was also time-consuming, it was appending *BabelMessage.toString* with *Host.toString*. After inspecting the code, we verified that the sendMessage method always concatenates strings with the plus (+) operator, and uses a logger to print them. String concatenation is an inefficient and expensive practice in terms of memory and performance, because strings are immutable, this results in a lot of garbage strings being created[93].

Overhead associated with Netty handlers For every connection, the current channel of Babel used a specific Netty handler to serialize messages before sending them to network sockets. This approach was suboptimal because Netty uses a linked data structure to store handlers and propagates events to all registered handlers before writing data to the network socket [73]. This is depicted in Figure 4.7a, starting from the third line to the fourth, shows the propagation of events through the registered handlers.

NetworkMessage Creation Something not shown in the profiler was that the current channel of Babel created a new instance of *NetworkMessage* for every message being sent. This object has the message being sent as the argument, and the object itself is the argument for the Netty handler that performs the serialization. The frequent creation of instances of this class leads to inefficient memory allocation and increased runtime, as well as additional CPU usage associated with destroying these instances.

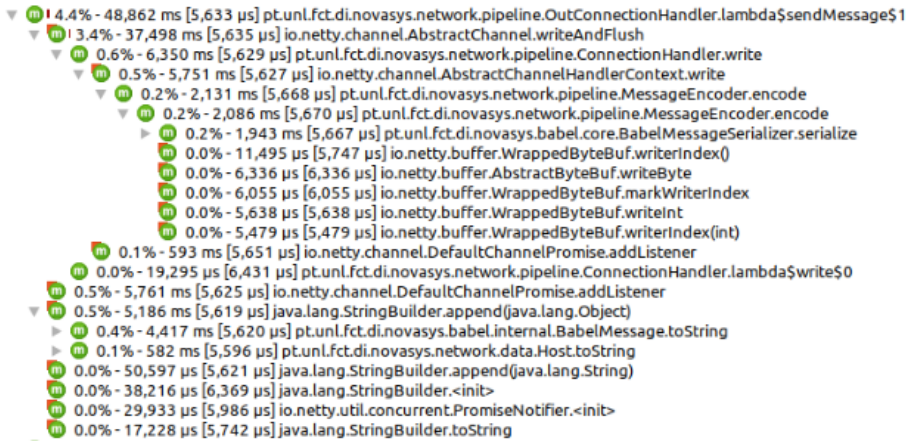
Figures 4.7 shows the call tree and execution time of the method *sendMessage* in the current channel of Babel and in the TCP channel we developed. From the Figures, we can see that a lot of things (which impact the performance) happen in the current channel of babel when sending a single message compared to what happens in our channel. When a message is sent, in our channel we allocate a Netty buffer to carry the bytes of the message, and next we call the serializer to write the bytes of the message of Babel into the new allocated buffer. We then send the buffer to the network sockets with a call to *writeAndFlush*. We addressed the issues found in the current TCP channel of Babel in the following way:

String Concatenation Redundant string concatenation for debugging purposes was removed. The new channels no longer perform frequent logging when sending a message. Instead, they send events to the Babel protocols when messages are sent, allowing developers to inspect these events to determine if messages are being sent or not.

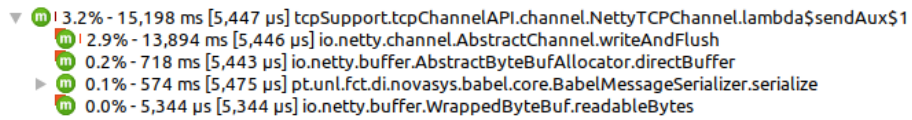
NetworkMessage and Serialization The creation of new instances of *NetworkMessage* and the Netty handler for message serialization were eliminated. Message serialization is now performed inside the body of the sendMessage operation. This change avoids the creation and destruction of unnecessary NetworkMessage instances. Additionally, a single Netty handler is now used for each connection, replacing the four [25] used by the current channel of Babel.

These changes were made to improve the performance and efficiency of the networking channels of Babel. Experiments showed that these decisions did not have a negative impact on the performance and correctness of the protocols of Babel when using the new channels. The optimizations aimed to reduce memory allocation, runtime overhead, and CPU usage associated with the previous implementation. In the end, the *sendMessage* of the current channel of Babel took an accumulated time of 48,862 milliseconds versus 15,198 milliseconds the new channel TCP took.

Finally, this and other minor optimizations, as well the much recent version of the Netty library, could be the reasons as to why the new TCP channel performs better than the current channel of Babel.



(a) Babel TCP channel



(b) New TCP channel

Figure 4.7: sendMessage call trees

4.5.1.2 New TCP channel vs New QUIC channel

The QUIC channel also falls behind the performance of our new TCP channel, and the profiler JProfiler[47] showed that there are no significant differences in the execution times between the methods of these two channels when sending and delivering messages. And this could be because we made the channels to be very similar to each other.

In the beginning we were not understanding what was happening that made the QUIC channel perform worst in some cases, but when we started the experiment described in Section 4.4, we got an insight of what could be the problem. In section 4.4 we described a client server application in which the server sends a file to various clients, and when using the QUIC channel, we found that some clients did not receive the whole file; after inspecting the logs we found that the event *connection idle* was being triggered for those clients; the event connection idle is only triggered after a connection spends a specific amount of time without any data flowing through it, and we found it to be surprising, since there were still bytes to be received. However, we decided to increase the timeout value, after which the *connection idle* event should be triggered, to a value as high as 3000 seconds. After this decision, we ran the experiment again and all the clients started receiving the file, still taking too long.

Additionally, Netty lets us choose the congestion control algorithm for QUIC servers and clients, they are RENO[66], CUBIC[29], and BBR[11]. We tried the file streaming and the peer-sampling experiment with all the algorithms, and the results did not change in any noticeable way.

4.5.2 Memory Analysis with Jprofiler

We conducted an analysis of memory allocation behavior within the replicas using our channels, with the objective of identifying potential memory-related issues. Our focus was primarily on two key aspects:

- Identifying instances where objects were being created and released unexpectedly and frequently. Such patterns could lead to excessive memory resource utilization and increase the overhead associated with garbage collection.
- Recognizing scenarios in which objects were retained in memory for extended durations, potentially causing memory leaks. This analysis aimed to pinpoint objects that were not being properly released, which could result in a gradual accumulation of memory consumption over time.

4.5.2.1 TCP channel memory analysis

Using the live memory feature of JProfiler[47] to monitor dynamic object allocation within the channel interactions, we observed notable trends in the allocation of certain objects. Notably, the following objects exhibited high instance counts due to their frequent instantiation during message transmission and reception:

BabelMessage This object serves as an encapsulator for the data transmitted by the Babel protocols to the network. A new instance of it is created when sending and receiving messages.

MessageInEvent Whenever a message is received, a new instance of this object is created to facilitate the delivery of received messages.

Host Among the most frequently instantiated objects, there is the object *Host*. When a message is received from the network, the address of the sender must be translated into an address used by the Babel protocols, which is represented by the *Host* object.

UnpooledSlicedByteBuf An essential component in this context is *UnpooledSlicedByteBuf*. Given that TCP is a stream protocol, the data it delivers can include bytes from multiple sent messages. To address this, we rely on *UnpooledSlicedByteBuf* to accurately slice the exact number of bytes corresponding to the first message and subsequently repeat this process for all the data.

This technique ensures that developers can read precisely the intended number of bytes for each message, preventing the risk of reading too few or too many bytes. In the current Babel channel, the network buffer, which may contain bytes from multiple messages, is handed over to the deserializer of the protocols of Babel. However, this approach carries a certain level of risk, as the data read from the network buffer is immediately discarded. If the deserializer reads an incorrect amount of data from the network, it can lead to the corruption of all subsequently delivered messages.

4.5.2.2 QUIC channel memory analysis

Similar to our observations with the TCP channel, we noticed a similar behavior within the QUIC channel. In this context, the same classes we previously highlighted exhibited a high number of instances, primarily due to the continuous exchange of messages. Notably, the instances of these classes exhibited a pattern of growth and subsequent reduction, which persisted until the clients concluded their activities, at which point the number of instances diminished to zero.

This behavior is significant as it enables us to identify objects that are frequently created and released. Such insights are important for detecting potential memory-related issues, especially when there are unexpected instances of an object with a notably high count in the graph.

Furthermore, it is essential to note that the observed objects did not exhibit any memory retention once the clients finished their requests. This absence of memory retention serves as a clear indicator that there are no memory leaks, as all instances are promptly and effectively destroyed when they are no longer in use.

4.5.2.3 UDP channel memory analysis

In our memory experiment, similar to the one conducted with TCP and QUIC channels, we observed that the clients did not complete their workload, leading to memory exhaustion on the servers. This outcome can be attributed to the inherent memory usage characteristics of the UDP channel. Notably, our implementation of the UDP channel incorporated features like message acknowledgement and retransmission. In this setup, messages are retained until we receive an acknowledgment for each message. Until that point, the messages are not subject to garbage collection.

To scrutinize instances of potential memory leaks, we reduced the number of client requests to facilitate the completion of the experiment. In these observations, we consistently found that the instances created were efficiently managed by the garbage collector, mirroring our findings in the TCP and QUIC channels.

In terms of overall memory usage, we leveraged the Linux System Monitor tool to inspect resource consumption. Our analysis revealed that applications utilizing the new TCP channel, the QUIC channel, and the current TCP channel maintained a memory footprint of approximately 800 MB, while the UDP channel exceeded 900 MB due to its distinctive memory usage characteristics, such as having a map data structure in the channel with all the allocated network buffer not yet acknowledged.

4.5.3 Summary

In this section, our primary goals were to inspect and identify objects in our channels that have unexpected number of instance count, in order to identify possible cases of memory leaks. And with the tools such as Jprofiler [47] we observed that there were not any issues that could cause memory leaks, the objects with high number of instances are the ones we expected, and they are related with the actions of sending and receiving messages. And the number of instances of those objects decline to zero when the clients stop making requests.

We also wanted to understand the difference of performance between the channels we developed and the current channel of Babel, and we found that the function *sendMessage* in the current channel of Babel performs more tasks, such as appending the fields of the message being sent for logging.

4.6 Discussion

In this chapter, we outlined the experimental configurations and assessments conducted to evaluate and validate the channels we have developed. Additionally, we conducted performance comparisons of the functions of the channels that send and receive messages.

Through our experimental evaluations, we aimed to validate the following key points:

Capability of Extending Network Features We sought to confirm our ability to enhance the network capabilities of Babel by introducing new transport protocols such as QUIC and UDP into the framework.

Peer-to-Peer Overlay Testing In our peer-to-peer overlay experiment (Section 4.2), we assessed the capacity of the channels to handle a high volume of connections. The outcomes revealed that the new TCP channel consistently demonstrated optimal reliability, always reaching 100%, while the QUIC channel failed to always maintain the 100% reliability, sometimes it did not go past 99%.

State Machine Replication We wanted to validate whether our channels could reliably support the development of high-speed applications, even under adverse network conditions. The results showed the effectiveness of both the new QUIC and TCP channels, even in challenging network scenarios.

Stream Transmission Assessment In Section 4.4.3, we focused on testing the ability of our channels for establishing connections to transmit streams of bytes, such as the bytes of a file. The findings indicated that the performance in this context was variable. The new TCP channel proved to be excellent for handling streams, whereas the QUIC channel encountered challenges, with some clients experiencing prolonged delays in receiving the file.

Performance and Memory profiling In Section 4.5 we used the Jprofiler [47] tool to inspect cases in our channels related to memory issues, such as unexpected memory creation and memory leak, and we did not observe such cases. We also compared the performance of the operation *sendMessage* of the current channel of Babel against the TCP channel we developed, and we found that the current channel of Babel does many redundant tasks when sending messages, such as redundant concatenation of strings and unnecessary memory creation.

4.7 Summary

In this chapter we presented the results of the experiences we conducted. In summary, the TCP channel we developed consistently demonstrated reliability and speed across various use cases. However, the QUIC channel exhibited a preference for reliability but fell short of expectations when utilized for streaming and the peer-sampling application, but it showed optimal results with state machine replication application. Additionally, our use of the UDP channel in the state machine replication experiment yielded undesirable results.

5

Final Remarks

This chapter presents the final remarks regarding the thesis project we developed, and points to things that still need to be done in order to make the network channels of Babel more extensible.

5.1 Conclusions

In this thesis, we developed three network channels to enrich, and extend the current network layer of Babel by supporting more transport protocols. The channels we developed support the UDP, QUIC, and TCP transport protocols.

The channels we developed also mitigated some defects that are present in the current channels of Babel, such as:

Streams The current channel of babel does not support streams, only messages, the developer has no way of taking advantages of the stream features offered by TCP. Also, Netty offers ready to use interfaces to send big files, and the current channels of Babel do not take advantages of this feature. Currently in Babel sending a file implies writing a lot of lines of code which consist in reading the file, making class objects to carry the bytes, and individually send each object to the channels as Babel messages.

Sharing a channel The current channels of Babel are not aware of the Babel protocols using the channels. A connection being used by a Babel protocol can easily be closed by other Babel protocols, leading to a lot of messages not being sent.

Multiple connections In any current channel of Babel, a Babel process can only open one connection to another Babel process. If the protocols of Babel want to have two connections, each Babel process has to initiate a connection. And if the protocols of Babel need more than two connections to the same remote Babel process, they can not have it, because it is not currently supported. With the new channels, we solved this issue.

Client channel connecting to multiple servers A client channel in Babel can only exchange messages with one server at a time. The new client channel can establish and keep connections to multiple servers.

Message isolation In the current channel of babel, a message deserializer has access to the network buffer with its data and also the data of other unrelated messages; this is dangerous because if various developers are using Babel to implement various Babel protocols and one of them has a deserializer that accidently reads less, or more than it was supposed to, it jeopardizes the work of the other developers, because in Netty, the data from the buffer are read in the order they arrive, and discarded immediately after being read.

Security With the new QUIC channels we can send secure data over the network without the effort of having to know a lot about security and how to implement it. The developer just needs to specify the security specifications, which include the paths of the keystore files.

Broadcasting The developer can use the new UDP channel to implement broadcasting related applications.

Stream Multithreading In the current version of babel, all the events in a Babel protocol are executed linearly, the channels put message delivery events into a queue, and they are processed one-by-one by the Babel protocol that owns the queue. And with our channels, when using streams the user can pass a function that is executed by the thread of Netty concurrently, leaving the thread of the Babel protocol to process other events from the queue.

5.2 Future Work

It is relevant to keep extending the network channels of Babel in supporting more channels with more networking protocols, such as the protocols we discussed in the related work section. Just as relevant would be the integration of middlewares such RabbitMQ [85], Kafka [1], and Zookeeper [101] in the network abstraction of Babel.

Bibliography

[1] Apache Kafka Communnity. *Appache KaftDocumentation*. 2023. URL: https://kafka.apache.org/documentation/ (cit. on p. 64).

[2] AWS Community. *Message Queue Basics*. 2023. URL: https://aws.amazon.com/message-queue/ (cit. on p. 14).

[3] AWS Community. *MQTT*. 2023. URL: https://aws.amazon.com/what-is/mqtt/ (cit. on p. 14).

[4] AWS Community. *Request-Reply*. 2023. URL: https://docs.nats.io/nats-concepts/core-nats/reqreply (cit. on p. 14).

[5] AWS Community. *What is Pub/Sub Messaging?* 2023. URL: https://aws.amazon.com/pub-sub-messaging/ (cit. on p. 14).

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

[7] 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).

[8] R. Barone. *What are libraries in programming?* 2020. URL: https://www.idtech.com/blog/what-are-libraries-in-coding (cit. on p. 14).

[9] L. Bhatia. *Message Queues - A comprehensive overview*. 2023. URL: https://wiki.aalto.fi/download/attachments/116673303/message_queues.pdf?version=1&modificationDate=1479385770054&api=v2 (cit. on p. 14).

[10] R. T. Braden. *TIME-WAIT Assassination Hazards in TCP*. RFC 1337. 1992-05. DOI: 10.17487/RFC1337. URL: https://www.rfc-editor.org/info/rfc1337 (cit. on p. 9).

[11] N. Cardwell et al. "BBR congestion control". In: *IETF Draft draft-cardwell-iccrg-bbr-congestion-control-00* (2017) (cit. on pp. 32, 59).

[12] chromium.org. *QUIC Protocol*. 2023. URL: https://www.chromium.org/quic/ (cit. on pp. 6, 11–13).

[13] CloudFlare. *What is the OSI Model*. 2023. URL: https://www.cloudflare.com/learning/ddos/glossary/open-systems-interconnection-model-osi/ (cit. on pp. 4, 5).

[14] Comptia Community. *What Is NAT?* 2020. URL: https://www.comptia.org/content/guides/what-is-network-address-translation (cit. on p. 13).

[15] B. F. Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154 (cit. on p. 49).

[16] P. A. Costa, A. Rosa, and J. Leitão. "Enabling wireless ad hoc edge systems with yggdrasil". In: *Proceedings of the 35th Annual ACM Symposium on Applied Computing*. 2020, pp. 2129–2136 (cit. on pp. 23–25).

[17] S. M. de la Cruz. "Protocol Composition Frameworks and Modular Group Communication: Models, Algorithms and Architectures". PhD thesis. Verlag nicht ermittelbar, 2006 (cit. on pp. 24, 25).

[18] Docker Community. *Docker Overview*. 2023. URL: https://docs.docker.com/get-started/ (cit. on pp. 45, 49).

[19] H. Du and D. J. S. Hilaire. "Multi-Paxos: An implementation and evaluation". In: *Department of Computer Science and Engineering, University of Washington, Tech. Rep. UW-CSE-09-09-02* (2009) (cit. on pp. 49, 52).

[20] e. a. Fairhurst. *Services Provided by IETF Transport Protocols and Congestion Control Mechanisms*. 2017. URL: https://www.rfc-editor.org/rfc/rfc8095.html#page-4 (cit. on pp. 6, 8–10).

[21] G. Fairhurst. *The User Datagram Protocol (UDP)*. 2008. URL: https://erg.abdn.ac.uk/users/gorry/course/inet-pages/udp.html (cit. on p. 34).

[22] G. Fairhurst. *The User Datagram Protocol (UDP)*. 2023. URL: https://github.com/netty/netty-incubator-codec-quic (cit. on p. 15).

[23] I. Fette. *The WebSocket Protocol*. 2022. URL: https://www.rfc-editor.org/rfc/rfc6455#page-5 (cit. on p. 17).

[24] P. Fouto and colleagues. *Network layer*. 2022. URL: https://github.com/pfouto/network-layer (cit. on p. 21).

[25] P. Fouto and colleagues. *OutConnectionHandler.java*. 2018. URL: https://github.com/pfouto/network-layer/blob/bdb64e883953efcbe9ce0ffff525028a0a6963e1/src/main/java/pt/unl/fct/di/novasys/network/pipeline/OutConnectionHandler.java#L22 (cit. on p. 58).

[26] P. Fouto et al. "Babel: A Framework for Developing Performant and Dependable Distributed Protocols". In: *arXiv preprint arXiv:2205.02106* (2022) (cit. on pp. 1–4, 20, 21, 23–25, 47).

[27] P. Fouto et al. "Babel: A framework for developing performant and dependable distributed protocols". In: *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2022, pp. 146–155 (cit. on pp. 1–4, 20, 21).

[28] J. Fruehe. *How do network virtualization and network abstraction compare?* 2019. URL: https://www.techtarget.com/searchnetworking/answer/How-do-network-virtualization-and-network-abstraction-compare (cit. on p. 4).

[29] S. Ha, I. Rhee, and L. Xu. "CUBIC: a new TCP-friendly high-speed TCP variant". In: *ACM SIGOPS operating systems review* 42.5 (2008), pp. 64–74 (cit. on pp. 32, 59).

[30] HashiCorp. *What is Stream Multiplexing*. 2016. URL: https://github.com/hashicorp/yamux/blob/master/spec.md (cit. on p. 17).

[31] P. Hintjens. *ZeroMQ Pipeline*. 2023. URL: https://rfc.zeromq.org/spec/30/ (cit. on p. 20).

[32] P. Hintjens. *ZeroMQ Publish-Subscribe*. 2023. URL: https://rfc.zeromq.org/spec/29/ (cit. on p. 19).

[33] P. Hintjens. *ZeroMQ Request-Reply*. 2023. URL: https://rfc.zeromq.org/spec/28/ (cit. on pp. 19, 20).

[34] M. Hock, R. Bless, and M. Zitterbart. "Experimental evaluation of BBR congestion control". In: *2017 IEEE 25th International Conference on Network Protocols (ICNP)*. 2017, pp. 1–10. DOI: 10.1109/ICNP.2017.8117540 (cit. on p. 32).

[35] 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. 24).

[36] IBM Community. *TCP flow control and the sliding window*. 2013. URL: https://www.ibm.com/docs/de/tsm/7.1.0?topic=tuning-tcp-flow-control-sliding-window (cit. on p. 9).

[37] IBM Community. *What Is PMTUD?* 2021. URL: https://www.ibm.com/docs/fi/zvm/7.2?topic=terminology-what-is-pmtud (cit. on p. 10).

[38] Imatix authors. *ZeroMQ Message Transport Protocol*. 2023. URL: https://rfc.zeromq.org/spec/23/ (cit. on pp. 18, 19).

[39] Imatix authors. *ZeroMQ Message Transport Protocol*. 2023. URL: https://rfc.zeromq.org/spec/13/ (cit. on pp. 18, 19).

[40] C. 9. Information Sciences Institute University of Southern California 4676 Admiralty Way Marina del Rey. *INTERNET PROTOCOL*. 1981. URL: https://www.rfc-editor.org/rfc/rfc791 (cit. on pp. 6, 7).

[41] IPFS Project Community. *Libp2p*. 2022. URL: https://docs.ipfs.tech/concepts/libp2p/ (cit. on p. 17).

[42] 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. 12, 13).

[43] J. Postel. *INTERNET CONTROL MESSAGE PROTOCOL*. 1981. URL: https://www.rfc-editor.org/rfc/rfc792 (cit. on p. 7).

[44] M. T. Jana Iyengar. *QUIC: A UDP-Based Multiplexed and Secure Transport*. 2021. URL: https://datatracker.ietf.org/doc/rfc9000/ (cit. on pp. 2, 32, 33).

[45] D. Joao. "New Babel Network Abstraction". In: *https://github.com/* (2023). DOI: https://github.com/DanielAlmeidaJoao/nettyStarting (cit. on p. 26).

[46] D. Joao. *thesisCaseStudies*. 2023. URL: https://github.com/DanielAlmeidaJoao/thesisCaseStudies (cit. on p. 45).

[47] JProfiler Community. *Java Profiler*. 2023. URL: https://www.ej-technologies.com/products/jprofiler/overview.html (cit. on pp. 45, 57, 59, 61, 62).

[48] L. Kalita. "Socket programming". In: *International Journal of Computer Science and Information Technologies* 5.3 (2014), pp. 4802–4807 (cit. on p. 7).

[49] 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 pp. 5, 6).

[50] 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, 5, 11–13, 31).

[51] G. Lee. *Cloud networking: Understanding cloud-based data center networks*. Morgan Kaufmann, 2014 (cit. on p. 12).

[52] 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. 45).

[53] 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 pp. 45, 47).

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

[55] Libp2p Project Community. *Implementations*. 2022. URL: https://libp2p.io/implementations/ (cit. on p. 16).

[56] Libp2p Project Community. *WebRTC*. 2022. URL: https://docs.libp2p.io/concepts/transports/webrtc/ (cit. on p. 17).

[57] Libp2p Project Community. *WebTransport*. 0202. URL: https://docs.libp2p.io/concepts/transports/webtransport/ (cit. on p. 17).

[58] Libp2p Project Community. *What is libp2p*. 2022. URL: https://docs.libp2p.io/concepts/introduction/overview/ (cit. on pp. 14, 16, 17, 20, 25).

[59] Libp2p Project Community. *What is Stream Multiplexing*. 0202. URL: https://docs.libp2p.io/concepts/multiplex/yamux/ (cit. on p. 17).

[60] Libp2p Project Community. *What is Stream Multiplexing*. 2022. URL: https://docs.libp2p.io/concepts/multiplex/overview/ (cit. on p. 17).

[61] Libp2p Project Community. *What is Stream Multiplexing*. 2022. URL: https://docs.libp2p.io/concepts/multiplex/mplex/ (cit. on p. 17).

[62] N. Lincs. *Nova FCT home page*. 2023. URL: https://www.fct.unl.pt/ (cit. on p. 4).

[63] Linux TC Community. *Linux TC*. 2023. URL: https://man7.org/linux/man-pages/man8/tc.8.html (cit. on p. 50).

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

[65] I. M. Duke F5 Networks. *Network Address Translation Support for QUIC*. 2020. URL: https://www.ietf.org/archive/id/draft-duke-quic-natsupp-03.html (cit. on p. 13).

[66] M. Mathis and J. Mahdavi. "Forward acknowledgement: Refining TCP congestion control". In: *ACM SIGCOMM Computer Communication Review* 26.4 (1996), pp. 281–291 (cit. on p. 59).

[67] N. Maurer and M. Wolfthal. *Netty in Action*. Manning Publications, 2016, pp. 77–77 (cit. on p. 29).

[68] N. Maurer and M. Wolfthal. *Netty in action*. Manning Publications, 2016 (cit. on pp. 15, 16, 41).

[69] N. Maurer and M. Wolfthal. *Netty in action*. Manning Publications, 2016, pp. 49–49 (cit. on p. 16).

[70] 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. 21, 23, 24).

[71] 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. 23, 25).

[72] 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. 14).

[73] Netty Community. *ChannelPipeline*. 2018. URL: https://netty.io/4.0/api/io/netty/channel/ChannelPipeline.html (cit. on p. 57).

[74] Netty Community. *Class ChunkedNioFile*. 2008. URL: https://netty.io/4.1/api/index.html?io/netty/handler/stream/ChunkedNioFile.html (cit. on pp. 29, 41).

[75] Netty Community. *Copy direct ByteBuf with buffer to OutputStream*. 20218 (cit. on pp. 41, 54).

[76] Netty Community. *Transferring a ByteBuf to Outputstream without duplicating memory*. 2018. URL: https://github.com/netty/netty/issues/7804 (cit. on pp. 41, 54).

[77] Netty Project Community. *Netty*. 2023. URL: https://netty.io/ (cit. on pp. 14, 15, 21, 25, 26, 34, 53).

[78] Netty Project Community. *Netty/Incubator/Codec/Quic 0.0.1.Final released*. 2020 (cit. on p. 20).

[79] Nova Lincs. *Nova Lincs Home page*. 2023. URL: https://nova-lincs.di.fct.unl.pt/ (cit. on p. 4).

[80] J. Onisick. *Why We Need Network Abstraction*. 2012. URL: https://www.networkcomputing.com/networking/why-we-need-network-abstraction (cit. on p. 5).

[81] Oracle Community. *Class InputStream*. 2023. URL: https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html (cit. on p. 28).

[82] Oracle Community. *Class OutputStream*. 2023. URL: https://docs.oracle.com/javase%2F7%2Fdocs%2Fapi%2F%2F/java/io/OutputStream.html#:~:text=An%20output%20stream%20accepts%20output,writes%20one%20byte%20of%20output. (cit. on p. 28).

[83] Y. Qiao and F. E. Bustamante. "Structured and unstructured overlays under the microscope". In: USENIX. 2006 (cit. on p. 45).

[84] Quiche Community. *Struct quiche Config*. 2018. URL: https://docs.rs/quiche/latest/quiche/struct.Config.html#implementations (cit. on pp. 32, 33).

[85] RabbitMQ Communnity. *AMQP 0-9-1 Model Explained*. 2022. URL: https://www.rabbitmq.com/tutorials/amqp-concepts.html (cit. on p. 64).

[86] E. Rescorla. *Diffie-hellman key agreement method*. Tech. rep. 1999 (cit. on p. 11).

[87] RFC. *TRANSMISSION CONTROL PROTOCOL*. 1981. URL: https://www.ietf.org/rfc/rfc793.txt (cit. on pp. 2, 5–9, 13).

[88] J. P. RFC. *User Datagram Protocol*. 2023. URL: https://www.rfc-editor.org/rfc/rfc768 (cit. on pp. 2, 5, 6, 10, 13, 33).

[89] R. Schollmeier. "A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectures and Applications". In: *Proc. of the First International Conference on Peer-to-Peer Computing* (2001-09), pp. 101–102. DOI: 10.1109/P2P.2001.990434 (cit. on p. 35).

[90] 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. 17).

[91] I. Stoica. "A Comparative Analysis of TCP Tahoe, Reno, New-Reno, SACK and Vegas". In: *Communication Networks, Student Project* (2005) (cit. on p. 32).

[92] M. Sústrik et al. "ZeroMQ". In: *Introduction Amy Brown and Greg Wilson* (2015) (cit. on p. 18).

[93] Y. H. Tian. "String Concatenation Optimization on Java Bytecode." In: *Software Engineering Research and Practice*. Citeseer. 2006, pp. 945–951 (cit. on p. 57).

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

[95] I. Wigmore. *How do network virtualization and network abstraction compare?* 2021. URL: https://www.techtarget.com/whatis/definition/abstraction (cit. on p. 4).

[96] Wikipedia Community. *Messaging pattern*. 2023. URL: https://en.wikipedia.org/wiki/Messaging_pattern (cit. on pp. 4, 13).

[97] J. M. Winett. *Definition of a socket*. Tech. rep. 1971 (cit. on p. 7).

[98] ZeroMQ Project Community. *Get Started*. 2023. URL: https://zeromq.org/get-started/ (cit. on pp. 14, 18, 25).

[99] ZeroMQ Project Community. *ØMQ - The Guide*. 2023. URL: https://zguide.zeromq.org/docs/chapter3/ (cit. on p. 18).

[100] 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. 6, 11–13).

[101] ZooKeeper Community. *ZooKeeper Overview*. 2023. URL: https://zookeeper.apache.org/doc/current/zookeeperOver.html (cit. on p. 64).