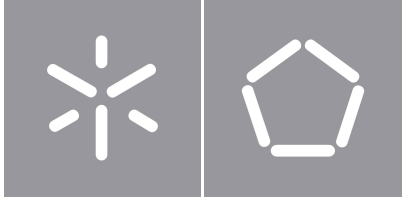
University of Minho
School of Engineering

Alexandre Silva Martins

Design and implementation of messaging middleware over transport protocol with exactly-once guarantee

july 2025



University of Minho
School of Engineering

Alexandre Silva Martins

Design and implementation of messaging middleware over transport protocol with exactly-once guarantee

Master's Dissertation in Informatics Engineering

Dissertation supervised by
Paulo Sérgio Almeida

july 2025

Copyright and Terms of Use for Third Party Work

License granted to users of this work:

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisor, Professor Paulo Sérgio Almeida, for his availability, patience, and invaluable advice throughout the development of this thesis. His expertise in the field had a significant impact on the direction and quality of this work, especially during times when I felt lost. His guidance consistently helped me find solutions to the problems that arose.

I would also like to acknowledge the projects ZeroMQ, nanomsg, and NNG, which served as key sources of inspiration in shaping the features and design of the middleware developed in this thesis. Additionally, studying the Linux kernel offered crucial insights into the implementation of polling mechanisms and strategies to reduce contention – knowledge that inspired many performance aspects of this work.

I am deeply thankful to my parents for their unwavering support – physically, by working tirelessly to provide all the necessary tools for my academic success and well-being, and emotionally, by encouraging me to pursue my goals with determination and confidence.

I would also like to thank my brother for always being available to listen and for serving as an inspiration in my pursuit of success.

To the rest of my family – as much I would like to write about each of you individually, doing so would make this an autobiography instead of a thesis – thank you for your constant encouragement and belief in my potential. Your support was crucial in this pursuit of a master's degree in Software Engineering.

Last but not least, I am grateful to my friends, who not only provided moral support during this journey but also shared valuable insights whenever I came to them with challenges related to this thesis.

Statement of Integrity

I hereby declare having conducted this academic work with integrity.

I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, Braga, july 2025

Alexandre Silva Martins

Abstract

The use of a messaging middleware (MoM), such as **ZeroMQ**, simplifies and speeds up the development of distributed applications by offering a variety of communication patterns. Combining the utility and flexibility of such middleware solutions with the exactly-once (EO) delivery guarantee can further simplify the development process. ZeroMQ, a widely popular middleware, does not provide this level of delivery guarantee. Even when using the TCP transport protocol, EO delivery is only ensured within a single connection. If a disconnection occurs, such as when the IP address changes, any buffered messages are lost.

Exon is a transport library that offers EO delivery guarantees, but only exposes a simple API, lacking the versatility of high-level messaging patterns. This thesis addresses the challenge of designing and implementing a lightweight messaging middleware, **A3M** (Martins, 2025), that combines the flexibility of messaging patterns like those in ZeroMQ with the simplicity of defining communication flows under the EO delivery guarantees provided by the Exon transport library.

This thesis discusses the challenges involved in adapting messaging patterns, exhibited by ZeroMQ, to work under exactly-once delivery semantics. For example, using the publish-subscribe pattern as-is is not feasible under EO semantics, thus, a synchronized variant was devised where the slowest subscriber sets the pace of message publishing. This thesis also explains how the Exon library was adapted and extended, especially to improve the middleware's reliability and flexibility in mobility scenarios – such as when a car moves out of range of one network station and recovers connectivity through another.

Keywords messaging, middleware, communication, exactly-once delivery, network, faults, sockets

Resumo

A utilização de um middleware de comunicação (MoM), como o **ZeroMQ**, simplifica e acelera o desenvolvimento de aplicações distribuídas ao disponibilizar uma variedade de padrões de comunicação. A combinação da utilidade e flexibilidade destas soluções de middleware com a garantia de entrega exactly-once (EO) pode simplificar ainda mais o processo de desenvolvimento. O ZeroMQ, um middleware amplamente utilizado, não fornece este nível de garantia de entrega. Mesmo utilizando o protocolo de transporte TCP, a entrega EO apenas é garantida dentro de uma única conexão. Caso ocorra uma desconexão, como por exemplo devido à alteração do endereço IP, as mensagens em buffer são perdidas.

O **Exon** é uma biblioteca de transporte que oferece garantias de entrega EO, mas expõe apenas uma simples API que não suporta padrões de comunicação de alto nível. Esta tese aborda o desafio de conceber e implementar um middleware de comunicação, o A3M (Martins, 2025), que combina a flexibilidade dos padrões de comunicação presentes no ZeroMQ com a simplicidade na definição de fluxos de comunicação, sob as garantias de entrega exactly-once fornecidas pela biblioteca de transporte Exon.

Esta tese discute os desafios associados à adaptação de padrões de comunicação, como os utilizados pelo ZeroMQ, para funcionarem segundo a semânticas definida pela entrega EO. Por exemplo, o padrão publish-subscribe tradicional não é viável sobre a garantia EO. Por isso, uma variante com sincronização foi desenvolvida, compatibilizando o padrão ao utilizar o subscritor mais lento para definir o ritmo da publicação. Nesta tese é também explicado como a biblioteca Exon foi modificada, especialmente para aumentar a fiabilidade e flexibilidade do middleware ao suportar cenários de mobilidade – como quando um automóvel sai da área de cobertura de uma estação de rede e recupera a conectividade através de outra.

Palavras-chave mensagens, middleware, comunicação, entrega única, rede, falhas, sockets

Contents

List of Figures

List of Tables

Part I

Introductory material

Chapter 1

Introduction

1.1 Context and Motivation

In today's realm of applications and systems, where things are spread out over networks, reliable and efficient communication systems become crucial to enable seamless interaction between the components.

Reliable communication comprehends multiple aspects, with one important aspect being the assurance that messages are delivered precisely one time, especially in situations where messages must not be lost or delivered more than once. This delivery guarantee is critical for several applications and systems, like in financial transactions (e.g. payments) or important data exchanges.

In the pursuit of the creation of fault-tolerant distributed systems, ensuring exactly-once message delivery is a theme vastly explored by researchers and engineers to face network uncertainties, system failures and asynchronous communication. As the demand for reliable communication grows, creating a messaging middleware becomes an essential solution. This middleware will prevent developers from repeatedly implementing the same (or similar) solutions, in addition, to acting as a robust bridge that enables better communication between components in a networked environment.

Through this thesis, I seek to contribute to the knowledge of this area by designing and implementing a messaging middleware over a transport protocol that provides exactly-once delivery guarantee.

1.2 Problem

The main challenge addressed in this thesis is providing exactly-once delivery guarantee. All design decisions must be made with this requirement in mind to ensure it is not violated. with this in mind, several key questions arise:

- Which features should the middleware provide?

- What should the API look like to support these features effectively?
- What architecture can comply with exactly-once delivery, support the selected features and maintain acceptable performance?
- How should the Exon transport library be integrated, and what modifications (if any) are required?
- What should the middleware's communication protocol look like, and how does the use of an exactly-once transport layer influence its design?
- How can message patterns be implemented while preserving exactly-once semantics?

1.3 Main Goals and Expected Results

Before starting to explore the complexities of creating a messaging middleware, it is crucial to outline the specific goals and tasks. These objectives aim to provide a general understanding about the roadmap taken during the investigation, conception and implementation of this messaging middleware solution:

1. Survey existing messaging middleware solutions, while focusing on analyzing communication patterns, architectural decisions, and features, in addition to specifically delving into the extent of reliability and fault tolerance offered to clients.
2. Adapt and restructure the Exon protocol library (Kassam et al., 2021), which provides exactly-once delivery guarantee, to support a messaging middleware in providing reliable communication.
3. Develop a prototype of a lightweight messaging middleware that offers reliable communication with exactly-once message delivery, resilience to network faults and versatility regarding messaging patterns. The term "lightweight" here implies that the messaging middleware will adopt a strategy similar to that of ZeroMQ and nanomsg, focusing on performance without the need for state persistence or transaction handling to recover from node faults.
4. Rethink and attempt to optimise communication patterns made available by existing messaging middleware solutions while aiming to enhance their versatility and adaptability for different situations.
5. Evaluate the prototype.

Overall, the expected result for the thesis was developing an innovative messaging middleware that enables and eases the development of resilient and versatile distributed applications.

1.4 Document structure

This section provides an overview of the document's structure.

The thesis is divided into two main parts: introductory material and core content.

In the first part, this introductory chapter contextualizes the problem and its challenges. Chapter 2 reviews several existing messaging middleware solutions that served as sources of inspiration. And, Chapter 3 presents the exactly-once delivery concept and the transport protocol selected to be the foundation of the devised solution.

The second part comprises seven chapters. The first of these, Chapter 4, is the main chapter of the thesis, presenting most design decisions made during the development of the messaging middleware and the rationale behind them. It is followed by Chapter 5 related to the specialized sockets that implement various messaging patterns, offering further insight into design choices and serving as inspiration for the creation of custom sockets. Chapter 6 documents the modifications made to the Exon library. Chapter 7 illustrates how the middleware can be used in practice and how it simplifies the implementation of specific scenarios compared to ZeroMQ. Chapter 8 presents performance metrics for the middleware, including comparisons with ZeroMQ and Exon. Finally, Chapter 9 concludes the thesis by summarizing the findings and contributions, and briefly discussing unresolved issues.

As a final note, an effort was made to highlight all code snippets, as in the following example: `helloWorld()`. When describing methods, UML syntax is used, while Java is used for code examples, as it was the language of choice to implement the messaging middleware.

Chapter 2

Message-Oriented Middleware Solutions

A Middleware is a tool that can be reused in multiple situations, and thus, its existence prevents developers from re-implementing the same thing over and over. In specific, a Message-Oriented Middleware (MOM), also called Messaging Middleware, is software that aims to facilitate the exchange of messages between distributed systems. This type of middleware strives to reduce the number of communication-related concerns of application developers, allowing them to focus exclusively on sending, receiving and processing messages, and stay clear from working directly with transport protocols, and other details involved in the creation of reliable communication systems.

In this chapter, we will be diving into the realm of messaging middleware solutions, exploring their features, architecture decisions, real-world applications, etc.

2.1 Overview of Messaging Middleware Concepts

This section will focus on providing a brief overview of messaging middleware concepts, that will be broadened through the whole chapter by the exploration of multiple messaging middlewares.

2.1.1 Interoperability

A key functionality that is usually provided by messaging middlewares is the capacity to enable the communication between heterogeneous components that may be running in different operating systems, may be implemented with different programming languages, etc.

Interoperability can happen at multiple levels, such as middleware protocol, application, message format, and more.

2.1.2 Support for a variety of transports

Messaging middleware solutions aid developers in getting different components/applications to start talking to each other. This can be done, solely by using transport protocols, however, these solutions can offer their users a centralized way[1] of using a variety of transports, along with other features built over the transports.

Some MOMs, in addition to supporting communication over different transport protocols, may also enable the integration of custom transport protocols.

By supporting a diverse set of transports, developers can easily create topologies using multiple heterogeneous transport protocols, leverage the strengths of each protocol within the topology, and consequently, optimize each connection.

2.1.3 Messaging Patterns

Messaging patterns, as the name suggests, are patterns that often appear in communication problems, some examples are: Request-Reply, Publish-Subscribe, One-way Pipeline (also known as Push-Pull), etc.

There are numerous Messaging Middleware solutions. Some focus on providing a generic API that contains the tools necessary for developers to create their messaging patterns. Others focus on providing out-of-the-box messaging solutions.

[1] By centralized way, I want to convey that instead of using the different APIs of each transport protocol, a developer may use the middleware's API to utilize the different transport protocols in a standardized way.

The messaging middlewares that provide messaging patterns are less versatile, however, they allow developers to quickly create functional topologies, instead of requiring communication specialists to create the messaging patterns from generic components.

In the following sections, we will explore different messaging middlewares, their messaging patterns, and if and how they allow the creation of custom messaging patterns.

2.1.4 Asynchronous Communication

A Messaging Middleware usually allows communication in an asynchronous way. By sending and receiving messages asynchronously, possibly in background threads, work can be done while messages are being sent/received.

2.1.5 Broker-based versus Brokerless

Some messaging middleware solutions require intermediary devices, called brokers, to work. These solutions usually consist of brokers[2] and clients, where the brokers act as a central piece of the topology and may have multiple functionalities, such as queuing and routing of messages, decoupling of clients[3], load balancing, persistence of session data, security, reliable message delivery, etc.

There are also messaging middlewares that do not require such intermediary devices. They are characterized as brokerless and spare developers from administration tasks that may be necessary when using brokers. Even though, they are called brokerless, it does not mean that brokers/intermediary devices cannot be designed using the functionality provided by these messaging middleware; in fact, that is usually the case.

2.1.6 Scalability and Load Balancing

The focus of a messaging middleware is to facilitate the creation of communication between components. By providing explicit messaging solutions, based on messaging patterns, the task of creating topologies becomes even easier.

By helping developers create topologies, quickly and easily, scaling a component horizontally is also a simple and quick task.

Additionally, messaging middlewares attempt to provide load-balancing functionality. This can be offered by using brokers, or, depending on the messaging pattern semantics, by the sockets themselves. For

2 In some cases, also addressed as servers.

3 Clients do not connect to other clients, preventing the need for clients to be online at the same time, or knowing about each other.

example, a messaging middleware that provides the Request-Reply pattern usually offers load-balancing capabilities in the socket used for sending requests. The requests-socket, connected to multiple services of the same type, can send requests fairly among all services, in an attempt to evenly distribute the workload among all services.

2.1.7 Fault Tolerance and Reliability

Messaging middlewares commonly offer fault tolerance and reliability measures.

There are multiple types of faults, but we will pay special attention to network faults as messaging middleware is closely related to network concerns. We will also talk about node faults.

We saw above that messaging middlewares usually allow scalability of components and load balancing. These are redundancy and fail-over mechanisms that help face node failures and network partitions. By scaling a component horizontally, the services can still be provided by redirecting requests to a functional instance. Load balancing helps distribute the workload between the same type of components, preventing components from being rendered useless due to resource exhaustion.

There are also other features that can help develop a fault tolerant as well as reliable distributed system. Some of them are persistence, delivery guarantees, and message and connection retries.

Persistence

Not all messaging middlewares provide persistence capabilities. Generally, the ones that provide such capabilities are broker-based, with the persistence capabilities being primarily implemented on the broker/server, but can also be present on the clients.

The persistence functionality allows the developer to define if data (such as messages) should be persisted or not. And if persistence is required, where it should be stored: in a volatile (memory) or non-volatile (e.g. disk) storage. The amount of time the data may be persisted will depend on the semantics of the actual middleware. But may also be a configurable option.

This feature helps a messaging middleware recover from node faults, like a crash, and ensure message delivery reliability, for example, by persisting the messages directed to a client while it is offline after an abrupt disconnection.

Delivery Guarantees

A messaging middleware may let developers choose the delivery guarantee required for the exchange of messages, be it per message, per connection, or per another factor.

Delivery guarantees refer to the requirements regarding the delivery of messages. Some common delivery guarantees are:

- **At-most-once:** This delivery guarantee ensures that a message reaches the intended recipient at most one time. When this guarantee is selected, a message is sent only one time, and discarded right after.
- **At-least-once:** This delivery guarantee assures that a message is delivered to the intended recipient at least one time. This is usually achieved by sending a message until an acknowledgement of the receipt is received. Duplicates of the messages may be received.
- **Exactly-once:** This delivery guarantee ensures that a message is delivered to the target application exactly one time. This is required when the delivery of messages is mandatory but the reception of duplicated messages is not tolerated by the application. This guarantee is usually achieved through a 4-way handshake. An in-depth explanation of a 4-way handshake is given in a later chapter referent to the Exon transport protocol.
- **In order delivery:** This delivery guarantee assures that messages are delivered in the order they are sent. For example, if Alice sends 2 messages, A and B, to John, following the presented order, then the messages must be delivered to John, for processing, in the same order, i.e., A should be delivered first and then B.

The delivery guarantees can be coupled with the persistence mechanism to ensure that the delivery guarantee holds under node/process failures.

Retries

We will be talking about 2 types of retries, that can be provided by a message-oriented middleware: message retries and connection retries.

Message retries usually happen when a response is expected after sending a message, however, there can be other situations that result in a retry. After some delay, if the response has not yet been received, the message is resent. The message is resent until an answer arrives. Depending on the delivery guarantee or some administration rules, after an X amount of tries, the message may not be resent.

The other type of retries is related to connections. After an unexpected disconnection occurs (i.e., when one of the peers does not inform the other that it will disconnect), the messaging middleware may attempt to reestablish the connection automatically (auto-reconnect).

These retries are important to recover from both node and network faults.

2.1.8 Message Routing:

A Messaging middleware contemplates message routing mechanisms. Generally, these mechanisms are use-case-specific. For example, in a Publish-Subscribe the routing mechanism may be related to the subscriptions, but a request-reply pattern does not know what a subscription is, therefore its routing mechanism cannot be based on subscriptions.

The routing mechanism may provide filtering capabilities based on some factors. In the Publish-Subscribe example above, the routing mechanism checks the subscriptions and filters the subscribers who are not interested in the message by checking their subscriptions.

The routing mechanism may also provide load balancing, for example using a round-robin algorithm. It can even use the concept of priorities[4], allowing certain destinations to be prioritized, leaving others, with lower priorities, as fail-overs.

2.1.9 Security

When working on the communication between components, security is usually one of the developers' concerns. Because of this, messaging middlewares are normally equipped with security functionalities, such as authentication/authorization, encryption and integrity check of messages, blacklisting of IPs (to prevent DoS[5] attacks), etc. These mechanisms prevent access and block users with malicious intent, and ensure that private data is safely transmitted.

4 Present in nanomsg (2018) middleware.

5 DoS, short for Denial of Service, is a type of attack that consists in overloading the target machine with requests, preventing it from working normally.

2.2 ZeroMQ

ZeroMQ (also spelled as ØMQ, 0MQ, or zmq) is known for being a brokerless scalable and high-performance messaging library that allows the development of distributed and concurrent applications. It offers a BSD sockets-style API, well known by developers, in order to ease its use and integration. It allows the transmission of messages through various transport protocols such as TCP or multicast, and even across processes. With an extensive range of language APIs compatible with various operating systems, it enables communication between all sorts of programs. Additionally, it supports common communication patterns that simplify the construction and structuring of network topologies.

2.2.1 Key Features and Architecture:

- **Asynchronous messaging:** ZeroMQ provides asynchronous messaging which enables it to perform well under high concurrency workloads
- **High-performance:** ZeroMQ has a sophisticated implementation that implements multiple optimizations having high performance in mind (Martin Sustrik, 2012). Some of the optimizations are:
  - Avoiding unnecessary memory allocations that hinder performance. For example, depending on the size of the messages, copying the content to preallocated memory may be cheaper than allocating memory space and passing the pointer to the allocated block (small *vs* big messages).
  - Batching of messages when the message rate exceeds the bandwidth of the network stack. Latency is the priority, and to optimize it, batching is disabled initially. But when data arrives faster than it can be processed, batching is used to improve the throughput.
  - I/O in background threads[6], as opposed to the traditional one thread per socket.
  - Following the classic actor model, i.e., the communication between threads is made using asynchronous messages (events).
  - Use of lock-free algorithms to avoid idle time caused by lock contention.
  - Employs a highly complex object scheduler, that manages the ZeroMQ objects. All ZeroMQ objects are event-driven and tightly bound to a worker thread.

[6] As default only one background thread is initiated upon the creation of a ZeroMQ context. However, the number of background threads is configurable. As a rule of thumb, there should be one I/O background thread per gigabyte of data in or out per second.

  - In addition to all optimizations above, by applying a limit of one thread per CPU core, the worker threads can run at full speed, without being slowed by context switches, locks, or other overhead.

- **Brokerless:** ZeroMQ operates in a peer-to-peer model, eliminating the need of dedicated message brokers.

- **Supports a variety of transports:** TCP, in-process, inter-process, multicast, WebSocket, and more, although for most ZeroMQ engines the focus is the TCP transport protocol.

- **No need for administration:** By being a library instead of an application, ZeroMQ removes the need for non-trivial administrative efforts.[7]

- **Payload agnostic:** For ZeroMQ, messages are opaque binary data that fit in memory, enabling developers to choose their own data representation (the developers need to perform their own data serialization).

- **Multipart messages:** A ZeroMQ message consists of frames, also called message parts. A ZeroMQ frame is a block of binary data and its corresponding size in bytes. ZeroMQ ensures that either all parts of a message are delivered, or none of the parts are delivered. Both single and multipart messages need to fit in memory.

- **Socket-based API:**

  - ZeroMQ offers a socket-based API that promotes ease of use due to its familiar format to developers.

  - The sockets transfer discrete messages, which differ from conventional sockets that either transfer a stream of bytes or datagrams.

  - Provides a variety of socket types, each designed for a specific communication pattern. It is important to notice that an incorrect pairing of sockets will result in a non-functional application.

  - A socket presents an abstraction for an asynchronous message queue. The queue semantics differ based on the socket type as different communication patterns require different handling of messages.

[7] Affirmation inspired in Martin Sustrik's words found at Martin Sustrik (2012).

- **High-Water-Mark:** The High-Water-Mark is the limit of messages queued in memory for a single peer that is communicating with a specific socket. Different socket types implement different ways of handling the messages when the limit is reached, which consists in either dropping messages or blocking the socket.

- **Auto-reconnect:** ZeroMQ features automatic reconnect. This allows components to be dynamic, they can appear and disappear, and ZeroMQ will manage the reconnection.

- **One socket, multiple connections:** In ZeroMQ, the majority of socket types, allow multiple connections using a single socket. These sockets greatly simplify communication with multiple peers by eliminating the need to manage numerous individual sockets, while also providing additional functionalities like load-balancing.

- **One socket, multiple endpoints:** A ZeroMQ socket can bind to many endpoints using a single socket, this translates to the acceptance of connections across different transport protocols.

2.2.2 Communication Patterns and Sockets:

In this section, we will explore the multiple communication patterns offered by ZeroMQ, and take a closer look at the respective sockets of each pattern.

The patterns offered by ZeroMQ are:

- **Request-reply**, which is designed for service-oriented architectures, by enabling a set of clients to connect to a set of services.

- **Pub-sub**, which is intended for data distribution, connects a set of publishers to a set of subscribers.

- **Pipeline**, which is oriented parallel task distribution and collection, allows setting up a series of stages where the messages flow unidirectionally from one stage to the next.

- **Exclusive pair**, which is oriented for communication between two threads that belong to the same process. Connects two sockets exclusively.

Currently in a draft stage are additional patterns that are fundamentally a thread-safe version of existing ones:

- **Client-server**, which is designed to connect a single server to multiple clients. This pattern will allow an asynchronous exchange of messages between client and server.

- **Radio-dish**, which aims to provide distribution of data from publishers to subscribers by using group memberships.

Request-Reply pattern

- Allows a set of clients to connect to a set of servers.
- Used for remote procedure calls and task distribution.
- Two variants: synchronous (*REQ* and *REP* socket types) and asynchronous (*DEALER* and *ROUTER* socket types).

  REQ socket

  - Used by a client to send requests to and receive replies from a service.
  - Only allows an alternating sequence of send and subsequent receive calls.
  - Can connect with any number of REP and ROUTER sockets.
  - Uses round-robin to decide which (connected) service should receive the request. The reply must be received from the service that received the request.
  - Send operations will block until at least one service becomes available.
  - This socket type does not discard any messages.

  REP socket

  - Used by a service to receive requests from and send replies to a client.
  - Only allows a sequence of send and subsequent receive calls.
  - Compatible with REQ and DEALER sockets.
  - Requests are fairly queued among all clients.
  - A reply is routed to the client that issued the most recently processed request. If the client no longer exists, the message is discarded.

  DEALER socket

  - Talks to a set of anonymous peers. Peers are considered anonymous since the DEALER communicates with peers without having explicit knowledge or control over which specific peer will receive a particular message.
  - Sends messages using a round-robin algorithm.

- Received messages are fairly queued among all connected peers.
- Does not have any restrictions regarding the sequence of send and receive operations. This socket is the asynchronous version of REQ.
- This socket can interact with REP, ROUTER, and DEALER sockets.
- When no peers are connected, or when the socket enters a mute state due to having reached the high water mark, all send operations will block until a peer connects or the mute state ends. The socket does not drop any messages.

ROUTER socket

- Talks to a set of peers, but unlike the DEALER socket, each message is delivered using explicit addressing to ensure it reaches the desired peer.
- When receiving a message, adds routing identification of the source peer to the message.
- When sending a message, extracts and removes routing identification from the message, and uses it to deliver the message to the (next) intended target. If the peer no longer exists, discards the message.
- This socket can work as the asynchronous substitute of REP.
- Messages (received) are fair-queued among all connected peers.
- If the high water mark for all peers is reached, the socket enters a mute state, discarding any message sent to the socket until the mute state ends.
- If a message is routed to a peer for which the individual high water mark has been reached, then the message is also dropped.

Socket type	**REQ**	**REP**	**DEALER**	**ROUTER**
Compatible peer sockets	REP, ROUTER	REQ, DEALER	REP, ROUTER, DEALER	REQ, DEALER, ROUTER
Direction	Bidirectional	Bidirectional	Bidirectional	Bidirectional
Send/receive pattern	Send, Receive, Send, Receive, ...	Receive, Send, Receive, Send, ...	Unrestricted	Unrestricted
Outgoing routing strategy	Round-robin	Fair-queued	Round-robin	See text
Incoming routing strategy	Last peer	Last peer	Fair-queued	Fair-queued
Action in mute state	Block	–	Block	Drop

Table 1: Summary of Request-Reply sockets' characteristics

Publish-Subscribe pattern

- Used for distribution of data from publishers to subscribers.

- The distribution of data is based on topics (represented as an array of bytes at the beginning of the message[8]).
- Subscribers, as the name suggests, have to specify the topics they are interested in through subscriptions.
- A subscriber receives a message if the topic of the message has a prefix that matches any of the subscriber's subscriptions.
- The filtering (by topic) operation may occur on the publisher or subscriber side. The side in which it occurs depends on the protocol used for the communication. If a connection-based protocol is used, such as TCP or IPC, the subscriptions of the specific subscriber may be associated to the connection on the publisher side, allowing the filtering to be performed by the publisher. However, when using a multicast protocol, the publisher cannot maintain peer-specific information, and so, the filtering is required to happen on the subscriber side.

PUB socket

  - Used by publishers to distribute data, by providing a basic way of one-way broadcasting.
  - Follows a best-effort delivery approach, i.e., messages are sent only one time. Should only be used when the loss of messages is not problematic and consistent low memory usage is preferred.
  - When using a connection-based protocol, outgoing messages are sent several times, one time per subscriber that has a subscription that matches the message topic.
  - Cannot receive any messages (other than subscriptions). Any messages received will be silently dropped.
  - Subscribe and unsubscribe messages are not exposed to the application.
  - If the high water mark is reached for a particular subscriber, the send operation is not blocked, instead, any messages directed to that subscriber are dropped until the mute state for that subscriber ends.
  - Can connect to SUB and XSUB sockets.

SUB socket

  - Used by a subscriber to subscribe to data distributed by publishers.

[8] The topic may be isolated from the data using multipart messages, however, it should be present in the first part.

- Initially does not have any subscriptions.
- Can only send subscribe and unsubscribe messages. Not allowed to send any data messages.
- Compatible with PUB and XPUB sockets.
- Incoming messages are received using a fair-queuing strategy.
- Incoming messages are silently dropped when the publisher's queue is full.

XPUB socket

- Identical to the PUB socket, but can receive data messages and expose the subscribe/unsubscribe messages to the application.
- Exposing the subscribe/unsubscribe messages to the application allows the creation of proxies.
- Similarly to the PUB socket, it does not block the sending operation, therefore messages are silently dropped if the queue of a subscriber is full.
- Incoming messages are received using a fair-queuing strategy.
- Can connect to SUB and XSUB sockets.

XSUB socket

- Extends the SUB socket with the ability to send messages.
- Unlike the SUB socket, to subscribe, a subscription message needs to be created explicitly using the appropriate format.
- Incoming/outgoing messages are silently discarded if the publisher's incoming/outgoing queue is full.
- Incoming messages are received using a fair-queuing strategy.
- Does not block on send.
- Compatible with PUB and XPUB sockets.

Socket type	PUB	SUB	XPUB	XSUB
Compatible peer sockets	SUB, XSUB	PUB, XPUB	SUB, XSUB	PUB, XPUB
Direction	Unidirectional	Unidirectional	Unidirectional	Unidirectional
Send/receive pattern	Send only	Receive only	Send messages, receive subscriptions	Receive messages, send subscriptions
Outgoing routing strategy	Fan out	—	Fan out	—
Incoming routing strategy	—	Fair-queued	—	Fair-queued
Action in mute state	Drop	—	Drop	Drop

Table 2: Summary of Publish-Subscribe sockets' characteristics

Pipeline pattern

- Directed for task distribution, usually with a series of stages where messages flow unidirectionally, with work being pushed to workers and, afterwards, the results pushed to collectors.
- Reliable in the sense that a message is not dropped unless a node disconnects unexpectedly.
- This pattern is highly scalable due to nodes being able to join at any time.

PUSH socket

- Connects and sends messages to a set of anonymous PULL peers. Similarly to what was mentioned earlier for the DEALER socket type, the peers are considered anonymous since the socket does not provide a mechanism to address a specific peer explicitly.
- Messages are sent using a round-robin algorithm.
- Cannot receive messages since this pattern focuses on unidirectional flow of data.
- When the high water mark for all peers is reached, or if there are no connected peers, the send operations will block. As mentioned above, no messages are dropped unless a node fails unexpectedly.

PULL socket

- Talks with a set of anonymous PUSH peers.
- Can only receive messages, i.e., the send operation is not available for this socket.
- Receives messages using a fair-queuing algorithm.

- No messages should be dropped due to memory exhaustion. This is because a PUSH socket will not send a message to a PULL socket if the outgoing queue of the PUSH socket is full.

Socket type	**PUSH**	**PULL**
Compatible peer sockets	PULL	PUSH
Direction	Unidirectional	Unidirectional
Send/receive pattern	Send only	Receive only
Outgoing routing strategy	Round-robin	—
Incoming routing strategy	—	Fair-queued
Action in mute state	Block	Block

Table 3: Summary of Pipeline sockets' characteristics

Pair pattern

- Used to connect two peers exclusively.
- Designed for specific use cases where two peers are architecturally stable.
- Can only be used for inter-thread communication, within one process.

PAIR socket

- Can only be connected to one peer at a time.
- Enters mute state when the high water mark is reached, or when no peer is connected. When in this state, the send operation will block, therefore, no messages are discarded.
- Due to this socket's inability to reconnect and because any new incoming connections are rejected while a previous connection exists, TCP is not a suitable protocol for this socket.

Socket type	**PAIR**
Compatible peer sockets	PAIR
Direction	Bidirectional
Send/receive pattern	Unrestricted
Outgoing routing strategy	—
Incoming routing strategy	—
Action in mute state	Block

Table 4: Summary of PAIR sockets' characteristics

Notes on ZeroMQ sockets

- *Bind vs connect:*
  - ZeroMQ sockets, similarly to BSD sockets, provide *bind* and *connect* functions. However, unlike BSD sockets, who binds and who connects, and when they do, do not carry the same weight[9] due to the auto-reconnect feature. This is because a client trying to contact a server, for example, using TCP sockets, will fail to connect if the server is not already bound. However, a ZeroMQ socket can try to connect even if the socket that should bind has not done it yet. This is possible due to the auto-reconnect feature, which periodically attempts to reconnect until it succeeds.
  - The difference between connecting and binding, for ZeroMQ sockets, lies in the creation of the underlying message queues. When a socket is bound, the socket cannot know how many peers will be connecting to it, therefore no queues are created upfront. In contrast, when connecting, the socket knows that there will be at least a single peer, for that reason, it can create the queue(s) immediately, allowing messages to be queued even if no connection has been established.

2.2.3 Message Transport Protocol ZMTP

ZeroMQ employs its own transport layer protocol, named ZeroMQ Message Transport Protocol (ZMTP). This protocol is designed for exchanging messages between two peers over a **connected** transport protocol

[9] Even if, for ZeroMQ, there is no difference regarding who binds or who connects, as a general rule, if one socket is architecturally stable, it should be the one binding.

like TCP. The ZMTP is specified as a formal protocol to allow the implementation of the protocol on any platform and any language. This section will focus on version 3.0 of the ZMTP protocol, as described by RFC 23 (ZeroMQ, a) found in ZeroMQ (b).

Goals

The ZMTP protocol was developed to solve several problems faced when using TCP:

- TCP works with streams of bytes with no delimiters, but as mentioned above, ZeroMQ sockets exchange discrete messages. ZMTP reads and writes frames that consist of a size and body.
- Provides a *flags* field, on each frame, which is required to transport metadata (about the frame). An example of a flag is if the frame is part of a multipart message.
- ZMTP specifies a greeting that announces the implemented version number and defines how the version is negotiated. This is required to allow the framing of messages to evolve without turning older implementations obsolete.
- ZMTP defines a security handshake, for peers to create secure connections, and a range of security protocols (from fast with no security, to slow with security).
- ZMTP defines a standard set of metadata properties related to the connection (such as socket type, identity, etc) that peers exchange after agreeing on the security mechanism.

Implementation

A ZMTP connection is defined by 4 main stages:

- **Greeting:** The two peers exchange data in order to reach an agreement regarding the protocol version and the security mechanism of the connection. If the peers do not reach an agreement, the connection is closed.
- **Security handshake:** The two peers handshake the security mechanism by exchanging zero or more commands[10]. Any error during this stage will result in one or both peers closing the connection.
- **Metadata exchange:** Each peer sends metadata about the connection. After the verification of the received metadata, each peer decides either to continue or close the connection.

[10] The number of commands and which commands are exchanged depend on the security mechanism which the peers agreed upon.

- **Data exchange:** Finally, the peers can send messages to each other. The connection may be closed at any moment.

Version Negotiation The ZMTP protocol offers two possible approaches regarding version negotiation. A peer may attempt to detect and work with previous protocol versions[11], or it can require capabilities from its peers[12].

When negotiating the protocol version:

- A peer must always use its own protocol (including framing) when talking to a peer using a protocol version that is equal or higher;
- A peer may lower its protocol version to talk with a lower protocol peer. If the peer is not capable of lowering its version, then it must close the connection.

Topology The ZMTP protocol is a peer-to-peer protocol, thus it does not need peers to have a role such as client and server. However, the ZMTP protocol allows its behaviour to be extended by extension security protocols, which may require the peers to have such roles. Having said that, this information may be sent using the 'as-server' field which is part of the greeting frame.

Security While acknowledging the importance of security, authentication, and confidentiality, these are not the main focus of this research and will be put aside for future consideration.

Error Handling During the security mechanism handshake, the peers are allowed to respond with an ERROR command. This command must be faced as fatal, thus making the receiving peer disconnect and not attempt to reconnect using the same security credentials. Any other error (temporarily unavailable, overloaded, etc), must be signalized by closing the connection. A peer must handle an unexpected end of the connection as a temporary error, and attempt to reconnect after a randomized delay. After each failed attempt of reconnection, the peer should increase the delay.[13]

Framing Aside from the greeting frame which has a fixed size of 64 bytes, the remaining data are sent as frames, that can carry either a command or a message. A frame consists in a *size* field[14], a *flags* field[15] and a body. The flags field is made of 3 flags:

[11] In this first approach, the peer starts by sending only the first part of the greeting frame that is necessary to trigger a version detection.

[12] To demand capabilities from its peers, the peer sends the entire greeting frame.

[13] The randomization and increase of the delay after each reconnection are measures that should be taken to avoid the thundering herd problem.

[14] Refers to the size of the body rather than the frame.

[15] The *flags* field has the size of a byte, however, at the moment, only 3 bits are used, the remaining bits are reserved for future purposes.

- "Command frame": Indicates if the frame is a command or a message;
- "Long frame": Indicates if the *size* field is represented using 1 byte or 8 bytes.
- "More frames to follow": Indicates if more frames will follow. (for multipart messages)

Commands Commands are generally not visible by the application, except in some cases. Commands consist of just one frame, which contains the name of the command, a null octet separator, and the data.

Essentially, the commands are only used by the security mechanisms, therefore, the security protocols may use any command names they need to. When the peers agree to use no security mechanism, they are actually agreeing to use the called 'NULL security mechanism'. The 'NULL security mechanism' is a very simple mechanism that only defines two commands: READY and ERROR. The ERROR command was explained above, and the READY command is used by the peers to exchange the connection metadata required to decide if the connection should be closed, or if it may remain open.

Messages Messages carry application data. These messages may be formed by one or more frames (multipart messages), which must be sent/delivered in an atomic way, i.e., either all frames are sent/delivered, or no frame is sent/delivered.

Connection Metadata The security mechanisms must allow the peers to exchange metadata. Each mechanism is free to define its metadata properties apart from:

- "Socket-Type": used to specify the socket type of the source. This property is mandatory, as each peer needs to validate if the other peer's socket is compatible. If the socket is not compatible, the peer must send an ERROR command and close the connection.
- "Identity": used to define the identity of the source's socket. When a compatible socket connects to a ROUTER socket, it may specify its identity[16]. The identity is used by sockets of type ROUTER as an addressing mechanism.

Socket semantics

Each socket has specific semantics related to the communication pattern it implements. However, there are a few rules that apply to every ZeroMQ socket:

[16] If no identity is specified, the ROUTER will automatically assign one.

- They must accept connections (when binding to an address) and make connections (i.e., connect to other peers when not bound to an address);
- When a connection is broken, they must attempt to reconnect after an adequate delay;
- Messages should be sent and delivered in an atomic way[17];
- A message should not be delivered more than once to any peer;
- And, all messages between two adjacent peers should be delivered in an ordered manner.

2.2.4 Fault Tolerance and Reliability:

ZeroMQ is a messaging system, that aims to facilitate the job of developers when it comes to communication between applications or nodes. Therefore, it does not provide any recovery mechanisms regarding node failures, leaving that to be implemented by the application developer. Instead, it focuses on providing network fault tolerance and reliability.

For fault tolerance, it provides an auto-reconnect feature. Following a connection failure, sockets will continuously attempt to reconnect, after a delay, until the connection is re-established, allowing the communication to be resumed.

Regarding reliability, it is the result of a combination of the delivery guarantees provided by the underlying transport protocol and ZeroMQ's message management (which depends on the semantics of the socket type). ZeroMQ operates on a best-effort delivery model. Once a message is delegated to the transport layer, it is immediately removed from the outgoing queue, which means that automatic retransmission capabilities are not provided. The main transport protocol used by ZeroMQ is TCP which provides a reliable delivery of messages within a connection, however, if the connection fails while messages are being sent, this will result in the loss of those messages. Additionally, the reliability is influenced by the socket semantics, potentially leading to messages being dropped when the queues reach full capacity. Furthermore, replies and queued messages associated with a disconnected client are usually discarded.

The offered reliability is not very assuring. Therefore, a range of (Request-Reply) reliable patterns, that can be implemented over ZeroMQ, are presented in chapter 4 of the ZeroMQ Guide (ZeroMQ, c).

2.2.5 Limitations and Challenges:

In this section, we will dive into the limitations and challenges of the ZeroMQ communication library.

[17] When sending a message, all frames should be queued in memory until the last frame (of the message) is sent.

Sharing of global state

ZeroMQ is a communication library that uses the concept of context. "Technically, the context is the container for all sockets in a single process, and acts as the transport for inproc sockets, which are the fastest way to connect threads in one process. If at runtime a process has two contexts, these are like separate ZeroMQ instances." (ZeroMQ, c, Chapter 1, Section "Getting the Context Right")

While for most cases it makes sense not to have shared global state, the same cannot be said for a communication library. The purpose of communication is the sharing of information. By allowing the creation of local contexts, modules that should be able to observe each other, become isolated. The removal of ZeroMQ-like contexts simplifies the API and removes unnecessary security risks by allowing all INPROC[18] addresses to be visible within the whole process, instead of being restricted by the local context. (Martin Sustrik, 2013)

Support for device mobility As one can observe by ZeroMQ's popularity, it is a useful communication library that can be employed in multiple scenarios. However, it is not appropriate when it comes to a mobility context. Consider a car (connected to a service) that is moving from one city to another. At some point, it moves out of the coverage area of one network station and thus needs to transition to another station to maintain continuous connectivity. This transition, results in the change of the IP address of the vehicle, and consequently, in losing the messages that were already queued.

Sockets are not thread safe Using the ZeroMQ library involves adapting to its programming model, the actor model. ZeroMQ sockets are not thread safe, and therefore, should only be assigned to a single thread. A classic approach which involves sharing a socket between message generating/consuming threads is not allowed with ZeroMQ sockets. ZeroMQ good practices suggest that threads communicate with each other using messages, i.e., through ZeroMQ sockets (with INPROC addresses). These practices, initially, are strange for a developer who is not used to the programming model, i.e., a developer used to object-oriented programming (e.g. Java), will have to change his way of thinking, compared to a developer who already has experience with Erlang, which follows the same programming model as ZeroMQ. At the moment, there are already some ZeroMQ communication patterns being designed, which are essentially a thread-safe version of already existing communication patterns.

[18] INPROC refers to the ZeroMQ's transport protocol for message communication within a process.

2.3 nanomsg

From my research, I found that *Martin Sustrik* is the creator of both ZeroMQ and nanomsg[19]. In his journey of developing ZeroMQ, valuable lessons were learned. These lessons evolved to a new project called nanomsg (nanomsg, a), which aimed to address ZeroMQ's flaws[20] and undefined behaviours, and further optimize its performance.

Having that in mind, this section will focus on providing insight about key similarities and differences between ZeroMQ and nanomsg.

2.3.1 General Comparison of nanomsg with ZeroMQ

The information presented in this section was extracted from multiple sources: nanomsg (2018), nanomsg (b) and nanomsg (c).

Communication library nanomsg, just like ZeroMQ, is a brokerless communication library that provides a socket API, designed to help developers build fast and scalable distributed applications.

Messaging Patterns nanomsg provides a variety of messaging patterns, including the patterns provided by ZeroMQ[21], and two additional patterns, Bus and Survey. The key implementation differences between the shared patterns, and the new patterns will be explained in a following section.

Transport Protocols Similarly to ZeroMQ, for unicast messaging, nanomsg provides as transport protocols:

- **INPROC:** For communication between threads, and modules, within a process.
- **IPC:** For inter-process communication within the device.
- **TCP:** For network communication using TCP.
- **WS:** For Web-sockets communication (via TCP).

.

[19] Can be confirmed in the "Documentation" page of nanomsg (2018).

[20] All Martin Sustrik's articles can be verified in Sustrik (2011-2022) (only a few are related to ZeroMQ and nanomsg). In special, the article where he explains why ZeroMQ should have been written in C instead of C++ can be found in Sustrik (2012).

[21] The patterns provided by ZeroMQ are Pair, Request-Reply, One-way pipeline and Publish-Subscribe.

Transports and Protocols extensibility Unlike ZeroMQ, nanomsg provides a way for developers to integrate new transport protocols, as well as develop custom messaging protocols[22]. This enables the extension of the middleware beyond the provided transport protocols and communication patterns.

Concept of context In ZeroMQ's limitations section, we verified that Martin Sustrik was no longer a fan of contexts (objects that hold global state variables). This concept was completely removed in nanomsg[23].

Thread-safe sockets nanomsg addresses what can be considered a limitation in ZeroMQ. nanomsg provides thread-safe sockets, a feature that is lacking in ZeroMQ.

Routing priorities ZeroMQ's load-balancing mechanism is based on a round-robin algorithm. While it may provide equal distribution of tasks[24], it does not address an issue that nanomsg solves. ZeroMQ's load-balancing flaw is assuming that all destinations are equal. Destinations may differ in resource capacities, location, etc. nanomsg provides a way of setting routing preferences, and fall-back destinations when higher-priority destinations are not available.

No ROUTER-like socket nanomsg does not offer a socket with routing capabilities similar to ZeroMQ's ROUTER socket, and suggests the implementation of this functionality using transport layer protocols. "There's no generic UDP-like socket (ZMQ_ROUTER), you should use L4 protocols for that kind of functionality." (nanomsg, c)

Unified Buffer Model "ZeroMQ has a strange double-buffering behaviour. Both the outgoing and incoming data is stored in a message queue and in TCP's tx/rx buffers. What it means, for example, is that if you want to limit the amount of outgoing data, you have to set both ZMQ_SNDBUF and ZMQ_SNDHWM socket options. Given that there's no semantic difference between the two, nanomsg uses only TCP's (or equivalent's) buffers to store the data." (nanomsg, c). From the previous citation, I understand that nanomsg discarded the concept of queues present in ZeroMQ, opting for the sole use of the transport layer buffers.

No multipart messages ZeroMQ offers a way to create messages with multiple independent parts. This feature is not present in nanomsg.

[22] I.e., allows the creation of custom communication patterns.

[23] This can be verified by checking nanomsg's manual (nanomsg, b) which does not contain any method related to the creation/destruction of contexts.

[24] ZeroMQ does not provide equal distribution of work, because the granularity of the tasks is not considered.

2.3.2 Communication Patterns:

As mentioned earlier, nanomsg includes ZeroMQ communication patterns: Pair, Request-Reply, Pipeline and Publish-Subscribe. For these patterns, we will be exclusively addressing the differences with respect to ZeroMQ. The functionality provided is essentially the same, only the way of programming needs to be adapted to the new API. As for the new communications patterns, Bus and Survey, we will briefly explore how they work.

Request-Reply pattern

The key differences between the two middlewares, regarding the Request-Reply pattern, are:

- nanomsg only provides 2 socket types, REQ and REP. These sockets are originally synchronous just like ZeroMQ's REQ and REP sockets. There are no DEALER and ROUTER sockets in nanomsg, therefore, when in need of similar functionality, the developer must use the RAW[25] versions of these sockets. However, the developer will be responsible for handling the message headers which contain the routing information required to deliver the answer to the intended target.

- When using the non-RAW version of these sockets, the nanomsg's REQ and REP sockets offer a way to cancel the ongoing processing. To accomplish this: a REQ process may send a new request without waiting for a reply; as for a REP socket, it may retrieve and start processing a new request without responding to the previous one.

Publish-Subscribe pattern

For this pattern, the difference between nanomsg and ZeroMQ lies in:

- nanomsg only performs message filtering on the subscriber side, while ZeroMQ may perform on the publisher or subscriber side (depending on the underlying transport protocol). This means that all messages are sent to all subscribers over the network.

- There are no XPUB and XSUB sockets used to create proxy/intermediary devices. However, it is still possible to create intermediary devices. The intermediary devices connect to publishers with a SUB socket and subscribe to all messages, and then forward them to subscribers using a PUB socket.

[25] Working with the RAW version of a socket, means omitting end-to-end functionality provided by the non-RAW version of the socket. Therefore, the developer will be responsible for handling the framing, protocol-specific behaviour, and other low-level details.

- nanomsg implements a data structure, used to store and match subscriptions, that is far superior regarding memory efficiency, called Patricia Trie. More can be learned about the structure in the following article: Sustrik (2013a).

Bus pattern

The Bus pattern (nanomsg, d) was designed for many-to-many communication and routing. The pattern provides a BUS socket that broadcasts messages to all peers belonging to the topology. The RAW version of the BUS socket can be used to route messages to all peers (of the topology) except to the one from which the message was received.

Survey pattern

The survey pattern helps developers query all connected devices. Instead of using a Request-Reply pattern and sending a request individually for each device, the survey pattern allows broadcasting a query and collecting all the responses within a time frame.

The survey pattern offers two sockets:

- **SURVEYOR:** Used to start a survey, and deliver it to all connected devices. Surveys can be answered within a time interval. When the time runs out, if any respondent does not answer during that interval, an error is returned.
- **RESPONDENT:** This socket is used to answer the survey and may be connected to just one device.

2.3.3 Fault Tolerance and Reliability:

Similarly to ZeroMQ, nanomsg is a communication library that focuses on offering a solution that eases the creation of topologies and the set-up of communication between devices. Consequently, the reliability and delivery guarantee provided by nanomsg is determined by the underlying transport protocol[26]. Just like ZeroMQ, nanomsg provides an auto-reconnect feature, however, that seems to be the only mechanism related to network faults. Regarding node faults, nanomsg leaves the responsibility of implementing mechanisms that provide fault tolerance, such as persistence, for the developers that use the library.

[26] nanomsg does provide a formal internal API that enables developers to add other transport protocols, depending on their needs.

2.3.4 Limitations and Challenges:

Regarding nanomsg's limitations and challenges, nanomsg is similar to ZeroMQ, and so are its limitations and challenges (excluding the limitations addressed by this project).

Some of ZeroMQ's limitations solved by nanomsg are: the lack of sockets that can be safely used by multiple threads; and the need to create/destroy contexts.

The limitations of nanomsg are the following.

Poor reliability The reliability offered by nanomsg is determined by the underlying transport protocol. Let us take the TCP protocol as an example, which is the main network transport protocol of nanomsg. This protocol provides a reliable and ordered stream of bytes, ensuring exactly-once delivery within the connection, however, when the connection fails, all messages are lost as they are only stored in the transport protocol buffer. The auto-reconnect feature helps resume the communication but all the messages from the previous connection that did not reach the recipient are lost, unless the application developer, while aware of this limitation, deliberately saves the messages.

Lack of support for device mobility Due to the poor reliability and lack of client take-over mechanism, and because only trivial state may be kept per connection (no backups are made for the important content, the messages), nanomsg is not an ideal solution for mobility scenarios where disconnections are frequent.

2.4 NNG

NNG (D'Amore), also known as 'Nanomsg Next Generation', is a lightweight and brokerless communication library that succeeded nanomsg. While preserving certain features of nanomsg, this messaging middleware introduces some new capabilities along with optimizations and improvements over its predecessor's functionality (D'Amore, 2020).

NNG is the consequence of lessons learned by working on the nanomsg project. Having said that, to avoid being repetitive, only relevant information and key differences between NNG and nanomsg will be presented.

In this section, we will be taking a look at key features provided by the middleware, followed by architectural decisions related to sockets, message handling, asynchronous operations, transports and communication patterns, and finish with observations related to fault tolerance, reliability, and limitations.

2.4.1 Key Features

As announced on the official website (NNG, 2024), the following are the main features of the NNG middleware:

- **Reliability:** Reliability that is associated with crash-resilience. The middleware tries to cover all cases that may lead to crashes. As for the message delivery reliability, similarly to nanomsg, it depends on the chosen transport protocol.
- **Scalability:** NNG achieves scalability by taking advantage of the multiple cores present on the machine through a specific asynchronous I/O framework. The framework allocates a pool of threads that share the work without exceeding the system limits.
- **Maintainability:** Designed with a modular architecture that can be easily grasped by other developers. NNG follows a different approach from nanomsg, which resorted to state machines. Another approach was taken since tracking the flow of the state machines was considered extremely difficult.
- **Extensibility:** In addition to providing, by default, all transports implemented by nanomsg[27], NNG allows the creation and integration of new protocols (messaging patterns) and transports.
- **Security:** Provides security protocols for authentication, encryption, and protection against malicious attacks.

[27] The transports implemented, by default, by both nanomsg and NNG are: INPROC (intra-process), IPC (inter-process), TCP and Websocket.

- **Usability:** Provides intuitive APIs that can be easily understood by developers.

Other features and architectural aspects discovered by exploring the documentation are:

- **Bind and connect:** Like ZeroMQ and nanomsg, the NNG sockets can connect/bind to multiple endpoints. However, performing one does not prohibit the other from happening. NNG sockets may freely connect and bind to endpoints.

- **Context creation:** The concept of context for NNG differs from the concept presented before. The context, here, is not related to the global state variables of the library. Instead, the contexts are linked to sockets. NNG enables the establishment of multiple contexts on a single socket. These contexts can be viewed as multiple separate instances of the socket's protocol designed for concurrency scenarios where the same socket is utilised. A more detailed explanation will be given ahead.

- **Asynchronous I/O framework:** NNG allows the creation of certain asynchronous operations - sleep, send and receive - and the association of an on-complete callback. NNG also supports the cancellation of ongoing asynchronous operations. The cancellation may be issued manually[28] or automatically when the defined timeout, for completion, expires.

- **Statistics:** Allows the verification of statistics about the sockets, listeners, dialers, and more. This feature enables the application's behaviour to be observed, and consequently, helps in performing administration tasks and troubleshooting.

- **Byte streams:** NNG is a message-based communication library by default, however, it provides a mechanism that enables reading from / writing to sockets through byte streams.

- **HTTP support:** NNG offers HTTP capabilities, both for servers and clients, enabling the creation of real web services, such as REST APIs.

2.4.2 Sockets

NNG sockets are composed of both listeners and dialers, with listeners being responsible for binding to an endpoint and accepting connections, while dialers are responsible for establishing connections by connecting to specific endpoints. In short, NNG allows an application to act as a server (accept connections) and a client (request connections) using the same socket. An NNG socket may be composed of multiple dialers and listeners simultaneously, with each using a transport of choice.

[28] To cancel an asynchronous operation manually means to invoke a "cancel" function using the object associated with the operation.

Dialers

A socket can connect to a specific endpoint by creating a *dialer*. Dialers use the URL specified at creation time to initiate a connection with a remote listener. When the connection is established successfully, a *pipe* is created and then associated with the socket.

When the pipe is closed, the dialer is responsible for attempting, periodically, to reestablish the connection (auto-reconnect).

A dialer must have only one pipe at a time.

Listeners

A socket may also listen for connection requests on specific endpoints. To do this, the socket creates a *listener*, which binds to a specified endpoint and accepts connections initiated by remote dialers.

Listeners, unlike dialers, may possess multiple open pipes at the same time.

Pipes

Pipes can be perceived as a single connection. They are always associated with a single listener or dialer and a single socket.

Accessing directly an individual pipe is possible for greater control over the delivery of the messages. However, most applications should only require the standard delivery provided by the semantics of the socket.

Pipes inherit the configuration options defined by configuring the parent of the pipe - the dialer or the listener - before it is created.

Pipes allow the association of callbacks. A callback[29] may be set for each of the following events:

- NNG_PIPE_EV_ADD_PRE: After the connection establishment and negotiation, but before adding the pipe to the socket.
- NNG_PIPE_EV_ADD_POST: After the pipe is added to the socket. After the invocation of this event, it is possible to communicate through the pipe using the socket.
- NNG_PIPE_EV_REM_POST: Event that takes place after the pipe is removed from the socket. The transport may be closed at this moment, and it is no longer possible to communicate using the pipe.

[29] The callbacks must not access the socket as locks are used to prevent unwanted accesses and therefore a deadlock will happen as a result.

Contexts

Some NNG protocols allow the establishment of contexts. These contexts are used to separate the processing done by the protocol from the socket to a context, allowing multiple contexts to be created using a single socket. By having multiple contexts associated with a single socket, independent flows of messages may coexist within the same socket without interfering with each other. Each context has an identifier to distinguish them.

Not all sockets allow the creation of contexts, as the semantics of the protocol (messaging pattern) do not justify such functionality. Currently, only the sockets related to the Request-Reply and Survey patterns support such functionality. Taking the Request-Reply pattern as an example, let us consider two nodes, A and B, where A desires two distinct flows of requests and replies with node B. Instead of creating 2 REQ sockets on node A and 2 REP sockets on node B to allow the existence of 2 separate flows, it is possible to achieve such functionality by using a single pair of REQ and REP sockets. This can be achieved through the establishment of 2 different contexts on both sockets. Through these contexts, it is possible to receive requests and send replies concurrently without the requests and replies of the first context interfering with the ones of the second context and vice-versa.

Essentially, the contexts enable the functionality that would otherwise require multiple sockets, to be achieved through the establishment of multiple contexts on the same socket. Additionally, it simplifies acquiring a behaviour, such as an asynchronous variant of the Request-Reply protocol, which would otherwise necessitate the utilization of the sockets' raw mode. This, in turn, would require analysing and altering message headers and re-implementing default protocol behaviour present in the "cooked" mode of the sockets.

Devices and Relays

NNG is also equipped with a functionality that aids developers in the creation of *forwarders* and *relayers* which route and forward messages from one socket to another. This can be done by using a *nng_device*.

When creating a *device*, only sockets in RAW mode may be used. This is required as the semantics of the protocol should be bypassed to allow messages to be routed without restrictions.

To correctly route a response, back to the requester, the devices use a *backtrace* field present in the headers of the messages.

Some protocols may also set a "time-to-live" property to protect against loops. The devices are responsible for updating this value, which is present on the message headers, and for discarding the messages when their time expires.

The delivery service provided by the devices follows a best-effort model. The devices discard messages when needed, for example, when a queue is full due to backpressure.

2.4.3 Message Handling

NNG is by default message-based, i.e., the communication is defined by the exchange of messages. The NNG messages are composed of a header and a body. The header contains protocol-specific information, and the body contains the payload (the actual content).

When a message is received, an object is created that associates the "message" with other information, such as the pipe used to receive the message[30]. Knowing which pipe originated the message is useful for protocols that need to route replies to the appropriate destination. To send a message through a specific pipe (if applicable), the API provides a method that allows the association of such information with the message.

Supports zero-copy, which is a feature that allows messages to be sent without requiring the message to be copied, i.e., duplicated. This is especially useful when large messages need to be sent or when memory is limited.

NNG supports blocking and non-blocking send and receive operations. When a non-blocking operation is desired, the NNG_FLAG_NONBLOCK must be set as a configuration option of the operation. When the flag is not set, the socket will block until a message is sent/received or until the specified timeout is reached. If the operation is non-blocking, then the function will return immediately regardless of whether a message was sent/received or not.

NNG allows accessing and modifying the headers of the messages. This is permitted to provide a more versatile behaviour, like the creation of proxies to forward messages. To take advantage of this functionality, the RAW modes of the sockets must be used. A socket created in RAW mode enables direct access to the transport protocol, bypassing processing related to the messaging pattern. This capability facilitates the development of custom protocols.

2.4.4 Communication Patterns

NNG, in addition to implementing the same communication patterns as its predecessor - Pair, Request-Reply, Publish-Subscribe, One-way Pipeline (Push-Pull), Survey, and Bus - it allows developers to create and use new protocols.

[30] Not all protocols (messaging patterns) support the selection of the pipe to deliver a message, therefore, this information may not be present.

2.4.5 Fault Tolerance and Reliability

NNG incorporates fault tolerance through its auto-reconnect feature, which restores communication after disconnections. It also attempts to handle all possible crash scenarios, aiming for crash resilience.

In terms of message delivery reliability, most of its default sockets operate on a best-effort basis, allowing messages to be dropped without reaching their destination. Some socket types, like REQ, offer improved reliability by retransmitting messages. However, if a new request is issued before the previous one is delivered, the earlier message is discarded. Ultimately, delivery guarantees depend on the underlying transport protocol, which may not ensure reliable transmission.

2.4.6 Limitations and Challenges

NNG shares similar limitations with nanomsg, including its dependence on the transport protocol for reliability and lack of built-in support for mobility. Although NNG is fundamentally connection-oriented, there seems to be ongoing efforts to support connection-less protocols like UDP (currently experimental). This may eventually enable the use of connection-less transport protocols that ensure exactly-once delivery outside of connections (limitation of TCP).

2.5 MQTT

MQTT (Message Queuing Telemetry Transport) is a protocol that enables the efficient and reliable exchange of messages between devices scattered across the Internet, making it the most widely used messaging protocol for the Internet of Things (IoT). This protocol, based on the publish-subscribe communication pattern, defines how sources (publishers) communicate with receivers (subscribers) through topics. MQTT is a protocol that utilizes brokers to establish the connection between publishers and subscribers, and to filter and distribute messages. The use of brokers allows the decoupling of publishers and subscribers.

2.5.1 Key Features

In this section will talk about the key features of MQTT, which will be discussed in more detail in the following sections.

- **Simple implementation:** The protocol was designed to be extremely easy to implement on the client side.
- **Lightweight and efficient:** Requires minimal resources, making it suitable for constrained devices and limited networks.
- **Bi-directional:** Clients can be both publishers and subscribers simultaneously, allowing for bi-directional communication.
- **Scalable:** The MQTT architecture allows the use of multiple brokers, clustering[31], and load balancing to scale to millions of connected devices.
- **Robust Connectivity:** Performs well over unreliable networks (with low bandwidth and signal).
- **Secure:** Can provide secure connections and authentication.
- **Data agnostic:** MQTT messages follow a binary format, allowing any kind of data to be transmitted as the payload.
- **Broker-based:** Uses message brokers as intermediaries between the clients (publishers and subscribers). The brokers are responsible for receiving published messages and forwarding them to the appropriate subscribers.

[31] Some MQTT Brokers implementations support clustering, which enables multiple instances of the broker to work together in handling large numbers of clients and messages.

- **Configurable delivery guarantee:** Supports 3 QoS (Quality of Service) levels related to the delivery guarantee — at-least-once, at-most-once, and exactly-once.
- **Persistent Sessions:** Allows the establishment of a persistent session between the client and the broker. This enables sessions to be restored after recovering from a connection failure.
- **Retained messages:** The MQTT brokers can store the last message of a topic. This allows new subscribers to receive a message immediately after subscribing to the topic, without needing to wait for a new message to be published.
- **Last Will And Testament:** MQTT clients can inform MQTT brokers of a message, called last will, that is sent by the broker when the client disconnects unexpectedly. This allows informing the subscribers that the publisher disconnected.

2.5.2 Architecture

MQTT follows Publish-Subscribe architecture. In a Publish-Subscribe architecture, there are publishers which generate and publish messages, and the subscribers which receive the messages. Publish-Subscribe is a wider concept that can be implemented in various manners, with different protocols and technologies. In the case of MQTT, the publish-subscribe pattern is implemented using brokers and is based on topics. Also, MQTT relies on the TCP/IP protocol for its transport, which provides reliability, ordered data transfer, and error checking, essential for constrained deployment environments.

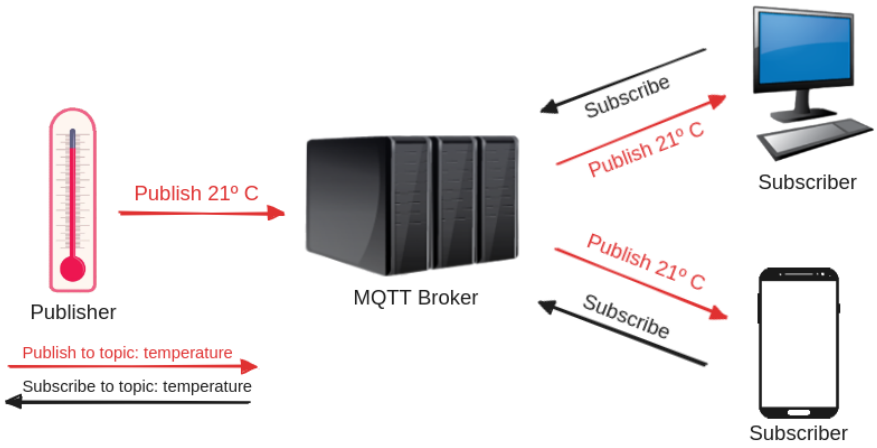


Figure 1: MQTT Publish/Subscribe Architecture

Also, MQTT was created for the purpose of sending small quantities of data over networks with limited bandwidth and connectivity. Due to this requirement, MQTT packets were modelled to carry minimal

overhead, which results in extraordinary performance when it comes to transferring data over the wire. In addition, the minimal and simple packet format makes MQTT the ideal protocol to work with devices with limited resources (with low processing power, memory, battery, etc.). A brief explanation of the message format will be given ahead.

MQTT Components

The MQTT architecture consists of 2 main components: MQTT Client and MQTT Broker.

Publishers and subscribers can both be referred to as MQTT Clients. An MQTT Client can be a publisher, a subscriber, or both simultaneously. A publisher is a client that generates data, that may be of interest to subscribers, and sends it to an MQTT Broker. A subscriber informs an MQTT Broker of its interests by making subscriptions, making it eligible to receive the data generated by publishers that match the presented preferences. The MQTT protocol is based on TCP/IP, therefore, any device that speaks this protocol and has software that implements the MQTT client functionality can be an MQTT Client.

The MQTT Broker, also known as MQTT Server, plays a critical role in the MQTT environments, by being a central piece responsible for managing the communication between clients and ensuring the reliable delivery of messages. The functionalities provided by MQTT Brokers include:

- Handling of large quantities of concurrent connections: By being a central piece in the topology, the MQTT Brokers need to be able to handle large amounts of concurrent connections.
- Filtering and routing messages: The broker filters incoming messages and distributes them according to the clients' subscriptions.
- Session management: The broker keeps the session data of the clients, allowing clients that disconnected to recover their subscriptions and missed messages.
- Authentication and Authorization: MQTT brokers are responsible for authenticating and authorizing clients. The security mechanisms are extensible, which allows authentication and authorization by using credentials, but also using other methods, such as certificates.

Decoupling of Clients

By using brokers as intermediaries, the subscribers can be decoupled from the publishers. The decoupling happens in three dimensions: space, time and synchronization. Decoupled in space, because a publisher and a subscriber do not need to know each other, only the IP address and port of the broker are needed.

Decoupled in time, because a publisher and a subscriber do not need to be online at the same time. The broker can store the messages for clients while they are offline. The only conditions that need to be met for the messages to be stored, while the client is offline, are: the client must have a persistent session (will be explained later) and must be subscribed to a topic with a Quality of Service superior to 0. Decouple in synchronization, since the publishers and the subscribers do not need to be interrupted when publishing or receiving messages, i.e., since the clients do not connect directly to each other, there is no need to synchronize the sending/receiving operations.

Connection Establishment and Maintenance

MQTT connections always happen between clients and brokers. Clients cannot interact directly with other clients. A connection is initiated with a CONNECT message, by the client, to which the broker responds with a CONNACK message. After a successful connection establishment, the connection is maintained open until an explicit disconnect command is sent by the client, or the connection breaks unexpectedly.

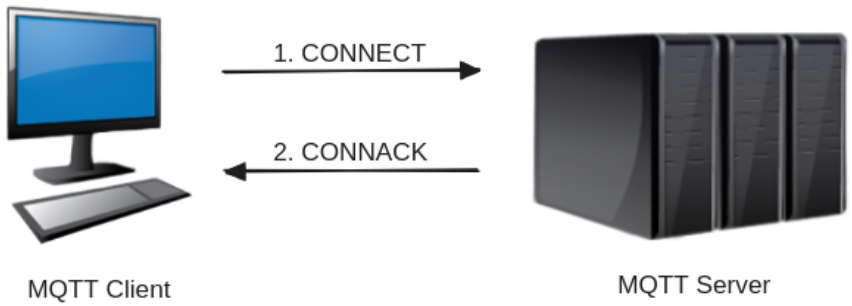


Figure 2: Connection Establishment

If the CONNECT command message is malformed, or if after opening the socket, the client takes too much time to send a CONNECT command, the broker should terminate the connection to avoid being slowed by potential malicious clients.

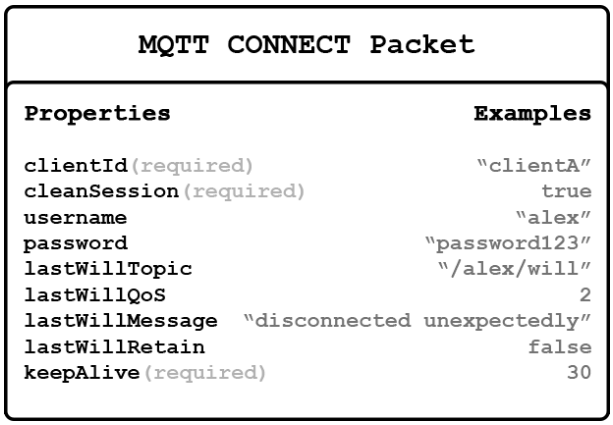


Figure 3: CONNECT packet structure

Let's dive into the essential details of a CONNECT command:

- **Client Id**: Unique identifier used to distinguish each client that connects to the broker. This allows the broker to keep track of the client's session state.
- **Clean Session:** Flag that allows the client to specify if he wants to establish a persistent session. If the value is 'true', then the broker will store the subscriptions and missed messages (subscribed with a QoS level above 0) of the client. If the value is 'false', the broker will not store any of the information for the client and will discard any previous persistent session information.
- **Username/Password:** These attributes are optional and are used by the broker to authenticate and authorize the client.
- **Last Will Message:** The attributes with the prefix "lastWill" refer to a MQTT feature called "Last Will and Testament". When a client disconnects abruptly, the broker will inform the interested subscribers of this event.
- **Keep Alive:** This attribute defines the amount of time the broker and the client can remain silent before the connection is defined as not active. To maintain the connection active, the client regularly sends PING Request commands to the broker, to which it responds with PING Response commands. This attribute is used by the broker to detect when the will message should be sent.

The MQTT protocol defines that a CONNECT message must be answered with a CONNACK message.

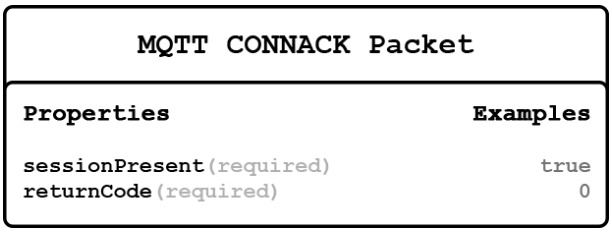


Figure 4: CONNACK packet structure

Having said that, let's take a look at the fields carried by a CONNACK message:

- **Session Present:** Flag used by the broker to inform the client if a persistent session is present. When the value of the 'sessionClean' flag, present in the client's CONNECT message, is 'false', the broker will always send the 'sessionPresent' flag as 'false', even if there was a persistent session present. This flag helps the clients understand if they need to re-subscribe to topics, or if the broker already possesses this information.
- **Return Code:** The return code is used to inform the client if the connection establishment was successful or to specify the error that led to the failure.

Messaging Model

The messaging model of MQTT is based on topics and subscriptions. Publishers publish messages to a topic, and subscribers make subscriptions (subscribe to topics).

Subscriptions are used to inform the broker of which topics a client is interested in receiving messages from. A subscription also contains the maximum Quality of Service intended for the messages of that topic. If a client publishes the messages for that topic using a Quality of Service higher than the one specified by the subscriber, the broker will downgrade to the subscriber's Quality of Service.

When a client subscribes to a topic, the broker will keep track of the subscription and forward any message to the subscriber that is published to that particular topic.

A client can be subscribed to multiple topics at once, and a topic can have multiple subscribers simultaneously.

Message Filtering

Message filtering is an important aspect of the architecture as it ensures that the clients exclusively receive messages that are aligned with their specific interests.

Topic-based filtering, also known as Subject-based filtering, is the only type of filtering explicitly mentioned in the MQTT specification. However, some implementations like HiveMQ provide other filtering options like content-based filtering and type-based filtering. For more information about these types of filtering refer to part 2 of HiveMQ Essentials (HiveMQ, 2015a).

A topic, in MQTT, is a UTF-8 string that resembles a file path. The file path format allows topics to be structured hierarchically, where forward slashes (/) are used to delimit the levels. An example of a topic is "myhome/livingroom/lamps/lamp1".

When using the topic-based filtering option, the broker filters the messages based on the topic or subject. The subscribers specify, to the broker, which topics they are interested in via subscriptions, and the broker routes the messages according to the topic hierarchy described in those subscriptions.

MQTT provides more flexibility by supporting the use of wildcards when subscribing to topics[32]. Wildcards allow subscribing to multiple topics that follow a certain pattern using a single subscription. The wildcards offered by MQTT are: '+' (plus symbol) which matches a single topic level, and can be placed anywhere; and '#' (hash symbol) used to match multiple levels, but can only be placed at the end of the topic. For a better understanding, let's explore some examples. ✓ and × will be used to identify if a topic matches or does not match, respectively, the topic being discussed. Consider the following topics:

1. **"Europe/Portugal/news":** This topic represents the news related to the Portugal country.
    - "Europe/Portugal/news" ✓
    - Any other topic ×
2. **"Europe/+/news":** This topic represents the news related to any country in Europe.
    - "Europe/Portugal/news" ✓
    - "Europe/France/news" ✓
    - "Europe/Portugal/weather" ×
    - "Asia/Japan/news" ×
3. **"Europe/#":**
    - "Europe/Portugal/news" ✓
    - "Europe/Germany/weather" ✓
    - "Europe/France" ✓

[32] Wildcards cannot be used when publishing messages.

- "Europe" ×
- "Asia/Japan/news" ×

4. **"#/news":** This is not a valid topic. The '#' wildcard can only be used at the end of the topic.

There is one more special symbol used in MQTT topics. The symbol '$' used at the beginning of topics, is reserved for internal information and statistics of the MQTT broker, which means that clients are not allowed to publish messages using topics that begin with '$'. Also, topics that start with '$' will not be matched by a subscription using the wildcard '#'.

Publishing Messages

Publishing messages is an important aspect of the MQTT architecture as it involves defining formats for the packets and the communication flow between the devices. With that being said, we will be discussing how messages are published in MQTT as shown in parts 4 and 6 of HiveMQ (2015a).

Immediately after establishing the connection with the broker, a client may start publishing messages. To do this, the PUBLISH command message is used.

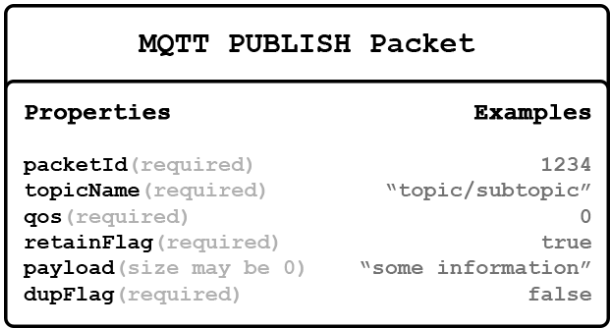


Figure 5: PUBLISH packet structure

The PUBLISH packet is composed of several attributes:

- **Packet identifier (PacketId):** The packet identifier is used to identify the message and to ensure that the delivery order matches the publishing order. When the Quality of Service (QoS) defined is 0, the message must not contain a packet identifier. The message identifier is then used in messages like PUBACK, PUBREC, PUBREL and PUBCOMP (these commands will be briefly explained later).
- **Topic name:** Clients subscribe to topics, therefore, the topic name is a fundamental field that will dictate who will receive the message.

- **Quality of Service (QoS):** This field determines the level of reliability and delivery guarantee of the message. Refer to QoS section for more information.
- **Retain Flag:** This flag determines if the message should be stored by the broker as the most recent value of the topic. This allows the last retained message to be sent immediately to all clients that establish a new subscription that matches the topic of the message. Thus, enabling new subscribers to instantaneously receive a message from the topic without needing to wait for a client to publish to the topic.
- **Payload:** The payload is the actual content of the message and is transmitted in byte format, which enables clients to send any type of data.
- **DUP flag:** This flag is used to indicate that the message is a duplicate. The flag is set to 'true' when the receiver has not confirmed receiving the original message. The message should be ignored when a message with the same ID has already been received.

After understanding the requirements to form a PUBLISH packet, let us explore how it is used. For a published message to reach a subscriber, a publishing process[33] is realized between a publisher and a broker, and then another one[34] happens with the broker and the subscriber as participants. The publisher-broker publishing process is very similar to the broker-subscriber publishing process. However, it is important to notice that while the publisher-broker process follows a branch related to a certain Quality of Service (QoS) level, the broker-subscriber process may follow a branch associated with another QoS level, as it is discussed further ahead, in 2.5.2.

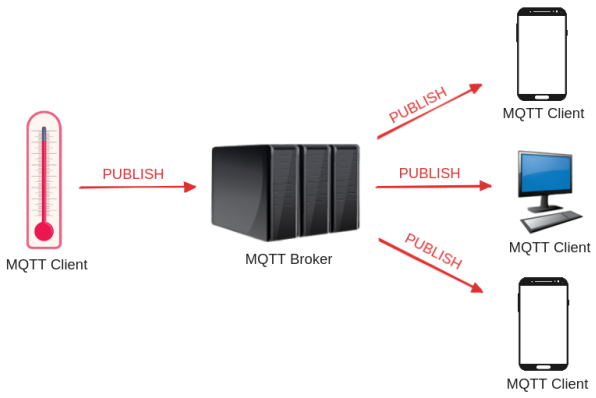


Figure 6: MQTT Publish Flow Example

[33] We can imagine the publishing process as the action of traversing a tree of steps, from the root to a leaf, where some conditionals define which branch should be taken. Examples of conditionals are: if the flow is from publisher to broker, or broker to subscriber; the QoS level; etc.

[34] Except when a client publishes a message using a QoS level of 0 (at-most-once delivery guarantee), which does not ensure the arrival of the message at the broker. If the broker does not receive the message, then the publishing process is realized only one time.

The publishing process from a client (publisher) to a broker, is the following:

1. **Publish:** The client sends a PUBLISH message to the broker.

2. **Publish received:** Upon receiving a PUBLISH message, the broker verifies its syntax and format. Afterwards, and depending on the QoS level, the acknowledgement of the receipt may follow:

   - For QoS 0, the broker processes the message without sending an acknowledgement.
   - For QoS 1, the broker initiates the process of delivering the message to the subscribers and sends an acknowledgement message (PUBACK) to the publisher.
   - For QoS 2, there are two possible alternatives[35]:

     (a) The broker initiates the process of delivering the message to the subscribers, and afterwards, acknowledges the receipt of the PUBLISH message by sending a PUBREC message.

     (b) The broker sends a PUBREC message as acknowledgement. The process of distributing the published message is initiated after receiving a PUBREL message from the publisher.

3. **Publish Release (QoS 2):** When the QoS equals to 2, the communication flow between the publisher and the broker, regarding this message, corresponds to a 4-step handshake. After receiving a PUBREC message, the client deletes the PUBLISH message, keeping only its identifier, and responds with a PUBREL message. The broker, after receiving the PUBREL message and completing the delivery of the message to the subscribers can discard (release) the message.

4. **Publish Complete (QoS 2):** Finally, the broker sends a message to confirm that the message was received and processed[36].

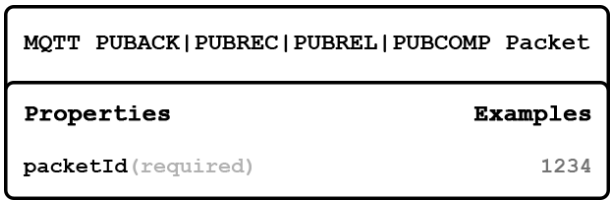


Figure 7: Structure of PUBACK, PUBREC, PUBREL and PUBCOMP packets

[35] As described in section 4.3 of the MQTT 3.1.1 specification (OASIS Open MQTT Technical Committee, 2014), the broker must implement one and only one of the two alternatives.

[36] The broker is not required to have completely processed the PUBLISH message, as it is referred in the following citation: "The receiver is not required to complete delivery of the Application Message before sending the PUBREC or PUBCOMP."(OASIS Open MQTT Technical Committee, 2014).

A publisher should keep track of the Packet IDs it has sent and received in order to guarantee that messages are not lost or duplicated.

For a better understanding, the flow of messages, for each QoS level, can be seen below:

Figure 8: Publish process with QoS 0 (at-most-once)

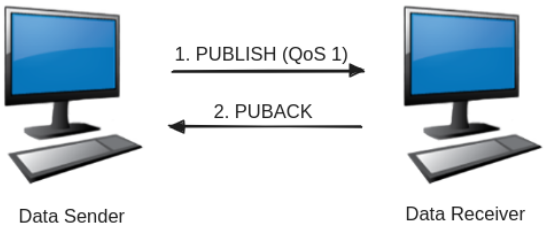


Figure 9: Publish process with QoS 1 (at-least-once)

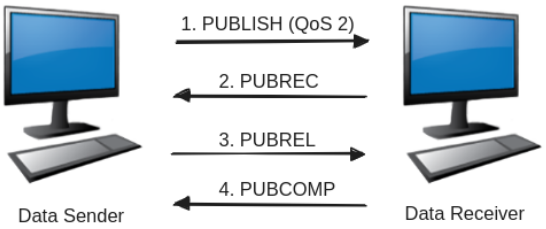


Figure 10: Publish process with QoS 2 (exactly-once)

Subscribing to topics

Clients cannot receive messages if they do not subscribe to any topics, therefore, in this section, we will explore the characteristics and flow of messages that enable the subscription of topics.

To make a subscription, a client uses a SUBSCRIBE message. The SUBSCRIBE message has a simple structure. It is composed of a packet identifier and a list of subscriptions. A subscription is a pair that

contains a QoS level and a topic string (which may contain wildcards). The QoS level represents the desired level of delivery reliability that the subscriber wants for the topic(s).

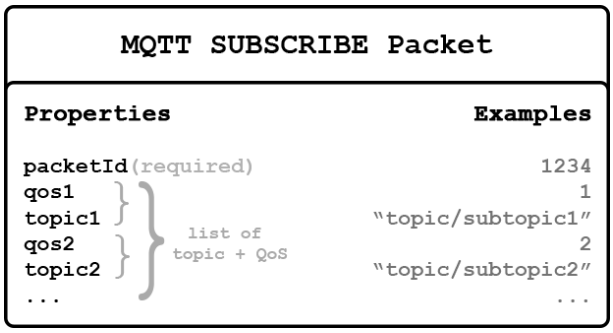


Figure 11: SUBSCRIBE packet structure

It's important to note that when the broker receives a (published) message that matches more than one subscription of the same subscriber, the message will be sent using the highest QoS level of the subscriptions.

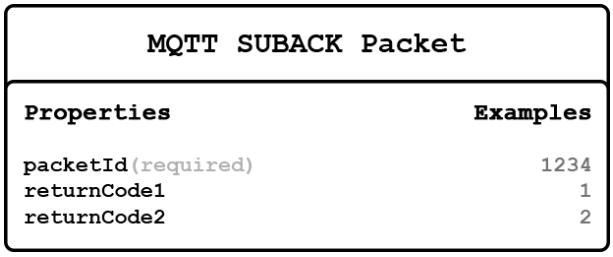


Figure 12: SUBACK packet structure

After sending a SUBSCRIBE message, the client expects the broker to respond with a SUBACK message confirming the receipt. SUBACK messages contain a packet identifier, required by the client to understand which message the broker is referring to, and a return code for each subscription present in the SUBSCRIBE message. The return codes are crucial as they allow the client to tell which subscriptions were approved and which were declined. If the subscription is approved, the return code also communicates the maximum degree of QoS that will be provided by the broker. A subscription may be rejected when it is malformed, when wildcards are being used but not allowed by the broker, etc.

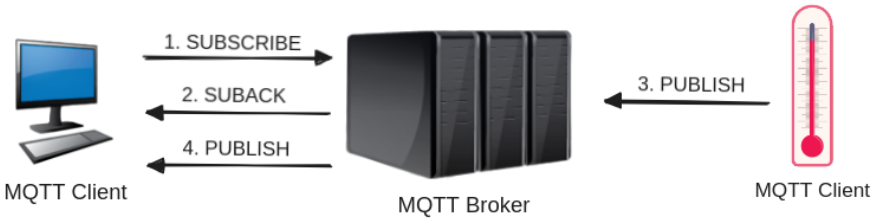


Figure 13: MQTT Subscribe Flow Example

Unsubscribing topics

With the ability to subscribe to topics of interest, a client must also be able to undo this operation, i.e., the client must be able to unsubscribe topics. Having said that, we will delve into the characteristics of the UNSUBSCRIBE message, and the corresponding acknowledgement message (UNSUBACK).

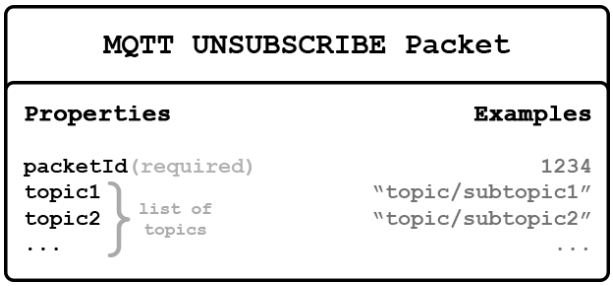


Figure 14: UNSUBSCRIBE packet structure

The UNSUBSCRIBE message contains a packet identifier and a list of the topics from which the client wishes to stop receiving messages from. The format of this packet is similar to the format of the SUBSCRIBE packet. The difference lies in the lack of QoS levels following the topics.

When a client wants to unsubscribe from a topic or a list of topics, it sends an UNSUBSCRIBE message to the broker. After receiving the UNSUBSCRIBE message, the broker removes the client's subscriptions present in the message and confirms the unsubscription of the topics by sending a UNSUBACK message.

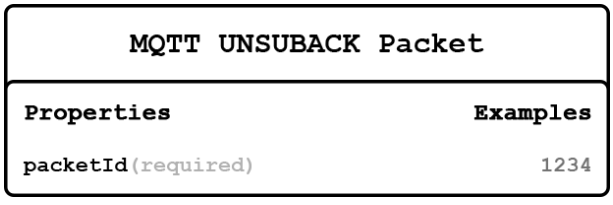


Figure 15: UNSUBACK packet structure

An UNSUBACK message is formed solely by the packet identifier that identifies the original UNSUB-

SCRIBE message, required by the client to discern which UNSUBSCRIBE message was received and processed by the broker, and by consequence, allowing the client to figure out which subscriptions were revoked.

By receiving a UNSUBACK a client can assume that the well-formed subscriptions present in the UNSUBSCRIBE message were deleted. Even though the unsubscription was acknowledged by the broker, any message with a QoS higher than 0, that was already set to be sent to the client, before the unsubscription occurred, will be delivered.

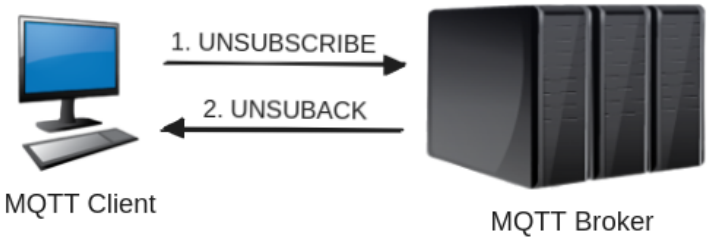


Figure 16: MQTT Unsubscribe Flow Example

Message Format

Having limited networks and constrained devices in mind, MQTT messages follow a binary format, which allows reducing the size of messages and promotes efficiency in communication and message processing.

The basic structure of an MQTT packet consists of:

- Fixed header, present in all MQTT Control Packets
- Variable header, present in some MQTT Control Packets
- Payload, present in some MQTT Control Packets

The figure 17 represents the format of the fixed header. The fixed header carries information related to the type of control packet, flags specific to the type of control packet, and the remaining length. The remaining length starts at the second byte and is used to inform the number of bytes remaining within the current packet, including data in the variable header and the payload.

Bit	7	6	5	4	3	2	1	0
byte 1	MQTT Control Packet type				Flags specific to each MQTT Control Packet type			
byte 2...	Remaining Length							

Figure 17: MQTT Fixed Header Format (source)

For a complete understanding of the MQTT packets refer to the sections 2 and 3 of the MQTT specification (OASIS Open MQTT Technical Committee, 2014).

Quality of Service (QoS)

MQTT supports 3 levels of quality of service (QoS):

- **QoS 0:** This level provides the "at most once" delivery guarantee. Messages are sent without needing confirmation and may be lost. The sender does not store nor wait for an acknowledgement from the receiver. Adequate for situations where the loss of messages is acceptable.
- **QoS 1:** Provides the "at least once" delivery guarantee level. At this level, it is necessary to acknowledge the reception of messages. The message is retransmitted (after a small delay) until the reception is confirmed. This level of reliability is ideal when the loss of messages is not acceptable, but the duplication of messages is tolerable.
- **QoS 2:** Provides "exactly once" delivery guarantee level. At this level, messages are acknowledged and resent until the message is received exactly once. The broker and the publisher participate in a 2-step confirmation, in which the message must be retained by the broker until its receipt is confirmed by the subscriber. Required when neither loss nor duplication of messages is acceptable.

As the QoS increases, more packets flow through the network and more resources are demanded from the devices (both computational and storage-wise), resulting in increased latency and potentially a lower throughput. This is meant to emphasize the importance of thoroughly analysing the use case before selecting a Quality of Service, in order to obtain the intended balance between reliability and performance.

We have seen that both a publisher and a subscriber define QoS levels, one when publishing and the other when subscribing. But, how does MQTT handle both? After discovering that both clients define a QoS level, I wondered if the QoS level used was defined for each connection, i.e., a message would be sent by the publisher to the broker using the QoS level present in the message, and the broker would then send the message to the subscriber using the QoS level established with the subscription. However, after consulting multiple sources[37], I've come to discover which of the QoS levels prevails. The QoS level defined by the publisher (on the PUBLISH packet) is used when communicating with the broker. After receiving the message, the broker will then route the message to the subscribers. The QoS level used by the broker to

[37] Light (2024) and HiveMQ (2015b)

deliver the message is the lowest value between the one defined by the publisher (on the publish message) and the subscriber (on the topic subscription). Consider the following table for clarification:

Publisher QoS	Subscriber QoS	Publisher → Broker QoS	Broker → Subscriber QoS
0	1 or 2	0	0
1 or 2	0	1 or 2	0
1	2	1	1
2	1	2	1

Table 5: QoS levels across transmission (Light, 2024)

Persistent Sessions

MQTT provides persistence sessions to ensure, when required, that subscriptions and messages are not lost in the event of a network or node failure (HiveMQ, 2015c). When a client establishes a connection, it has the option to request a persistent session. The client can do so by setting the CONNECT packet's cleanSession flag to 'false', as we have seen in the section related to Connection Establishment. The session state is then preserved by the broker until the client explicitly connects with cleanSession=true[38]. However, some exceptions, such as administrative message expiration policies or memory limit constraints, may lead to the loss of the session state.

It should be noted that when a persistent session is requested, the broker is not the only one that should retain session data. A client that requests a persistent session plays an important role in maintaining the session and assuring an appropriate communication and processing flow. Message flows related to QoS 1 and 2, require both sides to keep information about the exchange because retransmissions occur while the expected message does not arrive. If one of the devices does not have the necessary session information to continue the exchange, the flow does not end. Consequently, the session data that a client must keep when participating in a persistent session is:

- Flow of all published messages with QoS 1 or 2 for which an acknowledgement/completion message has not been received from the broker;
- And, all messages received with QoS 2 for which a completion message has not been sent to the broker.

[38] Recall that a client sets the cleanSession to 'true' when a non-persistent session is desired. This results in any session state kept by the broker to be discarded.

Client Take-Over

MQTT also provides a very important mechanism named "Client Take-Over". This mechanism is a key feature that "guarantees smooth connectivity and resilient communication in the face of intermittent network interruptions." (HiveMQ, 2015d). After a client suddenly disconnects, it may attempt to reconnect. When reconnecting, the client provides a client identifier[39] that allows the broker to check for an existing connection. Any previous connection associated with the client identifier is closed, allowing the client to reconnect and not be blocked by an existent half-open connection[40]. Additionally, this mechanism can help a client reconnect and recover its persistent session in a mobility scenario, i.e., when the device requires changing to a different network to maintain internet connectivity.

2.5.3 Fault Tolerance and Reliability

Overall, MQTT provides a lot of features regarding fault tolerance and reliability. However, some of them may only be available in certain implementations, as they may not be required by the specification.

MQTT is implemented over the TCP transport protocol, due to its capability in providing an end-to-end ordered and reliable stream of bytes. By using TCP as the underlying transport mechanism, MQTT can ensure that messages reach the destination even when the devices are deployed in constrained networks. However, the reliability of message delivery cannot be assured exclusively by using the TCP protocol. This is because, in the event of a node crash or a sufficiently prolonged network failure, TCP won't be able to recover the connection, resulting in the loss of messages present in the TCP buffer. Having this in mind, MQTT provides persistent sessions that store session data for proper recovery in the presence of node/network faults. It is important to notice that MQTT allows developers to choose the desired level of reliability. This is done by choosing a Quality of Service (QoS) level which directly influences whether a message is persisted or not[41].

For further reliability and fault tolerance, some MQTT implementations may persist data in non-volatile memory, allowing session state recovery after a crash or restart.

In addition, MQTT's publish-subscribe model is broker-based, which enables replication, load balancing, and clustering of brokers to maintain a functional system in the event of node failures.

[39] Sent in the CONNECT packet.

[40] A TCP half-open connection occurs when one of the peers participating in the TCP connection does not answer correctly to the 3-way handshake, or when it disconnects abruptly leaving the other peer unaware of the disconnection.

[41] For a better understanding, refer to the sections about QoS levels and Persistent sessions.

2.5.4 Limitations and Challenges

In this section, we will talk about the limitations and challenges of the MQTT middleware:

Message ordering The following comment is based on the information provided in section 4.6 of MQTT 3.1.1 specification (OASIS Open MQTT Technical Committee, 2014). One of MQTT's limitations is associated with message ordering. MQTT specifies that a topic should be treated as an ordered topic, i.e., the delivery of the messages from a specific topic, to a client, should happen in the order they were published by the broker. Also, for a given topic and QoS the order at which the messages were received by the broker from a specific publisher, should be maintained when forwarding the messages to the subscribers. These rules, defined by MQTT 3.1.1 specification, suggest that the overall order of messages may not be provided.

Use case specific The MQTT middleware only offers the publish-subscribe pattern. It is possible to implement the Request-Reply pattern over MQTT's pub-sub, however, when considering performance requirements (like low latency), MQTT might not be the ideal choice.

Chapter 3

Exon Protocol and Exactly-Once Delivery

One of the essential parts of designing a messaging middleware that aims to guarantee that messages are delivered once and only once (exactly-once delivery guarantee) is the choice/design of a protocol capable of providing such a guarantee. In the area of messaging systems, ensuring exactly-once delivery is crucial to maintain the integrity and reliability of communication. That being said, in this chapter, I will be presenting the Exon transport protocol (Kassam et al., 2022) and its capability to provide exactly-once delivery guarantee. Furthermore, I will provide an overview of the architecture and features of the protocol's current implementation, *Exon-lib* (Kassam et al., 2021), and discuss some of the limitations that may need to be addressed before the integration in the messaging middleware.

3.1 Exon Transport Protocol

Exon is an innovative message-based transport protocol that achieves simultaneously "exactly-once delivery, obliviousness, efficiency and no dependence on timing assumptions for correctness"[1].

The communication reliability provided by the exactly-once guarantee is achieved by satisfying, simultaneously, both at-most-once and at-least-once delivery guarantees. At-most-once is satisfied by assuring that a user message (payload) can only be delivered after requesting a unique slot, consumed upon delivery, at the receiver. At-least-once is assured because the sender does not give up (re)transmitting a user message until a confirmation, that the message was consumed by the receiver, arrives.

The exactly-once message delivery is achieved through a strategy that involves a 4-way exchange per message. A 4-way strategy per message seems inefficient and is if no special mechanism is employed. Exon reaches efficiency while maintaining correctness, through the use of soft half-connections, which are transparent to the API. By using soft half-connections, the first 2 steps of the 4-way exchange can be shared by several messages, and thus allow the protocol to achieve great performance. In addition, the

[1] Citation from Kassam et al. (2022).

protocol is order-less, i.e., it does not ensure ordered delivery of messages. Consequently, it can run at full speed without suffering from problems like head-of-line (HOL) blocking. If required, a higher layer may implement message ordering over Exon.

Another characteristic of Exon is being oblivious. Exon's obliviousness is accomplished by safely cleaning up all per connection state that is no longer needed (when the communication stops). The protocol only requires a sequence number as a persistent state, making it suitable for environments defined by devices with limited resources, like in IoT.

Finally, the Exon protocol does not depend on timing assumptions to achieve correctness, since it does not follow the concept of connections. By identifying nodes through logical identifiers instead of relying on a combination of IP address and port, the protocol allows for long-lived communication, that can even survive IP changes, such as in vehicular networks.

In the following sections, we will discuss these concepts in greater detail.

3.1.1 System model

Before presenting the algorithm of the protocol, it is important to understand the assumptions made about the environment when developing the protocol. This is especially important because different assumptions from the ones established for the development of the messaging middleware mean that some architectural decisions need to address the shortcomings not contemplated by the transport protocol.

Exon's system model presumes:

- A network environment with an arbitrary number of nodes;
- Nodes require a local memory;
- Nodes can communicate for an arbitrary amount of time;
- Nodes can crash, however, the state prior to the crash must be recovered eventually. As a consequence, it is assumed that the node possesses stable persistent storage.
- The network is asynchronous, i.e., no timing assumptions are made, neither relating to the time it takes for a message to reach the recipient nor relating to the processing time of a message.
- Messages may be lost, duplicated and reordered, but not corrupted.
- There may exist network partitions, however they will eventually cease to exist.
- The existence of a transport layer communication channel to send messages in any form.

3.1.2 Algorithm Overview

We will now be seeing an overview of Exon's algorithm. For a more detailed explanation, the paper (Kassam et al., 2022) can be consulted.

Definitions

Before jumping to the steps that define the algorithm, let's understand some essential concepts:

- **Soft half-connections:** S-connections, as previously mentioned, are utilized for enhancing performance. The term "soft" denotes the implicit nature of these connections, being created "on-demand" by the library when there are messages to be sent or received, without the need for explicit connect/close API methods. The term "Half-connection" means that each connection operates in only one direction. Depending on the direction of the half-connection, either a sender-side record (s-record) or a receiver-side record (r-record) is associated. These records store the necessary state for the s-connection. At any given moment, it is possible that only the s-record or the r-record exists. This occurs because an s-record is the first to be created when a message needs to be sent, while the r-record is generated upon receiving a REQSLOTS message from the sender. The s-record is also the first to be garbage collected after a period of communication inactivity, with the r-record being only garbage collected when a REQSLOTS message, that renders any existent slots invalid, is received.

- **Slot:** A slot represents the obligation of the receiver to deliver a message *m* sent by the sender node unless the sender forgoes this obligation. A slot is stored in a receiver-side connection record.

- **Envelope:** An envelope, created from a slot, represents the possibility of creating a token with the same ID as the envelope (consumes the envelope) and a payload (user message). The same payload cannot be associated with more than one envelope. The envelope is stored in a sender-side connection record.

- **Token:** A token is associated with a user message *m*, and represents the obligation of the sender to request the receiver to deliver *m* until the delivery is acknowledged. A token is stored in a sender-side connection record.

Messaging steps

The basic behaviour of Exon's algorithm is the following:

1. The sender node has a user message to be sent to the receiver node. It starts by sending a REQSLOTS message, to request the allocation of a slot at the receiver node.

2. The receiver node, upon receiving the REQSLOTS message, creates a slot and sends it to the sender node through a SLOTS message.

3. After receiving the slot, the sender node creates an envelope using that slot. Then, a token is created from the envelope and the user message, and sent in a TOKEN message to the receiver.

4. The receiver node receives the TOKEN message, deletes the slot identified by the envelope present in the token, delivers the user message (payload) to the application, and sends an acknowledgement message. This ACK message tells the sender node that it can safely discard the token, as the user message was successfully delivered.

Let us explore in more detail the process of sending payloads:

1. **Requesting slots:** To send a user message, the sender node needs to "book" a unique slot at the receiver through a REQSLOTS message. Multiple slots can be requested using a single REQSLOTS message[2].

    - If the sender does not have an available envelope to send the user message, the message is queued, and a REQSLOTS message is sent to the receiver. This procedure is also performed by the sender when it does not have a previous s-record (sender-side record) related to the receiver. In this case, before realizing the procedure, an s-record is created.
    - When the sender does possess available envelopes, a token is created from one of the available envelopes and the user message. Afterwards, the token is sent to the receiver in a TOKEN message. The envelope is consumed, ensuring that no more user messages are associated with the same slot from the receiver. From here, the process jumps to step 4 of the algorithm.

2. **Sending slots:** The receiver node upon receiving a REQSLOTS message, creates the requested slots and sends them to the sender. A REQSLOTS message is also used by the receiver for garbage

[2] By requesting multiple slots in a single REQSLOTS message, the first two steps of the algorithm are shared by multiple messages. If 10 slots are requested at once, the number of steps required to send 10 payloads is 2 + 10 * 2 = 22 steps. When requesting slots individually, the number of steps required to send 10 payloads is 10 * 4 = 40 steps. The number of slots requested depends on a base number of slots that should be available for the sender at the receiver, the number of queued messages and the number of envelopes.

collection purposes, more specifically, to remove outdated slots[3].

3. **Sending payload (token):** The sender node receives slots from the receiver, and converts them to envelopes. Then, the sender generates tokens from the available envelopes and queued payloads and forwards the tokens to the receiver. This step does not end without requesting more slots. This is done for two reasons: the first is because the envelopes may not have been enough for all the payloads in the queue; the second is for efficiency since potentially more messages will be sent, and by reserving slots in advance, only one RTT is needed to deliver and confirm the delivery of the message.

4. **Payload delivery and acknowledgement :** The receiver upon receiving a token, checks for the existence of a slot that corresponds to the envelope inside the token.

    - If there is a corresponding slot, the slot is deleted, the payload is delivered to the application, and an acknowledgement message is sent to the receiver, informing that he can discard the token.
    - Else, the token is assumed to be a duplicate, resulting only in an ACK message being sent. This acknowledgement is required, as a previous ACK message may have been lost.

There are still some details not presented here. To check the full algorithm and the proof that it provides exactly-once delivery guarantee, check Kassam et al. (2022).

3.2 Exon-lib

In this section, I will be presenting some architecture decisions and features of the Exon-lib, a comparative analysis with the TCP protocol, and lastly, some known limitations of the Exon protocol.

3.2.1 Architecture and Features

The current implementation of the Exon protocol, named Exon-lib, is a Java library developed on top of the UDP transport protocol. The implementation is multi-threaded and makes use of two threads: one to run the algorithm code, and another to read messages from the network. The communication between the threads (including the local client thread) is done using two ArrayBlockingQueues. One queue is used

[3] To ensure the oblivious property, when the communication ceases (i.e. when there are no user messages to send or to be acknowledged), the sender node sends a REQSLOTS message that forces the receiver to discard all the allocated slots. This allows the connection state, of this sender-receiver pair, to be discarded at the receiver side as well.

to deliver the messages to the local client; the other is used by the main thread to receive all messages, both the ones created whenever the local client wants to send a payload, and the protocol messages originated from the network. The Exon-lib contemplates a flow control, that blocks the local client thread when a threshold of unacknowledged messages is reached. The library also provides a mechanism that automatically calculates the number of slots that should be requested in advance through a formula based on the bandwidth and latency of the connection.

I recommend reading Kassam et al. (2022) for a better understanding of these issues, and also, to check the performance comparison between Exon and TCP.

3.2.2 Comparative analysis with TCP

I'll be presenting some observations found in the paper relative to Exon and TCP protocols.

The TCP protocol is one of the most common choices when reliability is a requirement. TCP provides exactly-once delivery guarantee within a connection, and on top of that, good performance. However, it has some shortcomings, that are addressed by the Exon protocol. The first problem is that exactly-once delivery is not assured when the TCP connection fails. This problem is solved by Exon by not requiring timing assumptions to ensure correctness. The second problem is the lack of performance that results from the combination of HOL blocking and TCP multiplexing. This latter problem gets even worse in the presence of message loss due to TCP's congestion control. Exon's order-less property prevents head-of-line blocking and thereby avoids delayed message delivery in both a particular stream and the remaining multiplexed streams[4].

The paper presents various experimental evaluations. These evaluations are performed under different parameters, such as loss rate, bandwidth, and delay. The protocols were tested in a simple topology with only one router between two hosts, where the router was set up to produce message loss and delay in some situations.

The evaluations performed show that Exon is significantly superior to TCP regarding latency and throughput when under packet loss. The HOL blocking of TCP allied to the congestion flow under packet loss drastically affected TCP's performance. In an RPC[5] scenario, with an RTT of 10ms, a bandwidth of 100 Mbps, and a loss rate of 5%, the Exon protocol was able to answer 12.8 times the amount of requests attended by TCP. Also, in an environment with no loss rate, the Exon protocol achieved similar performance to TCP, being slightly worse due to the overhead of the REQSLOTS and SLOTS messages. This overhead

[4] Exon does not employ the concept of a stream as TCP does. However, for this comparison, let's assume these streams as a set composed of the messages sent, currently being sent, and those yet to be sent from a particular source to a specific destination until a graceful conclusion of the communication.

[5] RPC stands for Remote Procedure Call.

was shown to reduce with the increase of both channel bandwidth and round trip times (RTT). With an increase of channel bandwidth from 10 Mbps to 100 Mbps, the overhead, compared to TCP, went from 7% to 1%. And, with the increase in RTT from 10 ms to 100 ms, the overhead dropped from 10% to 2%.

Summing up, Exon is capable of providing exactly-once delivery guarantee under network failures (admitting an eventual recovery from the network failure), and demonstrated the capability of performing at nearly the same level as TCP under normal fault-free conditions, and much better as the packet loss increases.

3.2.3 Problems and Limitations

While Exon exhibits great capabilities, it is not flawless, and so we will now be delving into some of its problems, limitations and missing features. The issues presented here should eventually be addressed, through an update of the Exon-lib or handled by the messaging middleware, to ensure reliable messaging behaviour under different environments and requirements.

Slight performance loss under low bandwidth and low network latency In the comparative analysis above, we observed that Exon's overhead over TCP is higher in low bandwidth and low latency networks. This slight limitation is acknowledged in the paper, and some suggestions are provided to minimize the overhead, such as carrying REQSLOTS and SLOTS inside TOKEN and ACK messages.

Order-less We have seen that Exon is a protocol that does not offer message ordering guarantees, opting for performance instead. Even though the ordering of messages is not provided by Exon, it can still be provided if required through the implementation of a higher layer. Message ordering is a mechanism required by several messaging patterns, thus it is one of the main problems to be addressed.

Integrity check Exon-lib does not currently feature an integrity check mechanism, meaning that corrupted messages may be delivered. The defined system model assumes that messages are not corrupted, however, the employment of error detection and correction mechanisms is crucial to ensure reliable message delivery.

Security Exon-lib does not provide channel security features such as authentication, authorization and message encryption. These features are a must-have when the components that utilise the protocol are put in production, to ensure a secure deployment in real-world scenarios. Implementing channel security prevents malicious users from disrupting the component's normal behaviour and ensures protection

against leaks of private information.

Flow control Exon-lib provides a flow control mechanism to prevent memory build-up and performance issues. This mechanism blocks the client thread when a threshold of unacknowledged messages is reached. Despite being a useful and wanted feature, blocking the client thread may not be the ideal solution for some situations. So, allowing this feature to be turned off, returning with a value that informs that a message cannot be sent at the moment, or providing an event-driven mechanism that informs when a client may send another message can be beneficial, as it enables the client thread to perform other work, in contrast to being idle.

Message fragmentation and merging Having message fragmentation and merging features in Exon-lib is really important. Let's consider a client that wants to send a message 50 kilobytes large. Due to the MTU[6] limit, the network can only handle packets of about 1.5 kilobytes, consequently the message cannot be sent through the network without being split into smaller packets. The fragmentation can be done at the IP level, however, if one of these smaller packets does not reach the destination, the whole message needs to be retransmitted, causing unnecessary network traffic. By performing fragmentation and merging at the Exon protocol level (or higher), the 4-step algorithm would need to be performed for each fragment, which results in a higher overhead due to REQSLOTS and SLOTS messages, but ensures that only the tiny fragment lost is retransmitted, not the entire 50 KB message. So, not providing message fragmentation and merging, can seriously hinder the performance of the communication.

The concept of fragmentation and merging, though applied in reverse order, is also true for smaller messages. For very small messages, merging them at the sender's side and unmerging at the destination can reduce the total number of steps required for delivery, leading to improved network efficiency and enhanced performance.

Lack of congestion control As seen above, the Exon-lib incorporates a flow control mechanism to avoid overwhelming the destination when faced with a high volume of incoming messages. However, a congestion control mechanism can also be beneficial to avoid exhausting the network bandwidth. This is important to maintain an optimal performance of both the Exon protocol and other applications relying on the network.

[6] Stands for "Maximum Transmission Unit", and represents the maximum size of a packet that can be transmitted over the network without fragmentation.

Part II

Core of the Dissertation

Chapter 4

Middleware Architecture: A Generic Framework for Socket Extensibility

This chapter introduces the Middleware Architecture, designed with usability and extensibility at its core to enable the creation and integration of custom socket implementations over a transport layer that assures exactly-once delivery. This architecture serves as a prototype for exploring communication patterns in a controlled environment under exactly-once semantics.

The primary design goals of this middleware include usability, reliability and ease of integration for specialized communication protocols. While the architecture does not prioritize scalability, efficiency, security and persistence — important considerations for future iterations — it lays a robust foundation for extensibility, allowing developers to experiment with different socket designs, and for simplicity of communication by ensuring exactly-once message delivery even under mobility scenarios.

Since security mechanisms, such as authentication and encryption, as well as persistence mechanisms, were intentionally excluded from this prototype, the system assumes a private, trusted environment where nodes behave cooperatively and do not crash, enabling the architecture to focus on usability and extensibility.

Through this chapter, we delve into the structural design, operational principles and design decisions of the middleware prototype, while detailing how it achieves its goals.

4.1 Middleware Design Goals

The primary goal of the messaging middleware is to deliver a highly usable, reliable, and extensible platform to facilitate communication between distributed systems. Having ZeroMQ, nanomsg, and NNG as inspiration, this middleware aims to provide a robust foundation for communication while addressing specific challenges, namely exactly-once delivery. Below are the key design goals that guided the development

of the middleware.

4.1.1 Usability

A major focus of the middleware design is usability, ensuring that developers can easily integrate and leverage the library in a variety of communication scenarios. Usability is enhanced by the following features:

1. **Exactly-Once Delivery Semantics**

   The middleware facilitates the creation of communication by ensuring exactly-once delivery of messages, even in the face of network problems, such as prolonged network partitions. This reliability removes the need for developers to implement additional mechanisms to ensure message delivery, a common challenge when using middleware like ZeroMQ, which does provide exactly-once delivery when using TCP as the transport protocol, but ends up discarding not-delivered messages when the connection breaks.

2. **Mobility**

   The middleware supports exactly-once delivery under mobility scenarios. Nodes can move between networks while maintaining communication reliability, addressing challenges inherent to mobile and dynamic network environments.

3. **Broker-less Architecture**

   Follows a broker-less design to eliminate the need for additional resources and administrative tasks typically required by broker-based systems, thereby facilitating the deployment and reducing overall system complexity.

4. **Thread-Safe Operations**

   Unlike ZeroMQ, where sockets are designed to be used by a single-thread, this middleware supports thread-safe socket operations. This feature simplifies system design by allowing multiple threads to send and receive messages using the same socket concurrently. For example, a single socket can act as a task source, allowing multiple threads to read tasks from it and send back their responses. This approach removes the need for individual sockets for each worker thread and an extra socket for a master thread to distribute tasks, which is usually necessary when using ZeroMQ.

5. **Asynchronous Communication**

   To enhance performance and minimize blocking in client threads, the middleware adopts an asynchronous communication model. When sending a message, a client thread submits the message

to the middleware, which handles transmission independently. However, due to the flow control mechanism, sending messages may be blocked under certain conditions to prevent congestion. Similarly, message reception is decoupled from client thread execution, ensuring efficient non-blocking operations for reading messages.

6. **Socket API**

   The middleware offers an intuitive API with core operations similar to traditional sockets, such as `link()`[1], `close()`, `send()`, `recv()`, and `poll()`, making it easier for developers to learn and use due to its familiar interface. The underlying communication mechanisms, such as establishing and maintaining socket connections, are abstracted for simplicity and will be explained in detail in subsequent sections.

7. **Data Agnostic**

   The middleware is designed to be data-agnostic, allowing clients to send any type of data as part of their payload. This flexibility ensures the middleware can support diverse application requirements, from simple text messages to complex binary objects, without imposing restrictions on the type of data being exchanged.

8. **Multiple "Plug & Play" Patterns**

   The middleware supports multiple communication patterns to simplify the creation of solutions for different scenarios and requirements with minimal configuration. The provided patterns are: Request-Reply, Push-Pull and Synchronized Publish-Subscribe[2].

4.1.2 Modularity and Extensibility

The middleware is designed to be modular and extensible, enabling developers to expand its behavior to meet specific requirements. Key considerations include:

1. **Extensibility and Modularity**

   The middleware is designed with extensibility in mind, allowing developers to extend its functionality without compromising the integrity of the system. This extensibility is particularly focused on sockets, ensuring the middleware can evolve to meet diverse application requirements by enabling developers to create custom socket types tailored to their specific use cases.

[1] Analogous to connect().

[2] Publish-subscribe variant where publishers ensure synchronization across the subscribers. For a subscription topic, the subscribers receive the messages at the speed supported by the slowest subscriber.

2. **Separation of Concerns**

   The design tries to adhere to the principle of separation of concerns, ensuring that individual components are focused on specific responsibilities. This modular approach simplifies development and testing.

4.1.3 Security and Misuse Prevention

While security is not a primary focus of this iteration of the middleware, some foundational measures have been incorporated to safeguard against potential misuse and malicious scenarios. These considerations ensure a degree of robustness and prepare the middleware for future enhancements in security.

1. **Protection Against Malicious Scenarios**

   Although features such as encryption and authentication are beyond the scope of this iteration, the middleware includes design choices that inherently provide protection against certain malicious behaviors, such as:

   - Incompatible sockets cannot communicate due to a strict linking mechanism.
   - Messages received before the successful establishment of a link are automatically discarded.
   - Flow control mechanisms ensure that the receiver dictates the volume of messages it is willing to process, mitigating potential spam attacks.

2. **Misuse Prevention**

   The middleware minimizes the risk of misuse by carefully designing the visibility and accessibility of methods, attributes, and classes. This ensures that both developers using the library and those modifying (extending) it are safeguarded from introducing errors or unintended behavior.

4.1.4 Scalability

While usability is the primary focus, the middleware aims to provide acceptable performance that make it a viable solution. Although it may not match solutions optimized for performance like ZeroMQ, the middleware strives to present acceptable performance that enable it to support a range of applications.

4.2 System Model

The middleware's design is based on a set of assumptions that guide its functionality and limitations. With Exon-lib as the underlying transport, its assumptions become part of the middleware's assumptions by

transitivity.

1. **Node Network and Node Identities**

   Considers a networked or distributed system of any number of nodes uniquely identified.

2. **Node Stability**

   Node interactions can be long-lived (persistent) or transient.

3. **Node Mobility**

   Nodes may exchange (move to other) networks during interactions, for example, as in vehicular networks.

4. **Node Reliability**

   Nodes are assumed to be reliable and to not crash during operation.

5. **Asynchronous network**

   "The network is asynchronous, with no global clock, no bound neither on the time it takes for a message to arrive, nor on the processing speeds." (Kassam et al., 2022)

6. **Network unreliability**

   "The network is unreliable: messages can be lost, duplicated or reordered (but are not corrupted)." (Kassam et al., 2022)

7. **Network Partitions**

   "The network may have long partitions, but these will eventually heal." (Kassam et al., 2022)

8. **Collaborative and Secure Environment**

   The middleware assumes a safe, private environment where nodes cooperate, with no malicious entities intentionally acting to sabotage the system, and where the exposure of sensitive information does not pose risks due to the lack of security features, like encryption and authentication.

4.3 Core Components

This section explores the core components of the messaging middleware, essential for achieving the system's design goals. We begin with a System Overview, offering a high-level explanation of how the middleware operates. This overview introduces the key entities and their basic relationships, providing a foundation for the detailed component descriptions that follow.

The class diagram of Figure 18 illustrates a main view of the system's architecture, focusing on primary components and relationships. Additional detailed diagrams may be included in the respective subsections for a better understanding.

By the end of this section, the reader will have a comprehensive understanding of the building blocks and architecture of the middleware.

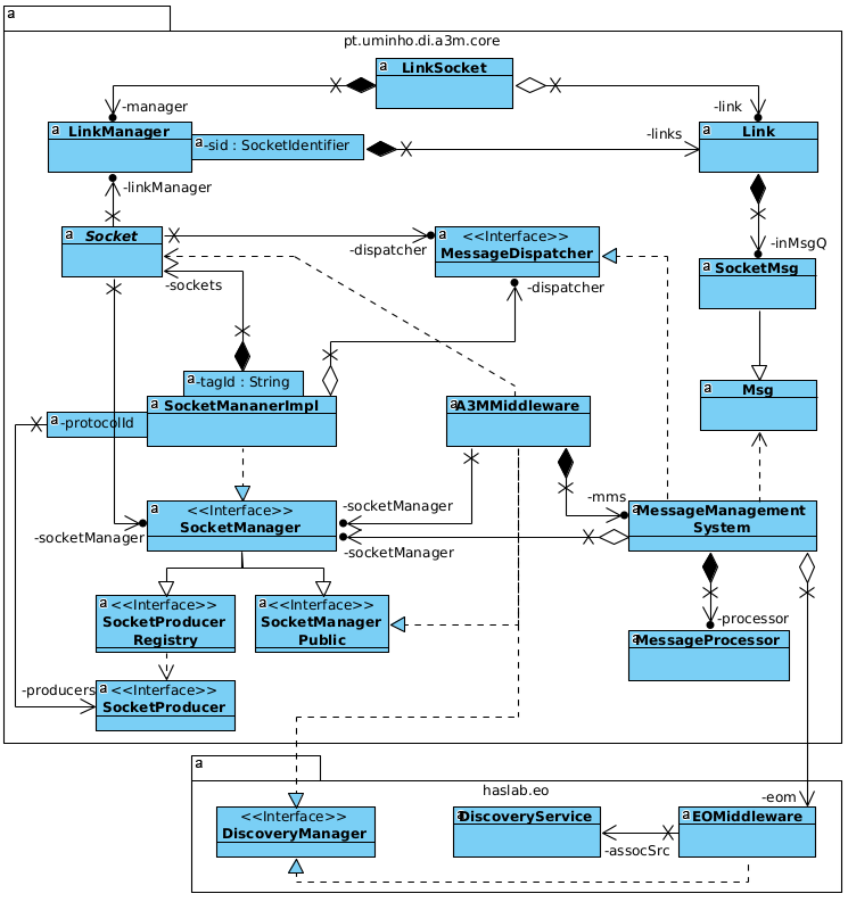


Figure 18: Middleware Architecture Class Diagram

4.3.1 System Overview

The core of the middleware system is the **Node**, the central unit that represents an instance of the middleware.

Each **Node** is uniquely identified by a **NodeId** within the network. While nodes are referenced by logical identifiers (such as "Library"), the middleware still requires transport layer addresses to route **messages**. These associations between node identifiers and transport addresses are typically managed by a **Discovery Service**.

Nodes act as containers for **Sockets**, which are the communication endpoints within the system. Each

socket is identified by a unique **TagId** within the node. The combination of a node's identifier (**NodeId**) and the socket's **TagId** creates a globally unique identifier known as the **SocketId**. Communication between nodes occurs through these sockets.

To enable communication between two nodes, the developer selects or implements sockets that match the desired communication requirements. Once the appropriate sockets are created, a **Link** must be established between them. A link represents a conversation, and it can only be established when both nodes agree to communicate. The agreement is primarily based on the compatibility of the **Messaging Protocols** – the socket's protocol must be present in the remote socket's list of compatible protocols.

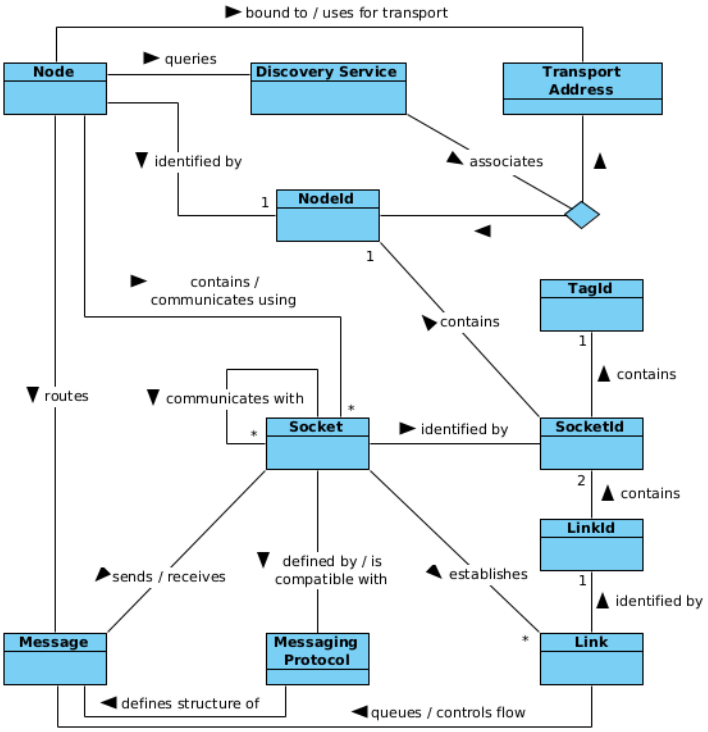


Figure 19: Middleware Domain Model

4.3.2 A3MMiddleware

The **A3MMiddleware** instance is the entry point for using the middleware, with each instance representing a **Node** in the topology. This class serves as the central component of the system, which acts as a facade, centralizing and encapsulating functionality, particularly for general configuration and socket management.

The encapsulation provided by **A3MMiddleware** enhances the middleware's usability by allowing clients to perform operations directly through the middleware instance rather than interacting with underlying components. For example, clients can create a new socket with

```
middleware.createSocket(...)
```

instead of accessing the socket manager directly via

```
middleware.socketManager().createSocket(...)
```

This approach not only simplifies the API but also acts as a safety measure, restricting direct access to the internal components of the middleware.

In terms of relationships:

- **A3MMiddleware** interacts with the **Socket Manager** for tasks such as registering new socket types and managing the lifecycle of sockets.
- It initializes the transport layer of the middleware, the **Exon** library, and interacts with it to apply configuration changes requested by the client.
- It also sets up the **Message Management System**, supplying it with necessary components: the **Exon** instance and the **Socket Manager**.

By centralizing these responsibilities and interactions, **A3MMiddleware** plays a crucial role in ensuring seamless integration and operation of the middleware's subsystems.

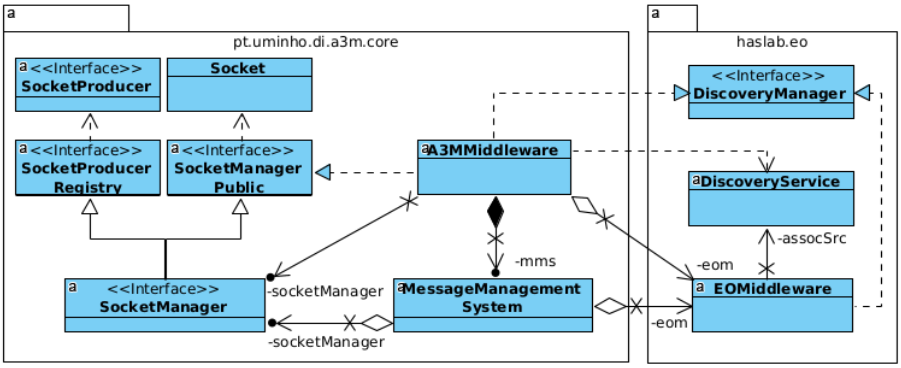


Figure 20: A3MMiddleware Component Relationships Diagram

Identifying a Node

A **Node Identifier** (or **NodeId**) uniquely identifies a node within an overlay network built using the middleware. The identifier is a crucial part of the message routing information.

Addressing communication endpoints with logical identities like "Book-Service", is far more convenient than using transport addresses that are a combination of an IP address and a port. In addition to being

much more readable and memorable, it also transcends spatial and temporal locality, making developing and maintaining code much simpler. The node identifier can be used to query a discovery service which contains or will eventually contain the transport layer information of the node of interest. Nodes that are to be made available and visible to other nodes, usually called static nodes of a topology (servers), can have their transport layer addresses introduced in a discovery service, which the middleware contacts to learn where to forward the messages.

When assigning identifiers to nodes, it is crucial to ensure their uniqueness within the network. The node identifier plays a key role in routing messages, where duplication can lead to routing errors. Multiple nodes sharing the same identifier are perceived as one, causing misdirected messages. To account for mobility scenarios – as detailed in Support for Mobility Scenarios) – the transport address associated with a node identifier is updated from received messages. With messages arriving from different transport addresses but with the same node identifier, the associated transport address will switch back and forth. As a result, messages are unlikely to reach their intended destination, and delivery state corruption may occur. This compromises exactly-once delivery, potentially leading to issues such as infinite retransmissions (undeliverable messages due to state corruption) or incorrect acknowledgments for messages that were never received (overlap of message identifiers).

State Management

Middleware instances maintain a state attribute to manage their lifecycle and delimit when specific operations are allowed.

Currently, the middleware instance can take one of two values: `CREATED` or `RUNNING`. The `CREATED` state enables initial configuration related to transport association discovery and socket producer registration. The `RUNNING` state activates socket management operations.

The initial idea included two more states: `CLOSING` and `CLOSED`. However, these were not employed for the following reasons:

1. Since messages can arrive at any time, permanently closing a node would break exactly-once delivery guarantees.

2. With a persistence mechanism, temporarily closing a node would be possible, as long as all data required to restart the middleware instance without breaking exactly-once delivery guarantees was persisted. However, because nodes crashes were not a focus, the persistence mechanism was considered not essential and was deferred for future work.

Key Methods

The key methods of a middleware instance can be categorized into three sections: **Initialization**; **Node Discovery**; and **Socket Management**.

In general, the implementation of these methods – besides constructors – involves delegating the received parameters to the corresponding component methods that share the same signature, while performing concurrency control and state verification to avoid conflicts and ensure the methods are executed only when the system is in an appropriate state.

Initialization Methods

The **Initialization** methods include various constructors and "start" methods, used to configure the middleware instance and transition its state to enable the execution of socket-related operations.

The available constructors accept a subset of the arguments from the following constructor, which provides the most customization:

```
+A3MMiddleware(nodeId, address, port, N, socketProducers)
```

These constructors provide different configuration options, using the following parameters:

- **nodeId**: A string identifier that uniquely distinguishes the node within the topology.
- **address**: A string representing the local IP address for binding the Exon library's UDP socket. In mobility scenarios, the address must be set to `null` to bind the UDP socket to a wildcard address, enabling it to listen on all network interfaces.
- **port**: An integer between 0 and 65536 (inclusive) representing the local port for binding the Exon library's UDP socket;
- **N**: A flow control configuration value for the Exon library specifying the maximum number of slots a node can request to another node at once. Each slot reserved allows a message to be in transit. If this value is not specified or is `null`, a default of 100 slots is used.
- **socketProducers**: A list of producer objects that create different types of sockets. If not provided or `null`, or if the list is empty, a default set of producers is used.

The `start() : void` method can be called after creating a middleware instance to transition the **Middleware** and its **Message Management System** (MMS) to a running state. This activates the

middleware's socket management operations and allows the MMS to begin processing messages received by the transport library (**Exon**).

Node Discovery Methods

The **Node Discovery** methods are part of the `DiscoveryManager` interface, which is implemented by the underlying transport library (**Exon**). Although the middleware instance does not manage routing directly, it exposes the relevant API for configuring the transport layer. By doing so, the middleware centralizes access to these operations, enabling clients to interact with the discovery functionality through the middleware instance. The methods made available for this purpose include:

- `+setDiscoveryService(DiscoveryService)`: Enables setting a discovery service.
- `+registerNode(nodeId, taddr)`: Enables local registration of an association between a node identifier and a transport address (a combination of IP address and port).
- `+unregisterNode(nodeId)`: Enables local removal of a previously registered association.

Socket Management Methods

The **Socket Management** methods are part of the `SocketManagerPublic` interface, which is implemented by the underlying **Socket Manager** component. These methods provide clients with the ability to manage sockets directly through the **A3MMiddleware** instance. The following methods are available via the middleware instance:

- `createSocket(tagId : String, protocolId : int) : Socket`: Creates a socket using a specified **tagId** (a locally unique identifier for the socket) and **protocolId** (an integer that identifies the socket type).
- `createSocket(tagId, protocolId, socketClass : Class<T>) : <T extends Socket>`: Similar to the above method, but with an additional **socketClass** parameter that allows the created socket to be cast to the specified class. If the protocol identifier does not match the provided class type, the socket creation will be aborted and an exception will be thrown. This method is useful for sockets with custom APIs (e.g., `PubSocket` from the Publish-Subscribe messaging pattern, which has additional methods to manage subscriptions). For example:

```
PubSocket pubSocket = middleware.createSocket("NewsPublisher",
        PubSocket.protocol.id(), PubSocket.class);
```

Instead of:

```
Socket socket = middleware.createSocket("NewsPublisher",
        PubSocket.protocol.id());
PubSocket pubSocket = (PubSocket) socket;
```

Additionally, the default socket classes provide static methods that allow for easy creation of sockets, assuming the relevant producers have been registered in the middleware. For instance, a `PubSocket` can be created as follows:

```
PubSocket pubSocket =
        PubSocket.createSocket(middleware, "NewsPublisher");
```

- `startSocket(...)` methods, having the same signatures as `createSocket()`. These methods start the socket immediately after creating it.
- `closeSocket(Socket) : void`: This method closes a socket.

In addition to the **SocketManagerPublic** interface methods, the middleware provides one additional method related to socket management. This method is exposed by the A3MMiddleware for customization, enabling developers to extend the middleware's capabilities by defining and registering custom socket types:

- `+registerSocketProducers(List<SocketProducer>)`: Registers custom socket producers to enable the creation of new types of sockets.

4.3.3 SocketManager

Each middleware instance has a **Socket Manager**, a key component of the middleware architecture, responsible for several socket-related functionalities. Its responsibilities include:

- **Socket Lifecycle Management**: Creating sockets of specified types, tracking and maintaining all active sockets, and ensuring proper cleanup when sockets are deleted.

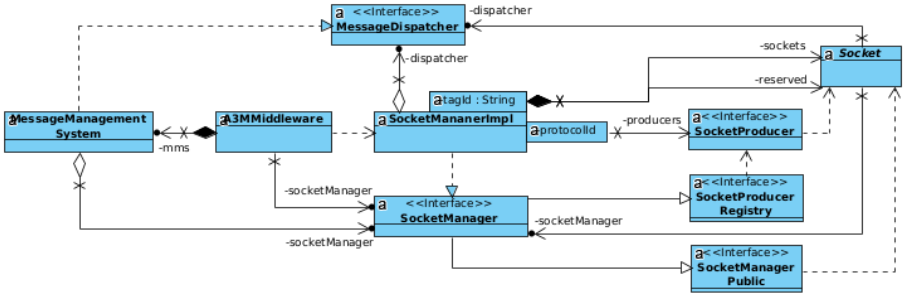


Figure 21: Socket Manager Relationships Diagram

- **Socket Producer Registry**: Acting as a registry for socket producers[3], which define the types of sockets that can be created. The Socket Manager handles the registration and removal of these producers, enabling dynamic customization and extensibility of supported socket types.

Key Methods

The `SocketManagerImpl` class implements the `SocketManager` interface, which combines two interfaces, `SocketManagerPublic` and `SocketProducerRegistry`, and an additional method, `startSocket(Socket)`.

The `SocketManagerPublic` interface defines the methods for socket lifecycle management:

- `+createSocket(...)` **methods:** These methods begin by validating the tag identifier to ensure it is valid and unique. Next, they verify the existence of a socket producer for the requested socket type. Once these conditions are met, a new socket instance is created and associated with the socket manager.

- `+startSocket(...)` **methods:** These methods create the socket if it not yet created and then start it. The starting procedure consists of: executing the socket's specialized initialization procedure[4], exposing the socket to the Message Management System to enable receiving capabilities, and assigning a **Message Dispatcher** to enable sending capabilities.

- `+closeSocket(Socket)` **method:** If the socket manager owns the socket and its state is neither closing nor closed, the closing procedure is initiated. When immediate closure is not possible, the procedure continues asynchronously, with the socket waiting until its state allows closing. Once the socket is ready for closing, the final steps are executed: the socket's specialized closing

[3] A `SocketProducer`, mapped using a protocol identifier, is a function that takes a single argument, a socket identifier, and uses it to create and return a socket instance.

[4] Each socket type may have a specific initialization procedure.

procedure is executed, its state is changed to `CLOSED`, and it is removed from the socket manager's collection.

To ensure consistency and atomicity in a multi-threaded environment, certain methods acquire write locks on both the socket manager and the socket. Also, while the `A3MMiddleware` instance handles client-thread concurrency, internal threads also access the `SocketManager` – namely, the Message Management System's thread for reading operations – hence the need for its own locking mechanism for concurrency control.

Finally, the `SocketProducerRegistry` interface methods, for managing socket producers, are implemented as follows:

- `+registerProducer(SocketProducer)`: Validates the provided socket producer to ensure it creates valid sockets. A valid socket must: not be null; be in the `CREATED` state; not have a message dispatcher or socket manager already assigned; and, have a non-null protocol identifier that has not been registered previously.
- `+removeProducer(protocolId : int) : boolean`: Removes the socket producer associated with the specified protocol identifier, if it exists.

Preventing Communication with Unstarted Sockets

The middleware allows sockets to be created without being immediately started due to enable critical pre-configuration operations. With this in mind, allowing an unstarted socket to receive messages for processing would not be logical.

The **Socket Manager** keeps unstarted sockets separated from started sockets. This design ensures that only started sockets receive messages while reserving identifiers of unstarted sockets to prevent collisions. Additionally, using separate collections improves efficiency, when handling messages of unstarted sockets, by avoiding unnecessary operations, such as acquiring locks for unstarted sockets only to determine that they cannot handle messages[5].

4.3.4 Socket

The **Socket** component serves as the main building block of the messaging middleware, acting as the communication endpoint that allows the exchange of messages between applications.

[5] An unstarted socket should be perceived as non-existent by other sockets. However, this does not prevent other sockets from dispatching messages to it – specifically, link establishment requests. Handling these request involves sending a simple error message stating the socket does not exist, which does require interacting with a socket.

Designed with **extensibility** in mind, the `Socket` component is implemented as an abstract class. This design choice allows it to act as a template for socket specialization.

It incorporates two categories of methods:

1. **Unmodifiable Methods**: Establish the core behavior of the socket and provide the foundation for designing specific socket types.
2. **Abstract and Modifiable Methods**: Enable customization and extension to meet specialized communication requirements.

Relationships

The **Socket** component collaborates with various elements of the middleware to fulfill its role:

- Issues the intervention of the **Socket Manager** for lifecycle-related operations, such as starting and closure. The **Socket Manager** is responsible for completing these operations, through the execution of procedures outside the "jurisdiction" of the **Socket** component.
- Manages and interacts with **Link Sockets** for message processing operations.
- Delegates outgoing messages for dispatching to the **Message Dispatcher**.
- Receives incoming messages from the **Message Management System**, enabling bi-directional communication.
- Delegates incoming messages related to the Link Management Protocol to its own **Link Manager** instance.

By coordinating with these components, the **Socket** enables the capability of the middleware to handle diverse communication scenarios.

Key Methods

The `Socket` class is one of the most complex classes in the middleware, with a variety of methods that differ in visibility and modifiability. To simplify understanding, the methods are not categorized based on these characteristics but are instead grouped by functionality and discussed in the appropriate sections.

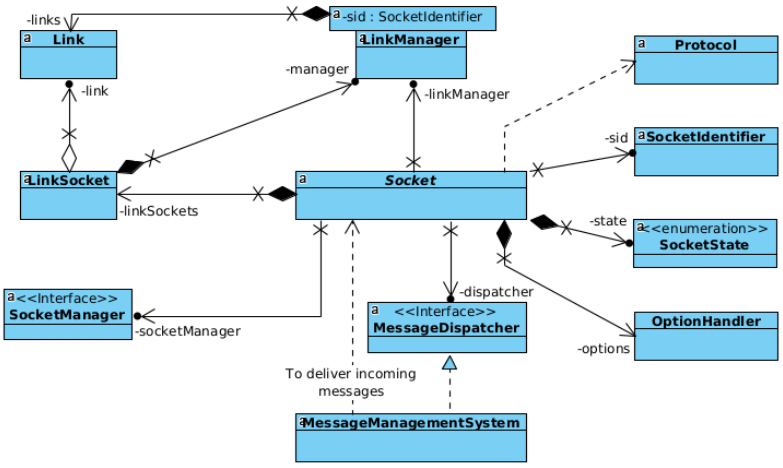


Figure 22: Socket Component Relationships

Socket Lifecycle

This section explores the lifecycle of a socket, describing its possible states, the transitions between these states, and the methods associated with each transition. For a clear explanation, the discussion is organized around the socket's four primary states: `CREATED`, `READY`, `CLOSING`, and `CLOSED`.

`CREATED State`:

Every socket begins its lifecycle in the `CREATED` state, which is initialized upon invoking the constructor: `#Socket(SocketIdentifier)`.

To maintain a generalized approach to socket creation, constructors for specialized sockets should also take only a single argument: the socket identifier. Following this design is advised to enable using the constructor as a `SocketProducer` to register a new socket type in the middleware.

With socket constructors taking a socket identifier as the single argument, a `CREATED` state enables pre-configuration. Any required setup steps can be performed in this state ensuring the socket is properly configured before being started and exposed for communication.

It is important to note that while constructors for specialized sockets may be publicly accessible, directly instantiating them will result in non-functional sockets. This is because communication can only occur when a socket is associated with a messaging layer. As such, to create functional sockets, users must utilize the middleware's API for proper socket instantiation and initialization.

`READY State`:

A socket transitions from the `CREATED` state to `READY` through a starting procedure, which is initiated

either directly on the socket or through the middleware instance.

The starting procedure is a collaborative process between the socket manager and the socket itself. The socket manager handles tasks, such as assigning a message dispatcher to enable message transmission, and registering the socket to make it visible to the message management system, enabling message reception. As for the socket, it is responsible for executing any custom initialization tasks before transitioning to the `READY` state.

Custom initialization logic for a socket is defined by the abstract method `#init() : void`, allowing subclasses to allocate resources or perform other setup actions required for their specialized logic.

`CLOSING/CLOSED States`:

Sockets have two states related to closing: `CLOSING` and `CLOSED`. The `CLOSING` state is necessary for scenarios where the socket cannot be immediately closed due to pending tasks that must be completed first.

A socket cannot close while it has active links. These links must undergo a proper unlinking process to avoid corrupting the linked sockets' conditions. Consider a socket distributing tasks across its established links in a round-robin fashion. If a linked socket closes without unlinking first, the distributing socket would continue to send tasks to the closed peer, unaware of its inoperability.

Before a socket transitions to `CLOSING`, custom closing conditions can be enforced by overriding the default closing methods. Overriding allows preventing closure by throwing an exception when certain conditions have not yet been met, or initiating a custom closing procedure where specific conditions are tracked before invoking the default closing process.

A socket can be closed using the middleware's `closeSocket()` method or directly on the socket. The available methods allow choosing between blocking and non-blocking behavior: `close()`, `close(timeout[6] : Long)`, `asyncClose()`.

Similarly to the starting procedure, the socket manager and the socket work together to complete the closing process and properly disconnect the socket from the middleware instance. To ensure the socket remains visible to the Message Management System during the `CLOSING` state, allowing the unlinking process to proceed without disruption, the socket manager only removes the socket from its internal collection after it transitions to the `CLOSED` state.

The socket instance plays a key role in the closing process via its `-closeInternal()` method. If no links exist, the socket's state is immediately set to `CLOSED`, the abstract `#destroy() : void` method is invoked to execute any custom cleanup logic, and any threads waiting on the socket are woken

up. If active links exist, the closing process is deferred until all links are properly unlinked. Once the last link is closed (while in the CLOSING state), the `closeInternal()` method is invoked again to transition the socket to CLOSED and perform the final cleanup steps.

Link Management

This section introduces the core functionalities of sockets related to **Linking** and **Unlinking**.

Each socket is characterized by a **Messaging Protocol**, identified by a unique **ProtocolId**. This identifier, representing the socket's type, is exchanged during the linking process for compatibility verification. Since protocol compatibility is crucial for establishing a link, all sockets must implement the following methods:

- `+getProtocol() : Protocol`: To retrieve the protocol of the socket.
- `+getCompatibleProtocols() : Set<Protocol>`: To return a set of protocols with which the socket is compatible.

For link management, the following methods are available to initiate and terminate links with a specific peer: `+link(peerId : SocketIdentifier)` and `+unlink(peerId : SocketIdentifier)`. These methods initiate the linking and unlinking processes, and then return, leaving the processes to be completed asynchronously. Furthermore, the methods can only be used when the socket is in the READY state and may be overridden by subclasses to enforce additional conditions before execution.

Since the `link()` and `unlink()` operations are asynchronous, having methods that allow waiting for link establishment and closure events is desirable:

- `{final} +waitForAnyLinkEstablishment(timeout : Long, notifyIfNone : boolean) : SocketIdentifier`: Waits for the establishment of any arbitrary link within the specified `timeout`. If `notifyIfNone` is true, an exception is thrown if no links are available.
- `{final} +waitForLinkEstablishment(socketId, timeout) : int`: Waits for the establishment of a link associated with a specific socket within the specified `timeout`. The method returns an event mask that indicates the outcome of the operation – whether it timed out, the link was established, or the link does not exist or was closed.
- `{final} +waitForLinkClosure(socketId, timeout) : int`: Waits for the closure of a link associated with a specific socket within the specified `timeout`. Also returns an event mask to determine whether the operation timed out, or if the link does not exist or was closed.

Additionally, there are several internal methods supporting the underlying linking and unlinking processes.

When a link is successfully established, the link manager notifies the socket by invoking the following internal method: `{final} ~onLinkEstablished(Link) : void`.

This method performs the following sequence of operations:

1. **Link Socket Creation and Registration**: The established link passed as an argument is used to create a `LinkSocket` and register it in the socket's collection, making it accessible to the socket's custom logic via a getter method.

   The creation of a link socket is done via:
   `#createLinkSocketInstance(peerProtocolId : int) : LinkSocket`, which subclasses can override the link socket creation method to return a specialized link socket instance that may depend on the peer's protocol.

2. **Incoming Queue Setup**: An incoming queue is created and associated with the link. Subclasses can override the `#createIncomingQueue(peerProtocolId) : Queue<SocketMsg>` method to return a queue with customized behavior, such as ordering logic.

3. **Custom Logic**: The abstract method `#customOnLinkEstablished(LinkSocket)` is invoked to allow subclasses to define any additional procedures needed upon the link establishment.

4. **Thread Notification**: Finally, any client threads waiting for a link to be established are woken up.

Just as a socket is notified when a link is established, it is also notified when a link is closed via: `{final} ~onLinkClosed(Link) : void`.

This method performs the following tasks:

1. **Link Socket Removal**: The associated `LinkSocket` is removed from the socket's internal collection, making it unaccessible to socket subclasses.

2. **Custom Logic**: The abstract method `#customOnLinkClosed(LinkSocket)` is invoked, enabling subclasses to define custom behavior for link closure.

3. **Last Link Steps**: If the closed link corresponded to the last link of the socket, the following steps are performed:

   - All threads waiting for the socket to be link-free are notified. [7]

[7] Certain operations provide an option to set a "notifyIfNone" flag. Setting this flag means wanting the waiting operation to be canceled when the socket becomes link-free.

- If the socket is `CLOSING`, the `closeInternal()` method is invoked to complete the socket's closing process.

Sending and Receiving Messages

With an understanding of links and their associated methods, which are required for communication between sockets to occur, we can now dive into the socket methods related to sending and receiving messages.

The `Socket` class provides three variants of the `send()` method to clients, with each successive variant being built on the previous by using default argument values:

- `+send(payload : byte[], timeout : Long, notifyIfNone : boolean) : boolean`
- `+send(payload : byte[], timeout : Long) : boolean` (Assumes `notifyIfNone` as false)
- `+send(payload : byte[]) : boolean` (Assumes `timeout` as null[8] and `notifyIfNone` as false)

The `notifyIfNone` flag determines whether the waiting operation should be cancelled with a `NoLinksException` if the socket is link-free (i.e. if it has no links).

Similarly, three variants of the `receive()` method are provided:

- `+receive(timeout: Long, notifyIfNone: boolean) : byte[]`
- `+receive(timeout: Long) : byte[]`
- `+receive() : byte[]`

These sending and receiving methods can be overridden to meet the requirements of each socket. For instance, if the operation is not supported, the method can be overridden to throw a `UnsupportedOperationException`. Additionally, if custom arguments are required, new methods can be created to fulfill such requirements.

The `Socket` class contains several other relevant methods related to sending and receiving processes. However, since these processes are critical to the design of the messaging middleware, they have dedicated sections to cover all the important details: Sending a Message and Receiving a Message.

[8] A `null` timeout indicates no time limit for the operation to complete.

Polling Events

The `Socket` class provides a variety of event-related methods, designed to enhance both usability and efficiency. These methods depend on the polling mechanism and its related entities, which are explained in detail in Polling Mechanism and Wait Queues.

The first group consists of static methods that allow clients to take advantage of the socket's event polling capabilities:

- `+addToPoller(poller : Poller, socket : Socket, events : int)` : This method registers the socket in a `Poller` which enables monitoring events of multiple objects.
- `+poll(socket : Socket, events : int, timeout : Long) : int` : This method was designed for occasional polling of events of a specific socket.

The above methods depend on the following internal methods:

- `#getAvailableEventsMask() : int` : This method verifies and returns the events currently available for the socket. Since read and write events depend on the specific semantics of each socket, the default implementation only reports error and close events. To support read and write events, specialized socket implementations should override this method.
- `#poll(pt : PollEntry) : int` : This method inserts an event waiter, defined by the `PollEntry`, in the socket's wait queue, and returns an event mask with currently available events. By default, the waiter is only queued if the socket state is not closed, and the currently available events are retrieved using `getAvailableEventsMask()`.

Options and Configuration

Sockets inherently come with options and configurations. As such, to support default and simple custom configurations, a flexible option handler mechanism is provided. For more complex scenarios, the `Socket` class is designed to allow specialized sockets to expose their own APIs, enabling custom configuration through dedicated methods.

The `Socket` class maintains a map of option handlers, each defined by two methods: `get() : value` and `set(value)`. This design facilitates configuration actions, such as parameter setting, while also supporting custom behavior. Examples of their usage are provided ahead.

Three key methods are related to socket configuration:

- `{final} <Option> +getOption(String, Class<Option>) : Option`: This method finds the option handler associated with the specified option identifier and uses it to retrieve the corresponding value. Since each option may be represented by a different class, the `Class<Option>` argument is required to determine the appropriate type casting. The following example demonstrates the method's usage:

```
int capacity = socket.getOption("capacity", Integer.class);
```

- `{final} <Option> +setOption(String, Option) : void`: This method finds the option handler associated with the specified option identifier and invokes its `set()` method, passing the provided `Option` object as an argument. Notice that the `set()` method is not obligated to replace the existing object associated with the option identifier. Instead, it may perform transformations or updates on the current object.

- `{final} #registerOption(String, OptionHandler<?>) : void`: This method enables the registration, removal, or replacement of option handlers. Its visibility is `protected` to prevent unwanted tampering with the option handlers, as improper modification of options could result in crashes when logic dependent on those options encounters unexpected values.

The default options available for the `Socket` class are as follows. These options will be further discussed in detail when their associated mechanisms are introduced:

- `maxLinks`: Sets a limit on the number of established links. Setting a lower value than the current number of established links has no effect on established links nor on pending link requests. The default value is `Integer.MAX_VALUE` which is equivalent to not having a limit.

- `allowIncomingLinkRequests`: Enables or disables the acceptance of incoming link requests. This option allows disabling link requests for sockets that should not receive them, effectively preventing this action. It does not affect established or pending links. By default, this option is enabled (`true`), meaning that accepting incoming link requests is allowed unless explicitly disabled.

- `retryInterval`: Defines the interval of time to wait before retrying the linking process when a non-fatal negative reply is received. By default, the interval is 50 milliseconds, starting from the moment the reply is processed.

- `capacity`: Associated with the flow control mechanism, this option sets the default link capacity – the number of incoming messages a socket is willing to queue at any given time for a linked socket. The default is 100 credits.

- `batchSizePercentage`: Also related to the flow control mechanism, this option specifies the size of batches for flow control credits as a percentage of the link capacity. By default, the batch size is set to 25%, meaning that if the capacity is 100 credits, flow control credits will be sent to the transmitter in batches of 25.

Most of these default options are implemented using a generic option handler. This handler essentially acts as a wrapper, enabling getting and setting of option values. Custom behavior for options can be implemented via specialized `OptionHandler` instances. Here are two examples:

1. **Constraints enforcement**: The `batchSizePercentage` handler enforces constraints in the `set()` method, ensuring that the value is not `null` and is situated between 0 and 1, thereby preventing invalid configurations.

2. **Value Immutability**: In some cases, an option may need to be immutable. For instance, consider implementing a `PAIR` socket of the `PAIR` messaging pattern where two sockets interact exclusively with each other. The `maxLinks` option handler can be replaced with an immutable handler with a value of 1. This immutable handler would have the `set()` method throw an exception when modification is attempted. With the value of `maxLinks` set to 1 and the `set()` operation prohibited, the `PAIR` socket would be restricted to a single link, ensuring exclusive communication.

4.3.5 LinkManager

Each **Socket** has its own dedicated **Link Manager** instance, which functions as an extension of the socket. Its responsibilities include:

1. **Link Management Protocol Handling**: Manages the dispatch and processing of messages related to the Link Management Protocol. This includes handling link establishment requests, replies and flow control messages.

2. **Lifecycle Management of Links**: Responsible for creating, maintaining and closing `Link` instances.

3. **Link Configuration**: Utilizes socket configuration options during the link establishment process:

    - **Link capacity** to configure the flow control mechanism of the socket's links. The capacity is sent as metadata during the link establishment process in the form of credits. These credits enable data messages to be sent once the link is established.

- **Retry interval** to schedule link establishment reattempts when the link establishment process fails due to the reception of a non-fatal negative reply.

Initially, these were supposed to be responsibilities of the `Socket` class, but since it already has many responsibilities, these link handling responsibilities were extracted to a `LinkManager` class, enhancing the code's modularity.

Key Methods

The majority of the `LinkManager` functionality is implemented through private methods that support the Link Management Protocol. These methods handle key tasks such as creating, dispatching, parsing and processing core messages. Since the protocol will be explained in a dedicated section, this section focuses only on the main methods, leaving the logic of the auxiliary methods that support them to be covered in the protocol's description.

The first key methods are:

- `+link(SocketIdentifier)`: Initiates a link establishment process (also referred to as the linking process) with the socket identified by the given socket identifier. This method only proceeds if there is no existing link associated with the identifier and if the maximum number of links has not been reached. If these conditions are satisfied, a link request containing the socket's metadata is dispatched to the identified socket.

- `+unlink(SocketIdentifier)`: If a link is established with the socket identified by the given socket identifier, this method initiates a link closure process (also referred to as the unlinking process) by sending an `UNLINK` message. Additionally, it can be used to cancel an ongoing or scheduled linking process.

Another key method is `~handleMsg(SocketMsg) : boolean`. Every message received by the socket is first delegated to the link manager through this method. Core messages, such as those related to link management and flow control, are intercepted and fully processed here. Data and control messages require approval before being returned to the socket for processing based on the specialized logic. The approval depends on whether the message corresponds to the current conversation between the two sockets.

The final key methods are part of an interface called `LinkDispatcher`. To prevent custom logic from introducing inconsistencies by directly sending core messages (excluding the `DATA` type) through the

transport layer, the `LinkManager` acts as the dispatcher for its own links. The methods in this interface include:

- `+onOutgoingMessage(Link, Payload) : void`: Verifies the payload to be sent over the specified link, ensuring its type is either a data message or a (custom) control message. If valid, this method constructs a socket message using the payload and sends it to the socket associated with the link.
- `+onBatchReadyEvent(Link, batch : int)`: Creates a flow control message containing a number of credits equal to the `batch` argument and sends it to the socket associated with the link.

Currently, the `LinkManager` does not provide support for extending the linking and unlinking processes with custom logic. However, the Link Management Protocol was designed with future extensibility in mind. Few modifications are required to enable such customizations.

4.3.6 Link

The conceptual definition of **Link** is a bond between two sockets. This bond is formed when both sockets agree on establishing it, and translates to the authorization of messages to flow between these two sockets, enabling them to hold a conversation.

The `Link` class has the following main responsibilities:

- Symbolize the connection between two sockets and holding the attributes required for its management. The existence of a `Link` object means two sockets have or are attempting to create a bond.
- Enable targeted communication. Since each socket can communicate with multiple sockets, a link enables message exchange with a specific socket.
- Allow queuing of incoming data messages until their delivery to the application.
- Control the flow of outgoing data messages to avoid overwhelming the remote socket.
- Like sockets, comply with polling mechanism (defined in section 4.4), enabling event waiters to subscribe and monitor notifications of read, write and close events.

Key Methods

The `Link` class contains a variety of methods belonging to multiple categories. Because the functionality displayed by these methods will be explained in detail in the specialized sections that follow this **Core Components** section, we will dive in brief descriptions of the categories comprising the methods of this class:

- **Link Management**: Methods required by the Link Management Protocol to manage the link, such as establishing and closing it.
- **Sending and Receiving**: Methods required to provide the message exchanging capability. The main methods of this category include:
  - `~queueIncomingMessage(SocketMsg)`: Queues an incoming data message and notifies the readiness to read a message;
  - `+receive(deadline : Long) : SocketMsg`: Attempts to retrieve a message from the incoming queue within the given deadline.
  - `+send(Payload, deadline : Long)`: Attempts to send a message with the provided payload within the specified deadline. If the payload corresponds to a data message, a credit is required. The deadline sets a time limit for waiting until a credit becomes available.
- **Flow Control**: Comprises methods for credit management, credit dispatching and configuration (such as adjusting the capacity or size of credit batches).
- **Event notification**: Most methods in this class are associated with events that need to be notified (such as read, write, and close), hence, event notification is spread across the methods in the categories above. The central method for subscribing to these events is `+poll(PollEntry) : int`[9]

4.3.7 LinkSocket

A **Link Socket** acts as a wrapper, exposing an API for interacting with the link while hiding core methods to prevent unintended behavior. The `Link` class includes several methods essential to the middleware's core functionality, which should only be accessible to core components—particularly `Socket` and `LinkManager`. To prevent these internal methods from being exposed to `Socket` subclasses, a wrapper is required.

[9] The `+poll(PollEntry) : int` method belongs to the `Pollable` interface, which must be implemented by objects for event polling to be possible. For further details, refer to the Polling Mechanism and Wait Queues section.

Link Sockets also serve as a specialization point, allowing the creation of a custom API while enabling default methods to be overridden for link-specific behavior. For example, to ensure FIFO ordering within a link, the `send()` method could be overridden to add an ordering identifier to the payload before transmission.

Additionally, because links manage concurrency and sockets cannot close until all their links are closed, links operate independently of their owner socket to some extent. If no socket-level requirements prevent direct and concurrent link usage, Link Sockets can be exposed to client applications, allowing them to determine message routing. However, if exposing link-level operations, it is crucial to prevent unintended client access – for instance, preventing the sending of internal control messages that bypass the flow control mechanism. This can be achieved either by wrapping the Link Socket or by overriding the link socket methods to enforce restrictions.

Key Methods

The `LinkSocket` class exposes certain `Link` methods without adding additional operations. These include:

- `send(Payload, deadline)` and `receive(deadline) : SocketMsg`;
- `link(SocketIdentifier)` and `unlink(SocketIdentifier)`;
- Flow control configuration methods for managing link capacity and credit batching.

To customize the behavior of these methods or introduce new ones, a subclass of `LinkSocket` can be created.

4.3.8 MessageDispatcher

The **Message Dispatcher** is an interface designed to decouple sockets from the message dispatching layer while enabling the addition of new dispatching-related features.

Each socket is assigned a message dispatcher when started to enable message sending. However, since verifying the validity of messages is not the responsibility of the dispatcher, it is not directly accessible to `Socket` subclasses. This restriction prevents improper dispatching of core messages, which could lead to inconsistent states and unintended behavior.

With dispatching abstracted from sockets, modifications – such as switching the transport protocol used to route messages between nodes – are facilitated due to their transparency to sockets.

Currently, the `MessageDispatcher` corresponds to the middleware's `MessageManagementSystem`, which simply delegates messages to the transport layer (Exon library). However, as the dispatching process becomes more elaborate, additional dispatching handlers could be introduced, forming a chain of responsibility where messages pass through multiple processing stages. For example, to address node crashes in the future, message persistence in non-volatile memory would be critical. In this case, a persistence layer could be integrated as a dispatching handler before handing messages over to the transport layer.

Key Methods

A **Message Dispatcher** must implement the following methods:

- `dispatch(msg : SocketMsg)`: Immediately dispatches a message.
- `scheduleDispatch(msg, dispatchTime) : AtomicReference<SocketMsg>`: Schedules a message to be dispatched when `dispatchTime` is reached.

  Since scheduled transmissions may need to be canceled, this method returns an atomic reference to the message, allowing cancellation to be performed lock-free.

  Cancellation is achieved by swapping the message reference with `null`. If the reference is already `null`, the message has already been sent and cannot be canceled.

4.3.9 MessageManagementSystem

The **Message Management System (MMS)** serves as a bridge between sockets and the transport layer. Its key responsibilities include:

- Serializing and forwarding messages to the transport layer for transmission.
- Scheduling and executing dispatching events.
- Retrieving incoming messages from the transport layer and directing them to the appropriate sockets.
- Replying with an error message when the destination socket of an incoming message does not exist.

Message Processor

The MMS owns a dedicated thread, `MessageProcessor`, commonly referred to as the middleware's thread. This thread is responsible for:

- Listening to the transport library for incoming messages.
- Identifying the destination socket and executing the necessary delivery procedures.
- Handling scheduled dispatching events.

Retrying Linking Process

The MMS plays a crucial role in the linking process. When a link request is made, the destination socket may be temporarily unavailable or may not yet exist. In such cases, rescheduling the linking process is necessary. This is achieved by notifying the requester of the unavailability, allowing it to reschedule the dispatching of a new link request.

Since resending a link request after a certain interval requires a thread, rather than creating new threads for this sole purpose, the idea of having the MMS support scheduling arose. This approach allows leveraging the middleware's thread to efficiently manage scheduled dispatching events while keeping message dispatching decoupled from sockets (and their linking logic).

4.3.10 EOMiddleware

The `EOMiddleware` corresponds to **Exon's transport library**, which is responsible for ensuring exactly-once delivery between nodes. Once a message is submitted to Exon, the dispatching process is considered complete, leaving Exon to asynchronously handle message transmission.

To support mobility scenarios and provide meaningful identifiers for addressing nodes, the Exon library was modified (as will be discussed in the Support for Mobility Scenarios section). It was adapted to work with arbitrary identifiers and utilize a discovery service to find the transport addresses required for routing over the network. As a result:

1. When creating an instance of the library, a non-empty string must be provided as a unique identifier to represent the node within the topology.
2. For sending and receiving methods, a string node identifier is used to identify the receiver/sender instead of a combination of IP address and port.

3. Methods for discovery purposes were created. These methods were previously described as part of the `DiscoveryManager` interface when presenting the middleware instance's methods.

4.3.11 DiscoveryService

The **Discovery Service** is used for translation purposes. It serves to find the transport address associated with a node identifier and vice-versa.

4.4 Polling Mechanism and Wait Queues

While not a core component, a polling mechanism is essential for messaging middleware, enabling concurrent monitoring of multiple objects and ensuring responsive and efficient event handling.

Efficiently managing multiple objects is key to achieving high performance and scalability. Instead of dedicating a separate thread to each object, a well-designed polling mechanism allows a single thread to handle numerous concurrent events. This approach reduces resource consumption and latency while improving scalability.

The polling mechanism in this middleware was initially designed for internal use by sockets to manage their links, but also to allow efficient socket monitoring by the middleware users. By avoiding tight coupling with a specific entity, the mechanism remains flexible and can be extended to monitor other objects, making it adaptable to a wide range of use cases.

To achieve these goals, the polling mechanism was designed as a simplification and adaptation of two system calls of the Linux Kernel: `poll()`[Kerrisk et al. (2024a)] and `epoll()`[Kerrisk et al. (2024b)]. For occasional polling or cases where the monitored objects change frequently, users can utilize `Poller.poll()` [10], which functions similarly to the Linux kernel's `poll()`. When polling targets remain relatively stable, the more efficient approach is to use the mechanism inspired in `epoll()`. In this case, a `Poller` instance[11] is created using `Poller.create()`, objects of interest are registered via `add()`, and the `await()` method is invoked to wait for and retrieve available events.

This section explores the design decisions, architecture details and rationale behind the polling mechanism.

[10] The `Poller.poll()` method may also be addressed as: `poll()`'s adaptation; or, spontaneous poll

[11] A `Poller` instance may also be addressed as: poller; or `epoll()` adaptation.

4.4.1 Design Rationale

Before implementing the polling mechanism of the Linux Kernel, several attempts were made to develop an efficient and scalable solution.

The initial approach aimed to design an original algorithm. However, multiple issues rendered these attempts less than ideal:

- **Inflexibility** due to the avoidance of callbacks, making event notification analogous to solely waking up threads, without a direct way to process specific events.
- **Inefficiency** when a thread repeatedly monitored the same set of objects.
- **Tight coupling to links/sockets**, making it difficult to extend to other objects.

Struggling to find a decent solution, I realized that exploring existing polling mechanisms was necessary. This led to an investigation of well-established solutions: `select()`, `poll()`, and `epoll()`. The `epoll()` mechanism was particularly relevant due its wide use in messaging middleware solutions such as ZeroMQ, nanomsg, and NNG. Even Java's `Selector` relies on `epoll()`. Given their proven utility, integrating such mechanisms into this middleware seemed promising. However, unlike these middleware solutions that monitor physical sockets, this middleware operates over a single UDP socket – managed by the Exon library – where all middleware sockets are virtual and multiplexed over the same socket.

A potential alternative involved using Java's `Selector` with dummy files (e.g., pipes) to trigger events for each pollable object. However, since Linux polling mechanisms rely on system calls, this approach would introduce expensive user-to-kernel transitions, negating the performance benefits of a purely user-level solution.

Recognizing these drawbacks, and due to the lack of comprehensive explanations of these algorithms, I analyzed the Linux Kernel's source code(Torvalds and the Linux community, 2025) of `select()`, `poll()`, and `epoll()`. This study allowed me to understand how experts in the field approached this problem, allowing me to develop a Java-based adaptation that operates entirely in user space, thus contributing to the preservation of the benefits of multiplexing all the traffic over a single physical socket.

This research led to a simplified adaptation of `poll()` and `epoll()` while revisiting initially discarded ideas, such as the use of callbacks. Callback registration, initially perceived as risky, proved valuable when limited to internal middleware components. With well-tested internal components, concerns about unwanted behavior were mitigated. Additionally, maintaining registered interests until explicitly removed, addressed initial inefficiency concerns related to removing interest shortly after adding it. Furthermore, in-

corporating level-triggered and edge-triggered modes, exclusive interest registration, and temporary interest suspension (`EPOLLONESHOT`) made this approach highly desirable for flexibility and efficiency purposes.

4.4.2 Polling Mechanism Overview

Before describing the polling system in detail, an overview of the key components can ease understanding. These components are shown in figure 23.

The action of polling is executed over an object that we want to monitor. Such objects are referred to as **Pollables**. The primary purpose of this mechanism is to enable monitoring of these objects – i.e., waiting for events – rather than merely retrieving currently available events. To achieve this, each *Pollable* must have a **Wait Queue**.

As the name suggests, a *Wait Queue* is a queue for waiting. When an entity – whether a thread or another object – wants to monitor the events of a *Pollable* object, it is inserted into this queue, where it will wait for event notifications. Each waiter is represented by a *Wait Queue Entry*, which contains a **Wait Queue Function** – a wake-up function used to notify events and execute any related custom operations – as well as a private object that is typically required by the *Wait Queue Function* to provide context for these operations. An example of a *Wait Queue Function* could be signaling a thread to wake up and process a received event if an event of interest was notified, with the private object managing the thread's parking state while it waits for the event.

A *Pollable* object usually hides its *Wait Queue* to prevent unauthorized access. To enable instant event polling and queuing, a `poll(PollEntry)` method is exposed. The **Poll Entry** provided as an argument specifies the events of interest (*key*), a **Poll Queuing Function**, and a private object to provide context for this function. Upon receiving a *Poll Queuing Function*, if the *Pollable*'s state currently allows queuing, the function is invoked, allowing it to queue a waiter while also executing any required custom procedure associated with the queuing operation.

Finally, the component responsible for enabling concurrent monitoring of multiple *Pollable* objects is the **Poller**. It utilizes all the components described above: a *Wait Queue* to manage its own waiters, a *Poll Queuing Function* to register itself as a waiter in each *Pollable*'s queue, and a *Wait Queue Function* to define how events from *Pollables* are processed. Its processing of the event notifications ultimately corresponds to marking the *Pollable* that notified the event as ready and signaling the *Poller*'s waiters about the availability of that *Pollable*.

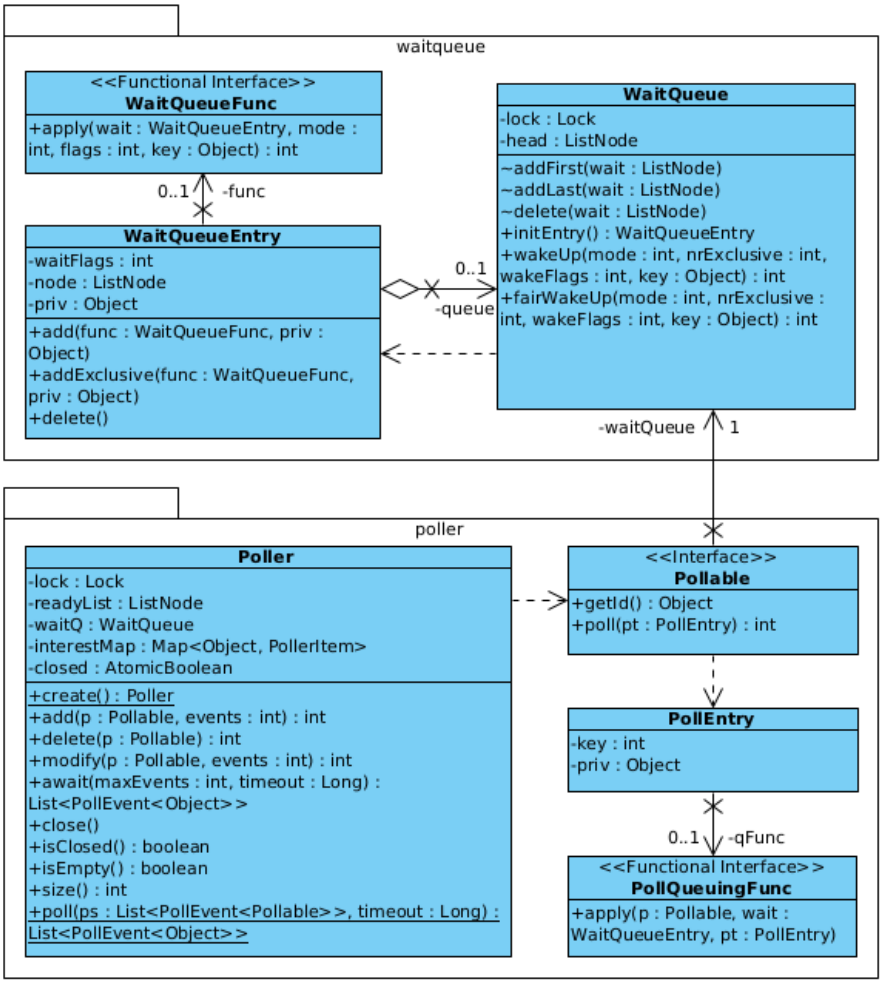


Figure 23: Polling Mechanism Class Diagram

4.4.3 Wait Queue

Wait queues are at the core of event polling and handling. Each object that must notify events, must have a wait queue. These queues manage *wait queue entries*, each corresponding to an entity interested in events of the queue's owner, which we call "waiters" for simplification.

The adaptation of Linux Kernel's *waiting system*, was designed to prevent unauthorized objects from manipulating *wait queues*. Rather than following the kernel's approach that allows new waiters to directly access the wait queue (referred to as `wait_queue_head`), this adaptation provides a **wait queue entry**, initialized to have a private reference to the queue. This design ensures waiters can only manage their own *wait queue entry* and enables the queue owner to reject a new waiter simply by not providing an initialized entry.

Entries Lifecycle

The lifecycle of a waiter involves the following steps:

1. **Wait Entry Initialization:** The owner of the *wait queue* creates a *wait queue entry*, which remains associated with the queue throughout its lifetime. This entry is then given to the waiter that wishes to register itself in the queue.

2. **Wait Entry Queuing:** The waiter, whenever desired, can add itself to the queue using either the `add()` or `addExclusive()` method. These methods accept a notification function (`WaitQueueFunc`[12]) and a private object, which the function uses for custom event handling. For instance, the private object might specify the events of interest, such as the read event (`POLLIN`), along with the necessary state to wake up a thread. When the event is notified, the function is invoked with the private object as an argument. If the notified event matches `POLLIN`, the function can then wake up the thread, allowing it to receive and process the message.

3. **Wait Entry Notification:** The waiter is informed of available events via the `WaitQueueFunc`.

4. **Wait Entry Dequeuing:** When the waiter is no longer interested in the events, it must disassociate itself through the `delete()` method of the entry, rendering the entry invalid for further use.

[12] A wait queue function signature is the following: `<function_name>(WaitQueueEntry, mode : int, wakeFlags : int, key : Object) : int`.

Types of entries

This adaptation supports two types of waiters: non-exclusive and exclusive, distinguished by the presence of the `EXCLUSIVE` *wait flag*. A **non-exclusive entry** refers to a waiter that does not contest event notifications, allowing them to be shared with other entries. It is registered using `add()` which inserts it at the head of the queue. On the other hand, an **exclusive entry** refers to a waiter that wants exclusivity regarding notifications. This kinds of entry is registered using `addExclusive()`, which inserts the entry at the tail of the queue to ensure exclusive entries registered before are prioritized.

Queue's Wake-up Methods

To better understand how exclusivity works, delving into the specifics of the queue's notification (wake-up) methods is needed.

We will start by briefly describing the arguments shared by the notification methods:

- `nrExclusive`: Specifies the maximum number of exclusive entries that can be woken up. However, when the value is zero or less, all entries are woken up[13].
- `key`: Indicates available events, usually in the form of an (integer) event mask. For broader use, the key is allowed to be any type of object.
- `mode` and `wakeFlags`: Do not serve a specific purpose in this adaptation. Their meaning can be determined by the owner of the queue, as done to implement the socket's *Notify-If-None mechanism*.

The first notification method, `wakeUp()`, adapted from the kernel, iterates over the entries, starting from the head. For each entry, it invokes the corresponding `WaitQueueFunc` with `mode`, `wakeFlags`, `key`, and the entry's private object as arguments. The iteration stops when the specified number of exclusive entries have been successfully woken up or when the end of the queue is reached. Essentially, this method wakes up all non-exclusive entries (since they are inserted at the head) and a number of exclusive entries up to the specific value (`nrExclusive`)[14].

[13] Global wake-up calls are typically executed when the object of interest is closed. The global "close" notifications are commonly triggered by a combination of a hang-up flag (`POLLHUP`) – indicating object closure or that a peer hung up – and a free flag (`POLLFREE`) – indicating the object wishes to be released, requiring waiters to remove themselves from the queue to enable a graceful closure.

[14] To ensure true exclusivity, when non-exclusive and exclusive entries must coexist, the non-exclusive entries must not "consume" events. This ensures the events are not depleted before exclusive entries can process them. For instance, if a single read event is triggered and the non-exclusive entry receives the available message, the exclusive entry will find no message left when it is woken up.

The second notification method, `fairWakeUp()`, is designed as a fair version of `wakeUp()`. It addresses scenarios involving multiple exclusive waiters, where fair distribution of events is essential. The original wake-up method, derived from the kernel, does not alter the position of exclusive entries after successfully waking them. As a result, non-transient[15] exclusive entries retain their relative position to the beginning of exclusive entries in the queue. This can lead to event monopolization, particularly in the most common scenario where only one exclusive entry is woken up. In such cases, only the first exclusive entry gains access to events unless the wake-up attempt fails[16]. To resolve this issue, `fairWakeUp()` moves non-transient exclusive entries to the tail of the queue upon successful wake-up, ensuring that all exclusive entries have an equal chance of being notified.

4.4.4 Polling Concepts

The architecture of the polling mechanism relies on several core concepts that are shared between the `poll()` and `epoll()` adaptations. In this section, we will explore these key concepts, which form the backbone of the polling system's architecture.

Poll Flags

Poll flags are essential for the polling mechanism. *Event-related flags* signal specific conditions on *pollable* objects, while *mode-related flags* determine the behavior of the polling operation. Understanding the role and functionality of these flags is crucial for effectively managing and interpreting the events during the polling process.

The **event-related flags** are used to manifest the events of interest – when using poller instances or `Poller.poll()` – and by *pollable* objects to indicate their current status, such as readiness for reading and writing, or the presence of an error. The available event flags are:

- **`POLLIN`:** Associated with read operations.
- **`POLLOUT`:** Associated with write operations.
- **`POLLERR`:** Associated with error or exception conditions. [17]

[15] A non-transient entry is one whose wake-up function does not immediately dequeue it after successful notification. For example, a poller continuously monitors events of the object of interest, therefore, it must not remove itself after being notified.

[16] The *wait queue function* associated with each entry for event notification can return 0 (zero) to indicate that the waiter was not woken up. For instance, if the waiter is a thread still handling another event, it may not be in a waiting state, allowing the wake-up method to proceed to the next waiter.

[17] The polling mechanism always reports this event regardless of whether it was specified as an event of interest or not.

- **`POLLHUP`:** Associated with hang up. Used to inform when a socket closes. Could also be used to signal that a peer closed the connection. Depending on the *pollable*'s semantics, this event does not necessarily mean that operations like reading are impossible. There may still be data available to read after the *pollable* is closed. [18]

- **`POLLFREE`:** Special event flag used by *pollable* objects to notify their waiters of the will to be released[19]. Adding `POLLFREE` to the events of interest mask has no effect since it is not returned by a `poll()` method.

The **mode-related flags** – declared when registering events of interest in `Poller` instances – define the polling behavior:

- **`POLLET`:** When set, events are registered in edge-triggered mode rather than level-triggered. Unlike level-triggered mode, where the `Poller` continuously reports a pollable object as available as long as the operation remains possible, edge-triggered mode only notifies waiters when the poller's perception of the pollable object transitions from "no events" to "events available."

  For example, consider the registration of interest in read events of a socket in edge-triggered mode. If the socket becomes available for read events, the poller will notify a waiter of that availability and then immediately mark the socket as having no available events, even if it remains readable. The notified waiter must then handle all available data[20], since the poller will only notify the availability of that socket again when a new read event occurs (i.e., when new data arrives).

- **`POLLONESHOT`:** Disarms interest in a *pollable* after an event notification. All event-related bits are cleared from the events of interest mask, which effectively halts any further notifications from the `Poller` instance regarding the *pollable*. To resume receiving those notifications, the events of interest must be rearmed through the `modify()` method.

  This flag is useful for gaining precise control over event notifications, especially in conjunction with `POLLET`. In edge-triggered mode, when a waiter is notified, events of the same *pollable* are all "cleared" from the `Poller` instance, but this does not prevent the *pollable* from triggering new notifications.This might result in a new waiter being woken up by the `Poller` to handle the same

[18] Like `POLLERR`, this event is always reported regardless of interest.

[19] When a *pollable* is closed, the `POLLFREE` and `POLLHUP` flags are reported to all waiters, signaling them to remove their *wait queue entry* as no more events will be reported.

[20] In edge-triggered mode, when a waiter is notified, it should process all available data (or events). This can be done using non-blocking methods until the operation is no longer available (i.e., would block). This ensures that no events are missed, as the `Poller` only notifies on state transitions.

events as the first waiter. By using `POLLONESHOT`, we can ensure that only one waiter of the poller is notified to handle the pollable object's events.

- **`POLLEXCLUSIVE`:** Prevents the thundering herd problem by registering the `Poller` instance as an *exclusive* waiter of the *pollable* of interest. This flag is relevant only when used with edge-triggered events (`POLLET`) and is incompatible with `POLLONESHOT`. It ensures that only one waiter is notified of an event, reducing the risk of resource contention.

Poll Entry and Poll Queuing Function

The **Poll Entry** is a key component in the polling mechanism, allowing the caller of a pollable's `poll(PollEntry)` method to register events of interest. It consists of three attributes: an (integer) bitmask with events of interest, a poll queuing function and a private object.

The **Poll Queuing Function** (PollQueuingFunc)[21] registers *wait queue entry* in the *pollable*'s *wait queue* if possible, but might also perform specific complementary actions. When the `poll()` method determines that queuing is allowed, it calls this function with an initialized *wait queue entry* and its private object. The private object provides context for the queuing function, ensuring proper registration and execution of caller-specific actions.

Regardless of queuing a waiter, the *pollable* returns a bitmask with available events.

Note: Do not confuse poll queuing functions with wait queue functions. A poll queuing function queues an entry in the wait queue, while the wait queue function is called to notify events.

Pollable

The polling mechanism operates on objects of interest, called *pollables*. To be eligible for polling, an object must implement the `Pollable` interface, which defines two key methods:

1. `getId() : Object`: This method retrieves the unique identifier of the *pollable*. Each *pollable* must have a locally unique identifier to avoid conflicts when registering interest in a `Poller` instance, which uses this identifier as a key for internal mapping.

2. `poll(PollEntry pt) : int`: This method retrieves the *pollable*'s currently available events as a bitmask. If a `PollQueuingFunc` is provided, it also queues an entry for the caller.

Each *pollable* manages a single wait queue, and the `poll()` method handles queuing waiters. Queuing is only possible if the `PollEntry` includes a `PollQueuingFunc` and its associated private object.

[21] The signature of a poll queuing function is `<function_name>(Pollable, WaitQueueEntry, PollEntry)`.

However, queuing is not guaranteed, as the pollable may disallow queuing under certain conditions, such as when it is closed. In such cases, the queuing function is invoked with an *uninitialized wait queue entry* to signal that queuing is not allowed.

4.4.5 Adaptation of poll()

We will now delve into the algorithm of the `Poller.poll()` method – the adaptation of the kernel's `poll()` system call – which is designed to monitor a set of objects of interest and detect their availability for I/O operations.

The method takes three arguments: a list containing the pollable objects and the corresponding events of interest, a timeout value (optional) and a maximum number of pollable objects, with available events, to be returned.

- **Initial Iteration and Queuing:**
  - The `Poller.poll()` method begins by iterating over the list of objects of interest. During this initial iteration, it registers itself in the *wait queue* of each object while checking whether any events are immediately available.
  - If an event is detected during this first iteration, the method ceases further queuing and limits itself to polling events from the remaining objects. Queuing is unnecessary since the method will return the detected event(s) at the end of the iteration.
  - If the specified maximum number of objects with available events is reached, the iteration is stopped, the queued *wait queue entries* are deleted, and the events are returned.
  - If no events are found during this first iteration, the method will have registered itself as a waiter on each object. This ensures that it will be notified of any events of interest during any following idle-wait periods.
- **Idle-Wait and Event Detection**:
  - Once the thread is queued as a waiter for all objects, it enters an idle-wait state. In this state, the client thread remains inactive until an event is signaled, the specified timeout period elapses or the thread is interrupted.
  - If the timeout is reached or the thread is interrupted, the method proceeds to clean up by removing the entries from all wait queues. It then returns, indicating either a timeout or interruption, as appropriate.

- **Re-Polling After Wake-Up**:
  - If the thread is woken up by an event, the method performs another iteration over the objects of interest. This time, however, it focuses solely on fetching available events, bypassing the queuing step.
  - During this traversal, the method collects any events that have become available, stopping when the number of maximum objects with available events is achieved or when all the objects have been polled. If any events are found, the *wait queue entries* are removed and the events are returned to the user.
- **Repeating the Process if Necessary**:
  - If, after being woken up, no events are found during the re-polling iteration, the process repeats. The thread re-enters the idle-wait state, waiting once more for an event, a timeout, or an interrupt signal.
  - This loop continues until either events are detected, leading to a return, or the operation is terminated by a timeout or signal.

4.4.6 Adaptation of epoll()

Unlike `poll()`, the Linux Kernel's `epoll()` is implemented as a set of system calls, designed for high-performance event monitoring with minimal overhead. This performance is achieved through active monitoring. In contrast to `poll()`, which requires iterating over the whole list of monitored objects each time a check for events is performed, `epoll()` uses an event-driven approach. Once an object is registered, the `epoll()` instance monitors it for events, allowing immediate responses to changes without requiring a scan over the entire list, which significantly reduces the overhead when dealing with a large number of monitored objects.

The adaptation of `epoll()` to a user-level environment, the `Poller` instance, follows a similar design, though it does not encompass all the features provided by the `epoll()` instances, like the ability to nest `epoll()` instances. The `Poller` implementation comprises six key methods: `create()`, `add()`, `modify()`, `delete()`, `wait()`, and `close()`.

Before delving into the algorithm's details, it is essential to understand three key collections about the poller instances: the *interest list*, the *ready list* and the *waiters queue*. The *interest list* comprises pollable objects that the poller instance monitors. The *ready list* contains events that have been notified and are

thus considered "ready" for handling. Finally, the poller instance has a *waiters queue*, where the several client threads waiting for events are queued as exclusive waiters to prevent interference between them.

With this understanding, we can now explore how this adaptation of `epoll()` operates.

- **`create() : Poller`**

  Initializes a new instance of the event poller, setting up the internal structures necessary for monitoring the specified objects.

- **`add(p : Pollable, events : int) : int`**

  Registers a new object of interest along with the corresponding event mask that specifies which events should be monitored.

  The method follows these steps:

  1. Validates the object of interest and ensures that the provided event mask is valid. Certain flag combinations, like exclusivity with "one-shot", are checked for compatibility.
  2. Automatically includes the error (`POLLERR`) and hang-up (`POLLHUP`) flags in the event mask to ensure these events are always reported.
  3. Adds the object of interest to the poller's *interest list* after successful validation.
  4. Polls on the object to fetch available events and to register itself as a waiter to be notified of future events.
     - The poller instance is added as an exclusive or non-exclusive waiter based on the presence of the exclusivity flag on the event mask.
     - The *wait queue function* registered with the *wait queue entry* enables the monitored objects to add themselves to the poller's *ready list* when reporting the availability of events being monitored. A waiter of the poller instance is also woken-up to retrieve the available events.
  5. If any events are available during this registration process, the object is added to the *ready list*, and a waiter is notified.

- **`modify(p : Pollable, events : int) : int`**

  This method is used to modify the event mask of a monitored object of interest or to re-arm it when the "one-shot" flag is used. It immediately polls on the object to check for available events. If events are found, the poller instance notifies a waiter to retrieve them.

When modifying the events of interest, these must also verified, as neither the new event mask nor the current mask can have the exclusivity flag set, as doing so results in an error. Exclusivity notification relies on the assumption that when an event is notified it will be properly handled. Attempting to change the events of interest violates this assumption.

- `delete(p : Pollable) : int`

  Deletes an object of interest from the poller instance's *interest list*, effectively stopping its monitoring. When an object is registered with the exclusivity flag, careful removal is necessary to avoid dismissing events that could have notified another exclusive waiter.

- `await(maxEvents : int, timeout : Long) : List<PollEvent<Object>>`

  This method waits for any events from the monitored objects to become available. It returns up to a specified maximum number (`maxEvents`) of objects with available events. If no events are immediately available, the method enters a loop, performing idle-waiting until: the thread is woken up and events are available to be returned (ready list is not empty); the specified timeout elapsed; or an interrupt signal was received.

- `close() : void`

  Invoking this method is similar to calling `delete()` on all objects in the *interest list*. Additionally, it wakes up every waiter of the poller instance, allowing them to perceive the closure of the instance and return graciously.

4.4.7 Comparison with Linux Kernel's Implementation

The adaptations of `poll()` and `epoll()` share the goal of mimicking these efficient polling mechanisms while solely operating in a user-level environment and avoiding the inherent complexities of the kernel-level.

These user-level implementations are very straightforward and simple, eliminating kernel-level overhead, avoiding system calls, low-level resource management, reference counting, efficient memory allocations, cache allocations, memory barriers, power management functionalities (`EPOLLWAKEUP`), file descriptor management (mapping, ownership verification, etc), scheduling and thread management, nesting of poller instances, among other things.

4.5 Messages: Structure and Types

4.5.1 Message Structure

The initial idea for the middleware was to have nodes serve as containers for services, with services being called sockets. Since nodes form the foundation of the middleware, it initially made sense to tailor the message format to them. However, this would unnecessarily complicate the message domain. Node-to-node communication can still be achieved through a dedicated service, so the base message format is designed for service-to-service communication.

As a result, the message structure remains straightforward, consisting of two parts: header and payload.

Header

The header of a message is comprised by a fixed set of fields that are essential for communication between two sockets:

- **Version (2 bytes):** Indicates the protocol version, allowing interoperability between different versions of the middleware and detecting compatibility issues. Currently unused in this prototype phase.
- **Source Node Identifier (Omitted)**[22]**:** Identifies the node that sent the message.
- **Source Tag Identifier:** Identifies the socket that sent the message. A tag identifier functions as a "service tag."
- **Destination Node Identifier (Omitted):** Identifies the node that should receive the message.
- **Destination Tag Identifier:** Identifies the socket that should receive the message.
- **Message Type (1 byte):** Specifies the type of message, which is crucial for processing.
  To keep the format compact, only 1 byte is allocated, allowing 256 message types. These are divided into:
  - **Core messages:** Reserved for middleware functionality.

[22] Node identifiers are omitted because the Exon library already includes them.

  - **(Custom) Control Messages:** Used for messaging protocols. Messaging protocols must not assign control messages to reserved core message types, as they will be rejected for transmission. Since communication between sockets is only allowed after protocol compatibility verification, the entire custom space can be used without conflict between different messaging protocols.

- **Incarnation Identifier (4 bytes):** Each message carries a link incarnation identifier (clockId) that determines its presentness. Since two sockets may establish and terminate links multiple times, this identifier ensures that messages are correctly associated with the intended link instance.

Node and tag identifiers have variable lengths, providing flexibility to the user. However, shorter identifiers are recommended to minimize overhead, as they are included in every exchanged message.

For the future, to support internal node services (e.g. for statistics), tag identifiers prefixed with a special character[23] should be restricted from user registration.

Payload

The payload contains the actual content of the message. Any necessary header extensions can also be included here.

4.5.2 Message Types

The middleware defines the following core message types:

- `ERROR (0x01)`**:** Error message.
- `LINK (0x02)`**:** Link request. Initiates the establishment of a link between two sockets.
- `LINKREPLY (0x03)`**:** Link reply. Carries a socket's response to a link request.
- `UNLINK (0x04)`**:** Link termination. Initiates the process of terminating a link between two sockets.
- `FLOW (0x05)`**:** Link flow control message. Once a link is established, this message type manages the flow of `DATA` messages by granting or revoking transmission credits from the remote socket.
- `DATA (0x06)`**:** Data message. Represents application data exchanged between sockets. Flow control is applied to prevent senders from overwhelming receivers.

[23] Similar to MQTT, where topics prefixed with "$" are reserved for internal use.

Error Messages

Currently, there is only one type of error message: **Socket Not Found**.

This error occurs when a received message's destination tag does not match any socket registered on the receiving node. The error message is constructed as if the nonexistent socket was replying, meaning that the "source tag" corresponds to the tag of the missing socket.

This message informs the sender that the destination socket is unavailable. However, since the socket may be created in the future, the sender can retry the link request after a short delay to avoid overwhelming the receiving node.

4.5.3 Message Serialization Format

Messages are generated using **Protocol Buffers (Protobuf)**. Protobuf (Google, 2025) facilitates both writing and reading of structured data, while ensuring efficient serialization/deserialization. Furthermore, Protobuf's cross-language compatibility enhances interoperability, potentiating the middleware's use across different systems and programming languages.

4.6 Link Management Protocol

Enabling socket extensibility – specifically the creation of new socket types – introduces a challenge: sockets cannot spontaneously begin exchanging messages. With multiple socket types comes a variety of messaging protocols, which may be incompatible. As a result, messages cannot be sent arbitrarily because the recipient may not be able to interpret them. Furthermore, the receiving socket may not want to communicate with a sender due specific conditions unrelated to compatibility.

To address these issues, a mechanism is needed to ensure that communication between sockets is both possible and mutually allowed. This mechanism, called the **Link Management Protocol**, requires sockets to engage in a link handshake before communication can occur. During this handshake, the involved sockets declare their intent to establish a link based on various factors, with protocol compatibility being the main consideration.

4.6.1 Assumptions

The **Link Management Protocol** – primarily responsible for establishing and closing links – was designed under the following assumptions:

1. **Exactly-Once Delivery**: The underlying transport layer guarantees that each message is delivered exactly once.

2. **Unordered Delivery**: The underlying transport layer does not provide any ordering guarantees for message delivery.

3. **Socket Extensibility**: The protocol must be designed to support future socket enhancements or additions without requiring significant changes.

4. **Safety**: The protocol must operate without risk of errors, crashes, or unwanted behaviors, ensuring correct operation under all circumstances and gracefully handling invalid or malicious messages.

4.6.2 Design Challenges

This section presents the two main challenges faced during the development of the protocol.

Symmetry vs Asymmetry

During the design phase, two protocol variants were considered: a symmetric protocol and an asymmetric protocol. The fundamental difference lies in how the decision to establish a link is reached.

The **symmetric protocol** defines the link establishment process as a function that takes the metadata from both sockets and returns the same result regardless of the order:

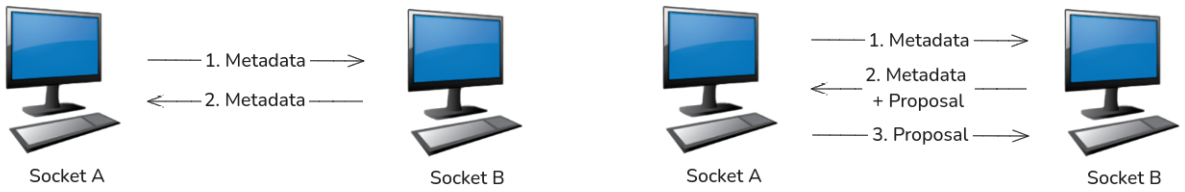
$$f(MD_{\text{A}}, MD_{\text{B}}) = f(MD_{\text{B}}, MD_{\text{A}})$$

Here, each socket can independently determine whether to establish a link simply by exchanging metadata. When Socket A receives Socket B's metadata, it arrives at the same conclusion that Socket B would upon receiving Socket A's metadata.

Conversely, the **asymmetric protocol** does not allow the conclusion to be derived solely from the remote socket's metadata. Instead, each socket uses the remote socket's metadata to generate a decision proposal. These proposals are then exchanged, and only after having both proposals can a socket compute the final decision regarding the link establishment.

Both approaches are valid, however, there are several factors that motivated the implementation of the asymmetric variant.

Extensibility: A two-phased process – first exchanging metadata, then exchanging proposals – naturally supports extensibility. This design enables a wider range of features (e.g., authentication/authorization, parameter negotiation, time-based verification) to be implemented with minimal or no changes



(a) Symmetric Linking Protocol

(b) Asymmetric Linking Protocol

to the core logic and message structure of the protocol. For instance, in a peer-to-peer scenario where mutual authentication is required, each socket sends the authentication data with the metadata, enabling the success of the operation to be perceived through the proposal's reply.

Safety - Memory Exhaustion Prevention: In any of the protocol variants, one socket inevitably establishes the link before the other. In the symmetric protocol, Socket B establishes the link immediately upon receiving Socket A's metadata, while Socket A still has to wait for Socket B's metadata. This early establishment can lead to a safety issue: once a link is perceived as established, a socket may begin sending data and control messages even though the recipient has not yet established the link on its side.

Given the assumption of unordered delivery, the data and control messages might arrive before the metadata. The receiving socket, lacking the sender's metadata, which includes the socket type, would be unable to process them. Since our middleware guarantees exactly-once delivery, such messages must be queued. While the queuing of data messages is managed by the flow control that respects link capacity[24], control messages – not overseen by the flow control mechanism[25] – could accumulate and eventually lead to memory exhaustion. The asymmetric protocol mitigates this risk by ensuring that data and control messages are only transmitted after both the metadata and proposal have been exchanged, so that the recipient can immediately handle incoming messages and discard any invalid or malicious ones.

Given the infrequency of link establishment processes, the extra overhead of one or two[26] additional messages in the asymmetric protocol is minimal, making it a small drawback when compared to the symmetric protocol. Consequently, when considering the advantages of the asymmetric variant, its use seems justified.

[24] The sockets include their link capacity as metadata.

[25] Control messages are not limited as safety measure against deadlock scenarios.

[26] When both sockets initiate the link establishment process simultaneously, the asymmetric protocol requires four messages: two metadata messages + two proposal messages.

Out-of-order Message Delivery

The **Exon library** was chosen as the underlying transport protocol in the early stages of development. Since one of its key characteristics is lack of ordering guarantees, the link management protocol was devised under this no-ordering assumption.

To address the challenges of unordered delivery, two measures were implemented: the use of incarnation identifiers to distinguish conversations and the enforcement of FIFO order of the link-management messages inside a conversation.

Incarnation Identifiers: A link – identified by combining the identifiers of the involved sockets – can be established and terminated repeatedly over time. Since each link represents an individual conversation, closing the link signifies the end of that conversation. The combination of these facts with unordered message delivery implies that multiple conversations between the same two sockets may overlap. If every conversation is solely identified by the identifiers of the participating sockets, distinguishing messages from different conversations would not be possible.

The solution was to introduce incarnation identifiers. Each socket maintains a counter that serves as a unique conversation identifier. When a link is created, the current counter value is assigned as the link's incarnation identifier, after which the counter is incremented to ensure uniqueness. Thus, a conversation is uniquely identified by the combination of the link identifier and the pair of incarnation identifiers (one from each socket). By having messages carry the local incarnation identifier, the system can ignore messages from previously terminated conversations, accept messages associated with the ongoing conversation and detect the beginning of a new conversation[27].

Enforcing FIFO Order of Link-Management Messages Within a Conversation: While incarnation identifiers prevent corruption and ensure proper ordering across separate conversations, enforcing *FIFO order* within a single conversation remains a challenge. Message ordering can simplify protocol development, however, it may also increase latency due to head-of-line blocking when ordering is not necessary. With that in mind, the chosen solution confines order enforcement to messages related to the link's lifecycle, while leaving ordering of data and control messages to the specialized logic of each socket if required.

To ensure order, we do not rely on the typical method of using sequential message identifiers. Instead, due to the minimal set of messages needed to establish and close a link, and the inherent causal relationships between them, a state machine is enough to implement *FIFO order*, while avoiding the need for every

[27] A message with an incarnation identifier related to a newer conversation is only received when the remote socket is trying to establish a new link, indicating that the current conversation has already been terminated on the remote socket's end.

message of a conversation to carry 4 additional bytes as an order identifier. With only 3 message types to manage the lifecycle of links – `LINK`, `LINKREPLY` and `UNLINK` –, there are only two scenarios where out-of-order delivery is possible. One of them is solved by temporarily storing the out-of-order message and processing it after the preceding message arrives, while in the second scenario, inferring the content of the missing message is possible, enabling the out-of-order message to be processed immediately. Each of these scenarios will be explained in more detail below.

4.6.3 Link States

The **Link Management Protocol** was designed around a state machine that represents the socket's perception of the link state. This machine consists of five states:

- **LINKING**: The link establishment process starts when the `link(SocketIdentifier)` method is called. This method not only sends a link request but also creates an instance of `Link` in this `LINKING` state. The link remains in this state until it gathers the necessary information – namely, the remote socket's metadata and proposal – to determine whether to establish or close the link.
- **CANCELLING**: If the `unlink(SocketIdentifier)` method is called while in the `LINKING` state (i.e., during the linking process), the link transitions to the `CANCELLING` state. In this state, the socket shifts its focus from establishing the link to attempting to close it. Depending on the information exchanged before entering this state, the link may either transition directly to the `CLOSED` state or to the `UNLINKING` state. The latter occurs when the link has already been established at the remote socket's side, while the local socket transitioned to the `CANCELLING` state before receiving the proposal that would have established the link. Since the link was successfully established remotely, the unlinking process must be performed to properly close the link.
- **ESTABLISHED**: A link transitions to the `ESTABLISHED` state from `LINKING` when both the local and remote sockets exchange positive proposals after receiving each other's metadata.
- **UNLINKING**: The link enters the `UNLINKING` state when an unlinking process begins, either due to an `unlink(SocketIdentifier)` invocation or upon receiving an `UNLINK` message. The link remains in this state until all conditions for closing it are met.
- **CLOSED**: The link transitions to the `CLOSED` state when it is effectively closed.

4.6.4 Establishing a link

Having decided on an asymmetric protocol, for a link to be established, both sockets must agree on linking with each other. In short, the link establish process consists in the sockets sharing their metadata, generating proposals, and then exchanging them. A proposal is generated based primarily on the received metadata, but may be influenced by internal socket conditions (e.g. permission to accept incoming link requests).

Linking-related Messages

The linking process employs two types of messages: `LINK` and `LINKREPLY`. The `LINK` message – referred to as "link establishment request" or "link request" – initiates the linking process[28] and carries the socket's metadata. The `LINKREPLY` message is mainly used to carry the socket's decision proposal, but it can also include socket's metadata if it has not yet been sent.

Metadata

The main goal of the link establishment process is to ensure compatibility, which can only be determined through the exchange of metadata. Currently, the only required metadata is the socket type (i.e. its protocol identifier) and the receiver's capacity in the form of credits. However, future extensions – be it for internal or specialized features, such as authorization/authentication procedures or custom handshake logic – might require their own metadata. By enabling the embedding of specialized metadata in the middleware's linking process, the number of messages exchanged can be reduced in comparison to requiring additional handshakes after the link establishment.

When discussing `LINKREPLY` messages, references to the presence or absence of metadata will be made. Determining whether the received `LINKREPLY` contains metadata is essential to the protocol's algorithm, as it provides a cue on whether an incoming `LINK` message is in transit and has been overtaken by the `LINKREPLY`. It is important to notice that the metadata mentioned in these statements corresponds to critical information (such as the socket type) required by sockets to formulate a proposal. This should not be confused with additional metadata that may be included for purposes like parameter negotiation.

[28] "Linking process" is an alias for "link establishment process".

Types of replies

We will now discuss the three existing types of replies – Positive, Non-Fatal and Fatal – and their impacts. We are adopting the name of "reply" instead of proposal, since in some situations, a certain proposal actually corresponds to the actual fate of the link.

A **Positive reply** indicates acceptance, meaning its sender is willing to establish the link if the remote socket also replies positively.

A **(Negative) Non-Fatal reply** indicates temporary rejection. This type of reply is sent when the socket cannot establish the link at the present moment, but may be able to do so in the future. Upon reception, the receiving socket may schedule a re-attempt to establish the link. The non-fatal replies that currently exist are:

- `TMP_NAVAIL`: Generic reply code implying temporary link establishment unavailability. Currently, this code is only used when the limit of established links is met, meaning a new link cannot be established until one of the established links is closed. In the future, if custom extensions of the linking behavior are allowed, the socket logic should use this reply code to request the rescheduling of the link establishment process if necessary.

- `LINK_EXISTS`: Indicates temporary unavailability to link due to a link between the two sockets still existing in the remote socket. When linking/unlinking, there is always one socket that perceives the process as finished first, i.e., it establishes/closes the socket before the remote socket. This fact along with the lack-of-order message delivery, implies that a `LINK` message, regarding a new linking process, might arrive at the destination before the unlinking process ends. In this situation, the socket finishing the unlinking process, noticing that the message corresponds to a new conversation, sends a reply with this code to trigger the initiating socket to re-schedule the link establishment process, therefore providing time for the unlinking process to finish.

Lastly, a **(Negative) Fatal reply** indicates fatal rejection, meaning that its reception implies the link must be closed. This type contemplates four reply codes:

- `INCOMPATIBLE`: Used when sockets are incompatible. If the sockets' protocols are incompatible, they cannot communicate with each other, hence why this is a fatal reply that results in the closure of the link without triggering rescheduling.

- `INCOMING_NOT_ALLOWED`: As mentioned earlier, sockets can be configured to block incoming link requests. With this option set, any link request received is fatally rejected using this code.

- `CANCELED`: Used when cancelling an ongoing linking process. If `unlink()` is called before a reply has been sent, a reply with this code is sent immediately to close the link.
- `CLOSED`: Sent to reject link establishment requests when the socket is in the process of closing or has just closed.

Linking Algorithm

Having gained an understand of the types of messages, metadata, and types of replies – as well as their impact – we are now ready to explore the specifics of the linking algorithm, as depicted in the state machine diagram (25).

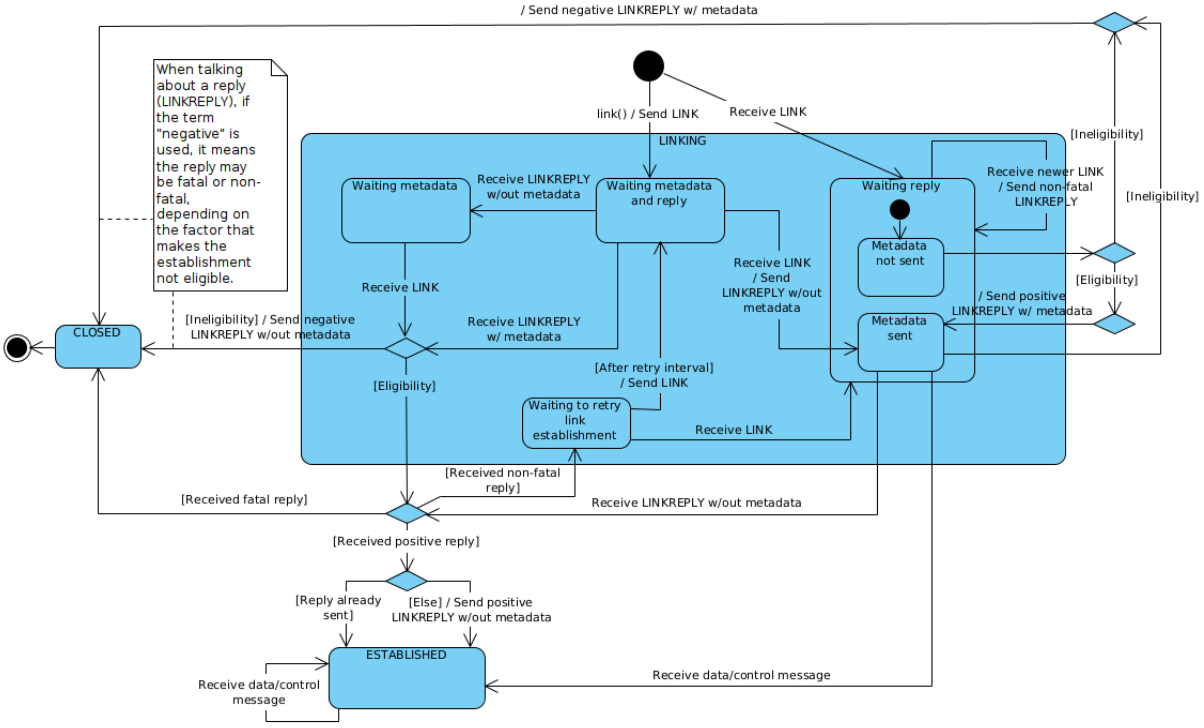


Figure 25: Linking Process State Machine Diagram

A link always begins in the `LINKING` state, triggered either by invoking `link(SocketIdentifier)` or by receiving a link request. In the former case, the linking process is initiated by sending a `LINK` message.

The `LINKING` state consists of four sub-states:

1. **Waiting for metadata and reply:** When a link is created through the invocation of the `link()` method, it immediately transitions to this sub-state. As the name suggests, the link is awaiting the peer's metadata and proposal (reply).

2. **Waiting for metadata:** If a `LINKREPLY`, without metadata, is received after initiating the linking process, the link transitions to this sub-state. The absence of metadata in the reply suggests that a `LINK` message is still in transit and must be received before a decision can be made. Once the missing `LINK` message arrives, the socket possesses both the remote socket's metadata and reply, allowing it to determine whether to establish the link, close it, or reschedule the linking attempt. A reply is sent back only if the received reply is positive, as a negative proposal cannot be overridden by a positive one – both sockets must agree to establish the link. Metadata is not included with the reply since it was already sent in the initial link request.

3. **Waiting for reply:** The link transitions to this sub-state upon receiving a link request in one of two scenarios: The link did not yet exist when the request was received; Or, the local socket has already initiated a linking process.

   In both cases, the received metadata is used to formulate a proposal rather than a final decision, as no reply has been received yet. The local proposal is then sent in a `LINKREPLY` message, which may or may not contain metadata, depending on whether it was sent earlier or not.

   If the link is deemed not eligible for establishment – due to metadata incompatibility or an internal reason – it is immediately closed upon sending a negative reply.

   If a positive proposal is sent and a data or control message is received while still awaiting a reply, the socket can infer that the remote socket has established the link on its end. Consequently, it transitions to the `ESTABLISHED` state and enables data/control message processing.

   This sub-state represents one of the few where a newer link request may arrive. If the metadata and reply have been sent, the remote socket has all the necessary information to determine if the link should be established. If the decision results in closing the link, the remote socket becomes free to initiate a new linking process – creating the possibility for a new link request to arrive before the reply to the previous request.

4. **Waiting to retry link establishment:** This final sub-state occurs when the remote socket is deemed eligible for link establishment, but a non-fatal reply has been received from it, indicating that the link cannot be established at the moment. In this temporary state, the socket waits for a specified duration before attempting a new link establishment, upon which it transitions to the *Waiting for metadata and reply* sub-state.

A transition to the `ESTABLISHED` state can occur from any of the first three sub-states within `LINKING`. This happens once the remote socket's metadata and reply have been received, allowing a final decision

on the link establishment to be reached. If both sockets determine that the link can be immediately established, the state transitions to `ESTABLISHED`.

A socket transitions to the `CLOSED` state if:

- The link establishment is deemed ineligible after receiving the peer's metadata.
- The peer determines that the link cannot be established (fatal reply), even if the local socket finds it viable.

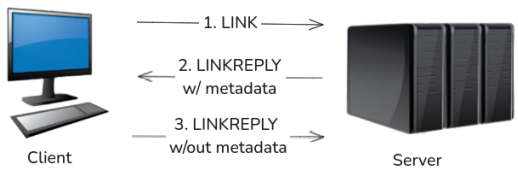
Single vs Concurrent Linking Processes

The link establishment process begins with a link request, represented by a `LINK` message. In a typical scenario, such as in a client-server model, a single socket initiates the linking process. However, it is possible that both sockets may attempt to initiate the linking process simultaneously, as in a peer-to-peer model.

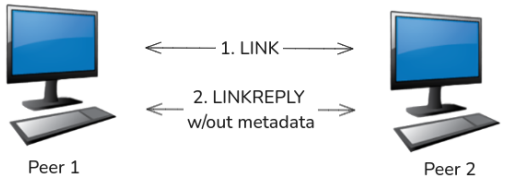
To illustrate how the linking process works, the two main flows are presented below. Variations involving different types of replies and out-of-order message delivery will be discussed in detail ahead.

- **Scenario 1 (Single Linking Process)**:
    1. Socket A sends `LINK` message to socket B;
    2. Socket B formulates a proposal using the received metadata and sends a `LINKREPLY` message containing its proposal and metadata;
    3. Socket A uses the received metadata to formulate its proposal and sends it to socket B in a `LINKREPLY` message (without metadata[29]). Socket A is now in possession of both proposals, therefore, it can reach the conclusion on whether the link should be established or not.
    4. Socket B receives Socket A's proposal, enabling it to reach the same conclusion as Socket A did.
- **Scenario 2 (Concurrent Linking Processes)**:
    1. Socket A and Socket B both send `LINK` messages to each other;
    2. Each socket uses the metadata of the remote socket to formulate their proposal;
    3. Both sockets share their proposals via `LINKREPLY` messages (without metadata);
    4. Both sockets analyze the proposals and reach a conclusion on the establishment of the link.

[29] When a socket has sent its metadata in a `LINK` message, including its metadata along the proposal in the `LINKREPLY` message is not required.



(a) Linking Scenario 1: Client-Server



(b) Linking Scenario 2: Peer-to-Peer

Skipping Transmission of Own Proposal

Previously, it was stated that a socket requires both proposals to finalize the establishment of a link. However, this is not always necessary. For instance, if the socket that did not initiate the linking process formulates a negative proposal, after sending it, it can immediately close the link on its end. If the proposal is fatal, then regardless of the initiating socket's proposal, the link cannot be established. In this case, upon receiving the metadata and fatal proposal, the initiating socket can simply close the link, skipping the need to send its own proposal.

If the proposal is non-fatal, it indicates that the responding socket is unable to establish the link at the moment and will discard the associated link data. This shifts the responsibility of link establishment to the initiating socket, which can then either close the link – if it is no longer needed or if the remote socket is deemed ineligible based on its metadata and the socket's internal conditions – or reschedule the linking attempt.

Out-of-order Reply

One scenario that can lead to out-of-order delivery occurs when a reply message arrives before its corresponding link establishment request. This situation is possible only when both sockets initiate the linking process concurrently. Recall that a reply – carrying the proposal – is sent only after a socket receives a `LINK` message. Consequently, a socket's reply can reach the remote socket before its own link request does.

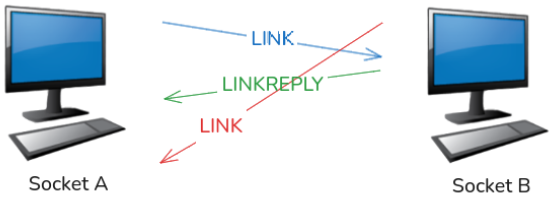


Figure 27: Reply message arriving before Link Request

Receiving a LINKREPLY message that lacks metadata indicates that the remote socket has sent its metadata in a LINK message. If the metadata is still missing, it means the corresponding LINK message is still in transit. Waiting for the missing LINK message is crucial for two reasons:

- **To Formulate the Local Proposal:** The metadata is essential for the socket to generate its own proposal, and, consequently, reach a conclusion regarding the link establishment. For instance, if the remote socket sends a positive proposal and the local socket's proposal is also positive, the link is established. However, if the remote proposal is non-fatal but the local proposal ends up being fatal, the link must be closed.
- **To Prevent Faulty Linking Processes:** If the remote proposal is fatal (indicating that the link should be closed), it might seem logical to immediately close the link. However, doing so without waiting for the LINK message could trigger faulty linking processes and potentially lead both sockets into an inconsistent state. After closing the link, when the LINK message is received, the socket will respond to the link request. At this point, the state of the sockets becomes inconsistent because the local socket is sending a LINKREPLY to the remote socket that has already closed the link. This situation is not anticipated by the state machine, which could lead to several outcomes, including the possibility of one or both sockets entering a perpetual linking state.

In the cases where only one socket initiates the linking process, this out-of-order scenario does not occur. The initiating socket will never receive a reply before a remote link request because the remote socket will include the necessary metadata with its proposal in a LINKREPLY message. As for the socket that does not initiate the process, it will not observe this situation, since the initiating socket, to send a reply, must first receive the necessary metadata, which is only sent by the remote socket upon receiving the initiating socket's link request.

Out-of-order Data/Control Message

Another out-of-order scenario is characterized by the arrival of a data or control message when expecting a reply from the remote socket. Because data and control messages can only be sent when the link is established, receiving a message of any of these types means the link has been established at the remote socket's end. Hence, the reply can be inferred as positive.

When a data or control message is received, as long as its incarnation identifier is a match, the link transitions to established, enabling bi-directional communication. Once the late reply arrives, it is simply ignored.

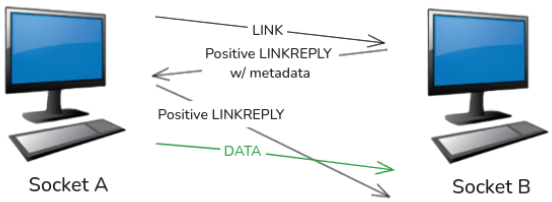


Figure 28: Data Message Received Before Reply

New Link Request During Ongoing Linking Process

One of the sockets will always perceive the link as established or closed before the other. This opens the possibility of a link request related to a new linking process, to arrive before a reply that will terminate the previously linking process unsuccessfully. This can be perceived easily through the following example:

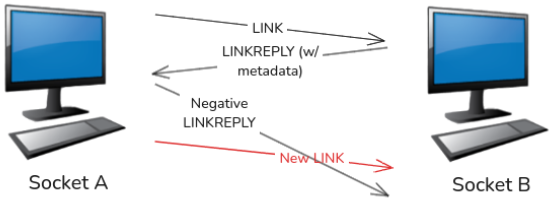


Figure 29: New Link Request Received During Ongoing Linking Process

4.6.5 Unlinking / Cancelling a link

If a link can be established, closing it must also be a possibility. We call this process of closing a link as **unlinking**.

If the socket's `unlink(SocketIdentifier)` method is called for a link that is established, the unlinking process begins. Consider the brief explanation of the process:

1. Socket A transitions to an `UNLINKING` state, preventing any more data messages to be sent through the link, and sends an `UNLINK` message to Socket B.

2. Upon receiving the `UNLINK` message, Socket B also transitions to the `UNLINKING` state, and sends back, to Socket A, an `UNLINK` message.

3. Once the closing conditions are met, each socket closes the link on its side.

Currently, there is only one *closing condition*. This condition consists in having delivered all data messages sent by the remote socket. Each socket includes the number of messages it has sent through the link in the `UNLINK` message. By exchanging the `UNLINK` messages, the sockets are not only saying that they have transitioned to an `UNLINKING` state, but also informing the number of data messages they have sent, which will not change as their transition to such state, prevents it. When processing the `UNLINK` message and after each data message is delivered, the socket checks its *deliver counter* against the *delivery goal* received. When these values are perceived as equal, the link is closed. Notice that if both sockets decide to initiate the unlinking process simultaneously, the end result is the same. The difference lies in not requiring to send an `UNLINK` message when the remote socket's one is received, since it was already sent.

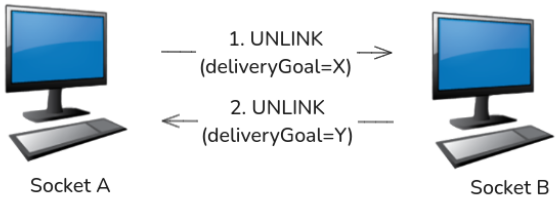


Figure 30: Unlinking Scenario

If a socket attempts to close the link during the link establishment, it may be possible to cancel the ongoing process. This can be done by sending a fatal proposal when the remote socket's metadata is received, effectively bypassing the proposal formulation. The socket waits for the remote metadata to account for situations where the remote socket may have a `LINK` message in transit, since closing the link immediately could lead to faulty behavior as described earlier. However, if a reply has already been sent, the socket must wait for the link establishment to complete. If the link is successfully established, the unlinking process is immediately initiated.

Unlinking Algorithm

The diagram 31 illustrates all transitions related to closing a link, most of which were covered when introducing the unlinking and cancelling processes above. However, certain scenarios involving the `LINKING` state require further attention:

- **Immediate link closure**: If `unlink()` is invoked while a linking process is scheduled (not yet active), the link is closed immediately.
- **Transition to the `CANCELLING` state**: A link moves to this state in the following cases:

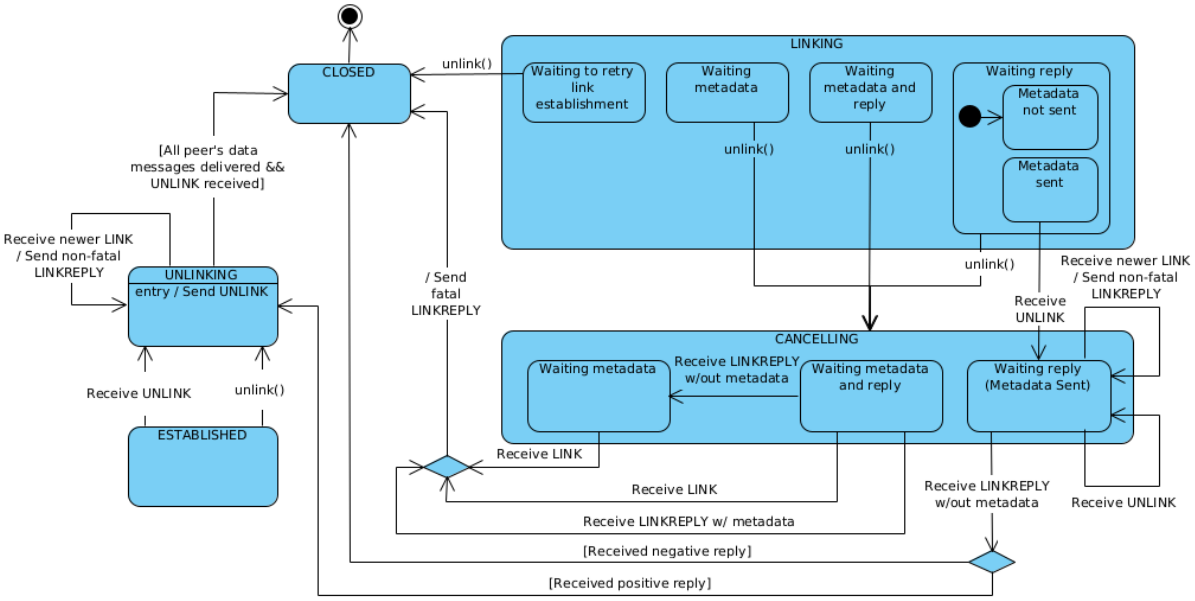


Figure 31: Unlinking Process State Machine Diagram

- **`unlink()` is called:** If the linking process is ongoing, the link transitions to `CANCELLING`, where it waits for the linking process to complete. If no proposal has been sent yet, a fatal reply is sent upon receiving the remote metadata, ensuring the linking process fails. If a proposal has already been sent, the socket awaits the remote reply, which can either close the link (negative reply) or establish the link (positive reply). In the latter case, the link transitions to `UNLINKING` and sends an `UNLINK` message, initiating the unlinking process.
- **Receiving an `UNLINK` message:** This represents the last out-of-order delivery scenario. An `UNLINK` message is only sent after the link is perceived as established. If such a message is received, it means the remote socket completed the linking process and then initiated the unlinking process. Since the link was established at the remote socket, a positive reply is expected. With this in mind, the link transitions to `CANCELLING` to handle the reply properly, ensuring a transition to `UNLINKING` when it arrives.

The remaining transitions include initiating the unlinking process from the `ESTABLISHED` state – either by receiving an `UNLINK` message or calling `unlink()` – and handling new link requests while in the `UNLINKING` state. Similar to the linking process, one socket may perceive the link as closed before the other, allowing it to initiate a new linking attempt while the remote socket is still completing the unlinking process. A non-fatal reply is sent to re-schedule the new linking process.

4.6.6 Future Enhancements

Here are some ideas for potential future enhancements related to the Link Management Protocol.

Exponential back-off

Currently, link establishment retries happen after a fixed delay. Ideally, an exponential back-off algorithm should be implemented, mitigating situations where a socket is bombarded, at a constant rate, with link establishment processes it will non-fatally reject. By introducing delay randomness and iterative growth, it addresses the thundering herd problem (bursts of linking requests) and reduces congestion.

Forced Unlink

Introducing a "force" flag in the `UNLINK` message could enable forcibly closing links with sockets deemed prejudicial by a policy. While this is not supported in the prototype due to its clear violation of exactly-once delivery semantics, enabling forced unlinking could enhance the middleware's versatility, offering a more practical solution for real-world scenarios.

Specialized Linking Conditions

As described above, the linking protocol is designed to support extensions to the linking process. This can be achieved by allowing custom metadata to be included alongside the existing metadata exchanged during the protocol. Additionally, an abstract socket method – called, for example, `customOnLinkRequest()` – could be defined to implement logic that evaluates the custom metadata and determines whether the link should be established.

Specialized Unlinking Conditions

To enable more control over the link closure, a future enhancement could introduce an event-driven mechanism for handling specialized unlinking conditions. This event would notify the socket of unlinking attempts, allowing it to track specialized conditions and either proceed with unlinking or defer it until the custom unlinking requirements are met.

If the unlinking conditions are dictated locally, the custom logic should override `unlink()` to enforce them, preventing a premature unlinking process. If conditions depend on the remote socket, `unlink()` can be enabled, since the transmission of the remote `UNLINK` can be delayed until all the necessary messages are sent. The delivery goal embedded in `UNLINK` will ensure the local socket delivers all the

messages before closing the link.

This enhancement enables finer control over the link lifecycle, however, careful implementation is required to prevent scenarios where a socket remains indefinitely in a waiting state (e.g. conditions are never met; or, `UNLINK` message is not sent).

4.7 Flow Control Mechanism

Flow control is critical for a well-functioning communication system. It enables throttling message transmission, thus preventing the receiver from being overwhelmed by messages it cannot process. Without flow control, resource exhaustion – both in memory and computation – can occur, potentially causing the receiver to crash.

To ensure senders follow the receiver's processing rate and to prevent senders from monopolizing resources of receivers, the middleware employs a credit-based mechanism at the link level to manage the flow of `DATA` messages.

4.7.1 Flow Control and Delivery Assumptions

The design of the flow control mechanism is based on two key assumptions:

- **Exactly-Once Delivery:** Guarantees that flow control messages are eventually delivered to their destination.
- **Unordered Delivery:** Flow control messages are delivered out of order, both relative to other flow control messages and other message types. This unordered characteristic aligns well with the flow control, as it ensures that flow control messages take effect immediately, without being delayed by unrelated missing messages.

These assumptions did not pose a problem, requiring only the design of a convergent mechanism to ensure that the two interacting sockets eventually reach a consistent state.

4.7.2 Per-Link State

Each link maintains state information for both incoming and outgoing traffic.

For incoming flow control:

- **Message Queue:** Stores incoming data messages until they are retrieved by the application for processing.

- **Capacity:** Defines the maximum number of data messages that can be queued for processing.
- **Batching Attributes:** To optimize network and computational efficiency, credits are issued in batches, with the batch size being determined as a percentage of the link's capacity.

For outgoing flow control, only a single attribute is required: the number of available transmission credits.

4.7.3 Flow Control Algorithm

This section details the algorithm of the flow control mechanism.

Since flow control operates at the link level, the sockets must exchange their capacities. This exchange happens during link establishment, where the sockets include their socket's default per-link capacity as transmission credits in the metadata.

Once the link is established, message transmission begins. A data message can only be sent if the link has an available credit. If a credit is available, the link is considered ready for sending, and the credit is consumed upon transmission.

When a data message is received, the socket processes it and stores it in the incoming queue. However, a credit is not immediately returned to the sender. Instead, credits are granted only after the message is delivered to the application, ensuring the imposed queue capacity is not exceeded. If credits were immediately returned, more messages could be transmitted by the sender ultimately leading the incoming queue to have an infinite amount of messages.

Once the accumulated credits reach a predefined threshold (a percentage of the capacity), a flow control message containing the batch of credits is sent to the remote socket to replenish the consumed credits (Figure 32).

Flow control messages are also used for link capacity adjustments (Figures 33 and 34), such as granting or revoking transmission credits. To increase a link's capacity, the difference between the new and current capacity is sent as positive credits. To decrease capacity, the difference is sent as negative credits to cancel excess credits on the sender's side. Although explicit capacity values can be set (Figure 34), the socket communicates these changes as relative adjustments. This approach ensures state convergence, as setting operations are not commutative with addition and subtraction operations, requiring strict message ordering guarantees for correctness.

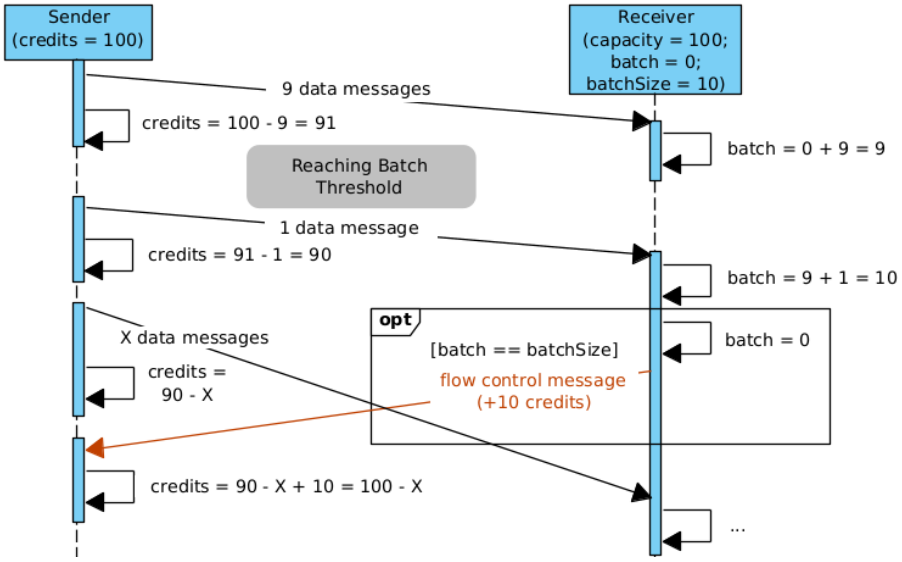


Figure 32: Replenishing credits (Assume data messages as immediately delivered upon reception)

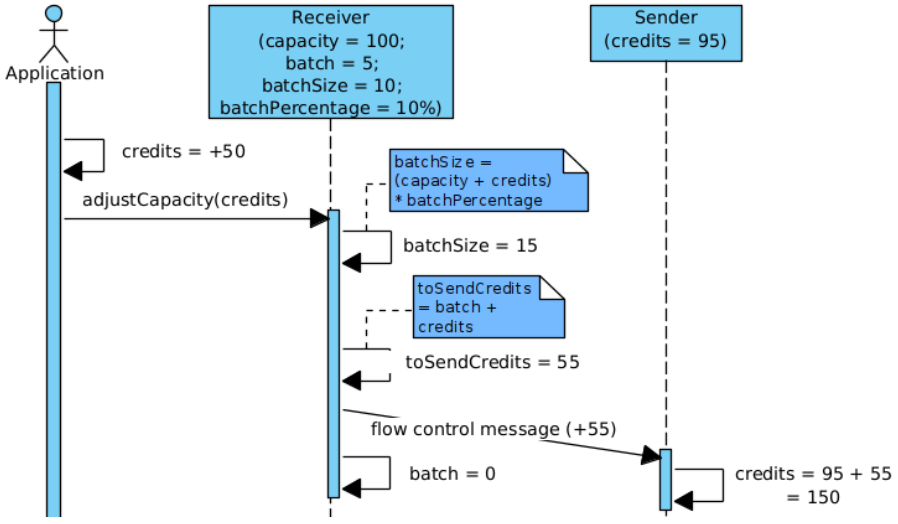


Figure 33: Adjusting Link Capacity

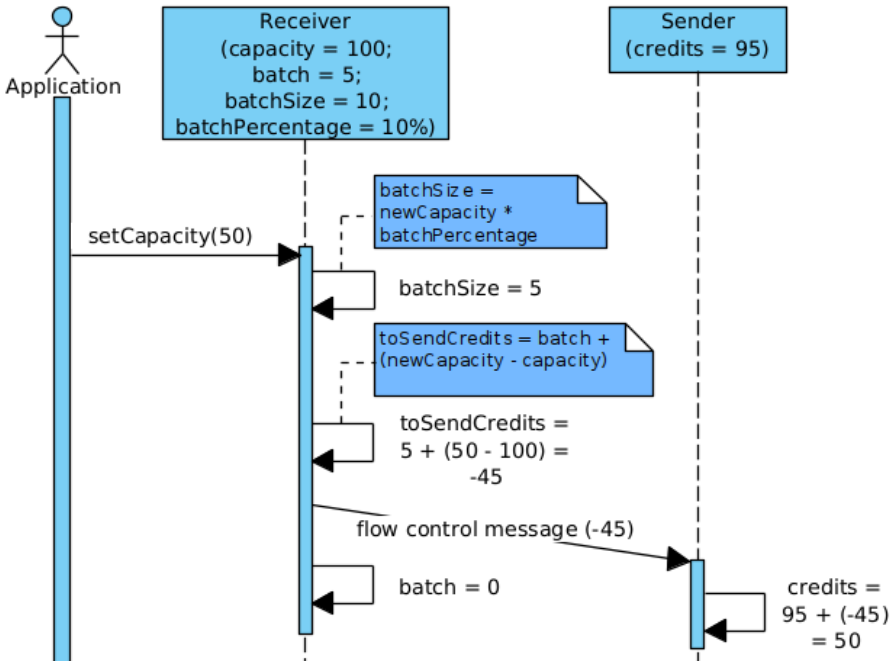


Figure 34: Setting Link Capacity

4.7.4 Flow Control Configuration

The initial capacity of links can be configured by adjusting the socket's `capacity` option.[30]

To modify the capacity of an established link, the `adjustCapacity(int)` (Figure 33) or `setCapacity(int)` (Figure 34) methods of the corresponding link socket can be used.

4.8 Sending a Message

Messages are the core of a messaging middleware. Having explored their structure, we will now focus on the lifecycle of a message, which can be broken down into two main phases: sending and receiving. We will start by exploring the details of the former, more specifically, a message's life from its creation to delegation to the transport layer.

The process of **sending data and control messages** involves multiple layers, with each component playing a specific role in ensuring reliable transmission. The socket determines the message's destination(s) and initiates transmission, the link socket enables communication through a specific link while allowing for customization, and the link ensures flow control and final submission to the message dispatcher. Finally, the message dispatcher serializes the payload and submits it to the Exon library, which

[30] By combining the socket option to set a maximum number of links with the initial capacity setting, the total number of messages that can be queued for application delivery can be determined as nMsgs = limitOfLinks × capacity. This estimation helps in roughly estimating memory consumption.

guarantees that the message reaches its destination exactly once.

For *data messages*, the sending methods must exhibit a blocking[31] behavior that temporarily prevents the client application from sending additional messages. This is crucial to avoid overwhelming receivers and ensure the middleware's exactly-once delivery requirements are met, which stipulate that messages must not be discarded until they are properly processed by the destination.

The sequence diagram (35) below provides a simplified representation of this process, illustrating these key components that a message traverses before being transmitted over the network. The following subsections will explore each component's responsibilities in more detail and identify areas where specialized behavior can be introduced to customize the process.

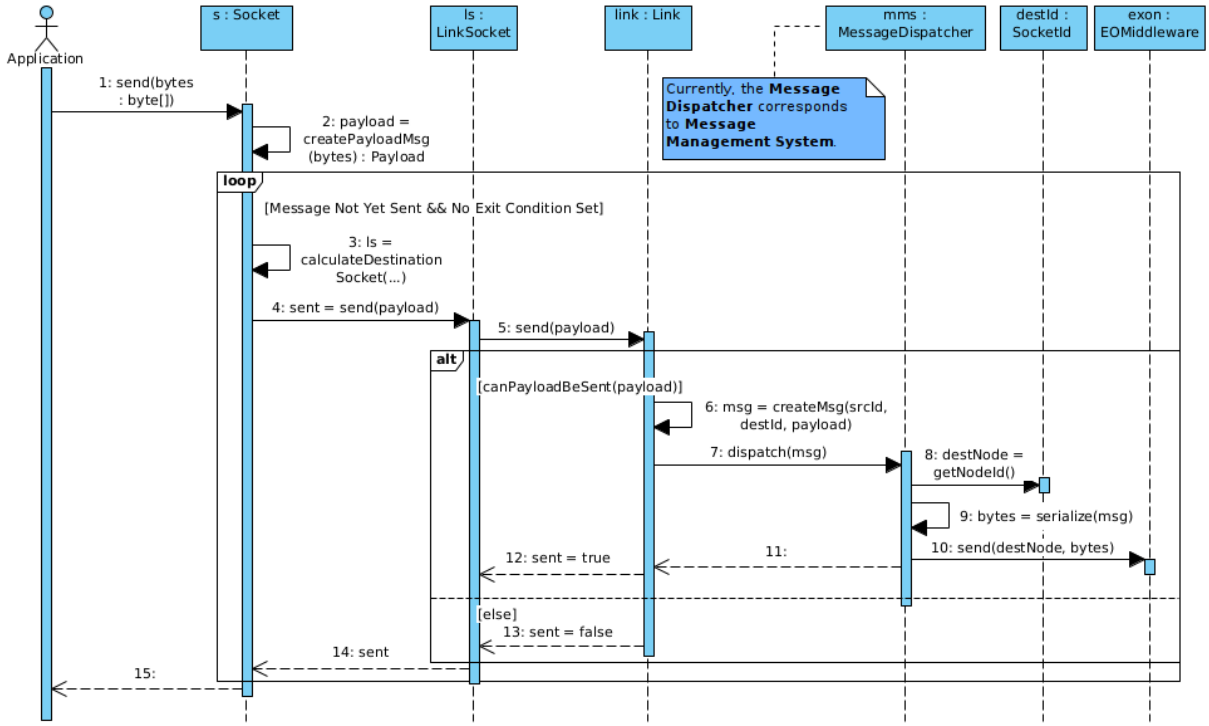


Figure 35: Sending a Message - Overview Sequence Diagram

In addition to data and control messages, other message types follow different sending and receiving processes, although, certain aspects of the process are shared. Currently, the only additional messages are link-management messages, which are generated by the **Link Manager** and immediately forwarded to the socket's **Message Dispatcher**, effectively bypassing the procedures required for data and control messages prior to handing them over to the message dispatcher.

[31] If a timeout or deadline is set for the sending operation, the operation will fail once the limit is reached, returning the responsibility over the data message to the caller.

4.8.1 Socket's Sending Method

The sending process, depicted in the sequence diagram, begins when the application invokes a socket's sending method, to which a message in the form of a byte array is provided. However, an application is not necessarily required to pass a byte array. Each socket may define its own objects to represent payloads, simplifying its usage by delegating serialization to the socket's internal logic.

The sending process for data and control messages follows a similar path, differing mainly in two aspects: (1) data messages are managed by a flow control mechanism, while control messages are not; and (2) the transmission of data messages is usually triggered by the application, while the transmission of control messages is mostly issued internally by the socket itself. Although an application may invoke a socket method that results in sending a control message, it is safer to restrict these messages to internal socket operations. The lack of flow control for these messages opens the possibility of overwhelming receivers.

The logic of the sending methods depends on the socket's semantics. While each socket type may have additional specific requirements, handling the following tasks is usual:

- **Creating a payload message:** Some sockets may require specific payload structures to carry additional metadata. For example, a socket might require backtracing information to allow messages to return to senders after having traversed multiple sockets. However, creating a custom payload is not always necessary, nor is it mandatory to be done at this stage (can be done in the Link Sockets' sending method).

- **Determining the destination socket:** The `calculateDestinationSocket(...)` method, as shown in the sequence diagram, represents the need to determine where the message should be sent. Typically, destinations are not explicitly provided as arguments to the socket's sending methods unless its semantics permit doing so. Instead, the socket itself must decide the destination(s), among the established links, based on a custom routing algorithm. Examples of factors that contribute to this decision are: load balancing (for single-destination delivery) and subscription-based filtering (for multi-destination delivery).

- **Defining exit conditions:** In the sequence diagram the only explicit returning condition is the delivery of the message to the Exon library for transmission. However, other conditions may cause an early return, such as timeouts, socket closure, and other socket-specific conditions.

Socket's Default Sending Method

The `Socket` class, as described in the Socket section, provides three public `send()` methods for message transmission. Since sending behavior depends on each socket type's semantics, these methods are overridable, allowing specialized implementations when needed. For example, in the typical *One-Way Pipeline* messaging pattern, a *Pull-Socket* does not support message sending. So, it can override these methods to throw an exception if the application attempts to send a data message.

With the intention of facilitating socket extensibility, while also acknowledging that control messages must still be allowed to be transmitted even if data messages are not, a default sending algorithm is implemented by the protected `sendPayload()` method. This method allows both data and control messages to be sent, and can be employed by the public `send()` method that restricts its use to data messages only. Since `sendPayload()` accepts a payload object, which includes the message type, it must remain protected to prevent applications from freely sending control messages. Unlike data messages, control messages are not managed by a flow control mechanism, making them susceptible to misuse, such as crashing a destination socket by overwhelming it with excessive transmissions.

The default sending method, `#sendPayload(Payload, timeout : Long, notifyIfNone : boolean)`, can simplify socket development (depending on the requirements) by acting as a template method, only requiring subclasses to implement a *`#trySending(Payload)`* method. Its algorithm is the following:

1. **Determine operation type:** The method calculates the deadline based on the given timeout, and determines whether the operation is blocking or non-blocking.

2. **Perform initial verifications:** These include checking that the payload is not null, verifying that the socket is in a state that permits sending, and ensuring links exist if the `notifyIfNone` flag is set. If any condition fails, the method immediately returns with an error.

3. **Register blocking operation:** If the operation is blocking, the client thread is added to the socket's wait queue as an exclusive waiter[32], having as events of interest: writing, errors, closure, and, if `notifyIfNone` is set, link unavailability.

4. **Handle non-blocking operation:** If the operation is non-blocking, the `trySending()` method is called directly, and the method returns immediately regardless of success.

[32] Registering client threads as exclusive waiters ensures that only one thread is woken up at a time.

5. **Enter blocking loop:** If the operation is blocking, the method enters a loop where the thread remains until the message is sent or a exit condition is met.

6. **Check exit conditions:** Before attempting to send, the method ensures the socket is not `CLOSED` and that the client thread has not been interrupted.

7. **Attempt transmission:** The `trySending(Payload)` method is called. This method is typically responsible for selecting the destination(s), retrieving the corresponding link socket(s), and invoking its(their) `send(Payload)` method. If the selected destination is no longer available to send (i.e. lacks credits to send data messages) and there are other possible destinations, the method should keep attempting to send until successful or until there are no available destinations. If the transmission is successful and the socket's state allows additional messages to be sent, the method should also trigger a writable event (`POLLOUT`). This ensures that a waiter, if existent, is notified and can proceed with handling the event. For instance, if the waiter is another client thread, it can attempt the transmission of its own message.

    - *Note:* When a socket relies on this default sending method and implements the `trySending()` method, it is generally recommended to use a non-blocking deadline[33] for the link sending operation. Doing so prevents the sending process from being blocked by a single link that is not essential for message delivery, enabling the socket to immediately attempt transmission through an alternative link.

8. **Handle transmission success:** If `trySending()` returns `true`, the message has been sent successfully, and the method exits.

9. **Handle transmission failure:** If the attempt fails, the thread waits using the default wait queue function.

10. **Wait for wake-up:** The client thread remains in a waiting state until it is woken up by one of the subscribed socket events, the timeout expires or the thread is interrupted.

11. **Verify wake-up reason:** If the wake-up event was due to a lack of links or timeout expiration, the method returns immediately.

[33] Although not shown in the sequence diagram, the Link Socket's and Link's sending method signature is `send(Payload, deadline : Long)`, where the deadline is a required argument. Non-blocking methods are often prefixed with "try" to highlight their non-blocking nature. This is why choosing a non-blocking deadline is important in this context.

12. **Handle remaining wake-up reasons:** If the thread was woken up for another reason, the method proceeds to a new loop iteration, where the exit conditions are re-evaluated and a new sending reattempt may occur.

13. **Perform cleanup:** Before exiting the method, the client thread is removed from the wait queue.

This default sending algorithm aids the implementation of sockets that deliver messages to a single destination. However, when a socket's requirements do not align with this approach, developers can override the `send()` method or define custom sending methods. Every socket subclass has access to all the necessary mechanisms to modify the sending process, with the only essential requirement for implementing specialized routing algorithms being the access to the link sockets of established links.

4.8.2 Link Socket's Sending Method

The primary role of a socket in the sending process is to implement a routing mechanism, i.e., determining the destination(s) of a message among the established links. The actual transmission of the message to each destination is performed using the corresponding link socket. This section briefly discusses the sending method of link sockets.

In the sequence diagram, the link socket's `send()` method appears to simply delegate the payload to the link's `send()` method. By default, this is indeed the behavior. However, by overriding the `send()` method or create additional sending methods, link sockets can implement specialized logic to meet custom requirements for targeted communication. An example of modification would be enforcing *FIFO* ordering for data messages exchanged within the link. Implementing such behavior at the socket level would be less efficient and more complex, as it would require additional data structures to track each link's state for ordering purposes, in addition to the management operations that require a shared locking mechanism for concurrency control[34]. Instead, creating a specialized subclass of `LinkSocket` is the ideal approach. To utilize this custom link socket, the socket's `createLinkSocketInstance(peerProtocolId)` method must be overridden to create instances of the new subclass when links are established.

It is also important to note that link sockets were designed while considering the possibility of being exposed[35] to the application if the socket's semantics allow selecting a destination socket. If exposed, the link socket's sending method becomes the first step in the sending process. This highlights that the sequence diagram presents just one possible process for message transmission. Regardless of the starting

[34] Using a shared locking mechanism means unnecessary contention between threads using different links.

[35] When exposing link sockets to the application, it's critical to prevent client applications from sending control messages. This can be achieved by either creating a wrapper class or overriding the `send(Payload, deadline : Long)` method to only accept data messages.

point, the message flow must remain unidirectional to prevent deadlocks: $Socket \rightarrow LinkSocket \rightarrow Link$. If any complementary sending logic is required at the socket level, the link socket should not be exposed to the client. Instead, to maintain this order and avoid circular dependencies, creating a sending method on the socket with the destination socket identifier as an argument is preferred.

4.8.3 Link's Sending Method

The sending method of the `Link` class falls within the core functionality of the middleware. Any required custom sending behavior must be implemented in a previous stage, either at the socket or link socket level.

The actual signature of the method is `send(Payload, deadline : Long)`, and its algorithm is very similar the socket's default sending algorithm. This similarity arises because, if the sending process starts at the link socket, the method must support blocking behavior to wait until a data message can be submitted for delivery.

The main differences between the two algorithms come from the different roles and characteristics of sockets and links. While both do similar checks, each one works based on its own state and attributes, not the other's. The key differences are:

- **State verification:** The socket's sending algorithm verifies the socket's state, while the link's algorithm verifies its own state. The link does not need to check the socket's state because if a message is being sent, the link must have already transitioned to the `ESTABLISHED` state – otherwise, the corresponding link socket would not exist. Additionally, when a socket transitions to `CLOSING`, no new links can be established, and the existing links begin unlinking. Therefore, for message transmission, the link only needs to verify that its own state is `ESTABLISHED`.

- **Payload verification:** In addition to ensuring the payload is not null, the link validates that its type corresponds to either a data or control message.

- **Transmission attempt:** Unlike the socket, which allows customization via `trySending()`, the link's transmission attempt is fixed. After the transmission is validated, a message is generated – using the payload along with the source and destination socket identifiers – and is then immediately forwarded to the Message Dispatcher.

- **Flow control verification:** The link manages its own flow of messages. It enables control messages to bypass flow control and be sent immediately, while credit availability is required to send

data messages. When trying to send a data message, the link attempts to consume a credit. If successful, the message is delegated to the Message Dispatcher, otherwise, the client thread may wait for a credit if the operation is blocking and no exit condition has been set.

- **Notifying waiters:** Each link maintains its own wait queue. After transmitting a data message, if credits remain available, the link emits a writable event to notify its own waiters rather than the socket's waiters.

4.8.4 Message Dispatcher's Dispatch Method

Each socket's **Message Dispatcher** currently corresponds to the **Message Management System**. Upon receiving a message, the MMS extracts the destination node's identifier, serializes the payload[36], and invokes the Exon library's `send(nodeId : String, msg : byte[])` method.

4.8.5 Exon Library's Sending Method

When the Exon library receives a message, it ensures the message is delivered exactly once to the destination node, thus completing the sending phase of the message lifecycle. Calling this method effectively submits the message for delivery, which is handled asynchronously by the Exon library's internal threads.

4.9 Receiving a Message

Receiving a message is a process that involves most of the middleware components, each playing a specific role in message processing. This receiving process is characterized by an arrival phase at the destination node (Figure 36), which all messages go through. However, data messages may[37] undergo an additional phase: delivery to the application (Figure 37).

A message arrives at the destination node via the underlying transport library (Exon). Once the message is received, the Message Management System is notified of this arrival, and proceeds to retrieve the message for processing. The message processing begins with parsing the received bytes into the generic message format discussed earlier. After successfully parsing the message, determining the destination socket becomes possible by consulting the socket manager for a socket identified by the message's destination tag identifier. If the corresponding socket was found, the message is handed over to it[38]. In the

[36] Each `Payload` class implements a `serialize()` method that converts the object's attributes into a byte array.

[37] The reason why data messages may not reach the remote application will be explained ahead.

[38] If the corresponding socket was not found and the message corresponds to a link request, the middleware replies with an error message to trigger the

socket, the message is first processed by the link manager, which validates the message for further processing. If valid, the message undergoes specialized processing determined by the socket's requirements. In the case of control messages, their processing ends here, with the socket's custom processing. Data messages, on the other hand, have the end of their receiving process determined by the socket's custom processing method. The return value of the method may either indicate the end of the processing, meaning the delivery can be acknowledged[39], or that the message must be inserted in the corresponding link's queue.

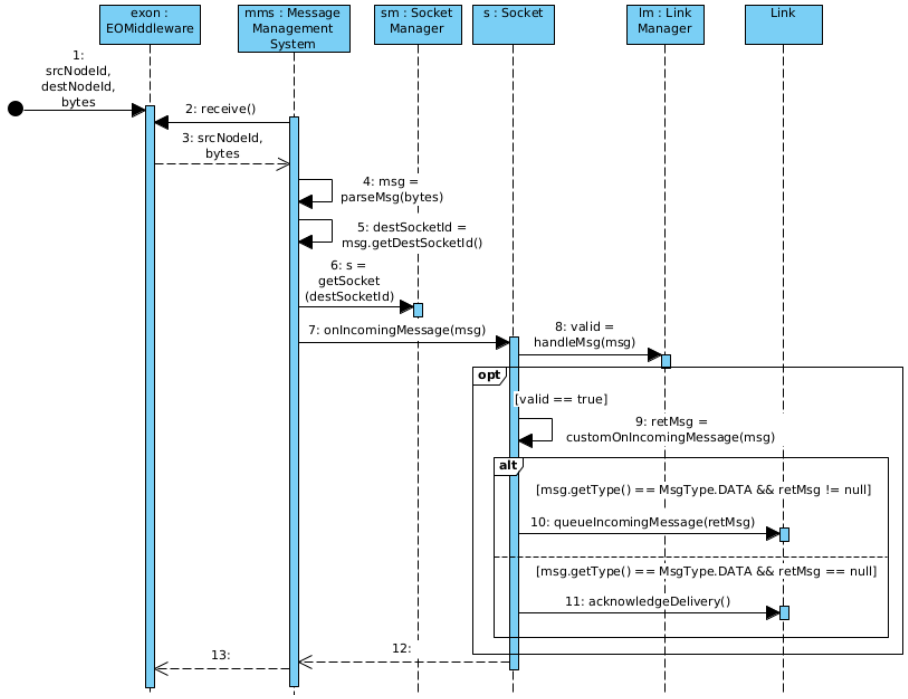


Figure 36: Message Arrival at Destination - Overview Sequence Diagram

The delivery of data message to the application happens when a socket's receive method is invoked. A receive method, like a send method, enters a loop where it awaits a message to be ready for delivery, which is determined by the socket's semantics. While there are several ways of programming a socket, the most common is to have a mechanism that finds a link with a message ready to be received. In that case, the data message is polled from the link's queue and the delivery is acknowledged.

4.9.1 Socket's Core Message Processing

When a message is received, the socket's core responsibilities are:

1. Direct all messages to the link manager.

re-scheduling of the link request. Otherwise, if the message is not a link request, it is discarded.

[39] When the delivery of a data message is acknowledged, a transmission credit can be returned to the sender (when the batch of credits is full).

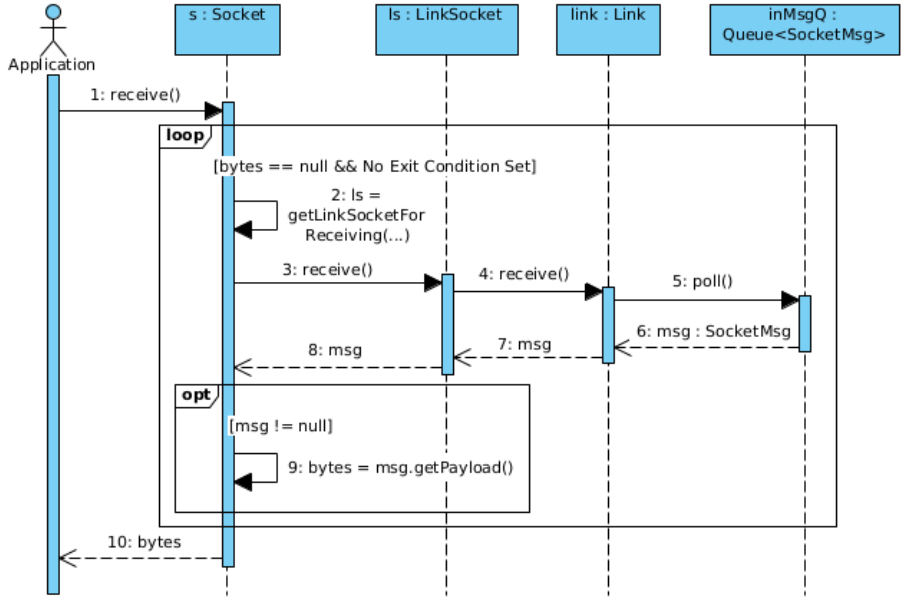


Figure 37: Message Delivery to Application - Overview Sequence Diagram

2. Submit validated data and control messages for specialized processing.

3. Confirm the delivery of data messages or place them in the corresponding queue.

Link Manager's Role

Having messages go through the link manager serves two key purposes:

1. **Intercept** messages related to core functionality, such as link management and flow control.

2. **Validate** data and control messages for further processing. During link establishment, data and control messages may act as positive replies, consequently finalizing the linking process. Additionally, whether for link establishment or general message validation, the link manager verifies the message's incarnation identifier to ensure it corresponds to the current conversation between the two sockets.

Automatic Flow Control

The socket's core processing method (`onIncomingMessage(SocketMsg)`) also executes an additional procedure for data messages related to the credit-based flow control mechanism. Since sending a data message requires transmission credits, and these credits are finite, they must be replenished to maintain communication flow.

To simplify the development of new socket types, sockets are designed to automatically provide credits to the sender. Since the primary goal of flow control is to prevent the sender from overwhelming the receiver, credits should only be replenished after the data message has been processed. So, automatic credit provisioning occurs when a data message is dequeued from the link's incoming queue or when the socket's custom processing method returns `null`, indicating that the data message has been handled and does not require queuing.

If `customOnIncomingMessage(SocketMsg)` returns a data message, it is added to the corresponding link's queue, and credit replenishment is deferred until the message is retrieved for delivery to the application.

Manual control of the flow control state is not permitted to prevent unintended tampering that could lead to inconsistencies. However, if manual control is necessary, a simple workaround can be employed. For instance, the specialized processing method could return dummy data messages and then "receive" them when wanting to return transmission credits to the sender.

If this approach is not desirable, a custom socket can set the link's capacity to an extremely high value, effectively taking the default flow control mechanism out of the equation and allowing the implementation of a custom flow control strategy. Since flow control states are public classes, they can be reused if needed.

4.9.2 Socket's Specialized Message Processing

Each socket must implement a message processing method, `customOnIncomingMessage(SocketMsg)`, responsible for executing any specialized logic required to handle incoming data and control messages.

Control messages are assumed to be handled immediately. If a control message needs to be queued – for example, to maintain order – the socket's custom logic must implement its own queuing mechanism for these messages, whether within the socket itself or the link socket.

For data messages, the socket may choose to process them fully, partially, or not at all. The specific processing of a data message is beyond the scope of the socket's core functionality. However, this method is expected to return either a data message or `null`, so that determining whether to acknowledge delivery or queue the message is possible.

Let us now explore the different levels of data message processing. If the socket does not require any custom processing, this method may simply return the received message as is. This ensures that the message is automatically queued in the corresponding link. In other cases, partial processing may be necessary – such as parsing the message into a custom object facilitating access to information necessary for further processing. For such scenarios, the method is allowed to return a modified version of the original

data message. For instance, a data message object that allows consulting an ordering identifier required by the link's queue for ordered delivery. Finally, a message may be fully processed, requiring no further action. In such cases, `null` is returned, acknowledging the delivery.

It is important to note that delivery of data messages to the application is not always guaranteed. These cases, where data messages are fully processed, serve additional purposes, such as internal socket operations. For example, certain metadata exchanges, triggered by the remote application, may be encapsulated within data message payloads. This approach ensures that such "control" messages are subjected to flow control, preventing excessive resource consumption caused by potential spamming from the remote application.

4.9.3 Socket's Receiving Method

To receive data messages, a socket must implement a receiving method. This can be done by overriding the default `receive()` method, which returns a byte array, or by implementing a custom receiving method that returns a specialized object.

For example, a *Router Socket* in the request-reply pattern routes messages using socket identifiers. A custom object simplifies route management while still allowing the byte payload to be set, eliminating the need for the application to handle serialization and deserialization of routing identifiers.

The receiving method's logic typically handles tasks such as:

- **Determining the receiving link:** The primary function of the receiving method is to determine which link a message should be received from. This is typically done by narrowing down the eligible links based on the socket's semantics and retrieving a message from one that is available for receiving[40].

- **Extracting relevant information:** When sending a message, additional metadata may be included (e.g. ordering identifiers). When this additional information is only needed internally, the receiving method must extract the relevant information while discarding parts that are unnecessary to the client application.

- **Defining exit conditions:** Similar to the sending method, the receiving method must establish conditions, depending on the socket's requirements, under which the operation should be canceled.

[40] Availability for receiving means having a message ready to be processed. While a link's queue may contain messages, depending on the socket's semantics, they may not yet be eligible for delivery.

Note: Most statements made regarding the socket's, link socket's and link's sending methods also apply to the receiving methods, however, with slight adjustments that align with receiving behavior. To minimize repetition, we will focus on the key differences between them.

Socket's Default Receiving Method

The `Socket` class provides three public variants of the `receive()` method, each requiring behavior specification according to the socket's semantics. While implementing a receiving method from scratch is possible, using `#receiveMsg(timeout : Long, notifyIfNone : boolean) : SocketMsg` – the default mechanism for `receive()` – can significantly speed up development if it aligns with the socket's requirements. This method is very similar to the default sending method, `sendPayload()`, but is designed for receiving instead of sending.

The core algorithm of these two methods is essentially the same, with the following differences:

- **Subscribing to read events:** For blocking operations, the method subscribes to the read event (`POLLIN`) instead of the write event (`POLLOUT`). This ensures the method waits for the socket to be ready to deliver a message rather than checking its availability to send one.
- **Receiving messages:** The method relies on a custom implementation of `tryReceiving() : SocketMsg` instead of `trySending()`. `tryReceiving()` should iterate over link sockets, deemed eligible for receiving based on the socket's semantics, until a message is retrieved. If receiving fails – when none of the eligible link sockets are available for receiving – the method returns `null`. For blocking operations, the default receiving method waits for a read event before retrying, i.e. invoking `tryReceiving()` again. Upon success, if additional messages are available, the method notifies socket waiters using `POLLIN`, allowing another client thread to proceed with the receiving process.

4.9.4 Link Socket's Receiving Method

Once a link is selected for receiving, the next step is invoking the link socket's receiving method. However, this does not guarantee a message will be received. Until the entire receiving process is completed – i.e., polling a message from the link's queue – it is not possible to confirm message availability. For example, another client thread may have concurrently retrieved the only available message.

By default, this method calls the underlying link's receiving method. However, it can be customized by overriding it, or alternatively, a new method can be implemented to tailor the link socket's receiving process.

An example of customization is a receiver-side message filtering mechanism, which might invalidate certain messages before delivery. If no messages remain in the link, the receiving operation fails, requiring the socket's receiving method to select another link.

Additionally, unnecessary message information can be removed at this stage rather than at the socket level.

4.9.5 Link's Receiving Method

The link's receiving method, `receive(deadline : Long) : SocketMsg`, follows the same core algorithm as the sending method, but adapted for receiving. This includes polling messages from a queue, working with read events for queuing and notifying waiters, and acknowledging message delivery to replenish the sender's credits.

An additional key difference between the two methods is that the receiving method must track the number of messages delivered. When unlinking, the link can only close once the number of deliveries matches the number of messages sent by the remote socket.

4.9.6 Link's Queue Polling Method

A successful receiving process concludes with polling a message from the link's queue. The `poll()` method plays a crucial role in this process, as it enables indirect customization of a link's receiving behavior. While the `Link` class does not allow direct modification of its methods, its receiving method behavior can be customized through a custom queue.

When a link is established, the socket assigns a queue to the link, obtained via `createIncomingQueue(peerProtocolId : int) : Queue<SocketMsg>`. Since this method can be overridden, it allows defining a queue with specialized behavior. For example, FIFO ordering can be achieved with a custom queue[41] that sorts messages by their ordering identifier and only returns a message when the one at the head of the queue matches the expected identifier. Combined with the default flow control mechanism, the link will exhibit behavior that resembles a sliding window.

[41] Custom queues must follow specific guidelines, but the most critical requirement is that `poll()` and `peek()` return `null` when there are no messages ready to be received, even if the queue is not empty.

4.10 Event Notification

In Polling Mechanism and Wait Queues, a generic polling mechanism was discussed. This section briefly discusses how that mechanism is put to use by links and sockets for event notification.

Every socket and every link have a wait queue for notification purposes, mainly of the events: write (`POLLOUT`), read (`POLLIN`), close (`POLLHUP`), and error (`POLLERR`).

Their wait queue not only allows managing client threads waiting for those events but also facilitates behavior extensibility by notifying internal components. When an internal component subscribes to events, it must be registered as a *non-exclusive waiter* to avoid interfering with client threads by consuming the events.

Registering internal components as waiters has specific guidelines that must be followed:

- **Queuing order:** When registering internal components as non-exclusive waiters, especially when their behavior is correlated, it is important to ensure the components are queued in the correct order. Since non-exclusive waiters are prepended to the queue and the notification calls of a wait queue begin at the head of the queue, queuing these components must be done in the reverse order in which we want them to be notified.

- **Careful use of locking mechanisms:** When registering a waiter, it is important that the locking mechanisms used while queuing (e.g., in a *poll queuing function*), do not correspond to the mechanisms used in the *wait queue function*. This is essential to prevent deadlocks[42] between threads attempting to queue a waiter and a thread attempting to notify an event. The ideal approach is using lock-free mechanisms such as atomic variables, etc.

Link's Event Notification

We will now discuss how each of the main events is notified by links.

A *write event* is notified in two situations:

1. When a link receives a flow control message that transitions the credit balance to a positive amount;

2. When a send operation succeeds after blocking the thread.

Notifying only in these situations minimizes the number of events notified, contributing to efficiency. If the credit balance is already positive, a send operation will not block, thus eliminating the need for

[42] Learning the polling mechanism of the Linux Kernel by exploring its source code served as a great introduction to lock-free strategies, helping me devise my own strategies.

notification since no threads are being blocked while attempting to send. However, when the credit balance becomes positive or a thread is blocked waiting for credit, notifying the write event ensures that a waiting thread is awakened.

The same principles are applied with *read events*, which are notified when:

- The queue becomes ready to return a message when polled;
- A receive operation succeeds after blocking the thread.

Since a queue may contain messages but not necessarily have one ready for immediate retrieval, the link verifies whether queuing a message transitions the queue into a state where polling a message is possible. At that moment, notifying the read event is appropriate, as a waiting thread can effectively retrieve a message.

Finally, the link notifies the *close event* to all waiters when the link transitions to `CLOSED`.

Socket's Event Notification

Regarding how events are notified by sockets, most of them – *read, write, and error events* – are handled by the socket's custom logic, as each socket type may have different requirements for when to notify these events.

Since reading from and writing to a socket are inherently linked to its associated links, custom socket implementations typically subscribe non-exclusively[43] to the events of their links. This allows the socket to process these events, update its state accordingly, and potentially notify the same event itself.

The *close event* is the only event that is part of the socket's core functionality. It is notified when the socket transitions to `CLOSED`.

Additionally, sockets have a unique event called the *link-free event*[44]. Each time a link is closed, the socket checks if there are no remaining links. If no links exist, a wake-up call is performed for all waiters. This wake-up call is performed differently in comparison with the remaining events: it utilizes the `mode` argument of *wait queue functions* to enable interested waiters to detect the link-free event. Additionally, it provides an empty event mask as the `key` to ensure that it does not interfere with waiters that are only aware of default event types.

[43] Remember that a non-exclusive waiter receives all notifications, regardless of the existence of exclusive waiters.

[44] The link-free event is associated with the `notifyIfNone` flag present in the default `receive()` / `send()` methods. If this flag is set, the client wants the waiting operation to be canceled once the link-free event is received.

4.11 Concurrency and Scalability Design

Performance is not the primary focus of the middleware, however, achieving reasonable performance is crucial for its viability as a solution. This section discusses the programming model and related design considerations.

4.11.1 Programming Model

A key challenge in designing a messaging middleware is selecting an appropriate programming model. Since messaging systems inherently involve concurrency, this aspect significantly influences the selection of a model. Considering that a middleware instance can manage multiple virtual sockets with independent traffic and processing requirements, the actor model might seem ideal. However, implementing the actor model in Java would essentially translate to a thread-based model, where each socket is assigned its own thread for parallel message processing. Unlike actor-based languages such as Erlang, where processes are extremely lightweight, Java threads are comparatively expensive, making this model a less desirable choice in this context.

Considering that the incoming messages processing required by most sockets is minimal, a design choice was made to use a **single middleware thread** to read messages from the transport library and perform any essential processing determined by the destination socket. Given that most of the middleware use cases do not require extremely high data throughput, having a single thread handle processing for all sockets is potentially more efficient than using multiple threads, which would compete for resources, increase contention, and add context-switching overhead. In the future, if a single thread proves insufficient, a thread pool could be introduced to handle incoming messages. The pool size should be carefully chosen by the application based on its demands, as excessive threading could degrade performance due to resource contention and increased context-switching.

If a socket requires intensive message processing, it may delegate this work to additional threads, such as client threads or dedicated worker threads[45], preventing excessive load on the middleware's main thread, which affects system responsiveness.

Regarding outgoing messages, the entire sending process until the submission to the transport library can be managed by client threads. Due to exactly-once semantics, client threads, before returning, must wait until message submission is possible, thus making client threads suitable to perform any operations

[45] Dedicated worker threads can be initialized via the socket's `init()` method. This method is reserved for custom initialization procedures, and is invoked when the socket is started.

related to the sending process.

To complement this threading model and enhance performance, an **event-driven programming** approach was adopted. This allows the middleware to react to events as they occur, eliminating the need for additional threads and avoiding inefficient busy-waiting mechanisms.

4.11.2 Link-Level Concurrency

One important design consideration for improving performance is enabling link-level concurrency, such as sending messages through multiple links simultaneously. This is made possible by the close relationship between a link's state and its owner's (socket) state. A link cannot be established until the socket is started, and a socket cannot be closed until all its links are closed. Therefore, from the moment a link is established until it is closed, the socket is guaranteed to allow communication.

As long as the socket does not have specific requirements that demand concurrency control at the socket level – like preventing consecutive sends (like the Request Socket) – this socket-level concurrency control can be limited to core operations. This approach minimizes contention and boosts performance. For instance, when developing a socket's sending method, a strategy to improve concurrency could involve acquiring the socket's read lock to retrieve an available link socket, quickly releasing it, and then working exclusively with the link's socket. If needed, concurrency control can be implemented at the link socket level – for instance, to assign message order identifiers – otherwise, links already manage concurrency for core operations such as sending and receiving.

4.11.3 Other Performance Considerations

Considering the highly concurrent environment of the middleware, especially given the thread-safety usability requirement, the use of locking mechanisms is crucial to ensure consistency. However, these mechanisms introduce contention, which negatively impacts performance. To maintain a responsive system, minimizing contention is a critical design goal.

The following principles were applied in the design of the middleware and its default sockets, and should be followed when designing new ones:

- Minimize contention by holding locks only for operations that strictly require them.
- Use or develop lock-free mechanisms and strategies, such as atomic variables, whenever possible. For example, the socket's state is stored using an atomic variable, enabling lock-free verifications

of the socket's state by the sending and receiving algorithms, thereby preventing unnecessary contention.

- When monitoring events, prefer reactive mechanisms like polling for responsiveness.
- Prevent unnecessary event notifications. Avoid notification loops and ensure that events are only notified when relevant[46]. Additionally, carefully select the number of waiters to be notified to minimize context-switching overhead[47].

When designing new socket types, applying these principles and minimizing the workload of the middleware thread during message processing is particularly crucial, as it is directly linked to system responsiveness.

[46] For links, the send event signals credit availability for transmission. Triggering it for every received credit is unnecessary unless the link transitions from no credit to having credit(s).

[47] For example, notifying two threads to send a message when they can't operate concurrently may result in one thread to return to a waiting state right after being woken up.

Chapter 5

Specialized Sockets

After designing the core of the middleware, the next step is to verify its support for various messaging patterns. Three messaging patterns, having ZeroMQ as the source of inspiration, were implemented to verify this: Request-Reply, Push-Pull (One-way Pipeline), and Publish-Subscribe.

The strategies used for implementing these sockets can also serve as inspiration for developing new socket types.

5.1 Configurable Socket

Analyzing the ZeroMQ sockets revealed that many socket types share similarities, differing only in a few aspects. To reduce code duplication, a configurable socket was designed, enabling the creation of various socket types by extending this base class and making a few adjustments[1].

The configurable socket:

- Supports FIFO message ordering, with separate ordering for data and control messages. This ordering guarantee is optional.
- Allows enabling or disabling sending and receiving operations.
- Uses round-robin when receiving and transmitting data message, but skipping unavailable sockets.

5.1.1 Socket Creation

Reading, writing, and ordering capabilities are set via the socket's constructor. These capabilities remain fixed after creation.

[1] When a new socket type is created, the main modifications involve defining a protocol identifier that uniquely represents the socket type and specifying a set of compatible protocols.

When reading or writing is enabled, a `Poller` instance is created to monitor the corresponding events of established links. For example, enabling writing creates a poller specifically for tracking *write events*, referred to as the *writing poller* for simplicity.

If reading is disabled, the socket's default "capacity" option is replaced with an immutable option handler, setting link capacities to zero. This prevents remote sockets from sending data messages and disables capacity configuration.

5.1.2 Establishing a Link

When establishing a link, the configurable socket behaves as follows:

1. The `createLinkSocketInstance()` method is overridden to associate a specialized link socket with each new link. This is relevant only if ordering is required, as it customizes sending and receiving behavior.

2. The `createIncomingQueue()` method is also overridden. If ordering is needed, a special queue is created for the link. This queue ensures messages are retrieved in the correct order based on their identifiers and that polling is only possible when the correct message is at the front of the queue.

3. The link is registered in the reading and writing pollers if those capabilities are enabled. For reading, the reading poller subscribes to the link's read events, and the same applies to writing if the capability is enabled. If the link is immediately available for the operation, a socket's waiter is notified.

To ensure round-robin fairness, the socket relies on the pollers' underlying logic. When a link is retrieved, it is moved from the front to the back of the ready list, ensuring other links are used before it is selected again. Separate pollers are required for each event type to maintain fairness. For example, if a link is used for receiving, it should not be moved to the back of the queue, which would interfere with its ability to be used for sending.

5.1.3 Closing a Link

When a link is closed, it is removed from the socket's internal pollers it was registered in.

5.1.4 Sending a Message

The writing capability determines whether the socket can send messages. If sending is disabled, the default `send()` methods throw an unauthorized operation exception.

When writing is enabled, the configurable socket follows the behavior defined below.

Configurable Socket Sending Method

The configurable socket's `send()` methods utilize `sendPayload()`, which relies on the implementation of `trySending()`.

The `trySending(Payload)` algorithm follows these steps:

1. The *writing poller* is utilized to retrieve a link socket that has signaled a write event.
2. A non-blocking attempt is made to send the message through that link socket.
3. If successful, the socket signals the write event to wake another thread waiting to send and returns success.
4. If unsuccessful, the process repeats until the message is sent or no links are available.

Link Socket Sending Method

The `trySending()` method passes the payload to a link socket that signaled availability for sending. If FIFO ordering is not required, the message is simply forwarded by the link socket. Otherwise, the (specialized) link socket enforces ordering behavior.

For ordered sending, the specialized link socket maintains two counters – one for data messages and another for control messages – to determine the next message identifier.

When sending a message:

- Control messages cannot be rejected, so an atomic integer is enough to ensure consistency when incrementing the counter on each send.
- Data messages require a locking mechanism to prevent inconsistencies. The lock safeguards the process since the acquisition of an ordering identifier until the message submission confirmation. If sending fails due to insufficient credits, the counter remains unchanged, avoiding gaps or duplicate identifiers.

For example, with an atomic counter rather than locking, thread A might attempt to send a message with ID 1, while thread B sends with ID 2. It is possible for thread A to be rejected due to lack of credits but for thread B to succeed after the link receives flow control credits. This would result in an identifier gap. Using a lock prevents such inconsistencies.

This completes the configurable socket's sending process.

5.1.5 Receiving a message

Receiving shares many similarities with sending, so we will focus on the key differences.

The reading capability determines whether the socket can receive messages. If disabled, the default `receive()` methods throw an exception.

When enabled, the behavior is as follows:

Configurable Socket Message Processing

When a message arrives at the socket, the type of message determines the processing approach:

- For `DATA` messages, if ordering is required, the message is parsed to separate the ordering identifier and application payload, and then inserted into the link's queue. If ordering is not required, the message is processed according to custom logic, but by default, it is simply queued.
- For control messages, if ordering is required, the message is queued in an additional queue within the specialized link socket. Once queued, an attempt is made to poll and process all control messages ready to be dequeued. If ordering is not required, the control message is processed immediately.

Configurable Socket Receiving Method

The `receive()` methods utilize the default receiving algorithm, `receiveMsg()`, which requires the implementation of `tryReceiving()`.

The `tryReceiving() : SocketMsg` method works similarly to `trySending()`. The differences are as follows:

- The *reading poller* is used instead of the *writing poller* to identify available links for receiving.
- It signals a read event to wake a waiting thread for receiving, rather than notifying a write event.
- The method returns when a message is received or when no links are available.

Link Socket Receiving Method

The link socket's `receive()` method simply calls the link's receiving method. This is sufficient for receiving data messages, both with and without ordering. If ordering is required, the link's queue ensures that messages are only dequeued in the correct order and when the message at the head has the next expected identifier.

5.2 Push-Pull Messaging Pattern

The Push-Pull messaging pattern, also known as One-way Pipeline, is easily implemented using the *configurable socket*.

Both `PushSocket` and `PullSocket` are subclasses of the configurable socket. The `PushSocket` enables sending with ordering capabilities, while the `PullSocket` supports receiving with ordering capabilities.

The configurable socket provides all the necessary logic, requiring only these sockets to define a unique protocol identifier and specify each other as the compatible protocols.

5.3 Request-Reply Messaging Pattern

5.3.1 Request Socket

The Request Socket (`ReqSocket`) is part of a synchronous variant of the request-reply messaging pattern. After sending a request, it expects a response before issuing a new request.

Requests are sent in a round-robin fashion, as handled by the configurable socket.

A unique aspect is introduced due to its thread-safety requirement:

- Once a request is sent, any thread may retrieve the reply.

  If a dedicated receiving thread is used, it must employ a polling mechanism to monitor the socket, either via a poller instance or the socket's `poll(events : int, timeout : Long) : int` method. Attempting to receive a message before a request is sent results in an exception indicating the absence of a pending request.

- When multiple threads attempt to send requests concurrently, the `send()` method blocks until the socket is available for sending. If a request is already in progress, new transmissions are only allowed after the corresponding reply is received.

A synchronization mechanism ensures that request submissions are sequentialized across threads, preventing concurrent submissions.

The `send()` method does not throw an exception when waiting to send while a response is pending. This allows applications to use separate threads for sending and receiving. Alternatively, if multiple threads send requests and receive the respective replies, by waiting on `send()` until the request is transmitted, and only then invoking `receive()`, the threads are guaranteed to receive the answer to their request.

Establishing a Link

To maintain round-robin fairness among destinations, this socket follows the same approach as the *configurable socket*. When a link is established, it is registered in the *writing poller* to track *write events* and determine when there are destinations available for receiving.

Closing a Link

When a link is closed, it is removed from the *writing poller*. If the closed link corresponds to the expected replier, the socket signals a *read event* so that the receiving thread detects the closure. This situation can be avoided by ensuring that the requester is responsible for closing links only when there is no pending request.

Invoking `unlink()` with the identifier of the replier is not permitted. To force the operation, the socket provides a `forceUnlink(SocketIdentifier)` method, which forcefully closes the link from which a reply is expected. This effectively cancels the ongoing request and initiates the unlinking process.

Sending a Message

The sending method relies on the default `sendPayload()`, requiring an implementation of `trySending()`. The algorithm to attempt sending ensures that no ongoing request exists before iterating over available destinations provided by the *writing poller* until the request is successfully sent or there are no available destinations. The link socket does not introduce additional behavior, limiting itself to forward the message to the associated link, where the message transmission depends solely on the credit balance.

Receiving a Message

The receiving method utilizes the default `receiveMsg()`, dependent on `tryReceive()`. The tryReceive() method throws an exception if no reply is expected. If the link with the replier is closed, the receiving

thread clears the ongoing request and exits with an exception to indicate the situation. Otherwise, when a request is pending, an attempt to receive the reply is made. If successful, a *write event* is notified to allow another thread to send a new request.

5.3.2 Reply Socket

The Reply Socket (`RepSocket`) is the counterpart of `ReqSocket` in the synchronous variant of the Request-Reply messaging pattern, but can also interact with the `DealerSocket`.

This socket listens for requests and replies to them. It acts as the opposite of `ReqSocket`, processing a single request at a time, replying to it, and only proceeding to the next request.

Establishing a Link

When a link is established, it is added to the `reading poller` to monitor links for incoming requests.

Closing a Link

When the link is closed, it is removed from the poller.

Invoking `unlink()` to close a link with a socket from which a request has been retrieved but a reply has not yet been sent is not allowed. However, the `forceUnlink(SocketIdentifier)` method can force the operation, freeing the socket to handle new requests.

Sending a Message

The sending process uses the `trySending()` method to send a reply to the requester. If no requester is set, the operation fails. After sending the reply, a read event is triggered, signaling that a new request can be received.

Receiving a Message

When a request arrives, it is directly queued in the link's queue.

The `tryReceiving()` method uses the *reading poller* to find an available link with a request. Once a request is successfully retrieved, the corresponding link is set as the requester, so the reply destination is automatically set when invoking the sending method.

5.3.3 Dealer Socket

The `DealerSocket` is the asynchronous variant of the `ReqSocket`. It can send and receive messages in any order, without synchronization between operations.

This socket is a subclass of the *configurable socket*, with both reading and writing capabilities but no ordering.

5.3.4 Router Socket

The final socket type of the Request-Reply messaging pattern, inspired in ZeroMQ, is the `RouterSocket`. It can act as the asynchronous counterpart to the `ReplySocket`, but its primary role is message routing. Using routing data, a router socket can forward messages back to their original sender or direct them to a specific destination.

This socket extends the *configurable socket* to inherit its receiving behavior, while the sending logic is entirely dictated by the routing data contained in the payload of messages.

Sending a Message

The `RouterSocket` is the first socket that does not use the default sending algorithm. Instead, it provides a specialized sending method: `send(msg : RRMsg, timeout : Long)`.

The `RRMsg` is a custom payload object containing both a byte payload – representing the actual application data – and a queue of routing identifiers.

Routing identifiers can be either socket identifiers or integer values. Integer values are typically used internally by the router socket to minimize message size by replacing socket identifiers – which can be relatively large – with the local incarnation identifier of the corresponding link. For example, when specifying a route for message transmission, socket identifiers are used as they are the only information known by the application. However, upon receiving a message, the router socket replaces the adjacent source socket's identifier with an integer, corresponding to its local incarnation identifier.

When sending a message, the routing identifier at the head of the queue is removed and used to determine the corresponding destination link. If the routing identifier matches a valid destination, the link socket's send method is invoked, which passes the message directly to the link for submission. If the routing identifier is invalid, the message is silently discarded, as the destination socket either does not exist or has been closed.

Receiving a Message

Upon receiving a message, the socket prepends the local incarnation identifier associated with the incoming link to the routing identifiers' queue. Using incarnation identifiers for routing has an additional benefit: it ensures that messages are routed back through the exact links they originally traversed, preventing messages from being delivered to unrelated conversations.

To deliver messages to the application, a custom receive method is provided: `recv(timeout : Long, notifyIfNone : boolean) : RRMsg`. This method utilizes the configurable socket's receiving mechanism to retrieve a message from an available link and parses the payload into an `RRMsg` object. This allows the application to access the actual application data while also enabling the modification of the routing identifiers if necessary.

5.4 Synchronized Publish-Subscribe Messaging Pattern

The implemented Publish-Subscribe messaging pattern is experimental. It is inspired by ZeroMQ but follows a synchronized variant. Under the exactly-once delivery guarantee, messages must not be discarded. So, if a publisher accumulates messages indefinitely for slower subscribers, it can lead to memory exhaustion and, ultimately, a crash.

To address this challenge, a synchronized approach was devised: the publisher adjusts its publishing rate to match the slowest subscriber. This ensures that messages are not discarded – adhering to the exactly-once delivery requirement – while also preventing excessive message accumulation.

5.4.1 Publisher Socket

The Publisher Socket (`PubSocket`) is responsible for publishing messages to specific topics and distributing them only to subscribers of those topics (publisher-side filtering).

This socket is a subclass of the configurable socket, taking advantage of its ordering mechanism and monitoring capabilities to track link availability for sending.

Manage Subscription Messages

Publishers determine message destinations by analyzing topic subscriptions. Each subscribed topic is associated with a `Subscription` object that maintains a list of interested subscribers. When publishing, the socket selects all subscriptions whose topics are prefixes of the published message's topic. To

enable efficient retrieval of these subscriptions – and consequently the intended destination sockets – subscriptions are stored in a *Patricia Trie*, as described by Sustrik (2013b).

These subscriptions are managed through the control messages: `SUBSCRIBE` and `UNSUBSCRIBE`. The `SUBSCRIBE` message signals interest in a topic, while `UNSUBSCRIBE` cancels it.

Since these messages are not commutative, the socket relies on the ordering mechanism of the configurable socket to ensure they are processed in the correct sequence.

Processing subscription messages is not an inexpensive task. To maintain system responsiveness, when such a message is received, the middleware thread first attempts a non-blocking acquisition of the required lock for its processing. If the lock is unavailable, the message is queued as a task, deferring its processing until a client thread attempts to publish a message. Before publishing, the client thread processes any pending subscription tasks to ensure the destination list will be up to date.

Note: In a controlled environment where subscribe and unsubscribe messages are not misused by subscribers to overwhelm receivers, treating them as control messages does not pose an issue. However, if the middleware is later adapted for broader use cases, it would be beneficial to convert these messages to data messages, enabling their transmission to be managed by the flow control mechanism.

Establishing a Link

Upon establishing a link, a specialized link socket, `PubLinkSocket`, is created and associated with it. This link socket extends the link socket with ordering guarantees of the configurable socket, to ensure FIFO ordering of both data and control messages.

The need for this link socket specialization is for synchronization purposes. Before publishing a message, the socket must reserve a credit in every destination. Only after successfully making a reservation for every link socket, can it begin sending the message to those destinations.

Closing a Link

When a link is closed, the system ceases monitoring its write events. Additionally, the subscriber is automatically unsubscribed from all previously registered topics, and any ongoing attempts to making a reservation for message transmission are canceled.

Publishing a Message

When publishing a message, the socket attempts a synchronized send across all subscribers of the topic. Synchronization ensures that once a message is sent to one of those subscribers, it must also be sent to

the others. If a timeout is provided, the operation will only timeout if at least one subscriber is unable to receive the message due to a lack of transmission credit.

The publisher socket provides a custom sending method, `send(PSPayload, timeout : Long)`, which takes a custom payload object (`PSPayload`) that separates the message topic from its actual content.

The sending algorithm is as follows:

1. Before sending, the client thread processes any pending subscription tasks.
2. The socket retrieves all subscriptions whose topics are prefixes of the message's topic.
3. It then iterates over all subscribers of these subscriptions, attempting to reserve a credit for each. If reservations succeed for all subscribers, the socket consumes the reservation and transmits the message for each subscriber. If making all reservations within the given timeout fails, any reservations made are canceled.

Since sending is blocked if any subscriber lacks credits, this algorithm effectively synchronizes message delivery to match the slowest subscriber.

Regarding ordering guarantees, these depend on the number of threads publishing to topics:

- If a single thread publishes, global order is maintained.
- If each thread publishes to a specific topic, order is guaranteed within that topic.
- If multiple threads publish to the same topic, no ordering guarantees exist.

5.4.2 Subscriber Socket

The Subscriber Socket (`SubSocket`) subscribes to topics by communicating its interests to linked publishers and awaiting messages related to those topics.

Compared to the Publisher Socket, which handles most of the messaging pattern logic, this socket has a much simpler implementation. It extends the configurable socket, leveraging its reading and ordering capabilities, with subscription management as its only additional logic.

While publishers are responsible for managing subscriptions, subscribers must also track their own subscriptions to ensure publishers remain up-to-date with their interests.

Managing subscriptions

To receive messages related to topics of interest, a subscriber must communicate those interests to linked publishers. For this purpose, the socket provides two methods: `subscribe(topic : String)` and `subscribe(topics : List<String>)`. These methods register the topics locally, create a subscription message, and forward it to all linked publishers.

Given the possibility of sharing interests, the subscriber must also be able to withdraw them. Two variants of the `unsubscribe()` method allow unsubscribing from a single topic or a list of topics. These methods remove the topics from local registration, generate an unsubscribe message, and send it to the linked publishers.

Establishing a Link

When a link is established, the socket uses its locally registered topics to notify the newly linked publisher of its interests.

Receiving a Message

The socket provides the `recv(timeout : Long, notifyIfNone : boolean) : PSPayload` method for receiving messages, allowing the receiver to identify the topic associated with the received payload.

5.4.3 X-Publisher Socket

The X-Publisher Socket (`XPubSocket`) extends the Publisher Socket, providing an additional method, `recv(timeout : Long, notifyIfNone : boolean) : SubscriptionsPayload`, which exposes received subscription messages to the application.

This socket is typically used alongside the X-Subscriber Socket to create a broker that connects subscribers with publishers while keeping the publishers' locality transparent to subscribers, and vice versa. An X-Publisher Socket links with Subscriber Sockets, exposing their subscription messages so they can be handed over to the X-Subscriber Socket, which then shares them with its linked publishers.

5.4.4 X-Subscriber Socket

The X-Subscriber Socket (`XSubSocket`) extends the Subscriber Socket by adding a new method, `send(SubscriptionsPayload)`, which enables forwarding subscription messages. This method

also interprets the message as an invocation of the corresponding `subscribe()` or `unsubscribe()` method, thereby locally registering or removing the topics contained in the message.

5.5 Multipart Messages - Future Modification

Developing an approach that enables the transmission of multipart messages, like ZeroMQ but compatible with the thread-safety requirements, could enhance the middleware's usability for client applications but also facilitate the creation of new socket types.

Chapter 6

Exon - Adaptation and Modifications

The `Exon` library is a critical component of messaging middleware. This transport protocol was initially selected to provide the exactly-once delivery guarantee. As an open-source project, its implementation could be modified to meet integration requirements.

In this chapter, we describe the modifications made to this transport library, adapting it to be compatible with the messaging middleware requirements.

6.1 Support for Mobility Scenarios

Supporting mobility scenarios allows nodes to seamlessly switch between different networks while maintaining reliable communication. Middleware solutions like `NNG`, `nanomsg`, and `ZeroMQ` – the most similar to this middleware solution – natively lack this feature, especially due to their predominant use of the TCP protocol for reliability, but also because they lack a way to integrate a discovery service.

Since the `Exon` library utilizes the `UDP` protocol, which is connectionless, the issue of connection breaks disrupting communication reliability – like with TCP – does not arise. By binding the `UDP` socket to a wildcard address[1], instead of a specific IP address, the device can receive messages on any available network interface in addition to not enforcing the received traffic to have a specific destination address in order to be accepted. If the device switches networks and obtains a new IP address, communication can continue as long as other nodes detect the change and update their destination addresses accordingly.

To provide support for mobility scenarios, the following changes were required:

- Use of globally unique identifiers instead of transport addresses for node identification.
- Mapping of globally unique identifiers to transport addresses.
- Inclusion of sender and receiver identifiers in every message.

[1] `0.0.0.0` in IPv4.

The implementation of these changes is discussed in detail below.

6.1.1 Node States

Before jumping to the explanation of the changes, it is essential to refer to some aspects related to the nodes' states.

As briefly mentioned in (Exon Protocol and Exactly-Once Delivery), for each payload[2], a 4-way exchange is required to deliver it:

1. The sender requests a slot.
2. The receiver allocates the slot and sends it to the sender.
3. The sender stores the slot as an envelope, creates a token using the envelope and the payload, and sends the token to the receiver.
4. After receiving the token, the receiver acknowledges the reception of the message and discards the associated slot.
5. After receiving the acknowledgement, the sender discards the token.

These exchanges require both the receiver and the sender to hold state related to the exchange. As we can observe in the example 38, the state is divided into a *send record* (SR) and a *receive record* (RR), with each record being mapped with the identifier of the remote node as the key [3]. A send record holds a queue for payloads without an associated envelope, a list for unused envelopes, and a set for tokens[4]. The receive record is responsible for keeping track of slots for which a client message has not yet arrived.

It is clear that the sender's send record and receiver's receive record operate together. To ensure the correctness of the algorithm – i.e., to provide exactly-once delivery guarantee – it is fundamental for a received message to be matched with the proper record. This is because messages are formed with the state of the counterpart record in mind.

To better understand the problems posed by matching a message improperly, let us examine an example similar to the one presented in the figure 38. Considering the a node B's receive record with some allocated slots, $[0..9]$, and its counterpart send record (of node A) with a token, $0 \rightarrow$ "Hi!", to be delivered. Let's now assume that node A changes to a different address after receiving the slots from B

[2] A client message is referred to as *payload* to avoid confusion with `REQSLOTS`, `SLOTS` and `ACK` messages.

[3] The original implementation of Exon uses transport addresses as node identifiers, as seen in the figure, by looking at the keys used to map the records.

[4] Remember that tokens are the combination of an envelope and a payload

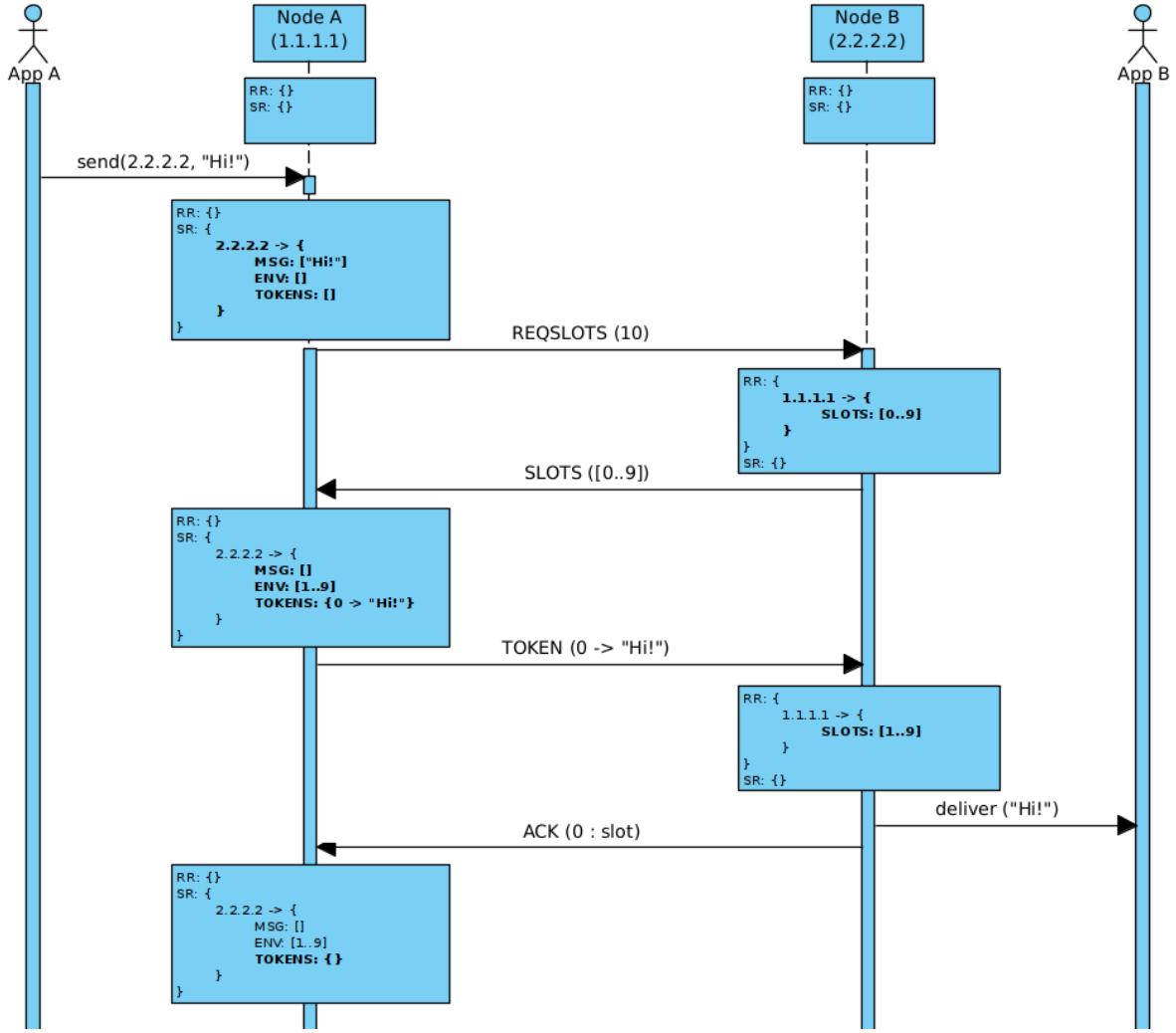


Figure 38: Example of 4-way exchange

but before sending the token. Since the original implementation of Exon uses transport addresses as the identifiers to map the records, the following situations are possible when node B receives the token from A:

- The transport address of the datagram that carries the token does not have a matching receive record.
- The transport address of the datagram that carries the token matches the receive record of another node that had its transport address changed.

Both the situations described above present significant issues. Messages are created with specific attributes tailored to the corresponding record, such as clocks, slots, etc. Consequently, when incoming messages are not matched with the appropriate record, either the message is perceived as incoherent and is thus discarded, or it corrupts the record. With "corruption," meaning that a message not intended for the matched record alters the state of the record. For instance, let's consider a `TOKEN` message that is perceived as coherent by the receive record, but it was not intended for that specific record. Let us also assume that the token's envelope matches a slot of the receive record. Since the message is perceived as valid, an unintended payload will be delivered to the application. Furthermore, the slot will be consumed, which obstructs the delivery of the actual payload that should have utilized the slot[5].

In conclusion, ensuring that messages are consistently matched with the appropriate records is the foundation for supporting mobility scenarios, as it results in the correct behaviour of the Exon algorithm.

For simplicity, the following sections will merge the concept of send records and receive records into only "state." A state mapped to a node X implies the presence of either a send record, a receive record, or both. Additionally, the type of the messages and the content of the states will not be presented as the examples would become too extensive.

6.1.2 Globally Unique Identifiers

The original Exon library uses transport addresses as node identifiers. This is a valid solution when nodes are not expected to switch to different addresses, however, when nodes do change, this solution can no longer ensure exactly-once delivery. Furthermore, the address change may result in the inability to communicate with nodes for which the node already has an associated state.

To support mobility scenarios, transport addresses cannot be used as node identifiers since switching to a new transport address means being perceived as a new node. Consequently, the nodes with which

[5] A payload within a token is only delivered to the application when the token's envelope has a corresponding slot present in the receiving record.

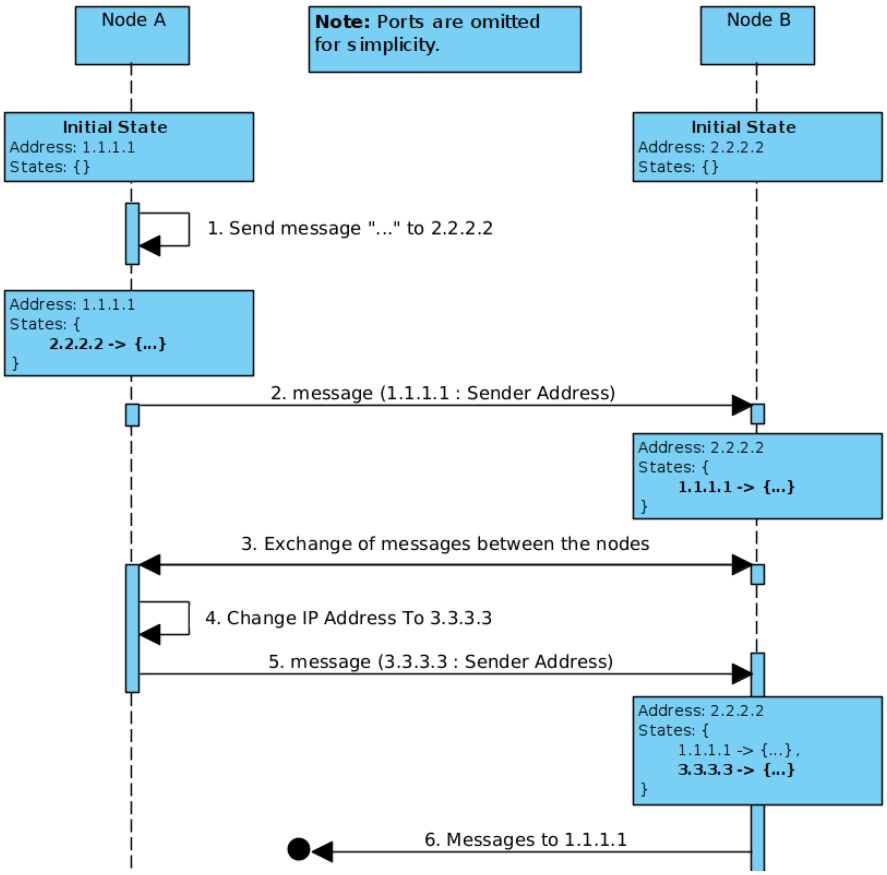


Figure 39: Transport Addresses as Node Identifiers in a Mobility Scenario

communication was being held will create and use a new state instead of continuing the conversation using the state established prior to the address change. Additionally, these nodes will not be able to deliver pending messages from the previous state to the moved node. The diagram in Figure 39 illustrates these issues.

To solve this problem, nodes are now identified with globally unique identifiers, allowing correct matching of messages with communication states regardless of the transport address in use.

6.1.3 Associations

While nodes will now be addressed by arbitrary identifiers, transport addresses are still required for communication, more specifically, to send a message. Therefore, associations between node identifiers and transport addresses (IP address + Port) are needed.

There are several ways to discover associations:

- **Local manual registration:** A new method of the API, `registerNode(nodeId : String, taddr : TransportAddress)`, is provided to manually register associations in a local struc-

ture.

- **Query Discovery Service:** A discovery service, also called *Association Source*, can be contacted to determine the transport address related to a known node identifier. To avoid repeatedly querying the discovery service, retrieved associations are stored in a local structure. The new API enables the registration of a discovery service via `setDiscoverService(DiscoveryService)`.

- **First Contact with Node:** An association can also be discovered, and then registered locally, upon being contacted by another node. Every message carries the necessary information to create/update an association. The Exon message (UDP datagram's payload) carries the *source node identifier*[6], and the UDP datagram carries the source IP address and port, which combined define the source transport address.

Notice that associations can be updated when messages are received from other nodes. This enables an immediate perception of a node's transition to another transport address and, consequently, an adjustment of the transport address to ensure that any messages sent in the future are properly routed.

The figure 40 demonstrates arbitrary identifiers supporting a mobility scenario.

6.1.4 Exon Messages and Node Identifiers

To support mobility scenarios, another modification to the original library was required. Every message[7] must include the identifiers of the source and destination nodes. We will now dive into the reasons behind requiring each of these identifiers.

Source Node Identifier

Including the source node identifier in every message has multiple purposes:

1. Enables creating an association between the sender's node identifier and its transport address, avoiding the need for registering every node in a discovery service. Registering static nodes (servers) is enough.

   Let's consider that Exon messages do not carry the sender's identifier. When a node receives a message, it must find the corresponding node identifier. If the identifier is not found locally nor on a discovery service (by querying with the transport address), the receiver is forced to discard the

[6] The reason behind every message carrying the node identifier will be explained ahead.

[7] `REQSLOTS`, `SLOTS`, `TOKEN` and `ACK` messages include these identifiers.

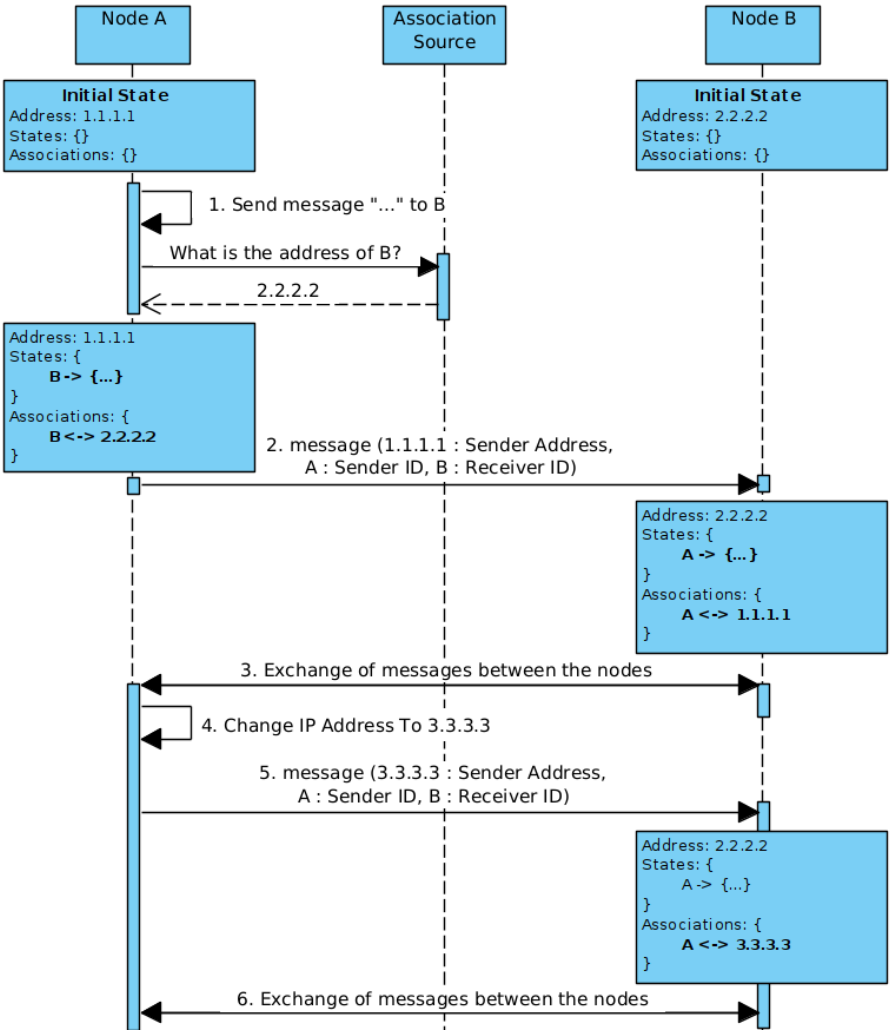


Figure 40: Arbitrary Node Identifiers in a Mobility Scenario

message (Section A of 41), as it cannot risk creating a state mapped to an identifier that does not correspond to the sender. If a node exists with the attributed identifier, the states of the two nodes would overlap.

By including the sender's identifier, the receiving node has all the necessary information to form the association, thus making it possible for ephemeral nodes (clients) to communicate without being previously registered in a discovery service.

2. Ensures that when a message is received, it is matched with the corresponding state.

   Since a state record is mapped using the remote node's identifier, by having messages include the source's identifier, regardless of whether a change in transport address occurred, the messages can be properly matched with the corresponding state, avoiding state "corruption" of another node's state.

   Suppose sender identifiers are not incorporated in every message. When receiving a message, it is necessary to translate the transport address to a node identifier by consulting the local structure or the discovery service. If the retrieved association is outdated, corruption might happen. Section B of figure 41 illustrates a situation where *node A* switches to the previous address of *node C* before the discovery service is notified of the address change that *C* experienced. As *B* is not yet aware of this change, messages received from *A* are matched with *C*'s state, potentially resulting in its corruption.

3. Allows communication to be resumed as quickly as possible after a node switches its transport address.

   With all messages possessing the required information to update a node's association, a change of transport address can be detected as soon as a message sent with the new address is received.

The figure 41 presents a scenario that demonstrates the need for the source node identifier within Exon messages. Keep in mind that carrying the source node identifier allows receivers to immediately update the sender's association so that outgoing messages can be routed correctly.

Destination Node Identifier

Including the destination node identifier enables nodes to discard messages not intended for them, thereby preventing state corruption. Returning to the scenario presented in figure 41, after node *A* switched to the previous address of *C*, messages from *B* addressed to *C* arrived at *A*, potentially corrupting *A*'s state

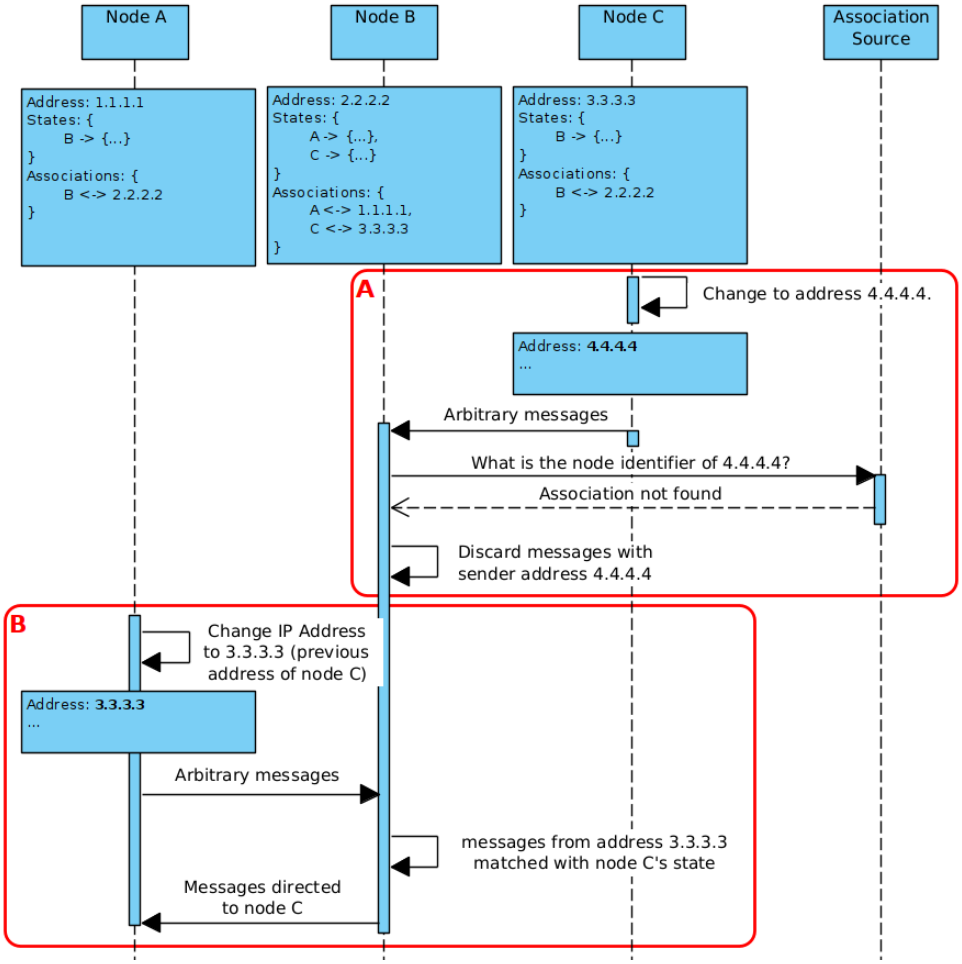


Figure 41: Source Node Identifier Requirement

associated with node *B*. To prevent processing of messages not directed to the node, the identifier of the destination node is included in all messages.

6.2 Self-messaging

Another modification made to the Exon library was done to bypass the exactly-once transport protocol when delivering messages to itself. When considering transport abstraction, like for the mobility scenario support, this feature should have been implemented as part of the middleware to avoid the need for re-implementation if swapping the transport library is desired in the future.

6.3 Closing Procedure

A proper closing mechanism is essential for any system to shut down gracefully. While an attempt was made to implement such a procedure in Exon, it was not entirely successful.

An instance of the Exon middleware cannot be closed arbitrarily due to the exactly-once delivery requirement. To provide this guarantee, nodes exchanging messages hold complementary state records, therefore, if a node discards its state prematurely, the correctness of the algorithm is compromised.

Since Exon is oblivious – meaning it discards unused state records automatically – a closing mechanism was designed around this behavior. Before closing, a node must wait for all its state records to disappear, ensuring no ongoing communication. At this point, closure is safe from violating the algorithm's correctness as long as the node's clock is persisted, which is required to resume communication without breaking the exactly-once delivery guarantee. This approach avoids persisting state records, as it was specifically designed for ephemeral clients, allowing them to close after completing their tasks.

This solution works well for ephemeral nodes that communicate only with static nodes. However, in a peer-to-peer scenario where both nodes eventually wish to close, it might fail. The issue arises when one node cannot discard its receive record associated with the other node. When no messages remain to be sent, the send record is removed and a `REQSLOTS` message is sent to nullify any unused slots at the receiver, thus enabling the receive record to be discarded. The problem is that if this message is lost and its sender has closed, the destination node will remain in an indefinite waiting state, periodically sending `SLOTS` in an attempt to "request" a `REQSLOTS` (which would allow discarding the receive record).

Without modifying the protocol to confirm the successful removal of receive records, under a peer-to-peer scenario, a successful closure is not guaranteed.

Even with such a modification, a closed node might still be selected as a message destination, making

reliable closure not achievable under the current protocol design. A potential, but impractical solution could involve explicit node de-registration from all nodes that may attempt to communicate with it.

For the reasons highlighted above, the closing procedure is not employed, as it does not guarantee a safe and reliable shutdown under the exactly-once delivery requirement.

Chapter 7

Usage Patterns

This chapter explores various ways to utilize the middleware and how it facilitates communication over other messaging middleware solutions such as ZeroMQ.

7.1 Exactly-Once Delivery and Mobility Scenarios

One of the key advantages of this middleware is its ability to guarantee exactly-once message delivery. This ensures that clients can focus solely on sending and receiving messages without needing to implement additional mechanisms to ensure message delivery reliability.

For instance, in unicast communication using ZeroMQ, the underlying transport protocol is typically TCP. While TCP ensures exactly-once delivery within an active connection, any messages in transit are lost if the connection breaks. As a result, ZeroMQ clients must account for this limitation and implement a message persistence mechanism to prevent data loss. In contrast, our middleware eliminates this concern – once a message is submitted, it is guaranteed to be delivered exactly once.

Another major advantage of this middleware is its support for mobile devices that transition between different networks in the midst of communication, as illustrated in Figure 42.

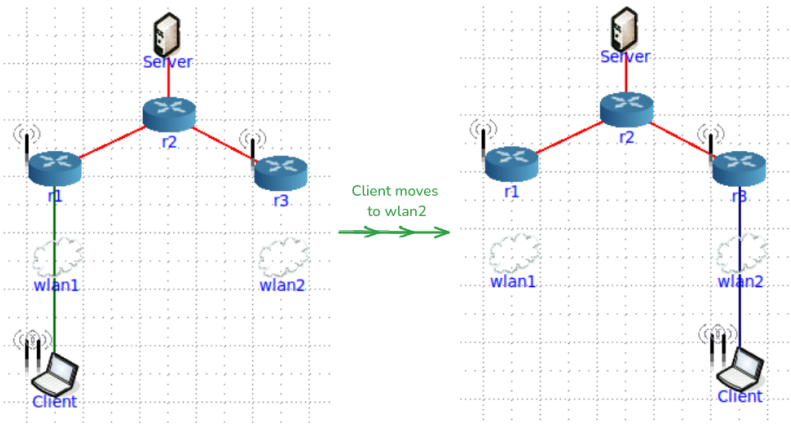


Figure 42: Client node transitioning between networks

To enable mobility, a middleware instance must be initialized with a wildcard IP address. This can be achieved by either specifying `null` or explicitly setting the wildcard address (”`0.0.0.0`” for IPv4 and ”`::`” for IPv6). If the constructor does not take an IP address parameter, the middleware uses a wildcard address as default.

In addition to setting a wildcard address, discovering relevant nodes is crucial. The recommended approach is to use a discovery service rather than manually registering node associations.

Below is an example demonstrating how a client can be easily configured, without requiring additional user-defined functionality to ensure reliable delivery and account for mobility:

```
// Initialize and configure the middleware instance.
// Assigns "Client" as the node identifier and binds to port 11111.
A3MMiddleware a3m = A3MMiddleware.startMiddleware("Client", 11111);
MyDiscoveryService ds = new MyDiscoveryService(...);
a3m.setDiscoveryService(ds);

// Create a socket with "Requester" as the tag identifier
ReqSocket reqSocket = ReqSocket.startSocket(a3m, "Requester");

// Establish a link with the socket
// identified as "Server-Responder".
reqSocket.link("Server", "Responder");

while(...){
    // Construct and send a request
    byte[] payload = ...;
    reqSocket.send(payload);

    // Receive and process the response
    payload = reqSocket.receive();
    processReply(payload);
}
```

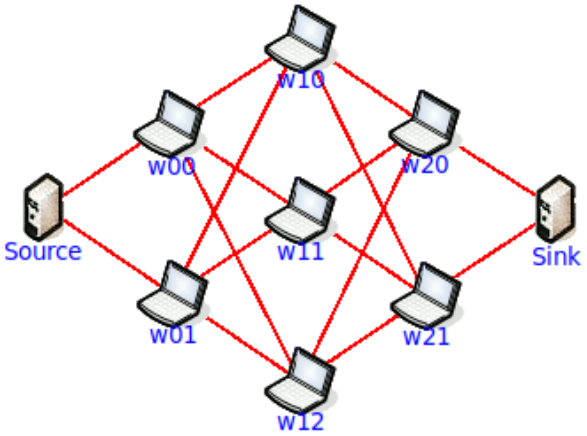


Figure 43: Scatter-Gather Topology Example

7.2 Concurrent Communication and Multiplexing

One of the advantages of this middleware is its efficient socket creation mechanism. In other middleware solutions, creating a socket typically involves creating a corresponding physical socket. In contrast, this middleware associates sockets with the middleware instance itself, allowing socket creation to happen at user-level. This approach not only speeds up the creation process but also contributes to overall resource efficiency.

This capability is particularly useful for managing multiple message flows and enabling concurrency. For instance, consider a scenario where multiple threads need to send requests and wait for their respective replies. Instead of implementing a demultiplexer that waits for responses and forwards them to the correct thread, a separate Request Socket (or Dealer Socket) can be instantiated and assigned to each thread. This approach ensures independent message flows and inherently acts as a demultiplexer.

7.3 End-of-Communication Detection

The middleware provides functionality that enables implicitly detecting the end of communication. To illustrate this, we present a scatter-gather example, depicting how it is implemented in ZeroMQ versus this middleware.

Consider a scatter-gather topology (like the one shown in Figure 43), where messages originate from a source, pass through rows of workers, and finally reach a sink that collects the results. We assume that each row of workers is responsible for applying some transformation before passing the message forward.

To implement this example, the following sockets are required:

- The Source uses a Push Socket to emit messages.
- The Sink uses a Pull Socket to receive messages.
- Each worker requires both a Pull Socket to receive messages from the previous stage and a Push Socket to send messages to the next stage.

A common challenge in this communication pattern is determining when no more messages will be received, allowing the execution of any necessary final actions.

In ZeroMQ, this detection is typically handled by having the source inform the sink of the expected message count. However, this approach requires additional sockets and coordination.

In contrast, this middleware does not need those extra requirements. The detection can be achieved by monitoring when a socket becomes *link-free*. When receiving messages, setting the `notifyIfNone` flag enables the thread to declare its intent to cancel the operation once the socket is detected as having no links. This allows executing appropriate termination actions without requiring explicit message counting.

Since ZeroMQ does not provide a built-in mechanism to detect when all connections are closed, a message counter must be obtained preemptively. This prevents the receiving thread from blocking indefinitely when no more messages are expected.

The following code segments demonstrate how the source, worker, and sink nodes can be implemented using this middleware, leveraging the link-free notification as an implicit end-of-file (EOF) signal.

7.3.1 Source Node Code

```
// Create source socket
PushSocket socket = PushSocket.startSocket(a3m, "Source");

// Establish links with next stage
for(SocketIdentifier sid : nextStageIds)
    socket.link(sid);

// Send messages
int nrMsgs = 100;
for(int i = 0; i < nrMsgs; i++){
    byte[] payload = ...;
    socket.send(payload);
```

```
}

// Close socket
socket.close();
```

Notice that there is no explicit method to wait for link establishment before sending messages. While this approach is valid, it is unnecessary in this case because the Push Socket supports link-free verification. By default, when such a mechanism is available, the notifyIfNone flag is set to false. Consequently, the send method only returns after the message has been sent over an established link. If no links are available, the operation is not canceled, instead, the method waits for a link to be established before sending.

Once all messages have been sent, the socket is no longer needed, so `close()` is called. This triggers the unlinking process for all links at once, eliminating the need for individual `unlink()` calls. Closing the socket does not affect delivery reliability, as it does not close until all links are closed and links do not close until all messages have been successfully delivered to their destination applications.

By linking and unlinking in the direction from the source to the sink, and only unlinking when no more messages are going to be sent, the event of a worker's or sink's Pull Socket becoming link-free after having established at least one link serves as an implicit end-of-file (EOF) signal, indicating that no further messages will be received. Consequently, the end of communication can be detected without requiring a message counter to be sent preemptively.

7.3.2 Worker Node Code

```
// Create sockets
PullSocket recvSocket = PullSocket.startSocket(a3m, "Pull");
PushSocket sendSocket = PushSocket.startSocket(a3m, "Push");

// Establish links with next stage
for(SocketIdentifier sid : nextStageIds)
    sendSocket.link(sid);

// Wait for a link to be established with the previous stage
recvSocket.waitForAnyLinkEstablishment();
```

```
// Receive, process and forward messages until link-free
byte[] payload;
Integer timeout = null; // for blocking receive
boolean notifyIfNone = true; // to receive link-free notification
try{
    while(true){
        payload = recvSocket.receive(timeout, notifyIfNone);
        payload = processPayload(payload)
        sendSocket.send(payload);
    }
} catch(NoLinksException ignored){
    // Interpret as EOF because all links with
    // the previous stage were explicitly closed.
}

// Close sockets
recvSocket.close();
sendSocket.close();
```

To clarify why a link-free event can be viewed as an end-of-file, consider the example of the source interacting with the first set of workers.

At the source:

1. The source initiates the link establishment process with all workers from the next stage.
2. Messages are distributed to the workers as their links are established.
3. After all messages are sent, the source starts the unlinking process with the workers. The unlinking process does not complete until all sent messages are successfully delivered to the workers for processing.

At the worker:

1. The worker begins the linking process with the next stage, as a preparation to forward processed messages. The middleware handles the linking process asynchronously – invoking `link()` merely initiates it.

2. The worker explicitly waits for a link to be established with the previous stage (the source). This step is necessary because the `notifyIfNone` flag is used when receiving messages. If the worker proceeds directly to the loop, the link with the source is likely not yet established, which could cause the worker to mistakenly interpret the Pull Socket's link-free notification as an EOF signal.

3. Once the link is established, regardless of the amount of messages received, processed, and sent forward, the worker considers its job complete when the Pull Socket becomes link-free (i.e., the link with the source is closed). At this point, the worker initiates the unlinking process for the links with the next stage, thereby propagating the EOF signal when the Pull Sockets of the next stage become link-free.

As an interesting point, the fact that send operations are blocking, and that new messages are not accepted when sending is not possible, ensures that earlier stages are effectively throttled by the stages that follow.

7.3.3 Sink Node Code

```
// Create sink socket
PullSocket socket = PullSocket.startSocket(a3m, "Sink");

// Wait for a link to be established with previous stage
socket.waitForAnyLinkEstablishment();

// Receive and process.
byte[] payload;
Integer timeout = null;
boolean notifyIfNone = true;
try{
    while(true){
        payload = socket.receive(timeout, notifyIfNone);
        payload = processPayload(payload)
    }
} catch(NoLinksException ignored){
    // EOF signal
```

```
    }

    // Do any final processing that may require all messages
    ...

    // Close socket
    socket.close();
```

The behavior of the sink is similar to that of the workers, with the exception that it does not forward messages after processing them.

Chapter 8

Performance Analysis

In this chapter, two tests will be presented, enabling a performance comparison between the devised solution, A3M, with the underlying transport library, Exon, and a widely used messaging middleware, ZeroMQ.

The Exon transport library is included in this performance analysis to enable the verification of the overhead introduced by the messaging middleware logic, but also to outline the performance limits the A3M middleware could exhibit since the middleware itself is not supposed to exhibit better performance than the underlying transport protocol.

The ZeroMQ middleware is included for comparison purposes, enabling to verify if the solution exhibits an acceptable performance, so that it can be considered as a viable option to facilitate the development of distributed applications.

The tests where performed in a single machine (localhost) characterized by a Intel® Core™ i7-12650H and 16GB DDR4 RAM, running 22.04.5 64-bits Ubuntu OS.

For both tests, two instances of each middleware were used, each in different processes.

8.1 One-way Throughput

The One-way Throughput test is used to determine how fast data can be pushed by each middleware instance.

This test was performed using PUSH-PULL sockets, with only one sender thread and one receiver thread (in addition to the threads required by each middleware).

The throughput was then measured for different message sizes: 1, 10, 100, 500 and 1000 bytes.

From figure 44 we can verify that Exon exhibits a higher throughput (around 22.9%) than A3M. This is expected as A3M needs to perform additional high-level logic, even though only a single socket is connected, and the messages have extra data not required by Exon.

The difference from A3M and Exon to ZeroMQ is highly accentuated for smaller message sizes and gradually reduces to a throughput higher but less exaggerated to those of Exon (29% higher) and A3M (90% higher) for a message size of 1000 bytes. This is believed to be the job of a batching mechanism that groups smaller messages together. The lack of this characteristic was highlighted in Chapter 3 as a limitation of Exon. By implementing such functionality in the future, we believe this huge difference can be mitigated.

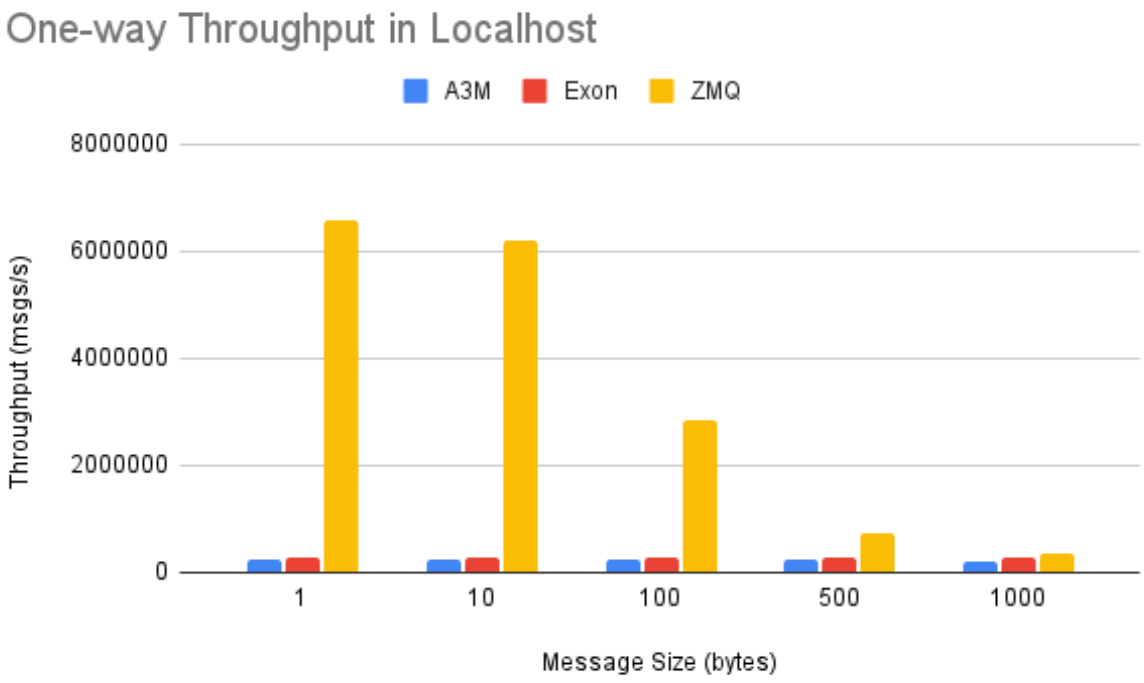


Figure 44: One-way Throughput in Localhost

8.2 RPC Throughput

The RPC throughput test measures how many round-trip calls a system can handle per second. Contrarily to the one-way throughput test, this test is affected by the latency, although, in local host, the latency is most likely negligible. Since this is latency-based, the average throughput (RPCs/s) is collected by each sender individually and then aggregated.

As can be perceived in 45, the RPC throughput is measured for different amounts of clients. As Exon is a basic transport library, not having the notion of sockets, the test case was simplified with multiple threads (clients) using the same middleware instance to send requests and retrieve the first message that they can regardless of it corresponding to their original request. As for ZeroMQ and A3M, clients use REQ

sockets while the server (receiver) uses a REP socket.

To avoid having the results capped by the bandwidth, the messages utilized have a size of 1 byte, allowing to focus on the actual performance of the messaging middleware when performing logic related to sending and receiving.

As can be perceived, the throughput begins to plateau around the 10-client mark. At this point, ZeroMQ exhibits a better performance than Exon and A3M by approximately 67%. Also, the 10-client mark coincides with the system's 10-core configuration, suggesting that the aggregated RPC throughput can scale with the number of available CPU cores.

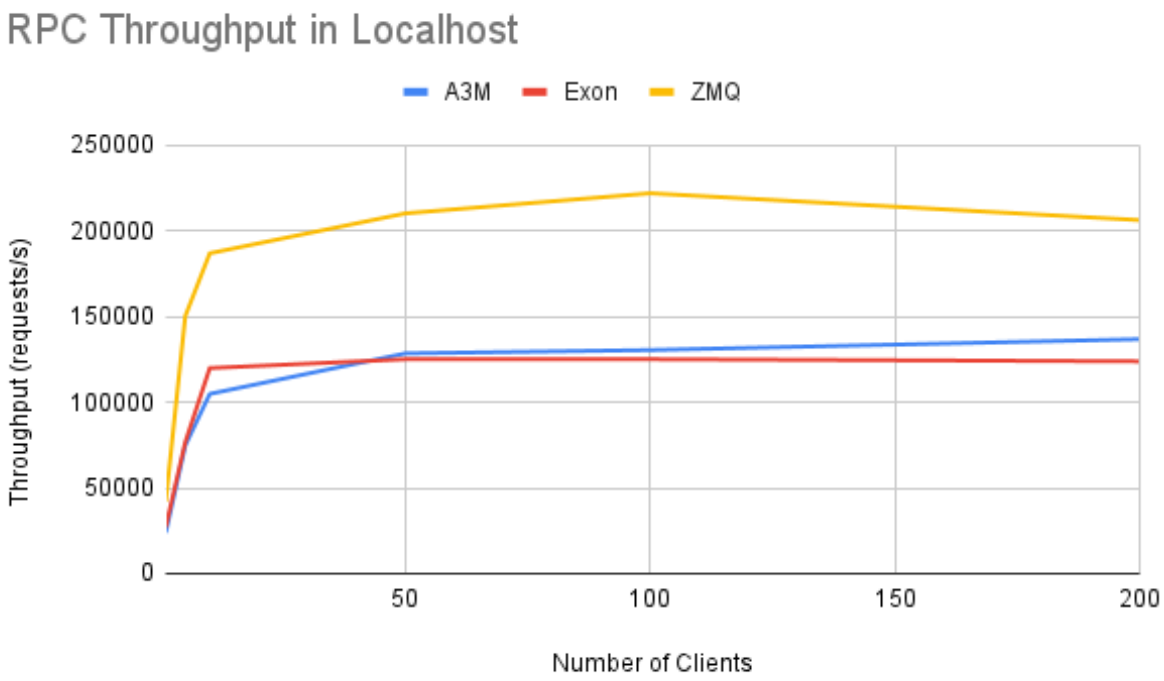


Figure 45: RPC Throughput in Localhost

Chapter 9

Conclusions and future work

9.1 Conclusions

This thesis addressed the challenges that surround designing and implementing a messaging middleware that offers exactly-once delivery guarantee. The main objective was to overcome limitations identified in existing messaging middleware, such as ZeroMQ, by developing a solution that prioritizes reliability, usability and adaptability over performance.

By being built on top of a transport protocol that ensures exactly-once semantics, the middleware enables reliable communication that is resilient to different network scenarios, such as mobility and partitions, which are usually situations where high-performance solutions, like ZeroMQ, often fail due to TCP's limitation. However, further research is necessary to identify middleware features required for a safe deployment in real-world scenarios, particularly in the presence of malicious nodes capable of executing denial-of-service (DoS) attacks by exploiting the retransmission mechanism[1].

This work also demonstrates the middleware's usability and flexibility, introducing features such as thread-safe sockets, support for custom socket development, link management, and lightweight virtual socket creation for concurrent communication flows. These features simplify and enrich the development of distributed systems, while maintaining acceptable performance levels.

In summary, the thesis contributes with a reliable and extensible middleware that eases the creation of fault-tolerant distributed applications. It also outlines future work with the goal of enhancing the middleware's capabilities and increasing its applicability in real-world scenarios.

[1] To ensure exactly-once delivery, messages are retransmitted until their delivery is acknowledged by the destination. That said, a malicious client may send a request and immediately remove itself, leaving the server wasting resources with indefinite replies. With a sufficient amount of requested messages, the server can crash.

9.2 Prospect for future work

To conclude, this section presents several areas that were either not addressed totally or partially in the current prototype. Addressing these aspects could significantly enhance the middleware's capabilities and overall value.

Transport Abstraction While the middleware was designed with extensibility in mind – particularly for socket behavior – introducing transport abstraction would widen its applicability to scenarios with varying levels of delivery guarantees.

Although the default sockets are currently tailored for exactly-once semantics, custom sockets designed with other delivery guarantees in mind can already be implemented. Pairing socket customization with transport abstraction would give developers the flexibility to choose a combination that best fits their needs.

Achieving this abstraction would require decoupling the middleware from the Exon library. Notably, mobility-related functionality currently within Exon would need to be extracted and incorporated into the middleware, where it arguably belongs.

To achieve this abstraction, decoupling of the middleware from the Exon library would be necessary. In specific, mobility-related behavior would have to be extracted to the middleware.

Graceful Closing Procedure A graceful closing mechanism is generally desirable in middleware systems. However, under strict exactly-once delivery semantics, a true graceful closing is not possible, as in-transit messages may be missed resulting in indefinite retries from the sender.

With a clear distinction of client (ephemeral) and server (static) nodes, and a proper communication flow with the client sending the last message as a signal for termination, client nodes could terminate safely. For a server node to also close, it would need to know the whole topology and enter in a contract with the clients as to delineate that it would be leaving and not returning.

Nevertheless, implementing a temporary shutdown procedure is a possible solution which does not require full knowledge about the topology. The node would need to persist all state information needed to resume communication without the violating exactly-once delivery guarantee.

State Persistence Although outside the scope of this thesis, which focuses on network fault tolerance, adding state persistence would further enhance the middleware's resilience by enabling recovery from crash faults.

In addition to improving overall robustness, persistent state would support the implementation of

temporary shutdowns and even allow the relocation of a middleware instance between devices if needed.

Presence Service Introducing a presence service can be used to optimize network resource usage. A presence service is similar to a discovery service but event-driven. It would allow nodes to be notified when other nodes become available, reducing the overhead caused by frequent message retries or busy polling to a discovery service.

However, implementing this would currently require modifications at the transport layer – specifically in Exon, since message retransmissions are handled there. Alternatively, the exactly-once delivery guarantee would need to be provided by a transport-independent library which would then delegate address resolution and message routing a separate layer.

Performance Optimization As the test results reveal, the current prototype does not offer great performance compared to existing solutions (ZeroMQ). Future work should include detailed profiling to identify bottlenecks and improve performance.

The last profiling done suggests that message creation/parsing and event notification are potential areas for optimization. Furthermore, the middleware's internal thread appears to be a bottleneck. Investigating whether this is due to excessive locking, high workload, or other factors is essential for improving throughput and scalability.

Monitoring and Logging Monitoring and logging are essential for diagnosing errors, verifying performance, and identifying scaling opportunities. A possible solution could involve reserving specific socket identifiers (e.g., those which start with a special character, such as "$" in MQTT) for internal purposes such as statistics, etc.

Transport-related features Some of the improvements suggested initially for the Exon library could be implemented at the middleware level. This would allow reusing them reusable with different transport protocols. Some relevant features include automatic flow control, message fragmentation and merging, and integrity verification mechanisms.

Security features Finally, incorporating security features such as encryption and authentication would significantly increase the middleware's applicability, particularly in sensitive or hostile environments.

Bibliography

Alexandre Martins. A3m, 2025. URL https://github.com/Alex-Sa-Ms/A3M-Messaging_Middleware_With_EO/.

Roger Light. Understanding MQTT Quality of Service or also known as MQTT QoS. https://cedalo.com/blog/understanding-mqtt-qos/, 2024.

Ziad Kassam, Paulo Almeida, and Ali Shoker. Exon Exactly-Once Oblivious Messaging Library. https://github.com/ziadkassam/Exon, 2021.

nanomsg. About nanomsg. https://nanomsg.org/, 2018.

Martin Sustrik. The Architecture of Open Source Applications (Volume 2) ZeroMQ. https://aosabook.org/en/v2/zeromq.html, 2012.

ZeroMQ. ZeroMQ Message Transport Protocol RFC. https://rfc.zeromq.org/spec/23/, a.

ZeroMQ. ZeroMQ RFCs. https://rfc.zeromq.org, b.

ZeroMQ. ØMQ - The Guide. https://zguide.zeromq.org, c.

Martin Sustrik. Getting Rid of ZeroMQ-style Contexts. https://250bpm.com/blog:23/, 2013.

nanomsg. nanomsg GitHub Repository. https://github.com/nanomsg/nanomsg, a.

Martin Sustrik. All articles. https://250bpm.com/toc/index.html, 2011-2022.

Martin Sustrik. Why should I have written ZeroMQ in C, not C++ (part I). https://250bpm.com/blog:4/index.html, 2012.

nanomsg. nanomsg Manual Page. https://nanomsg.org/v1.1.5/nanomsg.html, b.

nanomsg. Differences between nanomsg and ZeroMQ. https://nanomsg.org/documentation-zeromq.html, c.

Martin Sustrik. Optimising Subscriptions in nanomsg. https://250bpm.com/blog:19/index.html, 2013a.

nanomsg. Bus (Routing). https://nanomsg.org/gettingstarted/bus.html, d.

Garrett D'Amore. NNG Git Repository. https://github.com/nanomsg/nng.

Garrett D'Amore. Rationale: Or why am I bothering to rewrite nanomsg? https://nng.nanomsg.org/RATIONALE.html, 2020.

NNG. NNG: Lightweight Messaging Library. https://nng.nanomsg.org/, 2024.

HiveMQ. MQTT - Essentials. https://www.hivemq.com/mqtt/, 2015a. Last Update: 2023.

OASIS Open MQTT Technical Committee. MQTT Version 3.1.1. https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html, 2014.

HiveMQ. What is MQTT Quality of Service (QoS) 0,1, & 2? – MQTT Essentials: Part 6. https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/, 2015b. Last Update: 2023.

HiveMQ. Understanding Persistent Sessions and Clean Sessions – MQTT Essentials: Part 7. https://www.hivemq.com/blog/mqtt-essentials-part-7-persistent-session-queuing-messages/, 2015c. Last Update: 2023.

HiveMQ. What Is MQTT Keep Alive and Client Take-Over? – MQTT Essentials Part 10. https://www.hivemq.com/blog/mqtt-essentials-part-10-alive-client-take-over/, 2015d. Last Update: 2023.

Ziad Kassam, Paulo Almeida, and Ali Shoker. Exon: An Oblivious Exactly-Once Messaging Protocol (IEEE ICCCN 2022). *ResearchGate*, 04 2022. URL https://www.researchgate.net/publication/360235683_Exon_An_Oblivious_Exactly-Once_Messaging_Protocol_IEEE_ICCCN_2022.

Michael Kerrisk et al. poll(2) - linux manual page. https://man7.org/linux/man-pages/man2/poll.2.html, 2024a.

Michael Kerrisk et al. epoll(7) - linux manual page. https://man7.org/linux/man-pages/man7/epoll.7.html, 2024b.

Linus Torvalds and the Linux community. The linux kernel source code. https://github.com/torvalds/linux, 2025.

Google. Protocol buffers documentation, 2025. URL https://protobuf.dev.

Martin Sustrik. Optimising subscriptions in nanomsg, 2013b. URL https://250bpm.com/blog:19/index.html.