Universidade do Minho
Escola de Engenharia

Beyond Distributed Transactions through Exactly-once Exchanges

Ziad Kassam

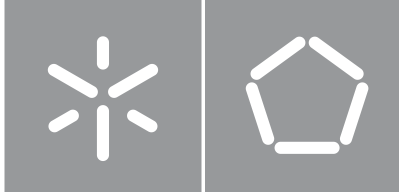
UMinho | 2024

Ziad Kassam

Beyond Distributed Transactions through Exactly-once Exchanges

Programa de Doutoramento em Informática (MAP-i)
das Universidades do Minho, de Aveiro e do Porto

April 2024



Universidade do Minho
Escola de Engenharia

Ziad Kassam

Beyond Distributed Transactions through Exactly-once Exchanges

Tese de Doutoramento
Programa de Doutoramento em Informática das Universidades do Minho, de Aveiro e do Porto

Trabalho Efetuado sob a orientação de
Dr. Ali Shoker
e de
Prof. Paulo Sérgio Almeida

April 2024

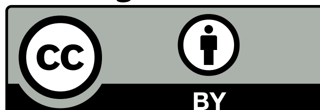
COPYRIGHT AND TERMS OF USE OF THIS WORK BY A THIRD PARTY

This is academic work that can be used by third parties as long as internationally accepted rules and good practices regarding copyright and related rights are respected.

Accordingly, this work may be used under the license provided below.

If the user needs permission to make use of the work under conditions not provided for in the indicated licensing, they should contact the author through the RepositoriUM of Universidade do Minho.

License granted to the users of this work



Creative Commons Attribution 4.0 International

CC BY 4.0

https://creativecommons.org/licenses/by/4.0/deed.en

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.9.0) [1].

Acknowledgements

I would like to begin by expressing my deepest appreciation to my two dedicated advisors, Ali Shoker and Paulo Sérgio Almeida, who have played a pivotal role in my academic journey. I am immensely grateful to them, who have consistently stood by my side, offering guidance, unwavering support, and remarkable patience throughout these past years. Without their guidance, this thesis would not exist.

I extend my sincere thanks to INESC TEC for generously covering my tuition fees over the last two years and for their support in handling papers submission payments.

I am grateful to my cousin Bassem Kheireddine, who was the initial inspiration for starting on this PhD journey. His unwavering assistance, along with his generous hospitality in providing me with accommodation, has been a tremendous blessing.

I also want to acknowledge my colleague and dear friend, Houssam Yactine. From the first day, we have started on this incredible journey together, experiencing both the highs and lows, offering each other unwavering encouragement, and sharing countless memorable moments during our travels, accommodations, meals, airport adventures, and more.

Last but certainly not least, I want to express my heartfelt gratitude to my family, father, mother, brothers and sisters, for their continous support.

Above all, my profound thanks go to my wife, Wafaa, whose support, encouragement, patience, understanding, assistance, and love have been the cornerstones of my success. To my children, Majd and Angela.

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 Universidade do Minho.

Braga, 19 April 2024

(Place) (Date)

(Ziad A. Kassam)

Resumo

Além das Transações Distribuídas por meio de trocas exatamente uma vez

As aplicações modernas estão cada vez mais interligadas e distribuídas, sendo a sua semântica muitas vezes garantida através de alguma forma de coordenação. Transações distribuídas e consenso podem levar à indisponibilidade perante partições de rede e sofrer elevada latência em redes de grande escala. Pat Helland, em "Life Beyond Distributed Transactions" propôs uma abordagem inspiradora, prometendo escalabilidade. Mas ao assumir *at-least-once*, dificulta a escrita de aplicações, forçando a lidar com questões de idempotência. A semântica de *exactly-once* é desejável em geral, mas difícil de obter de maneira correta e escalável. Abordagens correntes têm estado permanente relativo a cada *peer*, dificultando a escalabilidade, ou fazem assunções temporais para a correção.

Esta tese, inicialmente motivada pela melhoria da aplicabilidade da visão de Helland, introduz o protocolo Exon, que é simultaneamente 1) correto, garantindo *exactly-once* num modelo de sistema distribuído assíncrono, sem assunções relativas a tempo para correção; 2) oblívio, sem armazenar permanentemente estado relativo a um *peer* quando não existem mensagens pendentes para esse *peer*, tendo um inteiro como único estado permanente; 3) eficiente, conforme demonstrado por uma biblioteca implementada sobre UDP (Exon-lib) e respetiva avaliação, que mostrou desempenho quase igual a TCP em cenários normais sem falhas e melhor desempenho sob perda de pacotes.

A técnica base do Exon é um novo protocolo de quatro passos por mensagem, baseado na criação de *slots* únicos que dão a capacidade de consumir *tokens* únicos correspondentes, para garantir *exactly-once* de um modo oblívio. O núcleo é aumentado com "soft half-connections" a nível do protocolo, que são estabelecidas automaticamente quando necessário e descartadas com segurança, alcançando desempenho através da fusão e encadeamento das mensagens do protocolo de base.

A tese também estende o Exon para permitir a obtenção de *exactly-once* quando, perante uma partição, o destino não é alcançável e o nó remetente deseja suspender a execução. Isto é conseguido delegando a responsabilidade para outro nó intermediário, e funciona mesmo que o destino fique não alcançável com o envio já em curso. Finalmente, a tese apresenta um estudo de viabilidade de aplicações modernas relevantes em vários domínios, incluindo reservas online, internet de veículos e *middleware* para mensagens, para os quais Exon é um possível candidato, revisita a visão de Pat Helland assumindo o suporte de Exon, e conclui com um estudo de caso detalhado sobre agregação de dados.

Palavras-chave Exactly-once, protocolo oblívio, fiabilidade, escalabilidade

Abstract

Beyond Distributed Transactions through Exactly-once Exchanges

Modern applications are becoming increasingly interconnected and distributed. Their semantics are often guaranteed through some form of coordination. Distributed transactions and consensus may lead to unavailability under network partitions and suffer from high latency in wide area. Pat Helland, in "Life Beyond Distributed Transactions" proposed an inspiring approach, promising to be highly scalable. But by only counting on at-least-once messaging, it makes writing applications difficult, as the programmer has to deal with non-idempotency. Exactly-once messaging semantics is desirable in general, but difficult to achieve in a correct and scalable way. Current approaches need permanent peer-specific state, hindering scalability, or make timing assumptions for correctness.

This thesis, initially motivated by improving the applicability of Helland's vision, but going beyond it, introduces the Exon messaging protocol, which is simultaneously 1) correct, ensuring exactly-once under an asynchronous distributed system model, with no timing assumption for correctness; 2) oblivious, without permanently storing peer-related state when no pending messages to the peer remain, having a single integer as the only permanent state; 3) practically efficient, as demonstrated by an implemented lightweight library over UDP (Exon-lib) and its evaluation, showing it to have minimal overhead over TCP in normal fault-free scenarios and better performance under packet loss.

The core technique for Exon is a novel per-message four-way protocol, based on creating unique *slots* that give the ability to consume corresponding unique *tokens*, to ensure oblivious exactly-once messaging. The core is augmented with on-demand protocol-level "soft half-connections", that are established when needed and safely discarded, achieving performance through merging and pipelining the core protocol messages.

The thesis also extends Exon to allow exactly-once to be obtained when, under a partition, there is no path to the destination and the sending node wants to suspend for some time. This is achieved by delegating the responsibility to another node, working even if the final destination becomes suddenly unreachable while message sending was already in progress. Finally, the thesis presents feasibility studies of relevant modern applications in various domains, including online booking, automotive and messaging middleware, for which Exon is a possible suitable candidate, revisits Pat Helland's vision under Exon support, and concludes by a detailed case study of data aggregation.

Keywords Exactly-once messaging, oblivious protocol, reliability, scalability

Contents

List of Figures xi

List of Tables xiii

List of Algorithms xiv

1 Introduction 1

1.1 Motivation 1

1.2 Main Contributions 3

1.3 Bibliographic Note 4

2 Literature Review 6

2.1 Introduction 7

2.2 Life Beyond Distributed Transactions 8

2.2.1 Motivation 8

2.2.2 Key Concepts of Pat Helland's Model 10

2.3 Challenges with Message Delivery Semantics in Applications 14

2.3.1 Distributed Aggregation Applications 15

2.3.2 Edge Computing and Fog Networks 15

2.3.3 Online Booking Distributed Systems 16

2.3.4 Automotive Domain 17

2.3.5 Messaging Support for Distributed Middleware 19

2.3.6 Achieving Reliability with Exactly-Once Delivery Semantics 19

2.4 Distributed Transactions and Consensus 20

2.4.1 Distributed Transactions 21

2.4.2 Consensus Protocols 22

2.4.3 Challenges of Distributed Transactions and Consensus 23

2.5 Transport-level Protocols 23

2.5.1 TCP . . . 25
2.5.2 TCP Connection Recovery Solutions . . . 28
2.5.3 TCP Connection Fail-over Protocols . . . 30
2.5.4 UDP Based Protocols . . . 30
2.5.5 Logging in Message Delivery . . . 34
2.6 Message-oriented Middleware . . . 36
2.6.1 Apache Kafka . . . 36
2.6.2 ZeroMQ . . . 37
2.6.3 MQTT . . . 38
2.7 Oblivious Transport-level Messaging . . . 38
2.8 Discussion . . . 39

3 Exon Protocol **41**
3.1 Introduction . . . 41
3.2 System Model . . . 42
3.3 Overview . . . 43
3.4 The Algorithm . . . 44
3.4.1 Notations and Definitions . . . 44
3.4.2 Messaging Steps . . . 46
3.4.3 The "periodically" in the protocol . . . 54
3.5 Delving into the Intuition behind Key Aspects . . . 56
3.5.1 Requesting Slots, and Envelopes on Standby . . . 56
3.5.2 Incrementing Global Clock ck . . . 57
3.5.3 Closing Connection reqslots Message Loss . . . 59
3.6 Advanced Properties . . . 60
3.6.1 Soft Half-connections . . . 60
3.6.2 Obliviousness . . . 61
3.7 Correctness Proofs . . . 65
3.7.1 At-most-once . . . 65
3.7.2 At-least-once . . . 67

4 Exon-lib: an Exactly-Once oblivious library for lightweight messaging **68**
4.1 Introduction . . . 68
4.2 Overview . . . 68
4.3 Choice of Java for Implementation . . . 69
4.4 Understanding the API . . . 70
4.4.1 Why Use an API? . . . 70
4.4.2 API Overview . . . 70

4.4.3 API Example . . . 71
4.4.4 Simplicity from a Developer's Perspective . . . 72
4.5 Node State . . . 73
4.5.1 Sender and Receiver Connections Maps . . . 73
4.5.2 Half-connection Records . . . 73
4.6 Multi-Threaded Architecture . . . 74
4.6.1 Threads . . . 75
4.6.2 Message Types . . . 76
4.7 Flow Control . . . 77
4.8 Retransmission and Timeout . . . 79
4.8.1 Events . . . 80
4.9 Conclusion . . . 81
4.9.1 Architectural Complexity . . . 81
4.9.2 Learning and Growth . . . 82
4.9.3 Language Choice . . . 82
4.9.4 Remaining Challenges . . . 82

5 Exon Evaluation 83
5.1 Experimental Setup . . . 83
5.2 Evaluation Methodology . . . 83
5.3 Results . . . 85
5.3.1 Tolerance to Packet Loss . . . 85
5.3.2 Overhead under Normal Conditions . . . 87
5.4 Conclusion . . . 90

6 Partition Tolerance through Delegation 91
6.1 Motivation . . . 91
6.1.1 Messaging Protocols Limitations, Including Exon . . . 92
6.2 Alternative approaches for achieving delegation . . . 93
6.2.1 Extended API . . . 93
6.2.2 Modifying the Exon Distributed Protocol . . . 93
6.2.3 Basic Exon Algorithm with Structured Messages . . . 93
6.3 Extensible Exon Architecture . . . 94
6.4 Basic Delegation . . . 96
6.4.1 Possibility of dlv Oblivious Delegation . . . 96
6.4.2 Impossibility of token Oblivious Delegation . . . 97
6.4.3 Delegating Messages Scenarios - Messaging Steps . . . 97
6.5 Avoiding Nesting Along Delegation Chains . . . 101

6.6 Receiver-side Oblivious Delegation 101
6.7 Discarding State at the Sender 103
6.8 Summary 104

7 Applications **106**
7.1 Distributed Aggregation 106
7.2 Pat Helland's Vision using Exon 107
7.2.1 Remembering Messages as State 107
7.2.2 Activities: Managing State for Each Partner 108
7.2.3 Ensuring At-Most-Once Acceptance via Activities 109
7.2.4 Conclusion 109
7.3 Enhancing Online Booking Distributed Systems with Exon 110
7.3.1 Introduction 110
7.3.2 Challenges in Online Booking Distributed Systems 110
7.3.3 Partitioned State Systems 111
7.4 Automotive Domain 112
7.4.1 Challenges 113
7.4.2 Addressing Challenges using Exon 113
7.4.3 A Potential Problem with Exon 114
7.5 Messaging Support for Distributed Middleware 115
7.5.1 Distributed Actor Systems 115
7.5.2 Brokerless Messaging Middleware 115
7.5.3 Broker-based Messaging Middleware 116
7.6 Conclusion 116

8 Case study: Distributed Aggregation **118**
8.1 Introduction 119
8.2 Background and Related Works 120
8.3 The Technique Adopted 120
8.4 Evaluation 121
8.4.1 Experimental Setup 121
8.4.2 Convergence speed in a fault-free network 122
8.4.3 Convergence speed under loss and duplication 123
8.4.4 The impact of network partitions 124
8.4.5 The messaging overhead 125
8.5 Conclusion 126

9 Conclusions **127**

9.1 Limitations 127
9.2 Future Research Directions 128

Bibliography **130**

Appendices

A Full-Package Exon **138**

List of Figures

1 E-commerce platform scenario that uses two distributed databases 9
2 Two-phase commit protocol failure scenario 10
3 Two Layered Application 11
4 Data for an Application Comprises Many Entities 12
5 Entities interaction in an e-commerce model. 13
6 V2X Safety Message Dissemination System overview. [32] 18
7 The layers of the ISO/OSI model and their purposes in the ISO/IEC EN 14908 standard. . [39] 24
9 TCP packet received/lost uncertainty 28
10 FT-TCP architecture [40] 29
11 Zero RTT connection establishment with QUIC 32
12 Multiplexing in QUIC, avoiding Head-of-Line Blocking. [55] 33
13 Fault-free communication scenario. 43
14 Exon step-by-step example for node A communicating with node B. 47
15 Assigns values to the S-record elements, calculates “n”, and sends a reqslots message. 48
16 49
17 Assigns values to the R-record elements, modifies the clocks, creates slots, and sends a slots message. 50
18 Modifies the clocks, creates envelopes, creates a token from the envelopes, associates the message “m” to it, and sends a token message. 52
19 Removes the slot, delivers “m”, and sends an ack message. 53
20 Removes the token. 53
21 Node A sending messages to node B 60
22 Node A and node B exchanging messages 60
23 Node A requesting slots from node B, telling it to remove slots below l=51 62
24 Node A requesting slots from node B, telling it to remove slots below l=53 63

25 Exon-lib Architecture 76
26 Message Types 77
27 Flow-control, at the sender 78
28 Flow-control, at the receiver 79
29 Algorithm events 80

30 One-way throughput under packet loss (Setting: RTT=10ms, Bandwidth=100 Mbps) . . . 85
31 RPC latency under packet loss (Setting: RTT=10ms, Bandwidth=100 Mbps) 86
32 RPC throughput under packet loss (Setting: RTT=10ms, Bandwidth=100 Mbps) 87
33 One-way throughput as bandwidth and RTT varies 88
34 RPC throughput as bandwidth and RTT varies-log. scale 89
35 RPC Latency - same machine (Setting: RTT=0.04ms, Bandwidth=40 Gbps) 90

36 Delegating $payload$ 98
37 Delegating token - One Forwarding Node. 99
38 Delegating token - Multiple Forwarding Nodes. 100

39 If an entity works with many partners, it will have many activities. These are one per partner. 108
40 Entities using Exon 109
41 State-machine replication using consensus 111
42 Partitioned state system 112
43 V2X Domain. Source: [83] 112
44 Vehicles forming a platoon. Source: [87] 114

45 Convergence speed in fault-free scenario. 122
46 Estimated convergence speed under 20% message loss. 123
47 Root Mean Square Error under 10% network partition size and degree 10. 125

List of Tables

1 Delivery semantics summary 15

2 Estimated message's header size of FU, DRG, PS, and two fault tolerant piggy-backing patterns of PS. 125

List of Algorithms

1 Exon Algorithm 45

2 The Extensible Exon 95
3 The delegateConnection procedure added to the Exon algorithm 96
4 Avoiding Nesting Mechanism 102
5 Receiver-side Oblivious Delegation Mechanism 102
6 Discarding State Mechanism 104

7 Full-Package Exon 138

List of Listings

4.1 Server.java 71
4.2 Client.java 72

1 Introduction

As modern applications are becoming more networked and distributed, there is an increasing interest in reliable messaging protocols to ensure data integrity, fault tolerance, scalability, and real-time responsiveness. Distributed systems aim to provide seamless operation in the face of network partitions and varying conditions. However, achieving these goals is not easy, and traditional methods, such as distributed transactions, often struggle with the complexities of consistency and availability in the presence of network partitions. This has proven to be a daunting task if not powered by a reliable communication layer that provides the fundamental guarantees, as the exactly-once guarantee raised by Pat Helland in his famous visionary paper “Life beyond distributed transactions: an apostate’s opinion” [2].

This thesis delves into the realms of messaging protocols, obliviousness, and scalability to address the hurdles that have constrained the distributed systems. Through this exploration, we aim to shed light on the path toward more efficient and reliable distributed computing, not only in alignment with visionary ideals but also in practical applications that extend beyond their initial inspiration.

In this section, we lay the foundation for our motivation by examining the current landscape of distributed computing and the challenges it presents, setting the stage for our thesis’s core objectives and contributions.

1.1 Motivation

In the evolving landscape of distributed systems, the need for achieving fault tolerance, scalability, and reliability is very important. Attaining reliability and consistency between nodes is primarily achieved through the utilization of distributed transactions and consensus protocols. However, the use of distributed transactions may clash with the principles of the CAP theorem [3]. Emphasizing strong consistency through distributed transactions could compromise the system’s ability to handle network partitions and ensure high availability, which are essential aspects of many messaging systems designed to handle large-scale data streams and real-time events.

To address the inherent limitations of distributed systems, Pat Helland, in his paper "Life beyond distributed transactions: an apostate's opinion" [2], provides an approach, striving for "infinite scaling", aiming to tackle the traditional challenges in distributed systems. Pat Helland's visionary solution introduces the concept of entities, focusing on a single scope of serializability without transactions between different entities. This approach relies on utilizing at-least-once messaging guarantees and message idempotency at the application layer. While promising, this approach places a significant burden and complexities on programmers, hindering the seamless realization of this visionary approach.
Exactly-once messaging semantics is desirable in general, but difficult to achieve in a correct and scalable way. TCP often emerges as the preferred choice due to its perceived reliability, even in scenarios involving message-oriented middleware, such as ZeroMQ, and inter-actor communication, exemplified by distributed Erlang. However, TCP can fail with network issues, when the connection breaks, e.g. when switching between networks, necessitating the implementation of at-least-once and at-most-once delivery guarantees within the middleware layer above TCP. Moreover, the utilization of TCP in highly concurrent systems can lead to performance degradation. This degradation stems from the demand for TCP connection multiplexing, which can result in head-of-line blocking issues, hindering smooth data transmission.
Traditionally, achieving exactly-once messaging has involved the indefinite storage of messages sequence numbers for each source-destination pair. While this approach provides reliability, it falls short in terms of scalability, presenting a limitation in modern, large-scale distributed systems.

Therefore, "Exon" protocol was developed to enable oblivious exactly-once messaging, that achieves exactly-once messaging while maintaining an *oblivious* approach and preserving efficiency. Obliviousness, refers to the protocol's ability to function effectively without requiring continuous awareness (knowledge, information) of each source-destination combination, specially in the context of infinite scaling systems. The novelty of Exon is simultaneously achieving EO delivery, obliviousness, efficiency and no dependence on timing assumptions for correctness.
Moreover, we strive to alleviate the burden on programmers, freeing them from the complexities of ensuring reliable communication in their applications, and keep them concentrated on the core functionalities of their distributed applications.

Exon algorithm then entailed by the development of an open-source library named "Exon-lib", designed to implement the Exon protocol over UDP, offering a generic API that serves as a foundational building block for distributed applications. Subsequently, the empirical evaluation of Exon demonstrates significant improvements (40%) in throughput and latency under packet loss, while maintaining a negligible (8%) overhead over TCP in healthy networks.

Furthermore, the extended forwarding feature of Exon has emerged as a key addition to a broader applicability. Exon has been extended to enable exactly-once delivery even in scenarios where there is no direct

path to the destination due to network partitions or when the sending node needs to temporarily pause or disable its network connection. This achievement is realized by transferring the responsibility to another node, which continues to function, even if the ultimate destination unexpectedly becomes unreachable during an ongoing message transmission.

The thesis wraps up with feasibility study of relevant modern applications in various domains, including data aggregation, Pat Helland's vision under Exon support, online booking, automotive systems, and messaging middlewares, all of which offer interesting avenues for future exploration and validation. At the end, the thesis concludes with a detailed case study of data aggregation.

1.2 Main Contributions

In this section, I present the main contributions of this thesis:

The first major contribution, presented in Chapter 3, and published in [4], is a novel general-purpose protocol, referred to as Exon, was developed to enable oblivious exactly-once messaging. This protocol stands out due to its distinctive feature of remaining oblivious throughout its operation, ensuring that no connection information about the nodes communicated with is retained after closing connection. Moreover, this protocol has undergone rigorous analysis, demonstrating its robust safety and liveness properties, solidifying its capability to achieve exactly-once message delivery within distributed systems.

A second major contribution, presented in Chapters 4 and 5, and published in [4], encompasses the development and empirical evaluation of an open-source library named "Exon-lib", designed to implement the Exon protocol over UDP, offering a generic API that serves as a foundational building block for distributed applications. Its multithreaded design architecture ensures efficient thread management, facilitating concurrent and responsive message processing in distributed systems. Furthermore, the inclusion of sophisticated flow control mechanisms guarantee the orderly transmission of messages, preventing congestion and optimizing system performance. Additionally, Exon-lib incorporates robust retransmission and timeout handling strategies, enhancing message reliability even in the presence of network disruptions or delays. The abstract distributed algorithm of Exon just has "periodically" specified for these operations. However, during the implementation, careful consideration was given to the management of retransmissions and timeouts. This carefully designed library empowers developers with a potent toolset to create robust and high-performing distributed applications.
Subsequently, the experiments and empirical evaluations carried out in various network settings offer valuable insights into Exon's performance and effectiveness. These results assure its practical efficiency, where it can replace TCP as a transport layer in terms of efficiency, while retaining the good theoretical

properties. Notably, the empirical evaluation we conducted shows up to 40% throughput and latency improvements over TCP under packet loss, while posing minimal 8% overhead compared to TCP in stable networks.

The third major contribution is an extension of the Exon protocol, enabling nodes to delegate messages to intermediary nodes when direct communication with the final destination is temporarily interrupted due to network partitions. This capability allows for greater resilience in the face of network disruptions, enabling nodes and applications to suspend their operations gracefully during such partitions, confident that their messages will eventually reach their intended destinations once connectivity is restored. Detailed insights into this essential extension can be found in Chapter 6.

The fourth contribution, presented in Chapters 7 and 8, involves a feasibility study on relevant applications, and Data Aggregation case study. Chapter 7 revolves about the effective resolution of communication failures at the communication layer and the seamless integration of Exon into several applications. This approach significantly reduces the overhead associated with employing alternative fault tolerance techniques at higher system layers. Applications that benefits from this advancement include: Data Aggregation, that benefit from Exon to achieve reliability for protocols like PS and DRG. Pat Helland's Vision using Exon, offering "infinite scaling" and exactly-once messaging without application-level idempotency. In Online Booking Distributed Systems, Exon achieves message reliability between servers as data partitioned in order to enhance resilience and reduces errors. In the Automotive Domain, Exon's extended forwarding ensures information delivery in the presence of intermittent connectivity. Messaging Support for Distributed Middleware, including Distributed Actor Systems and Brokerless/Broker-based Messaging Middleware, Exon elevates reliability and efficiency.
Chapter 8 revolves about a detailed examination of Data Aggregation performance and the impact of Exon on protocol performance. This contribution is thoroughly discussed in Chapter 8 and has been published [5], providing valuable insights into the practical implications of Exon in the context of data aggregation.

1.3 Bibliographic Note

Parts of the work of this thesis have already been published in international conferences and workshops. A comprehensive list of these publications is provided below:

Kassam, Z., Almeida, P. S., & Shoker, A. (2022, July).
Exon: An Oblivious Exactly-Once Messaging Protocol.
In 2022 International Conference on Computer Communications and Networks (ICCCN) (pp. 1-10). IEEE. [4]

This paper presents Exon, an innovative oblivious exactly-once messaging protocol and a lightweight library implementation. It addresses the limitations of TCP in highly concurrent systems by ensuring both at-least-once and at-most-once delivery at the middleware layer. Exon achieves this through a unique per-message four-way protocol and on-demand protocol-level "soft half-connections" leading to improved performance, correctness, and obliviousness. Empirical evaluations show significant enhancements in throughput and latency under packet loss, with minimal overhead compared to TCP in stable network conditions.

Kassam, Z., Shoker, A., Almeida, P. S., & Baquero, C. (2017, October).
Aggregation protocols in light of reliable communication.
In 2017 IEEE 16th International Symposium on Network Computing and Applications (NCA) (pp. 1-4). IEEE. [5]
This paper examines aggregation protocols deployed in ad-hoc networks, where communication is often disrupted due to wireless technology. The study focuses on how these protocols behave when faced with communication failures like message loss, duplication, and network partitions. The research demonstrates that addressing communication failures at the communication layer through a reliable communication approach reduces the need for complex fault tolerance techniques at higher layers, while maintaining protocol accuracy and simplicity. The empirical study reveals that various aggregation protocols have trade-offs, and there is no universally suitable protocol for all scenarios.

Shoker, A., Kassam, Z., Almeida, P. S., & Baquero, C. (2016, December).
Life Beyond Distributed Transactions on the Edge.
In Proceedings of the 1st Workshop on Middleware for Edge Clouds & Cloudlets (pp. 1-3) [6].
The paper discusses Edge/Fog Computing, an extension of Cloud Computing designed to shift some of the workload from cloud data centers closer to the network's edge. While this model holds promise, there's a need for clearer foundations and ongoing exploration of new ideas. The paper references Pat Helland's vision presented in "Life beyond Distributed Transactions: an Apostate's Opinion", which proposes an approach to building highly scalable applications by treating data state as independent "Entities" with separate serialization scopes. This allows efficient local transactions within an entity but limits transactions involving different entities. The paper explores how Helland's vision aligns with the Edge Computing Model in terms of scalability, applications, and assumptions, while also discussing potential challenges.

2

Literature Review

In the realm of distributed computing, where interconnected networks power the backbone of modern applications and services, state-of-the-art technologies continue to shape the way data is exchanged, transactions are managed, and consensus is achieved. In the middle of these advancements, however, there exist many challenges related to message delivery semantics. This chapter not only delves into the cutting-edge advancements that have redefined the foundations of distributed systems, but also acknowledges the difficulties posed by ensuring reliable message delivery. Thus, it offers a comprehensive glimpse into the current state-of-the-art in this dynamic field.

We begin by Pat Helland's vision in his paper "Life beyond Distributed Transactions: an Apostate's Opinion" [2], discovering how it addresses the difficulties posed by traditional approaches and offers a new path towards enhanced distributed computing paradigms. This initiates an exploration into the complex realm of Message Delivery Semantics in Applications, shedding light on the challenges that underlie their implementation.
Next, we transition to exploring the solutions for ensuring reliable communication through the utilization of various tools and techniques. Our exploration starts by delving into the complications of Distributed Transactions. Subsequently, we delve into the foundational basics of communication within distributed environments – the transport-level protocols. These protocols lay the essential groundwork for data transmission among nodes.
Then, we delve into the realm of message-oriented middleware, a crucial component in facilitating asynchronous communication between distributed applications. By providing a decoupled and flexible approach to message exchange. Message-oriented middleware has revolutionized the scalability and resilience of distributed architectures.
Finally, we delve into the concept of oblivious transport-level messaging, a paradigm that focuses on nodes' ability to interact without maintaining connection-specific information about the nodes they communicate with.

As we dive deeper into these emerging techniques, we gain valuable insights into how the landscape of

distributed systems continues to evolve to meet the demands of an interconnected world.
With each section contributing a unique perspective on the advancements and challenges in distributed computing, this chapter offers a comprehensive exploration of the state-of-the-art technologies that drive the next generation of distributed systems. Through a deeper understanding of these innovations, we can forge a path toward building more robust, adaptable, and secure distributed solutions that stand at the forefront of modern computing.

2.1 Introduction

In the ongoing work of achieving almost-infinite scalable distributed systems, one brilliant idea has emerged in the form of a position paper by Pat Helland [2]. This paper not only encapsulates an idea for scalable systems, but also encourage us to rethink the foundations of exactly-once (EO) message delivery which serves as a cornerstone of reliable communication in scalable distributed systems.

The work on reliable communications, to provide both exactly-once (EO) delivery and good performance, date back to the early days of the ARPA network [7]. In particular, the research on the transport layer [8, 9] lead to the foundation of the Internet Protocol (IP) that results in two mainstream transport protocols: TCP and UDP. UDP [10] is a basic datagram protocol that provides no reliability guarantees, but stands as a communication primitive to support building other protocols as needed. The TCP protocol [11] is a stream- and connection-based protocol that provides reliability guarantees like exactly-once, FIFO ordering, and performance (e.g., congestion and flow control). This makes it the protocol of choice for most reliable IP applications today.

While these protocols stand as cornerstones of network communication today, it is worth noting that their development was significantly influenced by the OSI (Open Systems Interconnection) model [12], a conceptual framework formulated by the International Organization for Standardization (ISO). The OSI model, characterized by its division into seven distinct layers, each serving specific functions in the communication process, introduced a structured approach to understanding and standardizing network protocols.
The Transport layer, occupying a pivotal position in this model, resides above the Network layer and assumes responsibility for end-to-end communication and data flow management. As such, the evolution of the Transport layer, encompassing the research and design of protocols like TCP and UDP, played a vital role in shaping the dependable and high-performance networking applications we rely on today.

Nevertheless, the EO (Exactly-Once message delivery) guarantees in TCP only hold within a connection/session; when the connection fails (likely to occur in current WAN environments and long-lived communications), it either allows for message loss or duplication, as Belsnes [13] shows for any single-message communication. Attiya et al. [14] proved that when state information is not saved between

incarnations, the problem is solvable if and only if the network is FIFO—which is not the case for most networks. Therefore, to ensure EO, it is necessary to retain inter-connection information, e.g., at an upper layer. This requires using a "wrapper" fail-over protocol [15–18]. The alternative is to have the architects and developers develop some ad-hoc strategy to cope with the cumbersome delivery uncertainty of the last segment(s)— that may have been lost. (Note that modifying TCP is not desired due to standardization and compatibility reasons).

In this chapter, we delve through the solutions devised to achieve the goal of exactly-once message delivery. Each of these solutions is a response to the need of ensuring reliable communication while maintaining a lightweight nature, in line with the expansive distributed systems envisioned by Pat Helland. Despite this exploration, it becomes evident that no single solution offers a perfect resolution to the challenge. It was precisely this realization that gives me the motivation to delve into this realm, forming the foundation for my thesis — an exploration into ensuring reliable message delivery within the constraints of lightweight protocols that harmonize seamlessly with the visionary almost-infinite distributed systems envisioned by Pat Helland.

2.2 Life Beyond Distributed Transactions

2.2.1 Motivation

Numerous decades have been dedicated to the study of distributed transactions, encompassing protocols like 2PC [19], Paxos [20], and various quorum methodologies. These protocols create the perception among developers that they can attain global serializability in their applications. However, in practice, developers often refrain from implementing extensive and scalable applications that rely on distributed transactions. This is primarily due to the significant performance overhead [21] and inherent fragility associated with such transactions, leading many projects to flounder.

Cloud Computing [22] is an attractive computing model due to its pay-per-use business approach and seamless resource management. However, the surge in data from sources like the Internet of Things and social networks has challenged the model, leading to bottlenecks at cloud data centers [23]. To address this, Edge Computing [24] was introduced, aiming to alleviate cloud center load by performing data storage and computation closer to users. This offers benefits like improved user experience, load balancing, enhanced security, and optimized network usage [25]. Despite challenges in the distributed systems, will delve into them in the subsequent section, a model proposed by Pat Helland [2] suggests using separate data abstractions called "entities" to enable scalable and distributed applications. This approach aligns well with Edge Computing [6], emphasizing local data access and efficient communication channels.

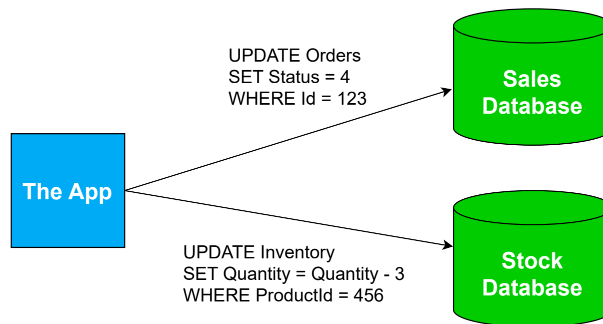


Figure 1: E-commerce platform scenario that uses two distributed databases

2.2.1.1 Distributed Systems Limitations

The increasing scale and complexity of modern distributed systems have exposed several significant limitations and challenges associated with traditional distributed transactions. One of the primary challenges lies in the geographic distribution of data and services, where nodes may span different regions, data centers, or even continents. As data and services are replicated and distributed to improve performance and fault tolerance, ensuring transactional consistency across these geographically dispersed nodes becomes a massive task [26][27].

Another critical limitation of distributed transactions is their impact on system performance and scalability [28]. As the number of participating nodes increases, the time required to coordinate and reach a consensus on a transaction's outcome also grows. The two-phase commit protocol, a widely used technique for coordinating distributed transactions, introduces latency and overhead that can significantly degrade system performance, leading to bottlenecks and reduced throughput [29].

Furthermore, distributed transactions are susceptible to various failure scenarios, such as network partitions, node failures, or software errors. In the face of such failures, ensuring the correctness and recoverability of distributed transactions becomes challenging. The need for error handling and recovery mechanisms further complicates the already complicated design of distributed systems.

2.2.1.2 Distributed Transactions Failures

We will explore the challenges of distributed transactions in the context of two databases: sales and stock. Imagine a scenario where an e-commerce platform uses distributed transactions to maintain consistency between these two databases during the process of fulfilling an order.

When a customer places an order, as shown in figure 1, the system needs to deduct the purchased items from the stock database and record the sale in the sales database. In a non-distributed system, this would be a straightforward process since both operations can be executed atomically within a single transaction. However, in a distributed environment where the sales and stock databases reside on different nodes or even in different data centers, ensuring the correctness and recoverability of this transaction becomes much more complicated.

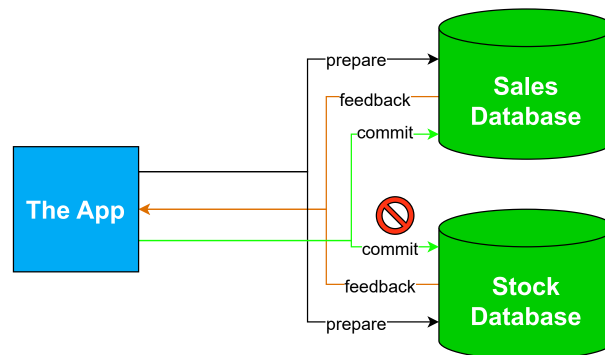


Figure 2: Two-phase commit protocol failure scenario

Several failure scenarios can occur during this process, such as network partition, node failure, and software errors. To address these challenges, developers need to implement extensive error handling and recovery mechanisms. They may need to design compensating transactions that can revert changes in case of failure. However, even with careful planning and recovery mechanisms, distributed transactions can introduce additional complexities, impacting system performance and potentially causing bottlenecks.

A notable issue with distributed transactions arises when utilizing the two-phase commit protocol. This protocol, while intended to ensure transactional consistency across distributed nodes, becomes a bottleneck in large-scale systems due to its reliance on synchronous coordination and blocking behavior. Figure 2 illustrates how the two-phase commit protocol can fail.

In light of these limitations and challenges, Pat Helland, in his paper "Life beyond distributed transactions: an apostate's opinion", proposes an innovative approach to tackle the complexities of distributed systems while maintaining data consistency and availability. The paper introduces novel concepts and techniques that empower developers to design and implement distributed systems that are more resilient, scalable, and efficient.

2.2.2 Key Concepts of Pat Helland's Model

Helland introduces a principle which is alternative to the traditional ACID (Atomicity, Consistency, Isolation, Durability) properties provided by distributed transactions. In distributed systems, maintaining strict consistency at all times may hinder scalability and availability. Instead, it advocates for a more relaxed consistency model where availability and partition tolerance are prioritized over immediate consistency. This is in line with the idea of the CAP theorem [3], which emphasizes that it is often impossible to simultaneously achieve strong consistency, high availability, and partition tolerance in a distributed system. According to the CAP theorem [3], when a distributed system experiences a network partition (P), you are faced with a trade-off between maintaining either consistency (C) or availability (A).

Delving into Pat Helland's concepts, we uncover fundamental assumptions and key opinions that underpin distributed computing. These principles serve as the foundation for his ideas.

Assumptions:

- Scalable Applications with Layered Architecture: Helland assumes that scalable applications are structured with at least two layers, as shown in figure 3. The lower layer manages the scaling process and provides a scale-agnostic programming abstraction to the upper layer. This abstraction allows the upper layer to write application code without concerning itself with scaling issues.

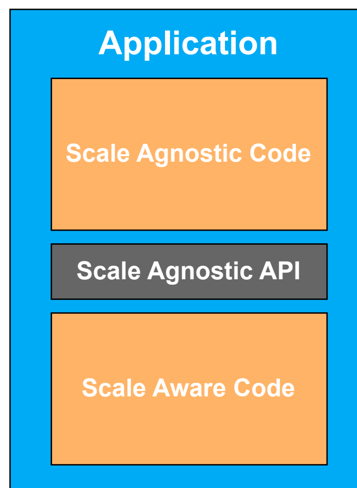


Figure 3: Two Layered Application

- Transaction Scopes: Helland addresses the challenges of achieving strongly consistent transactions over distributed systems and acknowledges that applications may often span multiple separate transaction scopes. This means that atomic transactions cannot cross entity boundaries, and developers must work within the scope of a single entity for each transaction.

- At-Least-Once Messaging: Helland highlights that most applications use at-least-once messaging, where messages can be delivered redundantly, leading to message retries and out-of-order delivery. As a consequence, applications must be designed to tolerate and handle such behaviors.

Opinions:

- Entities: The article advocates for scalable applications to utilize uniquely identified entities, each with a unique identifier or key, as shown in figure 4, representing disjoint sets of data. Atomic transactions should be limited to a single entity, and ensuring transactional consistency across multiple entities is not a viable approach in scalable designs.

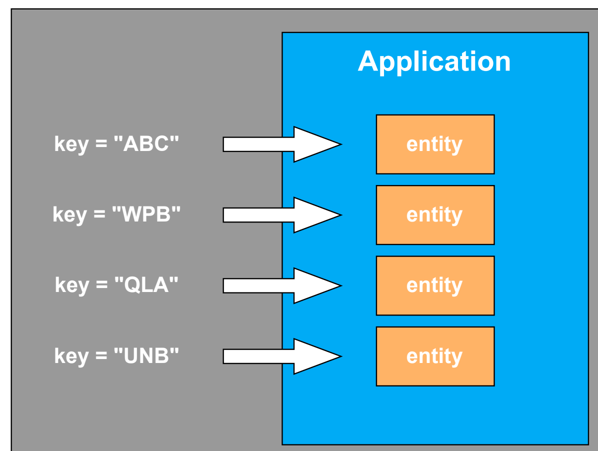


Figure 4: Data for an Application Comprises Many Entities

- Activities: Messages in messaging systems should be addressed to entities using their entity keys. Entities should manage per-partner state using activities, enabling fine-grained workflow-style solutions in a scalable environment. Activities store the state needed for handling messaging and loosely coupled agreements between entities.

2.2.2.1 Entities and Activities in Action: Sales and Stock Management Example

The process of handling distributed transactions is different from the traditional approaches like the two-phase commit protocol. Let us explore how this works in the context of the sales and stock example.

- Entities represent collections of named (keyed) data that can be atomically updated within the entity itself but not across multiple entities simultaneously. In our example, the SalesDB and StockDB databases would each be an entity, containing sales transactions and products information, respectively.

- Activities keep track of messages between entities. In our example, the activity acts as intermediary that manage the exchange of messages between the application and the SalesDB and StockDB entities.

As shown in figure 5, the process begins when a customer places an order through the application. The application initiates the sale process and the stock update activities. The stock update activity passes the ProductId and the Quantity to the StockDB, and the sale process activity passes the order Id to the SalesDB.

These passings are done asynchronously, without waiting for one another, which stands in contrast to the traditional Two-Phase Commit (2PC) protocol. In the entities and activities model, the application initiates both the stock update activity and the sale process activity independently. This decoupled approach enables each activity to handle its responsibilities autonomously and asynchronously, without the need for synchronous coordination or waiting for the other activity's completion. This eliminates the blocking

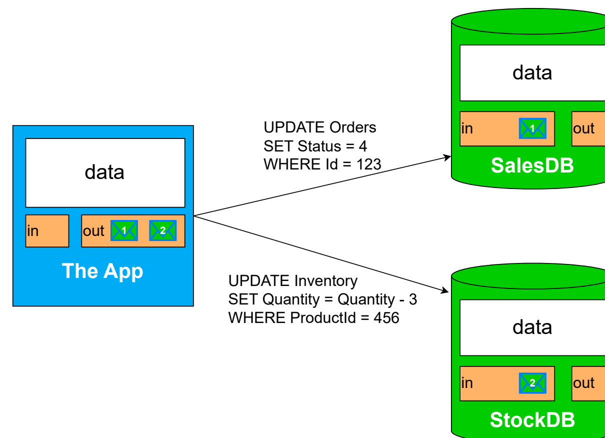


Figure 5: Entities interaction in an e-commerce model.

behavior and communication overhead often associated with traditional distributed transactions, thus enabling a more adaptable and resilient architecture for real-time data processing and messaging systems. However, a notable challenge in this decoupled approach is ensuring the reliable delivery of messages between these activities, as the absence of synchronous communication introduces the potential for message loss or duplication.

2.2.2.2 Idempotent Messaging

Messages exchanged between the application, the SalesDB and the StockDB, are designed to be idempotent. If message delivery fails, the activities can safely retry the messages without causing inconsistencies. The critical point to emphasize is that the idempotence mechanism for messages is currently not available.

> «The scale-agnostic (higher-level) portion of the application must implement mechanisms to ensure that the incoming message is idempotent. This is not essential to the nature of the problem. Duplicate elimination could certainly be built into the scale-aware parts of the application. *So far, this is not yet available.*» Pat Helland [2].

Consequently, developers of scale-agnostic applications are left with the task of implementing mechanisms to ensure the idempotency of incoming messages, which adds complexity and demands careful consideration to maintain data integrity and correctness.
In the following section, I will explore the challenges associated with message delivery semantics in applications and the pivotal role they play in shaping the reliability and consistency of distributed systems.

2.3 Challenges with Message Delivery Semantics in Applications

Message delivery semantics are important concepts in distributed systems and messaging protocols. They define how messages are delivered between components or nodes in a system. There are three common types of message delivery semantics: at-least-once, at-most-once, and exactly-once.

At-Least-Once delivery semantic ensures that a message will be delivered to the recipient at least once. This means that the sender will keep retrying to send the message until it receives a confirmation from the receiver that the message has been successfully received. If no acknowledgment is received, the sender will continue retransmitting the message until it receives confirmation. However, there is a possibility that a message might be delivered multiple times.

At-Most-Once delivery semantic, the message is delivered either once or not at all. If there is a communication error or disruption, consumers might not receive the message, resulting in non-delivery. This approach is well-suited for applications that prioritize high throughput and low latency, given its fire-and-forget nature.
The relatively low cost and implementation overhead make at-most-once delivery appealing. However, its drawback lies in potential data loss, which can be a concern for data-sensitive applications.

Exactly-Once delivery semantic ensures that a message is delivered exactly once to the recipient. It means that the message will not be lost, and it will not be duplicated during the delivery process. Achieving exactly-once delivery is more complex compared to the other two semantics and often requires additional mechanisms to guarantee it.
One of its key strengths lies in its ability to guarantee both the successful delivery of messages and the prevention of duplicates, ensuring data integrity and accuracy. By eliminating the possibility of message duplication, it ensures correctness and consistency in message processing, making it highly suitable for critical applications.
However, the implementation of exactly-once delivery can be more challenging compared to the other delivery semantics, as it demands additional mechanisms to track and manage message state. These complexities can make it harder to set up and maintain. Moreover, the pursuit of guaranteeing exactly-once delivery might come at the cost of higher overhead, potentially impacting system performance compared to simpler delivery models. Table 1 summarizes the behavior of all delivery semantics.

In the following, I will discuss the implications of the need for reliable message delivery semantics on certain applications.

	at-most-once	at-least-once	exactly-once
duplicates	No	Yes	No
data loss	Yes	No	No

Table 1: Delivery semantics summary

2.3.1 Distributed Aggregation Applications

Distributed data aggregation stands as a crucial task, enabling the decentralized determination of meaningful global properties, subsequently utilized to guide the operation of diverse applications. The outcomes are achieved through distributed computation of functions such as COUNT, SUM, and AVERAGE. Some application examples deal with the determination of the network size, total storage capacity, average load, majorities and many others [30].
Aggregation protocols allow for distributed lightweight computations deployed on ad-hoc networks in a peer-to-peer fashion. The utilization of wireless technology in such networks, however, introduces challenges, as the communication medium is frequently unpredictable. As a result, these protocols become vulnerable to issues concerning accuracy, consistency, and performance.

Message duplication or loss disrupts information flow between nodes, leading to inconsistencies: duplication inflates data importance, while loss causes inaccurate results due to missing input, blasting the protocol's reliability [5].
In chapter 8 , I will delve into this topic in details. Through experiments, I will demonstrate how message loss and duplication can significantly impact the overall correctness of aggregation protocols. By examining real-world scenarios, we will uncover the implications of these challenges on the practical deployment of aggregation protocols in various contexts.

2.3.2 Edge Computing and Fog Networks

Edge/Fog Computing [24] emerges as an expansion of the Cloud Computing [22] paradigm, primarily designed to carry a portion of the workload from centralized cloud data centers and distribute it towards the network's edge, closer to the clients.
However, message delivery semantics, specifically message loss or duplication, can impact the performance of edge computing applications in the contexts of IoT, Smart Cities, Industrial Automation, Healthcare, Military Operations, etc., potentially leading to critical scenarios:

- Internet of Things (IoT): In IoT applications, devices often generate streams of data that require timely processing. If message loss occurs due to unreliable communication links at the edge, vital sensor data might not reach the central system. This could lead to incomplete situational awareness, that may hinder the decision-making processes. For instance, in a smart home security

system, lost messages from motion sensors could result in delayed alerts about potential intrusions, compromising the safety of residents.

- Smart Cities: Smart city systems rely on real-time data from various sources to manage traffic, waste, and energy. In scenarios where message duplication happens, redundant data could overwhelm processing systems, leading to inaccurate insights and resource allocation. For example, if traffic sensor data is duplicated, traffic management algorithms might misinterpret congestion levels, causing incorrect adjustments to traffic signal timings and increase congestion issues.
- Industrial Automation: In industrial automation, edge computing is pivotal for maintaining efficient operations. If messages are duplicated, it could trigger redundant actions, causing equipment to operate incorrectly or out of sync. Conversely, if messages are lost, critical information for real-time control might not reach the edge controllers, leading to production delays or even equipment damage. For instance, in a manufacturing plant, duplicated messages to robotic arms could cause them to perform unnecessary actions, potentially damaging products or machinery.
- Healthcare: In healthcare applications, edge computing aids in monitoring patients' vital signs and providing timely interventions. Message loss could lead to delayed alerts about critical conditions, threatening patients' health. For instance, if messages from wearable heart rate monitors are lost, medical staff might not be alerted promptly to dangerous spikes or drops in heart rate, potentially resulting in severe consequences such as jeopardizing patient lives.
- Military Operations: In the context of military operations, the consequences of unreliable message delivery can be particularly terrible. When orchestrating crucial tasks like missile launches, any message loss or duplication could lead to catastrophic outcomes. If launch command messages are lost, delayed, or duplicated, it could result in the incorrect launch of missiles, compromising strategic objectives and causing unintended collateral damage.

In all these scenarios and not limited to, unreliable message delivery can result in catastrophic outcomes. Loss of critical data due to message loss could lead to delayed responses or incorrect decisions. On the other hand, message duplication can create confusion and waste computational resources, potentially causing unintended actions. Ensuring robust and reliable communication protocols at the edge becomes essential to prevent these scenarios and maintain the integrity and efficiency of these applications.

2.3.3 Online Booking Distributed Systems

2.3.3.1 Introduction

Online booking distributed systems, such as those for flights or hotel rooms, play a pivotal role within the travel and hospitality industry. These systems facilitate real-time booking, availability checks, and reservations through a network of brokers, agents, and service providers. However, coordinating various aspects

of these systems can be complex and challenging, often resulting in issues like duplicate bookings, data inconsistency, and inefficient communication. These challenges not only disrupt the operational efficiency of the system but can also have significant implications for the business model, potentially leading to financial losses and liabilities.

Online booking distributed systems, spanning reservations for flights, hotel rooms, and more, serve as the lifeblood of the travel and hospitality industry. These systems offer customers the convenience of real-time bookings, but behind the scenes, they grapple with a multitude of complex challenges.

The following are the key challenges faced by these systems:

- Distributed Architecture: Online booking systems inherently possess a distributed architecture involving multiple brokers, agents, and data sources. Coordinating communication and ensuring data consistency across these distributed components can present significant challenges.
- Message Reliability: Booking systems necessitate reliable message delivery to prevent issues like double bookings or reservation conflicts. Ensuring that messages are processed exactly once is crucial for maintaining system integrity.
- Scalability: With an increasing number of bookings and users, scalability becomes a critical concern. Systems must scale to handle a high volume of requests without compromising performance.
- Fault Tolerance: To maintain uninterrupted service, online booking systems must exhibit resilience to network failures, encompassing challenges such as message loss, duplication, and network partitions.

In the realm of booking systems for flights or hotels, the consequences of message unreliability can significantly affect businesses. Such unreliability can result in customer dissatisfaction, leading to inconvenience and potential revenue loss as customers seek alternatives. The operational impact includes resource wastage and added complexities in handling duplicate messages. Moreover, the legal and liability risks arising from incorrect bookings can result in financial losses and damage to the business's reputation. In a competitive industry, unreliable messaging can shrink a business's competitive advantage. To mitigate these consequences and ensure the integrity of operations, robust and dependable messaging protocols are of high importance in this domain.

2.3.4 Automotive Domain

The V2X (Vehicle-to-Everything) [31] domain in the automotive industry stands at the forefront of innovation, bringing about significant transformation. With the rising integration of connectivity and autonomous capabilities in vehicles, the V2X landscape encompasses a sophisticated system where vehicles engage in interactions that extend beyond their own kind.

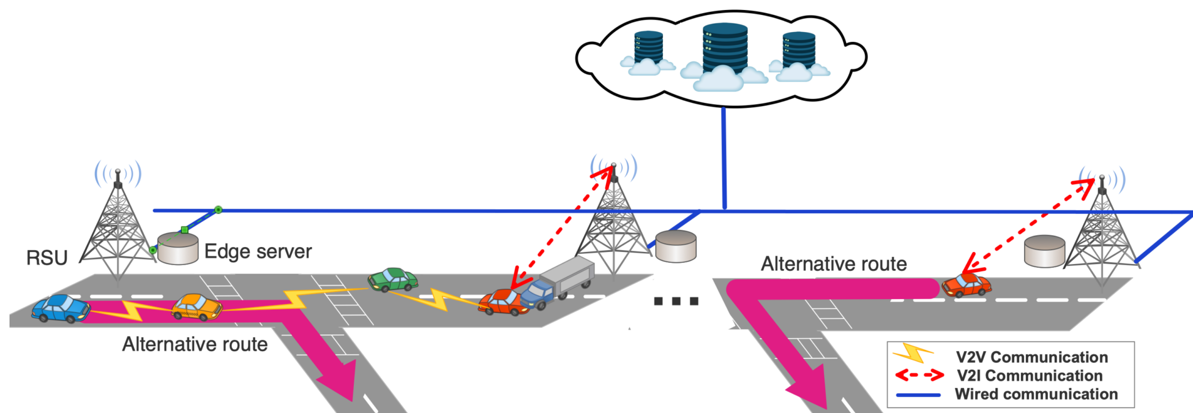


Figure 6: V2X Safety Message Dissemination System overview. [32]

In this domain, vehicles not only communicate with each other (V2V) but also establish connections with various entities. These encompass interactions with infrastructure (V2I), facilitating seamless communication between vehicles and the surrounding road systems (Road Side Units). Furthermore, V2X involves engagement with pedestrians (V2P), ensuring safety and efficient movement in shared spaces. Even interactions with cyclists and the broader transportation network (V2N) are encompassed within this complex web of connectivity and communication.

Message delivery semantics, specifically message loss or duplication, can impact the performance of V2X (Vehicle-to-Everything) applications, potentially leading to critical scenarios:

- V2V (Vehicle-to-Vehicle) Communication: Message loss or duplication in V2V communication can have dangerous consequences. For instance, in a scenario where a message about sudden braking or an obstacle ahead is lost, neighboring vehicles might not receive timely warnings, increasing the risk of collisions. On the other hand, duplicated messages could lead to unnecessary actions, causing confusion among drivers and potentially triggering accidents.

- V2I (Vehicle-to-Infrastructure) Communication: In V2I communication, where vehicles interact with infrastructure like traffic lights and road side units, as we can see in figure 6, message loss could result in vehicles being unaware of changing traffic conditions. This could lead to instances where vehicles fail to receive information about traffic signal changes, potentially causing congestion or even accidents. Message duplication could lead to redundant actions, such as unnecessary stops at intersections.

- V2P (Vehicle-to-Pedestrian) Communication: In V2P interactions, lost messages could mean that pedestrians do not receive warnings from nearby vehicles about their proximity. This could lead to accidents, especially in situations where pedestrians are crossing the road. Conversely, message duplication might cause pedestrians to misjudge the intentions of vehicles, leading to hesitation or risky actions.

- V2N (Vehicle-to-Network) Communication: In V2N scenarios, where vehicles communicate with the broader transportation network, message loss could affect the real-time traffic flow data collected by vehicles. This could lead to inaccurate traffic predictions and inefficient route recommendations, affecting overall traffic management. Duplicated messages might lead to misleading traffic data, causing unnecessary changing paths and increased congestion.

Given the safety-critical nature of V2X applications, ensuring reliable communication protocol is crucial to prevent these catastrophic scenarios and to maintain the integrity of the transportation ecosystem.

2.3.5 Messaging Support for Distributed Middleware

In the domain of distributed middleware, messaging support is of high importance for enabling efficient and dependable communication.
Distributed actor systems, exemplified by technologies like Erlang and Akka, are well-suited for constructing online applications with dynamically interacting entities, such as social networks, online games, and IoT applications. These systems simplify design by mapping application objects onto lightweight actors that encapsulate object state and logic, communicating asynchronously via messages. However, challenges arise, including TCP stream multiplexing and Head-of-Line (HoL) blocking.
Brokerless messaging systems, exemplified by technologies like ZeroMQ and Nanomsg/NNG, employ TCP as the transport layer protocol and aim to recover from temporary communication disruptions. However, a challenge arises concerning the potential loss of queued messages upon disconnection.
Broker-based messaging middleware employs central intermediaries (brokers) to manage message routing, distribution, and coordination between producers and consumers. In contexts like IoT, systems like MQTT have gained popularity for their efficiency and scalability. Recent developments seek to replace traditional transport mechanisms like TLS+TCP with advanced alternatives like QUIC, as seen in MQTT over QUIC [33].

Ensuring message reliability is paramount to prevent data loss, maintain system integrity, and deliver consistent performance. In this evolving landscape, a robust and dependable messaging protocol is crucial to ensure the seamless functioning of these diverse middleware approaches.

2.3.6 Achieving Reliability with Exactly-Once Delivery Semantics

Across an array of domains such as distributed aggregation, edge/fog computing, and the automotive sector, the assurance of reliability and accuracy in aggregated data and interactions is of high significance. While messaging delivery semantics like at-most-once and at-least-once provide certain guarantees, they also introduce vulnerabilities that can lead to inconsistencies and catastrophic scenarios. The potential for message loss due to at-most-once semantics, as well as message duplication resulted from at-least-once

semantics, holds the potential to disrupt critical operations and compromise safety.

To address these challenges, exactly-once message delivery semantics is the solution. By ensuring that messages are neither lost nor duplicated, exactly-once semantics provide a crucial foundation for maintaining data integrity and guaranteeing accurate aggregation results. In applications where precision, reliability, and safety are crucial, transitioning to exactly-once message delivery becomes essential.

However, it is important to acknowledge that achieving a universally accessible exactly-once message delivery remains a complex challenge. The complications of distributed systems, varying network conditions, and the need to handle failures can introduce complexities that hinder the implementation of exactly-once semantics. Additionally, even when exactly-once delivery mechanisms are devised, they may come with certain drawbacks. These include increased processing overhead, which can impact system performance, as well as the challenges posed by the memory-limited or constrained devices, where the management of state for tracking messages could strain available resources.
It is worth noting that even with non-constrained resource devices like vehicles, a unique challenge arises. For instance, vehicles might intermittently disconnect and reconnect, even over longer intervals like weeks or months. As a result, preserving complete communication histories with all previously connected vehicles, which is needed to ensure non duplicating messages, becomes a notably complex endeavor.
Thus, while exactly-once message delivery offers a promising solution, its practical implementation demands careful consideration of these challenges and trade-offs to strike the right balance between reliability and performance.

In the subsequent sections, I will delve into the strategies and techniques aimed at realizing exactly-once message delivery, however accompanied by trade-offs and additional challenges. Additionally, I will explore situations in which ensuring exactly-once semantics could be challenging due to the limitations and complexities.

2.4 Distributed Transactions and Consensus

A transaction is a logical unit of work that represents a sequence of operations performed on a database or a system. It is an essential concept in database management systems and plays a crucial role in maintaining data integrity and consistency [34].
Key Characteristics of a Transaction:

- Atomicity: Atomicity ensures that a transaction is treated as a single, indivisible unit of work. Either all the operations within the transaction are successfully completed, or none of them take effect. If any operation fails or encounters an error, the entire transaction is rolled back, and the database returns to its original state before the transaction started.

- Consistency: Consistency guarantees that a transaction brings the database from one consistent state to another consistent state. It ensures that all database constraints, such as integrity rules and data validations, are maintained throughout the transaction's execution. In other words, a transaction should preserve the overall correctness and validity of the data.

- Isolation: Isolation ensures that each transaction is isolated from the effects of other concurrent transactions. Concurrent execution of multiple transactions can lead to various concurrency-related issues, such as data inconsistency, lost updates, and dirty reads. Isolation mechanisms, like locking or concurrency control algorithms, prevent interference between concurrent transactions, maintaining data integrity.

- Durability: Durability guarantees that once a transaction commits and its changes are written to the database, they will persist even in the event of subsequent failures, such as system crashes or power outages. The changes made by committed transactions are considered permanent and should survive any system or hardware failures.

Transactions are widely used in various applications, especially in database management systems, where data integrity and consistency are critical. Transactions allow multiple operations to be grouped together, ensuring data reliability and recoverability.
In addition to databases, transactions are also employed in other systems and domains, such as distributed systems, financial systems, e-commerce platforms, and more. They provide a reliable mechanism for managing complex operations and ensuring the integrity of data and system state.

Distributed systems often involve multiple nodes or processes that need to coordinate and work together to achieve a common goal. However, ensuring consistency and reliability in such systems can be challenging due to factors like network failures, node failures, and concurrent updates. Distributed transactions and consensus protocols are two fundamental concepts used to address these challenges and provide reliability and correctness guarantees in distributed systems.

2.4.1 Distributed Transactions

A distributed transaction is a transaction that spans multiple nodes or processes in a distributed system. It involves a set of operations that need to be executed atomically, ensuring either the success of all operations or their complete rollback.

Key Concepts:

- ACID Properties: Distributed transactions aim to maintain ACID (Atomicity, Consistency, Isolation, Durability) properties, similar to transactions in a single-node database. Atomicity guarantees that either all operations within a transaction are successfully completed, or none of them take effect.

Consistency ensures that the system remains in a valid state before and after the transaction. Isolation provides concurrency control, ensuring that transactions do not interfere with each other. Durability guarantees that the effects of a committed transaction persist even in the event of failures.

- Two-Phase Commit (2PC) [35]: Two-Phase Commit is a widely used protocol for coordinating distributed transactions. It involves a coordinator node that interacts with participant nodes to ensure that all participants agree on committing or aborting the transaction. The protocol proceeds in two phases: prepare phase and commit/abort phase. The coordinator communicates with participants to determine if they can commit or need to abort the transaction. Once all participants agree, the coordinator issues a commit command, and participants apply the transaction changes. If any participant fails or disagrees during the process, the coordinator triggers an abort, ensuring consistency.

2.4.2 Consensus Protocols

Consensus protocols enable a set of nodes in a distributed system to agree on a single value or decision, even in the presence of failures or malicious behavior [36]. These protocols play a vital role in achieving consistency and coordination among distributed nodes.

Main Consensus Protocols:

- Paxos [20]: Paxos is a widely known consensus protocol that provides fault-tolerant consensus in a distributed system. It uses a leader-based approach, where a single node acts as a leader responsible for proposing values. The protocol proceeds through a series of phases, including proposal, acceptance, and learning, to reach a consensus on a proposed value. Paxos can handle failures and network partitions, ensuring safety and liveness properties.

- Raft [37]: Raft is another consensus protocol that simplifies the understanding and implementation of consensus compared to Paxos. It follows a leader-based approach similar to Paxos but introduces additional mechanisms like leader election, log replication, and safety properties to ensure fault-tolerant consensus. Raft divides time into terms, and a leader is elected for each term to coordinate the replication of log entries across the nodes.

- Practical Byzantine Fault Tolerance (PBFT) [38]: PBFT is a consensus protocol designed to tolerate arbitrary faults, including Byzantine faults, where nodes can behave maliciously. It achieves consensus through a series of message exchanges and relies on a threshold of correct nodes to make progress. PBFT provides a high level of fault tolerance at the cost of increased message complexity and communication overhead.

Consensus protocols are crucial in distributed systems that require strong consistency guarantees and fault tolerance. They are used in various applications, such as distributed databases, blockchain networks, and distributed file systems, to ensure agreement and consistency among multiple nodes.

2.4.3 Challenges of Distributed Transactions and Consensus

However, while distributed transactions offer strong consistency guarantees, they come with certain disadvantages, particularly in the context of messaging systems. One major drawback is their heavyweight nature, which can introduce performance overhead and limit scalability. Distributed transactions often require coordination and consensus among multiple nodes, involving communication and synchronization overhead. This can impact the throughput and latency of messaging systems, hindering their ability to handle high volumes of messages and real-time data feeds. Furthermore, the need for distributed transaction managers or protocols adds complexity to the system architecture and increases the likelihood of failures or bottlenecks.
Moreover, the use of distributed transactions may also clash with the principles of the CAP theorem [3], where the trade-offs between consistency, availability, and partition tolerance become more pronounced in distributed messaging systems. Emphasizing strong consistency through distributed transactions could compromise the system's ability to handle network partitions and ensure high availability, which are essential aspects of many messaging systems designed to handle large-scale data streams and real-time events.

As a result, in scenarios where high throughput and low latency are crucial, such as real-time data processing or event-driven systems, the use of distributed transactions may not be ideal. Alternative approaches, such as event sourcing, idempotent processing, and eventual consistency models, are often favored to achieve scalability, fault tolerance, and performance in messaging systems.

2.5 Transport-level Protocols

Networks operate based on a structured framework called the OSI (Open Systems Interconnection) model [12], which outlines a hierarchical arrangement of seven layers, as depicted in figure 7. As we ascend through the layers from the foundational Physical layer to the Transport layer, the focus shifts from basic signal transmission to higher-level aspects like message reliability and error correction.

- Physical Layer: At the bottom of the OSI model, the Physical layer deals with the actual transmission and reception of raw data bits over a physical medium, such as cables or wireless signals. It ensures that electrical, optical, or radio signals representing binary data are successfully transmitted between devices. While the Physical layer is responsible for the basic connectivity, it does not inherently guarantee message reliability or error correction.

OSI Layer	Purpose
Application	Application Program
Presentation	Data Interpretation
Session	Remote Actions
Transport	End-to-End Reliability
Network	Destination Addressing
Data Link	Media Access & Framing
Physical	Electrical Interconnect

Figure 7: The layers of the ISO/OSI model and their purposes in the ISO/IEC EN 14908 standard. . [39]

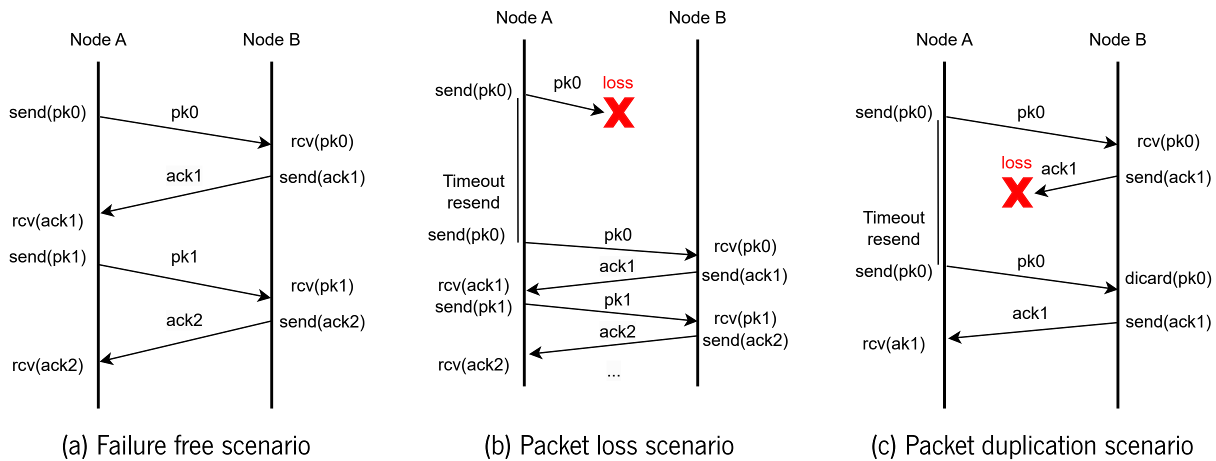
- Data Link Layer: Moving up to the Data Link layer, protocols here are concerned with framing data into frames, addressing, and error detection. Error detection mechanisms, like cyclic redundancy check (CRC), are employed to identify errors during transmission. While these mechanisms can detect errors, they do not necessarily ensure full message reliability or correction.

- Network Layer: The Network layer is responsible for routing data packets from source to destination across a network. Routers use logical addressing (IP addresses) to determine the best path for data delivery. However, while routing enhances efficiency, this layer does not inherently address the issue of message reliability.

- Transport Layer: As we ascend to the Transport layer, the focus shifts towards achieving message reliability and integrity. This layer ensures end-to-end communication between applications running on different devices.

  Two primary transport protocols, TCP and UDP, offer different trade-offs in this context:

  - TCP (Transmission Control Protocol): TCP provides reliable, connection-oriented communication.
  - UDP (User Datagram Protocol): UDP is a connectionless protocol that does not provide the same level of reliability as TCP. It is suitable for scenarios where low overhead and reduced latency are priorities, but without the guarantee of delivery or order.



(a) Failure free scenario (b) Packet loss scenario (c) Packet duplication scenario

Achieving message reliability evolves as we progress from the Physical layer to the Transport layer in the OSI model. While the lower layers ensure data transmission and error detection, it is the Transport layer's TCP protocol that introduces features like acknowledgment, retransmission, flow control, and ordered delivery, all working together to enhance the reliability and integrity of message communication across distributed systems.
By exploring protocols such as TCP and UDP, as well as addressing topics like connection recovery, failover strategies, and the role of logging in message delivery, this section sheds light on the communication protocols that play a pivotal role in maintaining the stability and resilience of distributed systems. Through an exploration of these transport-level protocols, this section delves into the heart of network communication, offering insights into both the challenges and solutions inherent in achieving reliable data exchange across distributed nodes.

2.5.1 TCP

TCP (Transmission Control Protocol) [11] is a widely adopted transport-level protocol that provides reliable, connection-oriented communication between devices in computer networks. It offers several key features and mechanisms to ensure accurate and ordered message delivery.

2.5.1.1 Features of TCP

Reliable Data Transfer TCP guarantees reliable data delivery by implementing mechanisms such as sequence numbers, acknowledgments, and retransmissions. Sequence numbers allow the receiver to reorder out-of-sequence segments, and acknowledgments ensure the sender is aware of successfully received data. In the event of packet loss, TCP retransmits the missing segments, ensuring reliable data transfer.

Figure 8a depicts a failure free scenario. In this scenario, Node A sends a packet pk0, which is assigned a sequence number of “0”. Upon reception, Node B sends an acknowledgment (ack1) for node A, by this,

node B informs Node A that the reception of pk0 was successful and that it is clear to transmit pk1.
Figure 8b shows a scenario where pk0 encounters a loss during its journey. As a result of not receiving an acknowledgment, Node A waits for a predetermined timeout period before deciding to retransmit pk0. Upon reaching the timeout, Node A retransmits packet pk0, ensuring its delivery to Node B. This retransmission is accompanied by the original sequence number "0". Upon receiving this retransmitted packet, Node B responds with an acknowledgment (ack1), signifying the reception of pk0 and the readiness to proceed.
Figure 8c depicts the duplication of packet pk0. In this situation, Node A sends packet pk0, which has the sequence number "0". Upon its reception, Node B acknowledges the successful receipt of pk0 by sending an acknowledgment (ack1) to Node A. However, due to unstable network or other factors, the acknowledgment (ack1) sent by Node B gets lost. Upon reaching the timeout, Node A retransmits packet pk0, ensuring its delivery to Node B. This retransmission is accompanied by the original sequence number "0". Upon receiving this retransmitted packet, Node B recognizes the packet as a duplication, thus, discards it. Node B proceeds to reissue acknowledgment (ack1) to acknowledge the correct reception of packet pk0.

Flow Control TCP incorporates flow control mechanisms to manage the rate of data transmission. It utilizes a sliding window technique, where the sender and receiver agree upon a window size that indicates the number of unacknowledged segments allowed at any given time. This ensures that the receiver can handle incoming data without being overwhelmed, thereby optimizing network performance.

Connection-Oriented Communication TCP establishes a connection between the sender and receiver before data transmission. This connection-oriented approach provides several benefits, including error detection and recovery, orderly data transfer, and congestion control. It also allows for the establishment of virtual circuits and enables bidirectional communication.

2.5.1.2 Message Delivery in TCP

TCP employs various mechanisms to ensure reliable message delivery, addressing challenges such as message loss and duplication. By utilizing sequence numbers, acknowledgments, and retransmissions, TCP effectively mitigates these issues.

Message Loss TCP encounters message loss when one or more segments fail to reach the receiver. Several factors can contribute to message loss, including network congestion, hardware failures, or transmission errors. To address this problem, TCP employs the following strategies:

- Positive Acknowledgment with Retransmission (PAR): When the sender transmits segments, it expects to receive acknowledgments (ACKs) from the receiver. If the sender does not receive an ACK

within a specified timeout period, it assumes that the segment was lost and retransmits it. This retransmission process continues until the sender receives the expected ACK.

- Timeout and Retransmission: TCP utilizes a timeout mechanism to detect segment loss. If an ACK is not received within a certain period, the sender assumes the segment was lost and retransmits it. The timeout duration is dynamically adjusted based on network conditions to optimize retransmission and minimize unnecessary delays.

Message Duplication occurs when a segment is unintentionally copied, resulting in multiple identical segments being received by the receiver. TCP implements mechanisms to detect and handle duplicates to ensure the integrity of the message:

- Sequence Number Comparison: TCP assigns a unique sequence number to each segment it transmits. The receiver checks the sequence number of incoming segments to identify duplicates. If a segment with the same sequence number is detected, it is considered a duplicate and discarded. This prevents the duplication of data within the reconstructed message.

- Acknowledgment Mechanism: TCP employs an acknowledgment (ACK) mechanism where the receiver sends ACK packets to the sender to confirm successful receipt of segments. This ensures that data sent from the sender is successfully received by the receiver.

2.5.1.3 Message Delivery Uncertainty in TCP Communication

While TCP ensures reliable message delivery within an established connection, it does not provide recovery options for long-term network outages. In the event of a connection breakdown, packets lost or duplicated during the outage are not recognized or recovered by TCP. This limitation poses a challenge for developers, as they must develop ad hoc strategies to cope with the uncertainty surrounding message delivery after a TCP connection failure. The last segment(s) of data transmitted before the failure may have been lost, and there is no built-in mechanism in TCP to handle this situation. Consequently, it becomes the responsibility of application developers to implement appropriate strategies for handling such cases, such as retransmission schemes, checkpointing, or alternative communication protocols.

Uncertainty Scenario Consider a scenario where a client and a server are communicating over a TCP connection. The client sends a series of messages to the server, and the server acknowledges each message upon successful receipt. However, due to a sudden network outage, the connection between the client and the server is lost.
In this situation, as depicted in figure 9, the client is unaware of whether the last message it sent was successfully received by the server. Since TCP does not provide recovery options for long-term network outages, the client cannot rely on the automatic retransmission of the last message. It becomes necessary for the client to develop an ad-hoc strategy to handle the uncertainty surrounding the delivery of the last

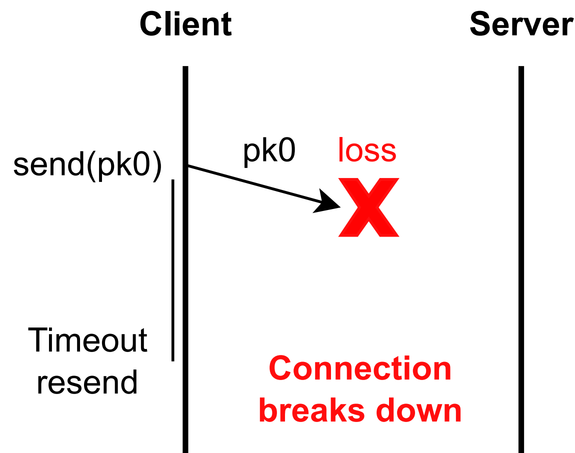


Figure 9: TCP packet received/lost uncertainty

segment.

One possible solution to address this uncertainty is for the sender and the receiver to maintain logs or records of each delivered message. The sender can add each sent message to the log and label them as "delivered" once an acknowledgment is received from the receiver. Similarly, the receiver can append each received message to its log and mark them as "delivered". In the event of a connection failure, the sender can refer to the log to determine the last successfully delivered message, while the receiver can examine each message to avoid duplicates. This log can serve as a reference point for resuming communication once the connection is reestablished.
By maintaining a message delivery log, developers can have a record of the messages that have been successfully transmitted and acknowledged by the receiver. This provides a mechanism to recover from a connection failure and continue communication from the point of interruption.

2.5.1.4 TCP Summary

In summary, while TCP does not offer built-in recovery options for long-term network outages, developers can implement solutions such as message delivery logs to handle the uncertainty surrounding the delivery of the last segment. These logs serve as a reference point for resuming communication and provide a means to track the status of message delivery even in the event of a connection failure.

2.5.2 TCP Connection Recovery Solutions

2.5.2.1 FT-TCP (fault-tolerant TCP)

FT-TCP [40] is a protocol designed to handle broken TCP sessions and mitigate the impact of connection failures. As figure 10 shows, this approach utilizes a logger to conceal these failures and maintains the logs in a stable storage located at the server.

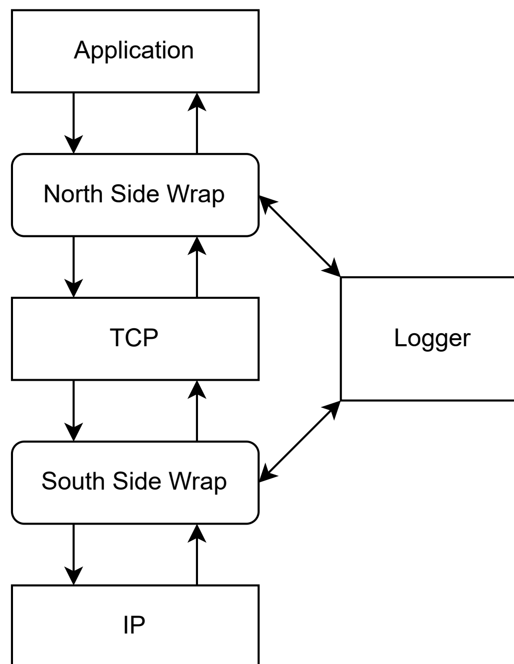


Figure 10: FT-TCP architecture [40]

The primary objective of FT-TCP is to provide fault tolerance in TCP connections. By employing the logger, FT-TCP masks connection failures, allowing the protocol to recover from interruptions and resume communication seamlessly. The logger plays a crucial role in recording the status of ongoing TCP sessions and the progression of data transmission.

One notable limitation of FT-TCP is its inability to recover from network failures. While it effectively addresses connection failures, it cannot handle scenarios where the network itself experiences disruptions. Consequently, the protocol can only guarantee at-most-once delivery of data in such cases. Also, the properties of the communication between the logger and the server in FT-TCP have a large and complex influence on the overhead [40].
Nevertheless, FT-TCP presents a valuable solution for maintaining robustness and reliability in TCP sessions by leveraging a logger to handle connection failures. It provides a mechanism to recover from common interruptions, ensuring that communication can resume smoothly. However, developers should be aware that network failures are outside the scope of this protocol's capabilities, and additional measures may be required to address such situations effectively.

2.5.2.2 RSocket

RSocket [17] is a solution that overcomes the recovery problems of TCP. It is implemented on top of TCP and ensures reconnection, transparently supporting network crashes by using control messages over a separate UDP channel, freeing the programmer from manually re-synchronizing. RSocket uses a unique connection identifier (CID) in order to distinguish between different incarnations. This means that it is not

oblivious: exactly-once can be ensured only by retaining node IDs for all peers ever involved in message exchanges.

As well as, iSAGA [41] is a protocol that uses the logging technique by saving each request to a stable storage to recover from node crashes. However, there is no guarantee that the recovered state is the same as the state before the crash.
Also, EOS [42] uses the logging mechanism on the client and the server sides, for the web based services in order to ensure exactly-once service.

Exactly-Once Middleware [43] is a solution that builds on RSocket and implements the exactly-once request-response interaction pattern using the logging technique combined with identified requests and message retransmission. This mechanism only supports the request-response pattern, and is not lightweight due to the persistent storage requirements.

2.5.3 TCP Connection Fail-over Protocols

To address the limitations of TCP in ensuring exactly-once delivery between connections, several fail-over protocols have been developed, which provide the necessary inter-connection information to achieve exactly-once guarantees.
Among the TCP fail-over protocols, Zandy et al. [15] used the idea of *Persistent Connections* to preserve the endpoint of a failed connection in a suspended state for an arbitrary period of time. Then, the protocol automatically reconnects using session fail-over to another process transparently. However, no guarantees are made beyond the defined time.

Similarly, Snoeren [16] used *Connection Migration* to migrate a connection session to another endpoint. Both Robust TCP (RTCP) and Exactly-Once Middleware used a similar technique, called *Connection Persistence*, when a TCP connection breaks [17, 43]. They used an out-of-band UDP connection recovery for exactly-once and FIFO. RTCP retains unique connection identifier (CID) in order to distinguish between different incarnations; this makes it non-oblivious, unlike Exon. FT-TCP [40] used a wrapper that saves the states in a *logger* (another process) to mask and recover the TCP connection, even under node failures. In general, all these fail-over protocols suffer from additional meta-data across connections, additional nodes, or their correctness is time-dependent.

2.5.4 UDP Based Protocols

Alternatively, there exist several UDP-based exactly-once protocols that provide exactly-once guarantees without the need for a connection-oriented approach. Since TCP is optimized for the general use, it

exhibits limitations in scenarios like bulk transfer, multicast, concurrent systems, computational grids, fast networks, etc. [44–49]. This motivated the foundation of reliable transport-level alternatives on top of UDP. Nevertheless, although these protocols managed to solve the ordering and performance reliability (congestion control) issues of TCP, they partially solved the exactly-once delivery. The reason is that they embraced the connection-based approach, which eventually led to the same TCP issues discussed above.

2.5.4.1 Reliable UDP

In particular, Reliable UDP (RUDP) [50] uses redundant connections over UDP. A connection failure is solved by signaling a timed state transfer to an upper layer protocol. If the latter does not transfer the state before the timer expires (after one second), the connection state is lost, and its buffers are freed—thus not ensuring exactly-once.

2.5.4.2 RTP

The RTP (Real-time Transport Protocol) [51] has the same issues as it provides UDP connections without exactly-once guarantees; but it relies on an RTP Control Protocol (RTCP) to maintain reliability through storing a lot of meta-data sessions with timeouts. The same holds for RBUDP [52] that must keep a tally of the packets to determine which packets must be retransmitted at the end of a bulk transmission under failure.

2.5.4.3 SCTP

SCTP (Stream Control Transmission Protocol) [44] is a transport layer protocol that provides reliable, message-oriented communication between two endpoints over an IP network. It was initially developed to address certain limitations of TCP and UDP in specific use cases.
SCTP is a transport protocol over IP, similar to TCP or UDP, although it is possible to tunnel it over UDP. Consequently, SCTP's utilization can be challenging due to limited support by routers, firewalls, and other network infrastructure elements.
SCTP employs a mechanism to detect and discard duplicate messages. The receiving endpoint keeps track of the sequence numbers of previously received messages. If a duplicate message is received, it can be identified and discarded to prevent duplicate delivery to the application layer.

2.5.4.4 UDT

UDT (UDP-based Data Transfer) [45] is a protocol specifically designed for high-speed data transfer over wide area networks (WANs). UDT is built on top of UDP (User Datagram Protocol) and aims to provide reliable, congestion-controlled, and high-throughput data transmission.
UDT offers reliable data delivery by implementing its own error detection and retransmission mechanisms on top of UDP. It uses acknowledgments and selective repeat techniques to ensure that all data packets

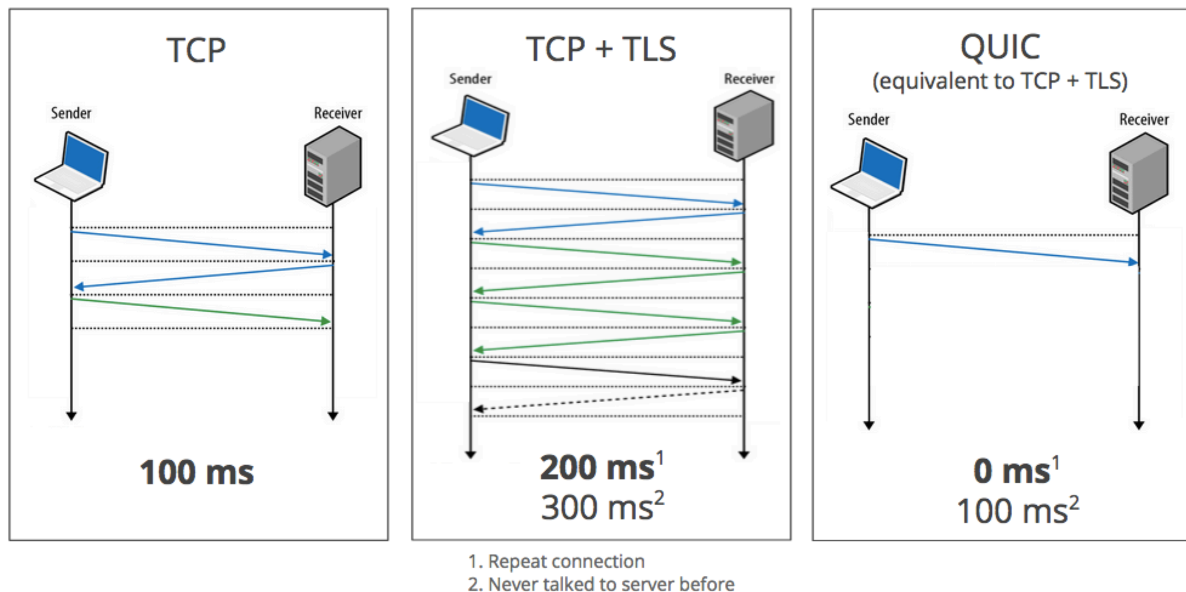


Figure 11: Zero RTT connection establishment with QUIC

are received correctly. UDT's reliability mechanisms provide end-to-end error recovery and retransmission, similar to TCP.

2.5.4.5 ENet

ENet [53] is a reliable, high-performance networking library designed for game development and real-time communication applications. It provides a simple and efficient API for sending and receiving data over unreliable UDP (User Datagram Protocol) connections.
ENet provides a reliable packet delivery mechanism over unreliable UDP connections. It divides the message data into smaller packets and sends them to the destination. ENet keeps track of each packet and ensures that all packets are received successfully by the recipient.

2.5.4.6 QUIC

QUIC [54] is a UDP-based application layer transport mechanism introduced by Google. QUIC was designed to be easily deployable and secure, and to reduce handshake and head-of-line blocking delays. Security and deployability are both helped by one of the key QUIC decisions – to base on top of UDP. As shown in figure 11, QUIC combines the cryptographic and transport handshake into one round trip when setting up a secure transport connection.

QUIC enables congestion and flow control, allows multiple data connections over the same UDP connection without HoL blocking, as depicted in figure 12 and also reducing the handshake delay at the beginning of a connection. QUIC's packet header carries information that help both loss recovery, where each QUIC

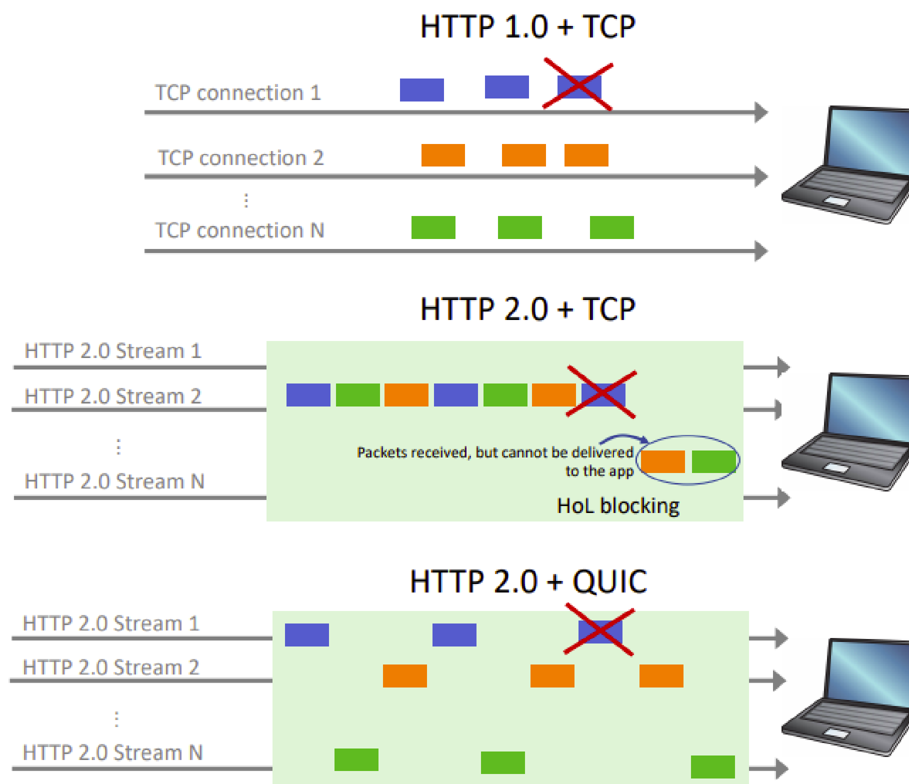


Figure 12: Multiplexing in QUIC, avoiding Head-of-Line Blocking. [55]

packet carries a new unique packet number that increases monotonically. Lost packets are then put into new outgoing packets that assigned new packet numbers.

In order to leverage the 0-RTT feature in QUIC, it is necessary to keep track of the identity of each communicating node. This can involve maintaining session IDs or other identifiers that can be used to establish trust between the client and server during subsequent communications.
The need to keep track of session IDs or other identifiers for each communicating node in order to use the 0-RTT feature in QUIC may be seen as a trade-off between performance and obliviousness. While QUIC is designed to provide better performance and security than traditional transport protocols, the need for session IDs means that the protocol may not be completely oblivious, as some information is required to be maintained and used for subsequent communications.

2.5.4.7 UDP Based Protocols Summary

Same as in TCP reliability optimizations, all the above UDP-based protocols tried to use a connection concept to maintain timed EO guarantees at the high price of storing significant amounts of meta-data. This makes them non-oblivious, contrary to what we present in Exon.

2.5.5 Logging in Message Delivery

Message logging refers to the practice of recording events, actions, and information related to the processing and delivery of messages within nodes on the network. In the context of message delivery, such as email servers, chat applications, and notification services, message logging involves keeping a record of various stages of message handling, including sending, receiving, processing, and any associated errors or exceptions.

Here is a basic process for message delivery between two nodes (Node A and Node B) in a distributed system, where logging plays a crucial role.

- Sending the Message: Node A composes and sends message M to Node B. Before sending, Node A creates a log entry in its local message log, indicating the intent to send message M. This log entry may include information like the timestamp, message ID, recipient (Node B), and the content of the message.
- Receiving and Acknowledging: Node B receives message M from Node A. Upon successful reception, Node B creates a log entry in its local message log, indicating the receipt of message M. This log entry contains details such as the timestamp, sender (Node A), message ID, and message content. Node B then sends an acknowledgment (ack) back to Node A to confirm the successful receipt of message M.
- Updating Message Status: Node A, upon receiving the acknowledgment from Node B, knows that message M has been successfully delivered. Node A can now update the status of message M in its local message log. This update could include marking the message as "delivered" or "acknowledged." By marking the message as delivered, Node A ensures that it will not attempt to resend the same message to Node B in the future.

Throughout these steps, both Node A and Node B maintain local message logs that record the events and actions related to message delivery. These logs capture crucial information about the communication process, including timestamps, participants, message content, and acknowledgments. The logging process helps maintain the reliability and consistency of message delivery. In case of any issues, failures, or the need for auditing, the logs can be reviewed to understand the sequence of events, identify potential errors, and ensure that messages are being correctly processed and delivered.

Implementing a message delivery log introduces certain costs in terms of processing and memory overhead [56]. The logging mechanism plays a crucial role in maintaining a record of delivered messages, aiding in recovery from connection failures and ensuring message reliability. However, it is important to consider the impact of logging on system performance and resource consumption. Another challenge associated with message logging is the potential necessity for intricate and resource-intensive protocols

aimed at ensuring accurate recovery. These protocols are designed to eliminate any inconsistencies among processes, safeguarding the integrity of the recovered data. [55].

2.5.5.1 Processing Overhead

Processing overhead associated with message logging arises from the frequent comparisons required between incoming messages and the log to identify duplicates. As the number of messages and the size of the log increase, this comparison process can become computationally intensive, potentially affecting the overall system performance [57]. In scenarios involving a large number of nodes, the processing overhead of duplicate detection can significantly impact the system's efficiency.

2.5.5.2 Memory Consumption

Implementing a message delivery log also incurs memory consumption. The log needs to store information about delivered messages, including unique identifiers and delivery status. As the number of messages and the size of the log grow, the memory requirements increase accordingly. In distributed systems with a high message throughput and a substantial number of nodes, the memory usage associated with maintaining a comprehensive log can become significant.
When dealing with a large volume of messages and numerous nodes, careful consideration is required to manage the memory usage of the message delivery log effectively. Balancing the need for accurate message tracking with the available memory resources is crucial to ensure optimal system performance and prevent potential memory-related issues.

2.5.5.3 Considerations in Message Logging

The concerns mentioned are not limited to resource-constrained devices; but extend to server environments as well. The efficient management of memory usage in message logging is a common challenge faced across various computing platforms.

By understanding the processing and memory implications of logging in message delivery, developers can make informed decisions regarding the implementation and optimization of the logging mechanism. These considerations are particularly important in scenarios involving high message volumes, large-scale distributed systems, and resource-constrained environments.
Even when checkpointing is combined with logging, it introduces challenges, particularly as the system scales with an increasing number of nodes and messages. The use of checkpointing imposes additional overhead in terms of processing and storage demands [56].

2.6 Message-oriented Middleware

There is a recent trend of providing exactly-once guarantees via using message queue (MQs) brokers such as Apache Kafka [58], AMQP [59], RabbitMQ [60], ActiveMQ [61], ZeroMQ [62], etc. However, these protocols are very heavy-weight as they stand as middlewares that support many underlying protocols like TCP, UDP, MQTT [63], and PGM [64]; and they implement several messaging patterns. Many of these middlewares make use of huge infrastructure to provide the so-called "effective exactly-once" to the application through using node fault tolerance and distributed commit log [65], as a cluster to offer high scalability. ZeroMQ is exceptionally broker-less, but it may drop queued messages upon disconnects, although it attempts TCP reconnections.

Another way to ensure reliable message delivery is by using a message broker [66]: an intermediate between sender and receiver that stores and orders messages until the consuming application can process them. Brokers can become a single point of failure, although it can recover from its own crashes by storing the messages in stable storage. Many queue-based messaging brokers were developed, such as Apache Kafka [58], AMQP [59], RabbitMQ [60], ActiveMQ [61], ZeroMQ [62], MQTT [63].

2.6.1 Apache Kafka

Apache Kafka [58] is an open-source distributed streaming platform developed by the Apache Software Foundation. It is designed to handle high-volume, real-time data feeds and offers scalable, fault-tolerant, and high-performance messaging capabilities. Kafka provides a publish-subscribe model where producers publish streams of records to topics, and consumers subscribe to those topics to process the data.
The core abstraction in Kafka is the commit log, which is a distributed, fault-tolerant, and append-only data structure. Records published by producers are appended to the commit log and organized into topics, which can be partitioned across multiple Kafka brokers. Each partition is replicated across a configurable number of brokers, ensuring high availability and fault tolerance.

Kafka allows for both real-time stream processing and event-driven architectures. It enables applications to consume data in real-time as it arrives, process it, and produce derived data streams. This makes Kafka well-suited for use cases such as log aggregation, stream processing, event sourcing, data integration, and messaging systems.
Key features of Apache Kafka include:

- Scalability: Kafka is designed to handle high-throughput, high-volume data streams by allowing horizontal scaling across multiple brokers and partitions.

- Fault Tolerance: Kafka provides built-in replication and leader election mechanisms to ensure data

durability and fault tolerance. If a broker fails, another broker automatically takes over as the leader for the affected partitions.

- Durability: Kafka retains published records for a configurable period of time, allowing consumers to rewind and read historical data. This durability makes Kafka suitable for use cases requiring data replay and recovery.

- Exactly-Once Semantics: While Kafka guarantees at-least-once message delivery by default, it also provides mechanisms such as transactional writes and consumer offsets management to achieve exactly-once semantics.

Apache Kafka, one of the most widely used systems, is designed around a distributed commit log [65] as a cluster to offer high scalability. It offers high throughput and low latency for both publishing and subscribing, which is essential to support real-time data feeds. However, the inventors themselves said that Kafka does not guarantee exactly-once delivery: "Kafka only guarantees at-least-once delivery. Exactly-once delivery typically requires two-phase commits and is not necessary for our applications." [58].

2.6.2 ZeroMQ

ZeroMQ [62] is a lightweight messaging library aimed for scalable distributed applications. Unlike most other queue-based messaging systems, ZeroMQ is broker-less. It expands the concept of socket with built-in messaging patterns: Request/Reply, Publish/Subscribe, Pipeline and Exclusive pair. However, regarding message delivery guarantees, even though ZeroMQ attempts TCP reconnections, it may drop queued messages upon disconnects.

ZeroMQ [62], is an open-source messaging library that provides high-performance, asynchronous communication between applications. It offers a socket-based API and supports various messaging patterns such as publish-subscribe, request-reply, and push-pull.
When it comes to message delivery guarantees, ZeroMQ provides different levels of reliability based on the chosen messaging pattern and socket type:
Best Effort (Fire-and-Forget):

- PUB/SUB Pattern: In the publish-subscribe pattern, ZeroMQ operates on a "best-effort" basis. Publishers send messages to subscribers, but there is no built-in mechanism to ensure delivery or guarantee that every subscriber receives every message. Subscribers join the pattern and receive messages that publishers send after they have subscribed.

- PUSH/PULL Pattern: Similarly, in the push-pull pattern, ZeroMQ employs a best-effort approach. Messages are pushed from the sender (PUSH socket) to one or more receivers (PULL sockets), but there is no built-in mechanism to guarantee that every message will be delivered to every receiver.

Reliable Delivery (At-Least-Once):

- REQ/REP Pattern: The request-reply pattern in ZeroMQ offers at-least-once delivery semantics. When a request is sent from a REQ socket to a REP socket, ZeroMQ ensures that the request is delivered to the REP socket and a reply is received. However, due to the underlying network and system conditions, it is possible to encounter message loss or duplicate replies. Application-level coordination is required to handle potential duplicates or lost messages.

It's important to note that ZeroMQ is designed to be a lightweight messaging library, optimized for high throughput and low latency, rather than focusing on complex reliability guarantees. It provides building blocks for building distributed systems and applications, and the responsibility of achieving higher reliability or delivery guarantees often lies with the application developers.
In summary, ZeroMQ provides various message delivery guarantees depending on the chosen messaging pattern and socket type. It offers best-effort delivery for publish-subscribe and push-pull patterns, at-least-once delivery for request-reply pattern, and basic one-to-one messaging without built-in reliability for the exclusive pair socket type. Developers need to consider their application requirements and implement any necessary reliability mechanisms on top of ZeroMQ if stronger guarantees are needed.

2.6.3 MQTT

MQTT [67] is lightweight messaging protocol developed by IBM, supporting the publish-subscribe pattern. MQTT offers three reliability modes, including exactly-once, by a Packet Identifier in its Variable Header. In this mode, the receiver of a PUBLISH packet acknowledges receipt with a two-step acknowledgement process. However, the receiver stores a reference to the packet identifier in order to avoid processing the message a second time.
However, this technique generally requires total table storage proportional to N^2 for N inter-communicating nodes [68], since a node, especially the receiver, has to keep all the IDs of the nodes contacted with, in order to avoid message duplication, and are therefore ineffectively suited to large distributed shared-memory machines. On the other hand, TCP does not address link failures properly, it breaks the connection if connectivity is lost for some duration, which may lead to message loss and duplication, and consequently it need a way to recover from broken TCP connections.

2.7 Oblivious Transport-level Messaging

Most of the messaging techniques and protocols mentioned above are heavyweight and/or ensure at-most-once or at-least-once message delivery. The few that ensure exactly-once are not oblivious: they require message identifiers or peer/connection specific information that needs to be kept to ensure de-duplication. This hurts scalability to many hosts because this information grows over time, as each host communicates with new hosts. As a result, for this de-duplication technique to work properly, "some

state" is needed (e.g. counter) between different connection incarnations [14].

Traditionally, exactly-once delivery has been achieved by keeping track of delivered messages. However, this approach incurs high storage and time overhead, particularly as logs grow larger. In edge environments with thousands of devices, this overhead becomes even more problematic. Furthermore, relying solely on TCP for exactly-once delivery is not recommended in hostile environments, as TCP does not provide guaranteed exactly-once delivery under failure conditions.

Indeed, Attiya et al. [69] proved that if the nodes have unbounded memory, a three-way handshake oblivious protocol exists, whereas a two-way handshake oblivious protocol does not exist. The same paper proved also that if a bound on maximum packet lifetime (MPL) exists and is known, then a two-way handshake oblivious protocol is possible, but at least MPL must elapse between the time two consecutive incarnations are established—which is impractical in most networked applications. Therefore, most reliable and efficient protocols are three-way handshake based, but they expose full-duplex connections in the API, and depend on timing assumptions for obliviousness.

Although Attiya's three-way handshake oblivious unbounded memory protocol [69] is only require a single integer as node state between incarnations, however, Attiya's protocol is classically connection based, with API visible full-duplex connections, which prevents safely ensuring exactly-once message delivery, as the receiving side can concurrently close the connection, preventing delivery.

Therefore, in large scale systems (millions and billions of machines), this become a harsh condition to be fulfilled, especially in IoT and edge systems that have bounded memory, where no incarnation management protocol exists. However, an oblivious protocol is possible with no need to retain connection specific information between incarnations, but just keeping a single unbounded clock for the whole node [69].

2.8 Discussion

Pat Helland's visionary perspective on activities and entities highlights the huge complexity of building reliable and scalable distributed systems. In the pursuit of creating systems that handle massive number of nodes, the importance of a protocol that not only ensures reliability but also scales gracefully becomes evident. As systems grow to encompass millions of nodes, such as in telecommunications, IoT, and automotive networks, the challenges increase.

Distributed transactions, a cornerstone of ensuring data integrity across distributed environments, encounter challenges caused by network partitions, latency, and node failures. A significant downside lies

in their substantial nature, which has the potential to introduce performance overhead and limit scalability. Distributed transactions frequently necessitate coordination and consensus across numerous nodes, which leads to overhead from communication and synchronization processes.

The need to guarantee data consistency, even in the face of unreliable nodes and network failures, has driven the exploration of various protocols. Existing transport-level protocols, including TCP, address reliability concerns to a certain extent. However, their inability to offer liveness guarantees and exactly-once message delivery in a fragile environment poses limitations.

Message-oriented middleware, such as AMQP, MQTT, and CoAP, introduces strategies to ensure message delivery in different ways. Commonly, a log-based approach is employed to track sent and received message IDs. However this method has some caveats, as connection breakdowns can lead to message loss or duplication. The challenge becomes even more pronounced in scenarios where maintaining a history of node IDs is essential to prevent message duplication. This work is done by the programmers, or there are some protocols the programmers can use and integrate them with their protocols to do what TCP cannot do it, e.g. reconnect with the same connection ID, save log to a durable memory...

Existing research, do not fully address the need for an oblivious, exactly-once message delivery protocol in highly scalable systems. This inspired the initiation of our research journey, aimed at bridging this gap. Our approach, rooted in a message-based four-way protocol, sets the foundation for exactly-once delivery and obliviousness. Augmenting this approach with on-demand half-duplex soft-connections, enhances efficiency without compromising on reliability, obliviousness, or exactly-once guarantees.
In summary, our work represents an advancement towards Pat Helland's vision of scalable and reliable systems. With a focus on achieving an efficient, oblivious, and exactly-once messaging protocol, we contribute to the evolving landscape of distributed systems, enabling the creation of powerful networked applications with high levels of reliability and scalability.

3 Exon Protocol

In the realm of messaging protocols, ensuring exactly-once message delivery has long been a big challenge. As explored in Chapter 2, existing solutions present trade-offs that hinder their seamless integration into large and scalable systems.
Logging, a method often used, introduces processing and memory overhead, impeding the efficiency of such systems. On the other hand, distributed transactions, while effective, exhibit the drawback of inducing blocking in distributed environments. Significantly, message-oriented middleware such as AMQP, MQTT, Kafka, etc., relies on a similar logging technique and maintains per-peer connection data to forestall message duplication. However, these approaches still unable to adapt in the extensive scalable distributed systems.
This chapter comprehensively introduces the Exon protocol, a pioneering solution addressing the challenges of messaging in large-scale distributed systems, that aligns with Pat Helland's vision [2] of the need for reliable messaging protocol in large scale systems, and embraces with Attiya's notion that an oblivious protocol is possible with no need to retain connection specific information between incarnations, but just keeping a single unbounded clock for the whole node [69].

It begins by presenting the System Model, then the Overview section follows, that provides a glimpse into the operational mechanics and functioning of the protocol. After that we delve to the Algorithm, providing an in-depth understanding of its inner workings. Rigorous Correctness Proofs reinforce the protocol's reliability and accuracy guarantees. The chapter then delves further into the underlying intuition behind pivotal aspects, enhancing the reader's grasp of Exon's mechanisms. Lastly, Advanced Properties of Exon are unveiled, such as the soft- half-connections and obliviousness, showcasing the protocol's capabilities.

3.1 Introduction

Exon is a host-to-host message-based protocol that is optimized to guarantee the exactly-once (EO) delivery of these messages. The core idea revolves around employing slots and tokens. Visualize a slot as a virtual placeholder. When the need arises to transmit a particular item to an individual, a request is made

for a placeholder. Upon receiving the request, the recipient generates the necessary placeholder dedicated to that specific item and conveys its identification back to the requester. Subsequently, numbered with this placeholder's ID, the requester creates a container/box designed solely for accommodating the placeholder. By this concept, items cannot be transmitted without being encapsulated within a container. Furthermore, a container cannot be received unless a corresponding placeholder is pre-established, ready and waiting to accommodate it.

In a nutshell, Exon has a combination of components which allows ensuring EO while being network and memory efficient, namely:
Soft- half-connections: connections are useful to group identifiers like sequence numbers and achieve performance. We have what we call soft- half-connections (s-connection), that group messages from the same sender-receiver pair, created on-demand if messages are requested to be sent. For performance, the s-connection can be discarded if there are no pending unacknowledged messages, after some non-short timeout. EO correctness is ensured without timeouts though (e.g., no TIME_WAIT as in TCP).
Oblivious: Exon achieves EO correctness without the need to keep s-connection-related information forever, keeping only a single integer per node as permanent state, when no s-connections are present.
Order-less: to be more generic, Exon is deprived from unnecessary ordering restrictions of messages. Message ordering (e.g., FIFO) can be implemented on top of Exon if required.
In the following, we emphasize Exon in more detail.

3.2 System Model

We consider a networked or distributed system of any number of nodes. A node can have wide or constrained capacities, but a local memory is required. Nodes can be long-lived sticky members of some service or transient ones (e.g., vehicular networks, mobile, IoT...). Nodes can crash but will eventually recover with the content of the last state prior to the crash. In this case, a stable persistent storage is assumed.

In our system model, we intentionally narrow our focus to the aspects of messaging protocol design and reliability, specifically omitting the complex domain of node crashes and their associated recovery mechanisms. This decision comes from the reality that the realm of crash recovery constitutes a distinct and complex domain in itself. Developers have introduced sophisticated crash recovery mechanisms within large-scale distributed systems. These mechanisms are precisely designed to counter the challenges posed by node crashes and ensure data integrity and system stability.

On the network side, any node can communicate with any other node via a network (e.g., Local Area Network, Wide Area Network, Wireless Sensor Network...). The network is asynchronous, with no global

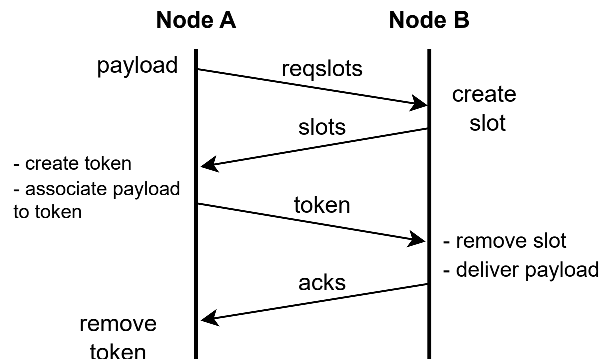


Figure 13: Fault-free communication scenario.

clock, no bound neither on the time it takes for a message to arrive, nor on the processing speeds. The network is unreliable, messages can be lost, duplicated, or reordered (but are not corrupted). The network may have long partitions, but these will eventually heal. Exon assumes the existence of an underlying transport-level communication channel—or any equivalent non-IP abstraction—to send messages in any form (bytes, datagrams, etc.); although our current Exon-lib is implemented on top of UDP.

The decision to implement Exon over the UDP protocol, rather than TCP, comes from strategic consideration of the unique characteristics and requirements of our reliable messaging protocol. One of the significant factors influencing this choice is the desire to avoid the potential Head of Line (HoL) blocking issue in TCP. Furthermore, Exon itself is designed to provide a robust mechanism for ensuring reliable message delivery, integrating TCP's inherent reliability with Exon's designed reliability could result in unnecessary time overhead, given that both mechanisms would strive to accomplish similar goals. By opting for UDP, we strike a harmonious balance. UDP's lightweight and connectionless nature mitigates the HoL blocking concern, while Exon's inherent reliability negates UDP stands as an abstract protocol catering to upper-layer protocols that can function effectively without requiring the specific reliability features associated with TCP.

3.3 Overview

Assume that node A intends to transmit a message to node B. In a fault-free scenario, this intention would trigger the initiation of a communication between the two nodes, as depicted in figure 13:
(1) Node A starts the communication where it has a message to send to Node B, it sends a reqslots message, (2) Node B creates a *slot*, i.e., a placeholder that allows sent message to be accepted, and sends a slots message to Node A. Then, (3) Node A receives the slots message, and creates a *token*, i.e., a container that associates a specific "payload", sends a token message that contains an encapsulated "payload" to Node B. This "payload" is associated with this specific *token*. After that, (4) when Node B receives the token message, it removes the *slot* specific to this *token*, delivers the "payload" to the specific

upper layer application, and sends an acknowledgment “ack” to Node A, where Node A can safely remove the *token*.
In order to optimize the slot-at-a-time request overhead, Node A can request a window of slots in advance using empty message place-holders, we call *envelopes*. Node A can associate message to these ready envelopes with reserved slots when needed.
Thus, we call it as slots message, since it represents many slots. Furthermore, it is worth noting that the term “Token” is not pluralized as “Tokens” intentionally, this is because each Token inherently corresponds to a unique payload, rather than merely representing an identification marker.

A crucial distinction exists between a *slot* and a slots message, as well as between a *token* and a token message. Notably, a *slot* itself is not transmitted from node B to node A. Instead, when node B creates a *slot* in response to a reqslots from node A, only the ID of that *slot* is conveyed to node A, encapsulated within a message categorized as a slots message.
Similarly, a *token* is not directly transmitted; rather, the ID of the *token*, along with its associated “payload”, is dispatched to node B as part of a token message.

3.4 The Algorithm

We now describe the algorithm details of Exon. To simplify the presentation, we refer to the corresponding lines in Algorithm 1. We also exemplify the algorithm with a simple instance, which we walk through to clarify how the state is changing at each step.
Algorithm 1 is the protocol for a generic node i. It conveys the node state, the types used in defining it, the atomic actions (both for when a send is requested locally and when messages arrive), and the called auxiliary procedures. The algorithm also presents a procedure that runs periodically, to show in a minimal way how to cope with message loss (an actual implementation could have, e.g., timeouts per half-connection to trigger these sends).

3.4.1 Notations and Definitions

Each node i has an Exon state. The state is a node-wide clock (ck_i) keeping a monotonically increasing integer, and a pair of maps: S_i keeping sender-side half-connection records of type $\mathbb{S}$ and R_i keeping receiver-side half-connection records of type $\mathbb{R}$. We use an i subscript to denote node state variables or actions and unsubscripted names for local temporary variables; we use := for assignment, typically to a state variable, and = for a let binding which binds a name to a value.
While in abstract, and for correctness, we use the concepts of slots, envelopes and tokens as globally unique entities, which are kept in the node state grouped in the sender-side or receiver-side half-connection records. We now define these concepts and describe how they are stored in nodes.

```
1  types
2      𝕀, node identifiers
3      𝕄, message payloads
4      𝕊 : record {
5          sck : ℕ, sender clock
6          rck : ℕ, receiver clock
7          msg : 𝕄*, messages queued to send
8          env : ℕ*, list of available envelopes
9          tok : ℕ ↪ 𝕄, tokens with messages
10     }, sender-side connection record
11     ℝ : record {
12         sck : ℕ, sender clock
13         rck : ℕ, receiver clock
14         slt : 𝒫(ℕ), set of available slots
15     }, receiver-side connection record
16 parameters
17     N : ℕ, number of slots requested in advance
18 state:
19     ck_i : ℕ = 0, node clock
20     S_i : 𝕀 ↪ 𝕊 = ∅, map of sender-side records
21     R_i : 𝕀 ↪ ℝ = ∅, map of receiver-side records
22 on EOsend_i(j, m)
23     if j ∉ dom(S_i) then
24         S_i[j] := 𝕊{sck : ck_i, rck : 0, msg : [m],
                     env : [], tok : ∅}
25         requestSlots_i(j)
26     else
27         c = S_i[j]
28         if c.env ≠ [] then
29             e = c.env.dequeue()
30             c.tok[e] := m
31             send_{i,j}(token, e, c.rck, m)
32             if |c.env| = N − 1 then
33                 requestSlots_i(j)
34         else
35             c.msg.enqueue(m)
36 proc requestSlots_i(j)
37     c = S_i[j]
38     n = N + |c.msg| − |c.env|
39     if n > 0 then
40         l =  if c.tok ≠ ∅ then min(dom(c.tok))
41              else if c.env ≠ [] then c.env[0]
42              else c.sck
43         send_{i,j}(reqslots, c.sck, n, l)
44     else if c.tok = ∅ and c.msg = [] then
45         send_{i,j}(reqslots, c.sck, 0, c.sck)
46         ck_i := max(ck_i, c.sck)
47         S_i.remove(j)
48 on receive_{j,i}(reqslots, s, n, l)
49     if j ∉ dom(R_i) then
50         R_i[j] := ℝ{sck : s, rck : ck_i, slt : ∅}
51         ck_i := ck_i + 1
52     c = R_i[j]
53     c.slt := {m ∈ c.slt | m ≥ l}
54     if n > 0 then
55         if s + n > c.sck then
56             c.slt.union({c.sck, ..., s + n − 1})
57             c.sck := s + n
58         send_{i,j}(slots, s, c.rck, n)
59     if c.slt = ∅ then
60         R_i.remove(j)
61 on receive_{j,i}(slots, s, r, n)
62     if j ∉ dom(S_i) then
63         send_{i,j}(reqslots, ck_i, 0, ck_i)
64     else if s = S_i[j].sck then
65         c = S_i[j]
66         c.rck := r
67         c.env.append([s, ..., s + n − 1])
68         c.sck := s + n
69         while c.env ≠ [] and c.msg ≠ [] do
70             e = c.env.dequeue()
71             m = c.msg.dequeue()
72             c.tok[e] := m
73             send_{i,j}(token, e, c.rck, m)
74         requestSlots_i(j)
75 on receive_{j,i}(token, s, r, m)
76     if j ∈ dom(R_i) then
77         c = R_i[j]
78         if r = c.rck and s ∈ c.slt then
79             c.slt.remove(s)
80             deliver_i(m)
81     send_{i,j}(ack, s, r)
82 on receive_{j,i}(ack, s, r)
83     if j ∈ dom(S_i) then
84         c = S_i[j]
85         if r = c.rck and s ∈ dom(c.tok) then
86             c.tok.remove(s)
87 periodically
88     for (j, c) in S_i do
89         for (s, m) in c.tok do
90             send_{i,j}(token, s, c.rck, m)
91         requestSlots_i(j)
92     for (j, c) in R_i do
93         send_{i,j}(slots, c.sck, c.rck, 0)
```

Algorithm 1: Exon Algorithm

Definition 1 (Slot)**.** A slot with id (j, i, s, r) represents the obligation of node i to deliver a message from j tagged by this id, or for node j to explicitly waive this obligation.

A (j, i, s, r) slot is kept at node i as a j entry in the R_i map, having $\textsf{rck} = r$ and $\textsf{slt}$ containing s.

Definition 2 (Envelope)**.** An envelope with id (i, j, s, r) represents the option (but not the obligation) of node i to generate a token with this same id (consuming the envelope) to which a user message is associated

An (i, j, s, r) envelope is kept at node i as a j entry in the S_i map, having $\textsf{rck} = r$ and $\textsf{env}$ containing s. For uniformity and compactness of presentation, this $\textsf{env}$ field is a list of integers, when an actual implementation would need only a pair of integers, as this list always contains a contiguous sequence.

Definition 3 (Token)**.** A token with id (i, j, s, r) associated with user message m, created from an envelope of this same id, represents the obligation of node i to request j to deliver message m until acknowledged.

An (i, j, s, r) token associated with user message m is kept at node i as a j entry in the S_i map, having $\textsf{rck} = r$ and $\textsf{tok}$ mapping s to m.

3.4.2 Messaging Steps

We now present the four main messaging steps of Exon by referring to Algorithm 1. For simplicity, we assume the communication is occurring between a sender node "A" and a receiver node "B". To ease tracking the steps of Algorithm 1, we added Figure 14.

Suppose that node A and node B are already in communication with other nodes, and as a result, their clocks (ck) are 30 and 20, respectively. Therefore, there is a formula must be calculated to know how many slots should be requested. The formula is as follows:
$n = N + |\textsf{msg}| - |\textsf{env}|$

- N, is the number of standby slots that should be presented in Node A.
- $|\textsf{msg}|$, is the number of messages in the message queue that needs slots.
- $|\textsf{env}|$, is the number of available envelopes that could be used to create tokens for the queued messages.

Furthermore, assume that the number of slots in advance is $N = 4$. Additionally, there is only one message in the message queue ($\textsf{msg}$), and no envelopes have been created at node A yet, therefore, $n = N + |\textsf{msg}| - |\textsf{env}| = 4 + 1 - 0 = 5$, window $n = 5$.

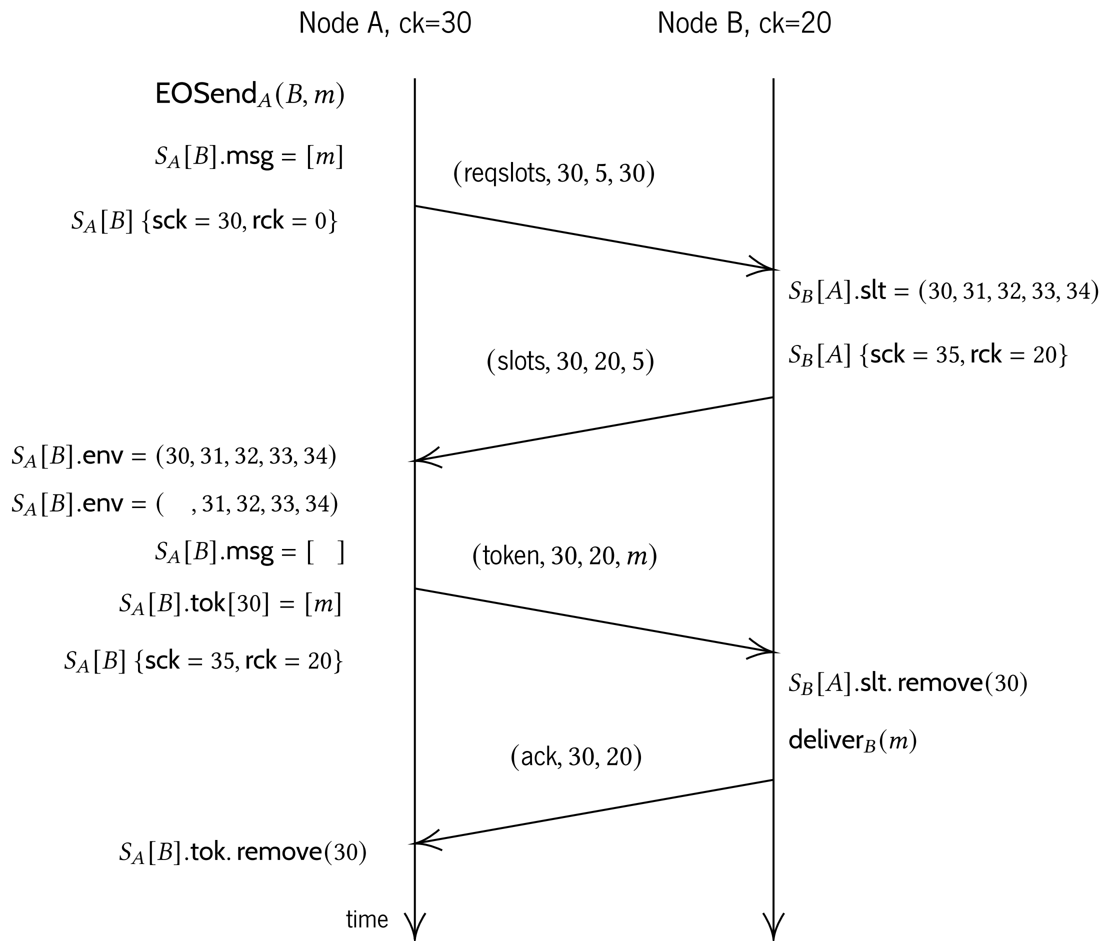


Figure 14: Exon step-by-step example for node A communicating with node B.

3.4.2.1 Step 1 - Requesting Slots 15

Step 1. Requesting Slots: Sender node A does not hold a previous S-record for receiver node B. Consequently, the former creates an S-record for B and requests "n" slots from B via reqslots.

The function EOSend$_A(B, m)$ is initially invoked at node A to initiate a payload transmission to node B. Subsequently, node A generates an S-record (sender-side half-connection) (line 24: $S_i[j] := \mathbb{S}\{\mathsf{sck} : ck_i, \mathsf{rck} : 0, \mathsf{msg} : [m], \mathsf{env} : [], \mathsf{tok} : \emptyset\}$) for node B as it is the first time the former is sending data to the latter. The message content, i.e., the payload "m", is enqueued in a message queue msg. Subsequently, node A invokes the function requestSlots at line 25.

In the requestSlots function, the value of the variable "n" is determined by subtracting the number of available envelopes from the sum of N (the number of envelopes in advance) and the number of messages in the message queue "msg". If n is greater than zero, a reqslots message is sent with certain values. However, if there are no tokens or messages in the message queue, a different set of values is sent in the reqslots message:

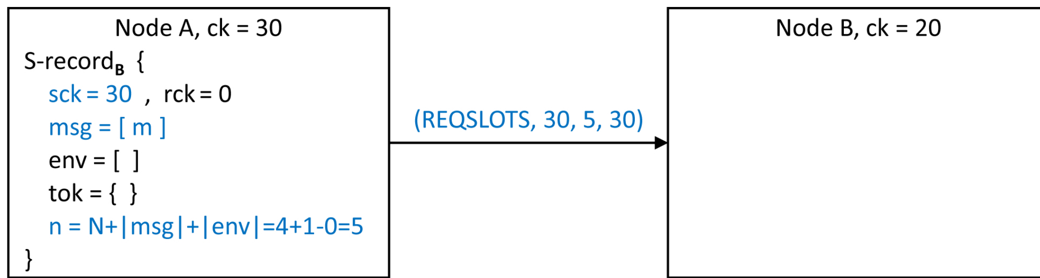


Figure 15: Assigns values to the S-record elements, calculates "n", and sends a reqslots message.

- In the first case, when "n" is greater than zero, the reqslots message includes the variables sck (sender clock), "n" (number of slots), and "l" (the Garbage Collection frontier, which guides node B in removing old slots that node A has no tokens for).

  The value of "l" is determined based on one out of three conditions. The first condition where "l" is assigned the lowest token number if there exist tokens for node B. The second condition where "l" is set to the number of the first envelope if there are no tokens for node B. The third condition where "l" equal to sck. The three conditions will be detailed and illustrated late in section 3.6.2.1.

- In the second case, when "n" is not greater than zero and there are no tokens or messages in the message queue, this occurs "periodically" (line 87) when node A dispatches all its tokens and already possesses N envelopes. The purpose is for node B to remove the R-record of node A after a given amount of time. In this case, a reqslots message is sent with sck, "n" set to zero, indicating that the node is not requesting slots, but this is for garbage collection purposes, and "l" is set to sck. When the reqslots message reaches node B, all of the slots will be removed based on sck (line 53: $c.\mathsf{slt} := \{m \in c.\mathsf{slt} | m \geq l\}$), and now that there are no slots, node B can remove the R-record of node A safely.

Message Loss/Duplication Message loss and duplication can occur due to various factors and conditions within a communication system. These issues may arise from network congestion, hardware failures, software glitches, or other unforeseen circumstances.

Assuming that the reqslots message is lost, the protocol incorporates a retransmission mechanism during the "periodically" phase to mitigate such occurrences. In this process, the protocol checks each node to determine if there are any pending tokens to be sent. Subsequently, it invokes the requestSlots function (lines 88-91), as seen below:
Within the requestSlots function, if there are new messages enqueued for transmission, a new reqslots message is generated. It is important to note that this retransmission does not impact the correctness of the previous reqslots message. This is because, at the receiving end, the creation of slots is performed using the union operation, which is associative.

```
88 for (j, c) in S_i do
89     for (s, m) in c.tok do
90         send_{i,j}(token, s, c.rck, m) ;
91     requestSlots_i(j) ;
```

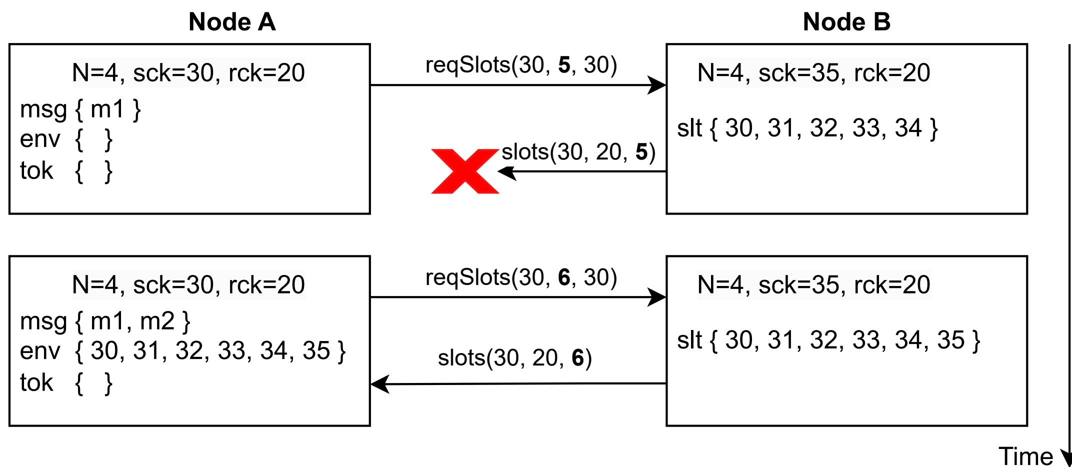


Figure 16

To illustrate this point clearly, as shown in the figure 16, let us consider a scenario where node A needs to send a message, denoted as m1, to node B for the first time. At this point, neither envelopes nor tokens exist. Message m1 is queued in the message queue "msg".

Subsequently, node A initiates a request for n=5 slots from node B. Node B responds by creating the requested 5 slots. However, the slots message gets lost in transit.

Simultaneously, another message, m2, is added to the message queue. As time progresses, periodically (as indicated in line 91 by $\mathsf{requestSlots}_i(j)$), the **requestSlots** function is invoked. In this instance, the calculation of n results in a value of 6. Consequently, the reqslots message, indicating the need for 6 slots, is dispatched to node B.

Now, since node B had already created 5 slots for node A previously, upon receiving another reqslots message specifying n=6, the "union" abstract operation will simply append the sixth slot to the slt queue.

In the event of duplicate reqslots messages, where slots with the same sequence are already present, the condition at line 55 ($s + n > c.\mathsf{sck}$) will not hold true since the sck is already incremented. Consequently, no new slots will be created. Instead, the protocol will simply send the previously generated slots with the same sequence (line 58: $\mathsf{send}_{i,j}(\mathsf{slots}, s, c.\mathsf{rck}, n)$). This is done to prevent the occurrence of duplicate reqslots messages. However, if the slots message is lost, this mechanism allows for the possibility of resending the slots message.

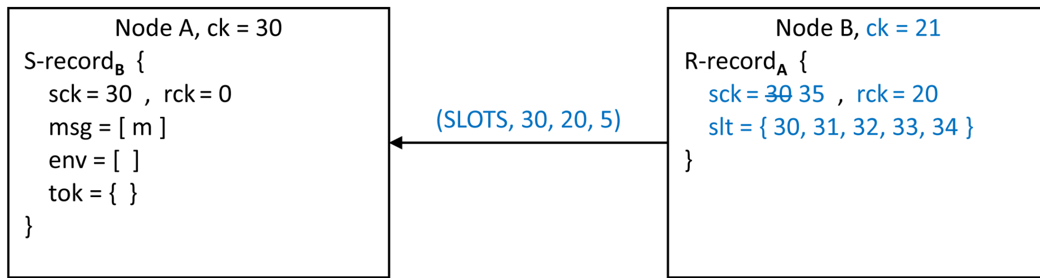


Figure 17: Assigns values to the R-record elements, modifies the clocks, creates slots, and sends a slots message.

3.4.2.2 Step 2 - Sending Slots 17

Step 2. Sending Slots: Node B creates the requested slots and sends them back to A. If B has a prior record for A with outdated or consumed slots, it checks if garbage collection can be done.

Upon receiving the reqslots message (line 48: $\textbf{receive}_{j,i}(\text{reqslots}, s, n, l)$), node B first checks if there exists a receiving record R-record (receiver-side half-connection) for node A from a prior communication. If not, node B proceeds to instantiate the R-record for node A (line 50: $R_i[j] := \mathbb{R}\{\textbf{sck} : s, \textbf{rck} : ck_i, \textbf{slt} : \emptyset\}$). In the R-record at node B, the variable **sck** is assigned the received clock "s" from node A. Furthermore, the variable **rck** is assigned the global clock **ck** as an incarnation number, and subsequently the global clock **ck** incremented. And of course, an empty slots (slt) set is created.
The assignment of the variable **rck** to the global clock **ck** as an incarnation number serves the purpose of uniquely identifying the current incarnation or version of the system. By setting **rck** to **ck**, we establish a clear link between the incarnation number and the global clock, making it easier to track and manage different versions or states of the system.
Subsequently incrementing the global clock **ck** is done to ensure that each incarnation of the system has a distinct identifier. This incrementation signifies a transition to a new state or version of the system, where **ck** reflects the most up-to-date incarnation. In essence, it helps maintain chronological order and provides a basis for recognizing when changes or updates occur in the system.
If node B is already has a previous communication with node A and has an R-record for it, therefore, it needs to verify whether any slots are still available and not yet removed for that node. This situation may arise if node A had requested slots in the past but did not utilize them and it removes the envelopes of those slots and advanced the clock.
This scenario occurs when node A intends to retire/quit. At that time, it sends all available tokens (if any) and invokes the **requestSlots** function (lines 88-91).
In this particular situation, within the **requestSlots** function, the value of "n" will be zero since there are no queued messages, but there are "N" envelopes available. Thus, we have $n = N - N$ (as there are N standby envelopes were requested in advance). As mentioned earlier, a reqslots message will be sent with $n = 0$. Subsequently, the clock ck advances, and the S-record of node B is removed from node A,

```
88 for (j, c) in S_i do
89     for (s, m) in c.tok do
90         send_{i,j}(token, s, c.rck, m) ;
91     requestSlots_i(j) ;
```

this will be explained in detail at section 3.5.2.2. However, due to a network failure, this reqslots message might get lost on its way to node B. In such a scenario, the slots will be retained at node B for future communication with node A.
Considering the aforementioned information, the slots below the frontier "l" will be removed (line 53: $c.\mathsf{slt} := \{m \in c.\mathsf{slt} | m \geq l\}$, keep the slots greater than or equal to l). This action helps streamline the slot management process, ensuring that only relevant slots are retained for potential future communication.

Next, node B evaluates whether the summation of the received sck (s) and the requested number of slots (n) is greater than its local sck, in order to determine whether the message is a duplicate or intended for garbage collection purposes. Node B creates the requested slots (line 56: $c.\mathsf{slt}.\mathsf{union}(\{c.\mathsf{sck}, \ldots, s + n - 1\})$) and sets the lower limit for the next range of slots to be created (line 57: $c.\mathsf{sck} := s + n$). The creation of slots involves adding a range of numbers starting from the current clock sck up to $s + n - 1$.

Once the slots are created, node B sends them (line 58: $\mathsf{send}_{i,j}(\text{slots}, s, c.\mathsf{rck}, n)$) by transmitting one message:

- the sender clock "s" received from node A, thereby enabling node A to verify that the requested slots are based on its sck,
- the incarnation number rck (node B receiver clock), and
- the variable n, denoting the number of created slots.

Message Loss/Duplication In the case of a lost slots message, the protocol includes a mechanism to address this scenario. The sender will retransmit a reqslots message at a later time, triggering the receiver to send the corresponding slots message.
On the other hand, if a slots message is duplicated, it will be discarded to prevent unnecessary duplication. This is achieved through the use of the condition $s = S_i[j].\mathsf{sck}$ (line 64) which ensures that the duplication is avoided. The reason for this is that the sck has already been incremented from the previous slots message. By checking if the slot sequence number (s) matches the sck, the protocol can identify and discard duplicate slots messages effectively.

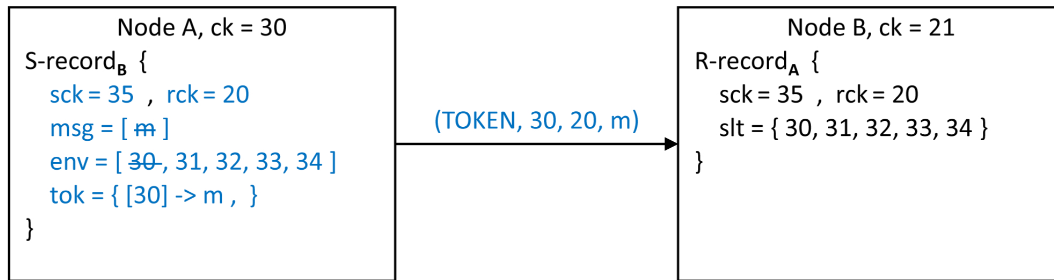


Figure 18: Modifies the clocks, creates envelopes, creates a token from the envelopes, associates the message “m” to it, and sends a token message.

3.4.2.3 Step 3 - Sending Token 18

Step 3. Sending Token (a payload m encapsulated in it): Node A creates a token from an envelope, associates a message m to it, and sends it.

Subsequently, node A receives the slots message (line 61: $\text{receive}_{j,i}(\text{slots}, s, r, n)$) comprising the base sender clock s, the receiver clock r denoting the incarnation number, and the number of created slots n. Node A then verifies whether the received slots message is based on the sender clock (sck) (line 64: $s = S_i[j].\text{sck}$) to avoid duplicates. Next, node A synchronizes the local receiver clock rck by assigning the received r to it (this step is essential so that both nodes has the same incarnation number), creates envelopes (line 67: $c.\text{env}.\text{append}([s, \ldots, s+n-1])$, as for the slots, adding a range of numbers starting from the current clock sck up to $s+n-1$), and advances the sender clock sck by n.

At this time, it checks if there are any messages in the message queue and an available envelope (line 69: $c.\text{env} \neq [\,]$ **and** $c.\text{msg} \neq [\,]$), node A dequeues an envelope (from env), dequeues a message from the message queue (msg), creates a token by adding the envelope number to tok and then encapsulates the message m. After that it can send a token message (line 73: $\text{send}_{i,j}(\text{token}, e, c.\text{rck}, m)$) that comprises the envelope number “e”, the incarnation number rck, and the encapsulated message “m”.

During this stage, node A performs a series of checks and operations. Firstly, it verifies if there are any messages in the message queue and an available envelope (line 69: $c.\text{env} \neq [\,]$ **and** $c.\text{msg} \neq [\,]$). If both conditions are met, node A dequeues an envelope from env and retrieves a message from the message queue, stored in msg. Subsequently, it generates a token by combining the envelope number with tok and encapsulates the message m. Once these steps are completed, node A transmits a token message (line 73: $\text{send}_{i,j}(\text{token}, e, c.\text{rck}, m)$). This token message consists of the envelope number e, the incarnation number rck, and the encapsulated message m.

If the sender exhausts all available envelopes while messages remain in the queue, it requests additional slots. Moreover, if the message queue becomes empty, the requestSlots function is invoked (line 74: $\text{requestSlots}_i(j)$) to request additional slots in advance and ensure that N envelopes are available for

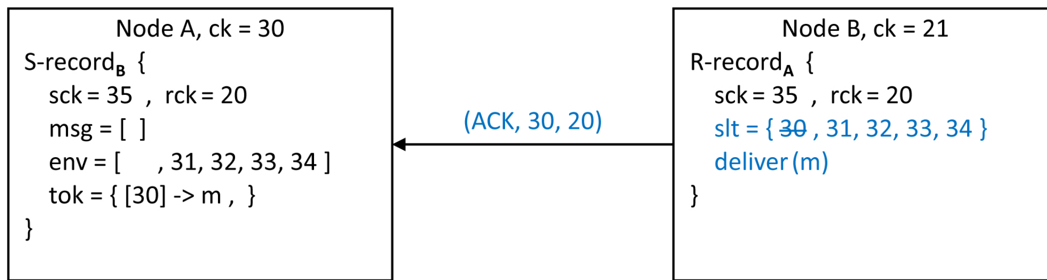


Figure 19: Removes the slot, delivers "m", and sends an ack message.

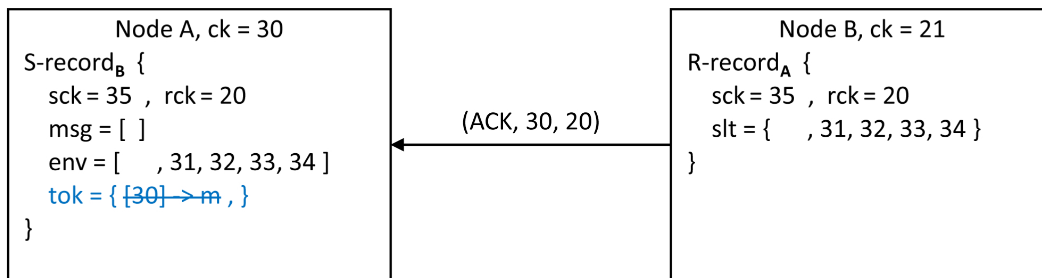


Figure 20: Removes the token.

future EOSends.

Message Loss/Duplication Token loss is addressed through the retransmission of tokens during the "periodically" operation. If a token message is lost, it will be resent in subsequent periodic cycles, ensuring its delivery to the intended recipient.
Token duplication is addressed in a different manner. Since each token corresponds to a single slot, and the slot associated with the duplicate token has already been removed due to the previous token, the duplicate token itself is discarded. However, an ack message is sent to the sender that has no effect since the token is already removed from the previous ack.

3.4.2.4 Step 4 - Processing token and Sending ack 19

Step 4. Receiving a token *message, payload delivery and sending* ack*:* Node B receives a token message from node A, delivers its payload if there is a corresponding slot, and deletes the slot. Node B then sends an ack to A.

Upon receipt of the token message (line 75: receive$_{j,i}$(token, s, r, m)), node B receives three pieces of information: s (the token number), r (the rck of node A), and the payload m. Subsequently, node B verifies that the received token message is not from a previous incarnation (has the same incarnation number as the rck), and that it has a corresponding slot in slt. If these conditions are met, the slot can be safely removed from node B, and the payload m can be successfully delivered to its intended destination layer.

Following this, an ack message is sent to node A (line 81: $\mathsf{send}_{i,j}(\text{ack}, s, r)$) to instruct it to safely remove the token (line 86: $c.\mathsf{tok}.\,\mathsf{remove}(s)$) 20.

Message Loss/Duplication ACK duplication is not significant as the token associated with the duplicated ACK has already been removed. Therefore, the duplicated ACK is discarded without any impact. However, in the event of an ACK message loss, the ACK itself is not retransmitted. Instead, the corresponding token is retransmitted by the sender. Upon receiving the retransmitted token, the receiver generates and sends an ACK message back to the sender.

3.4.3 The "periodically" in the protocol

In addition to the aforementioned operations, there is the retransmission part that is essential in our protocol, represented in the "periodically" (lines 87-93):

```
87 periodically
88     for (j, c) in S_i do
89         for (s, m) in c.tok do
90             send_{i,j}(token, s, c.rck, m)
91         requestSlots_i(j)
92     for (j, c) in R_i do
93         send_{i,j}(slots, c.sck, c.rck, 0)
```

This section in the algorithm indicates that, after a specific period of time, depends on the message type (e.g. reqslots, slots, token, ack), a specific action takes place. In this case, the action involves node A sending all the tokens it has accumulated and calling the requestSlots function, towards all the nodes it has communicated with. Along with the tokens, node A also transmits the slots that have been allocated for future communication.

However, it is important to note that this phase holds greater importance beyond message retransmission alone. It is a crucial component that helps ensure eventual communication and stabilization among the nodes in the system. The "periodically" feature in the Exon protocol plays a vital role in promoting stability, synchronization, and the eventual achievement of node obliviousness, enabling the protocol to function effectively even in the face of network challenges and uncertainties.

In detail, a reqslots message could be lost, after a specific time (the time amount discussed in chapter 4), the requestSlots function will be called, in order to re-send a reqslots message that may contain the same values as the previous reqslots, or other values, and this could happen if new messages queued to be sent.

If the slots message get lost, however, the retransmission of the slots message at line 93 $\mathsf{send}_{i,j}(\text{slots}, c.\mathsf{sck}, c.\mathsf{rck}, 0)$ is not designed to retransmit the lost slots message, but for garbage collection and closing connection

purposes. However, if a slots message get lost, the sender, after a specific timeout, will retransmit the reqslots message, which in turn resend the updated slots (if the values of the reqslots message changed) in a slots message (line 58).

The lost token messages will be sent again, for each node, lines 88-90. Also, the practical timeout for resending the token messages will be discussed in chapter 4.
For the sending slots messages (lines 92-93),

for (j, c) **in** R_i **do**
  $\mathsf{send}_{i,j}(\text{slots}, c.\mathsf{sck}, c.\mathsf{rck}, 0)$

as mentioned previously, is not designed to resend the lost slots message, but for closing connection purposes. This process occurs at regular intervals, but it takes a significant amount of time to initiate, especially when there has been a period of inactivity between nodes. In such cases, the receiving node, which has allocated slots for a particular node, instructs that node to terminate the connection (specifically, the half connections) if it has no messages to send. The initial content of the slots message includes the sck value as the first element and the rck value as the second element. These values serve the purpose of synchronization, indicating to the sender the most up-to-date clock information.
However, the final value in the message is deliberately set to zero. This serves as a signal to the sender node, indicating that this is not the lost slots message. In this case, the sender node will not generate new envelopes. Instead, the requestSlots function will be invoked at line 74. This action subsequently calculates 'n', resulting in a value of zero. If the sender node possesses neither tokens nor messages in its message queue (line 44: $c.\mathsf{tok} = \emptyset$ **and** $c.\mathsf{msg} = []$), it proceeds to remove the S-record from the node. This is done after sending a reqslots message with the intention of closing the half-connection (thus removing the R-record). The reqslots message contains three values: the first being sck, the second being '$n = 0$', and the third being another sck. This third value serves to instruct the receiving node to clear all the slots below the specified 'l' (line 53: $c.\mathsf{slt} := \{m \in c.\mathsf{slt} | m \geq l\}$). It also checks whether the slots are empty, and if so, it removes the R-record (as indicated in lines 59-60):

if $c.\mathsf{slt} = \emptyset$ **then**
  $R_i.\mathsf{remove}(j)$;

.

It is crucial to highlight that the "periodically" mechanism plays a pivotal role in ensuring stability within the protocol. The synchronization of clocks, as embedded in these communication exchanges, serves as a vital step in assessing the feasibility of removing half-connections. Removing half-connections, in turn, play a significant role in maintaining the node's obliviousness.

3.5 Delving into the Intuition behind Key Aspects

3.5.1 Requesting Slots, and Envelopes on Standby

It is essential to highlight the significance of requesting slots from other nodes in advance and maintaining envelopes as standby resources. By pro-actively requesting envelopes, nodes can ensure a continuous supply of tokens, enabling smooth communication and message propagation throughout the system. This approach minimizes the risk of message transmission delays or disruptions caused by insufficient envelope availability.
By keeping a standby pool of envelopes, nodes can rapidly respond to message requests without waiting for the creation or allocation of new envelopes. This proactive approach enhances the efficiency and reliability of the system, as nodes can quickly access and utilize envelopes as needed, eliminating unnecessary waiting periods or potential bottlenecks, and ensuring a smooth and uninterrupted flow of messages between nodes, unless standby envelopes are utilized, therefore the system would be subject to unavoidable waiting periods.

The allocation of standby envelopes and the determination of the number of requested slots can be described by the formula $n = N + |c.\mathsf{msg}| - |c.\mathsf{env}|$. In this equation, n represents the total number of requested slots that equals to, N corresponds to the number of standby envelopes available, plus $|c.\mathsf{msg}|$ denotes the count of messages currently queued, minus $|c.\mathsf{env}|$ signifies the number of available envelopes.

The process of requesting slots for message transmission follows a specific pattern. Initially, when an EOsend command is issued to send a message, a request for slots is initiated. However, it is important to note that this slot request (line 25: $\mathsf{requestSlots}_i(j)$) is made only if the sending node has no S-record for the receiving node. After that, when additional messages are to be sent, the request for slots is not repeated.
Once the initial request for slots is made (line 25: $\mathsf{requestSlots}_i(j)$), the subsequent messages are not immediately sent. Instead, they are enqueued in the message queue, awaiting the arrival of the requested slots. The message queue serves as a temporary storage space for messages until the required slots are received. Once the slots arrive, the corresponding envelopes are created, and tokens are generated. These tokens are then associated with the respective messages.

By adopting this approach, nodes can optimize the efficiency of slot allocation. Rather than requesting slots for each individual message, the system minimizes the frequency of slot requests, reducing unnecessary overhead. Instead, messages are enqueued and sent as soon as the requested slots arrive, allowing for a streamlined and efficient transmission process.

3.5.1.1 Successive Requesting of Slots

The process of requesting slots follows a specific condition. Initially, when there are no available envelopes, slots are requested (line 25: $\mathsf{requestSlots}_i(j)$). However, if envelopes are already accessible, the request for slots will occur when the number of available envelopes is one less than the standby envelopes number N ($|c.\mathsf{env}| = N-1$) (line 33: $\mathsf{requestSlots}_i(j)$). Importantly, this request for slots operates in a passive manner and does not hinder the sending of messages when envelopes are readily available.
By requesting slots only when the number of available envelopes reach a specific threshold, the system optimizes efficiency and minimizes unnecessary slot requests. This approach ensures that the system remains responsive while avoiding excessive overhead.

It is worth noting that the request for slots is triggered when the number of available envelopes reaches N - 1, rather than when it is simply less than N. This design ensures that the EOsend command is not issued for every message sent, optimizing the efficiency of the system. For instance, if the number of available envelopes is 1000 and EOsend is called 500 times, the request for slots will be made only once at that point when $|c.\mathsf{env}|$ reaches 999 (N - 1), and this will maintain the availability of N standby envelopes.

The algorithm has the capability to function without this feature, but its overall significance and benefits lie in its ability to avoid one round-trip time (1 RTT). By pro-actively requesting slots, even when envelopes are present, the system ensures a continuous and uninterrupted flow of messages. This avoids the potential delay caused by the need to request slots separately when they are eventually required.

By incorporating this feature, the algorithm optimizes message transmission efficiency by eliminating the need for additional round trips solely for slot requests. It streamlines the process, reducing potential latency and enhancing the overall performance of the system.

3.5.2 Incrementing Global Clock ck

The global clock, represented as ck, serves a crucial purpose in preventing message duplication by assigning it to two different clocks: rck as the incarnation number in the receiving half-connection, and sck as the starting point for requested slots.
By utilizing the incarnation number rck in the receiving half-connection, nodes can distinguish between different instances or incarnations. This ensures that duplicate messages from previous incarnations are detected and discarded, maintaining message integrity and avoiding redundant processing.
The clock sck acts as a reference point for requesting slots. When nodes need to request slots for message transmission, they use sck as the starting point. This helps in managing the allocation of slots effectively and avoiding conflicts or overlaps.

This distinctive approach of utilizing one global clock per node sets our protocol apart from others. Unlike other protocols that require maintaining extensive logs or complex synchronization mechanisms, our protocol simplifies the process by relying on a single global clock for each node.
By utilizing a single global clock, the need for maintaining exhaustive log entries or complex synchronization mechanisms are eliminated. The incorporation of incarnation numbers and slot request clocks derived from the global clock ensures effective message duplication prevention and resource allocation without the overhead of additional log management.
This approach not only simplifies the implementation and maintenance of our protocol but also enhances its efficiency and performance. The reliance on a single global clock per node reduces complexity, minimizes resource consumption, and ensures a more streamlined and reliable communication process.

3.5.2.1 Receiving a New Connection

When a new connection arrives from another node, the receiving node increments its clock, signifying the start of a new state or incarnation. This incrementation is done by incrementing the global clock, denoted as ck, by one.
The incrementation by one serves a specific purpose in maintaining a consistent and monotonically increasing clock value across the network. Each time a new connection is established, incrementing the clock by one ensures that the new state or incarnation is assigned a unique and higher clock value compared to the previous state.
By incrementing the clock by one, the receiving node effectively differentiates the new connection from the previous ones and ensures that subsequent messages or events associated with the new connection is assigned a higher clock value.

3.5.2.2 Closing Connection

Closing a connection initiated from the sender, as the receiver is the one that makes itself "available", and will typically have slots, that cannot decide to throw away. If at a moment, the sender has no messages are pending (in the message queue) nor inside tokens, then it decides to close the sender side connection for the other node. Hence, the invocation of the requestSlots function at line 91, which occurs periodically, serves the purpose of checking if there are any pending messages or tokens. If no messages or tokens are present, it sends a reqslots message with a value of n=0 (line 45).

By requesting slots with a value of n = 0, the sender node effectively communicates its intention to close the connection to the receiving node. This request signifies that the sender no longer requires any additional slots for message transmission. Additionally, the sender instructs the receiving node to remove all slots below the current slot clock (sck) indicated in the request.
Once the reqslots is sent, the sending node proceeds to increment its clock, signifying the transition to a new state or incarnation. This incrementation is based on the following formula: $ck = \max(ck, sck)$.

The formula ensures that the sending node's clock value is updated to the maximum value between its current clock value (ck) and the sender clock value (sck) specified in the reqslots message. By taking the maximum of these two values, the sending node ensures that its clock is always set to a value that is at least as high as the sender clock. Using the max function instead of simply adding sck to ck allows the sender node to handle scenarios where it may be communicating with multiple nodes simultaneously, rather than just the current receiving node. By taking the maximum value between ck and sck, the sender node ensures that its clock is set to the highest value seen across all incoming slot clocks.
This mechanism guarantees proper differentiation and synchronization between nodes, as subsequent messages or connections from the sender will be associated with a higher clock value.

It is important to note that envelopes, as they do not contain any payload, can be discarded without impact. Only token entries need to be retained until acknowledgment is received.

3.5.3 Closing Connection reqslots Message Loss

In the event of a Closing Connection reqslots message loss, the protocol handles it differently compared to regular message losses. When a regular message is lost, it is resent periodically until it reaches the intended recipient. However, in the case of a reqslots message that is lost during the closing of a connection (as indicated in lines 45-47), there is no retransmission of the lost message.
As a result, the S-record (sender-side half-connection record) associated with that connection is removed at the sender node (line 47). On the other hand, the R-record (receiver-side half-connection record) at the receiving node is still maintained, preserving the slots. However, the receiving node will initiate periodic resending of the slots.

When the sender node receives a slots message from the receiving node, it realizes that it no longer has an S-record for that specific node (line 62). In response, the sender node sends a new reqslots message to the receiving node. This new reqslots message contains the current clock value ck as the slot clock ($s = ck$), a request for zero slots ($n = 0$), and the last clock value ck as the value of l (line 53). By specifying $l = ck$, the sender node signals to the receiving node to remove all slots below this clock value. As a result of this process, the receiving node removes the slots below $l = ck$ (line 53), leading to an empty slot configuration. Subsequently, the sender node's R-record is removed from the receiving node (lines 59-60), finalizing the closure of the connection.

In summary, when a Closing Connection reqslots message is lost, there is no retransmission. The receiving node periodically resends the slots, prompting the sender node to send a new reqslots message with specific parameters. This ensures the removal of slots below a certain clock value, ultimately leading to the removal of the sender node's R-record at the receiving node, closing the connection.

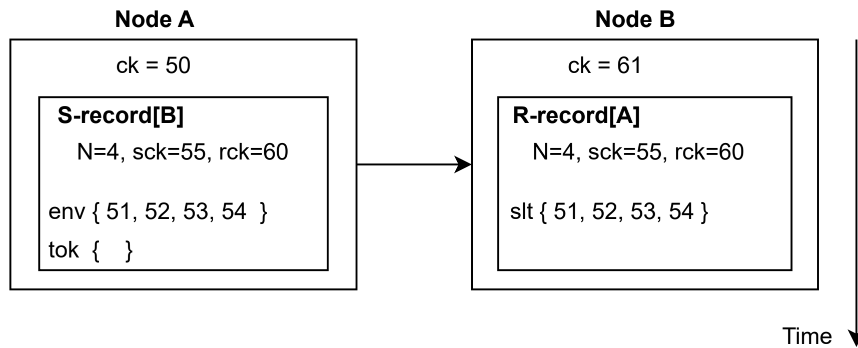


Figure 21: Node A sending messages to node B

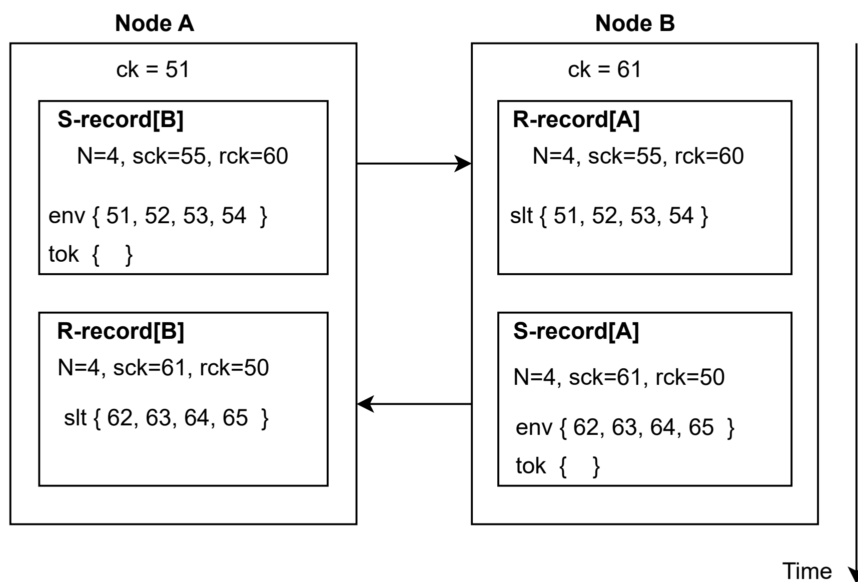


Figure 22: Node A and node B exchanging messages

3.6 Advanced Properties

3.6.1 Soft Half-connections

In the Exon protocol, connections play a crucial role in organizing message communication and achieving performance optimizations. Typically, connections are established between sender and receiver pairs to group messages and facilitate efficient delivery. However, the Exon protocol introduces the concept of "soft half-connections" (s-connection), which provides a flexible approach to handling message grouping. We have what we call soft- half-connections (s-connection), that group messages from the same sender-receiver pair, created on-demand if messages are requested to be sent. This means that if node A wants to send a message to node B, node A only needs to create an S-record for node B without the need for an R-record for it. Similarly, if node B is receiving a message from node A, it creates an R-record for node A without the need for an R-record for it.

As illustrated in Figure 21, when Node A transmits messages to Node B, only an S-record for Node B exists at Node A, while Node B maintains only an R-record for Node A. However, as demonstrated in Figure 22, when Node B initiates message transmission to Node A, it generates an S-record for Node A, and Node A in its turn establishes an R-record for Node B.

The s-connection offers performance benefits by dynamically managing message grouping without the overhead of establishing and maintaining full connections. It allows for efficient organization and processing of messages while avoiding unnecessary connection-related operations. Moreover, it is essential to emphasize that the s-connection can be intelligently discarded when it is no longer needed, a feature crucial for optimizing resource usage.
Furthermore, to optimize resource usage, the s-connection can be discarded if there are no pending unacknowledged messages associated with it, after a certain non-short timeout. This approach ensures that resources are released when they are no longer needed, contributing to the protocol's efficiency.

3.6.2 Obliviousness

In a general context, "obliviousness" refers to a state of being unaware, forgetful, or inattentive to certain details, events, or information. It implies a lack of consciousness or knowledge about specific aspects or occurrences.
In the context of messaging protocols, "obliviousness" refers to a property wherein nodes within the protocol are designed to operate without retaining or relying on specific information about other nodes for an extended period. This means that nodes can remove or discard information associated with other nodes without adversely affecting the correctness or functionality of the protocol.

Exon protocol achieves exactly-once correctness without the need to retain connection-related information indefinitely. Instead, it only maintains a single integer per node as permanent state when no connections are present. The "integer" in this context does not necessarily imply the standard integer data type; it can be any increasing numerical value, such as a long integer or a big integer.
In the following, we delve deeper into the core mechanisms that make Exon protocol a robust and efficient messaging solution. We will discuss key aspects such as "Removing Unused Slots", where we explore how Exon optimizes resource utilization by eliminating unnecessary slots; "Efficient Connection Removal and Clock Incrementation", which outlines the protocol's streamlined approach to connection management and clock synchronization; "Obliviousness Despite Message Loss", where we discuss how Exon maintains node obliviousness while handling message loss scenarios; and "Self-Stabilizing Against Old Duplicates", where we unveil Exon's techniques for ensuring stability even in the presence of older duplicate messages. Each of these aspects contributes to the protocol's remarkable capabilities and reliability.

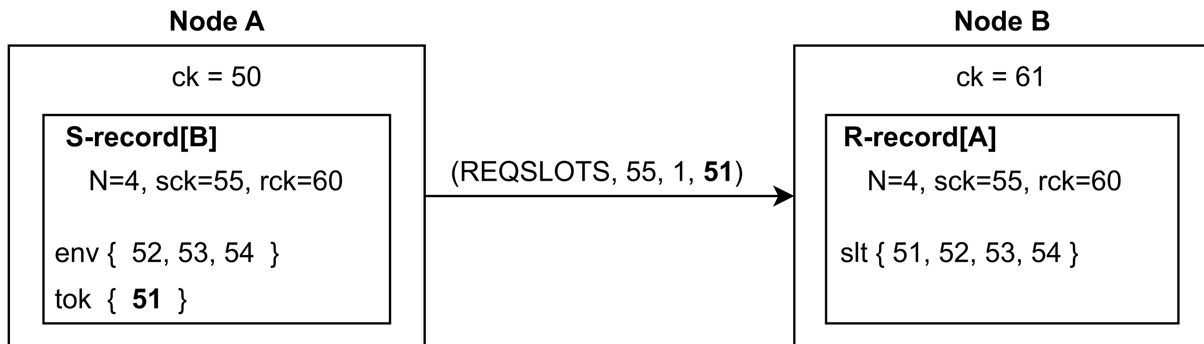


Figure 23: Node A requesting slots from node B, telling it to remove slots below l=51

3.6.2.1 Removing Unused Slots

One aspect of the protocol is that it requests more slots than the actual number of user messages. While it may initially seem wasteful to have unused slots, this design choice is not problematic. The reason lies in how the protocol handles the creation of tokens. Tokens, which are associated with message payloads, are created from envelopes. However, the protocol does not create tokens until a message payload exists. As a result, the envelopes themselves, which are not associated with any message, can be safely removed, if the sender wants to quit.

Each time a sender node requests slots, it conveys to the receiving node the safe frontier of slots, where slots below than that frontier can be removed safely. The value of "l" is determined based on one out of three conditions.

Condition 1 (Line 40: $l = \textbf{ if } c.\mathsf{tok} \neq \emptyset \textbf{ then } \min(\mathsf{dom}(c.\mathsf{tok}))$): If tokens are present for node B, then "l" is assigned the lowest token number, signifying that node B can safely remove slots that has no tokens for them at node A.
As illustrated in figure 23, a specific condition arises when there is a message payload to be transmitted to node B. At this point, there are a total of N envelopes available. The EOsend function is responsible for dequeuing one envelope from this set to create a token, numbered as 51. This action aligns with the condition described in line 32 ($|c.\mathsf{env}| = N - 1$). Consequently, the $\mathsf{requestSlots}_i(j)$ function is invoked (line 33).
As a result, at line 38, where the variable n is calculated as $N + |c.\mathsf{msg}| - |c.\mathsf{env}|$, its value becomes one. Now, since a token with the identifier 51 exists, this sets the variable "l" to 51. This informs node B that it is safe to remove slots that are below 51 where these slots for some reason still exists.

Furthermore, this scenario may happen when receiving slots, the node enters a loop, generating tokens and sending token messages (lines 70-73). In cases where all messages have been processed from the message queue but envelopes still exist, the $\mathsf{requestSlots}_i(j)$ function will be invoked (line 74).

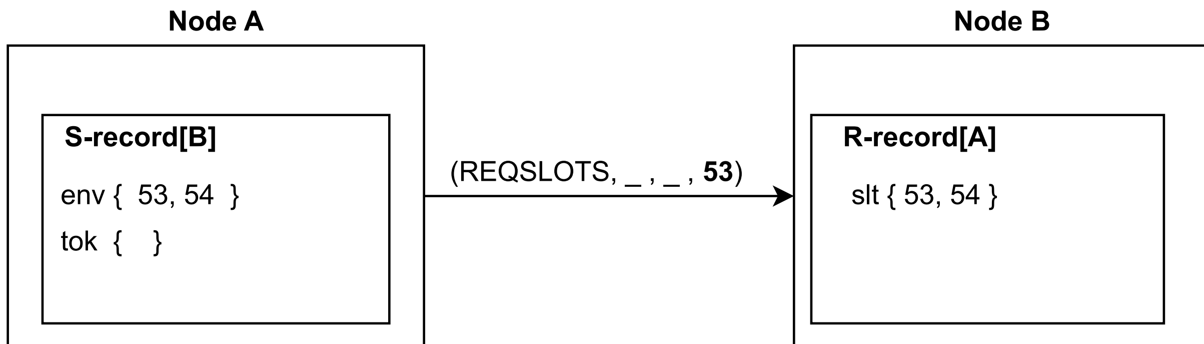


Figure 24: Node A requesting slots from node B, telling it to remove slots below l=53

Condition 2 (Line 41: **else if** c.env $\neq$ [] **then** c.env[0]): If there are no tokens but there are envelopes, in this case "l" is set to the lowest envelope in the sequence. Here, node A informs node B that it can safely remove slots below "l". A visual representation of this process is illustrated in figure 24.

Condition 3 (Line 42: **else** c.sck): is where there are no tokens nor envelopes, therefore "l" will be equal to sck, the receiver then can delete old slots based on this sck value.

3.6.2.2 Efficient Connection Removal and Clock Incrementation

Additionally, to ensure that the removal of a connection does not have any adverse effects on future communication with the same node, and to avoid the need for keeping connections open indefinitely, the protocol allows for the removal of connections. This removal process is achieved by incrementing the global clock, as mentioned previously.

By incrementing the global clock when removing a connection, the protocol effectively transitions to a new state or incarnation. This incrementation ensures that any subsequent messages or connections from the sender to the same node will be associated with a higher clock value. As a result, the receiving node can differentiate between messages from different incarnations and avoid any potential duplication.

This approach allows the protocol to manage connections efficiently without the need to maintain a separate connection state for each node indefinitely. Instead, by incrementing the global clock upon connection removal, the protocol can safely discard connection-related information while still guaranteeing the correctness of message delivery.

3.6.2.3 Obliviousness Despite Message Loss

In the Exon protocol, achieving obliviousness even in the presence of message loss is a key feature. While there are four types of protocol messages - reqslots, slots, token, and ack - it is the loss of ack messages

that can impact the removal of tokens and ultimately make the protocol oblivious.

It is important to note that all four types of messages can potentially get lost in the network. However, the protocol handles the retransmission of reqslots, slots, and token messages through the periodic execution of the appropriate mechanisms.
However, when an ack message is lost, there is no retransmission of the ack itself. Nevertheless, the token associated with the lost ack will be retransmitted, triggering the receiver to send an ack message back to the sender. This ensures that all tokens can be eventually removed if the network remains available. The removal of tokens is a critical step towards achieving obliviousness in the Exon protocol.

By designing the protocol to handle message loss and the retransmission of necessary messages, the Exon protocol ensures that even in the face of network challenges, such as lost ack messages, the state removal process can still be accomplished. This robustness allows the protocol to maintain its obliviousness property and continue operating reliably in dynamic network environments.

3.6.2.4 Self-Stabilizing Against Old Duplicates

After achieving obliviousness in the Exon protocol, there is a possibility of encountering an old duplicate message. This can lead to the creation of state elements, such as slots, that may persist for a certain duration. However, it is important to note that the protocol exhibits a self-stabilizing behavior, gradually returning to a state where no unnecessary or obsolete state remains.
There are four types of duplicates in the Exon protocol: reqslots, slots, token, and ack. Among these, token and ack are easily discarded without any state creation. However, reqslots and slots can lead to the creation of new state elements.

reqslots Duplication In the case of reqslots duplicates, the receiver node will create a new R-record for the sending node, creating slots that are not needed. Since the sender does not have any queued messages to send, these slots will remain unused. However, as the protocol continues to operate periodically, the sending of slots will eventually occur at the receiver. When the sender node receives the slots, it will check that there is no record for that particular node since it had removed it in the past. As a result, a reqslots message with n=0 will be sent to the receiver, instructing it to remove the unnecessary slots and subsequently discard the associated state.

slots Duplication In the case of slots duplicates, the sender node may receive duplicate slots messages that were not requested. Upon receiving such duplicates, the sender node (that receives this duplicate) checks if there is a record for that specific node (line 62). If no record exists, the receiver node sends a reqslots message to the sender node with $n = 0$, indicating the removal of slots

if $j \notin \mathsf{dom}(S_i)$ **then**
    $\mathsf{send}_{i,j}(\text{reqslots}, ck_i, 0, ck_i)$

.

At the receiver node, upon receiving the reqslots message, it checks if there is a record for the node that sent the reqslots message. If no record exists, the receiver node creates an R-record for that node. However, since $n = 0$ and the "slt" is still empty, the record for that node is directly removed.
This mechanism ensures that in the case of slot duplicates, unnecessary state elements are promptly detected and removed. By efficiently handling slot duplicates and dynamically managing the state records, the Exon protocol achieves the obliviousness efficiently.

3.7 Correctness Proofs

In this section, we provide a proof sketch of Exon exactly-once correctness, referring to Algorithm 1. In particular, we prove at-most-once which represents the *safety* property, and at-least-once, representing the *liveness* property. We analyze these properties under potential message failures, i.e., loss and duplication, demonstrating how Exon guarantees exactly-once despite such situations.

3.7.1 At-most-once

Exon is optimized for the at-most-once case by design. It achieves this through booking a slot for each message token to be sent. As long as this association is unique, the message will be delivered at-most-once. Consequently, the following should be proven correct: (1) Slots and tokens are unique, which requires an increasing clock across soft-connection (a.k.a., s-connection or incarnation). This is proven in Lemma 3.7.1. (2) Within an incarnation, the combination slot-envelope is unique. This means a slot is created at most once at the receiver, and the corresponding envelope at the sender is created at most once. These are proven in Lemmas 3.7.2 and 3.7.3. (3) A message payload is associated to at most one token and one envelope (Lemma 3.7.4); and a slot is consumed once by the receiver (Prop. 1).

Lemma 3.7.1. For each node i, each receiver-side s-connection identifier rck is instantiated at most once, whether for the same or different senders.

Proof. When a receiver-side's s-connection record is created (upon a first message receipt), the s-connection's identifier rck gets assigned the node's clock ck_i, that is then incremented (lines 50, 51). Since ck_i is always incremented across s-connections (line 46), then rck can be instantiated at most once. Under duplication, any subsequent (duplicate or new) message will belong to this incarnation (notice that no incarnation request exist). If the incarnation is deleted (e.g., after a long silence period), a new $\mathsf{rck}' > \mathsf{rck}$ is assigned to the new incarnation. □

Lemma 3.7.2. Each slot (j, i, s, r) is created at most once.

Proof. For each node i, from Lemma 3.7.1, slots created at i for different incarnations will have different rck values. For each incarnation with a given r, for some sender j, each slot (j, i, s, r) is created in a range with s starting from the sck in the connection record, which is incremented to become one past the last value in the range created (lines 56, 57). This makes the subsequent creations in the same incarnation to have non-overlapping ranges and no duplicate slots are created under retransmissions. □

Lemma 3.7.3. Each envelope (i, j, s, r) is created at most once.

Proof. For each node i, for each incarnation with a given r, for some receiver j, each envelope (i, j, s, r) is created in a range with s starting from the sck in the connection record, which is incremented to become one past the last value in the range created (lines 67, 68). This makes the subsequent creations in the same incarnation have non-overlapping ranges. When an incarnation terminates (e.g., not being used for long time, since Exon has no notion of closing connections), the node's clock ck_i is made to be at least as large as the sck in the incarnation being discarded (line 46), making subsequent incarnations have larger starting sck value, non-overlapping with previous incarnations to the same receiver. Therefore, no duplicate envelopes are created under retransmissions. □

Remark: different co-existing sending-side half-connection incarnations can have at some point overlapping values of sck, but as they are to different receivers, the uniqueness of each envelope created is maintained.

Lemma 3.7.4. A message payload m to be sent to j is associated into some token (i, j, s, r) at most once.

Proof. A message payload m is associated either immediately (lines 29-33) or later (lines 70-73) after being queued for some time to a uniquely created envelope that is immediately consumed. In addition, a token, associated to a single envelope and a message payload m, is deleted after message delivery (line 86), which makes it impossible to associate any message to that token (including the duplicated message itself). □

Proposition 1. *A sent message payload m to receiver node j is delivered at j at most once.*

Proof. From Lemmas 3.7.3 and 3.7.4, a message is associated to at most one token before it gets deleted. On the other hand, Lemmas 3.7.2 and 3.7.3 prove that at most one slot is created for an envelope (associated to a token). The proposition holds since a message is only delivered by consuming the slot corresponding to the token holding the message (lines 79, 80). A duplicate token message will not have a corresponding slot and will results in discarding it or sending an ACK if retained (line 81). □

Remark: in principle, deleting a token and delivering the message in lines 79 and 80 should be atomic. Since we do not assume node crashes, we exclude this from the algorithm.

3.7.2 At-least-once

To guarantee at-least-once message delivery, we need to prove the following: (1) A sender having a buffered message m will not give up until receiving a corresponding slot (Lemma 3.7.5). (2) A sender having received a slot will not give up retransmitting the corresponding token until the messages is delivered and ACKed (Lemma 3.7.6).

Lemma 3.7.5. A sender node of message payload m will eventually receive a slots message, to send a corresponding token.

Proof. The sender having a message to send will send a reqslots message to receive a SLOT. It however does not know if a reqslots message is lost or delayed. Therefore, it may send the same or a new reqslots message based on different cases: either some slots may have been received, or some EOSends are invoked (lines 87-93). In either case, a different slot range is calculated (line 38). Notice that the protocol will not block even if some tokens have been lost since a receiver node can still create new slots without limits. □

Lemma 3.7.6. A receiver node of message payload m will eventually receive a token message delivering m.

Proof. The sender having a message token to send will send a token message until it receives an ACK. The sender does not know whether a token or ack message is lost. Therefore it keeps retransmitting the token message (lines 89,90) until an ack is received. A duplicate token will have no effect because a slot has been already deleted at the receiver; however, the latter will resend the ack (lines 78,81). A duplicate ack will not have effect on the sender because the first ack deletes the corresponding token at the sender and the duplicated acks are just discarded. □

Therefore, Lemmas 3.7.5 and 3.7.6 ensure at-least-once message delivery under network loss and duplication.

We make two final notes on correctness. The first is that Exon is not blocking if a node is communicating with many nodes since each node has its specific independent record either for sending (S record) or in receiving (R record). The second is that the algorithm does not keep any garbage meta-data throughout the different phases. In particular, sending the ack messages in the end is important not only to stop the senders' retransmissions of token messages, but also to garbage collect these delivered message tokens.

4 Exon-lib: an Exactly-Once oblivious library for lightweight messaging

4.1 Introduction

Our study and experience throughout this work demonstrates the daunting task of designing for EO. This makes it more cumbersome for developers to get the implementation right. To make the developers' life easier, we have introduced Exon-lib, a self contained library that implements Exon efficiently, and provides a well defined API. To ensure correctness and efficiency, we faced many challenges that we solved through using both fundamental as well as state of the art engineering best practices. This manifests mainly in the implementation of modules related to Multi-Threaded Architecture, Flow Control, and Retransmission and Timeout mechanisms.

In explaining the Exon-lib library, I will provide a comprehensive overview of its key components and why we chose Java as the default programming language. Then, I will delve into "Node State", explaining how it manages and represents the state of network nodes within the system. After that, I will discuss the "Multi-Threaded Architecture" employed by Exon-lib, highlighting its approach to concurrent processing and resource management. Following this, I will explore "Flow Control", explaining the mechanisms it employs to regulate data flow and ensure efficient communication. Finally, I will delve into "Retransmission and Timeout", describing how Exon-lib handles message retransmissions and timeout management to enhance the reliability of data transmission in networked environments.

4.2 Overview

The Exon-lib architecture is multi-threaded, with a main algorithm thread and a network reader thread. Communication between threads is facilitated through the use of BlockingQueue data structures.

The Exon-lib is implemented as a Java library, it is available on GitHub [70], and it consists of approximately

1245 lines of code. It consists of a node state with a clock and two maps for sender-side and receiver-side half-connection records, a multi-threaded system with one thread for algorithm code and sending on the network, and another thread for reading from the network. In addition to flow control mechanisms to handle message overload scenarios. In terms of retransmission and timeout strategy, a PriorityQueue is used to schedule events for retransmitting protocol messages, such as reqslots, slots, and token, where each event has a specific timeout, and the PriorityQueue schedules events to be triggered in the future.

An important point to mention is that the current implementation of the middleware uses UDP for the entire message transport and leaves message fragmentation and merging as potential areas for future work. Also, we use IP addresses as node IDs, and all nodes use the same fixed port for receiving UDP datagrams.

4.3 Choice of Java for Implementation

Java was selected as the primary programming language for implementing this project for several compelling reasons.
Advantages:

- Widely Used: Java is one of the most widely adopted programming languages in the software industry, top 5 based on TIOBE Index [71]. Its extensive usage signifies a robust and reliable choice for developing a library of this nature.

- Modern Abstractions: Java offers modern and powerful abstractions that expedite the implementation process. Features such as object-oriented programming, well-defined libraries, and comprehensive APIs significantly contribute to faster development.

- Developer Familiarity: A substantial portion of software developers is proficient in Java. This widespread familiarity serves as a valuable asset, facilitating collaboration and potentially enabling future reimplementation efforts in different languages using Java as a template.

- Optimized JVM: While the Java Virtual Machine (JVM) has historically been criticized for its high memory consumption and slower startup times, modern JVM implementations have made significant strides in optimization [72].

One potential drawback of Java is its reputation for high resource consumption, particularly in terms of memory usage. However, it is essential to acknowledge that with advancements in JVM technologies and efficient memory management strategies, this concern is mitigated in modern JVM implementations.

4.4 Understanding the API

The Application Programming Interface (API) is a vital component of the Exon-lib library, designed with the aim of simplifying communication between nodes and enhancing the developer's experience. This section explains the key aspects of the API, emphasizing its purpose, usage, and the developer-centric approach adopted.

4.4.1 Why Use an API?

The Exon-lib API serves as the interface through which developers interact with the library's functionalities. It abstracts the underlying complexities of network communication and message passing, enabling developers to focus on the core logic of their applications. By providing a standardized set of methods and procedures, the API streamlines the development process, promotes code reusability, and ensures consistency across applications.

4.4.2 API Overview

The Exon-lib API consists of two main components: the receiving side API and the sending side API, each tailored to specific use cases.

4.4.2.1 Receiving side API:

EOMiddleware Initialization: Developers can initiate the EOMiddleware by calling EOMiddleware.start(PORT). This step establishes the receiving side communication channel and sets the port for incoming connections.

Message Reception and Forwarding: The receiving side listens for incoming messages using eom.receive().

4.4.2.2 Sending side API

EOMiddleware Initialization: Similar to the receiving side, developers can initialize the EOMiddleware using EOMiddleware.start(PORT). This step ensures consistency between the sending side and the receiving side configurations.

Message Sending: Developers can use eom.send(dstNode, msg) to transmit messages to the specified destination node. This functionality streamlines the process of sending messages across the network.

4.4.3 API Example

To illustrate the simplicity and effectiveness of the Exon-lib API, consider the following code snippets demonstrating its usage in a basic Echo client-server interaction.

In this server-side example (Server.java), the following steps are highlighted:
EOMiddleware Initialization: The server initializes the EOMiddleware using EOMiddleware.start(PORT), specifying the communication port.
Message Reception and Forwarding: Within an infinite loop, the server continuously receives incoming messages with eom.receive(). When a message is received, it can be forwarded to the specified destination node using eom.send(dstNode, request.msg).

Listing 4.1: Server.java

```java
package haslab.eo;

import haslab.eo.msgs.ClientMsg;

public class Server {
  private static final int PORT = 3456;

  public static void main(String[] args) throws Exception {
    String dstHost = "PC-A";
    int dstPort = 1234;

    EOMiddleware eom = EOMiddleware.start(PORT);
    NodeId dstNode = new NodeId(dstHost, dstPort);

    while (true) {
      ClientMsg request = eom.receive(); //receiving request
      eom.send(dstNode, request.msg); //sending response
    }
  }
}
```

In the client-side example (Client.java), the following actions are demonstrated:
EOMiddleware Initialization: The client initializes its EOMiddleware instance similarly to the server with EOMiddleware.start(PORT) to ensure consistent configurations.
Message Sending: Inside a loop, the client sends messages to the designated destination node using eom.send(dstNode, msg). This straightforward method streamlines the process of transmitting messages across the network.

Message Reception and Handling: The client can efficiently receive responses from the server through eom.receive(). This synchronous communication pattern simplifies the handling of real-time responses.
These examples showcase how the Exon-lib API enables developers to create client-server interactions with minimal code complexity.

Listing 4.2: Client.java

```
package haslab.eo;

import haslab.eo.msgs.ClientMsg;

public class Client {
  private static final int PORT = 1234;

  public static void main(String[] args) throws Exception {
    String dstHost = "PC-A";
    int dstPort = 3456;

    String m = "Hello Exon!";

    EOMiddleware eom = EOMiddleware.start(PORT);
    NodeId dstNode = new NodeId(dstHost, dstPort);

    for (int i = 0; i < 10; i++) {
      eom.send(dstNode, m.getBytes()); //sending request
      ClientMsg reply = eom.receive(); //receiving response
    }
  }
}
```

4.4.4 Simplicity from a Developer's Perspective

The Exon-lib API prioritizes simplicity and ease of use. By offering straightforward methods for initializing middleware, sending, and receiving messages, developers can focus on the application's logic without being burdened by low-level networking complexities. This developer-centric approach aims to enhance productivity and accelerate the development cycle.
The Exon-lib API plays a pivotal role in abstracting network communication complexities, offering a user-friendly interface for developers to build robust and efficient applications.

4.5 Node State

Each node has an Exon state. The state is a node-wide clock ck keeping a monotonically increasing integer (Long), and a pair of maps:

- ConcurrentHashMap (sr) keeping sender-side half-connection records of type SendRecord
- HashMap (rr) keeping receiver-side half-connection records of type ReceiveRecord.

4.5.1 Sender and Receiver Connections Maps

We used a ConcurrentHashMap (sr) for the sender-side connection records, that maps node (destination) ids to sender-side connection records "SendRecord", and a regular HashMap (rr) for the receiver-side connection records, that maps node (source) ids to receiver-side connection records "ReceiveRecord".

During the implementation of the "send()" procedure, it became necessary to access the map in order to check the semaphore (the semaphore discussed later in the flow-control section). Simultaneously, other thread (algoThread) is also accessing the same map. To address this concurrent access requirement, I decided to utilize a ConcurrentHashMap. It is crucial to understand that the sole purpose of using this ConcurrentHashMap is to ensure efficient access for the sender specifically to the corresponding Semaphore. Importantly, there is no involvement of shared protocol state in this process. The map functions solely as a mechanism for the sender to retrieve the relevant semaphore, allowing for effective control over concurrent access to specific resources or operations.

One potential improvement could involve utilizing a separate ConcurrentHashMap solely dedicated to storing the Semaphores. By doing so, the sender records could be stored in a regular map, uncontended by concurrent access concerns. This separation of responsibilities between storing Semaphores and sender records could potentially lead to enhanced efficiency.

ConcurrentHashMap in Java provides thread-safety while performing concurrent operations on a map. It is a highly optimized implementation of the Map interface, which allows multiple threads to access and modify the map concurrently, without causing any data inconsistencies or concurrency issues.

4.5.2 Half-connection Records

The half-connection records (S-record and R-record from the algorithm) are: SendRecord and ReceiveRecord, which are implemented as data structures encapsulating various fields such as sender and receiver clocks, message payloads, and Interval and Bitmap data structures.

- The SendRecord is implemented as a SendRecord data structure, which defines a class encapsulating the sender clock (sck: long), the receiver clock (rck: long), the env and the tok fields.
  - The env (envelopes) field is implemented as an Interval data structure, which contains a pair of integers representing an interval [from, to), including the starting point but excluding the ending point. This is possible, without the need for storing an explicit list, because operations on envelopes consist in either appending a contiguous range next to the current largest envelope, or dequeuing the lowest envelope.
  - The tok (tokens) field is implemented as a TreeMap, with sck as the key and a TokenRecord as the value, where the TokenRecord contains the message payload and other associated information. To facilitate the process of requesting slots in the algorithm, the sender needs to inform the receiver about the base or first token that is still at the sender side. This information is crucial for the receiver to remove the oldest slots below this token. In order to achieve this functionality effectively, A TreeMap data structure is employed. The choice of TreeMap was driven by its ability to provide a method called firstKey(), which serves the same purpose as the base or first token.
- The ReceiveRecord is implemented as a ReceiveRecord data structure, which defines a class encapsulating the sender clock (sck: long) and the receiver clock (rck: long). In addition to these fields, the ReceiveRecord also includes a slots set (slt) implemented as a SlidingBitmap data structure.
  - The slots set, denoted as slt, is implemented as a SlidingBitmap data structure. The decision to use a SlidingBitmap is driven by the requirement to remove slots as tokens arrive, even if they are out of order. Therefore, the Interval as in the envelopes is not enough. A SlidingBitmap is a resizable bitmap suitable for representing a set of elements in a sliding window. Elements are added as ranges of new elements contiguous to the previous range. Elements can be removed or tested for presence, but can only be added within the current window. The representation grows as needed and discards memory corresponding to elements before the smallest element still present (start of sliding window).

4.6 Multi-Threaded Architecture

Exon-lib is a multi-threaded middleware, with one middleware thread running all algorithm code and sending on the network, and a second thread for reading from the network. Clients doing sends or receives are other independent threads. Communication between threads is facilitated primarily through the use of BlockingQueue data structures.

The system's global state, which is visible to all threads, includes two primary data structures:

- a BlockingQueue algoQueue represents the main algorithmic queue that stores all protocol messages received from the network or from the client EOsend requests,

- a BlockingQueue deliveryQueue is responsible for delivering EO messages to the local node; the local node can use any client thread to deliver messages.

BlockingQueue is a Java class in the concurrent package that facilitates efficient communication and synchronization between sending and receiving threads in a multi-threaded environment. Acting as a bounded container, it holds a fixed number of elements and allows threads to add or retrieve elements. When the queue is full or empty, threads are automatically blocked, ensuring synchronized actions and avoiding race conditions or resource contention. This mechanism is commonly utilized in multi-threaded applications to enable safe and synchronized data exchange between producers and consumers.

4.6.1 Threads

As illustrated in figure 25, the system comprises two distinct threads, namely the main algorithm thread (AlgoThread) and the network reader thread (ReaderThread), which are responsible for executing different tasks.
The AlgoThread is responsible for running the bulk of the algorithm. Meanwhile, the ReaderThread is specifically dedicated to reading and deserializing protocol messages from the network. As for event related information, whether it related to the reception of protocol messages from the network or a client initiating a send operation, such information arrives to the AlgoThread through the algoQueue. This queue serves as the "channel" through which the AlgoThread retrieves data, wherein both client threads and the ReaderThread participate the enqueuing of this data.
Once a protocol message or a client EOsend request is obtained, the thread invokes the respective handler, which may trigger the sending of UDP datagrams, the enqueueing of EO messages to the DeliveryQueue (DQ), or the discarding of the protocol message due to overloading.
The remaining algorithm state is accessed exclusively by AlgoThread and does not require concurrency control. This includes the node clock (ck) and the receiver-side connection records (rr).

On the other hand, the network reader thread (ReaderThread) continuously reads UDP datagrams from other nodes using a single UDP socket, which is created but not connected at startup. The thread then enqueues the corresponding protocol message to the algoQueue, which can block in case the rate of incoming messages exceeds the rate at which AlgoThread dequeues them. The use of blocking queue as the algoQueue is essential to prevent unbounded memory growth.

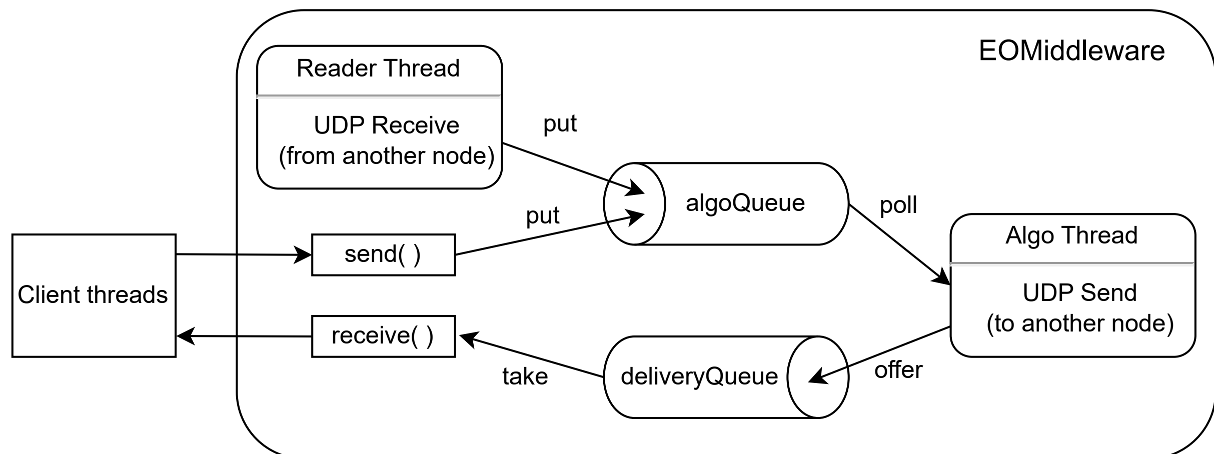


Figure 25: Exon-lib Architecture

4.6.2 Message Types

The message types used in the middleware are represented in the diagram using a class hierarchy. At the top level, there is a common superclass called Msg. This superclass serves as a generalized representation of messages within the middleware. It has four subclasses: ClientMsg, AQMsg, DQMsg, and NetMsg.

ClientMsg represents user messages, being created when a client thread invokes send. A ClientMsg contains the destination node and message payload. The middleware then encapsulates the ClientMsg within an AQMsg and queues it in the AlgoQueue for later processing.

The AQMsg is used in the middleware to encapsulate the ClientMsg before queuing it in the AlgoQueue for later processing. This encapsulation serves a specific purpose within the middleware's architecture. The AlgoQueue is where the information about all events arrive at the AlgoThread, whether it comes from client threads, or whether it comes from the network (after being deserialized); and that AQMsg is the type of the AlgoQueue elements, encapsulating either ClientMsg or the different kinds of NetMsg.
The AQMsg associates additional metadata or information with the message, which can be useful for processing and scheduling purposes. For example, the AQMsg may contain information about the the timestamp, or any other relevant data that can influence the order and handling of messages within the AlgoQueue.

The NetMsg class is responsible for sending messages over the network. It has four subclasses: ReqSlotsMsg, SlotsMsg, TokenMsg, and AcksMsg. These subclasses correspond to specific types of protocol messages that need to be transmitted. For bandwidth optimization, multiple Acks are sent together instead of individually.

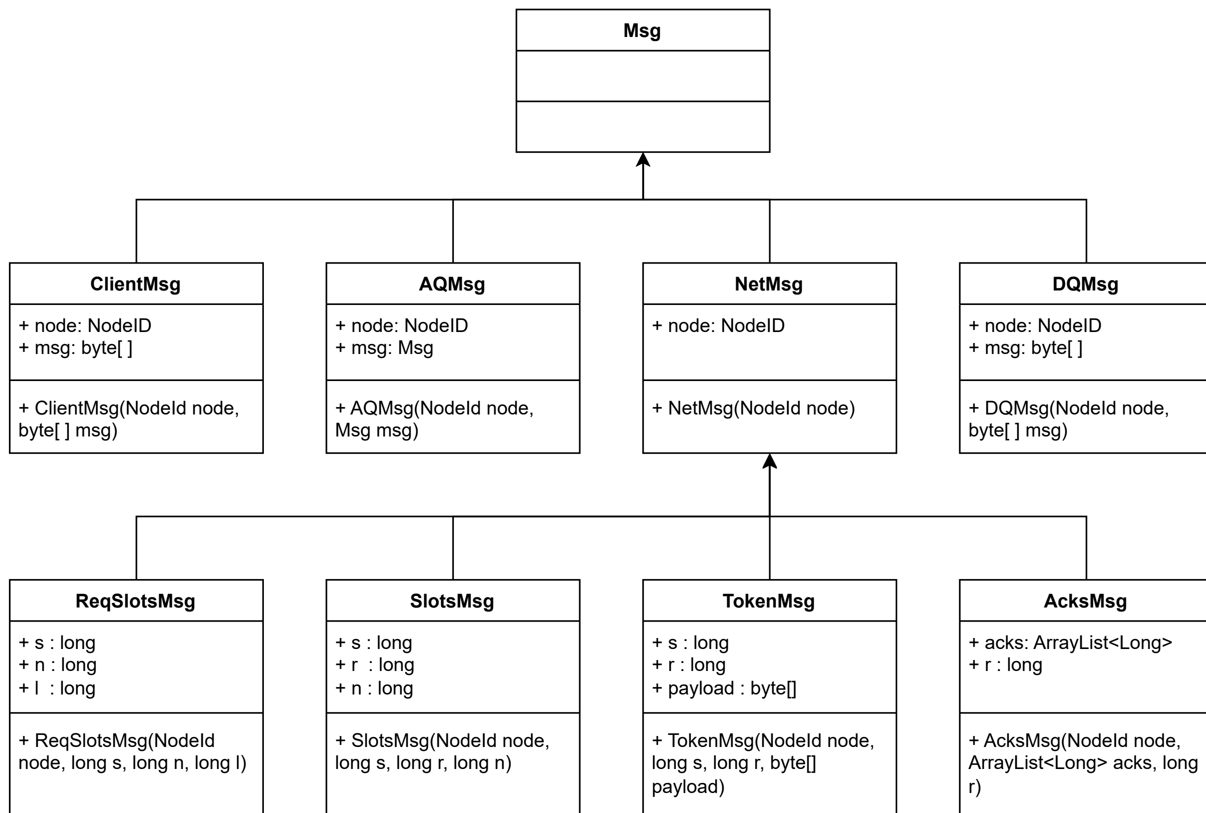


Figure 26: Message Types

On the other hand, the DQMsg class is used to store messages in the DeliveryQueue for eventual delivery to the client application.

Overall, the UML diagram in figure 26 illustrates the relationships between the message types and how they are used within the middleware. It provides a visual representation of the class hierarchy and the inheritance relationships.

4.7 Flow Control

To ensure smooth operation of a message queue, it's important to handle situations where the rate of sending messages exceeds the system's capacity to process them. Specifically, we look at two scenarios: when the sender is sending messages too quickly, and when the receiver is processing messages too slowly. In such cases, appropriate measures must be taken to prevent unbounded memory growth and ensure that the sender eventually blocks.

At the Sender Side, when the rate of EOsend by local client threads exceeds the capacity of the system to handle those requests, the send must be blocked. The issue is that blocking the send cannot be done through the message queue, as the AlgoThread needs to continuously dequeue messages in

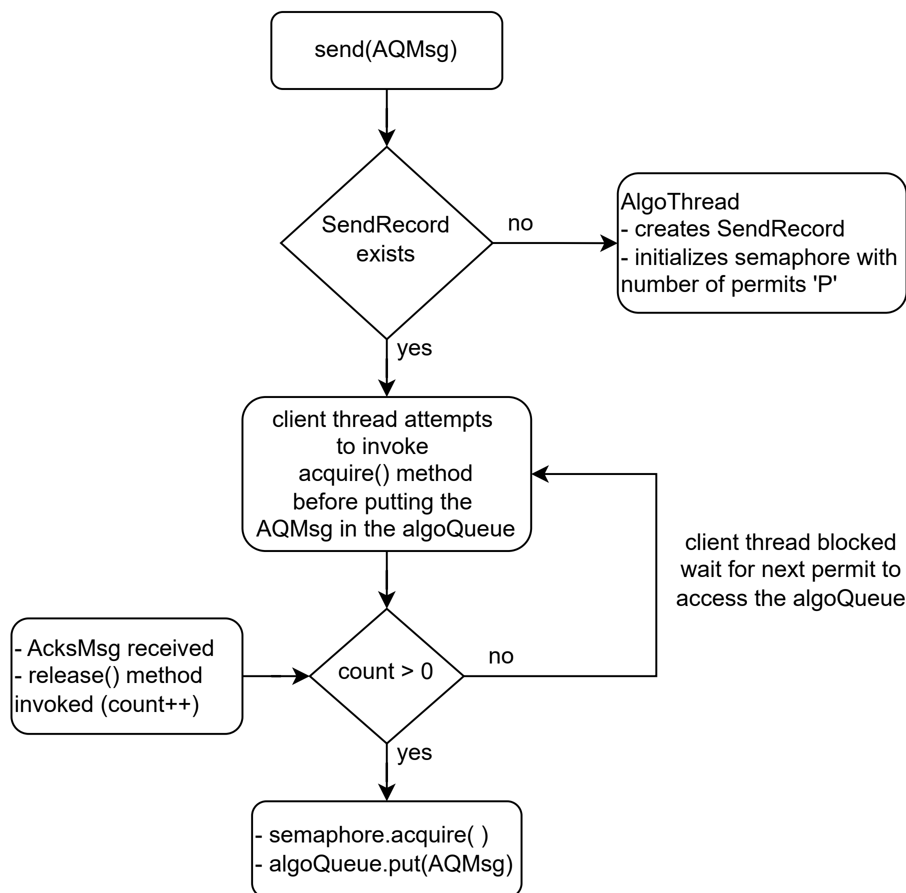


Figure 27: Flow-control, at the sender

order to receive protocol messages.

A solution to this problem is to use semaphores. Specifically, the SendRecord, will have an additional semaphore field. When a SendRecord is created, which can only happen during an EOsend, the semaphore is initialized to a parameter "P", which is a configurable parameter that represents the number of messages that can be sent to a given node without having yet been acknowledged.

A send operation is split into two parts: the send function which runs in a client thread and the handler that executes within AlgoThread. First, when a client thread executes a send(), it looks up the sr map, decrements the semaphore (acquire()), and enqueues the send object (AQMsg) into the algoQueue.

As we can see in figure 27, if a client thread doing a send sees no SendRecord for that destination, it means that there is no pending send, and no need to block. In this case, the client thread enqueues the message in the algoQueue. The AlgoThread then checks if there is no SendRecord for the destination node, it creates a SendRecord and instantiations a semaphore with initial value being one less than the pending sends constant P and enqueues the send object.

The semaphore is incremented (release()) when a token is removed as result of an ack message. This way, the message to be sent is correctly considered "pending", from the moment it is sent, whether it is

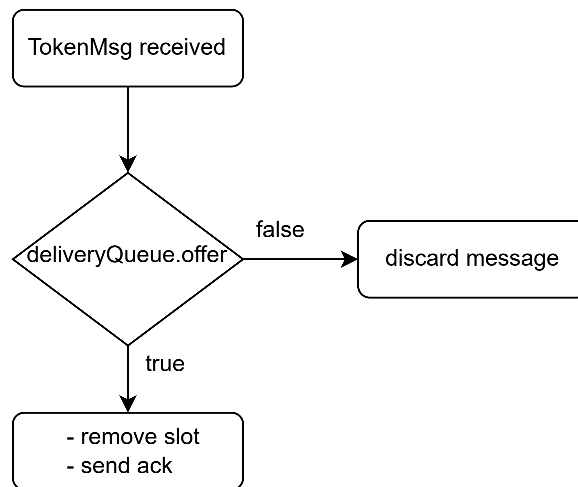


Figure 28: Flow-control, at the receiver

in algoQueue, in a msg field in a SendRecord, or in a tok field, until the token is removed.

At the receiver Side, if the receiver is slow to process messages, we need to avoid unbounded memory growth and propagate the knowledge to the sender, making it eventually block. This can be achieved by ignoring token messages that arrive, and not ACKing them. As shown in figure 28, this is done using the "offer" method, which attempts to insert the specified element into the queue and immediately returns false if there is no space available. If the "offer" method returns false, the slot will remain and consequently the ack message will not be sent to the sender.
This will make the sender keep the token, and eventually block when too many tokens exist. A retransmission will be done after some time, hoping that by that time the queue at the receiver has space.

4.8 Retransmission and Timeout

In this section, we will discuss the strategy used in the algorithm to handle retransmissions and timeouts. Specifically, we will focus on the "Periodically" procedure (in the algorithm) responsible for retransmitting protocol messages such as reqslots, slots, and token. The goal is to avoid bursts of resends and spread timeouts evenly to ensure efficient message delivery. Thus, in order to ensure efficient retransmissions and timeouts, we used the PriorityQueue data structure.

PriorityQueue is a class in Java that represents a priority queue, which is a special kind of queue where each element is assigned a priority, and elements with higher priorities are dequeued first.

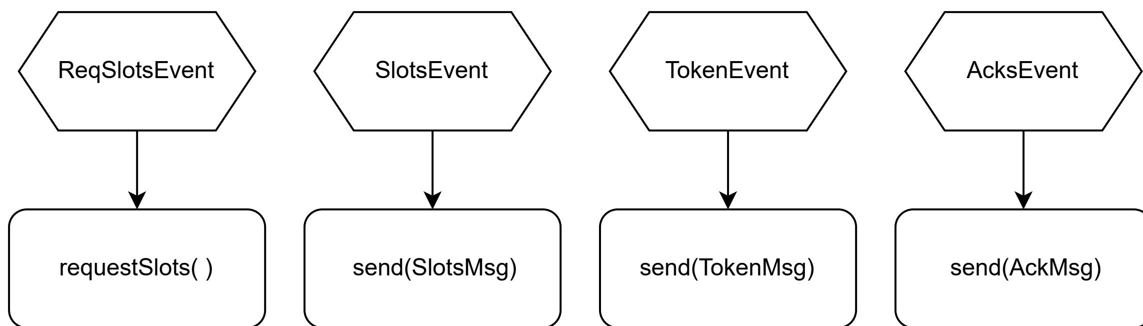


Figure 29: Algorithm events

4.8.1 Events

In the AlgoThread, the java PriorityQueue is utilized, with the comparator based on time. The PriorityQueue pq contains "events" (objects) with a time field used for comparison, which allows scheduling of "events" to be triggered in the future after a certain amount of time has passed.

Individual events are added to the pq for each type of operation that occurs in the "periodically" within the algorithm, such as requesting slots, sending slots, sending tokens, and sending acks (AcksMsg). These events use individual timeouts specific to each type of event. To achieve this, we have four subclasses of the Event class: ReqSlotsEvent, SlotsEvent, TokenEvent, and AcksEvent.
To provide a clearer visual representation of the retransmission's execution, a helpful flowchart (Figure 29) is available.

Within the AlgoThread context, a crucial loop fulfilled its purpose by retrieving messages from the algoQueue using the poll(timeout, unit) method. Following the execution of Exon protocol handlers, another loop examines the priority queue for events whose scheduled time has arrived. It processes all the events with elapsed time, ensuring that no message is left behind.

ReqSlotsEvent: The ReqSlotsEvent is responsible for handling the sending of reqslots messages. When a reqslots message is sent in the requestSlots procedure, a new instance of ReqSlotsEvent is created and added to the pq with a scheduled time equal to the current time plus the reqslots timeout. We chose the reqslots timeout to be two times the RTT, which is a conservative estimate that provides a balance between avoiding unnecessary retransmissions and minimizing the time spent waiting for reply. The corresponding SendRecord object for the connection maintains an additional field to store the time of the last reqslots message sent. The last reqslots send time will be checked if it is changed, and invokes requestSlots if it is still the same, or does nothing if it changed (meaning that another reqslots was sent meanwhile, scheduling another ReqSlotsEvent for the future).

SlotsEvent: The main purpose of the SlotsEvent is when a "closing" reqslots was lost, and later the receiver wants to remove the connection, which was already closed at the sender side. So, to avoid scheduling one event per slots message sent, an initial SlotsEvent added when the connection is opened, and each time a slots sent, the lastSlotsSendTime at the ReceiveRecord is updated with the current time without adding event to pq. When the event is handled, if the connection is already closed, no action is taken. Otherwise, check if send time changed, if it did not change, send a slots message and schedule another SlotsEvent for the future (add to pq) (regardless of change). The timeout for SlotsEvent is much larger than any of the other protocol messages as it is only about garbage collection. We chose it as an absolute number 50000ms.

TokenEvent: The TokenEvent is responsible for handling the sending of token messages. When a token is sent, a TokenEvent is added to the pq with some TokenEvent timeout. When TokenEvent is dequeued, if token does not exist (connection to receiver does not exit, or different connection (rck) or sck missing from tok map) do nothing; otherwise, token is retransmitted and another TokenEvent will be added to the pq. We chose the token timeout as the reqslots timeout, two times the RTT.

AcksEvent: The AcksEvent is responsible for handling the sending of acks messages. When an ack is sent, an AcksEvent is added to the pq with some AcksEvent timeout. It is important to note that this event differs from the rest in that, rather than sending each ack message separately, which could result in increased traffic, the choice is made to bundle multiple acks together, thereby reducing the overall transmission load, for this we name it acks message instead of ack message, which encompass one or many acks collectively.

4.9 Conclusion

The journey of developing the Exon-lib library has been marked by significant learning experiences, iterative design processes, and practical considerations. This concluding section aims to provide insights into the development journey, the challenges encountered, and opportunities for future enhancements.

4.9.1 Architectural Complexity

The initial phase of development presented a substantial challenge. Collaborative efforts with my supervisors were instrumental in designing a robust architecture, particularly the complexities of the multi-threaded design, effective use of semaphores, and the overall middleware structure. Overcoming these hurdles demanded careful planning and collaboration, setting the foundation for the library's functionality.

4.9.2 Learning and Growth

Throughout the development process, a strong emphasis was placed on continuous learning. Gaining insights into distributed systems, networking, and concurrent programming has been a rewarding experience, enhancing both technical knowledge and problem-solving skills.

4.9.3 Language Choice

Pragmatic Selection: Java was chosen as the initial programming language due to its prevalence in the field and my familiarity with it. The decision was driven by the ability to envision efficient implementation strategies and leverage Java's rich ecosystem of libraries and tools.

Library Utilization: The library's development journey benefited from the utilization of existing libraries, such as the blocking array and priority queue.

4.9.4 Remaining Challenges

A notable challenge that remains is the "dynamic" calculation of Round-Trip Time (RTT) and bandwidth during runtime. Implementing mechanisms to adaptively compute these metrics as the program runs would enhance the library's adaptability and real-time responsiveness.

In hindsight, the development of Exon-lib has been a fulfilling experience marked by growth and innovation. Addressing the challenge of dynamically calculating RTT and bandwidth stands as a key area for future improvement. As the project evolves, further refinements and optimizations will be explored, enhancing the library's utility and robustness in distributed systems.

5

Exon Evaluation

In this chapter, we evaluate the performance of Exon. Specifically, we focus on two aspects: (1) the tolerance of Exon to packet loss, and (2) the overhead of Exon under normal communication conditions, both in one-way messaging and remote procedure call (RPC) messaging scenarios. We describe the experimental setup and evaluation methodology, and present the results of our experiments.

5.1 Experimental Setup

To evaluate the performance of our protocol we prepared experiments that reflect a real-world environment. For this purpose, we used Emulab [73], an online network testbed that has a wide range of environments in which researchers can develop, debug, and evaluate their systems. We used in the experiments two machines, each with a 2.4 GHz 64-bit 8-Core Xeon E5-2630v3 processors, 64 GB DDR4 RAM and 20 MB cache, and running Linux 16.04 64-bit Ubuntu OS, We configure the network as two hosts connected to a router in between. The router is configured to induce delays and message loss, mimicking a real network.

5.2 Evaluation Methodology

We examine two common messaging patterns: a no-wait unidirectional pattern *one-way* without replies, and *Remote Procedure Call (RPC)*, a request/reply pattern where subsequent request is sent only after the ACK of the preceding request is received. We do not use any application processing time at the receiver in this case.

The evaluation focuses on throughput and latency, being the most relevant metrics in messaging protocols. We send a sufficient number of messages (between 10K and 1M) per burst to stabilize the network (in addition to warming up). We calculate the throughput, as messages per second, by dividing this number of messages by the measured time difference between the first and the last message delivered. As for the latency, we measure it by recoding the time elapsed between sending a message request and the delivery of the reply, and then computing the mean for all messages. We run as many runs as needed

until we get stable average results. In general, since we have dedicated machines without interference and without application overheads, we did not experience high variance across runs. Finally, we do not measure the latency in the one-way message because it is less important in this pattern and it is hard to measure in the lack of perfect synchronized clocks.

We compared Exon with TCP and UDT. We chose TCP as the most widely used reliable and efficient transport protocol—discarding the inter-connection EO issues; and UDP-based Data Transfer (UDT) [45] as a reliable messaging protocol on top of UDP. The UDT protocol version used for this testbed is the latest version (v4) found at the time this work was done. We had performed minor changes on the sender and receiver side in order to provide an easier way of sending number of packets with specified size.
The main idea of comparing with UDT (or any other reliable UDP) is just to show that any work/diligence to make UDP more reliable (TCP like) results in performance worser than TCP.

To compare the three protocols with large scale or highly concurrent systems, where each node has many objects/ processes/actors of unpredictable lifetimes, we perform RPC tests where a given number of actors perform independent (concurrent) RPCs to a server. Such systems that use TCP (e.g., distributed Erlang) normally use multiplexed TCP connections shared by many local actors. We run tests for different numbers of actors, until we saturate the bandwidth. With respect to message size, we opted for 1 KB messages, to have relatively “full” datagrams, allowing good use of bandwidth, without risking UDP fragmentation.

We run the experiments under two environments: *fault-free environment*, to test the performance of the protocols under normal circumstances, under different network latencies and bandwidths; *message loss environment*, to evaluate how well the protocols tolerate network faults. Thus, we tested the protocols under three variables: bandwidth, network latency, and message loss rate. We tried to set the parameter's values to common network cases. For bandwidth tolerance, protocols were tested under a fixed latency (RTT= 10ms) over different bandwidths: 1, 10, and 100 Mbps. For network latency tolerance, bandwidth was set to 10 Mbps, and RTT taking the values 6, 10, and 100 ms. The reason for choosing 6 ms and not 1ms is because it is the lowest achievable value on Emulab. For message loss tolerance, bandwidth and latency were set to 100 Mbps and RTT=10 ms respectively, while using message loss rates of 0%, 1%, and 5%. In each of these experiments, we tested the protocols using the two patterns described: one-way messaging and RPC. In the one-way messaging pattern, we measured the throughput (msgs/s) while in the RPC messaging pattern, we measured the throughput (requests/s) and response latency (ms).

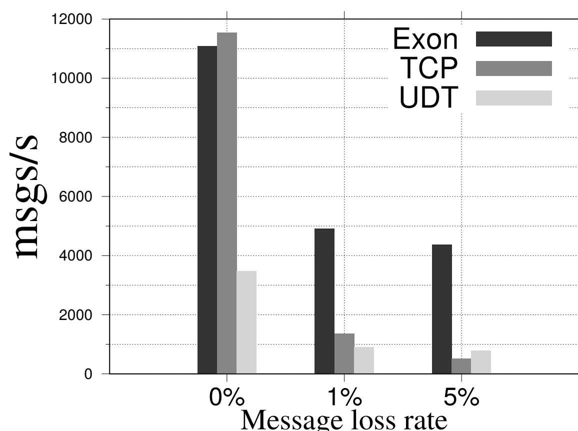


Figure 30: One-way throughput under packet loss (Setting: RTT=10ms, Bandwidth=100 Mbps)

5.3 Results

5.3.1 Tolerance to Packet Loss

In this experiment, we aim to compare the throughput and latency of Exon, TCP, and UDT under packet loss. In all cases, no messages were lost or delivered in duplicate, despite packet loss and retransmissions. We configured Emulab to induce packet loss, dropping packets at 1% and 5% rate. To assess packet loss overhead, we also provide a baseline experiment with 0% loss. We used for network parameters RTT=10 ms, and bandwidth of 100 Mbps.

5.3.1.1 One-way messaging

Figure 30 shows the throughput results of the protocols under 0%, 1%, and 5% message loss. In these scenarios the sender loops sending one million 1KB messages, as fast as possible (throttled only by flow control).

The throughput of Exon and TCP are very close in the 0% loss case, delivering more than 11K messages/sec (i.e., around 88Mbps), which is close to the maximum bandwidth capacity. This means that even though Exon is based on a four-way per-message exchange, it can pro-actively requests a batch of N slots in advance (where N is a parameter based on the BANDWIDTH*DELAY product), by the occasional REQ-SLOTS message, which causes negligible overhead over TCP. However, the UDT throughput is much worser than Exon and TCP. It is believed that being equipped with DAIMD has resulted in degrading the throughput [74].

Nevertheless, with a 1% and 5% packet drop rate, Figure 30 shows a significant throughput drop, i.e., around 50% in Exon, 80% in UDT, and more than 90% in TCP. This is not surprising due to the delays and network congestion caused by retransmissions of failed packets in the three protocols. The drop was however sharper in the TCP and UDT cases because of the congestion control they use. Indeed, following the retransmission timeout (RTO) degradation scheme [75] causes a faster drop in throughput than Exon as shown in the 5% packet loss case. To the contrary, the retransmission timeout in Exon is

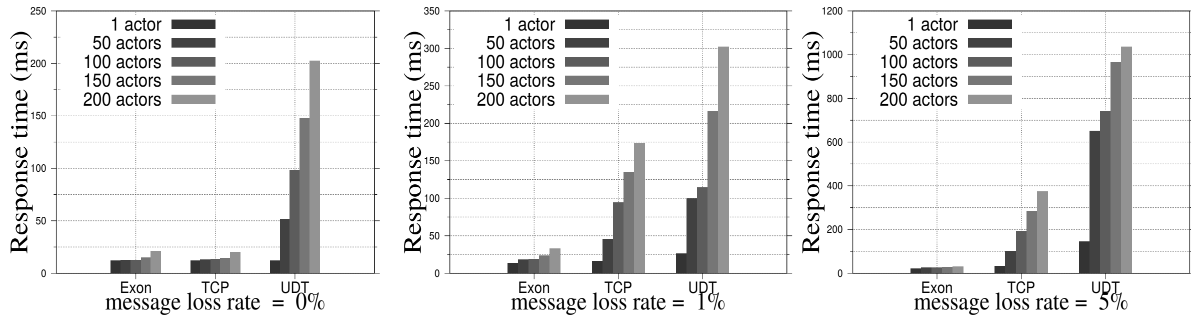


Figure 31: RPC latency under packet loss (Setting: RTT=10ms, Bandwidth=100 Mbps)

less pessimistic, i.e., proportional to the network RTT, which is possible given the protocol's core resilience to dropping and duplication. Also we can see that the dropping of UDT is lighter than that of TCP since that the DAIMD in UDT does not overreact to packet loss as the AIMD in TCP. In the 5% packet loss rate, Exon's throughput is 8 times higher than TCP and UDT, which demonstrates its high resilience to hostile networks with packet loss.

5.3.1.2 RPC messaging

Here we test the scenario in which a host (client) contains a given number of independent actors, each actor performing RPCs to the other host (server) in a sequential loop. For TCP, a single connection is shared by all actors, as is common in real general purpose distributed actor middleware, like Erlang. We aim to see how many requests per second (in aggregate) can be achieved and RPC latency, increasing the number of actors until we saturate the bandwidth.

Figures 31 and 32 convey the latency and throughput results for 0%, 1%, and 5% packet loss rate experiments, in the same WAN setting as before (RTT=10ms, Bandwidth=100 Mbps). In general, the conclusions are similar to the one-way, demonstrating the high resilience of Exon to packet loss, in terms of performance, compared to TCP and UDT.

For the *fault-free* test, UDT has the worst performance compared with Exon and TCP where they have similar performance, a little worse or better depending on the number of actors. It can be seen that as we increase the number of actors, the RPC latency increases, first slowly, and then abruptly when the number of client actors (200) saturates the available bandwidth; unlike the UDT case, where the RPC latency increases rapidly as the number of actors increase. As in the one-way case, UDT has the worst performance, where that of Exon and TCP increases roughly linearly with the number of actors, until reaching network saturation, when it even decreases slightly.

For scenarios with *packet loss*, even 1%, the performance of TCP and UDT drops drastically, compared with Exon, even more than for the one-way tests. The reason is that, while for Exon each message is delivered independently, not delaying other messages in case of packet loss, for TCP packet loss will delay the whole stream, which must be delivered in order. This means that, for TCP, for a single packet loss, all other concurrent actors will have their requests or responses delayed, increasing RPC latency, as can

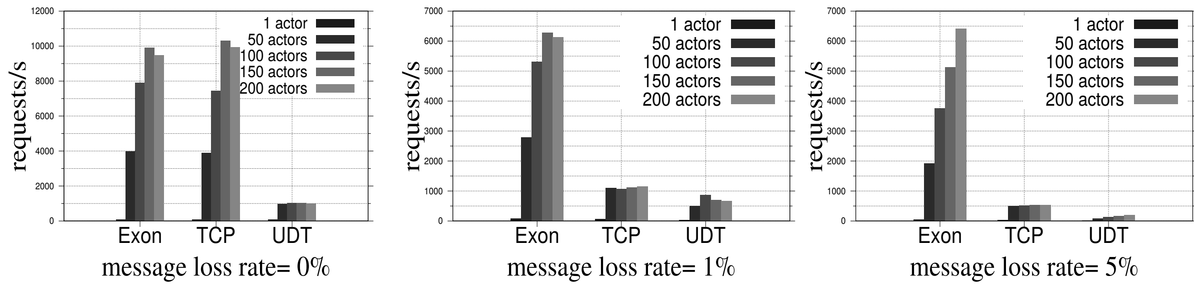


Figure 32: RPC throughput under packet loss (Setting: RTT=10ms, Bandwidth=100 Mbps)

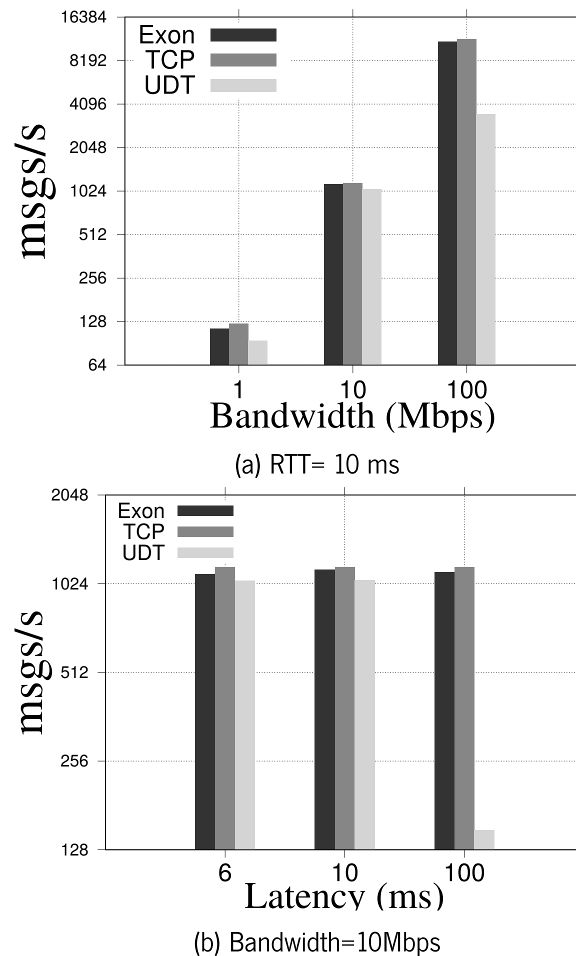
be seen in Figure 31, and delaying the issue of their next RPC. It can be seen in Figure 32, for both 1% and 5% packet loss cases that, while for Exon throughput still scales linearly with the number of actors (before saturating the network), for TCP and UDT throughput stops scaling much sooner. For 5% packet loss, TCP throughput stops at around 500 requests/s, UDT at around 200 requests/s, while Exon reaches around 6400 requests/s. This shows the serious impact of packet loss, due to HOL blocking, when using multiplexed TCP connections in general purpose middleware, something which unfortunately is common.

5.3.2 Overhead under Normal Conditions

In this experiment, we aim to compare the throughput and latency of Exon, TCP, and UDT under different bandwidths and latencies, in a fault-free scenario, for the two messaging patterns (one-way and RPC).

5.3.2.1 One-way messaging

Figure 33a shows the throughput of Exon, TCP, and UDT for RTT=10ms, and bandwidths of 1, 10, and 100Mbps. In the three cases, the three protocols make almost full bandwidth utilization except UDT under the 100Mbps case. This is expected since the network is saturated with the successive one-way 1KB messages, where the RTT effect is negligible. Nevertheless, the overhead in Exon is 8% in the worst case, and this is referred to the overhead of REQSLOT messages that asks for a window of N slots, as mentioned above. This small overhead justifies the use of Exon in systems that require EO guarantees and exhibit some packet loss.
Regarding UDT, its performance increases as bandwidth increases, however it does not reach the maximum bandwidth utilization with 100 Mbps, and it shows bad performance comparing with Exon and TCP.
Figure 33b shows that the full bandwidth utilization remain the same with a fixed bandwidth and different RTT (6, 10 and 100 ms) for the same reasons. However, UDT shows bad performance with high latency.



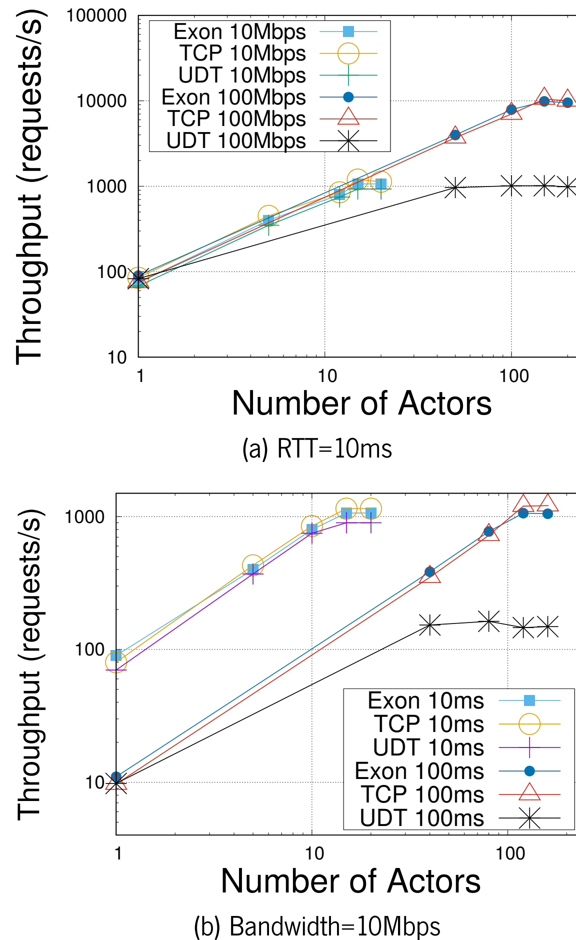
(a) RTT= 10 ms

(b) Bandwidth=10Mbps

Figure 33: One-way throughput as bandwidth and RTT varies

5.3.2.2 RPC messaging

We aim to see how many requests per second (in aggregate) can be achieved, using several concurrent actors, each performing RPCs in a loop, increasing the number of actors until we saturate the bandwidth. Figure 34a conveys the throughput with varying bandwidth of 10 and 100 Mbps (with RTT=10ms). We observe a max of 7% throughput overhead of Exon over TCP in the worst case. Both protocols hit the bandwidth limits as the number of actors increase (which is higher for higher bandwidth) unlike with UDT where as the number of actors increase, the performance got worse, and this is consistent with the one-way case in sending burst of messages.
On the other hand, the throughput is negatively affected when RTT varies (between 10 and 100ms) as shown in Figure 34b (with fixed bandwidth=10Mbps).
In the RPC pattern, the RTT is major factor since a successor message depends on the round-trip delay of the previous one. However, as more actors are used, the channel gets more utilized and the protocols hit the bandwidth limits (10Mbps). Also, UDT has the worst performance as the number of actors increase, and this conforms with the result of the throughput experiment one-way which has a similar scenario (sending one-way burst messages).



(a) RTT=10ms

(b) Bandwidth=10Mbps

Figure 34: RPC throughput as bandwidth and RTT varies-log. scale

A nice observation (in Figure 34b) is that as the RTT increases, the difference between Exon and TCP protocols almost fades away (i.e., from 10% to 2%). The same is observed in Figure 34a where the overhead degrades from 7% to 1% as the bandwidth increases.

This can be explained by the relative overhead of the REQSLOTS and SLOTS messages in these scenarios, as the algorithm sends a new REQSLOTS message upon receiving a SLOT (as long as some TOKEN messages where sent). This problem is easily fixed either by piggybacking REQSLOTS messages in TOKEN, and SLOTS in ACK, or by having a low-high watermark system for envelopes in reserve, so as to send REQSLOTS less frequently. We leave this improvement for further work.

5.3.2.3 RPC messaging - On the Same Machine

We aim to isolate the algorithmic performance of Exon and TCP without the influence of network factors. The experiments involved a client and server running on the same machine, eliminating network latency and bandwidth constraints. Figure 35 presents the response time comparison between Exon and TCP in the local machine scenario.

From the graph, we observe that TCP exhibits slightly better response times compared to Exon in this local

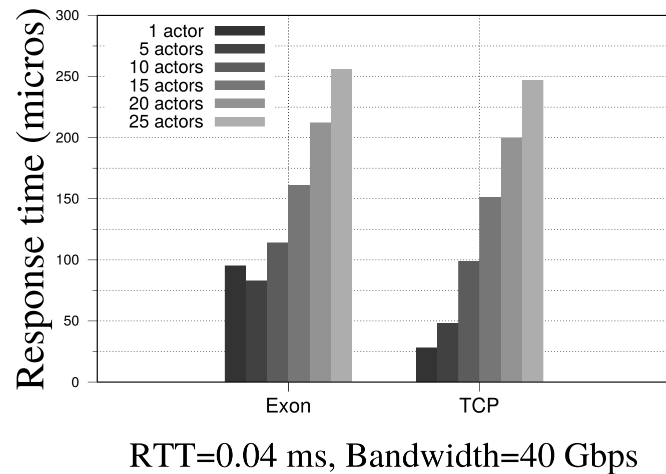


Figure 35: RPC Latency - same machine (Setting: RTT=0.04ms, Bandwidth=40 Gbps)

machine setup. However, the difference in performance between the two protocols is not significant, and almost fades away as the number of actors increases.
This can be explained by, as in the other experiments, the relative overhead of the protocol messages of Exon (REQSLOTS and SLOTS messages).

5.4 Conclusion

We evaluated the performance of the three protocols: Exon, TCP, and UDT, under different network conditions. We conducted experiments in a fault-free environment, as well as in a message loss environment, testing the protocols under three variables: bandwidth, network latency, and message loss rate. We used the one-way messaging and RPC messaging patterns to measure the throughput and response latency. The results showed that, under normal circumstances, Exon and TCP had similar throughput. However, under a 1% and 5% packet drop rate, Exon's throughput was significantly higher than TCP and UDT, demonstrating its high resilience to hostile networks with packet loss. We also found that the retransmission timeout in Exon is less pessimistic than TCP and UDT, resulting in less degradation of throughput.
Moreover, the evaluation shows that Exon imposes a negligible overhead over TCP in healthy networks, which makes it a promising candidate for improving network performance in lossy and congested network environments.

6 Partition Tolerance through Delegation

6.1 Motivation

The novelty of Exon is simultaneously achieving exactly-once message delivery, obliviousness, efficiency and no dependence on timing assumptions for correctness. These features make Exon fits anywhere, and a persistent need in many systems and applications. It fits in datacenters where thousands of micro-services need to connect, replacing TCP or other protocols. Also, it fits in systems that need exactly-once message delivery and having intermittent connections, such as mobile applications, V2X, and Peer-to-Peer networks.
In these systems, where they have a high number of communicating nodes, they need short-lived connections in order to push some messages to a stable or moving unit and go away without keeping any state information about that unit, and this is because these nodes are constrained devices, and even if they are not constrained (e.g. have enough memory), they do not want to keep connection specific information about other nodes, which can be accumulated over time and affect the performance, and if deleted may lead to message duplication.

The Exon features, specially the obliviousness, where a node keeps only a single integer per node as permanent state, are perfect for such applications and systems. However, in these systems, there is a problem in that if a sending node needs to send messages to another node and they are partitioned at the moment, and the sending node needs to go away (e.g. changing its location, draining battery, etc.), how can the sending node behave, and what to do with the messages. Another scenario is when the sending node is already communicating with another node and they become partitioned in the middle of a message exchange, then the sending node has a problem in what to do with the messages left (both payloads and/or tokens).

In general, systems with high number of nodes, are more exposed than others to the intermittent connections, for example and not limited to, the mobile applications, Automotive (V2X), etc.
In the Automotive domain, the vehicles are moving nodes, and a moving node may go online or offline

suddenly with reference to another node. For example, two vehicles may be communicating with each other, and each one goes suddenly from a different path, causing a connection break down.
Another example is mobile applications. Mobile devices are characterized by frequent disconnections which affect how distributed network communication can occur. These disconnections can be due to a number of reasons: out of range, device turned off, or application swapped away if an operating system is single tasking, and specially if these applications are not running in the background, where the connections need to be completely re-established every time the application is opened.

Therefore, there is a need for a feature in the Exon protocol, or the Exon protocol should be extended, where a sending node can send a message to an intermediate node in order to hand-off the message to a final destination node reliably. The intermediate node in its turn, playing the role of the forwarder, can hold the message and send it eventually to the destination as it sees it again.

6.1.1 Messaging Protocols Limitations, Including Exon

In many applications, nodes may become partitioned with respect to each other in a regular manner, either before starting a message exchange, or in the middle of a message exchange. If an application uses one of the message queues protocols, message reliability could be ensured between two "online" nodes, beside that they are heavy weight applications. Exon is an appropriate protocol to use in such applications. However, the protocol as it is, lacks the ability to deal with intermittent communications like most of the messaging protocols.
Exon communication (message exchange) exposed to two scenarios,

- nodes are already partitioned when attempting to EOSend having no envelopes
- nodes become partitioned after having received envelopes

Scenario 1 - Nodes are already partitioned when attempting to EOSend having no envelope: Node A wants to send a message to B, has currently no envelopes to B and, therefore, sends a reqslots message, but the two nodes are or become partitioned and no slots message arrives, node A needs to quit temporarily (become partitioned with respect to B), then, what to do with the queued messages.

Scenario 2 - nodes become partitioned after having received envelopes: In a node communication, intermittent nodes can become partitioned at any moment without giving any signal for the sending node to behave accordingly. For instance, node A is sending messages to node B by associating the upper layer messages to tokens. However, node B may become partitioned, and node A needs to quit soon for some reasons, either by changing its location, or the battery is going down, etc. In this scenario, node A has "some time" to hand-off the messages to another forwarding node. A forwarding node could be a regular node (as node A or node B), or a dedicated server that has a forwarding service for the other nodes.

The problem is what to do with the tokens of node B. tokens should not be removed before receiving an ack message from the destination, in order not to affect EO message delivery.

6.2 Alternative approaches for achieving delegation

This section explores several alternative approaches for achieving delegation in the Exon protocol.

6.2.1 Extended API

The upper layer messages could be queued in the message queue, or associated with tokens. Those messages could be delegated using an extended API, that can access the Exon node state, extract the queued messages or the tokens, and EOSends them to another node.
This needs an extended API, which allowed to it to introspect inside the protocol state, e.g., to query if there are messages in the queue, to dequeue messages, and then EOSend the messages to another available forwarding node. However, for that Exon-lib API would have to be much extended, and it would be difficult and burdensome to expose enough information for the client app to invoke appropriate commands (as the client does not know details about the different elements of the node state).
Furthermore, another issue arises with messages that are already associated with tokens. In such case, it may be challenging to retrieve the message from the token and send it with the extended API to another forwarding node, as this could result in message duplication. Also, transmitting the token itself to another node presents a more complex and challenging task.

6.2.2 Modifying the Exon Distributed Protocol

Extending or modifying the existing Exon distributed protocol to achieve delegation can be a challenging task that requires a significant amount of effort. This is because any modifications to the protocol can have significant implications on the protocol's correctness and performance, and must be thoroughly validated and tested to ensure that they do not introduce new bugs or issues.

In particular, extending or modifying the Exon protocol to support delegation would require the development of a whole new algorithm, which would need to be formally proven to be correct. This can be a complex and time-consuming process that involves analyzing the behavior of the protocol under different scenarios.

6.2.3 Basic Exon Algorithm with Structured Messages

One possible solution to address the issue in the aforementioned scenarios, and to avoid the difficulty and the burden of the API, is to employ message delegation in the Exon algorithm. However, rather than designing an extended protocol to enable delegation, which would require new proofs for a more complex protocol, the proposal is to reuse the unmodified Exon protocol with the same protocol messages.

However, the structure of the payload is extended to accommodate structured messages, which can be handled in various ways within the middleware upon being EO received. These methods include triggering delegating, forwarding, and other not yet conceived future usages.

The proposed approach allows for achieving delegation without requiring an extended API or significant modifications to the existing Exon protocol. By extending the payload structure of the existing protocol, message delegation can be enabled without introducing new complexities to the protocol that may affect its correctness or performance. Additionally, this approach is less burdensome for the client application as it does not require exposing detailed information about the node state or complex API calls. Moreover, it provides flexibility in handling structured messages, which can be useful for future use cases beyond delegation. Overall, this approach provides a simpler and more efficient solution for achieving delegation in the Exon protocol.

6.3 Extensible Exon Architecture

The Extensible Exon Architecture is a concept that allows for the creation of pluggable handlers, enabling the development of modular, extensible systems that can be easily customized to meet the needs of specific use cases. This architecture is based on the idea of structured messages and message handling, allowing the creation of complex message flows that can be easily managed and modified as needed.
With the Extensible Exon Architecture, it is possible to keep the existing Exon protocol without making any modifications to it. Instead, the focus is on using this protocol to deliver "commands" to other nodes that will be executed within the middleware and not only payloads.
In the following sections, I will provide an incremental presentation of the delegation process by defining a progressively more sophisticated delegateConnection function.

The architecture includes four message types: dlv, fwd, tok, and rmvslots, where dlv can be used to send simple payloads, fwd used to delegate messages to other nodes, and those delegated fwd messages could encapsulate simple payloads dlv or tokens tok to another node. Also the rmvslots message type that is used for Garbage Collection. However, it is important to note that other message types could be added in the future to support even more use cases.
Algorithm 2 shows the modified parts only, the rest being the same as in Algorithm 1. The event $\text{receive}_{j,i}(\text{token}, s, r, m)$ triggers the execution of the procedure $\text{consumeToken}_i(j, s, r, m)$ that contain the exact code as the previous handler for token messages, from Algorithm 1, except that the deliver procedure has been replaced by a new handler procedure handle that handles each message based on its type. If the message is a:

- dlv, the $payload$ will be delivered to the upper layer application

```
1  types
2      𝕀 : node id
3      ℙ : payload
4      𝕄 : (dlv, ℙ)|(fwd, 𝕀, 𝕄)|(tok, 𝕀, ℕ, ℕ, 𝕄)|(rmvslots, 𝕀, ℕ, ℕ)
5  on receive_{j,i}(token, s, r, m)
6      consumeToken_i(j, s, r, m)
7  proc consumeToken_i(j, s, r, m)
8      if j ∈ dom(R_i) then
9          c = R_i[j]
10         if r = c.rck and s ∈ c.slt then
11             c.slt.remove(s)
12             handle_i(m)
13     send_{i,j}(ack, s, r)
14 proc handle_i(m)
15     case m of
16         (dlv, payload) then deliver_i(payload)
17         (fwd, j, m) then EOSend_i(j, m)
18         (tok, j, s, r, m) then consumeToken_i(j, s, r, m)
19         (rmvslots, j, x, y) then removeSlots_i(j, x, y)
20 proc removeSlots_i(j, x, y)
21     c = S_i[j]
22     for i ← x to (y − 1) do
23         c.slt.remove(s)
24     c.sck := y
25     if c.slt = ∅ then
26         R_i.remove(j)
```

Algorithm 2: The Extensible Exon

- fwd, the $\mathsf{EOSend}_i(j, m)$ procedure is called in order to send/queue the message received to another node; either to another forwarder node, or to the final destination node
- tok, the $\mathsf{consumeToken}_i(j, s, r, m)$ procedure is called, and here the message is a token that is arrived from the first initiating node through a forwarder node or nodes.
- rmvslots, the removeSlots procedure is called for Garbage Collection purposes. (will be discussed in 6.6)

The main modification made to Exon involves altering the message type and incorporating the consumeToken procedure. This updated procedure uses the handle function instead of the previous deliver function to manage the various types of messages encountered in the system.

```
1 proc delegateConnection_i(j, k)
2     c = S_i[k]
3     while c.msg ≠ [] do
4         EOSend_i(j, (fwd, k, (dlv, c.msg.dequeue())))
5     for (s, m) in c.tok do
6         EOSend_i(j, (fwd, k, (tok, i, s, c.rck, m)))
```

Algorithm 3: The delegateConnection procedure added to the Exon algorithm

6.4 Basic Delegation

The algorithm extension for basic delegation introduces a novel procedure, denoted as $\text{delegateConnection}_i(j, k)$, which allows a node to delegate the responsibility to a forwarder node to handoff messages to another node (currently partitioned) by asking it to forward dlv or tok messages to other nodes.

The $\text{delegateConnection}_i(j, k)$ procedure presented in Algorithm 3 is responsible for delegating a connection for node k to a forwarder node j, and it accomplishes this by encapsulating messages and tokens in appropriate types of messages. This procedure iterates over the messages queued in the connection, either payloads or tokens. For each message that is not yet associated to a token, it encapsulates it in a dlv message, and then the dlv in a fwd message. The message m encapsulated in the dlv message is removed from the message queue.

For messages that are already associated to tokens, the procedure encapsulates them in a tok message, which is also encapsulated in a fwd message, but the tokens themselves are kept in the originating node. Therefore, the forwarder node is asked to forward the dlv or the tok message to another node.

Thus, why the $payload$ messages can be removed safely from the queue, but the tokens cannot.

6.4.1 Possibility of dlv Oblivious Delegation

If the destination node becomes partitioned with respect to the sending node, it means that the two nodes are no longer able to communicate directly with each other. In this case, the sending node may want to delegate the connection, where there are dlv messages in the msg queue, the sending node dequeues the dlv messages from the msg queue, and EOSend them to the forwarder node. However, since there are no corresponding slots for them at the destination, then the sending node can safely remove them from the msg queue.

If the partitioned node becomes reachable to the sending node at a later time, the destination node may send a slots message in response to a previously received REQSLOT message, and consequently create envelopes that will be ready for any new messages, and this scenario would not affect the correctness of message delivery, since the messages are already handed-off to another node.

6.4.2 Impossibility of token Oblivious Delegation

If a token $T_{A \to B}$ is created at a sending node A, but B becomes partitioned with respect to A, node A would not know if the corresponding slot S at B is still available. Then, if the token forwarding mechanism performs a complete garbage collection of the tokens and the connection record at A, and if B becomes connected to A before F (the forwarding node) forwards the token to B, node B could check (in the periodically) if the slots are still relevant trying to garbage collect the slots on the receiver-side connection. If there is no information in A, then A will send a reqslots with an advanced sck, effectively informing B that there are no tokens for that old range of slots, and B can garbage collect the receiver-side connection record, including slot S. Eventually, when the token T reaches B through F, node B will discard the token as slot S was not present, which can result in a message loss.

6.4.3 Delegating Messages Scenarios - Messaging Steps

I now present some examples on how the extensible Exon works. For simplicity, I assume the communication is occurring between a sender node A and a receiver node B, but if nodes A and B are partitioned at the moment, and node A needs to quit, therefore node A can send the messages via a forwarding node F.
Message forwarding can take place when node A has

(a) a dlv message to node B, and it does not yet has an envelope for it, and this is because node A requested some slots from node B and did not get any reply. Then A decides to handoff the dlv to F, where F can forward it to B eventually;

(b) an already created token for B, and it decides to send the token encapsulated in a fwd message to a forwarder node F, where F can send the token to B eventually;

6.4.3.1 Delegating dlv

Delegating dlv messages occurs when Node A intends to send a message to B, but has no available envelopes to do so. Node A sends a reqslots message to B to request slots, but in cases where the two nodes are partitioned or become partitioned, the slots message may not be received, resulting in a loss of communication. In such cases, and since node A needs to quit/suspend, it resorts to delegation.
Actions at node A: As depicted in Figure 36, node A needs to send a message payload m to node B via the $\text{EOSend}_A(B, m)$ command. Node A then proceeds to enqueue it in the message queue $S[B].\text{msg} = [(\text{dlv}, m)]$, followed by requesting slots from node B by sending a reqslots message. Due to network partitioning, it is possible that either the $\text{reqslots}_{A \to B}$ or the $\text{slots}_{B \to A}$ may be lost in transit. Therefore, no envelopes available at node A for node B.

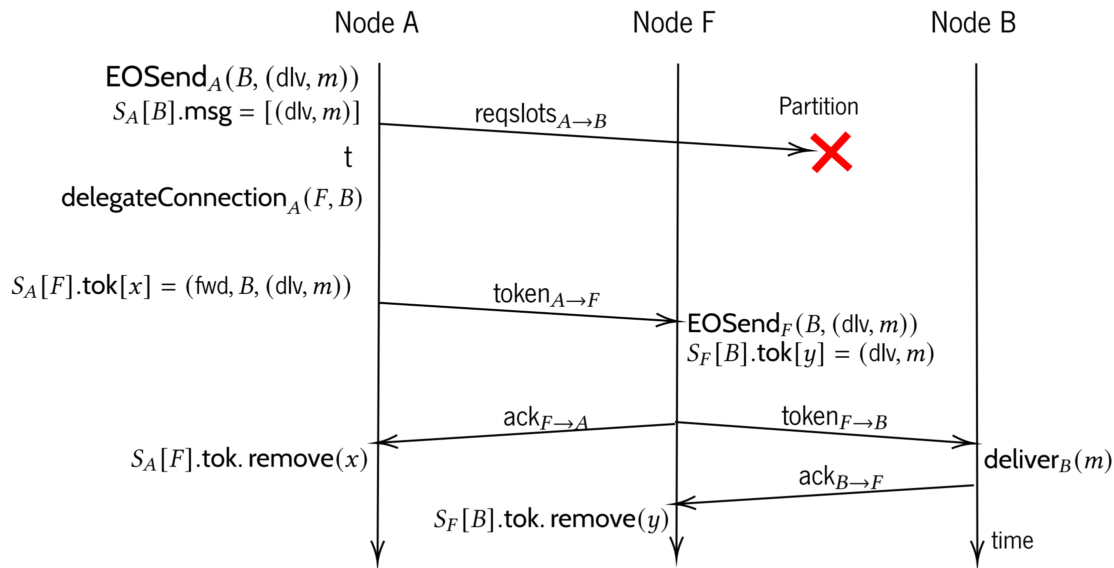


Figure 36: Delegating *payload*

After some time **t**, node A makes the decision to delegate the message queued for node B to the forwarder node F via the delegateConnection$_A(F, B)$ command. This decision is due to that node A wants to quit or suspend.

In the delegateConnection procedure, the message (dlv, m) will be dequeued from msg queue for node B, and EOsent to node F, in a fwd message (fwd, B, (dlv, m)).

For simplicity, to keep the example small, this example assumes that node A already has an envelope x for node F in order to avoid the reqslots – slots exchange with F.

Actions at node F: The received message will be handled, and given that it contains a fwd message, the enclosed message is forwarded to node B (EOSend$_F(B, (\text{dlv}, m))$).
Also for simplicity, this example assumes that node F already has an envelope y for node B. Therefore, as node F can communicate with node B, it sends the queued message to it.
Actions at node B: Since the message received is a dlv message, then the payload m will be extracted and delivered to the specific upper layer application (deliver$_B(m)$).

6.4.3.2 Delegating token - One Forwarding Node

Actions at node A: In this scenario, as shown in figure 37, and for simplicity, this example assumes that node A already has an envelope x for node B. Therefore, node A sends a token message to node B that contains (dlv, m). Then, due to network partitioning, it is possible that either the token$_{A\to B}$ or the ack$_{B\to A}$ may be lost in transit.

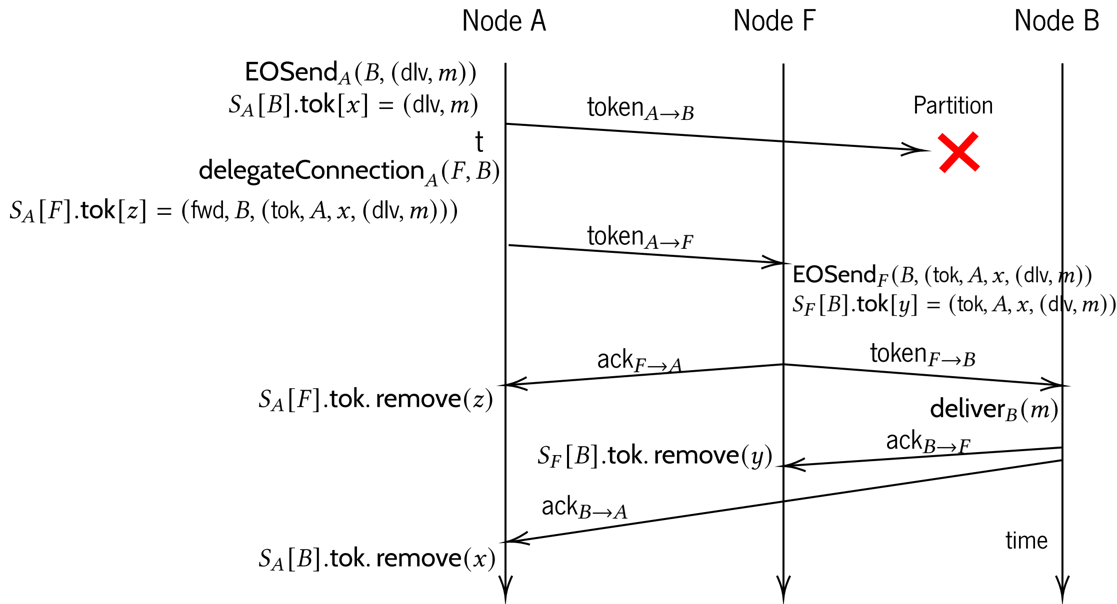


Figure 37: Delegating token - One Forwarding Node.

After some time **t**, node A makes the decision to delegate the token for node B to the forwarder node F via the delegateConnection$_A(F, B)$ command. This example assumes that node A already has an envelope z for node F. Then, node A EOsends the token$_{A\to B}$ to node F in a fwd message (EOSend$_A(F, (\text{fwd}, B, (\text{tok}, A, x, (\text{dlv}, m))))$).

Actions at node F: The received message will be handled, and given that it contains a fwd message, the enclosed message (with its content) is forwarded to node B (EOSend$_F(B, (\text{tok}, A, x, (\text{dlv}, m)))$).

This example assumes that node F already has an envelope y for node B. Therefore, as node F can communicate with node B, it sends the queued message to it via the token$_{F\to B}$.

Actions at node B: As the received message is of type tok, it will be processed by extracting the encapsulated dlv message and delivering it to the relevant upper layer application (deliver$_B(m)$).

6.4.3.3 Delegating token - Multiple Forwarding Nodes

This scenario involves token delegation, where the originating node delegates messages to a forwarder node, and the forwarder node can further delegate the messages to another forwarding node.

Figure 38 shows the scenario of four nodes, node A sends a message to node B, but as node B is partitioned, node A decides to delegates the messages to the forwarding node F. However, node F, and for some reasons, may decide also to delegate the messages through another forwarding node G, so that eventually G can send the messages to node B.

Actions at node A: As in the previous scenario, node A already has an envelope x for node B, and due to network partitioning, node A makes the decision to delegate the token for node B to the forwarder node F via the delegateConnection$_A(F, B)$ command.

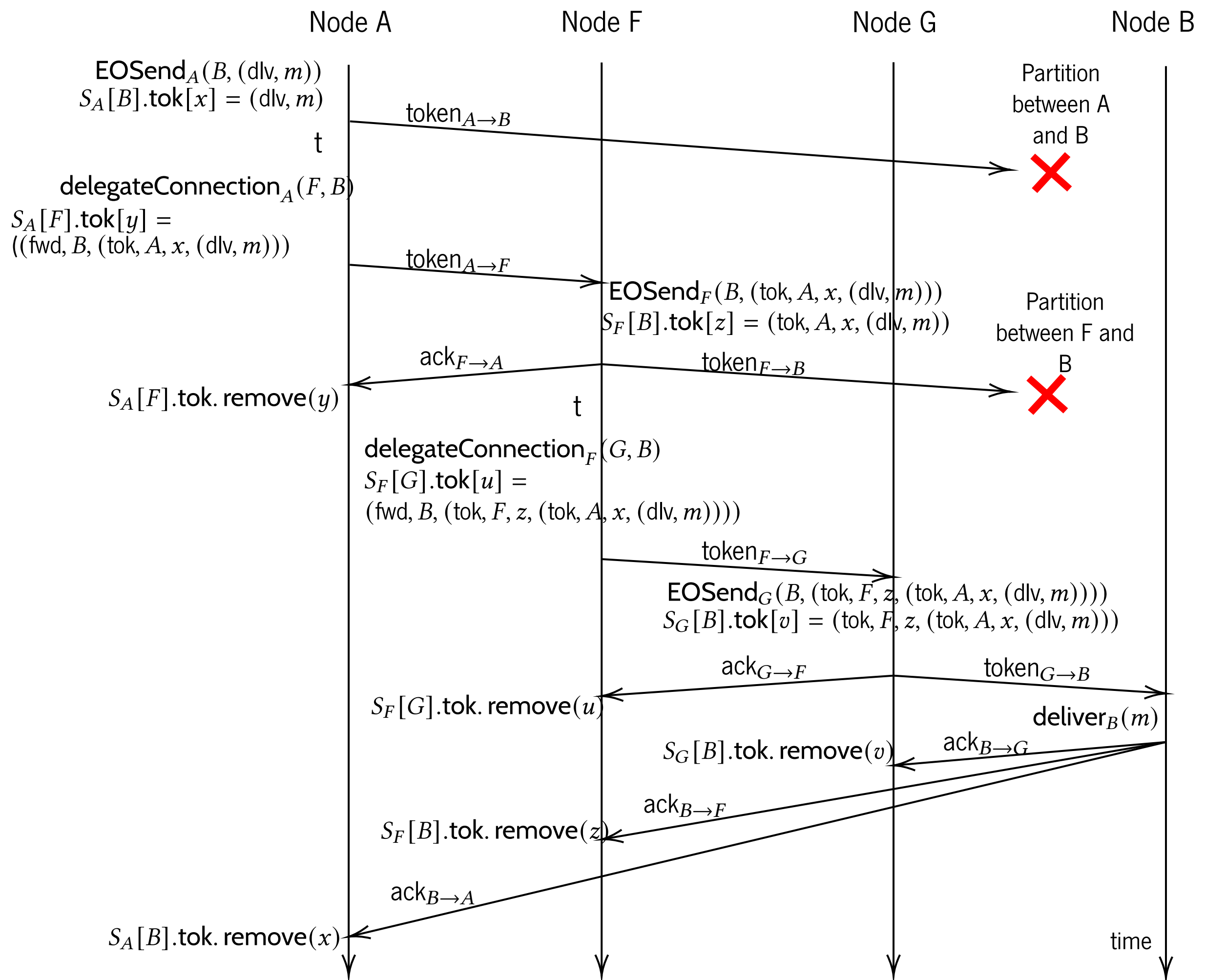


Figure 38: Delegating token - Multiple Forwarding Nodes.

The **delegateConnection** procedure at node A **EOSend**s the token$_{A \to B}$ to node F in a fwd message (**EOSend**$_A(F, (\text{fwd}, B, (\text{tok}, A, x, (\text{dlv}, m)))))$. This example assumes that node A already has an envelope y for node F.

Actions at node F: The received message will be handled, and given that it contains a fwd message, the enclosed message $(\text{tok}, A, x, (\text{dlv}, m))$ is forwarded to node B (**EOSend**$_F(B, (\text{tok}, A, x, (\text{dlv}, m)))$).

This example assumes that node F already has an envelope z for node B. Therefore, it **EOSend**s the queued message to B via a token$_{F \to B}$. However, due to network partitioning, node F makes the decision to delegate the token for node B to the forwarder node G via the **delegateConnection**$_F(G, B)$ command. This example assumes that node F already has an envelope u for node G. The fwd message sent from node F to node G contains the token$_{F \to B}$ that contains the token$_{A \to B}$ $(\text{fwd}, B, (\text{tok}, F, z, (\text{tok}, A, x, (\text{dlv}, m))))$
Actions at node G: Node G behaves as node F in the previous scenario. The received message will be

handled, and given that it contains a fwd message, the enclosed message $(\text{tok}, F, z, (\text{tok}, A, x, (\text{dlv}, m)))$ is forwarded to node B ($\textsf{EOSend}_F(B, (\text{tok}, F, z, (\text{tok}, A, x, (\text{dlv}, m))))$), assuming that node G already has an envelope v for node B.

Actions at node B: As the received message is of type tok $(\text{tok}, F, z, (\text{tok}, A, x, (\text{dlv}, m)))$, it will be "consumed" by extracting the encapsulated tok $(\text{tok}, A, x, (\text{dlv}, m))$, which will be also "consumed" by extracting the encapsulated dlv message (dlv, m) and delivering the payload m to the relevant upper layer application ($\textsf{deliver}_B(m)$).

6.5 Avoiding Nesting Along Delegation Chains

The increase of nesting happens when an intermediary forwarding node F is delegating to another forwarding node G the responsibility of forwarding to node B a token that contains a tok command, as the scenario in 6.4.3.3. The chain of the toks inside toks will grow as the number of forwarding nodes increases where node B is still partitioned with respect to those forwarding nodes, and they want to quit. In this case, the forwarding nodes may have two scenarios, either they already have envelopes from node B, or not. There will be no nesting in the latter case, however, in the first case, tokens will be created at the forwarding nodes towards node B, and as the forwarding node delegate connection that includes tokens to another forwarding node, the nesting will occur.

In other words, node A delegates the connection of node B to node F by sending a tok message that contains the information of the "$\text{token}_{A \to B}$". Node F then queue the received tok, creates a "$\text{token}_{F \to B}$", and encapsulates to it "$\text{token}_{A \to B}$". However, node F wants to quit and delegate the connection of node B to another forwarding node G, in this case, node F will have a "$\text{token}_{F \to G}$" that contains the "$\text{token}_{F \to B}$", where the latter contains the "$\text{token}_{A \to B}$". This chain could grow as the number of forwarders.

A possible solution could be done, as in Algorithm 4, by checking if the queued message is not of type dlv, which means that the message is a forwarded message. Therefore, instead of encapsulating in the fwd message the "$\text{token}_{F \to B}$" that holds the "$\text{token}_{A \to B}$", the "$\text{token}_{A \to B}$" only encapsulated.

Overall, the importance of this point is to ensure that delegation chains are efficient and do not lead to unnecessary nesting, which can affect the performance of the system. The proposed solution can help in reducing the nesting and improving the efficiency of delegation chains.

6.6 Receiver-side Oblivious Delegation

A receiver-side connection is only removed upon communicating with the sender, which only happens when the sender already removed the sender side connection.

```
1  proc delegateConnection_i(j, k)
2      c = S_i[k]
3      while c.msg ≠ [] do
4          EOSend_i(j, (fwd, k, (dlv, c.msg.dequeue())))
5      for (s, m) in c.tok do
6          if m instanceOf dlv then
7              EOSend_i(j, (fwd, k, (tok, i, s, c.rck, m)))
8          else
9              EOSend_i(j, (fwd, k, m))
10             c.tok.remove(s)
```

Algorithm 4: Avoiding Nesting Mechanism

```
1  proc consumeToken_i(j, s, r, m)
2      if j ∈ dom(R_i) then
3          c = R_i[j]
4          if r = c.rck and s ∈ c.slt then
5              c.slt.remove(s)
6              if c.slt = ∅ then
7                  R_i.remove(j)
8              handle_i(m)
9      send_{i,j}(ack, s, r)
10 proc delegateConnection_i(j, k)
11     c = S_i[k]
12     x = if c.env = [] then c.sck
13         else c.env[0]
14     y = c.sck + N + |c.msg| − |c.env|
15     c.env := []
16     c.sck := y
17     while c.msg ≠ [] do
18         EOSend_i(j, (fwd, k, (dlv, c.msg.dequeue())))
19     for (s, m) in c.tok do
20         EOSend_i(j, (fwd, k, (tok, i, s, c.rck, m)))
21     EOSend_i(j, (fwd, k, (rmvslots, i, x, y)))
```

Algorithm 5: Receiver-side Oblivious Delegation Mechanism

To achieve a receiver-oblivious delegation that enables node A to retire or suspend without requiring future communication with node B, a new mechanism is proposed. This mechanism involves a command that removes the receiver-side connection (R-record of the source node). This approach would keep the original Exon protocol messages intact.

The new "GC" command is rmvslots. It contains the source node and the range of slots to be removed,

that have no related tokens to them, to be carried in the forwarded messages as the dlv and tok, which when handled at the receiver side, removes that range of slots created for that sender.
The sender has to send the range of slots to be removed based on the envelopes it has. However, the receiver node can have created more slots that the sending node does not know about (if the slots message is lost). Therefore, the range will be between the oldest available envelope (sck if no envelopes exist) and the most recent requested slot that can be calculated using the following formula: $y = \text{sck}+N+|\text{msg}|-|\text{env}|$.

Thus, as in Algorithm 5, when the sender A wants to delegate the connection, in addition to delegating the dlvs and toks if exist, it delegates a rmvslots to the forwarded node, removes all the envelopes ($c.\text{env} := []$), and set the $c.\text{sck} := y$ where this has the effect of refusing to create envelopes below y in case such slots were created and a slots message arrives later. As the rmvslots reaches the final destination node, it removes any possible slots at it, that may have been created, for which there are no tokens, i.e., between x and y when the rmvslots message is handled 2.

In certain cases, the rmvslots message may reach the destination before the tokens, which would result in the R-record not being removed, except when receiving a reqslots with n=0. However, this delay in garbage collection can be resolved by making a small adjustment to the consumeToken procedure. The adjustment involves checking if the slot set is empty and removing the R-record. The system is not adversely affected by this small adjustment, as removing a receiver-side connection (R-record) only occurs when there are no slots.

6.7 Discarding State at the Sender

The sending node keeps the tokens (with its payload) of the destination node even if it delegates to other nodes the responsibility of delivering the tokens to the destination node, which could increase the memory consumption at the sender as well as the forwarding nodes. Removing the tokens at the sender is impossible as discussed in 6.4.2. However, an improvement could be made here by having a “null” payload at a token. I.e., using a token as placeholder with empty payload, to avoid the destination node from wrongly removing slots.
This could happen, as in Algorithm 6, when the sender node is delegating a token to a forwarder node, it then sets the token payload to null ($\text{tok}[s] := null$).
The receiver side should check, in the consumeToken procedure, if the received message is not null, it removes the slot, “handle”s the payload, and send the ack, otherwise, it just ignores it without sending an ack.

```
1  proc delegateConnection_i(j, k)
2      c = S_i[k]
3      while c.msg ≠ [] do
4          EOSend_i(j, (fwd, k, (dlv, c.msg.dequeue())))
5      for (s, m) in c.tok do
6          if m instanceOf dlv then
7              EOSend_i(j, (fwd, k, (tok, i, s, c.rck, m)))
8              c.tok[s] := null
9          else
10             EOSend_i(j, (fwd, k, m))
11             c.tok.remove(s)
12 proc consumeToken_i(j, s, r, m)
13     if j ∈ dom(R_i) then
14         c = R_i[j]
15         if r = c.rck and s ∈ c.slt then
16             if m ≠ null then
17                 c.slt.remove(s)
18                 handle_i(m)
19                 send_{i,j}(ack, s, r)
20         else
21             send_{i,j}(ack, s, r)
22     else
23         send_{i,j}(ack, s, r)
```

Algorithm 6: Discarding State Mechanism

6.8 Summary

In this chapter, I introduce the need for a reliable message hand-off feature in the Exon protocol, which allows a sending node to delegate a message to an intermediate node for eventual delivery to the final destination node. The chapter presents three alternative approaches for achieving delegation, including extended API, modifying the Exon Distributed Protocol, and using the Basic Exon Algorithm with Structured Messages. The chapter then focuses on the Basic Delegation approach and discusses the Extensible Exon Architecture, Receiver-side Oblivious Delegation, and Discarding State at the Sender. Additionally, the chapter covers the issue of avoiding nesting along delegation chains.

The modifications presented, which involve adding only the necessary parts for each feature in relation to the original algorithm, are not a change in the Exon protocol itself because they do not alter the fundamental rules or principles of the protocol. The modifications are related to the implementation of the protocol rather than the protocol itself. These modifications do not affect the underlying principles or rules of the protocol. Instead, it is an optimization to improve the performance or efficiency of the implementation.

All the features discussed in this chapter are implemented together in one package, you can refer to the full-package Exon implementation in the Appendix A.

7

Applications

In the evolving realm of modern communication systems, the need for scalability, efficiency, and reliability remains a persistent motivation. As networks expand and more devices interconnected increase, innovative solutions are required to meet the demands of applications.

In this chapter we describe how Exon fits several applications, application domains, or approaches to building applications, such as, Distributed Aggregation, Pat Helland's innovative approach to structuring applications, Enhancements in Online Booking Distributed Systems, the landscape of the Automotive Domain, and the critical role of Messaging Support for Distributed Middleware.

7.1 Distributed Aggregation

Data aggregation is a technique used to manipulate distributed data, control the network, and perform lightweight computations (like max, sum, count...) [76–78]. Aggregation protocols enable the deployment of distributed, lightweight computations within ad-hoc networks, following a peer-to-peer approach. However, most of these protocols do not tolerate message loss, whose occurence leads to computing a wrong aggregation result.

While these protocols are characterized by their simplicity, this simplicity comes with the trade-off of necessitating specialized fault tolerance techniques to address issues such as message loss and duplication. The solutions to these challenges can sometimes incur significant costs, as noted in the reference [79]. The Flow-Updating (FU) [80] protocol, unlike the others, is an aggregation protocol that relies on the concept of "flows". Each node maintains a record of the flow to each of its neighbors, and when a node sends a message to its neighbor, it carries the flow information. Then, each node calculates the average using contributions from in/out flows of neighboring edges and its initial value. Messages exchange serves the purpose of updating the flows, where the value conveyed in a subsequent message replaces the previous one, rather than adding to it. Message semantics are thus idempotent, and make the algorithm resilient to message loss or duplication.

On the other hand, PS [78] and DRG [77] are prone to communication issues. Using a robust messaging layer as Exon solves their mass conservation problem under unreliable networks, while keeping their simplicity and performance advantages in networks with high node degrees.

In Chapter 8, I will focus on the aggregation protocol as a case study. This involves a comprehensive examination of aggregation protocols and conducting evaluations to gain insights into how message failures might impact mass conservation.

7.2 Pat Helland's Vision using Exon

In his paper "Life beyond Distributed Transactions: an Apostate's Opinion" [2], Pat Helland envisions the construction of highly scalable (almost-infinite) applications by departing from traditional Distributed Transactions, which become impractical at scale due to their complexity and performance limitations. Helland's proposed paradigm involves structuring application data into "entities" with independent serialization scopes, facilitating efficient local transactions within each entity to ensure consistency and reliability. However, this model does not support transactions spanning multiple entities. Instead, remote data access is achieved through separate communication channels, adopting a message-oriented approach.

In such approach, achieving reliability necessitates a programmatic abstraction similar to a long-lived TCP connection. Such capabilities are "rarely available" to developers tasked with constructing scalable applications. Therefore, in most scenarios, we need to work with the "at-least-once" message delivery.
However, to avoid message delivery duplicates, the scale-agnostic (higher-level) portion of the application must implement mechanisms to ensure that the incoming message is idempotent. "Duplicate elimination could certainly be built into the scale-aware parts of the application. So far, this is *not yet available*. Hence, we consider what the poor developer of the scale-agnostic application must implement", Pat Helland.
In other words, ensuring reliability by assuming at-least-once message delivery, together with implementing message idempotency at the higher layer, makes writing applications becomes complicated. While using Exon, with its exactly-once message delivery will simplify the process of writing applications, making the approach more widely applicable. This abstraction allows developers to focus more on the core functionality of their applications.

7.2.1 Remembering Messages as State

To achieve idempotent processing for messages that lack natural idempotence, the entity must retain a record of their processing status. This record, essentially a form of state, accumulates as messages undergo processing.
Storing messages as state serves as a technique to establish idempotence in message processing. While this approach is effective, it introduces a challenge related to the potentially infinite expansion of state.

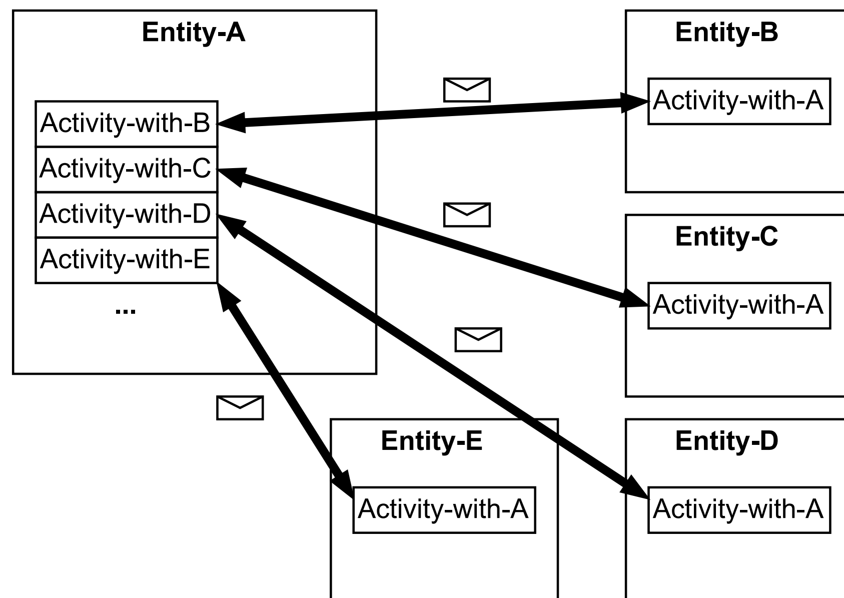


Figure 39: If an entity works with many partners, it will have many activities. These are one per partner.

With the incorporation of Exon, there is no necessity to maintain in the application state the information that a message has been processed. This is because the risk of duplicates is eliminated, and there is also no requirement to store a response message for potential resending in the event of loss, as message loss is no longer a concern.

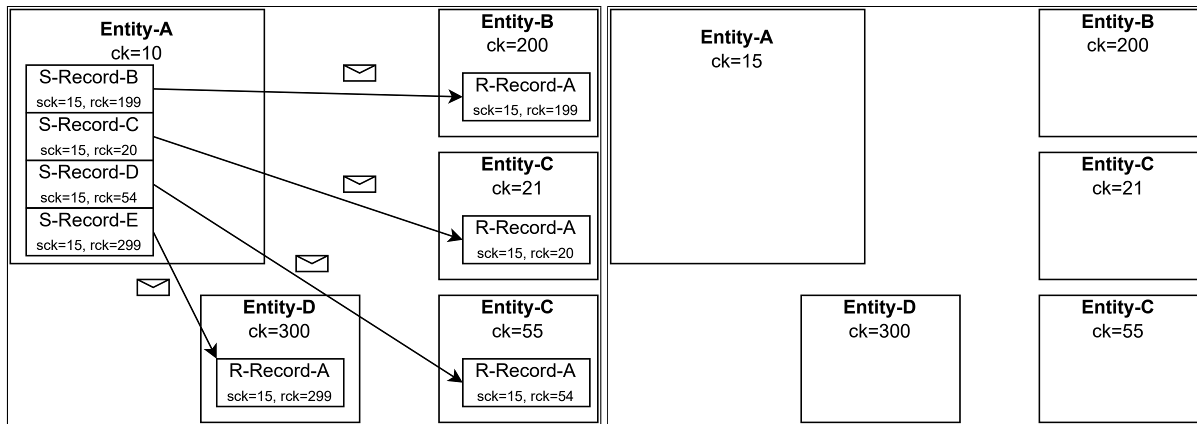
7.2.2 Activities: Managing State for Each Partner

Managing state information for each partner can become a significant operational challenge. This often arises when there is a necessity to retain partner-specific data, as depicted in figure 39, that cannot be efficiently merged into a common state. A practical example of this is the storage of a distinct flow for every neighbor node, a common requirement in protocols like Flow Updating [80].

For instance, as illustrated in Figure 40a, we can envision a scenario where entities utilize Exon to exchange information with other entities. In this scenario, "Entity-A" engages in sending messages to entities "Entity-B", "Entity-C", "Entity-D" and "Entity-E". Notably, for each entity, there exists an s-connection (S-record), and on the other side, each of these entities maintains an s-connection (R-record) specifically for interactions with "Entity-A".

Moving forward to Figure 40b, after the interaction done between entities, entities remove the unused slots, ensuring that resources are optimized. Consequently, the only state information retained on the entities is the integer "ck".

The distinction between using classical messaging protocols or using Exon with Pat Helland's model, lies in the fact that, when using Exon, after the interaction done between entities, it leaves only an integer behind, whereas using an at-least-once messaging protocol and application activities, a similar outcome is possible if the activities are designed to be cleaned up by garbage collection once the interaction between



(a) Exchange between Entities, keeping s-connection records

(b) Entities states after a specific time of inactivity, keeping only one integer "ck"

Figure 40: Entities using Exon

entities/partners concludes. However, using the classical method is challenging and would an ad hoc garbage collection approach for each application. Exon, on the other hand, aims to eliminate this need by transparently handling library-per-partner state garbage collection within the library, diminishing the need for explicit application level activities.

This becomes particularly crucial in the context of resource-constrained environments, where the scalability of systems is often limited. The traditional approach of preserving connection-specific information for each node becomes increasingly impractical and resource-intensive in such domains. Exon's ability to alleviate this burden is especially valuable, enabling efficient communication management even in resource-constrained devices.

7.2.3 Ensuring At-Most-Once Acceptance via Activities

In the realm of communication systems, dealing with messages that lack natural idempotence necessitates the assurance of processing each message at-most-once. This requirement often involves the implementation of mechanisms that maintain a unique characteristic of the message, ensuring it is not mistakenly processed multiple times. Notably, Exon ensures exactly-once, providing a robust solution to this challenge.

7.2.4 Conclusion

Helland's proposal of at-least-once messaging semantics, coupled with the obligation to design idempotency into application logic, highlights the trade-off between reliability and application complexity. However, Exon offers a transformative solution, it achieves "infinite scaling" with providing exactly-once messaging without the need for designing idempotent messaging at the application level. This significantly simplifies

application development, liberating developers from the constraints of idempotency design.

7.3 Enhancing Online Booking Distributed Systems with Exon

7.3.1 Introduction

Online booking distributed systems, whether for flights or hotel rooms, play a pivotal role in the travel and hospitality industry. These systems facilitate real-time bookings, availability checks, and reservations through a network of brokers, agents, and service providers. However, coordination between various parts of these systems can be complex and challenging, often leading to issues such as duplicate bookings, data inconsistencies, and inefficient communication. These challenges not only disrupt the operational efficiency of the system but can also have significant implications for the business model, potentially resulting in financial losses and liabilities.

This section explores the challenges associated with messaging in online booking distributed systems, and how what we call a *state partitioned system* approach using Exon, can address these challenges effectively.

7.3.2 Challenges in Online Booking Distributed Systems

Online booking distributed systems, encompassing reservations for flights, hotel rooms, and more, are the lifeline of the travel and hospitality industry. These systems provide customers with the convenience of making bookings in real-time, but behind the scenes, they face many complex challenges.

Here are the key challenges that these systems face:

- Distributed Architecture: Online booking systems are distributed by nature, involving multiple brokers, agents, and data sources. Coordinating communication and ensuring data consistency across these distributed components can be a significant challenge.

- Message Reliability: Booking systems demand reliable message delivery to prevent issues like double bookings or reservation conflicts. Ensuring that messages are processed exactly once is essential to maintaining system integrity.

- Scalability: As the number of bookings and users increases, scalability becomes a critical concern. Systems must scale to accommodate a high number of requests without compromising performance.

- Fault Tolerance: To maintain uninterrupted service, online booking systems must be resilient to network failures, including message loss/duplication, and network partitions.

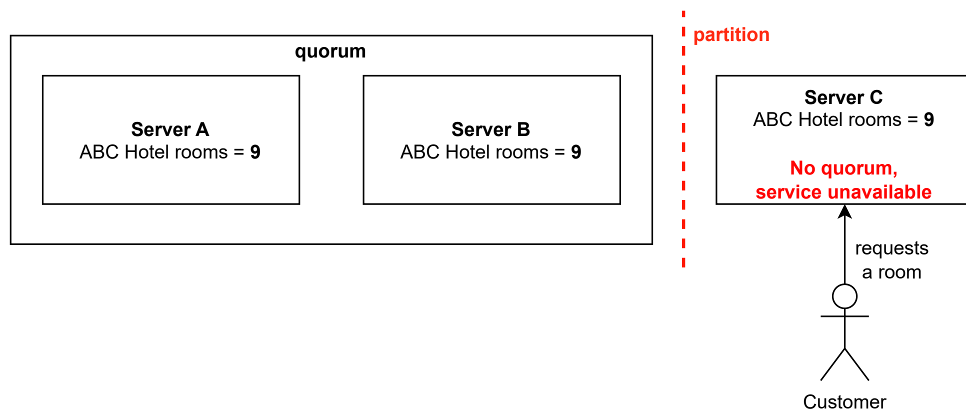


Figure 41: State-machine replication using consensus

7.3.3 Partitioned State Systems

In the realm of online booking distributed systems, where numerous agents and nodes interact, we propose a solution to tackle the complex challenges that often burden such systems. Our proposal centers around the concept of "partitioned state systems", which introduces an approach to managing distributed state and optimizing message coordination.

State-Machine Replication, is an approach, where state is replicated among all nodes contributing in the system. The main problem with this approach is that when a partition happens between servers, the service will become unavailable on the side where the servers do not form a quorum, as depicted in figure 41.
However, rather than replicating identical state information across all servers, where the cost implications can be substantial, especially in the presence of network partitions between servers, Partitioned State Systems is the solution, where the state partitioned among the servers involved. Each server independently maintains its own self-reliant state, as depicted in figure 42. However, within this partitioned state, the quantities are associated with values that can be updated locally. Partitioning can achieve availability by guaranteeing that in the event of one partition's failure, the remaining partitions can still respond to local requests [81].

This approach becomes especially valuable in the presence of a large number of nodes within a scalable distributed system, as it substantially reduces coordination overhead and associated costs. Additionally, this partitioned state system allows for efficient and instant booking confirmations, as each server directly manages and controls its allocated rooms, eliminating the necessity for coordination with other servers, and significantly increases the availability of the system.

However, in the partitioned state system, servers have the ability to request rooms from each other once they have finished with their own rooms. Towards this, having an exactly-once message delivery protocol

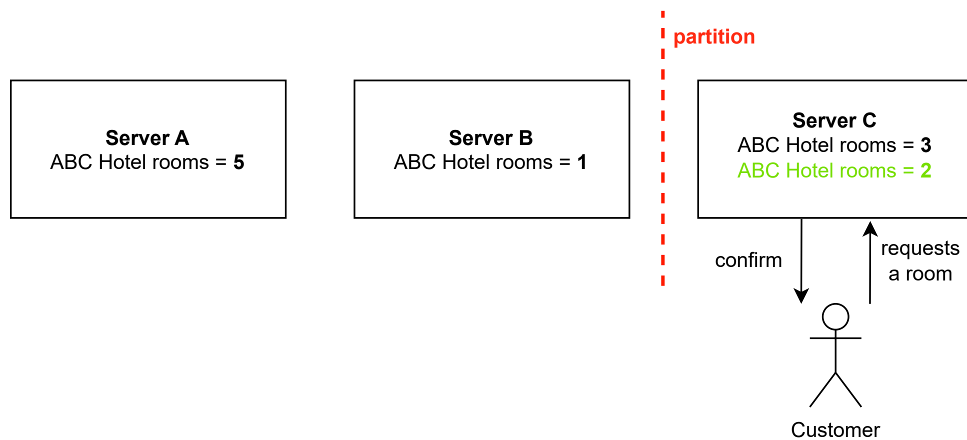


Figure 42: Partitioned state system

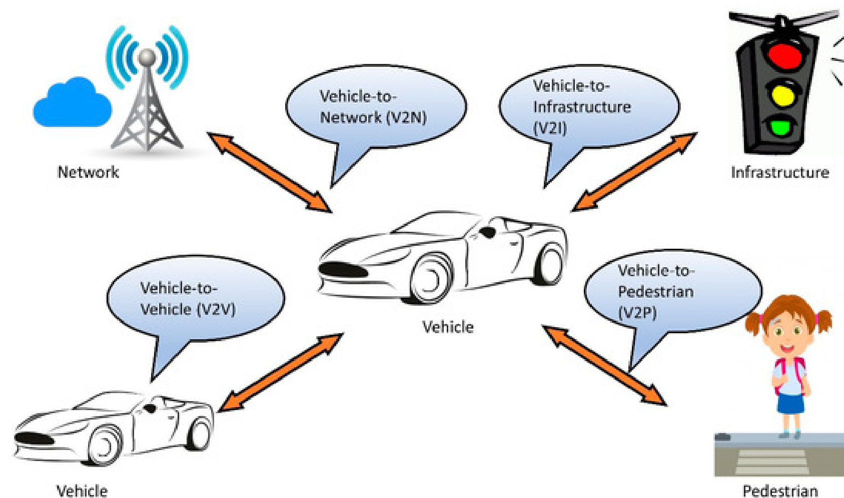


Figure 43: V2X Domain. Source: [83]

like Exon is essential to be able to get away with not using distributed transactions/consensus.

7.4 Automotive Domain

The V2X (Vehicle-to-Everything) automotive domain represents a cutting-edge and transformative realm within the automotive industry. As vehicles become increasingly connected and autonomous, the V2X domain consists of a system where vehicles interact not only with each other (V2V) but also with different entities including infrastructure (V2I), pedestrians (V2P), cyclists, and even the broader transportation network (V2N).
V2X communication can establish communication in one of two ways [82]: (i) through direct connections, such as using 802.11p-based technologies or LTE PC5/Sidelink interface, or (ii) by utilizing the LTE interface, which supports both uplink and downlink communication.

V2X encompasses various communication scenarios as shown in figure 43, each contributing to enhanced

safety and efficiency. V2V communication enables vehicles to broadcast their positions, speeds, and intentions, enabling nearby vehicles to expect movements and mitigate collisions. V2I communication allows vehicles to communicate with traffic lights, road signs, and infrastructure elements, road side units (RSU), enabling optimized traffic signal timing and smoother navigation. V2P communication enables vehicles to detect and respond to pedestrians, enhancing pedestrian safety.
Moreover, V2X plays a crucial role in advancing the development and deployment of autonomous vehicles, as real-time communication among vehicles and their surroundings is essential for safe and coordinated self-driving operations.

7.4.1 Challenges

In the V2X (Vehicle-to-Everything) domain, which encompasses communication among vehicles and various entities to enhance road safety and efficiency, several challenges related to node communication exist. These challenges include intermittent connectivity [84], often caused by factors like signal blockage, high mobility, battery draining, and network congestion. The dynamic network topology [85], which rapidly changes as vehicles and entities move in and out of range, makes it challenging to establish and sustain communication links, potentially leading to data exchange disruptions. Additionally, managing communication between a growing number of V2X nodes is complex, necessitating scalable communication infrastructure. In safety-critical scenarios, ensuring reliability and redundancy in communication mechanisms is crucial to mitigate severe consequences of communication failures [86].
Addressing these challenges requires innovative solutions in communication protocols. Exon, emerges as a promising approach to tackle these challenges and establish a robust V2X ecosystem.

7.4.2 Addressing Challenges using Exon

7.4.2.1 Forwarding Messages

In the realm of the Automotive domain, vehicles act as mobile nodes, and the status of a mobile node can transition suddenly between reachable and non-reachable states in relation to other nodes.
While messages between nodes in V2X communication often follow a broadcasting approach and may not necessarily insist on exactly-once message delivery, there are specific scenarios where message reliability becomes essential.

The platooning concept entails a group of vehicles traveling together in a coordinated formation, illustrated in Figure 44. Anticipated benefits of platooning encompass enhanced fuel and traffic efficiency, as well as safety and driver comfort [88].
Implementing vehicle platooning in intelligent transportation systems involves equipping vehicles with communication hardware and algorithms to enable coordination between vehicles. Ensuring the platoon's safety necessitates the reliability of V2V communication, such as verifying that all trailing vehicles

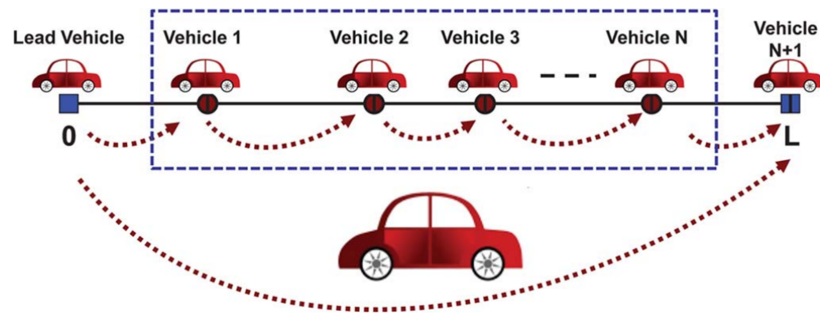


Figure 44: Vehicles forming a platoon. Source: [87]

maintain consistent contact with the lead vehicle and receive identical information [87].

To enhance message reliability, the integration of Exon is a viable solution. Additionally, in this scenario, vehicles must relay messages reliably from one vehicle to another, and the Exon extended forwarding feature can effectively address this need.

7.4.2.2 Obliviousness

In terms of achieving message reliability, preserving connection-specific information for future communication has been normally the practice. This avoids message duplication, particularly in scenarios where vehicles may not encounter each other for extended periods, possibly spanning months or years. Here, Exon's obliviousness feature becomes invaluable, providing a solution to this challenge by allowing for the efficient management of connection-specific data without compromising scalability.
Furthermore, in the context of scalability, Exon's design, coupled with its inherent oblivious feature, makes it an ideal choice for scalable applications, unlike approaches where nodes retain connection-specific information about all previously contacted nodes. It maintains only a single integer as permanent state per node when no connections are active, significantly reducing the overhead associated with maintaining extensive connection records.

7.4.3 A Potential Problem with Exon

In the context of Exon's communication model, a potential concern arises. In scenarios where two vehicles have a brief encounter and subsequently may never cross paths again, any undelivered messages or unused communication slots would lead to an accumulation of state. In the case where there are not anymore pending messages, Exon needs some time window to achieve oblivousness, which may not happen if the communication window is too brief.
However, a potential solution to this challenge involves incorporating periodic use of "global internet access", such as LTE connectivity. Vehicles can intermittently access this global network, allowing them

to establish connections with permanent nodes or other relevant vehicles that also utilize global access. During these intermittent connections, the accumulated garbage can be effectively garbage collected.

7.5 Messaging Support for Distributed Middleware

In the realm of distributed middleware, messaging support plays a pivotal role in enabling efficient and reliable communication. In this section, we explore two aspects of messaging support within the context of distributed middleware, specifically focusing on how Exon can enhance existing systems and pave the way for novel approaches.

7.5.1 Distributed Actor Systems

Distributed actor systems (e.g. Erlang [89], akka [90]) are a natural choice for constructing online applications characterized by a multitude of dynamically interacting entities, such as social networks, online games, and Internet of Things (IoT) applications. In modern actor systems, the design is simplified by mapping application objects onto lightweight actors which encapsulate object state and logic, and dynamically interact via asynchronous messages. For instance, within an online chat service, each user and chat room can be conceptualized as individual actors. In Distributed, Erlang a TCP connection is established between each pair of VMs, multiplexed by all communication between them.

Within this domain, certain challenges arise, particularly concerning TCP stream multiplexing and the possibility of Head-of-Line (HoL) blocking. Exon presents a solution for implementing distributed actor systems, such as Distributed Erlang, with a focus on mitigating these challenges.
In chapter 5, we showed how TCP stream multiplexing that leads to Head-of-Line (HoL) blocking can impacting the overall performance of the TCP protocol. Exon can be used between VMs, that allow designing more fine grained ordering constraints (e.g. FIFO per actor) on top of Exon, which itself does not impose ordering.

7.5.2 Brokerless Messaging Middleware

One notable category of distributed middleware involves brokerless messaging systems, exemplified by technologies like ZeroMQ [62] and Nanomsg/NNG [91]. Indeed, both ZeroMQ and Nanomsg, which use TCP as the transport layer protocol, do their best to recover from TCP reconnections when communication is temporarily disrupted. However, one notable challenge with these systems is the potential loss of queued messages upon disconnection. In other words, the challenge arises when the connection is disconnected. While the messaging systems can often reestablish the connection, any messages that were queued but not yet delivered at the time of disconnection may be lost. This is because TCP itself does not provide any guarantees about the delivery of queued data upon reconnection, either the data that was lost due to

TCP abnormal disconnection, or the messages in the middleware queues (e.g., ZMQ) that have not been handed to TCP, but will be discarded by the middleware upon disconnection.
In contrast, Exon emerges as an alternative to TCP, offering a solution that ensures message reliability even when under network intermittent connectivity. A new design can be achieved, that combines the strengths of ZeroMQ messaging patterns and flexibility with Exon's robustness in ensuring message reliability. The synergy between ZeroMQ and Exon can provide a powerful and dependable foundation for distributed applications, offering both the messaging patterns and the reliability necessary for various communication scenarios.

7.5.3 Broker-based Messaging Middleware

Broker-based messaging middlewares are a type of communication infrastructure used in computer software and distributed systems to facilitate the exchange of data and messages between different components or applications. These systems employ a central intermediary, known as a broker, that manages the routing, distribution, and coordination of messages between producers and consumers. It is important to note that "broker-based" does not inherently imply heaviness or persistence (e.g. IoT implementations using the MQTT protocol). In the world of IoT, broker-based systems like MQTT [67] have gained widespread adoption due to their efficiency and scalability.

Recent developments in this domain demonstrate the drive to replace traditional transport mechanisms like TLS+TCP with more advanced alternatives such as QUIC, as exemplified in the case of MQTT over QUIC [33]. This shift underscores the industry's recognition of the need for improved transport protocols to enhance the performance of broker-based systems.
Implementing MQTT over Exon becomes a realistic. The potential to replace TCP with Exon as a transport layer introduces opportunities for enhancing the reliability and efficiency of broker-based systems, particularly in scenarios like IoT, where message delivery and latency are critical factors.

This evolving trend of replacing TCP with superior transport mechanisms underscores the proposition that we can indeed surpass the limitations of TCP, and Exon emerges as a viable candidate for fulfilling this need. This shift toward advanced transport protocols demonstrates the relevance of Exon as a potential replacement and enhancement for existing communication frameworks. The implementation of MQTT over Exon can be referred to as a concrete topic to be done in the future.

7.6 Conclusion

The evolving landscape of modern communication systems continually demands scalability, efficiency, and reliability. In this chapter, we have explored a spectrum of applications, including Distributed Aggregation, Pat Helland's innovative application structuring approach, enhancements in Online Booking Distributed

Systems, insights into the Automotive Domain, and the pivotal role of Messaging Support for Distributed Middleware. Throughout our exploration, we showed how Exon seamlessly integrates into and enhances these diverse applications.

8 Case study: Distributed Aggregation

Aggregation protocols allow for distributed lightweight computations deployed on ad-hoc networks in a peer-to-peer fashion. Due to reliance on wireless technology, the communication medium is often hostile which makes such protocols susceptible to correctness and performance issues. In this chapter, we study the behavior of aggregation protocols when subject to communication failures: message loss, duplication, and network partitions. We show that resolving communication failures at the messaging layer, through a reliable messaging layer, reduces the overhead of using alternative fault tolerance techniques at upper layers, and also preserves the original accuracy and simplicity of protocols. The empirical study we drive shows that tradeoffs exist across various aggregation protocols, and there is no one-size-fits-all protocol.

This work was carried out in a early stage of the Exon protocol implementation. We utilized an initial primitive implementation of Exon, we refer to it as "Basic-Exon", that incorporates the fundamental concepts of Slots and Tokens, lacking some of the optimizations such as the multithreading, requesting Slots in advance, closing connection, etc.

The study focused on understanding how aggregation protocols perform when subjected to various communication issues such as message loss, duplication, and network partitions.
One key finding of this research was that resolving communication failures at the messaging layer, through the implementation of a reliable messaging layer, can significantly reduce the overhead associated with alternative fault tolerance techniques at higher layers. Additionally, this approach preserves the original accuracy and simplicity of the protocols under study.
It is worth emphasizing that this research laid the groundwork for further advancements in the field, including the development of the Exon protocol as it is now. The insights gained from studying communication failures and their impact on aggregation protocols have contributed to the ongoing efforts to design more efficient and robust protocols for distributed lightweight computations in ad-hoc networks.

8.1 Introduction

With the prominence of the Internet of Things (IoT), Mobile Computing (MC), and Wireless Sensor Networks (WSN), data aggregation is becoming a de-facto technique to manipulate distributed data, control the network, and perform lightweight computations (like max, sum, count...) [76–78]. In such systems, resource constrained devices are usually deployed over ad-hoc networks, mainly using wireless communication media (WIFI, Blue-tooth, Zig-bee [92]), in a Peer-to-Peer (P2P) fashion to avoid the usual bottlenecks of centralized solutions and communication failures. Consequently, the underlying messaging layer is often hostile and can compromise the correctness and performance of aggregation protocols; therefore, fault tolerance techniques are still being advocated to make these protocols more reliable and practical [76, 77, 80, 93, 94].

Specifically, convergence — to a correct common value across nodes — is considered the prime challenging correctness measure of aggregation protocols since it can easily be adapted to abstract other computations (like sum, count, max, etc.) [78, 80]. Convergence becomes more challenging when subject to environmental impacting factors like the number of nodes in the system, the topology of the network, the reliability of communication medium, the available resources of devices, etc. This leads to several trade-offs between simplicity, correctness, and performance. Among these protocols, gossip-based aggregation protocols are often considered more robust under such factors [95, 96].

In particular, there are two categories of gossip-based aggregation protocols in literature. The first, is simple, in which an (mutable) estimate of the average value is computed locally and then propagated to other nodes either in a completely distributed way, e.g., Push-Sum [78], or in a clustered way as in DRG [77]. This simplicity however comes at the price of dedicated fault tolerance techniques, required to handle message loss and duplication, whose solutions are sometimes costly [79].
Another category, like Flow-Updating (FU) [80], is immune to message loss and duplication by nature due to the concept of flows: idempotent averaging estimates are locally computed based on average flows received from other nodes in an immutable way. As the authors show in [80, 97], FU is not prone to transient communication failures, but as showed later, it exhibits some instability under long-lived communication failure and when the average degree (i.e., number of neighbors of a node) is high.

To take advantage of idempotency, several "hybrid" protocols [98, 99] tried to integrate the idea of flows and node backups with the Push-Sum protocol in an attempt to introduce a simple and fault tolerant aggregation protocol; unfortunately, this yielded other accuracy and performance issues [96]: (1) performance was significantly impacted by communication issues, and (2) some accuracy in convergence is detected.

In this chapter, an empirical study is presented that compares the above variants of gossip-based protocols leading to the following conclusion: once simple classical protocols, like Push-Sum [78], are supported

by a robust exactly-once underlying messaging layer, Basic-Exon in our case, they can outperform other protocols [80, 98, 99], and importantly, maintain the original convergence accuracy. A simulation is done to the above classes of protocols once subject to message loss, duplication, and network partitions.

8.2 Background and Related Works

In this section, a concise background is presented on three variants of aggregation protocols: Push-Sum (PS), Flow Update (FU), and Distributed Random Grouping (DRG). Despite the diverse data aggregation protocols introduced in literature [77, 98–100], we opt for the aforementioned protocols being well-known and most other protocols are variants using their core concepts.

Push-sum protocol [78] is a simple gossip-based protocol where each node divides its local value by half and propagates it to other peers until convergence is achieved. PS protocol converges faster when degree increases, however the algorithm correctness relies on "mass" conservation [78] where any kind of system failure violates mass conservation. Consequently, several variants like [79, 98, 99] were introduced to withstand network and node failures, which resulted in accuracy and performance overheads [96].

Flow-Updating (FU) [80], is based on the concept of idempotent computation through "flows". The idea is that each node calculates the average based on all the contributions of the in/out flows along the edges of the neighbors and its initial value. Since this depends on the direct flows (and the initial intact value), there is no need to retain corresponding mutable variables. This makes the algorithm natively tolerant to message loss and duplication, but suffers some instability period upon recovery from network partitions (as flows are missing).

LiMoSense [79], Push-Flow [98] and Push-Cancel-Flow [99] followed a hybrid model by using concepts from PS and FU through using flows for data exchange and mutable local histories to compute the estimates. However, these protocols induced correctness problems as accuracy and division by zero [95, 99].

DRG [77] is an aggregation protocol that essentially consists in the continuous random creation of groups across the network, in which messages are broadcast and aggregates are successively computed (averaged). In DRG, message loss between coordinators and neighbors may happen; and thus, partial fixes to avoid deadlock of nodes waiting forever may result in violating mass conservation [80].

8.3 The Technique Adopted

In this chapter, a technique inspired from the paper titled "Exactly-once quantity transfer" [101]. The foundation of this paper's concept is rooted in the principles of Handoff Counters [102], wherein the authors

engineered scalable eventually consistent counter CRDTs. These CRDTs are designed to operate accurately even in the presence of network partitions while mitigating identity explosion issues encountered in previous CRDTs such as G-Counters [103]. However, the paper [102] takes this concept a step further by extending it to encompass quantity transfers in any "splittable" datatype and expanding its applications.

To investigate the behavior of aggregation protocols, a locally developed simulator was utilized. The simulator was modified to integrate the concept of slots and tokens, aligning with the aforementioned technique. Before sending a regular message, the simulator was adapted to initiate a message requesting slots. Upon receiving a response and reserving a slot at the receiver, the sender could then proceed to send the actual messages.

As I said, we used a primitive version of Exon (Basic-Exon), and employed it within the simulator. The research aimed to evaluate the benefits of using an exactly-once message technique in the context of aggregation protocols. These findings contributed to the subsequent development of the Exon protocol, which further refined and expanded upon these concepts to address a wider range of scenarios and requirements.

8.4 Evaluation

8.4.1 Experimental Setup

To perform the experiments, the simulator is used that runs on a single machine, where message loss/duplication, network partition, network topology, number of nodes and the degree can be customized to serve my purpose. The experiments considered four network topologies with random generation of links: Bus, Ring, 2D Mesh, and random graph of several dimensions, i.e., different degrees (from 3 to 20). The usage of four network topologies aim to comprehensively assess the performance and adaptability of the Basic-Exon protocol across a spectrum of real-world network scenarios. However, I present those of random graph since it allows for a wide range of degrees, and more realistically represents wireless settings.

As mentioned earlier, we opted for the protocols PS, FU and DRG protocols being well-known and represent the state-of-the-art of gossip-based aggregation protocols. FTPS and FTDRG refer to the fault-tolerant versions of PS and FU where they used over the Basic-Exon protocol.

Finally, the protocols are evaluated through two main metrics: accuracy and speed, considering message loss and duplication under different graph degrees. Accuracy is expressed by the normalized Root Mean Square Error (RMSE) of the estimate in contrast to the target value, and the speed expressed by the number of iterations to reach this accuracy. An estimation of the messaging overhead with and without Basic-Exon is also presented at the end.

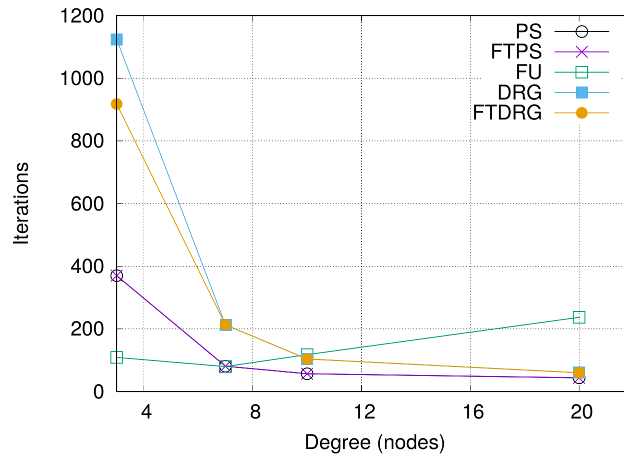


Figure 45: Convergence speed in fault-free scenario.

For the sake of completeness, I tried to implement and experiment the Push-Flow protocol, and I noticed that the accuracy is significantly lost with 1000 nodes. Indeed, this result is consistent with those in [96] that shows the accuracy problem starting with 60 nodes up.

8.4.2 Convergence speed in a fault-free network

Convergence speed is experimented as the degree increases from 3 to 20, using random graph of 1000 nodes. I present the number of iterations required by each protocol to achieve convergence in a fault-free scenario, where convergence is considered achieved once the $RMSE = 10^{-11}$.

Figure 45 interestingly shows that the convergence speed of FU improves until reaching degree 7, beyond which more iterations are needed to converge. The results look surprising for the first glance, however, they are consistent with the results in the original paper of FU [80] for degree 10. This behavior is referred to the direct dependence of FU on the in/out flows of direct neighbors in calculating the $estimate$ of the average. As the degree increases, the number of flows per node increases, thus significantly modifying the $estimate$. Indeed, this conforms with the analysis in [80] that shows FU under link failures converges faster than healthy network as the number of links drops.

To the contrary, PS and DRG converge much faster (log-scale is shown) with higher degrees and they significantly outperform FU starting at degree 7 and 12, respectively. (Notice that PS and DRG are confounded with FTPS and FTDRG, respectively, given that no faults occur.) This behavior is expected as the $estimate$ in DRG and PS is computed and spread to neighboring nodes, thus as the network is more connected, the information propagates faster.

Finally, the figure (Figure 45) shows that PS converges faster than DRG, especially, when the degree is low

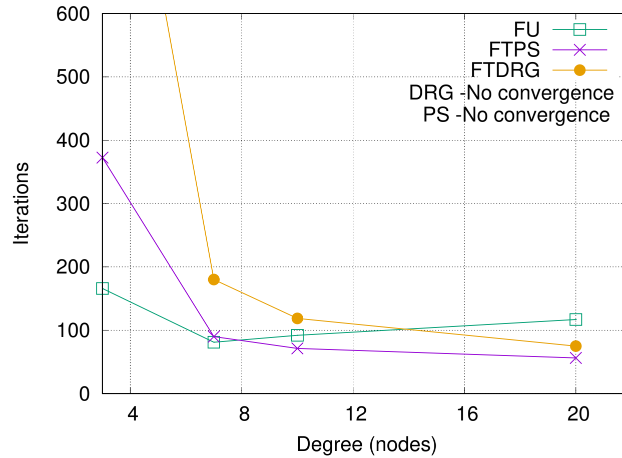


Figure 46: Estimated convergence speed under 20% message loss.

since DRG cannot create large groups in this case, and this broadcast to all system nodes takes longer, contrary to large groups (with large degree).

8.4.3 Convergence speed under loss and duplication

Under message loss, the behavior changes as shown in Fig 46. I performed experiments using different percentages of message loss, i.e., from 10% to 50%, and noticed that the patterns are very similar, and thus I only convey the 20% results in Fig. 46.
The convergence speed of FU under message loss remains equivalent as in the fault-free scenario (Fig. 45), or even better, which is also demonstrated in [80], where the number of flows is small and the computation of the estimate depends directly on it. Thus, no need to integrate the reliable messaging layer with FU. In contrast, PS and DRG cannot converge at all due to violating mass conservation.

Using Basic-Exon, the fault tolerant messaging layer, the variants of PS and DRG, i.e., FTPS and FTDRG, are again able to operate, however at an additional communication overhead. Since the messaging layer is transparent to the protocols' logic, this overhead does not manifest on the number of iterations needed. Counting an additional iteration per retransmitted message is unfair, since not the entire system is actually delayed, whereas discarding the retransmission time and overhead is biased to PS.

To handle this, I estimated the number of extra iterations needed through dividing the extra messages propagated due to failures, by the *average number of messages exchanged per iteration* in the fault-free case. This estimation turned out to be a polynomial equation of the form: $f(x) = \Sigma_{i=1}^{4} x^i$; where x is the retransmission ratio. In the case of 20%, $f(0.2) \approx 0.25$. Using this estimation, the convergence speed of FTPS and FTDRG is roughly 25% lower than that of PS and DRG in the fault-free case (Fig. 45), respectively. Therefore, FTPS only overtakes FU starting from degree 10 (instead of 7 in the fault-free case). The overhead is higher for FTDRG which overtakes FU starting from degree 16 and up, and thus I

believe it is not worth using DRG instead of FU in this case.

As for message duplication, FU and DRG are not affected since computing the aggregate is idempotent; and therefore, the convergence speed remains as shown in Fig. 45. To the contrary, PS suffers from message duplication and thus cannot converge at all — thus I do not plot the corresponding curve. The use of the fault tolerant messaging layer in FTPS overcomes the duplication problems as expected without significant changes in convergence speed to this presented in Fig 45.

Note that, though not manifested in the experiments, the overhead of Basic-Exon in this case will be very low [101] as compared to classical deduplication methods that depend on retaining logs — which is not tolerable in resource constrained devices.

8.4.4 The impact of network partitions

To experiment the protocols under network partitions, I used a random network of 1000 nodes, and manually identified some links that when broken can lead to network partition. Following this process, I have studied the time needed by the three protocols PS, FU, and DRG to converge under different settings: changing the partition size (to 10%, 25%, and 50%) and time of partitioning occurs (i.e., 20-200 iterations and 200-500 iterations) with degree $\approx$ 10. In particular, I studied the time needed to converge to a very small RMSE.

I only convey the important part of my results in Figure 47 which correspond to 10% partition size and partition time 20-200 iterations. This graph is chosen due to the following reasons. Changing the partition size to 25% and 50% lead to very similar patterns to those in Figure 47 with a slight difference that the impact of partitioning is a bit lower on the two partitions. This reason is referred to the fact that initial values are quickly propagated to most of the nodes in the two partitions, leading to faster averaging before partitioning occurs. This very reason lead me to omit the experiments of partitioning time 200-500 iterations. Indeed, the three protocols reached an almost stable state after 200 iterations which absorbs the impact of partitioning.

Considering Figure 47, the first observation is that, at the instant partitioning starts (i.e., 20 iterations), the two partitions continue to converge through tens of iterations before stabilizing almost at 100 iterations in all protocols. This was expected as the three protocols are completely decentralized and can operate as long as peers are reachable. The second observation shows that FU shows some fluctuation when the network heals from partition as shown by the spike at the iteration 200 in Figure 47.

To further understand this behavior, I tried to compute the value that each partition converged to before and after healing. My experiments show that FU tried to converge to different values on each partition

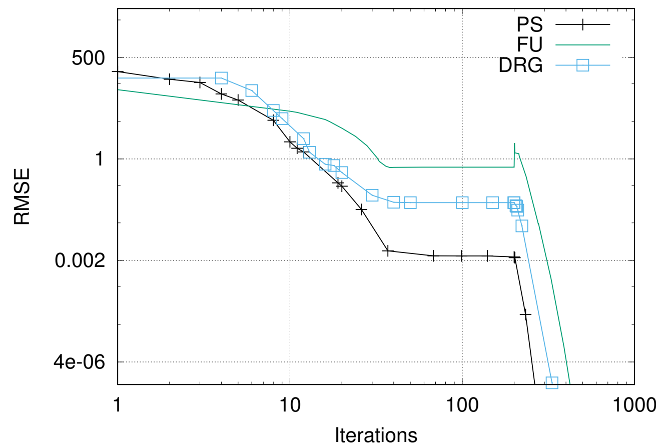


Figure 47: Root Mean Square Error under 10% network partition size and degree 10.

Table 2: Estimated message's header size of FU, DRG, PS, and two fault tolerant piggy-backing patterns of PS.

FU	DRG	PS	FTPSPB1	FTPSPB2
76	36 (max)	74	32	92

which is far from the optimal average. To the contrary, the values of PS and DRG keep getting closer to the optimal average, though slowly. This is referred to the high impact of using immutable flows in FU, which is consistent with the results in [99], and which is considered the major drawback in flow based protocols.

8.4.5 The messaging overhead

As for message's header size, Table 2 presents the average size of the three protocols FU, PS, and DRG: 76, 74, and 36, respectively. Despite this slight difference among protocols, it has no impact in a real network. In fact, given that the payload in aggregation applications is often small (e.g., few Bytes), the header and the payload can both fit in a single UDP datagram. (TCP is not recommended in such hostile settings.) To tolerate message loss and duplication, FTPS should also be considered. According to the Basic-Exon I used with PS, FTPS requires four different message types, in two round-trip delays, to deliver a single aggregation message and a corresponding ACK. The size of each message header ranges between 16 and 74 Bytes. However, under congestion, messages can be piggy-backed in two patterns depicted as FTPSPB1 and FTPSPB2 in Table 2. Therefore, the messaging overhead of the Basic-Exon layer is negligible as well.

8.5 Conclusion

Complementary to state-of-the-art studies on aggregation protocols, the study focused on taking failure prone protocols as PS and DRG, and integrate a reliable messaging layer that can preserve the simplicity of the classical protocols to overcome mass conservation risks.

By incorporating Basic-Exon into the network simulator, the research aimed to assess the effectiveness of ensuring reliability in order to overcome the limitations and vulnerabilities of failure-prone aggregation protocols. This approach allowed for an exploration of the trade-offs between reliability and protocol simplicity, providing insights into the potential improvements that can be made to enhance the performance of aggregation protocols in the presence of communication failures.

In conclusion, the Flow-Updating protocol [80] is natively robust to loss and duplication but not to network partitions which incur some temporary perturbation in the values. Since network partitions are more likely to occur once the degree is small (e.g., 3), it is recommended to avoid using FU unless the perturbation in the values is tolerable by the application; the alternative is to use other Flow-Updating variants like [98, 99] only if high accuracy is not a matter.

On the other hand, PS and DRG are prone to communication issues, and thus using a reliable messaging layer is crucial for mass conservation. The experiments conveyed show that providing a reliable messaging layer comes at a cost, which is not worth it in the case of DRG. To the contrary, PS protocol is more accurate and outperforms the other protocols when the $degree > 10$ despite the overhead of using Basic-Exon.

9 Conclusions

In conclusion, this thesis has made significant contributions to the field of distributed messaging and communication protocols. A novel protocol, named Exon, has been introduced, offering a robust solution for achieving oblivious exactly-once messaging. Its safety and liveness properties have been formally proven, providing a solid foundation for reliable message delivery. The development of the Exon-lib open-source library, implementing Exon over UDP, further enhances the practical applicability of the protocol by providing a flexible API for building distributed applications.

Empirical evaluations conducted across diverse network scenarios have shed light on the performance and effectiveness of the proposed solutions, offering valuable insights into their real-world viability. The performance evaluation showed Exon to have similar performance to TCP in normal conditions but considerably better performance under message loss.
The presented extension to Exon, enabling message delegation through intermediary nodes, contributes to the adaptability of the protocol suite.
Integrating Exon into various applications reduces overhead in fault tolerance, benefiting data aggregation, Pat Helland's Vision, Online Booking Distributed Systems, the Automotive Domain, and Messaging Support for Distributed Middleware.

These contributions collectively advance the understanding and capabilities of distributed messaging protocols, offering innovative solutions to challenges in achieving reliable and efficient communication in various network contexts. The theoretical foundations, practical implementations, and empirical insights presented in this thesis pave the way for further research and applications in the realm of distributed systems and networking.

9.1 Limitations

While Exon demonstrates several strengths, there are a few limitations worth noting:

- Exon's current prototype performs similarly to TCP under normal fault-free conditions and exhibits improved performance under small packet loss rates. However, we have identified a slight performance degradation under low bandwidth and low network latency. To address this limitation, we have devised a solution that will be implemented in the next library version.
- The evaluation of Exon's performance and robustness has been conducted in controlled environments, and further real-world testing is necessary to assess its behavior in diverse network conditions and under various types of workloads.
- Security considerations for Exon are needed to conduct a comprehensive security analysis, including vulnerability assessments and threat modeling. The focus will be on developing and implementing robust security mechanisms to enhance the secure messaging capabilities of Exon.

9.2 Future Research Directions

Based on the findings and limitations identified in this thesis, several directions for future research can be explored:
Further performance optimization: Investigate additional techniques to enhance Exon's performance under low bandwidth and low network latency conditions, e.g. message piggybacking, ensuring that it remains efficient and robust across a wide range of network scenarios.
Real-world deployment and evaluation: Conduct extensive real-world testing and evaluation of Exon, considering various network topologies, traffic patterns, and workload distributions. This will provide a more comprehensive understanding of its behavior, performance, and scalability.
Security analysis and enhancements: Conduct a thorough security analysis of Exon, including vulnerability assessments, threat modeling, and formal verification. Based on the findings, develop and implement additional security measures to ensure the protocol's resilience against potential attacks. There are several approaches can be considered to ensure the security of the middleware. These approaches could be whether through underlying layers, off-the-shelf solutions, or custom implementations, that will help safeguard the confidentiality, integrity, and availability of Exon protocol.
Extended use cases and applications: Explore additional use cases and applications where Exon can provide significant benefits, such as distributed computing, Internet of Things (IoT) systems, and blockchain networks. Investigate the integration of Exon with existing frameworks and protocols to leverage its advantages in various domains.
By addressing these research directions, we can further advance the understanding and applicability of Exon in practical distributed systems, contributing to the broader field of network protocols and communication technologies.

In conclusion, Exon offers a promising solution for achieving efficient and reliable message-based communication in distributed systems. Through its lightweight design, exactly-once delivery guarantees, and

support for partition tolerance through delegation, Exon demonstrates its potential as a robust transport-level protocol. The contributions of this thesis, including the development of Exon, and the evaluation of its performance, lay the foundation for future research and advancements in this area. By addressing the identified limitations and pursuing future research directions, Exon can further mature as a practical and effective protocol for a wide range of distributed applications, contributing to the advancement of network protocols and communication technologies.

As I am concluding this stage and bringing this dissertation to a close, I am filled with the sense that numerous new research opportunities are emerging. This journey has taught me that every ending is also a beginning, and with each closed door, there are multiple others waiting to be opened.

Bibliography

[1] João M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. url: `https://github.com/joaomlourenco/novathesis/raw/master/template.pdf` (cit. on p. i).

[2] Pat Helland. "Life beyond distributed transactions: an apostate's opinion". In: *Queue* 14.5 (2016), pp. 69–98 (cit. on pp. 1, 2, 6–8, 13, 41, 107).

[3] Eric A Brewer. "Towards robust distributed systems". In: *PODC*. Vol. 7. Portland, OR. 2000, pp. 343477–343502 (cit. on pp. 1, 10, 23).

[4] Ziad Kassam, Paulo Sérgio Almeida, and Ali Shoker. "Exon: An Oblivious Exactly-Once Messaging Protocol". In: *2022 International Conference on Computer Communications and Networks (ICCCN)*. IEEE. 2022, pp. 1–10 (cit. on pp. 3, 4).

[5] Ziad Kassam et al. "Aggregation protocols in light of reliable communication". In: *2017 IEEE 16th International Symposium on Network Computing and Applications (NCA)*. IEEE. 2017, pp. 1–4 (cit. on pp. 4, 5, 15).

[6] Ali Shoker et al. "Life Beyond Distributed Transactions on the Edge". In: *Proceedings of the 1st Workshop on Middleware for Edge Clouds & Cloudlets*. 2016, pp. 1–3 (cit. on pp. 5, 8).

[7] C. Carr, S. Crocker, and V. Cerf. "HOST-HOST communication protocol in the ARPA network". In: *AFIPS '70 (Spring)*. 1970 (cit. on p. 7).

[8] C. Sunshine and Yogen K. Dalal. "Connection Management in Transport Protocols". In: *Comput. Networks* 2 (1978), pp. 454–473 (cit. on p. 7).

[9] Vinton Cerf and Robert Kahn. "A protocol for packet network intercommunication". In: *IEEE Transactions on communications* 22.5 (1974), pp. 637–648 (cit. on p. 7).

[10] Jon Postel et al. *User datagram protocol*. 1980 (cit. on p. 7).

[11] Jon Postel. *Rfc0793: Transmission control protocol*. 1981 (cit. on pp. 7, 25).

[12] William Stallings. *Handbook of computer-communications standards; Vol. 1: the open systems interconnection (OSI) model and OSI-related standards*. Macmillan Publishing Co., Inc., 1987 (cit. on pp. 7, 23).

[13] Dag Belsnes. "Single-message communication". In: *IEEE Transactions on Communications* 24.2 (1976), pp. 190–194 (cit. on p. 7).

[14] Hagit Attiya, Shlomi Dolev, and Jennifer L Welch. "Connection management without retaining information". In: *Information and Computation* 123.2 (1995), pp. 155–171 (cit. on pp. 7, 39).

[15] Victor C Zandy and Barton P Miller. "Reliable network connections". In: *Proceedings of the 8th annual International Conference on Mobile Computing and Networking*. 2002, pp. 95–106 (cit. on pp. 8, 30).

[16] Alex C Snoeren, David G Andersen, and Hari Balakrishnan. "Fine-Grained Failover Using Connection Migration." In: *USITS*. Vol. 1. 2001, pp. 19–19 (cit. on pp. 8, 30).

[17] Richard Ekwall, Péter Urbán, and André Schiper. "Robust TCP connections for fault tolerant computing". In: *Ninth International Conference on Parallel and Distributed Systems, 2002. Proceedings*. IEEE. 2002, pp. 501–508 (cit. on pp. 8, 29, 30).

[18] Randall Stewart and Christopher Metz. "SCTP: new transport protocol for TCP/IP". In: *IEEE Internet Computing* 5.6 (2001), pp. 64–69 (cit. on p. 8).

[19] James N Gray. "Notes on data base operating systems". In: *Operating systems: An advanced course* (2005), pp. 393–481 (cit. on p. 8).

[20] Leslie Lamport. "Paxos made simple". In: *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)* (2001), pp. 51–58 (cit. on pp. 8, 22).

[21] Daniel Peng and Frank Dabek. "Large-scale incremental processing using distributed transactions and notifications". In: *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*. 2010 (cit. on p. 8).

[22] Ling Qian et al. "Cloud computing: An overview". In: *Cloud Computing: First International Conference, CloudCom 2009, Beijing, China, December 1-4, 2009. Proceedings 1*. Springer. 2009, pp. 626–631 (cit. on pp. 8, 15).

[23] Chaowei Yang et al. "Big Data and cloud computing: innovation opportunities and challenges". In: *International Journal of Digital Earth* 10.1 (2017), pp. 13–53 (cit. on p. 8).

[24] Weisong Shi et al. "Edge computing: Vision and challenges". In: *IEEE internet of things journal* 3.5 (2016), pp. 637–646 (cit. on pp. 8, 15).

[25] Flavio Bonomi et al. "Fog computing and its role in the internet of things". In: *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. 2012, pp. 13–16 (cit. on p. 8).

[26] C Mohan, Bruce Lindsay, and Ron Obermarck. “Transaction management in the R* distributed database management system”. In: *ACM Transactions on Database Systems (TODS)* 11.4 (1986), pp. 378–396 (cit. on p. 9).

[27] Alexander Thomson et al. “Calvin: fast distributed transactions for partitioned database systems”. In: *Proceedings of the 2012 ACM SIGMOD international conference on management of data*. 2012, pp. 1–12 (cit. on p. 9).

[28] Erfan Zamanian et al. “The end of a myth: Distributed transactions can scale”. In: *arXiv preprint arXiv:1607.00655* (2016) (cit. on p. 9).

[29] Mei-Ling Liu, Divyakant Agrawal, and Amr El Abbadi. “The performance of two phase commit protocols in the presence of site failures”. In: *Distributed and Parallel Databases* 6 (1998), pp. 157–182 (cit. on p. 9).

[30] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. “A survey of distributed data aggregation algorithms”. In: *IEEE Communications Surveys & Tutorials* 17.1 (2014), pp. 381–404 (cit. on p. 15).

[31] Jian Wang et al. “A survey of vehicle to everything (V2X) testing”. In: *Sensors* 19.2 (2019), p. 334 (cit. on p. 17).

[32] Hantao Li et al. “Effective safety message dissemination with vehicle trajectory predictions in V2X networks”. In: *Sensors* 22.7 (2022), p. 2686 (cit. on p. 18).

[33] EMQX. `https://www.emqx.com/en/blog/mqtt-over-quic` (cit. on pp. 19, 116).

[34] Jim Gray and Andreas Reuter. *Transaction processing: concepts and techniques*. Elsevier, 1992 (cit. on p. 20).

[35] Philip A Bernstein, Vassos Hadzilacos, Nathan Goodman, et al. *Concurrency control and recovery in database systems*. Vol. 370. Addison-wesley Reading, 1987 (cit. on p. 22).

[36] Yang Xiao et al. “Distributed consensus protocols and algorithms”. In: *Blockchain for Distributed Systems Security* 25 (2019), p. 40 (cit. on p. 22).

[37] Diego Ongaro and John Ousterhout. “In search of an understandable consensus algorithm”. In: *2014 USENIX annual technical conference (USENIX ATC 14)*. 2014, pp. 305–319 (cit. on p. 22).

[38] Miguel Castro, Barbara Liskov, et al. “Practical byzantine fault tolerance”. In: *OsDI*. Vol. 99. 1999, pp. 173–186 (cit. on p. 22).

[39] Andrzej Ożadowicz and Jakub Grela. “The street lighting control system application and case study”. In: *2015 International Conference on Event-based Control, Communication, and Signal Processing (EBCCSP)*. IEEE. 2015, pp. 1–8 (cit. on p. 24).

[40] Lorenzo Alvisi et al. “Wrapping server-side TCP to mask connection failures”. In: *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*. Vol. 1. IEEE. 2001, pp. 329–337 (cit. on pp. 28–30).

[41] Kaushik Dutta et al. “User action recovery in internet sagas (isagas)”. In: *International Workshop on Technologies for E-Services*. Springer. 2001, pp. 132–146 (cit. on p. 30).

[42] German Shegalov et al. “EOS: Exactly-Once E-Service Middleware”. In: *VLDB’02: Proceedings of the 28th International Conference on Very Large Databases*. Elsevier. 2002, pp. 1043–1046 (cit. on p. 30).

[43] Naghmeh Ivaki, Filipe Araujo, and Raul Barbosa. “A middleware for exactly-once semantics in request-response interactions”. In: *2012 IEEE 18th Pacific Rim International Symposium on Dependable Computing*. IEEE. 2012, pp. 31–40 (cit. on p. 30).

[44] Randall R. Stewart. *Stream Control Transmission Protocol*. RFC 4960. Sept. 2007. doi: `10.17487/RFC4960`. url: `https://rfc-editor.org/rfc/rfc4960.txt` (cit. on p. 31).

[45] Yunhong Gu and Robert L Grossman. “UDT: UDP-based data transfer for high-speed wide area networks”. In: *Computer Networks* 51.7 (2007), pp. 1777–1799 (cit. on pp. 31, 84).

[46] Adam Lindberg, Sébastien Merle, and Peer Stritzinger. “Scaling Erlang distribution: going beyond the fully connected mesh”. In: *Proceedings of the 18th ACM SIGPLAN International Workshop on Erlang*. 2019, pp. 48–55 (cit. on p. 31).

[47] Inc. Lightbend. *Apache Akka: Message Delivery Reliability*. `https://doc.akka.io/docs/akka/current/general/message-delivery-reliability.html`. Accessed: 2021-08-09. 2021 (cit. on p. 31).

[48] Cheng Jin, David X Wei, and Steven H Low. “FAST TCP: motivation, architecture, algorithms, performance”. In: *IEEE INFOCOM 2004*. Vol. 4. IEEE. 2004, pp. 2490–2501 (cit. on p. 31).

[49] Sangtae Ha, Injong Rhee, and Lisong Xu. “CUBIC: a new TCP-friendly high-speed TCP variant”. In: *ACM SIGOPS operating systems review* 42.5 (2008), pp. 64–74 (cit. on p. 31).

[50] Tom Bova and Ted Krivoruchka. *Reliable UDP Protocol*. Internet-Draft. Internet Engineering Task Force, Feb. 1999. 15 pp. url: `https://datatracker.ietf.org/doc/html/draft-ietf-sigtran-reliable-udp-00` (cit. on p. 31).

[51] Henning Schulzrinne et al. *RTP: A transport protocol for real-time applications*. 1996 (cit. on p. 31).

[52] Eric He et al. “Reliable blast UDP: Predictable high performance bulk data transfer”. In: *Proceedings. IEEE International Conference on Cluster Computing*. IEEE. 2002, pp. 317–324 (cit. on p. 31).

[53] Lee Salzman. *ENet v1.3.17: Reliable UDP networking library*. `http://enet.bespin.org/Features.html`. Accessed: 2021-08-09. 2020 (cit. on p. 32).

[54] Adam Langley et al. "The quic transport protocol: Design and internet-scale deployment". In: *Proceedings of the conference of the ACM special interest group on data communication*. 2017, pp. 183–196 (cit. on p. 32).

[55] Esteban Meneses, Celso L. Mendes, and Laxmikant V. Kalé. "Team-Based Message Logging: Preliminary Results". In: *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. 2010, pp. 697–702. doi: 10.1109/CCGRID.2010.110 (cit. on pp. 33, 35).

[56] Y-M Wang. "Reducing message logging overhead for log-based recovery". In: *1993 IEEE International Symposium on Circuits and Systems*. IEEE. 1993, pp. 1925–1928 (cit. on pp. 34, 35).

[57] Yi-Min Wang and W Kent Fuchs. "Optimistic Message Logging for Independent Checkpointing in Message-Passing Systems." In: *SRDS*. 1992, pp. 147–154 (cit. on p. 35).

[58] Jay Kreps, Neha Narkhede, Jun Rao, et al. "Kafka: A distributed messaging system for log processing". In: *Proceedings of the NetDB*. Vol. 11. 2011, pp. 1–7 (cit. on pp. 36, 37).

[59] Steve Vinoski. "Advanced message queuing protocol". In: *IEEE Internet Computing* 10.6 (2006), pp. 87–89 (cit. on p. 36).

[60] RabbitMQ. *https://www.rabbitmq.com/* (cit. on p. 36).

[61] Bruce Snyder, Dejan Bosanac, and Rob Davies. "Introduction to apache activemq". In: *Active MQ in action* (2017), pp. 6–16 (cit. on p. 36).

[62] Pieter Hintjens. *ZeroMQ: messaging for many applications*. "O'Reilly Media, Inc.", 2013 (cit. on pp. 36, 37, 115).

[63] Benjamin Aziz. "A formal model and analysis of the MQ telemetry transport protocol". In: *2014 Ninth International Conference on Availability, Reliability and Security*. IEEE. 2014, pp. 59–68 (cit. on p. 36).

[64] Tony Speakman et al. *PGM Reliable Transport Protocol Specification*. RFC 3208. Dec. 2001. doi: 10.17487/RFC3208. url: https://rfc-editor.org/rfc/rfc3208.txt (cit. on p. 36).

[65] Guozhang Wang et al. "Building a replicated logging system with Apache Kafka". In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1654–1655 (cit. on pp. 36, 37).

[66] Guruduth Banavar et al. "A case for message oriented middleware". In: *International Symposium on Distributed Computing*. Springer. 1999, pp. 1–17 (cit. on p. 36).

[67] OASIS Standard. "MQTT version 3.1. 1". In: *URL http://docs. oasis-open. org/mqtt/mqtt/v3* 1 (2014) (cit. on pp. 38, 116).

[68] Larry R Dennison. "Reliable interconnection networks for parallel computers". In: (1991) (cit. on p. 38).

[69] Hagit Attiya and Rinat Rappoport. "The level of handshake required for managing a connection". In: *Distributed Computing* 11.1 (1997), pp. 41–57 (cit. on pp. 39, 41).

[70] Ziad Kassam, Paulo Sergio Almeida, and Ali Shoker. *Exon Exactly-Once Oblivious Messaging Library*. `https://github.com/ziadkassam/Exon`. Accessed: 2021-08-09. 2021 (cit. on p. 68).

[71] TIOBE Index. *TIOBE Index for Programming Languages*. `https://www.tiobe.com/tiobe-index/`. Accessed: 2023-09-03 (cit. on p. 69).

[72] Oracle. *Oracle*. `https://www.oracle.com/`. Accessed: 2023-09-03 (cit. on p. 69).

[73] Brian White et al. "An integrated experimental environment for distributed systems and networks". In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 255–270 (cit. on p. 83).

[74] Madzirin Masirap et al. "Evaluation of reliable UDP-based transport protocols for Internet of Things (IoT)". In: *2016 IEEE Symposium on Computer Applications & Industrial Electronics (ISCAIE)*. IEEE. 2016, pp. 200–205 (cit. on p. 85).

[75] Bishwaroop Ganguly et al. "Loss-Tolerant TCP (LT-TCP): Implementation and experimental evaluation". In: *MILCOM 2012 - 2012 IEEE Military Communications Conference*. 2012, pp. 1–6. doi: `10.1109/MILCOM.2012.6415694` (cit. on p. 85).

[76] Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. "Gossip-based aggregation in large dynamic networks". In: *ACM Transactions on Computer Systems (TOCS)* 23.3 (2005), pp. 219–252 (cit. on pp. 106, 119).

[77] Jen-Yeu Chen, Gopal Pandurangan, and Dongyan Xu. "Robust computation of aggregates in wireless sensor networks: distributed randomized algorithms and analysis". In: *IEEE Transactions on Parallel and Distributed Systems* 17.9 (2006), pp. 987–1000 (cit. on pp. 106, 107, 119, 120).

[78] David Kempe, Alin Dobra, and Johannes Gehrke. "Gossip-based computation of aggregate information". In: *Foundations of Computer Science, 2003. Proceedings. 44th Annual IEEE Symposium on*. IEEE. 2003, pp. 482–491 (cit. on pp. 106, 107, 119, 120).

[79] Ittay Eyal, Idit Keidar, and Raphael Rom. "LiMoSense: live monitoring in dynamic sensor networks". In: *Distributed computing* 27.5 (2014), pp. 313–328 (cit. on pp. 106, 119, 120).

[80] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. "Fault-Tolerant Aggregation by Flow Updating." In: *DAIS*. Springer. 2009, pp. 73–86 (cit. on pp. 106, 108, 119, 120, 122, 123, 126).

[81] Carlo Curino et al. "Schism: a workload-driven approach to database replication and partitioning". In: (2010) (cit. on p. 111).

[82] Monowar Hasan et al. "Securing vehicle-to-everything (V2X) communication platforms". In: *IEEE Transactions on Intelligent Vehicles* 5.4 (2020), pp. 693–713 (cit. on p. 112).

[83] Khandaker Foysal Haque et al. "Lora architecture for v2x communication: An experimental evaluation with vehicles on the move". In: *Sensors* 20.23 (2020), p. 6876 (cit. on p. 112).

[84] Khadige Abboud, Hassan Aboubakr Omar, and Weihua Zhuang. “Interworking of DSRC and cellular network technologies for V2X communications: A survey”. In: *IEEE transactions on vehicular technology* 65.12 (2016), pp. 9457–9470 (cit. on p. 113).

[85] Sohan Gyawali et al. “Challenges and solutions for cellular based V2X communications”. In: *IEEE Communications Surveys & Tutorials* 23.1 (2020), pp. 222–255 (cit. on p. 113).

[86] Muhammad Awais Khan et al. “Robust, resilient and reliable architecture for v2x communications”. In: *IEEE Transactions on Intelligent Transportation Systems* 22.7 (2021), pp. 4414–4430 (cit. on p. 113).

[87] Carl Bergenhem, Erik Hedin, and Daniel Skarin. “Vehicle-to-vehicle communication for a platooning system”. In: *Procedia-Social and Behavioral Sciences* 48 (2012), pp. 1222–1233 (cit. on p. 114).

[88] Carl Bergenhem et al. “Overview of platooning systems”. In: *Proceedings of the 19th ITS World Congress, Oct 22-26, Vienna, Austria (2012)*. 2012 (cit. on p. 113).

[89] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent programming in ERLANG*. Prentice Hall International (UK) Ltd., 1996 (cit. on p. 115).

[90] akka. `https://akka.io/` (cit. on p. 115).

[91] Nanomsg-ZeroMQ. *Differences between nanomsg and ZeroMQ*. `http://nanomsg.org/documentation-zeromq.html` (cit. on p. 115).

[92] Jin-Shyan Lee, Yu-Wei Su, and Chung-Chou Shen. “A comparative study of wireless protocols: Bluetooth, UWB, ZigBee, and Wi-Fi”. In: *Industrial Electronics Society, 2007. IECON 2007. 33rd Annual Conference of the IEEE*. Ieee. 2007, pp. 46–51 (cit. on p. 119).

[93] Stephen Boyd et al. “Gossip algorithms: Design, analysis and applications”. In: *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*. Vol. 3. IEEE. 2005, pp. 1653–1664 (cit. on p. 119).

[94] Carlos Baquero, Paulo Sérgio Almeida, and Raquel Menezes. “Fast estimation of aggregates in unstructured networks”. In: *Autonomic and Autonomous Systems, 2009. ICAS’09. Fifth International Conference on*. IEEE. 2009, pp. 88–93 (cit. on p. 119).

[95] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. “A survey of distributed data aggregation algorithms”. In: *IEEE Communications Surveys & Tutorials* 17.1 (2015), pp. 381–404 (cit. on pp. 119, 120).

[96] Gerhard Niederbrucker and Wilfried N Gansterer. “Robust gossip-based aggregation: A practical point of view”. In: *2013 Proceedings of the Fifteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2013, pp. 133–147 (cit. on pp. 119, 120, 122).

[97] Paulo Jesus, Carlos Baquero, and Paulo Sérgio Almeida. “Flow updating: Fault-tolerant aggregation for dynamic networks”. In: *Journal of Parallel and Distributed Computing* 78 (2015), pp. 53–64 (cit. on p. 119).

[98] Wilfried N Gansterer et al. “Robust distributed orthogonalization based on randomized aggregation”. In: *Proceedings of the second workshop on Scalable algorithms for large-scale systems*. ACM. 2011, pp. 7–10 (cit. on pp. 119, 120, 126).

[99] Gerhard Niederbrucker, Hana Straková, and Wilfried N Gansterer. “Improving fault tolerance and accuracy of a distributed reduction algorithm”. In: *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:* IEEE. 2012, pp. 643–651 (cit. on pp. 119, 120, 125, 126).

[100] Ding Liu and Manoj Prabhakaran. “On randomized broadcasting and gossiping in radio networks”. In: *Computing and Combinatorics* (2002), pp. 643–654 (cit. on p. 120).

[101] Ali Shoker, Paulo Sérgio Almeida, and Carlos Baquero. “Exactly-Once Quantity Transfer”. In: *2015 IEEE 34th Symposium on Reliable Distributed Systems Workshop (SRDSW)*. IEEE. 2015, pp. 68–73 (cit. on pp. 120, 124).

[102] Paulo Sérgio Almeida and Carlos Baquero. “Scalable eventually consistent counters over unreliable networks”. In: *arXiv preprint arXiv:1307.3207* (2013) (cit. on pp. 120, 121).

[103] Marc Shapiro et al. “Conflict-free replicated data types”. In: *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*. Springer. 2011, pp. 386–400 (cit. on p. 121).

A

Full-Package Exon

```
1  types
2      𝕀 : node id
3      ℙ : payload
4      𝕄 : (dlv, ℙ)|(fwd, 𝕀, 𝕄)|(tok, 𝕀, ℕ, ℕ, 𝕄)
         |(rmvslots, 𝕀, ℕ, ℕ)
5      𝕊 : record {
6          sck : ℕ, sender clock
7          rck : ℕ, receiver clock
8          msg : 𝕄*, messages queued to send
9          env : ℕ*, list of available envelopes
10         tok : ℕ ↪ 𝕄, tokens with messages
11     }, sender-side connection record
12     ℝ : record {
13         sck : ℕ, sender clock
14         rck : ℕ, receiver clock
15         slt : 𝒫(ℕ), set of available slots
16     }, receiver-side connection record
17 parameters
18     N : ℕ, number of slots requested in advance
19 state:
20     ck_i : ℕ = 0, node clock
21     S_i : 𝕀 ↪ 𝕊 = ∅, map of sender-side records
22     R_i : 𝕀 ↪ ℝ = ∅, map of receiver-side records
23 on EOSend_i(j, m)
24     if j ∉ dom(S_i) then
25         S_i[j] := 𝕊{sck : ck_i, rck : 0, msg : [m],
                    env : [], tok : ∅}
26         requestSlots_i(j)
27     else
28         c = S_i[j]
29         if c.env ≠ [] then
30             e = c.env.dequeue()
31             c.tok[e] := m
32             send_{i,j}(token, e, c.rck, m)
33             if |c.env| = N − 1 then
34                 requestSlots_i(j)
35         else
36             c.msg.enqueue(m)
37 periodically
38     for (j, c) in S_i do
39         for (s, m) in c.tok do
40             send_{i,j}(token, s, c.rck, m)
41         requestSlots_i(j)
42     for (j, c) in R_i do
43         send_{i,j}(slots, c.sck, c.rck, 0)
```

Algorithm 7: Full-Package Exon

```
71  proc requestSlots_i(j)
72      c = S_i[j]
73      n = N + |c.msg| − |c.env|
74      if n > 0 then
75          l =  if c.tok ≠ ∅ then
              min(dom(c.tok))
76               else if c.env ≠ [] then c.env[0]
77               else c.sck
78          send_{i,j}(reqslots, c.sck, n, l)
79      else if c.tok = ∅ and c.msg = [] then
80          send_{i,j}(reqslots, c.sck, 0, c.sck)
81          ck_i := max(ck_i, c.sck)
82          S_i.remove(j)
83  proc delegateConnection_i(j, k)
84      c = S_i[k]
85      x =  if c.env = [] then c.sck
86           else c.env[0]
87      y = c.sck + N + |c.msg| − |c.env|
88      c.env := []
89      c.sck := y
90      while c.msg ≠ [] do
91          EOSend_i(j, (fwd, k,
              (dlv, c.msg.dequeue())))
92      for (s, m) in c.tok do
93          if m instanceOf dlv then
94              EOSend_i(j, (fwd, k, (tok, i, s, c.rck, m)))

95              c.tok[s] := null
96          else
97              EOSend_i(j, (fwd, k, m))
98              c.tok.remove(s)
99      EOSend_i(j, (fwd, k, (rmvslots, i, x, y)))
100 proc handle_i(m)
101     case m of
102         (dlv, payload) then
                deliver_i(payload)
103         (fwd, j, m) then
                EOSend_i(j, m)
104         (tok, j, s, r, m) then
                consumeToken_i(j, s, r, m)
105         (rmvslots, j, x, y) then
                removeSlots_i(j, x, y)
106 proc removeSlots_i(j, x, y)
107     c = S_i[j]
108     for i ← x to (y − 1) do
109         c.slt.remove(s)
110     c.sck := y
111     if c.slt = ∅ then
112         R_i.remove(j)
113 proc consumeToken_i(j, s, r, m)
114     if j ∈ dom(R_i) then
115         c = R_i[j]
116         if r = c.rck and s ∈ c.slt then
117             if m ≠ null then
118                 c.slt.remove(s)
119                 handle_i(m)
120                 send_{i,j}(ack, s, r)
121         else
122             send_{i,j}(ack, s, r)
123     else
124         send_{i,j}(ack, s, r)
125 on receive_{j,i}(reqslots, s, n, l)
126     if j ∉ dom(R_i) then
127         R_i[j] := ℝ{sck : s, rck : ck_i, slt : ∅}
128         ck_i := ck_i + 1
129     c = R_i[j]
130     c.slt := {m ∈ c.slt | m ≥ l}
131     if n > 0 then
132         if s + n > c.sck then
133             c.slt.union({c.sck, . . . , s + n − 1})
134             c.sck := s + n
135         send_{i,j}(slots, s, c.rck, n)
136     if c.slt = ∅ then
137         R_i.remove(j)
138 on receive_{j,i}(slots, s, r, n)
139     if j ∉ dom(S_i) then
140         send_{i,j}(reqslots, ck_i, 0, ck_i)
141     else if s = S_i[j].sck then
142         c = S_i[j]
143         c.rck := r
144         c.env.append([s, . . . , s + n − 1])
145         c.sck := s + n
146         while c.env ≠ [] and c.msg ≠ [] do
147             e = c.env.dequeue()
148             m = c.msg.dequeue()
149             c.tok[e] := m
150             send_{i,j}(token, e, c.rck, m)
151         requestSlots_i(j)
152 on receive_{j,i}(token, s, r, m)
153     consumeToken_i(j, s, r, m)
154 on receive_{j,i}(ack, s, r)
155     if j ∈ dom(S_i) then
156         c = S_i[j]
157         if r = c.rck and s ∈ dom(c.tok) then
158             c.tok.remove(s)
```