



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Georges Younes

Dynamic End-to-End Reliable Causal Delivery Middleware for Geo-Replicated Services

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



Universidade do Minho





Universidade do Minho

Escola de Engenharia

Departamento de Informática

Georges Younes

**Dynamic End-to-End Reliable Causal
Delivery Middleware for Geo-Replicated
Services**

Tese de Doutoramento

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



universidade de aveiro



Universidade do Minho



Trabalho realizado sob a orientação de

Carlos Baquero

e de

Ali Shoker

June 2023

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Este é um trabalho académico que pode ser utilizado por terceiros desde que respeitadas as regras e boas práticas internacionalmente aceites, no que concerne aos direitos de autor e direitos conexos.

Assim, o presente trabalho pode ser utilizado nos termos previstos na licença abaixo indicada.

Caso o utilizador necessite de permissão para poder fazer um uso do trabalho em condições não previstas no licenciamento indicado, deverá contactar o autor, através do RepositoriUM da Universidade do Minho.

Licença concedida aos utilizadores deste trabalho



Creative Commons Atribuição 4.0 Internacional

CC BY 4.0

<https://creativecommons.org/licenses/by/4.0/deed.pt>

Acknowledgements

I would like to express my gratitude to my advisors Carlos Baquero and Ali Shoker, for supporting my work with great advise, patience and encouragement. Also a special thanks to Professor Paulo Sérgio Almeida for his constant support and guidance and to Vitor Enes for being a great colleague and friend, both heavily involved in this work. To all of them, thank you for the continuous support and encouragement throughout this work and for the counseling provided. I learned a lot. Without their guidance and dedication this thesis would not have been possible.


Thanks to my all of my colleagues and friends at the HASLab laboratory, for the great work environment created. Finally, a special thanks to my friends and my family, for all the love and support given, and for always having confidence in me.

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.

_____, _____
(Place) (Date)


(Georges Younes)

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

Resumo

Causalidade Dinâmica Extremo a Extremo em Replicação Geográfica

O crescimento da dependência de serviços baseados na Internet, durante as últimas duas décadas, causou um aumento na adoção de sistemas geo-replicados. O desenho destes sistemas é enquadrado à luz do *teorema CAP*. Neste contexto os modelos de coerência relaxada – *Eventual Consistency* – permitem reduzir o tempo de resposta para com os utilizadores finais e, assim, aumentar a disponibilidade dos sistemas e obter interações mais rápidas. O advento de novas técnicas de convergência como *Conflict-free Replicated Data Types*, amplamente adotados na indústria de geo-replicação como seja no Facebook, PayPal, Microsoft, SoundCloud, entre outros, permitiu também um maior enquadramento formal destas técnicas. Em particular, o modelo de coerência causal, provou ser o modelo mais forte para sistemas sempre disponíveis. Assim torna-se relevante visitar as técnicas de comunicação causal em grupo, e associado middleware de transmissão, pois sendo que muitos destes sistemas foram originalmente construídas à perto de três décadas, precisam de ser adequados ao contexto actual de utilização. Esta tese principia pela análise de novas abstrações para a garantia de propriedades *end-to-end* no registo e entrega causal. Prossegue com a observação de anomalias e ineficiências resultantes de implementações *multi-threaded* de entrega causal, e com a identificação de uma primeira abordagem para garantir causalidade entres os dois extremos do sistema. Após a identificação de problemas na escalabilidade nas implementações que se baseiam em vectores versão ou relógios lógicos, é proposta uma nova solução baseada na manipulação de grafos de dependências e numa eficiente manutenção e simplificação dos mesmos, recorrendo à observação de propriedades de estabilidade. É também proposta uma nova API aos utilizadores do *middleware* de comunicação. A avaliação das soluções propostas foi feita com base num sistema programado em *Erlang* e foi feita a sua avaliação de desempenho e aplicação a quatro casos de estudo.

Palavras-chave: Coerência causal, Entrega causal, Replicação geográfica

Abstract

Dynamic End-to-End Reliable Causal Delivery Middleware for Geo-Replicated Services

The reliance on Internet-based services during the past two decades caused a leap in *geo-replicated* systems, as a means to target clients across the globe, in the light of the *CAP theorem*. Therefore, relaxed consistency models got a lot of attention to reduce the response time to end users, and thus boost the availability of the systems at the cost of delayed – *Eventual Consistency*–. Together with the advent of new convergence techniques like *Conflict-free Replicated Data Types*—widely adopted in the geo-replication industry like Facebook, PayPal, Microsoft, SoundCloud, etc., this led to the reliance on more useful tradeoff consistency models like the causal consistency model, proven to be strongest model for available systems. Intuitively, this suggested another visit to revise the causality techniques, broadcast middleware, and abstractions, originally built three decades earlier for a different set of digital services, i.e., applications, capabilities, and usage. The research in this thesis analyzes the end to end workflow of causality-based services, leading to the identification of new problems and shortcoming in state of the art causality techniques and abstractions, and proposing novel corresponding ones. First, this work discovers that, given that many applications are today multi-threaded, handling causality while overlooking this fact will lead into semantic pitfalls in some classes of applications. A corresponding technique is proposed in this thesis to apply end-to-end time-stamping at the application level instead of the causal middleware. Second, this thesis points out a scalability problem in state of the art causal broadcast middlewares that rely on vector clocks for timestamping. This thesis proposes the first graph-based abstraction for timestamping which is proven to be one order of magnitude more scalable and efficient than its state of the art counterpart. Third, this work identifies existing redundancy in the time-stamping methods used in both causal middleware and application logic, and thus proposes a slightly modified, but effective, API that reduces the bandwidth metadata overhead by half. The API includes the notion of *causal stability* that makes garbage collection fast and easy. Fourth, this thesis introduces the first technique for dynamic causality middleware, crucial in elastic services, leading to guaranteed causal delivery under dynamic membership. These contributions are then implemented in a comprehensive well-engineered codebase in *Erlang*. To demonstrate its usefulness and feasibility, this work has been applied to four practical use-cases and projects during the course of this thesis.

Keywords: Causal Consistency, Causal Delivery, Geo-Replication

Contents

List of Figures	xiii
List of Algorithms	xv
1 Introduction	1
1.1 Problem statement and objectives	2
1.2 Main contributions and results	3
1.2.1 Tagged Causal Delivery (Broadcast)	3
1.2.2 Graph-based Tagged Causal Delivery middleware	3
1.2.3 Causal Stability	3
1.2.4 Dynamic Tagged Causal Delivery middleware	4
1.3 Outline	5
2 State of The Art	6
2.1 Geo-replication	6
2.2 (Geo-)replication properties and trade-offs	7
2.2.1 FLP	8
2.2.2 Consistency in data storage systems	8
2.3 Consistency models	11
2.3.1 Data-centric consistency models	11
2.3.2 Client-centric consistency models	14
2.4 Causality tracking and broadcast	16
2.4.1 Causality tracking primitives	16
2.4.2 Causal Broadcast	22
2.4.3 Scalability solutions for Causal Multicast	25
2.5 Weakly-consistent data stores	29
2.5.1 Causally Consistency Data Stores	30

3	Tagged Causal Broadcast	37
3.1	Introduction	37
3.2	Classical Causal Delivery	38
3.3	Defining an End-To-End Happens-Before	40
3.4	The need for happens-before by applications	40
3.4.1	The Stock Trading Example	41
3.4.2	Implementing Conflict-free Replicated Data Types	42
3.5	Happens-Before as a Middleware Service	43
3.5.1	Two-Level Tagging using Current Middleware	43
3.5.2	Exact Tagging by the Middleware	44
3.6	Pitfalls in Exposing Middleware Timestamps	44
3.7	Tagged Causal Delivery	45
3.7.1	Lessons Learned and a General Solution	45
3.8	Explicit Causality versus Explicit Grouping	47
4	Middleware	49
4.1	Introduction	49
4.2	API and Architecture	49
4.2.1	Architecture	50
4.2.2	API	51
4.3	Vector-Clock-based algorithm for tagged causal delivery	52
4.3.1	Client process	52
4.3.2	Middleware process	53
4.4	Causal DAG	54
4.4.1	Reducing the causal information overhead	55
4.4.2	Reducing delivery time	56
4.4.3	Causal dependency graph: notations and functionality	56
4.5	Graph-based Algorithm for tagged causal delivery	57
4.5.1	Client Process	58
4.5.2	Middleware Process	59
5	Tagged causal delivery and Causal Stability	62
5.1	Causal Stability	62
5.1.1	Causal Stability in the vector clock-based algorithm for TCD	63
5.1.2	Causal Stability in the graph-based algorithm for TC	64
5.2	Causal Stability for VV-based Algorithm	66
5.2.1	Client process	66
5.2.2	Middleware process	66

5.3	Causal Stability for graph-based TCB Algorithm	69
5.3.1	Client Process	69
5.3.2	Middleware Process	70
5.4	Phantom Dots: An Optimisation for Active/Passive Node	74
5.4.1	Client Process	74
5.4.2	Middleware Process	75
6	Dynamic Membership	77
6.1	Causal Stability in Dynamic Membership	77
6.1.1	Causal Stability and Join Requests	77
6.1.2	Causal Stability and Leave Requests	78
6.2	Algorithm	79
6.2.1	System Startup	79
6.2.2	Joining Nodes	83
6.2.3	Leaving Nodes	85
6.2.4	Updating Group Membership	86
7	Causality Checker	87
7.1	Causal Check Algorithm	88
7.1.1	causalcheck()	90
7.1.2	handlesenderdot()	90
7.1.3	handledelivered()	91
7.1.4	handlestable()	91
7.1.5	handlepeerdot()	91
8	Evaluation	93
8.1	Configuring the experiments	93
8.1.1	Send interval	93
8.1.2	Network latency	94
8.1.3	Simulating slow links	95
8.2	Deploying environment	96
8.2.1	Architecture	96
8.2.2	Docker	97
8.2.3	Kubernetes	97
8.3	Comparing experiments	97
8.3.1	Memory Metadata	98
8.4	Broadcast Experiments	100

8.4.1	Experiment1: Classical end-to-end vector-based W vs our end-to-end graph-based TCB	100
8.4.2	Experiment2: Classical vector-based W vs graph-based	101
8.4.3	Experiment3: Constant amount of work for vector-based W vs graph-based	103
8.4.4	Causal Stability	103
8.4.5	Causal Delivery	105
8.4.6	Slow Links	107
9	Use Cases	109
9.1	ASPAS	109
9.1.1	Data Types for backup and recovery	110
9.2	Lasp	111
9.3	AntidoteDB	113
9.4	Redis with relaxed consistency	113
9.5	Minidote	115
9.6	Software, libraries and artifacts	117
10	Conclusions and Future Perspectives	118
10.1	Future Work	119
	Bibliography	120

List of Figures

1	A Geo-replicated system.	7
2	The CAP trade-offs.	9
3	Example for linearizability	12
4	Example of Sequential Consistency	13
5	Example of Causal consistency	14
6	Example of Monotonic Reads	15
7	Example of Monotonic Writes	15
8	Example of Read Your Writes	16
9	Diagram showing a distributed execution using Lamport's Happens-before	17
10	Diagram showing a distributed execution using Lamport clocks	18
11	Diagram showing a distributed execution using Causal Histories	19
12	Diagram showing a distributed execution using vector clocks	20
13	Diagram showing a distributed execution using version vectors	21
14	Diagram showing a distributed execution using the CBCAST algorithm	24
15	Trading example from Cheriton and Skeen's paper	41
16	Add-Wins Set Example	42
17	Difference between internal tagging and end-to-end tagging.	46
18	Tagged causal delivery architecture	50
19	Graphical representation of causal delivery using a delivery queue and a causal DAG	54
20	Stages of a dot in causal DAG (without causal stability)	60
21	Flowchart showing how the algorithm works upon receiving a new message (without causal stability)	61
22	Stages of a dot in causal DAG (with causal stability)	64
23	Graphical representation of causal delivery using a delivery queue and a causal DAG (with causal stability)	65

24	Evolution of the causal DAG (with causal stability)	73
25	Evolution of a new node joining process	83
26	Evolution of an existing node leaving process	85
27	Full mesh topology with a slow link between 0 and 1.	95
28	Experiment 1: transmission and memory, without stability.	101
29	Experiment 1: Memory and Non-causal Stability latency, with stability.	102
30	Experiment 2: transmission and memory metadata, without stability.	103
31	Experiment 3: transmission and memory, without stability.	104
32	Experiment 2 and 3: Median memory metadata, with stability.	104
33	Experiment 2 and 3: non-causal stability latency, with stability.	105
34	Experiment 2: non-causal delivery latency.	106
35	Experiment 3: non-causal delivery latency.	106
36	Experiment 2: non-causal delivery latency on slow link.	107
37	Experiment 3: non-causal delivery latency on slow link.	108
38	ASPAS architecture.	109
39	polog based Add-Wins Set with Clone functionality	111
40	Lasp runtime system.	112
41	Lasp pure op-based CRDTs.	112
42	Concrete Resettable Counter Implementation	113
43	The general architecture of our multi-master proposed solution.	114
44	High-level architecture of Minidote.	116

List of Algorithms

1	CBCAST algorithm using Vector clocks on process $i \in \mathbb{I}$	24
2	Vector-based tagged causal delivery algorithm without causal stability at client process for node $i \in \mathbb{I}$	53
3	Vector-based tagged causal delivery algorithm without causal stability at middleware process for node $i \in \mathbb{I}$	54
4	Graph-based tagged causal delivery algorithm without causal stability at client process for node $i \in \mathbb{I}$	58
5	Graph-based tagged causal delivery algorithm without causal stability at middleware process for node $i \in \mathbb{I}$	59
6	Vector-based tagged causal delivery algorithm with causal stability at client process for node $i \in \mathbb{I}$	66
7	Vector-based tagged causal delivery algorithm with causal stability at middleware process for node $i \in \mathbb{I}$	67
8	Graph-based tagged causal delivery algorithm with causal stability at client process for node $i \in \mathbb{I}$	69
9	Graph-based tagged causal delivery algorithm with causal stability at middleware process for node $i \in \mathbb{I}$	70
10	Graph-based tagged causal delivery algorithm with causal stability and phantom messages at client process for node $i \in \mathbb{I}$	75
11	Graph-based tagged causal delivery algorithm with causal stability and phantom messages at middleware process for node $i \in \mathbb{I}$	76
12	Dynamic tcb client process for node $i \in \mathbb{I}$	80
13	Dynamic tcb middleware process for node $i \in \mathbb{I}$	81
14	Causal check algorithm	89

Introduction

From social network platforms to entertainment services, storage and backup, collaborative tools and many more, online services are continuously increasing in scale and in demand. More and more users access these services around the world expecting a reliable service and a good, fast user experience. For system designers, this means ensuring a correct, uninterrupted operation, with fast response time. This technically translates to high availability which implies tolerance to both network and replica failures, as well as low latency of user requests.

A common approach to address the above challenges is to use geo-replication, where full or partial replicas are located in different regions to provide fast access to nearby users. On one hand, this allows the possibility of switching to other replicas when a certain replica crashes. On the other hand, distributed services provide a minimal degree of performance such as being highly-responsive to client requests. Reaching this quality of service can become a challenge, especially over large geographical areas and under large numbers of requests. Geo-replication allows distributing the data and placing it in the proximity of clients, often using relaxed consistency models. This decreases the access delays, resulting from the transmission of requests and responses between servers and clients. In addition to this, replication reduces the load of client requests by allowing different replicas to share that load and divide the work among them.

Data consistency is the main challenge that has to be faced when dealing with geo-replication. Ensuring that all replicas have a consistent state is expected by users, but it comes at the price of delaying the system and serving the users. Users also expect high-responsiveness and thus cannot always accept the high latency of ensuring data consistency.

To solve this, distributed systems offer a wide range of consistency models and tradeoffs between consistency and availability to choose from. Strong consistency models prioritize consistency over availability, while weak consistency models offer less guarantees for consistency and offer higher availability. Causal consistency [2] is the strongest achievable consistency that does not sacrifice availability under network partitions [3]. It is an attractive model for distributed systems and data stores as it guarantees data consistency that reflects causality between events.

This thesis identifies three main problems for ensuring correct, efficient, and dynamic causal delivery. We summarize these problems in the following section with hints on the proposed solutions. We then present the main contributions in the following Section 1.2. Both, problems and contributions, are then presented in detail in the different chapters of this thesis according to the organization described in Section 1.3.

1.1 Problem statement and objectives

Tracking causality and guaranteeing that causal dependencies are satisfied has taken a large interest in distributed systems and still does. It has also led to many contributions by distributed systems researchers and engineers such as causal delivery middleware, causality tracking data structures and mechanisms, garbage collection for causality metadata and so on.

End-to-end potential causality. Traditional causal delivery middleware [4] provide a delivery order in each process that is *consistent* with causality. This means that messages are guaranteed to be delivered at every process in an order respecting (consistent with) the causality between the messages. As for concurrent messages that are not causally related, the order of delivery is arbitrary and dictated only by their arrival at different processes. While such traditional causal delivery middleware guarantee causality, they do not provide the application with knowledge about concurrency between events. Given two messages m_1 and m_2 delivered in that sequence to some process, no information is provided to the application whether m_1 causally preceded m_2 or if they were originated independently (concurrently) of each other. In this thesis, we show that providing such knowledge to the application is necessary for a class of applications that require knowledge of concurrency in order to apply arbitration rules [5]. Ad hoc arbitration of concurrent messages does not influence causality but that might lead to anomalies and break constraints set by the application.

Causal stability. Causal stability concerns knowledge about the end-to-end happens-before regarding future deliveries at each client process, namely whether concurrent messages (to a given message) can still be delivered. The concept itself is not new, and it has been used many times, but hidden in applications, without being properly recognized as such. This has led to successive ad-hoc re-implementations, sometimes overly complex. As an example, causal stability is hidden inside the difficult to understand implementation of Replicated Growable Arrays (RGAs) [6], while not being recognized as a building block. We propose this concept is important enough to be recognized, advertised, and provided by the middleware as a building block, to avoid “reinventing the wheel”. For this we follow the *causal stability* concept, coined as such in a paper that defines pure operation-based CRDTs [7].

Dynamicity and node churn. Traditionally when nodes in group membership leave or join the group, the system blocks until the remaining nodes are aware of the new change in the membership and ready

to resume the distributed computation. This may not be problematic in some scenarios where node churn is rare. However, in highly dynamic scenarios where nodes join and leave regularly and often, this could have a high cost on the responsiveness and availability of the service.

1.2 Main contributions and results

Over the course of this thesis, we have made several contributions to the state-of-the-art of distributed data stores to address the aforementioned problems and challenges. Our main four contributions can be summarized by the following.

1.2.1 Tagged Causal Delivery (Broadcast)

In particular, we define a novel definition of end-to-end happens-before that focuses on application (client) level visibility for tracking causal dependencies. This definition was motivated by the lack of an end-to-end happens-before definition that has led to incorrect behaviour and anomalies in a class of applications as we explain in Chapter 3. We also show the pitfalls of trying to achieve end-to-end happens-before that distributed systems practitioners could fall in. And finally, we develop a Tagged Causal Delivery middleware that characterizes end-to-end happens-before, showing the correct architecture, API and algorithm for both a VV-based version and our graph-based TCB version.

1.2.2 Graph-based Tagged Causal Delivery middleware

A lot of implementation of traditional causal middleware use version vectors, delivery queues and other data structures that are not best suited for the partial order of causality. There have been works that improved the efficiency vector clocks to reduce the metadata sent on the network, and other works that try to optimise the causal delivery mechanisms. However, we argue that a causal delivery middleware should be viewed as a standalone service where all its components work together to provide an efficient and scalable service in a holistic fashion. We then present the correct API, architecture and algorithm of such middleware. We implement also, in addition to our graph-based TCB, an optimized end-to-end mechanism, E2E VV-based in order to have a fair evaluation that compares the performance of both. In the evaluation, we show how our TCB performs better than the optimized VV-based algorithm in terms of transmission of meta data, size of meta data in memory and non-causal delivery and stability latency.

1.2.3 Causal Stability

Our work throughout the thesis focuses around the importance on causal stability in the Tagged Causal Delivery middleware. We show how using causal stability allows designing and implementing correct causal middleware for applications like CRDTs, where knowledge about concurrency between events is available to the application and the causality order characterizes the end-to-end happens before. Moreover,

we show how causal stability allows the safe garbage collection of causality meta data. We also implement the causal stability mechanism for the VV-based version, which is completely different than that of the graph-based version, in order to compare the performance of both algorithms when the causal stability mechanism is running. Again, we notice that our TCB performs and scales better than the VV-based algorithm for causal stability.

Some other smaller contributions were made throughout the thesis, such as phantom messages, a feature to keep causal stability working even when the majority of peers are passive. We explain this in detail in Section 5.4. We also design and implement our own causality checker tool as shown in Chapter 7. This causal checker serves as a verification tool for causal delivery and stability.

1.2.4 Dynamic Tagged Causal Delivery middleware

In this thesis, we introduce, to our knowledge, the first dynamic causal middleware where new nodes can join and existing nodes can leave safely, without blocking the system. We also extend the TCD (or TCB) middleware, through causal stability, to provide a version of the algorithm suited for unreliable and dynamic environments. Our causal checker also verifies the correctness of causal delivery and stability when different nodes join and leave concurrently.

Publications. Part of the work described in this thesis was published or is under submission in several peer-reviewed conferences and journals:

- ASPAS: As Secure as Possible Available Systems. Published at IFIP DAIS'21. Best Paper Award at IFIP DisCoTec 2021.
- Integration Challenges of Pure Operation-based CRDTs in Redis. Published at EuroSys PAPOC workshop, 2016.
- Compact Resettable Counter through Causal Stability. Published at EuroSys PAPOC workshop, 2017.
- Pure Operation-Based Replicated Data Types. Journal under re-submission.
- The Pitfalls in Achieving Tagged Causal Stability. Presented at EuroSys PAPOC workshop, 2018.
- Tagged Causal Delivery: Efficient End-to-End Causal Middleware for Replicated Systems at Scale. Under submission.
- Graph-based Causal Middleware for Dynamic and Unreliable Networks. To be submitted.

1.3 Outline

The rest of this thesis is organized as follows:

- Chapter 2 gives a brief presentation on the required background and an overview of the state of the art literature relevant to the rest of the thesis;
- Chapter 3 presents Tagged Causal Delivery that characterizes end-to-end happens-before, the problems and pitfalls of achieving it incorrectly and the correct architecture and implementation of TCB;
- Chapter 4 presents our graph-based TCB middleware, its architecture, API and algorithm, with a detailed explanation of the dependency dots and causal DAG;
- Chapter 5 presents and motivates Causal Stability and how to implement it in both vector-based and graph-based algorithms;
- Chapter 6 presents the Dynamic version of our TCB algorithm;
- Chapter 7 presents our own causal checker that verifies a correct delivery and stability respecting the causality of events;
- Chapter 8 presents the empirical evaluation comparing our TCB to a vector-based causal delivery middleware;
- Chapter 9 presents use cases where our contributions were implemented in existing software and data stores;
- Chapter 10 is the conclusion of the thesis with some remarks about future work directions.

State of The Art

The work in this thesis is related to three main areas, namely, geo-replication, consistency models, and causality tracking and reordering. Consequently, we present in this chapter the state of the art in the three areas in order to provide an adequate background to understand the work described in the following chapters. I start with geo-replication, its importance nowadays, considering the high number of users and the large amount of data used in distributed services, and the challenges that geo-replication introduces for system designers and engineers. I discuss the properties of such systems and their trade-offs when they arise. I then narrow the focus to consistency in replicated storage systems, visiting notable theorems like CAP, PACELEC, CAC and TV. I then visit the different consistency models that range on a spectrum from Strong consistency to weak consistency. I focus mostly on causal consistency being the strongest consistency model for higher availability, scalability and fault-tolerance. Next, I explore the different solutions used for causality tracking and ordering and show the different data versioning and timestamping techniques that have been used to guarantee causal consistency. Finally, I touch upon different mechanisms, solutions and architectures used in building weakly consistency and causally consistent key-value stores. I conclude this chapter with a state of the art analysis and discussion, showing some of the motivations of my work.

2.1 Geo-replication

Although geo-replication (in Figure 1) solves many distributed systems problems, it generates new ones that have to be dealt with. Data consistency, or more precisely maintaining data/state consistency among replicas, is the burden brought by geo-replication. The challenge here becomes designing systems that guarantee high availability and low latency without sacrificing data consistency which was proven to be a hopeless endeavor due to the need of some form of synchronization [8–10]. What differentiates a system design from another are the tradeoffs and properties of such designs. Although consistency and availability are usually the most desired ones, other properties such as latency, throughput, and partition-tolerance, could not be overlooked due to their importance, that often comes from the semantics of the higher level applications. Some of these properties are achievable together while others are inherently antagonistic.



Figure 1: A Geo-replicated system.

The system properties are defined to satisfy its design goals, and whenever conflicts between these design goals are encountered, an array of trade-offs arises.

2.2 (Geo-)replication properties and trade-offs

A geo-replicated system is a variant of distributed system whose properties can broadly follow the classical Safety properties and Liveness properties [11]. Safety properties address the correctness of an algorithm which informally translates to "nothing bad will ever happen" while liveness properties translates to "something good will eventually happen". The categorization of distributed system properties as safety or liveness properties is helpful as the trade-offs between its properties are in some sense safety-liveness trade-offs. This is intuitively correct as the stricter (safer) the property the more complex the underlying protocols need to be and this negatively influences the performance (liveness) of the system.

In the following, I start with presenting the *FLP impossibility* result [12], which specifies that either the liveness or the safety of consensus of replicas must be sacrificed if those replicas are prone to failure in an asynchronous setting. I then narrow down the context from general setting State-Machine Replication (SMR) towards trade-offs in storage systems where operations are restricted to read and write operations. In this vein, I start with the CAP theorem [9, 10] and then the PACELC, considered as its extension [8]. I then visit the CAC and TV (throughput vs visibility trade-offs) theorems which focus on causal consistency, as a more specific form of consistency.

2.2.1 FLP

The FLP addresses the trade-off between safety and liveness by proving the impossibility of a deterministic algorithm solving consensus in an asynchronous system even under a single crash fault. The main idea is that in an asynchronous system, messages delays can be arbitrarily long which makes a slow replica indistinguishable from a one that crashed. On one hand, if the replica was falsely suspected of crashing by the other replicas, who would proceed with the execution, this leads to disagreement and thus breaking the safety property of consensus. On the other, if the replica was considered slow and it has actually crashed, the other replicas will wait indefinitely and thus breaking the liveness property.

The FLP impossibility results plays an important role in the understanding of replicated systems because consensus plays an important role in state machine replication [3]. The use of state machine replication (SMR) is widespread because it can replicate the state of any deterministic system to become a universal solution for many applications. However, this generality is not needed in all modern services (e.g., social networks) where relaxing the restrictions of strong semantics to benefit from other trade-offs. One example is storage services which support read and write operations. Such services are easier to implement than a general state machine because read and write are very simple operations. By relaxing the requirements on semantics, we can achieve fault-tolerance in storage services even in an asynchronous setting.

2.2.2 Consistency in data storage systems

2.2.2.1 CAP

In order to implement replicated storage system with consistency, communication between replicas is necessary. When network partitions happen, some replicas become isolated from others. Two scenarios may arise here: either all replicas continue serving the client requests without coordination among them, and thus breaking consistency, or replicas wait for the communication to recover, thus violating availability. This trade-off between consistency and availability is in fact a trade-off between safety and liveness as strong consistency is a typical safety property and availability, a typical liveness property. This was popularized by the CAP theorem [9, 10], with the catch-phrase: given Consistency, Availability and Partition-Tolerance, one must choose two of the three (see Figure 2).

Although the CAP theorem implies that there are three combinations, choosing partition-tolerance is inevitable in a geo-replicated system as they are part of failures and crashes that will most likely happen. Therefore, the common design choices are between CP systems (consistent) and AP systems (available). Both AP and CP systems are useful in specific scenarios. For instance, most traditional database management systems are CP (ACID systems) and offer strong consistency guarantees (Isolation in ACID) but sacrifice availability under network partitions. AP systems (BASE systems) such as distributed key-value stores remain available even under partitions by relaxing their consistency guarantees.

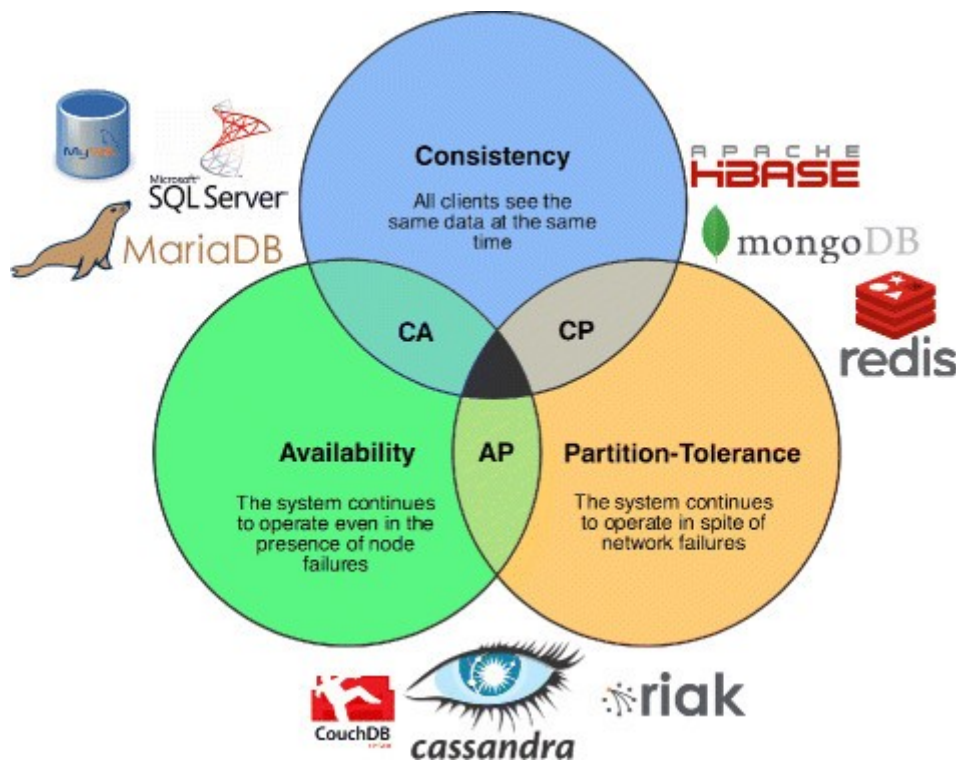


Figure 2: The CAP trade-offs.

2.2.2.2 PACELC

PACELC [8] can be seen as an extended version of the CAP theorem or even a refinement. While CAP addresses the trade-off between consistency and availability when network partitions arise, PACELEC adds to that even in the absence of partitions, there exists a trade-off between consistency and latency. In fact, most geo-replicated services have to provide a minimal degree of performance such as being highly-responsive to client requests. The degree of coordination required between replicas to ensure consistency may increase the latency of serving these client requests. If we consider for instance systems that guarantee strong consistency, the coordination required between replicas could result in latency in the order of hundreds of milliseconds. On the other hand, weaker consistency guarantees do not require such coordination and allow for lower latency and thus could serve client requests faster. In the next sections, I touch upon two theorems for such weaker models, namely, CAC and TV, discussing their consistency guarantees and trade-offs.

2.2.2.3 CAC

Causal consistency [2] is a weaker consistency model than strong consistency but it provides intuitive semantics for many applications as it guarantees that causal relationships between operations are respected. A formal specification on causal consistency is introduced in Subsection 2.3.1.3. The importance of causal consistency is that it is the strongest achievable consistency that does not sacrifice availability under network partitions [3]. This consistency model seems to be the best suited when dealing with the

consistency-availability trade-off. Formally, weaker consistency models can provide high availability during network partitions where replicas do not need to coordinate. However, such consistency models are not useful in practice as replicas can diverge. To address this, Mahajan et al. define convergence as a property to model this usefulness.

Convergence captures the coordination between replicas and could either be synchronous or asynchronous depending on the required consistency guarantees. For strong guarantees such linearizability, a stronger version of convergence is needed where replicas must synchronously coordinate and agree on a total order of operations. For weaker consistency models, such as eventual consistency, a weaker version of convergence is needed. In this case, replicas can coordinate asynchronously, and are allowed to diverge temporarily before converging to the same state when updates stop. This property of convergence is about replicas reaching the same state. The state must reflect all the updates from all replicas and in the case of causal consistency respect the causality between about updates. Since the order is a partial order, replicas may diverge temporarily under the convergence property.

Causal+ consistency. As mentioned earlier, causal consistency guarantees a consistent ordering of all causally related events, leaving the freedom for concurrent operations to be applied in different orders on different replicas. Causal+ (or convergent causal) consistency [13] dictates, in addition to causal consistency, that all replicas eventually and independently reach (converge to) the same state. In fact, operations or more accurately writes that are not causally related (concurrent) may generate conflicting results at different replicas without breaking causality. Causal+ strengthens causal consistency with strong convergence, which dictates that all replicas that have applied the same operations will have equivalent states.

2.2.2.4 Throughput versus Visibility (TV) latency

Although causal consistency seems like a sweet-spot solution for the availability-consistency trade-off, it also allows new trade-offs to surface. The one we discuss here is a trade-off between throughput and visibility [14, 15]. This tradeoff discusses the balance between the granularity of causality metadata propagation versus the visibility latency of the causal dependencies at the receiver. My thesis tackles this tradeoff to optimize both throughput and Visibility latency.

Although interesting, causal consistency protocols need to rely on generating and propagating metadata to track causal dependencies between updates, and thus guarantee that a causal order is respected across all replicas. Different techniques and tracking mechanisms were developed to ensure causality which we discuss in more details in Section 2.4. What these techniques all have in common is propagating metadata that encodes the causal relations between updates.

Propagating causal meta-data requires more bandwidth and thus has a cost on throughput. On the other hand without this metadata causality would not be guaranteed. The amount of metadata sent and its granularity is what we look to fine-tune in this throughput vs visibility trade-off. Fine-grained metadata

allows to track dependencies among individual objects and each replica may apply the received update as soon as those dependencies are satisfied. The visibility latency per update is minimized here because only the dependencies of that object are considered. However, fine-grained metadata is large that decreases throughput. On the other hand, tracking dependencies for groups of objects, or coarse-grained metadata, reduces the size of metadata required and thus improves throughput, but at the cost of increasing the visibility latency that could result from false dependencies.

2.2.2.5 A final note on Sticky versus non-sticky availability

Sometimes it is necessary to distinguish between *sticky* versus *non-sticky*, a.k.a, high availability, when we talk about consistency models and the trade-offs [16]. The system provides sticky availability when clients are assumed to stick to the same replica. Non-sticky (high availability) is when a client is permitted to switch from one replica to another.

To understand why this matter, consider the case of the CAC result. We mentioned that causal consistency is the strongest achievable consistency without sacrificing availability. This statement is correct for sticky availability [16], but does not hold for high availability. In the system model of CAC, availability is sticky availability because there was no distinction between clients and server replicas and so a client is always connected to the same replica. This is common in peer to peer systems, for instance, where the same replica is at the same time a client and a server. However, in a general case, clients are separate entities and may switch connections from one system replica to another.

I will differentiate between high-availability and sticky availability when visiting the consistency models in the following.

2.3 Consistency models

There are many consistency models proposed to guarantee data consistency in a replicated system. Consistency models are often classified as data-centric or client-centric. In the former, data consistency is observed at replicas, whereas the latter deals with consistency from a client perspective (often call session guarantee). Although the focus of this thesis is on data-centric models, I opt to overview the client-centric model for the sake of completeness and better understanding when I consider consistency while referring to application semantics. I also try to discuss these consistency models in light of the tradeoffs presented earlier.

2.3.1 Data-centric consistency models

In this section, I overview four main data-centric models starting with the stronger through the weaker consistency ones.

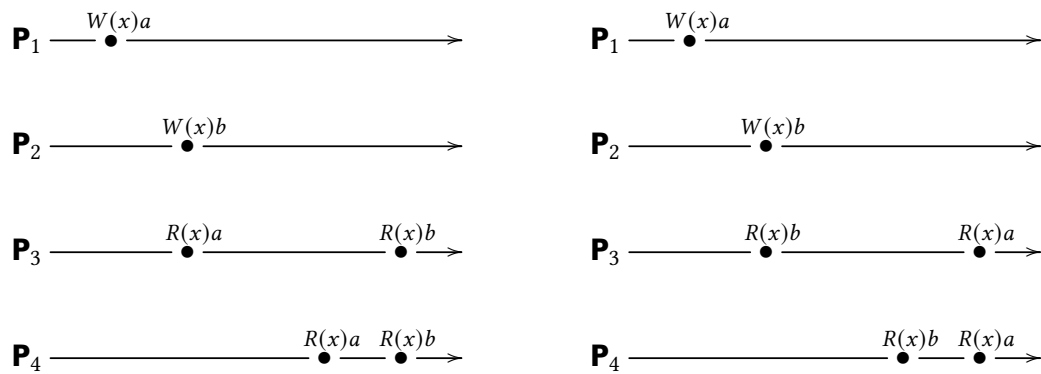


Figure 3: Example for linearizability

2.3.1.1 Linearizability

Linearizability [17] (also called Atomic Consistency, Immediate Consistency, External Consistency or Strong consistency) is a data-centric consistency model. It has the strongest consistency guarantees. Linearizability guarantees for single operations on single objects that is consistent with the real time of the operations. Informally, under linearizability, writes should appear instantaneous and atomic. All reads that are invoked (in real time) after a certain read or write operation on an object should return that value or the value of a later write. There are costs with linearizability: availability (both high and sticky) in face of failures (like network partitions) and performance. The latter can be also seen even in tightly coupled systems such as multi-processors: for performance reasons, variables across threads in different CPU cores are not guaranteed linearizability, since it would slow down the CPU [18].

To better understand linearizability, consider the sequence of operations in Figure 3. In the left figure, P_1 first performs a write operation on x with value a . Later (in real time), Process P_2 performs a write operation and sets the value of x to b . Both P_3 and P_4 read value a then b which is consistent with the sequence of operations across replicas in real time. We see an example of a linearizable execution. In the right figure, we see a violation of linearizability even though P_3 and P_4 observe the same sequence because the real time order of write operations at P_1 and P_2 is violated.

2.3.1.2 Sequential Consistency

Sequential consistency is a data-centric consistency model. It is a strong safety guarantee for distributed systems. It is weaker than Linearizability as it does not respect real time constraints. However, writes are totally ordered according to logical time across all replicas. Informally, sequential consistency implies that the result of any execution is the same as if the read and write operations by all processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program [19]. Sequential consistency is not suitable for availability (neither high nor sticky) under partitions.

In the left diagram of Figure 4, P_1 first performs a write operation on x with value a . Later (in real

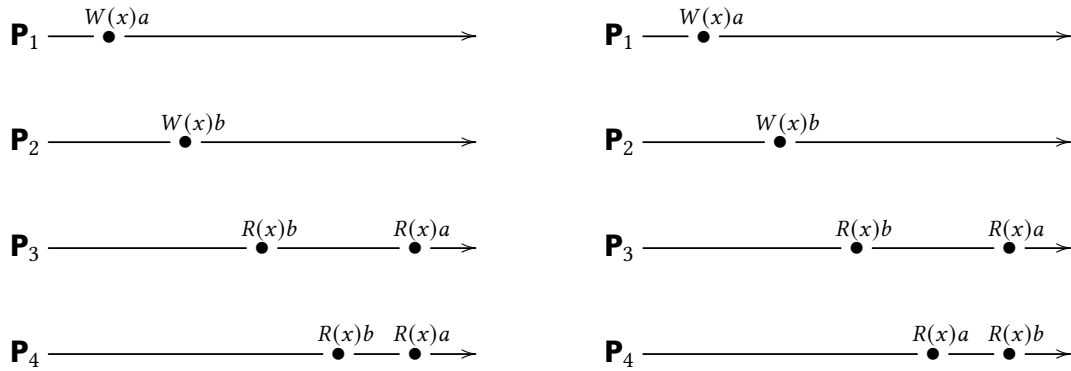


Figure 4: Example of Sequential Consistency

time), Process P_2 performs a write operation and sets the value of x to b . Both P_3 and P_4 read value b then a . Write operation of process P_2 appears to have taken place before that of P_1 . We see an example of a sequentially consistent execution. In the right diagram of Figure 4 we see a violation of sequential consistency as not all processes see the same interleaving of write operations. At P_3 it seems that x had value b then later a while at P_4 , x reads a and later b .

2.3.1.3 Causal consistency

The Causal Consistency model is the core consistency model targeted in my thesis. Causal consistency [20] is a data-centric consistency model. It ensures that causally-related operations to appear in the same order on all processes but does not guarantee any specific order for causally-independent (concurrent) operations. Causal consistency is a very interesting consistency model in the consistency spectrum. On one hand it is the strongest consistency that can be achieved without compromising availability [3, 21] under partitions, while still performing better than other consistency models such as linearizability or sequential consistency. On the other hand it seems that capturing causal relationships (cause and effects) is an important as it reflects physical reality and the way we experience it: we always see the effects after their causes.

It is important to note that when talking about causality here, *potential* causality is what is often referred to. The reason is that it is not always known if a certain operation was the real cause of another or if it just happened-before it. Causal consistency is sticky available: if a network partition occurs, every node can make progress, so long as clients never change which server they talk to. We discuss capturing causality in more details in the following section.

In the left diagram of Figure 5, $W_2(x)b$ potentially depending on $W_1(x)a$ because b may result from a computation involving the value read by $R_2(x)a$. The two writes are causally related, so all processes must see them in the same order. However we see a different order at P_3 and P_4 . This figure shows a violation of causal consistency. In the right diagram of Figure 5, we see a correct execution of causal consistency. On P_2 the read ($R_2(x)a$) is not there before the write, so $W_1(x)a$ and $W_2(x)b$ are now

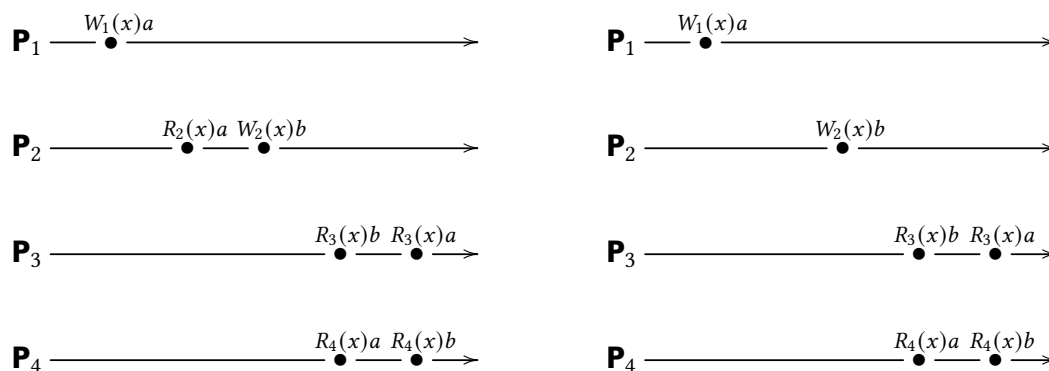


Figure 5: Example of Causal consistency

concurrent writes. A causally-consistent store does not require concurrent writes to be globally ordered and there for causal order has not been violated.

2.3.1.4 Eventual Consistency

Eventual Consistency [22] is a data-centric consistency model. It is what most databases offer as the weakest consistency level. It states that if writes to the database stop, eventually (i.e., after some unspecified time) the replicated state will be consistent, returning the same value to every process. Implicitly, there is the assumption that data aims to be stored, ruling out vacuous implementations which returned the same value forever. This consistency model can be thought simply as convergence guarantee, without any time guarantee.

2.3.2 Client-centric consistency models

In this section, I shed light on three well-known client-centric consistency models that provide application session guarantees: Monotonic Reads, Monotonic Writes, and Read Your Writes.

2.3.2.1 Monotonic Reads

Monotonic reads is a client-centric consistency model also called session guarantees. It ensures that if a process reads the value of a data item x , any successive read operation on x by that process will always return that same value or a more recent one. Monotonic reads only applies to reads invoked by the same process and does not apply to operations performed by different processes. Monotonic reads can be totally available: even during a network partition, all nodes can make progress.

We see in Figure 6 an example of a process P with two different local copies L_1 and L_2 . The read operations ($R(x_1), R(x_2)$) are carried out by the same process P . The $WS(x_i)$ operations are a set of write operations on x and not necessarily by P only that return x_i . Process P first performs a read operation on x at L_1 , returning the value of x_1 (at that time). This value results from the write operations in $WS(x_1)$ performed at L_1 . Later, P performs a read operation on x at L_2 , shown as $R(x_2)$. To guarantee

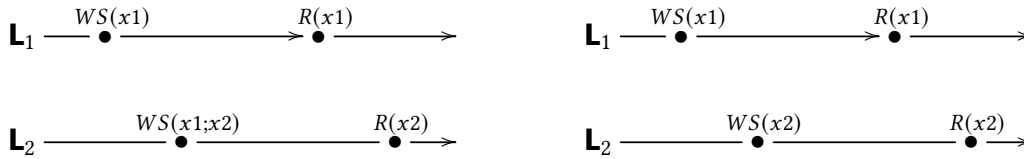


Figure 6: Example of Monotonic Reads

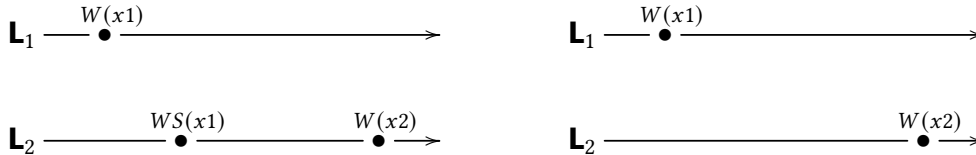


Figure 7: Example of Monotonic Writes

monotonic-read consistency, all operations in $WS(x1)$ should have been propagated to L_2 before the second read operation takes place. In the left diagram, we see that $WS(x1;x2)$ took place meaning all operation that happened at L_1 before $R(x1)$ have been propagated at L_2 before $R(x2)$ and therefore the reads are monotonic. In the right diagram,, no guarantees are given that the set $WS(x2)$ also contains all operations contained in $WS(x1)$. Thus, monotonic-reads consistency is not guaranteed.

2.3.2.2 Monotonic Writes

Monotonic writes is a client-centric consistency model also called session guarantees. It ensures that if a process performs two write operations $w1$, then $w2$ on an item x , then all processes will observe $w1$ before $w2$. Monotonic writes only applies to writes invoked by the same process and does not apply to operations performed by different processes. Monotonic writes can be totally available: even during a network partition, all nodes can make progress.

We see herein Figure 7 an example of a process P with two different local copies L_1 and L_2 . The write operations performed by a single process P at two different local copies of the same data store. Process P performs a write operation on x at local copy L_1 , presented as the operation $W(x1)$. Later, P performs another write operation on x , but this time at L_2 , shown as $W(x2)$. To ensure monotonic-write consistency, the previous write operation at L_1 must have been propagated to L_2 . In the left diagram, operation $W(x1)$ at L_2 takes place before $W(x2)$ and therefore writes are monotonic. In the right diagram, monotonic-write consistency is not guaranteed as the propagation of $W(x1)$ to copy L_2 is missing.

2.3.2.3 Read Your Writes

Read your writes, also known as read my writes, is a client-centric consistency model also called session guarantees. It requires that if the effect of a write operation by a process on data item x will always be seen by a successive read operation on x by the same process. Note that read your writes does not apply to operations performed by different processes. There is no guarantee, for instance, that if a process P



Figure 8: Example of Read Your Writes

writes a value successfully, that a process Q will subsequently observe that write. Read your writes is sticky available: if a network partition occurs, every node can make progress, so long as clients never change which server they talk to.

In Figure 8, Process P performed a write operation $W(x1)$ and later a read operation at a different local copy. Read-your-writes consistency guarantees that the effects of the write operation can be seen by the succeeding read operation. In the left diagram, this is expressed by $WS(x1;x2)$, which states that $W(x1)$ is part of $WS(x2)$. However, in the right diagram, $W(x1)$ has been left out of $WS(x2)$, meaning that the effects of the previous write operation by process P have not been propagated to L_2 .

2.4 Causality tracking and broadcast

Being at the core of this thesis, causality tracking techniques and broadcast are reviewed in this section. I address broadcast/multicast algorithms that are used to guarantee casual delivery. I also list key compaction techniques that were used to reduce the metadata needed for guaranteeing causality. Prior to that, I start by introducing key abstractions for causality tracking techniques. In particular, I start with physical clocks, explaining why we cannot rely on them in geo-replicated systems. Then, I introduce the notion of logical clocks and causality tracking that was introduced by Lamport [23], establishing the foundation for the subsequent mechanisms and theory [23–28]. I conclude this section with a small discussion on single-object and multi-object logical clocks.

2.4.1 Causality tracking primitives

2.4.1.1 Physical Clocks

The ability of timestamping events is crucial in a system to makes sense of the ordering of events. In most applications, some constraints on the order of events must be respected, and thus being able to timestamp those events is important in order to maintain these constraints. A simple way to address this is to rely on physical clocks as they are already present in computing machines and can be used to provide timestamps systems events. When we have a single machine, creating a timeline of events is easy to achieve, even with multiple processes or threads. The reason for this is the fact that all these events will be timestamped by the same physical clock providing a single source and a global view of time.

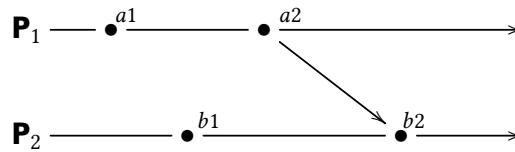


Figure 9: Diagram showing a distributed execution using Lamport's Happens-before

However, in the case of a distributed system, we cannot rely on physical clocks as a solution. This is because different machines have different physical clocks that are prone to diverge, and consequently there will be no guarantees that the timestamps provided by these clocks will give a correct ordering and consistent data versions at all replicas. In other terms, there is no global clock to provide a global view of time anymore. To mitigate the skew between these clocks it is required that these clocks periodically use some synchronization mechanism such as NTP, Coordinated Universal Time UTC, etc. However, this synchronization does not guarantee perfect synchronization as the quality depends on factors such as network load, or the NTP server could be offline, etc. Another issue is that even perfect physical timestamps would provide a total order of events and thus would not capture concurrency between events which could be needed for some applications. Some systems like Cassandra [29] workaround this by tagging each version of an object with a physical timestamp and use a last-writer-wins (LWW) policy as a tie-breaker when clocks are equal.

2.4.1.2 Happens-Before relation

Lamport introduced the *happens-before* relation to capture time between events logically, without the reliance on physical clocks. The happens-before relation characterizes the notion of causality or more accurately potential causality. It translates to the ability or possibility of one event to have influenced another event in its future. Formally, the happens-before relation can be described as follows. Given two events e and e' , we say that e happens-before e' denoted by $e \rightarrow e'$ if one of the following conditions holds true:

- e and e' are two events on the same process p_i where e occurs before e' ;
- e is the send event of a message m from process p_i and e' the receive event of m by process p_j ;
- There exists an event e'' such that $e \rightarrow e''$ and $e'' \rightarrow e'$.

Not all events are causally related by the happens-before relation. Such events are called concurrent events and are denoted by $e \parallel e'$ which is equivalent to $e \not\rightarrow e'$ and $e' \not\rightarrow e$.

In Figure 9, we see two processes P_1 and P_2 . We notice that $a_1 \rightarrow a_2$ and $b_1 \rightarrow b_2$ since a_1 occurred before a_2 on P_1 and b_1 before b_2 on P_2 . This is captured by the first condition. Also $a_2 \rightarrow b_2$ as a_2 is the send event of a message from P_1 and b_2 its relative receive event on P_2 captured by the second condition. Finally, $a_1 \rightarrow b_2$ captured by the transitivity in condition 3. Events a_1 and b_1 are concurrent or $a_1 \parallel b_1$.

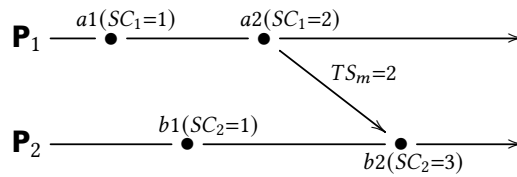


Figure 10: Diagram showing a distributed execution using Lamport clocks

2.4.1.3 Lamport Clocks

Lamport clocks [27] are scalar logical clocks that increase monotonically and used to enforce causal ordering. Each process p_i maintains a scalar clock SC_i which is basically a non-negative monotonically increasing integer counter that starts with $SC_i := 0$. The algorithm used to maintain this clocks is presented as follows. Considering an event e at p_i and SC_i set to 0:

- If event e is an internal event (excluding send and receive events), increment the value of SC_i by 1, $SC_i := SC_i + 1$;
- If event e is a send event, increment the value of SC_i by 1, $SC_i := SC_i + 1$ and send the new $TS_m := SC_i$ along with the message m ;
- If event e is a receive event of message m with TS_m , set $SC_i := \max(SC_i, TS_m) + 1$.

As scalar clocks are integers, the order on timestamps is the total order relation ($<$) among integers. For each process p_i , if event e occurs before event e' then we have $TS_e < TS_{e'}$ because the value of SC_i is strictly increasing. If e was the send event of message m and e' the receive event of the corresponding message, then also we have $TS_e < TS_{e'}$ because based on the third rule $TS_{e'}$ is greater than TS_e at least by 1. Finally, knowing that $<$ is transitive we get that if $e \rightarrow e'$ then $TS_e < TS_{e'}$. However, the converse is not true, i.e., $TS_e < TS_{e'}$ does not necessarily mean that $e \rightarrow e'$. The problem here is that the happens-before relation is a partial order while the scalar timestamps are totally ordered. This does not allow the distinction of concurrent events.

In Figure 10, we see two processes P_1 and P_2 . We notice that at P_1 , $a_1 \rightarrow a_2$ and therefore $TS_{a_1} = 1 < 2 = TS_{a_2}$. Similar observations at P_2 where $b_1 \rightarrow b_2$ and therefore $TS_{b_1} = 1 < 3 = TS_{b_2}$. Also $a_2 \rightarrow b_2$ and therefore $TS_{a_2} = 2 < 3 = \max(1, 2) + 1 = TS_{b_2}$. However, even though $TS_{b_1} = 1 < 2 = TS_{a_2}$ it does not mean that $b_1 \rightarrow a_2$ because in fact $b_1 \parallel a_2$.

2.4.1.4 Causal Histories

Causal Histories [28] are a simple intuitive way to capture causality. Every event has a unique identifier, and the causal history of an event the set of identifiers of events in its causal past. Every process in the system assign a unique identifier for every event, e.g. process/node name and a local increasing counter. Locally, a process P assigns for a new event the identifier $p_i + 1$ where i was the last value of the local

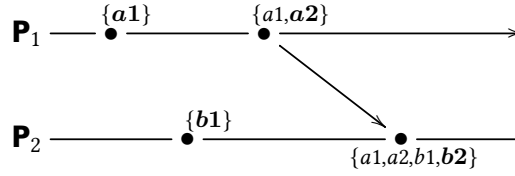


Figure 11: Diagram showing a distributed execution using Causal Histories

increasing counter (starting from 0 initially). When a process P sends a message, a new id is assigned for the send event and the causal history at P until that message is sent along with it. Upon receiving the message at process Q , a new id is assigned for the receive event and the causal history at Q is merged with the causal history in the message. The crucial point is that identifiers have to be globally unique to correctly represent causality. The partial order of causality can be precisely tracked by comparing these sets under set inclusion. In this case, an event e happens-before e' ($e \rightarrow e'$) is equivalent to the Causal History of e being included in the Causal History of e' ($H_e \subseteq H_{e'}$). In the same logic, $e \parallel e'$ if and only if $H_e \subseteq H_{e'}$ and $H_{e'} \subseteq H_e$. Marking the last local event added to the history (marked in bold) allows for a simpler test: $e \rightarrow e'$ if and only if $e \in H_{e'}$. Causal Histories correctly track causality relations, but have a major drawback: they grow linearly with the number of writes.

In Figure 11, we see two processes P_1 and P_2 . We notice that the second event at P_1 is assigned the id $a2$ and has as causal history $a1, a2$. The causal history $a1, a2$ is sent with message to be received at P_2 . On receiving, P_2 assigns the id $b2$ with causal history $a1, a2, b1, b2 = a1, a2 \cup b1 \cup b2$. We can clearly notice that $a1 \rightarrow b2$ since $a1 \in a1, a2, b1, b2 = H_{b2}$.

2.4.1.5 Vector Clocks

While causal histories can capture causality, they have a drawback: their size grows linearly with the number of events and therefore are not very compact. Vector Clocks [30, 31] address this problem by offering a more compact representation of causal histories. As we observed earlier, if a causal history H includes an event e occurring at process p then it also includes all events occurring at p and that precede e . Thus it suffices to store the latest event from every process and consequently reduce the size of these causal histories to the number of processes.

For instance, the causal history $H = a1, a2, b1, b2$ can be represented as $a \rightarrow 2, b \rightarrow 2$ or simply the vector $[2, 2]$. In causal histories, $e \rightarrow e'$ iff $H_e \subseteq H_{e'}$, which informally means that every event in H_e is also in $H_{e'}$ and there is at least one event in $H_{e'}$ which is not in H_e . In terms of vector clocks, this translates to checking if every entry $VC_e[i] \leq VC_{e'}[i]$ and there exists at least one entry $j \neq i$ for which $VC_e[j] < VC_{e'}[j]$, or more compactly: $e \rightarrow e'$ iff $VC_e < VC_{e'}$.

Furthermore, Lamport clocks or scalar clocks that $e \rightarrow e'$ means $TS_e < TS_{e'}$ but the converse is not true. This does not allow us to capture concurrency because timestamps are totally ordered. Vector Clocks do not face this problem as the order is a partial order and we can distinguish causally related events from concurrent ones. And by that we get the following:

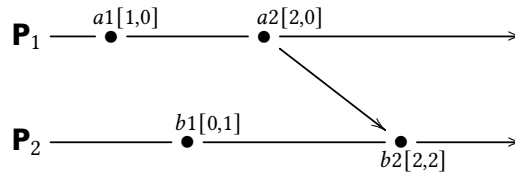


Figure 12: Diagram showing a distributed execution using vector clocks

- $VC_e < VC_{e'}$ iff $e \rightarrow e'$
- $e \parallel e'$ iff $VC_e \not\leq VC_{e'} \wedge VC_{e'} \not\leq VC_e$

The Vector clock mechanism works as follows. Each process p_i maintains a local vector clock VC_i , a vector of n scalars, where each entry is initially set to $VC_i[k] := 0$. Considering an event e at p_i :

- If event e is an internal event (excluding send and receive events), increment the value of $VC_i[i]$ by 1, $VC_i[i] := VC_i[i] + 1$;
- If event e is a send event, increment the value of $VC_i[i]$ by 1, $VC_i[i] := VC_i[i] + 1$ and send the new $TS_m := VC_i$ along with the message m ;
- If event e is a receive event of message m with TS_m , set every entry in VC_i to pair-wise maximum of VC_i and TS_m as in $VC_i := \max(VC_i[k], TS_m[k]), \forall k | 1 < k < n$, and then increment the local entry by one $VC_i[i] := VC_i[i] + 1$.

In Figure 12, we see two processes P_1 and P_2 . We notice that at P_1 , $a_1 \rightarrow a_2$ and also $TS_{a1} = [1, 0] < [2, 0] = TS_{a2}$. Similar observations at P_2 where $b_1 \rightarrow b_2$ and also $TS_{b1} = [0, 1] < [2, 2] = TS_{b2}$. Also $a_2 \rightarrow b_2$ and also $TS_{a2} = [2, 0] < [2, 2] = TS_{b2}$. Now when we try to compare b_1 and a_2 , we notice that $TS_{b1} \not\leq TS_{a2}$ and $TS_{a2} \not\leq TS_{b1}$ which means that $b_1 \parallel a_2$.

2.4.1.6 Version Vectors

Parker et al. [26] introduced versions vectors as a causality tracking mechanism that share the same structure as vector clocks, whereas the update differs. The reason is that the main purpose of using version vectors is to detect inconsistencies among replicas. Basically each replica has a version vector that tracks the last updates known from other replicas as well as from itself. Version Vectors do not need to keep track of every event in the distributed computation, but exclusively updates that alter the replica state. Two main operations are relevant in this case, the update of replicas state and the synchronization between two replicas leading to a convergent state. Therefore the mechanism of maintaining version vectors is different than the ones of vector clocks. Considering a system of n replicas, version vectors are updates as follows:

- When a replica r_i is updated, it increments the local entry of its $VV_i[i]$ by 1, $VV_i[i] := VV_i[i] + 1$;

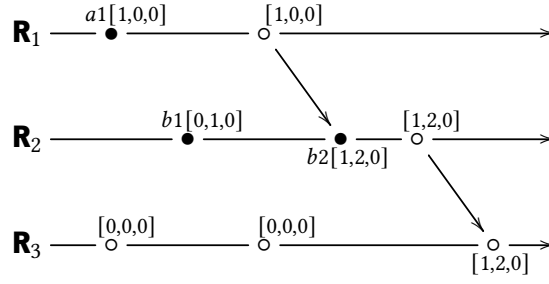


Figure 13: Diagram showing a distributed execution using version vectors

- When replica r_i synchronizes with a replica r_j , both versions vectors are updated such as $VV_i = VV_j := \max(VV_i[k], VV_j[k]), \forall k | 1 < k < n$

In Figure 13, we see three replicas R_1 , R_2 and R_3 . The \bullet represents replica updates that we want to causally track whereas \circ is used to denote events that are not considered for causality tracking as they do not alter the state of a replica. Replica R_1 suffers an update $a1$ and concurrently to that R_2 suffers another update $b1$. When a message with the update $a1$ from R_1 reaches R_2 , two concurrent updates are detected. A new update $b2$ merging the two updates $a1$ and $b1$ is created at R_2 leading to a new state. When the state of R_2 is propagated to R_3 , no concurrent updates are detected and therefore no new version is created and $VV_{R_2} = VV_{R_3} = [1, 2, 0]$.

2.4.1.7 Matrix Clocks

Matrix clocks are also a type of logical clocks that give the local node a global view of what every node in the system knows about other nodes. They are seen as extensions of vector clocks, a vector of vector clocks. When a message m from process i is timestamped with a vector clock, it provides information on how many messages were seen by i from every node prior to send message m . When the same happens using a Matrix clock as a timestamp, the information encoded provides the knowledge that process i has of what every node k has seen from others until the message m was sent. Formally, time is represented by $n \times n$ matrices of non-negative integers, where n is the number of nodes or processes in the distributed computation. A process p_i maintains $MC_i \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nn} \end{pmatrix}$

- $MC_i[i, i]$ denotes the local logical clock of p_i and tracks the progress of the computation at process p_i
- $MC_i[i, j]$ denotes the latest knowledge that process p_i has about the local logical clock, $MC_j[j, j]$, of process p_j
- $MC_i[j, k]$ represents the knowledge that process p_i has about the latest knowledge that p_j has about the local logical clock, $MC_k[k, k]$, of p_k
- The entire matrix MC_i denotes p_i 's local view of the global logical time.

To update its clocks, process p_i uses the following rules:

- If event e is an internal event (excluding send and receive events), increment the value of $MC_i[i, i]$ by 1, $MC_i[i, i] := MC_i[i, i] + 1$;
- If event e is a send event from p_i to p_j , increment the values of $MC_i[i, i]$ and $MC_i[i, j]$ by 1, $MC_i[i, i] := MC_i[i, i] + 1$ and $MC_i[i, j] := MC_i[i, j] + 1$ and send the new $TS_m := MC_i$ along with the message m ;
- If event e is a receive event of message m from p_j on p_i with TS_m , update MC_i as follows: $\forall k, l | 1 \leq k, l \leq n, MC_i[i, k] := \max(MC_i[i, k], TS_m[j, k])$ and $MC_i[k, l] := \max(MC_i[k, l], TS_m[k, l])$ and finally $MC_i[i, i] := MC_i[i, i] + 1$.

2.4.2 Causal Broadcast

Causal broadcast is a main group communication delivery technique used in causality middlewares. It makes use of the aforementioned causality primitives to provide a causal ordering of events between a group of processes, following some algorithms. The point of a causal broadcast [24] algorithm is to provide rules on how to use timing primitives, mainly logical clocks, to correctly order events, following the happens-before relation, also known as potential causality, among processes in a group communication environment.

In group communication, we consider a number of processes forming a group, where these processes communicate with each other through the sending and receiving of messages. A group could be a *closed*, where a process should be part of the group to be able to send a message to the group, or an *open* where this constraint does not need to be satisfied. Another thing to consider in a group is if the group is *static*, where the membership of that group is fixed, or if it is *dynamic*, where new processes can join and existing ones can leave.

In this section, I will consider closed, static groups for simplicity, and to be able to focus on the causal ordering aspect of the communication. To focus on the ordering, I also assume that the communication is reliable, meaning:

- No message is delivered more than once
- No message is delivered unless it was broadcast
- If p_i and p_j are correct, then every message broadcast by p_i is eventually delivered by p_j
- If one correct process delivers a message m , every correct process eventually delivers m

Remark. Message delivery is different from message receiving. We use the term receive to note that a message is received at a certain process via the communication layer, while delivering a message is

meant to describe that an already received message can be delivered to the application layer to be used or executed. This differentiation is important when discussing the ordering of events.

2.4.2.1 Causal ordering specifications

In a broadcast event, a process p_i sends a message m to all other processes in the system. An event $e = bcast(m)$ causally comes ahead another event $e' = bcast(m')$ if at least one of following condition is true:

- e and e' have been produced by the same process p_i and $bcast(m)$ happens before $bcast(m')$
- e and e' have been produced by different processes p_i and p_j respectively and the delivery of message m precedes e' at p_j
- $\exists m'' | bcast(m) \rightarrow bcast(m'') \wedge bcast(m'') \rightarrow bcast(m')$

The safety guarantee states that: Given two broadcast messages m and m' such that $bcast(m) \rightarrow bcast(m')$ then each process have to deliver m before m' ($deliver(m) \rightarrow deliver(m')$). The liveness guarantee states that: eventually every message will be delivered. Consequently, if 2 messages m and m' are such that $bcast(m) || bcast(m')$ then m and m' can be delivered in different order on the other processes.

2.4.2.2 Causal Broadcast (CBCAST) algorithm

Algorithm 1 presents an example of a causal broadcast (CBCAST) implementation. The algorithm assumes the following:

- Each process step takes a finite time to occur
- Message transfer delays are unpredictable but finite
- Communication channels are reliable
- Computation is failure free

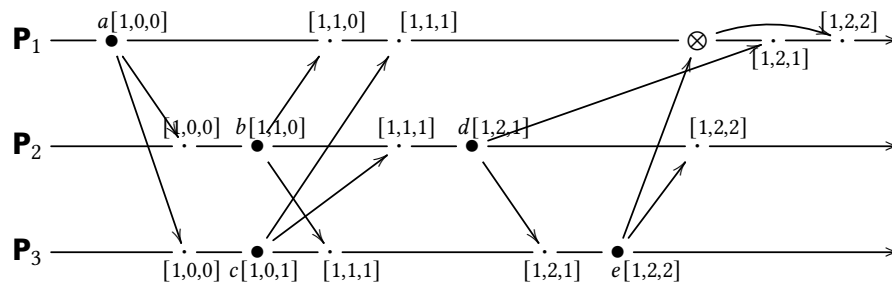


Figure 14: Diagram showing a distributed execution using the CBCAST algorithm

ALGORITHM 1: CBCAST algorithm using Vector clocks on process $i \in \mathbb{I}$

```

1 state:
2 |  $VC_i : \mathbb{I} \leftrightarrow \mathbb{N}$ , local vector clock
3 proc  $init_i()$ 
4 |  $VC_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
5 proc  $cbcast_i(m)$ 
6 |  $VC_i[i] := VC_i[i] + 1$ 
7 | for  $j \in \mathbb{I}, j \neq i$  do
8 | |  $send_i^j(m, VC_i)$ 
9 on  $receive_i^j(m, TS_m)$ 
10 | if  $TS_m[j] = VC_i[j] + 1 \wedge \forall k \neq j, TS_m[k] \leq VC_i[k]$ 
11 | |  $VC_i[j] := VC_i[j] + 1$ 
12 | |  $deliver(m)$ 
13 else
14 | |  $enqueue(m, TS_m)$ 
    
```

I explain the CBCAST algorithm through presenting the following example execution:

In the example of Figure 14, we notice three processes P_1 , P_2 and P_3 participating in a distributed computation and using the CBCAST algorithm to causally order events. First, P_1 broadcasts a with $VC_{P_1} = [1, 0, 0]$, and we noticed that it was delivered upon being received at P_2 and P_3 as it satisfies the condition in algorithm 1 line 10. Events b and c are concurrent and that is reflected in their timestamps ($[1, 1, 0] \parallel [1, 0, 1]$) and therefore they can be delivered in different order without breaking causality. d with timestamp $[1, 2, 1]$ is causally in the future of events a , b and c . It was delivered by P_3 and then the event e with timestamp $[1, 2, 2]$ is broadcast to P_1 and P_2 . Clearly $d \rightarrow e$ as $[1, 2, 1] \leq [1, 2, 2]$ and therefore d must be delivered before e on all processes. However, because some message delay, e is received on P_1 before d . When e is received on P_1 it cannot be delivered as it breaks the causality conditions in algorithm 1 line 10. We notice the \otimes symbol to denote not delivering e and delaying its delivery until after delivering message d .

2.4.3 Scalability solutions for Causal Multicast

Causal multicast provides an advantageous basis for scalable communication. However, a single communication round uses messages that should carry some kind of causal history and the amount of such information depends on the system size. As a result, in order to multicast messages to as many servers as possible, some mechanisms should be devised to reduce the overall size of that causal history. Here I discuss some scalability solutions for causal multicast, and present the advantages and drawbacks of each of them.

2.4.3.1 First generation

The first generation of causal multicast protocols used to record causal dependencies among events in way similar to Lamport clocks [23]. However, instead of adding a logical clock value to the new messages being transmitted, those protocols preferred to re-transmitting the causally precedent messages. This means that when sending a new message, the sender also sends previously received messages as the causal history. When receiving the message, the receiver process checks the causal history of the new message, and proceeds to deliver each of these messages if it has not delivered it yet, and in the same order they were sent. Once all those are delivered, the new message can be delivered.

Causal order is trivially ensured in first generation protocols since the messages included in the causal dependencies of a new multicast message are ordered appropriately on send, and delivered in the same order if needed on receive. Another advantage in this group of protocols is the *non-blocking* behaviour on delivery. As all messages on which the new multicast messages are sent, the receiver doesn't need to wait for other messages to deliver the current one. Blocking and waiting for dependencies to arrive is not needed here. However, preserving causal order and benefiting from the non-blocking behaviour that results from the inclusion of precedent messages comes at a price: the size of meta data for each message could be very large when the message sending rate and number of processes in the group increase. This major drawback affects scalability and explains the reason why this approach is abandoned for other approaches that we visit next.

The first causal multicast protocol was introduced by the CBCAST protocol [24] by Birman and Joseph in 1987. (The implementation is however different from the one described in Algorithm 1.) Birman's implementation had the advantages and drawbacks we discussed above. Some subsequent designs built on top of CBCAST were later introduced with a few modifications. Instead of sending the entire message for the causal predecessors in the causal dependencies, these protocols included only logical identifiers of the precedent messages. Psync's conversations [32] introduced a first solution of this kind in 1989.

A similar solution was developed by Ladin et al. [33], called lazy replication in 1992. Obviously, the size of meta data consisting of the logical identifiers of causal predecessors is reduced compared to sending the entirety of messages (including their contents). This definitely addresses the drawback of the first generation protocols, but on the other hand leads to the loss of the non-blocking feature. At delivery, if one of the predecessors was not already delivered at the receiving process, the new multicast message

cannot be delivered and therefore delivery must be blocked temporarily until all predecessors have been delivered. These new identifier-based blocking solutions were generalised with the introduction of vector clocks that we address next.

2.4.3.2 Second generation exploiting vector clocks

The second wave of causal multicast protocols targeted the bandwidth utilization issue of causal histories through using vector clocks in different ways. In 1991, Birman et al. [24] developed the first vector-clock-based protocols which turned out to be efficient and more scalable than the previous generations ones (Schiper et al. [34] and Raynal et al. [35] presented similar approaches for point-to-point communication). This kind of causal multicast protocol is the most adopted in GCSs (group communication systems) such as Isis [24], Transis [36], Spread [37], JGroups [38], etc.

As presented in subsection 2.4.1.5, the use of vector clocks makes it easier to track causal dependencies among multicast messages without the need to include the causally precedent messages in each multicast. This reduces the metadata needed to track causality and provides a more compact size that improves scalability.

Nevertheless, the use of vector clocks in these protocols comes at a price: since the contents of the causal dependencies are not sent along, the non-blocking benefits of the first generation protocols are lost. For instance, when a message is dropped or when the propagation delays lead to the reception of a message m_2 before the reception of one of its causal predecessors m_1 at process p , the process p has to delay the delivery of m_2 until m_1 (and every other missing predecessors of m_2) is re-sent, received and delivered. Another drawback is that, even though the size has been reduced, a vector clock's size increases linearly with the number of processes, and logarithmically with the values held in each entry. As multicast services are usually run for a long time, vector clocks will hold long integers. Moreover, when the number of processes is large, the number of entries becomes large. In a system of n processes, where each vector entry has b bits of size, the size of a vector clock will be of $(n \times b/8)$ bytes. If we consider a large system of 200 processes and 32-bit entries, there is an 800 bytes of meta data per message. This is a non-negligible communication overhead for a scalable service. However, replicated services using vector clocks rarely consist of such a large number of replicas or when needed they could use sharding [39] to partition the data among a small subset of processes so replication factor stays relatively small.

Bounded Version Vectors. Bounded Version Vectors was proposed by Almeida et al. [40] in 2004 as a mechanism to bound the size of vector entries in version vectors in point-to-point communication. The mechanism combines the transmission of version vectors as well as some historical meta data that will be used, at each node, to prune the history maintained in every version vector component. Due to this, the values of version vector entries can be bounded and re-used (for a system of N nodes, the upper bound can be set to N^2).

Dynamic membership. Another challenge in vector clocks is when considering dynamic memberships. When new processes join or existing ones leave, this leading to a change in system's membership,

and thus vector clocks need to be adjusted to reflect the changes. This adjustment of vector clocks is usually hard to do without blocking the communication temporarily, which affects the availability of the systems especially when membership changes occur frequently. To the contrary, temporary re-initialization of clocks could have some positive consequences by limiting the unbounded growth of vector clocks though bounding the size of the entries. Therefore, vector clocks should be readjusted for dynamic memberships. In 2008, Almeida et al. proposed an elegant solution [41] called ITCs (Interval Tree Clocks). ITCs allow the reconfiguration of clocks using local information in dynamic systems.

With this, I introduce in next section the compaction approach which present similar compaction approaches of vector clocks.

2.4.3.3 Vector compaction approach

The vector compaction approach aims to improve system's scalability by minimize the size of vector clocks. The main idea here is to compact the vector size by reducing the amount of information sent to only the necessary and sufficient information needed. This is done by sending only the entries that have changed since the last sent message. I will discuss two different mechanisms: the general causal communication approach (using point-to-point communication) described in [42], and a more specific but simpler approach for multicast-based approach described in [24].

The **general approach** [42] is often used in point-to-point communication. Every process p maintains, along with its local vector clock VC_p , two additional vector clocks, LU_p and LS_p . LU_p is the last update vector clock such that $LU_p[q]$ holds the value of $VC_p[p]$ when p last updated its entry $VC_p[q]$. LS_p is the last sent vector clock such that $LS_p[q]$ holds the value of $VC_p[p]$ when p sent its last message to q . When p decides to send a message to q it only needs to propagate the sequence of pairs $\langle r, VC_p[r] \rangle$ that have been updated since the last message and such that $LS_p[q] < LU_p[r], r \neq q$.

For the **multicast-based approach** [24], the optimization is more trivial. As every process will receive every message, a process p will not need to maintain the LS_p vector clock since all its entries will have the value $VC_p[p] - 1$. As for the last update vector clock, an $LU_p[q]$ entry will hold the value $VC_p[p]$ when the last message from q was delivered at p . Note that, instead of maintaining such LU_p , p only needs to keep the value of its VC_p when it multicast its last message which we call LM_p , and only propagate the entries where $LM_p[r] < VC_p[r]$ for $r \neq p$.

This multicast-based compaction technique is mostly advantageous when not all members have the same sending rate and most of them are passive and hardly send any messages. This would allow the active processes that are sending messages to compact their vector clocks a lot compared to the original full size of the vector clocks. Consider a group membership of n processes, where b is the number of bits needed to hold a sequence number (the value of a vector clock entry), m is the number of bits needed for a process identifier and k is the number of entries needed to be propagated at every multicast. This compaction technique would be considered useful when $k < n \times b$ is satisfied. $\frac{m+b}{n \times b}$ is the full size of a non-compacted vector clock, and $m+b$ is the size of a single compacted entry that needs to be propagated.

Chandra et al. have proven in [43] that the above expression holds in most systems. Moreover, the performance analysis in [43] (where only systems with less than 100 processes are considered) shows that this technique does not only improve scalability but also saves on bandwidth even for systems with a not so large membership. The price to pay for this technique is that it requires maintaining an additional copy of the vector clock. Nevertheless, the price is minimal and negligible knowing that the same amount of memory is needed when appending the whole vector clock to the message in the non-compaction approach. In addition to that, the computing overhead for managing the compacted vector clocks depends on the sending rate of processes in the group and is not considered a big cost compared to the benefits, especially in the multicast-based approach.

Other related works. As already mentioned, the first solution was presented by Birman et al. [24] and Stephenson [44]. More solutions have built upon of this early one. Inspired by [24], Singhal and Khemkalyani proposed a more general solution in [42]. It was later refined and formalized by Prakash et al [45] and Khemkalyani and Singhal [46]. Chandra et al. [43] evaluated the the degree of compaction of these solutions showing that in an optimal scenario the size of a vector clock can be reduced to 8% of its original size. Pomares Hernandez et al. [47] proposed the propagation of the compacted information through light control messages (LCMs) which showed to improve scalability in their performance evaluation. Finally, Mostefaoui and Raynal [48] used the particular case of multiple groups with mutual processes to propose another solution. It is a cheaper solution than the one in [24, 44], but sometimes requires synchronization messages. This method is seen as the precursor of the interconnection approach that we present next.

2.4.3.4 Scaling out approaches

The interconnection approach [49, 50] proposes an interesting solution to scale causal mutlicast-based systems. In such scenario, there exist multiple subgroups where each have an internal causal multicast protocol. The different subgroups do not have to use the same internal causal multicast service. At least one process in each subgroup is also an interconnection server (IS). Every IS runs an interconnection protocol that allows the different IS of the existing subgroups to deliver the messages delivered in the internal subgroups. Informally, an IS ensures that the messages multicast in its local subgroup are delivered in all other subgroups and analogously, all the messages multicast in remote subgroups are delivered in their local subgroups.

The fact that a process of a subgroup is an IS is transparent to the subgroup, and does not interfere with the internal multicast. Causal separators [51] isolate each subgroup as a different causal zone and provide the basis to implement such protocols. Rodrigues and Verissimo [52] proved that forwarding causal history to other subgroups is not needed. In fact, it only suffices that the interconnection protocol uses an independent causal multicast protocol between the interconnection servers [53], or a FIFO transmission of messages [54], when the interconnected servers are paired. This would guarantee a causal order when only two processes are involved.

As every subgroup in this approach could be considered as its own causal zone, the scope of causal history needed to be propagated is limited to the subgroup itself. This allows keeping the causal history smaller and relative to the size of the subgroup instead of depending on the overall size of the system and therefore improves the scalability of causal multicast systems using this approach. In a generic scenario where only a single causal multicast protocol is used among all processes, every message is multicast to all other processes through point-to-point communication. Using the knowledge of the topology of the underlying network, the interconnection approach reduced the number of packets needed to be sent as the interconnection protocol requires only a single message to be communicated to other IS for every multicast. Therefore the interconnection approach improves the performance by reducing the communication overhead of the system. Finally, this approach helps with dynamic memberships. As every subgroup has its internal causal multicast protocol, it is not affected with the global system membership. This becomes increasingly important in environment with high churn [55], where the interconnection approach can ensure a good level of scalability.

On the negative side, as an interconnection server requires a short forwarding step, some authors (e.g., [47]) argue that this could introduce a delay that might not be affordable by all kinds of applications. For instance, media stream applications might require specific, more flexible causal multicast protocols that can tolerate message losses in order to meet real-time constraints [56–58]. Although not specific to interconnected subgroups, the convoy phenomenon which could arise in any kind of overlapping groups, can introduce delays that generate a bursty propagation behaviour. This was reported by Kalantar and Birman in [59]. When message propagation between IS due to temporary message loss or some delay in causal delivery, some subsequent messages get blocked. Therefore, if messages need to cross multiple subgroups in order to get delivered, the delay of the propagation generates further delays for all servers of that path, leading to the convoy phenomenon.

A combination of using the previously visited first generation protocols that include precedent messages, and having small subgroups where the causal history of each multicast would be relatively small, could solve this problem that is commonly caused by the loss of precedent messages.

2.5 Weakly-consistent data stores

In the previous sections of this chapter, we presented the different trade-offs faced by system designers and architects when designing distributed systems. Defining these trade-offs is an important step in the complex process of system design as it allows the system designer to prioritize some properties and make the correct compromises that would be more relevant and suitable for the type of systems they are designing and the class of applications can use them. As we are interested in this thesis with highly-available data stores, we focused on trade-offs and consistency models that suit most this class of applications, namely causal+ consistency, the strongest achievable consistency that remains available under network

partitions, and that combines both causal consistency and strong convergence. Maintaining causal relationships among events is an expected behavior for most applications and reflects a behavior aligned with our perception of reality. In addition, as we are interested in data stores, data convergence is an important property as users expect that the replicas converge to equivalent states despite if messages are concurrent or not and no matter what happens under the hood. We also visited the different mechanisms and protocols used in the literature to track and guarantee a causal order of events and messages across replicas. But all these properties, trade-offs, algorithms and protocols are as useful as the systems they help design and the range of applications they serve. In this section, we address some techniques used in designing and optimizing causally consistent data-stores and review the most known systems that advanced the literature and made an impact in the complex world of distributed systems.

2.5.1 Causally Consistency Data Stores

We will briefly describe some of the most well known causally consistent systems in the literature.

2.5.1.1 COPS

In [13] Lloyd et al. present COPS and its variant COPS-GT as causally consistent geo-replicated data stores. COPS stands for Clusters of Order Preserving Systems and uses causal+ consistency to preserve a causal order and a convergent state across its datacenters (DC). COPS-GT also provides get transactions (GT) to the client. The implementation of COPS a small number of clusters, where each is contained in a datacenter. Each datacenter holds a copy of the whole data partitioning the keyset into N linearizable partitions. Each datacenter serves requests locally and then replicates to other DCs asynchronously in the background. The inter-DC replication is causally consistent. Each update is assigned a Lamport clock as a unique version number, and a list of explicit causal dependencies is used as its causal history. Clients are in charge of maintaining their causal history, which is called a context. When a client reads a new object or a newer value of an existing object, they add the object version to its context. When a client wants to update an object, they look at its context which holds a list of dependencies that would consist of the causal history of the new update. To reduce the overhead of causal metadata, COPS offers optimizations by sending less metadata using a transitive reduction on the causal dependencies and sending the “nearest dependencies”. After that, the client sends a message containing the object’s key, the new value and the nearest dependencies to the local server. The server has all the dependencies as the client keeps a session with it and so the server can immediately apply the update, tag it, and return the new timestamp to the client which replaces the value of its context with it. The replication protocol for remote updates follows similar steps. However, as some dependencies might not have been delivered at the remote server, every new update received is blocked at the remote server until all its causal dependencies are delivered. And only after that check the update becomes visible. For concurrent updates, COPS uses a Last Writer Wins (LWW) strategy which is a simple approach but one that is prone to data loss and errors. Also note that client can have different contexts for the same server connection. And as dependencies are tracked per

client context, independent operations can have less common dependencies which as a result reduces the time for remote updates to be visible (visibility latency). Moreover, COPS offers a garbage collection mechanism to reduce the local causal metadata that becomes obsolete.

2.5.1.2 Eiger

Eiger [60] is a scalable, geo-replicated storage system which could be seen as a successor of COPS. It supports causal+ consistency but uses a column family data model (popularized by Cassandra [17]) instead of a key-value model as most systems. It supports both read-only and write-only transactions and uses operations instead of data versions as dependencies. Write transactions are achieved with a modified two-phase commit protocol that functions differently depending if it is running on the DC that accepted the operation initially or in a remote DC.

2.5.1.3 Orbe

Orbe [61] offers causal-consistency by using data versions as dependencies like COPS, but instead of Lamport clocks it uses version vectors that are encoded in a sparse matrix. It has an extension to the protocol that allows for read transactions that added physical timestamps to the dependency matrix. It does not support partial replication or concurrent writes with a multi-value API. Client sessions are also bound to the DC they are running. It also suffers from the same problem as COPS and Eiger regarding data replication, since it needs to apply updates in causal order, thus potentially blocking replication until dependencies arrive. Similarly to COPS, Orbe [61] ensure a causal consistency by using data versions as dependencies. However, instead of using Lamport clocks, it uses a protocol based on dependency matrices (DM). Each datacenter is divided into N partitions, and fully replicated at M different datacenters. Clients only contacts servers in the same datacenters (sticky availability) and are in charge of maintaining causal metadata to track dependencies in the session. A client stores the nearest dependencies of its session in a $N \times M$ matrix. To read an object the client send a get request with the key of the object. The server looks up the object and return the latest value, an update timestamp and the source replica index. When the client receives the reply, if the entry pointed by the replica index has a smaller value than the returned timestamp, it overwrites that position with the newer value. To update an object, a client sends the key, the new value and its dependency matrix to the local server responsible for the partition in question. Before tagging the update clock time, the server has to increment its local clock, guaranteeing that the assigned value is unique. After storing both the objects value and metadata in an atomic, non-blocking manner, the server replies with the updates clock and its index. Similarly to COPS, the client resets its causal history (dependency matrix) to default values, only saving the returned clock at the given index. Note that the matrix sent by the client in the update request is used for inter-DC replication. For inter-DC replication, each partition in a datacenter communicates updates with its replicas in remote datacenters. The partition forward the updates in their creation order along with the saved metadata comprising the key, value, creation clock, dependency matrix and replica index. Each partition maintains

their own version vectors with one entry per replica. To check if remote updates can be safely applied, the receiving server compares the dependency matrix of this updates with their version vector. To be able to safely apply the update, the server must do the following checks: For the same partition, the server checks the corresponding row in the matrix, which has a vector format. If the server's version vector is larger or equal than the vector (row) of the matrix, the server can proceed to the next step. For other partitions, every time there is a value different from the default one in the partition column, it means that the operation depends on other elements that the current datacenter may not store. So, the server contacts the respective partition, making an explicit dependency check. If all checks pass, the update can safely be made visible and the corresponding version vector's entry is updated.

2.5.1.4 GentleRain

GentleRain [62] is a causally consistent geo-replicated data store that could be considered as the successor to the Orbe system and by the same authors. The main difference with Orbe is that it uses scalar timestamps derived from loosely synchronized physical clocks and only keeps a single scalar to track dependencies and enforce casual consistency. This also means that the timestamps have a fixed size that is not affected by the number of objects or replicas. This reduces the storage and communication overhead but also increases the update visibility latency. Each datacenter is divided into N partitions, and fully replicated at M different datacenters. The data store is a multiversion one, so older versions of an object are kept when the object is updated, until they can be deleted by the garbage collecting mechanism. Clients maintain two timestamps each: a dependency time (DT) which holds the highest update timestamp for all object accessed by the client session, and a global stable time (GST) that will be used to decide the visibility of remote updates. Servers maintain three timestamps each: a version vector (VV), a local stable time (LST) and a global stable time (GST). The VV is a vector of M timestamps, one entry per datacenter. The LST of a server is the smallest update timestamp (minimum entry value in its VV). The GST of a server is the lower bound on minimum LST of all partitions in the same datacenter. To read an object, a client sends a get request to the local server including the key of the object to read and its GST. The server updates its GST if it was smaller than the client's. The server fetched the latest version of the object for that key if it was created at the local datacenter, otherwise its update timestamp has to be no greater than the server's GST. The server returns the fetched value, the update timestamp and the server's GST. The client updates its GST if the server's one was larger and sets its DT to the returned update timestamp if the latter was larger. To update an object, a client sends a put request to the local server including the object's key and new value and its DT. The server must guarantee that the update has a larger timestamp than the client dependency time which may result in blocking the operation until its physical clock time is larger. This new update timestamp replaces the current datacenter's entry in the server's VV. The server returns the update timestamp to the client to update its DT. For inter-DC replication, the receiving server updates the remote datacenter's entry of its VV with the update's timestamp. The update is visible only when its timestamp is lower than the server's GST. The GST calculation happens periodically inside every

DC and is computed by as the minimum of all partitions' LSTs. As this computation happens periodically and it would be costly to partitions broadcast their LSTs, GentleRain uses a tree structure to disseminate the LSTs (from leaves to root) and then re-disseminate the GST back (from root to leaves) more efficiently. GentleRain uses GST to decide when certain objects are safe to be visible and to garbage collect metadata. Therefore, clock skew resulting from the use of physical timestamps should be kept to a minimum since it affects GST.

2.5.1.5 Cure

Cure [63] is a distributed causally consistent key-value store. Although this is not the focus of this chapter or thesis in general, Cure was the first system to allow causally consistent transactions for both object reads and updates. Prior systems used to allow either one or the other. Each datacenter is divided into N non-overlapping partitions, and fully replicated at D different datacenters. The causal dependencies are maintains through the use of vector timestamps with one entry per datacenter and derived from loosely synchronized physical clocks. Each partition maintains two vector clocks. The first (PVC) tracks remote updates received from the replicated partitions at remote DCs. The second maintains the latest globally stable snapshot (GSS) known to the partition. PVCs hold the values of commit timestamps derived from physical clocks. They are updated when a local or remote update is received. The GSS is calculated by the partitions at a datacenter by exchanging their PVCs and computing the lower bound minimum. When a client wants to start a transaction, it must first contact a coordinator of local datacenter to obtain a transaction identifier. The coordinator in a local datacenter is any partition participating in the transaction and it is responsible for committing the client transaction. This transaction identifier is to guarantee the client future reads during the transaction only include objects having version with a lower timestamp that the transaction identifier. As for the client writes, they are buffered locally until the commit phase. During the commit phase, the client sends the buffered writes to the coordinator. The coordinator then contacts all the source replicas of objects included in the transaction. Each of the contacted replicas send their physical clocks back to the coordinator which chooses the maximum as the commit timestamp and propagates it back to inform the replicas. The transaction effects will be visible t the client since the dependencies are locally satisfied. After a successful commit, the updates are propagated to remote replicas asynchronously. Inter-DC replication is a pair-wise process. The receiving server updates the sender's DC entry in its PVC to the update timestamp. Partitions in the same DC exchange timestamps to calculate a GSS. Remote updates with timestamps lower or equal than the GSS can be made visible.

2.5.1.6 ChainReaction

ChainReaction [64] is a geo-distributed key-value datastore that, as the name hints, uses (a variant of) the chain replication [65] technique to provide causal+ consistency using minimal metadata. A datacenter consists of data servers and client proxies. Data servers serve read and write operations for a number of data items. Client proxies receive the requests from end users or client applications and forward them to

the appropriate data server. Data servers are organized in a DHT ring fashion and use consistent hashing to partition data items. Also every data item is replicated on R consecutive servers, forming a replication chain. The head of chain receives write requests, the updates are propagated down until reaching the tail. If the tail has received an update then it is safe to assume that every server in the chain has received it too. To track causal history, clients maintain a table, where every entry is per object accessed by that client. Each of these entries contain the object's key, its version's timestamp and the chain index which is an index to the node in the chain that last processed and replied to a request of this object. When a client sends a read request, any node between the head of the chain and the node at the chain index can server the request without waiting for a remote update. When a client sends an update request, it sends the object's key, new value and a compression of the metadata of all objects accessed since the last update. The client proxy forwards the request to the head of the chain. The head of the chain increments its replica's entry in the timestamp and assigns it to the new object's version. The update is then propagated down the chain. The update is k -stable when it is replicated on k nodes in the chain and then return to the client the object version and the index of the last receiving node in the chain. The update continues to be propagated down the chain until reaching the tail. When it does, the update is DC-Write-Stable for that replica. Updates are delayed until every object it depends on is DC-Write-Stable, preventing clients from reading inconsistent versions. Moreover, DC-Write-Stable updates need not to be kept in the client's table. An update is globally stable when it is DC-Write-Stable on all chain replicas in remote DCs. For inter-DC replication on the update timestamp is needed and remote-proxies are the responsible entities of exchanging these updates across DCs. When a remote update arrives, the receiving remote-proxy compares the updates timestamp to its own. The latter makes sure that the entry value of the sending DC is one greater than its own and equal or higher values for all other entries in its timestamp vector. This means that the dependencies are stable locally and the update can be applied. Otherwise, the update is delayed until the conditions are met.

2.5.1.7 Saturn

Saturn [66] is not a data store itself but was designed as a metadata service for existing geo-replicated systems. The main idea behind Saturn is to efficiently provide causal consistency to systems that do not guarantee it by design. In Saturn, there is a separation between handling the updates themselves and their corresponding metadata. Saturn makes sure that, for every update, the metadata is propagated to the different datacenters so that delivery happens in order that respects causality. It uses physical scalar timestamps to track causality and a tree structure and FIFO channels to propagate them across datacenters. The metadata used to tag updates is called a label. Labels are generated by gears associated to every storage server. A label sink is a component that collects all labels in a DC and is responsible of propagating them to other DCs in an order respecting causality. Remote proxies are responsible of applying remote updates in a causal order. Before being able to send requests, a client has to attach to a datacenter and provide the latest label it has observed. As a result its causal past is visible to the

datacenter and thus can safely interact without breaking causality. To read an object, the client sends a request with its key. The gear receiving the request, the value of the object and the label associated with it and returns them to the client. If the returned label is more recent, the client updates its label. To update an object, the client sends the key, new value and its label. The gear receiving the request, creates a new label and stores it along with the new value in the server's persistent storage. Both update data and metadata are sent to the label sink for inter-DC replication and the new label is returned to the client to replace the old label. The propagation of data and metadata to remote DCs happen separately. Labels are sent following a tree topology through serializers while data is sent in any order. Remote updates can only become visible when their labels arrive. As the the labels sent are scalar, concurrent operations cannot be differentiated from causally related ones in the serialization process. The choice of the serialization then can impact the latency experienced by clients and remote updates visibility latencies when labels (false dependencies) are delivered prematurely. The label dissemination tree and serializers are optimized to minimize such latencies by choosing the best serializations and sometimes delaying label arrivals.

2.5.1.8 Occult

Occult [67], which stands for Observable Causal Consistency Using Lossy Timestamps, is causally consistent geo-replicated data store that aims to solve the problem of slowdown cascades. Each replica is located in a datacenter with full copy of the data, sharded on different physical hosts. The system uses asynchronous master-slave replication instead of multi-master replication. Writes happen on master shards and are then replicated asynchronously but in order to the slave shards in remote DCs. Clients read from their local replica and write to the master shard. Causal consistency is enforced through the use of shardstamps and causal timestamps. Every shard has a shardstamp representing the number of writes it has accepted. A causal timestamp is a vector of shardstamps having one entry per shard. Causal timestamps are used to encode the last state of the data store observed by a client and also to capture the causal dependencies of writes. A client usually reads from their local replica but can contact any replica for load balancing purposes. A client sends a read request with the object's key. The receiving server replies with the object's value and its causal timestamp and the shard's shardstamp. The client compares the received shardstamp to the shard's entry in its causal timestamp. If the client's entry is at least equal to the shardstamp, then the version is consistent. If the replica contacted by the client was the master, the check will succeed. The check may fail however if the replica was a slave due to replication delays. In the case of a failed check, the client can retry again or contact the master. When the check succeeds, the client updates its causal timestamp to reflect the dependencies in the returned timestamp. For an update, a client sends a request to the master replica including the object's key, new value, and the client's timestamp. The master uses the client's timestamp to derive the update timestamp by assigning it shardstamp in the replica's entry and returns the shardstamp to the client. The client uses the returned shardstamp to update its causal timestamp. Before returning to the client, the master starts the replica processes by sending the updates to the slaves with their associated timestamps and the master's shardstamps by their

replicated order. The slaves apply the updates and update their shardstamps to the master's shardstamp. As the number of partitions tends to be very large, Occult uses three kinds of optimizations to compress the timestamp size: structural compression, temporal compression and isolating datacenters. This lead to compact timestamps but also increases the amount of false dependencies leading to higher visibility latency for updates.

2.5.1.9 Kronos

Kronos [68] is a centralized service that uses a dependency graph to encode relationships between events and provide a time ordering for distributed applications. Despite it not being a causally consistent data store, we mention it here for its use of a graph data structure to track dependencies between events. The nodes in the graph represent events and the edges define the causal relationships between them. An identifier of an event is called a reference. A client contacts Kronos to get a reference to an event. After gathering a set of references, a client can contact the service with the set as input to get a sequence of those references. This sequence represents an order of the events that does not violate causality. As concurrent operations can be applied in different order without affecting causality, Kronos can return different results for different clients even for the same set of references. A client can create an event and receive its reference. This adds a node with that reference to graph. The client can then use the references of created events to assign them an order. There are two ways to order events: must and prefer. A must relationship encodes a causal ordering between two events. A prefer ordering is used to assign relationships between concurrent events. In an assign call, all must conditions are applied or the operation is aborted. For prefer events, the operation is not aborted in case of consistencies, but the ordering is discarded. As Kronos is not a geo-replicated solution, there are no remotes updates to be propagated. This leads to a simple design. However, one of the weaknesses of Kronos is that if a client does not receive a certain reference that depends on another already sent to server, the resulting ordering will create an inconsistency at the client state. The client must have all the references to get a global ordering.

Tagged Causal Broadcast

3.1 Introduction

In an ideal world, distributed services are able to guarantee strong consistency for data, high availability for reads and writes, and manage to stay operational under failures and partitions. Unfortunately, this was proven to be impossible in what is known as the CAP theorem [9, 10]. As there is no “one-size-fits-all” solution, a range of consistency models exist, providing different guaranties and making different trade-offs, each suitable to some classes of applications and services.

Causal consistency [69] is the strongest consistency model achievable providing availability under network partitions [21, 70]. Under causal consistency, systems can aim for high availability while providing a rich set of session guaranties [71].

Traditional causal delivery middleware [4] provides a delivery order in each process that is *consistent* with causality, i.e., delivering messages at each process in some order which does not contradict causality. However, it does not provide client applications with knowledge about concurrency under the partial order of causality. Given two messages m_1 and m_2 delivered in that sequence to some process, no information is provided to the application whether m_1 causally preceded m_2 or if they were originated independently of each other.

In this thesis, we argue that providing such knowledge to the application is a mandatory feature for several classes of applications that require knowledge of concurrency in order to apply arbitration rules [5]. An example is an application that given two concurrent bids will arbitrate to consider only the higher bid. Since current implementations of causal delivery middleware lack this feature, they are only suitable for a more limited class of applications where this knowledge is not needed, and thus cannot be used as a general abstraction. We also show that trying to overcome this limitation, using current middleware, often leads to solutions that are inefficient in terms of metadata size and/or delivery delays due to false dependencies.

We make a case for a *Tagged Causal Delivery* (TCD) middleware, which delivers messages together with causality tags (e.g. logical clocks) that characterize the *end-to-end* happens-before, defined according to client-visible events in the client process order, and ignoring purely internal middleware events (such

as receiving a message and queuing it for later deliver). We contrast this with previous definitions which can leak internal middleware events, such as the one in [4].

We present both traditional and modern application examples that require the end-to-end happens-before. First, we revisit the well known critique of CATOCS (Causally and Totally Ordered Communication Support) paper [72] and show that in their stock trading example, if the ordering is exposed to the application, it would be possible to overcome the anomaly. This means that their diagnose that neither causal nor total multicast would work is valid only for current middleware, but not for TCD. As for modern applications targeting high availability, we consider the case of Conflict-free Replicated Data Types (CRDTs) [73]. We argue that to efficiently implement operation-based CRDTs causal delivery is necessary, but not sufficient, as the partial order of operation invocations is needed for the data-type semantics [7].

Together with the ability to compare causality tags in events, TCD provides information about *causal stability*. We clarify this concept and its importance at the API level, contrast it with classic message stability [4] (which is a purely implementation concern), and argue that it has been reimplemented in several applications (unknowingly).

We then focus on the implementation aspects. Achieving TCD should be easy by extending current causal delivery middleware. One could think that it would be a matter of simply exposing to the client API the causality tags (e.g., vector-clocks) that already exist in causal delivery middleware implementations. Surprisingly, we came across several pitfalls in doing so, which we describe in Section 3.5, as well as how to overcome them. These pitfalls originate from the models of possible interactions between the middleware and the client process, together with the current design goal of simply not contradicting causality, which would lead to an incorrect characterization of causality if internal tags were naively exposed.

Finally, we address *explicit causality* [14], and the tagging by the client application of each individual message with its predecessors. We argue that, although more powerful, explicit causality should (and can always) be left as a last resort, making TCD together with the more “static” choice of delivery groups as the way to achieve cross-object causal consistency among selected groups of objects.

3.2 Classical Causal Delivery

Causality ordering is about ensuring that effects are observable only after their causes. In a group communication context, causal delivery guarantees that a message is delivered only after all causally preceding messages. For instance, consider the following scenario where a group of friends communicate using some kind of distributed chat service:

- Ross sends to the group: “I won’t be able to join, I have to give a late lecture”
- Rachel: “Oh, too bad!”
- Then Ross replies: “Lecture canceled, joining soon!”

- Phoebe: “Great news!!”

If causality is not preserved (only some weaker delivery semantics such as FIFO), Joey, who is reading the conversation, could see the following:

- Ross sends to the group: “I won’t be able to join, I have to give a late lecture”
- Phoebe: “Great news!!”
- Then Ross replies: “Lecture canceled, joining soon!”
- Rachel: “Oh, too bad!”

Joey then might feel confused on why Rachel and Phoebe are not happy with Ross being able to join their gathering. This, however, would not have happened if the chat service used a causal delivery service.

Causal delivery was introduced by Birman [24] and Schiper [34], and made popular by the ISIS toolkit. It was further developed in [4], where Birman specifies a potential causality relation (\rightarrow), similarly to the happens-before relation introduced by Lamport [23], but allowing for each process p a partial-order (\xrightarrow{p}), as the transitive closure of:

1. if $\exists p \cdot e \xrightarrow{p} e'$, then $e \rightarrow e'$;
2. $\forall m \cdot \text{send}(m) \rightarrow \text{receive}(m)$

It distinguishes the receipt of a message (`receive`) from its subsequent delivery (`deliver`) when causal dependencies are satisfied, and makes \xrightarrow{p} , which intends to reflect the dependence of events at process p , respect $\text{receive}_p(m) \xrightarrow{p} \text{deliver}_p(m)$. A causal delivery middleware ensures delivery order, at each process, for causally related messages, by making them respect potential causality. For any processes p, q, r , if two sends are related by potential causality, i.e., $\text{send}_q(m_1) \rightarrow \text{send}_r(m_2)$ the causal delivery service guarantees that $\text{deliver}_p(m_1) \xrightarrow{p} \text{deliver}_p(m_2)$ at every process p where they are both delivered. Two immediate remarks can be made about this classic specification:

- it defines a partial order over all system events (a “system happens-before”), including internal events from the middleware (such as `receive`), which are not visible to the client;
- it defines delivery guaranties provided to the client using this relation, therefore leaking internal events to the specification.

3.3 Defining an End-To-End Happens-Before

In practice, without going for an “explicit causality approach” (as we discuss below) which would allow an arbitrary \xrightarrow{p} , the more simple approach for a general purpose middleware is to consider each process as sequential, with all events from each process being related in a total order, as the classic Lamport happens-before does. However, if internal middleware events, such as `receive`, `leak` and `are` are included in this total order, then the resulting potential causality will over-order events relevant to the client, leading to unnecessary delayed deliveries. ISIS does not have this problem, and effectively relates deliveries to subsequent sends, as desired, using vector-clocks [31], having a \xrightarrow{p} which is almost, but not, a total order. However, not having delivery guarantees defined exclusively over events visible to the client, but also in terms of internal middleware events, one cannot rely on this system-wide happens-before towards exposing it to clients.

A specification of causal delivery guaranties should only involve client-visible events. And the way to obtain a simple specification, not requiring explicitly client tagging of each message is precisely assuming a total order over client-visible events at each client process. This is the approach in the classic definition of causal memory [69], on which further work on causal consistency is based. There, client-visible events are a write or a read from memory; each client process is assumed to be sequential, having a total order of write and read events; and the causality order is defined as the transitive closure of the union of per-process order and a writes-into order relating writes to reads which see them.

By analogy, for causal delivery: client-visible events are sends and deliveries, and each $\text{send}_i(m)$ should be related to each $\text{deliver}_j(m)$. Together with a total-order at each process over these operations (but ignoring any internal middleware event), an end-to-end happens-before (\rightarrow) causality order is obtained as the transitive closure of:

1. $\forall i, j, m \cdot \text{send}_i(m) \rightarrow \text{deliver}_j(m)$
2. $\forall i \cdot e_i \rightarrow e'_i$ if e_i is before e'_i at process i

This is the definition that is intuitively desired. But we draw the attention to the fact that current causal delivery middleware does not care about providing it to the client application, because it only needs to enforce some unspecified delivery order which does not contradict it. And as we discuss later in Section 3.6, trying to make it available by exposing internal tags (e.g., vector-clocks) of current middleware does not work, leading typically to an over-ordering. We now argue that having this end-to-end happens-before available to the client is essential to satisfy many classes of applications.

3.4 The need for happens-before by applications

In 1993, Cheriton and Skeen wrote a paper on the limitations of causally and totally ordered communication support (CATOCS) [72]. In that paper, the authors address some limitations, present applications

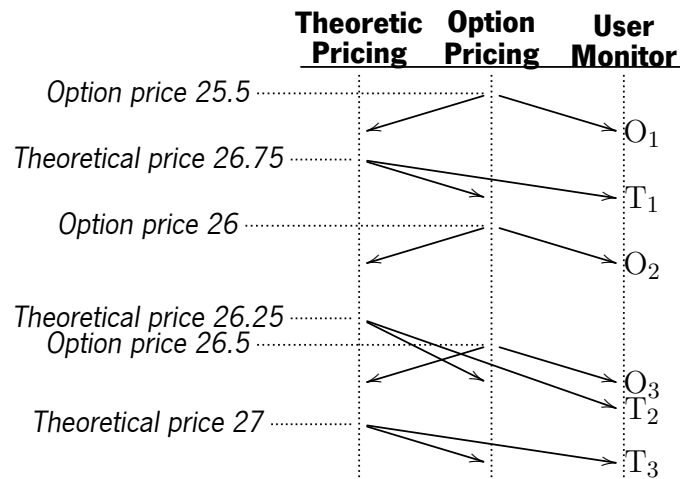


Figure 15: Trading example from Cheriton and Skeen's paper

that motivate the use of CATOCS and show how CATOCS fails to satisfy the expected requirements of those applications. The reason why CATOCS fails to do that is based on the end-to-end argument [74]: CATOCS tries to solve a semantic problem at the communication level. Instead they propose a state-level solution that, as they argue, renders the use of CATOCS irrelevant. In this section, we revisit some of their concerns regarding the limitations of CATOCS, namely current causal delivery middleware, in providing a solution that preserves the end-to-end semantics. First we revisit their example of a stock trading application; then we address the use of causal delivery middleware to support modern applications, namely in the implementation of Conflict-free Replicated Data Types [73].

3.4.1 The Stock Trading Example

We begin by explaining the trading application example of Figure 15. One server multicasts the price of an option. Another server calculates the theoretical price based on the received option price. For a correct behavior of the application, a theoretical price only makes sense in relation to the option price from which it is derived. It only makes sense to report it after that option price (which causal delivery ensures), but it is rendered obsolete if the option price has changed, and should be ignored in such case.

In their example, if the option server multicasts a new option price ($O_3 = 26.5$), this multicast will be concurrent to a theoretical price ($T_2 = 26.25$) multicast which has been derived from a previous option price ($O_2 = 26$). Given that they are concurrent, T_2 can be delivered to the user monitor after O_3 , but in this case it should be ignored. However, the application has no way to know if they are concurrent and should ignore T_2 , or if it is derived from the current option price (as it was the case for the first two messages delivered to the user monitor) and should report it. The authors conclude that neither causal nor total order delivery can avoid the anomaly.

We observe that delivery respecting causality (i.e., end-to-end happens-before of client-visible events) is useful, but not enough in itself. This causal order is preserved by CATOCS, but information about causal

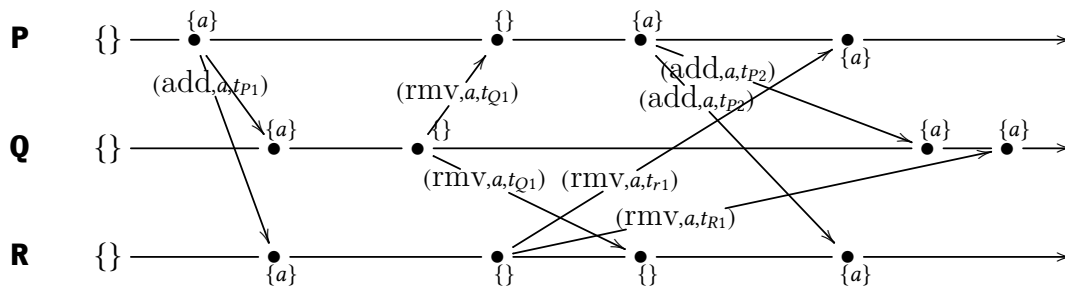


Figure 16: Add-Wins Set Example

concurrency is also needed for this application, something that CATOCS does not provide. But having causal delivery plus information about end-to-end happens-before is sufficient to overcome the problem. The anomaly described in the trading application example could similarly happen in other applications, where the semantics require knowledge about the causal relationship between events.

3.4.2 Implementing Conflict-free Replicated Data Types

A recent trend for building highly-available data stores, is to use well-defined data abstractions that provide conflict resolution by design through considering the application semantics. In particular, Conflict-free Replicated Data Types [73] (CRDTs) are recently getting a lot of attention and adoption by the industry mostly in building scalable, highly-available data stores [75–77] and collaborative editing software [78, 79]. CRDTs ensure data convergence once all executed operations are delivered by all replicas, provided that operations are carefully designed to resolve possible conflicts deterministically and in a semantically sound manner. Operation-based CRDTs [7] are data abstractions in which each replica propagates locally executed operations to other replicas. Supporting applications that exhibit causal semantics, an op-based CRDT assumes the presence of a Causal Delivery middleware to ensure causal delivery of operations. Despite this, the CRDT still requires causality metadata, e.g., timestamps, in order to reconcile conflicts caused by concurrent operations.

To better illustrate how such a data type works, we consider the case of the operation-based Add-Wins Set (AWSet). A set by nature is not commutative as concurrent (add, a) and (rmv, a) do not commute (i.e. being delivered to the application in different orders leads to different states). To guarantee state convergence, the AWSet resolves this ambiguity by providing Add-Wins semantics, where an (add, a) would “win over” a concurrent (rmv, a) . In order for this to work, the application requires the knowledge of the happens-before relation between operations. When operations are delivered, they are added to the state with the causality metadata. Using this metadata, a query issued by a user is able to return a state consistent with the end-to-end semantics.

In Figure 16, we present an example of an AWSet replicated at 3 processes P, Q, R . A causal delivery middleware ensures the causal delivery of the operations (here, (add, a) and (rmv, a)) respecting their causal order of delivery. For each operation issued by a process (e.g. process P), additional causality information (here t_{P1}, t_{P2}) are added based on what was delivered at that process. (add, a, t_{P1}) happened

before (rmv, a, t_{Q1}) as it shown in the figure, and as causal delivery is guaranteed by the middleware we show a correct evolution of the state from element a being in the set $\{a\}$ to a removed from the set $\{\}$. The more interesting case is when (add, a, t_{P2}) was issued at P concurrently with (rmv, a, t_{R1}) at R . At process Q , (add, a, t_{P2}) was delivered leading to a being added to the set. Then, (rmv, a, t_{R1}) was delivered, however, not leading to the removal of a from the set as expected from the Add-Wins semantics of the set. Preserving the end-to-end semantics was possible here, because the knowledge that (add, a, t_{P2}) was concurrent with (rmv, a, t_{R1}) was exposed to the application at Q through t_{P2} and t_{R1} . If this knowledge (t_{P2} and t_{R1}) was not exposed, the Add-Wins semantics would have been violated leading to an inconsistent state $\{a\}$ at P , $\{\}$ at Q and $\{a\}$ at R .

3.5 Happens-Before as a Middleware Service

In the previous section, we showed that the end-to-end happens-before relation is crucial to the semantics of many applications, and should be provided. In this section we argue that this information should be exposed by the causal delivery middleware itself. We observe that current implementations of operation-based CRDTs solve this problem by explicitly generating causality tags, manipulating them and embedding them in messages. We explain that while this solves the problem, it presents a duplication of effort and could lead to unnecessary delay on message delivery.

3.5.1 Two-Level Tagging using Current Middleware

In this section, we describe a way to satisfy the requirements of applications mentioned in the previous section, using traditional causal delivery middleware, as well as its drawbacks that motivates the need for more suited middleware.

Currently, the implementors of such applications tend to rely on off-the-shelf causal delivery middleware to obtain an exactly-once delivery respecting happens-before. However, the happens-before information, which is needed by the application, is not exposed in the causal delivery middleware API. This makes the implementors explicitly add timestamping information (e.g., vector-clocks or globally-unique tags) to the messages and application state in order to track the happens-before relation. This means additional state and more information in the message payload, which is a waste and a duplication of effort, considering that the causal-delivery middleware is already tracking happens-before, but not exposing it to the client application.

Besides this duplication of effort and the overhead in terms of additional causality metadata, another problem arises. As the tags provided by the application are opaque to the middleware, it could happen that two operations that are concurrent and, therefore, tagged as concurrent by the application, could be ordered (causal delivery timestamps) as one in the future of the other by the middleware. The reason for this is that operations are tagged at the middleware level depending on their incidental arrival and pre-delivery processing, not necessarily reflecting the (end-to-end) delivery order as seen by clients.

As we discuss in Section 3.6, in current middleware, one operation o_a could be waiting in a queue at some process p_i to be tagged by the middleware while another operation o_b , concurrent to it, is being processed by the middleware at p_j . Process p_j could tag o_b and multicast it to other processes, namely p_i , that delivers it to the application. When p_i processes o_a it will tag it as in the future of o_b when in fact the operations are concurrent. This would not affect the semantics of the application because the application would “know” that o_a and o_b are concurrent based on the application-level tags. However, this would add an unnecessary delay due to the over-ordering dictated by the middleware: at all other destinations, the middleware would have to wait for the delivery of o_b before delivering o_a .

3.5.2 Exact Tagging by the Middleware

The above discussion and observations makes us conclude that it will be useful to make end-to-end happens-before information available from the causal delivery middleware itself. Not only it spares programmers from constant reimplementations efforts, avoids duplication of roles and metadata between the application and middleware, and can also sometimes even remove unnecessary delays caused by over-ordering that happen in current middleware. But to obtain a correct characterization, some care must be taken, as we discuss in Section 3.6.

3.6 Pitfalls in Exposing Middleware Timestamps

To avoid the effort of implementing TCD from scratch, one can think of adapting current causal delivery middleware to provide the TCD API. At first glance, it might seem trivial to implement a causal delivery that characterizes the end-to-end happens-before, by using any traditional causal delivery middleware service and exposing to the client application the timestamps (e.g. vector-clocks) that are used internally to ensure causal delivery. However, when considering concrete implementations, some unexpected problems arise. We show how naively exposing the timestamps would lead to an incorrect characterization of causality, in either of the two typical interaction models between middleware and client code: callback-based and with independent threads/processes.

Callback-based In an event-driven architecture with a single process, the application code runs as callbacks invoked from the middleware code when messages need to be delivered to the application logic to be processed, e.g., $\text{deliver}(m, t)$ for message m tagged by t timestamp. To avoid reentrancy problems, when a send is invoked inside the deliver callback, the send simply adds the message to a queue, to be handled by middleware code when the callback finishes. It can happen that the middleware has a set of messages ready to be delivered, and invokes the deliver callback for each one, before handling sends which have been enqueued. If the middleware creates timestamps for messages to be sent only upon dequeuing them, then a message will be tagged as causally in the future of all messages that were delivered after the send action by client code and before dequeuing occurred. This means that some

messages that are actually concurrent are tagged as causally related, making timestamps reflect a larger relation than happens-before, over-ordering some events. While this does not break causal delivery, as it is consistent with end-to-end happens-before, it means that these timestamps cannot be exposed as precisely characterizing end-to-end happens-before.

Independent threads/processes/actors In other architectures we have two independent processes: a client process and a middleware process. Here, in addition to the queue of messages to be sent, as above, we will typically also have a queue of messages ready to be delivered. The middleware tags and enqueues messages to the deliver queue, while the client dequeues and processes them. When doing a send, the client enqueues a message to the send queue. This message will be tagged by the middleware process as in the future of other messages not yet delivered by the client (namely, those that are still in the delivery queue but have already been handled by the middleware process), when they are in fact concurrent. Note that what defines the end-to-end happens-before is the order of send and deliver events as observed by each client process; other events, e.g., when a message was enqueued or dequeued by the middleware process, are irrelevant (i.e., internal middleware events, but invisible to the API).

3.7 Tagged Causal Delivery

To correctly characterize and end-to-end happens-before, a message being sent must be tagged reflecting the causal knowledge according to all delivery events at the application, up to the send event (at the application). Therefore, a Tagged Causal Delivery middleware provides in its API a `tcbroadcast` callback method that multicasts a message to all other processes and a `tcdeliver` callback method that delivers a message to the application with some causality tag. We do not address the implementation of `tcbroadcast` and `tcdeliver` in this section but we discuss it in subsection 3.7.1. However, we reserve this section to present causal stability and extend the TCD middleware's API with a `tcstable` that provides causal stability information.

3.7.1 Lessons Learned and a General Solution

Exposing current middleware tags towards achieving TCD does not work in general, as they do not characterize the end-to-end happens-before. The general reason for that is the unspecified behavior in terms of the relation between client-visible events (send and deliver) and all other internal middleware events. This should not be surprising, after some thought, as current middleware was designed with the only constraint of delivering in some order consistent with end-to-end happens-before.

There is, however, a simple solution which allows adapting any middleware using internal tags that describe a partial-order (e.g., vector-clocks) between client-events. It works whether the middleware is callback-based assuming a single process, or involves interactions between a client process and at least

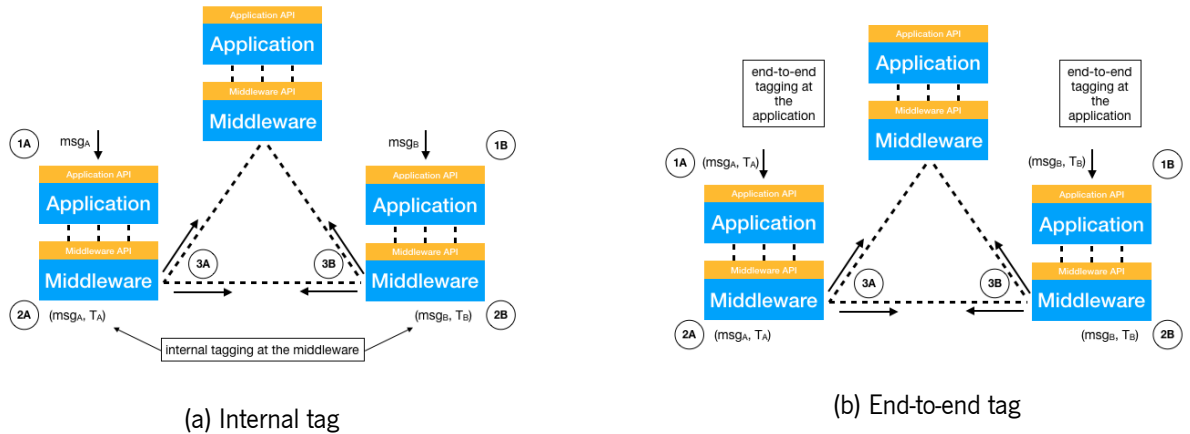


Figure 17: Difference between internal tagging and end-to-end tagging.

one more process. The key insight is that timestamp generation must be made by middleware functions which operate on metadata stored inside some (opaque) middleware object given to the client and stored at the client process. Then:

Figure 17a shows an example of how messages are tagged in traditional causal broadcast middleware. The middleware is responsible of tagging the messages upon their arrival. We call it internal tagging as the tagging is dictated by the internal communication layer. In this figure, we notice that in the case where 3A (respectively 3B) which is the broadcast and reception of message msg_A at B (respectively msg_B at A) happens-before 2B (respectively 2A) which is the internal tagging of message msg_B (respectively msg_A), then msg_A (respectively msg_B) will be tagged as happens-before msg_B (respectively msg_A). While this does not break the causal order provided it has two flaws:

- correctness: From the application perspective, msg_A and msg_B are concurrent. This concurrency is not preserved and therefore end-to-end causal order is not provided. This is important for a class of application where there is priority/order between concurrent messages.
- efficiency: As both messages are in fact concurrent, one could be delivered independently of the delivery of the other. However, as they were tagged as one happens-before the other, this over-ordering could lead to an extra delay in the delivery mechanism.

The general problem stems from trying to solve an application-level problem (semantics, end-to-end) at the communication-level. This renders traditional causal broadcast middleware not suitable for a class of application that, for a correct behavior as expected by the user, requires the preservation of this end-to-end happens-before semantics.

Conserving the end-to-end semantics is important for a class of applications where there is priority/order between concurrent messages, and therefore not conserving this order leads to breaking the semantics of the application. In fact two “ingredients” are necessary to be provided by a causal broadcast middleware to fill this gap. The first is providing a way to tag every message at the application (client)

based on what was seen (delivered) by that application. The second is delivering all messages with that tag and providing a way to compare those tags as causally related or concurrent.

We should note that exposing current middleware tags towards achieving tagged causal delivery does not work in general, as they do not characterize the end-to-end happens-before. The general reason for that is the unspecified behavior in terms of the relation between client-visible events (send and deliver) and all other internal middleware events. This should not be surprising, after some thought, as current middleware was designed with the only constraint of delivering in some order consistent with end-to-end happens-before.

A simple solution is to allow adapting any middleware using internal tags that describe a partial-order (e.g., vector-clocks) between client-events. The key insight is that timestamp generation must be made by middleware functions which operate on metadata stored inside some (opaque) middleware object given to the client and stored at the client process. In Figure 17b, the end-to-end happens-before order is preserved. The reason is that messages are being tagged at the application level (steps 1A, 1B) independently of the order of their arrival at respective nodes. This follows the next mechanism:

- Each time a send is invoked by the client, a timestamp should be immediately generated as part of the (atomic from the client perspective) send event, using the middleware object at the client. If the message is then added to a queue, it has already been tagged.
- Each time a deliver event happens at the client process, the timestamp metadata at the middleware object should be immediately updated, before the client action (e.g., a callback) runs. This means that if the client action invokes a send, it will be considered in the causal future (i.e., as resulting from) that delivered message, but not in the future of some ready-to-be-delivered messages that are still in some middleware queue.

3.8 Explicit Causality versus Explicit Grouping

In [72], the authors focus primarily on the limitations of CATOCS and mention that a more suited solution that does not violate the end-to-end argument would be a state-level solution. Although the mentioned solution is not covered in detail it seems to be one that provides an “explicit causality ordering” in contrast to the “potential causality” order provided by traditional middleware and by our TCD proposal.

Explicit causality [14] is an interesting approach that provides application-specified causal dependencies. Instead of considering all delivered messages at a process as dependencies of the next message to be sent by that process, the application specifies the explicit dependencies (“actual causes”) of the current message.

The advantage is a considerable reduction in the causal dependencies that are enforced by the system at the cost of requiring the application to provide accurate information on the dependencies to enforce. In practice this requirement can be hard to achieve, placing a burden on application programmers. A

benign example, where this is achievable, is to consider in a discussion forum with multiple threads of topic conversations (e.g., Reddit¹) only the causal dependencies inside each given topic, as opposed to the whole forum. The limitation is that possible causal connections between topics can be lost, in contrast to the far more encompassing potential-causality that would track them.

A middle-ground between potential causality (assuming a total order per process) versus explicitly listing actual causal dependencies, can be obtained by a partitioning into groups of related objects and using TCD to enforce end-to-end happens-before within each group. In the discussion forum example, each topic could be contained in its own group or packed together with other topics in a shared group according to some desired application policy. Explicit grouping is not as versatile, but it is simpler and enough to cover many cases; explicit causality can be left as a last-resort, when everything else fails.

¹<https://www.reddit.com>

Middleware

4.1 Introduction

In chapter 3, we made a case for Tagged causal delivery which provides an end-to-end happens-before and preserves concurrency information as part of the causal metadata. This chapter is the complementary of the previous one and addresses the implementation of a new class of middleware that provide Tagged causal delivery and stability. Our aim was to develop a generic middleware that could replace current implementations that provide the classical causal delivery, which did not address the problems mentioned in chapter 3, and therefore be more suitable for a wider range of applications. In our development of the middleware, different contributions were made and that we present incrementally in this chapter in order to reduce the complexity of the content and offer an easier outline that maximizes the reader's understanding.

In section 4.2 we present the architecture of the system, the internal processes it comprises, what every process does, and how they communicate with each other. We then describe the API of both the system in general so the reader can get the gist of what it is supposed to do. In section 4.3 we start with the most basic implementation of a causal delivery middleware that offers tagged causal delivery using state of the art techniques and data structures such as version vectors and delivery queues. Then, in section 4.4 we start explaining the causal directed acyclic graph (DAG) and dependency dots, one of the main contributions that aims to improve the performance of our algorithm, explain how it works compared to the previous presented techniques in section 2.4, and present a better, simple implementation which uses the dependency dots and the causal DAG in section 4.5.

4.2 API and Architecture

In this section, we present a middleware service providing end-to-end happens-before delivery, an efficient tagging and causal delivery protocol. We start by presenting the communication pattern between client and middleware processes in subsection 4.2.1 and then the API of our system in subsection 4.2.2.

4.2.1 Architecture

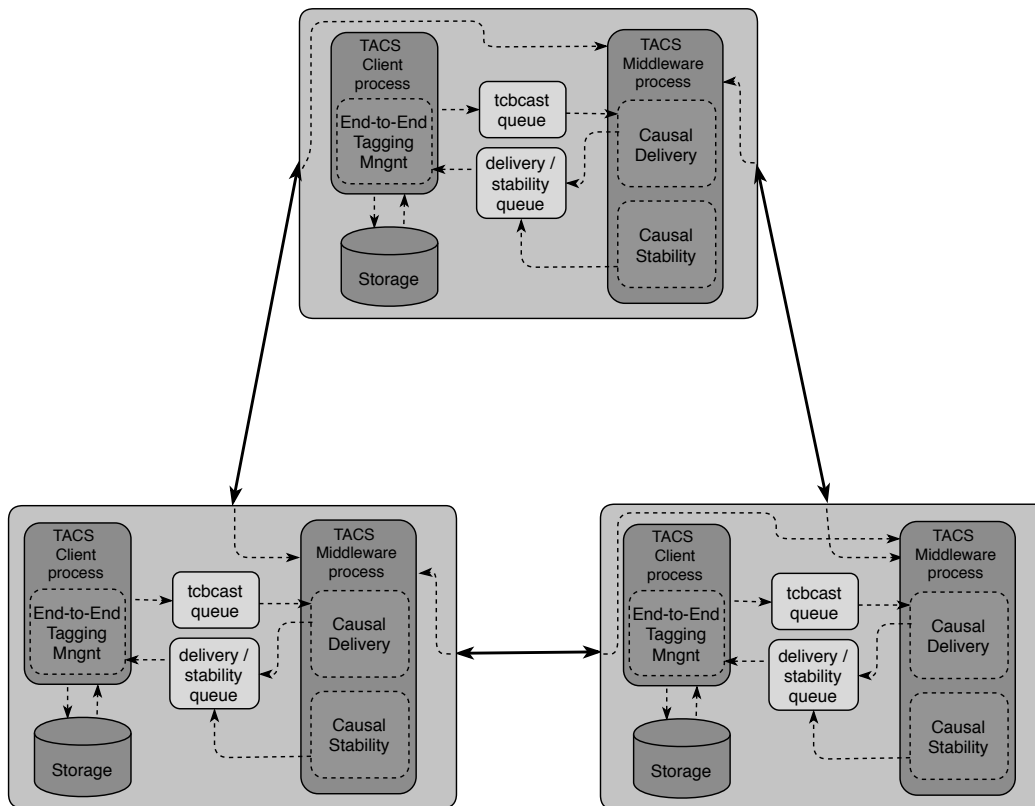


Figure 18: Tagged causal delivery architecture

In Figure 18, we present the architecture of our system, the processes that run on every node (replica) in our system in order for the tagged causal delivery algorithm to function correctly as well as the communication pattern between the different processes, both intra- and inter-nodes. In every node, runs a client process and a middleware process. The client is a client library that the application using the tagged causal delivery service uses to communicate with the middleware process. The middleware process is the one responsible with most of the work required in ensuring a tagged causal delivery and stability.

All requests to query or mutate the state of data objects arrive from the application at the client process. The query requests are trivial: query requests are not broadcast, and do not mutate the state at the client or middleware processes. Therefore, we will only focus on update requests issued by a user and received at the client process. Upon arrival, the client process prepares the request (or message) before being broadcast by the middleware process to every node in the group. The prepare, basically tags the message by a globally-unique identifier and the local context, then enqueues (calls $\text{enqueue}_i^{cm}(\text{dot}_i, \text{ctxt}_i, \text{msg})$) the prepared message in a FIFO queue, called tbcast queue as shown in Figure 18. Every update message has a globally-unique identifier. The local context at the client process is a set of globally-unique ids that represent the last messages delivered at the client process. The local context represents the causal dependencies of a new update and it is the task of the client process to keep this local context and

manage the end-to-end tagging.

The middleware process, on the other hand, handles the receiving, delivery and stability of updates both from its client process as well as from its peer nodes. There are multiple mechanisms involved in the management of correct and reliable transmission of messages, causal delivery as well as causal stability. We keep those for later sections. The middleware process dequeues the local messages from the `tcbroadcast` queue, updates the local data structures as needed (that we discuss later) and then broadcasts the message to all other peers. The middleware also receives messages from peers through `receivej,i(dotj, ctxtj, msg)`, updates the local data structures as needed, and checks the delivery/stability of received messages. When a message is ready to be delivered (*state* is DLV), it is enqueued in another FIFO queue called delivery/stability queue as shown in Figure 18. Similarly to delivery, when a message is locally marked as causally stable (*state* is DLV and *bstr* is $\{0\}^N$), the stable messages are enqueued in the same queue used for delivery. When the client process dequeues messages from the delivery/stability queue, through `dequeueimc()`, it checks (by pattern matching) if it should deliver a new message or stabilize an already delivered message. In the former case, the `tcdeliveri()` callback is invoked, in the latter `tcstablei`. Upon delivering a message, the client process updates its local context to reflect the changes in causal dependencies.

4.2.2 API

To correctly characterize and end-to-end happens-before, a Tagged Causal Delivery middleware must provide two main functionalities:

- a message being sent must be tagged reflecting the causal knowledge according to all delivery events at the application (client process).
- a message must be delivered, by the middleware to the application, with its tag that carries causality information, making it comparable (using end-to-end happens-before) to other messages.

Therefore, TRCB provides in its API, the following callbacks:

- `tcbroadcast`: (tagged causal broadcast) broadcast a message to every process in the group membership. The client process will be responsible for tagging the message to be sent with the causal predecessors known (delivered) by the client process.
- `tcdeliver`: (tagged causal delivery) delivers a message to the application with the tag. For instance, in the case of pure operation-based AW-Set S_a , `tcdeliver` would add the operation (*add*, *e*, and respective timestamp *t*) to the state S_a . It uses the existence of comparable timestamps for operations to update the state, based on the semantics of AW-Set.
- `tcstable`: (tagged causal stability) We postpone explaining this callback to the next chapter.

4.3 Vector-Clock-based algorithm for tagged causal delivery

A vector clock is a mechanism to determine the causal relation between events. In a group with N peers, each of these has a version vector with N entries. Initially every position in the vector starts at 0 and increments by 1 each time an event happens in the peer corresponding to that entry. The events that trigger an update of a node's version vector can either be the broadcast or delivery of a message. When sending, the peer increments its entry in the local version vector before copying it to the message. Upon receiving a message, the peer compares the message's version vector with its own. A message can only be delivered if all the events that caused it have also been delivered. For example, peer i can only deliver a message m from j with version vector V_m if:

$$V_m[j] = V_i[j] + 1 \wedge V_m[k] \leq V_i[k] \forall k \neq j$$

When this condition is not met, the message is queued in a data structure called message queue, until it can be delivered later on when this condition is met. The components for this version include vector clocks are used here as timestamps to characterize causality between events, a queue that stores the messages that cannot be delivered yet, as well as the previous condition presented above to check causal delivery of a message.

We extend current causal middleware to provide tagged causal delivery and we show that in the next subsections 4.3.1 for the *client process* and 4.3.2 for the *middleware process*.

4.3.1 Client process

The algorithm for the client process can be seen in **Algorithm 2**. The *Client* keeps locally as its state a version vector V_i , with length equal to the group size (number of participants or nodes). Initially every position of this vector starts at 0 and is incremented upon a send or delivery.

An application uses the client library (process) to broadcast an update or any kind of message by calling *tcbcast*. For peer i the $V_i[i]$ entry of its version vector is incremented. The *tcbdeliver* callback is called and the payload along with the process' updated version vector are enqueued using enqueue_i^{cm} (the superscript cm denotes the direction from client to middleware) to *tcbcast* and sent to the *Middleware* process.

dequeue_i^{mc} is a another *FIFO* queue storing messages that are tagged by the middleware process as ready to be delivered by the client process. The sender's entry in V_i is incremented before calling the *tcdeliver* callback.

ALGORITHM 2: Vector-based tagged causal delivery algorithm without causal stability at client process for node $i \in \mathbb{I}$

```

1 state:
2 |  $V_i : \mathbb{I} \rightarrow \mathbb{N}$  /* delivered version vector */
3 proc  $\text{init}_i()$ 
4 |  $V_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
5 proc  $\text{tbcast}_i(msg)$ 
6 |  $V_i := V_i[i] + 1$ 
7 |  $\text{tcdeliver}_i(i, V_i, msg)$ 
8 |  $\text{enqueue}_i^{cm}(msg, V_i, msg)$ 
9 on  $\text{dequeue}_i^{mc}(j, V_m, msg)$ 
10 |  $V_i[j] := V_m[j]$ 
11 |  $\text{tcdeliver}_i(j, V_m, msg)$ 
12 fun  $\text{tcdeliver}_i(j, V_m, msg)$ 
    | /* delivery callback */

```

4.3.2 Middleware process

The algorithm for the *Middleware* process is shown in **Algorithm 3**. Each peer i has a version vector R_i for received messages, a version vector V_i for delivered messages and a delivery queue DQ_i . The version vectors R_i and V_i have, respectively, the id of the last received and delivered message. Initially, their entries are initialized to 0 and delivery queue DQ_i is set to empty ().

dequeue_i^{cm} dequeues the local updates from the client process, updates V_i to reflect the delivered message and broadcasts a message containing the tag msg to denote its type, the replica id i , its version vector V as sent by the client process, and its payload msg .

When a message from j is received at i the $R_i[j]$ entry is incremented and when a message is delivered the $V_i[j]$ entry is incremented. If a received message cannot be immediately delivered then it is enqueued to DQ_i to be later delivered after all its causal predecessors have also been delivered. On the other hand, when a received message is delivered, DQ_i is traversed until there are no more messages that can be delivered at the moment. This is because upon a message delivery, other messages that were queued could be ready to be delivered due to one of their causal predecessors being received and delivered.

Removing an element from the middle of a vector can hinder performance since all the elements after it have to be shifted to the left. Instead of this, while looping through the queue to check delivery, if a message cannot be delivered then it is individually shifted left to the right place, otherwise it is enqueued to tcdeliver . When the end of DQ_i is reached and at least one delivery was made then it is again traversed from the beginning because other messages might be deliverable. If at the end no deliveries were made then DQ_i is truncated, since the first messages on this queue were considered received but not deliverable, and they were shifted to the left while traversing, leaving unused positions in the end of the vector.

ALGORITHM 3: Vector-based tagged causal delivery algorithm without causal stability at middleware process for node $i \in \mathbb{I}$

```

1  state:
2   $V_i : \mathbb{I} \rightarrow \mathbb{N}$       /* delivered version vector *#21
3   $R_i : \mathbb{I} \rightarrow \mathbb{N}$       /* received version vector *#22
4   $DQ_i :$                 /* delivery queue *#23
5  on  $\text{init}_i()$ 
6   $V_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
7   $R_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
8   $DQ_i := \emptyset$ 
9  on  $\text{dequeue}_i^{cm}(\langle \text{msg}, V, m \rangle)$ 
10  $V_i[i] := V_i[i] + 1$ 
11  $\text{broadcast}_i(\langle \text{msg}, i, V, m \rangle)$ 
12 on  $\text{receive}_i^j(\langle j, V_m, m \rangle)$ 
13 if  $R_i[j] < V_m[j]$  then
14    $R_i[j] := R_i[j] + 1$ 
15   if  $V_m[j] = V_i[j] + 1 \wedge V_m[k] \leq V_i[k] \mid \forall k \neq j$  then
16      $\text{enqueue}_i^{mc}(\langle \text{dlv}, j, V_m, m \rangle)$ 
17      $\text{deliver}_i()$ 
18   else
19      $DQ_i := DQ_i + (j, V_m, m)$ 
20 proc  $\text{deliver}_i()$ 
21  $\text{from} := 0$ 
22  $\text{to} := 0$ 
23 while true do
24   if  $\text{from} \geq \text{len}(DQ_i)$  then
25     if  $\text{to} \geq \text{from}$  then
26        $\text{break}$ 
27      $\text{truncate}(DQ, \text{to})$ 
28     if  $\text{len}(DQ_i) = 0$  then
29        $\text{break}$ 
30      $\text{from} := 0$ 
31      $\text{to} := 0$ 
32   else
33      $(j, v, v_m) = DQ[\text{from}]$ 
34     if  $v_m[j] = V_i[j] + 1 \wedge V_m[k] \leq V_i[k] \mid \forall k \neq j$  then
35        $V_i[j] = V_i[j] + 1$ 
36        $\text{enqueue}_i^{mc}(\langle \text{dlv}, j, V_m, m \rangle)$ 
37     else
38        $DQ[\text{from}] := DQ[\text{to}]$ 
39        $\text{to} := \text{to} + 1$ 
40      $\text{from} := \text{from} + 1$ 

```

4.4 Causal DAG

In the section we revisit the use of data structures and algorithms used in the classical causal delivery implementations and propose different ones that we consider more suitable and efficient for representing causality. In the following we explain why we moved to using “dependency dots” instead of vector clocks and then propose a causality Directed Acyclic Graph to hold the causal metadata as well as replace the traditional message queue.

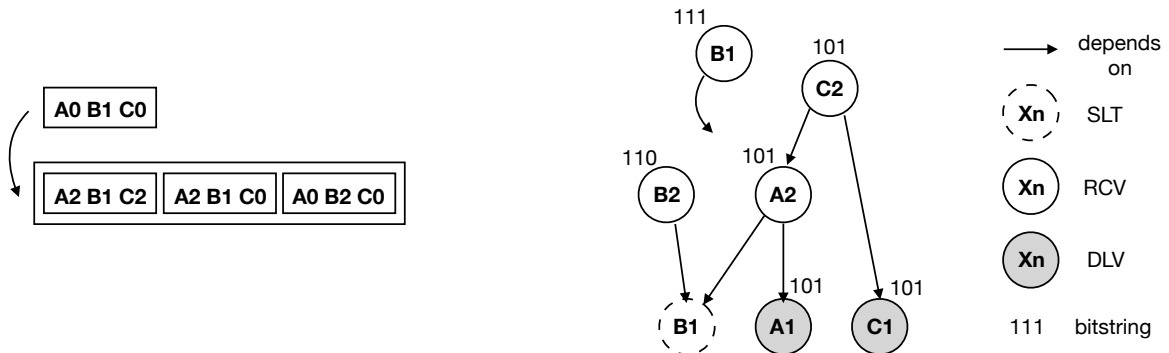


Figure 19: Graphical representation of causal delivery using a delivery queue and a causal DAG

4.4.1 Reducing the causal information overhead

In the initial implementation, we used vector clocks to track causal information between operations. This information is coupled with each operation to allow the delivery of the operations in an order respecting causality of events. Vector clocks have been used widely to track causality and achieve causal delivery in a group communication environment. However, as their size grows linearly with the number of participants in the group, they become less efficient in large scale settings. To support large scale group communication services, reducing the causal information overhead imposed by the use of vector clocks is a problem to be solved.

We tackled this problem by implementing “dependency dots” (see Figure 19). Those “dependency dots” are similar to some approaches used to compact the size of vector clocks [4, 45, 46]. We also applied the transitive reduction on those dependency dots that allowed reducing the size even more. While similar ideas exist in the literature, we present the implementation of the protocol using this alternative and how the transitive reduction can be applied, which is not tackled previously.

The main idea here is intuitive. A vector clock has one slot per peer representing the number of updates seen (delivered) from the peer. This is a very compact information to represent causality. However in large scale settings two things happen: the number of slots needed increases which makes the vector clocks large in size, and most often only some of the nodes (at least at any point in time) are the ones making updates. What this means is that a whole vector clock for every update when only a few slots need to be updated seems like a lot of redundant information that could be trimmed down to compact the size of the vectors and reduce the causality information overhead.

A dot is represented as $\doteq \mathbb{I} \times \mathbb{N}$ and is used as a unique identifier for a message. When a message m_1 represented by the dot d causally happens-before another message m_2 represented by the dot d' it means that d' causally depends on d and therefore we call d a causal dependency of d' . A dot can have more than one dependency dot, a set of dependency dots to characterize causality. At most the size of this set of dependency dots is equal to the number of participants, thus making the size of our “dependency dots” equal to the size of a vector clock only in the worst case scenario. But in general it small as a node only needs to send a number of dots equal to the number of updates delivered since its last send. Obviously, many updates could be delivered at a node between two consecutive sends, however, these updates are not necessarily concurrent and therefore there exists some dependency between them. This information can be reduced to only keep the most recent dots in the set of dependencies and it's called a transitive reduction on the dependency graph. For example, looking at Figure 19, we can see that C_2 could have in its set of dependencies $\{A_1, B_1, A_2, C_1\}$. By applying a transitive reduction on this subgraph, we only need to keep $\{A_2, C_1\}$ because C_2 directly depends on only A_2 and C_1 but transitively on A_1 and B_1 . So having A_2 in C_1 's dependency dots is enough to mean A_1, B_1 , and A_2 . Now, the size is of our dependency dots that a node need to send in a message is reduced even more to be equal to the number of *concurrent* updates delivered at that node since its last sent message.

4.4.2 Reducing delivery time

Due to communication delays that usually happen in distributed settings, some received messages might not be ready to be delivered if they fail to satisfy the causal delivery conditions. Only when the messages and updates that causally precede them are delivered, they are ready to be delivered. State of the art blocking causal delivery implementations tend to store in a delivery queue until they can be delivered again. Intuitively, whenever a new message is delivered, it could lead to the delivery of other messages that are in the queue itself. The same is true for every new message that gets delivered in the queue, triggering other messages to be delivered. This would lead to multiple loops of the whole queue where the causal delivery check is tested against every vector clock, slot by slot resulting in unnecessary delays, especially when the queue grows a lot.

Queues by design are more suitable for sequential ordering of events and are therefore a bad design choice to represent partially ordered messages. Causal order is a partial order, an update can causally depend on one or more updates, the causal dependence relation is non reflexive, meaning that we cannot have two updates a and b where $a \rightarrow b \wedge b \rightarrow a$. A DAG, or directed acyclic graph is a more suitable data structure to represent causal dependencies. This fits great with the use of “dependency dots” mentioned before to track causal relations in the system. Each message is tagged/timestamped by a globally unique identifier that we call dot. A dot is a couple (id, ctr) where $id \in \mathbb{I}$ is the replica identifier and $ctr \in \mathbb{N}$ is a monotonically increasing counter. At each replica, the causal order between messages/events is captured using a DAG stored and updated by the middleware process. Each vertex in the DAG represents a message/event and each directed edge represents the causal dependency between those messages/events. This makes it faster to track dependencies of each message which is not possible with queues, resulting in less delays in the delivery as only the undelivered successors of a message need to be checked and not the whole queue. Therefore faster a delivery mechanism which may not be very significant in a normal scenario, but is very significant when the number of undelivered messages grows due to a partition or message loss. The system therefore could recover faster in such scenarios.

4.4.3 Causal dependency graph: notations and functionality

Definitions and notations of data structures used in the causal dependency graph:

$dot := \mathbb{I} \times \mathbb{N}$, message unique identifier that we call a dot;

$DS := stage \times bits \times pred \times succ \times pyld$, data (record fields) associated with a dot in the dependency graph;

$bit : \mathbb{I} \leftrightarrow \mathbb{N}$, map from node id to bit mask;

$\mathcal{G} := \hookrightarrow DS$, a DAG representing dependencies between dots;

Every dot in the dependency graph is associated with a value with the following record fields:

$stage := \{slt, rcv, dlv, stb\}$, message stage as slot (empty), received, delivered and stabilized respectively;

bits := $\{0, 1\}^N$ bit string where each bit position represents a member id used for checking delivery and stability;

pred := $\mathcal{P}()$ set of predecessors (ids of messages on which the delivery of this message depends);

succ := $\mathcal{P}()$ set of successors (ids of messages whose delivery depends on the delivery of this message);

pyld := message payload;

Each replica has a globally unique identifier $id \in \mathbb{I}$, such as ids are comparable and can be totally ordered. Every replica i in the group has knowledge of the full membership which is stored in hashmap bit that maps each replica id to value equals $2^{pos(id)}$ where $pos(id)$ is the index of id in the ordered list of ids. For instance, if the list of ordered replica ids is $List := (a, b, c) | List[0] = a...$, then $bit(a) := 2^0$.

For convenience, we implement the DAG abstraction as a hashmap that we call \mathcal{G} , that maps each dot to some data structure we call \mathcal{DS} . The \mathcal{DS} data structure is composed of 5 attributes. The **stage** attribute which is an *enum*(slt, rcv, dlv, stb) encodes the stage of the message/events represented by the dot. The stage takes one of the constant values slt, rcv, dlv, stb. slt encodes a placeholder for a dot that has not been received, but is listed as a dependency of another received dot. rcv encodes a received dot that has not been delivered yet. dlv/stb encodes a dot that is ready to be delivered/stable by the client process. The **bits** attribute is one of the most useful as it encodes lots of information on just one bit string of size N . This bit string encodes information used for delivery and for stability depending on the value of **stage**. When a dot's **stage** is rcv, then **bits** encodes the bit positions of every not delivered predecessor. Initially, the value of **bits** is 1 for the position of every not delivered yet predecessor, and 0 for the others. When **bits** is 0, the dot is ready to be delivered. Similarly, when the value of **stage** is dlv, **bits** encodes causal stability. Initially, the value of **bits** is 1 for every peer. The bit position of a certain peer (j) becomes 0 (at i), when a message coming from j and is in the future of this dot is delivered at i. When **bits** is 0, the dot is ready to be stable and **stage** becomes stb. The attributes **pred** and **succ** encode respectively the set of dots representing the predecessors and the successors of the current dot. For instance, if we consider 3 dots d_i^a , d_j^b and d_k^c such that d_i^a has dependency d_j^b and d_j^b has dependency d_k^c , d_i^a would have **pred** set to $\{d_j^b\}$, d_j^b would have **pred** set to $\{d_k^c\}$ and **succ** set to $\{d_i^a\}$, and d_k^c its **pred** set to $\{d_j^b\}$. Note that in this chapter, although we mention causal stability during the explanation of the causal DAG, we do not tackle causal stability until the next chapter.

4.5 Graph-based Algorithm for tagged causal delivery

We started by presenting the vector based version of our tagged causal delivery algorithm, and then explaining how we can use dependency dots and the causal dependency graph to capture causal dependencies between updates and ensure causal delivery. Now we merge both into a graph-based algorithm providing tagged casual delivery that we explain below.

4.5.1 Client Process

ALGORITHM 4: Graph-based tagged causal delivery algorithm without causal stability at client process for node $i \in \mathbb{I}$

```

1 state:
2    $d_i \in \mathcal{D}$ 
3    $ctxt_i \in \mathcal{C}$ 
4 proc  $init_i()$ 
5    $d_i := (i, 0)$ 
6    $ctxt_i := \emptyset$ 
7 proc  $tcbroadcast_i(msg)$ 
8    $tcdeliver_i(d_i, ctxt_i, msg)$ 
9    $enqueue_i^{cm}(msg, d_i, ctxt_i, msg)$ 
10   $d_i := (i, d_i.1 + 1)$ 
11   $ctxt_i := d$ 
12 on  $dequeue_i^{mc}(dlv, d, ctxt, msg)$ 
13   $ctxt_i := \{d\} \cup ctxt_i \setminus ctxt$ 
14   $tcdeliver_i(d, ctxt, msg)$ 
15 fun  $tcdeliver_i(d, ctxt, msg)$ 
     $/* \text{delivery callback} */$ 

```

We start by **Algorithm 4** which describes the client process. The client process at node i stores the dot dot_i as the id to be used for the next update, and the context $ctxt_i$ which holds the set of dots of the last concurrently delivered messages (after the transitive reduction) which would be the immediate predecessors of the next broadcast message.

$init_i()$ initializes the state with an initial dot $dot_i^0 \leftarrow (i, 0)$ and an empty context $ctxt_i^0$.

On $tcbroadcast_i(msg)$, the update is delivered locally by invoking the $tcdeliver_i()$ callback. The client process prepares a message using the dot saved in the state, gets C_i and the message payload and enqueues the prepared message by calling $enqueue_i^{cm}(dot_i, ctxt_i, msg)$ to be dequeued and broadcast at the middleware process. Now, the local dot is incremented and the local context $ctxt_i$ is updated by adding the last delivered dot, namely dot_i .

$dequeue_i^{mc}(deliver, dot, ctxt, msg)$ checks the arrival of any messages originating at other peers and marked as ready to be delivered by the middleware process. Whenever this queue has a message ready, if the message then the callback $tcdeliver_i(dot, ctxt, msg)$ is invoked to deliver the message to the application. Also the context C_i is updated by removing any dot in C_i that already exists in C_j of the message to be delivered, and adding $\{dot_j\}$. This is what makes the context at each client process hold only the immediate predecessor without any transitive predecessors.

ALGORITHM 5: Graph-based tagged causal delivery algorithm without causal stability at middleware process for node $i \in \mathbb{I}$

```

1 inputs:
2 | bit :  $\mathbb{I} \rightarrow \mathbb{N}$  /* bit mask given node id */
3 state:
4 |  $G_i : \mathcal{D} \leftrightarrow \mathcal{M}$  /* message graph */
5 |  $V_i : \mathbb{I} \rightarrow \mathbb{N}$  /* delivered version vector */
6 on  $\text{init}_i()$ 
7 |  $G_i := \emptyset$ 
8 |  $V_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
9 on  $\text{dequeue}_i^{cm}(\langle \text{msg}, d, P, m \rangle)$ 
10 |  $\text{broadcast}_i(\langle \text{msg}, d, P, m \rangle)$ 
11 |  $V_i[d.0] := d.1$ 
12 on  $\text{receive}_i(\langle \text{msg}, d, P, m \rangle)$ 
13 if  $V_i[d.0] < d.1 \wedge \neg(G_i[d].\text{stage} = \text{rcv})$  then
14 |  $P' = \{p \in P \mid p.1 > V_i[p.0]\}$ 
15 | for  $p$  in  $P'$  do
16 | | if  $p \notin \text{dom}(G_i)$  then
17 | | |  $G_i[p] := \{\text{stage} : \text{slt}, \text{succ} : \emptyset\}$ 
18 | | |  $G_i[p].\text{succ} := G_i[p].\text{succ} \cup \{d\}$ 
19 | | |  $b = \sum \text{bit}(p.0)$ 
20 | | |  $S = \text{if } d \in \text{dom}(G_i) \text{ then } G_i[d].\text{succ}$ 
21 | | | | else  $\emptyset$ 
22 | | |  $G_i[d] := \{\text{stage} : \text{rcv}, \text{bits} : b$ 
23 | | | | |  $\text{pred} : P', \text{succ} : S, \text{msg} : m\}$ 
24 | | | if  $b = 0$  then
25 | | | |  $\text{deliver}_i(d)$ 
26 proc  $\text{deliver}_i(d)$ 
27 |  $\text{enqueue}_i^m(\langle \text{dlv}, d, G_i[d].\text{pred}, G_i[d].\text{msg} \rangle)$ 
28 |  $(j, n) = d$ 
29 |  $V_i[j] := n$ 
30 |  $G_i[d] := G_i[d] \{\text{stage} : \text{dlv},$ 
31 | | |  $\text{bits} : \sim(\text{bit}(i) \mid \text{bit}(j))\}$ 
32 | for  $s$  in  $G_i[d].\text{succ}$  do
33 | |  $G_i[s].\text{bits} := G_i[s].\text{bits} \& \sim \text{bit}(j)$ 
34 | | if  $G_i[s].\text{bits} = 0$  then
35 | | |  $\text{deliver}_i(s)$ 

```

4.5.2 Middleware Process

We move now to the middleware process described in **Algorithm 5**. The middleware process of node i stores a data structure called \mathcal{G}_i which represents the immediate dependency graph, and holds causality metadata needed to ensure a correct delivery as well as the payload of the messages. Although, \mathcal{G}_i is a dependency graph, it is implemented as a hashmap that maps the globally-unique identifier of each message \mathcal{D} to its relative \mathcal{DS} data structure that holds the metadata and payload.

$\text{init}_i()$ initializes both the dependency graph \mathcal{G}_i and V_i , a version vector of delivered messages (used for simplicity).

On $\text{dequeue}_i^{cm}(\langle \text{msg}, d, P, m \rangle)$, The message is broadcast to all peers in the group, V_i is updated as the message is local and can be delivered. There is no need to add an entry in the dependency graph for this update because the V_i holds that information.

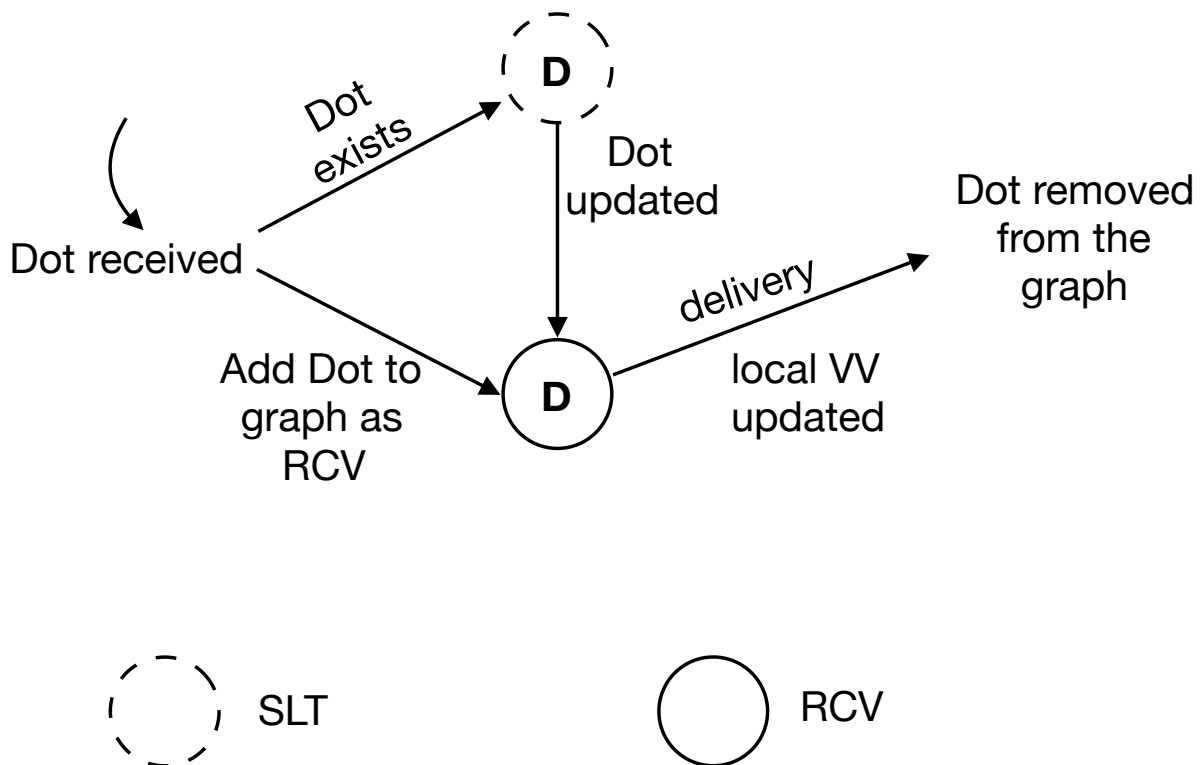


Figure 20: Stages of a dot in causal DAG (without causal stability)

In Figure 20, we show the different stages of a dot in our causal DAG without causal stability (at this point). A dot representing a single unique message can have three different stages: slt, rcv and dlv. When a dot in the causal DAG has a stage as slt it means that this dot has not been received yet. However, due to message delays and reordering, a dot that causally depends on it has arrived before it and this add was in its set of predecessors. Therefore we create a “slot” version of this dot only for it to exist as a predecessor for the arriving dot. A dot in a slt stage, becomes rcv upon its arrival or receipt by the middleware process. A dot does not have to go through the slt stage and could be directly be created and added as rcv upon its arrival. When all the predecessors of a dot have been delivered, the dot itself becomes ready to be delivered. Technically, upon delivery the dot is removed from the graph and therefore does not exist anymore in the graph with a stage dlw. We left that only to illustrate the stage at which the dot becomes delivered. In the next chapter, when we introduce causal stability, the dlw (as well as stb) is used in the graph.

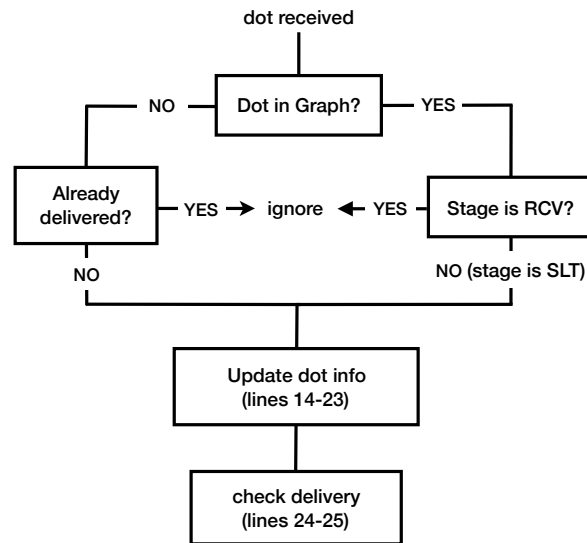


Figure 21: Flowchart showing how the algorithm works upon receiving a new message (without causal stability)

In Figure 21, the flowchart shows that upon receiving a dot, 4 possibilities exist. If the dot has been received before and therefore already added to the graph then it's a duplicate message and can be discarded with no additional action. This is the same case for when a dot is already delivered. And as seen in the algorithm, the version vector is used to check if a dot has been already delivered and in that case no action is taken. In the case where a dot has not been received or delivered before, it either does not exist in the graph and should be added or it was created as a slot and needs to be updated.

On $\text{receive}_i(\langle \text{msg } d, P, m \rangle)$, the message is received from a certain peer. The algorithm checks if this message has been received and waiting to be delivered or has been already delivered before to discard it. Otherwise, it adds it to the graph, and updates the DAG as needed. First, it checks if all its predecessors (that have not been delivered already) exist and creates "slot" placeholders otherwise. Then it adds the received dot d to the set of successors of all its predecessors. In the bitstring of the dot, it sets to 1 the corresponding bit of every predecessor that is not delivered yet. In the case where the dot d is a slot, it updates it and keeps the information it has about its successor. After adding and updating the dot and its predecessors accordingly, it checks if it can be delivered.

$\text{deliver}_i(d)$ delivers the dot d to the client process, updates the graph by removing the current dot entry and updating its successors and predecessors, and increments the V_i correspondingly. When a dot is delivered, the bitstring of all its successors is updated by setting to 0 the corresponding bit of the (provenance of the) delivered dot. Then it tries to deliver all successors of d that could have been made ready to be delivered.

Tagged causal delivery and Causal Stability

In this section, we introduce causal stability, explain its definition and how it differs from other stability related concept in distributed systems. We then explain how the implementation of causal stability works in both the vector-based and graph-based versions. Then we show a new version of the tagged causal delivery algorithms extended to provide causal stability, also in both vector-based and graph-based versions.

5.1 Causal Stability

Stability is one of the most overloaded terms in distributed systems. Examples are *stable storage*, for durable storage which survives process failure in a crash-recover model, or *self stabilization* [80] to refer to reaching some global predicate within a finite number of steps. For causal delivery middleware, Birman [4] defined *message stability* to mean that it is known that a message has been received at all intended nodes (and *k-stability* to mean that it has been received by *k* nodes). This concept is relevant for implementation purposes, namely for garbage collecting messages in a fault-tolerant system, as it allows a message to be discarded when it been delivered locally and it has become stable.

A different concept is *causal stability*, named as such in a paper that defines pure operation-based CRDTs [7], which make use of causal delivery. Causal stability concerns knowledge about the end-to-end happens-before regarding future deliveries at each client process.

Definition 1 (Causal Stability). *A causal timestamp t , and corresponding message, is causally stable at node i when all messages subsequently delivered at i will have causal timestamp $t' > t$, according to the end-to-end happens-before.*

This implies that no message with a timestamp t' concurrent with t can be delivered at i once t is causally stable at i . For a message to be causally stable at some node, not only is it needed to have been received everywhere, but also that it has been delivered everywhere, and that no further concurrent messages may be delivered at that node. Therefore, causal stability is a stronger notion, implying classic message stability. We notice that, depending on how knowledge is piggybacked, middleware can infer that some message has been received everywhere, therefore being stable, even if itself or other causally

concurrent messages have not yet been delivered. This notion differs from classic message stability in two fundamental ways:

- it is a per-node property (a message may be causally stable at node i but not at node j), while a message is stable if (it is known that) it was received everywhere;
- it is about message deliveries, therefore, a client-visible concept, relevant for the client API, while message stability is essentially an implementation aspect about internal middleware events.

So, even if there are some similarities, causal stability is a different concept, and we consider it important to use a different name, to avoid confusion with classic message stability. In fact, we have already observed this confusion many times, namely in discussions about implementing causal stability. The concept itself is not new, and it has been used many times, but hidden in applications, without being properly recognized (because it has not been clearly identified). This has led to successive ad-hoc re-implementations, sometimes overly complex. As an example, causal stability is hidden inside the difficult to understand implementation of Replicated Growable Arrays (RGAs) [6], while not being recognized as a building block. We argue that this concept is important enough to be recognized and provided by the middleware.

The TCD middleware can offer causal stability information through extending its API with a $tcstable_i(\tau)$ event, which informs the client application that message with timestamp τ is now causally stable locally. Node i can check this by verifying if a message with timestamp $t > \tau$ has already been delivered at i from every other node j in the set of nodes \mathbb{I} . More formally, if $deliver_i()$ represents the set of message timestamps that have been delivered at node i and $src(t)$ denotes the node that sent the message corresponding to t :

$$tcstable_i(\tau) \text{ if } \forall j \in \mathbb{I} \setminus \{i\} \cdot \exists t \in deliver_i() \cdot src(t) = j \wedge \tau < t.$$

One possible implementation of causal stability would be similar to the RGA tombstone deletion algorithm [6]. Essentially, each node only needs to keep the last causality timestamp delivered from each origin.

5.1.1 Causal Stability in the vector clock-based algorithm for TCD

Each node i keeps a map L_i (from node identifiers to vector clocks) with the last vector clock, delivered locally, from each other node. This allows defining a function low for the greatest lower bound on messages issued at j and delivered to all nodes:

$$low(L, j) \doteq \min(\{L(k)(j) | k \in \mathbb{I}\}).$$

This allows knowing at node i that, e.g., if $low(L_i, j) = 4$, then all nodes have delivered at least message number 4 from node j . Using this function we can now define a causal stability oracle. A timestamp τ is

causally stable at node i if

$$\tau(\text{src}(\tau)) \leq \text{low}(L_i, \text{src}(\tau)).$$

This ensures at i that message τ was already delivered at all nodes and that each node issued a message after delivering τ , that is already delivered at i . Since any messages concurrent to τ from any node k must have been issued prior to τ being delivery at k then, due to causal delivery, they must have also been delivered at node i .

5.1.2 Causal Stability in the graph-based algorithm for TC

In the graph-based implementation there is no need for keeping an extra matrix or other data structure for stability. In fact, the bit string used for checking causal delivery of a dot can be used for checking causal stability once the dot is marked as DLV. Every bit position in the bit string represents one of the peers in the membership. When the bit string reflects that every peer have delivered this dot it can become stable at that node. Using bit strings is efficient as it requires minimal space in memory, and the bitwise operations are not costly in time complexity.

Whenever a message from node j is delivered at i , we know that it has also been delivered at j as well as all its predecessors. Therefore, all the bit position of node j can be updated in the bit strings of that dot and every predecessor, immediate and transitive.

Marking a dot a stable by the middleware happens in an order respecting causal order, meaning that older dots get stabilized before their successors and so on. Whenever a dot is marked as stable, the client is notified to invoke the `tstable` callback and then the client informs the middleware that this dot has been stabilized and it is safe to be removed from the graph.

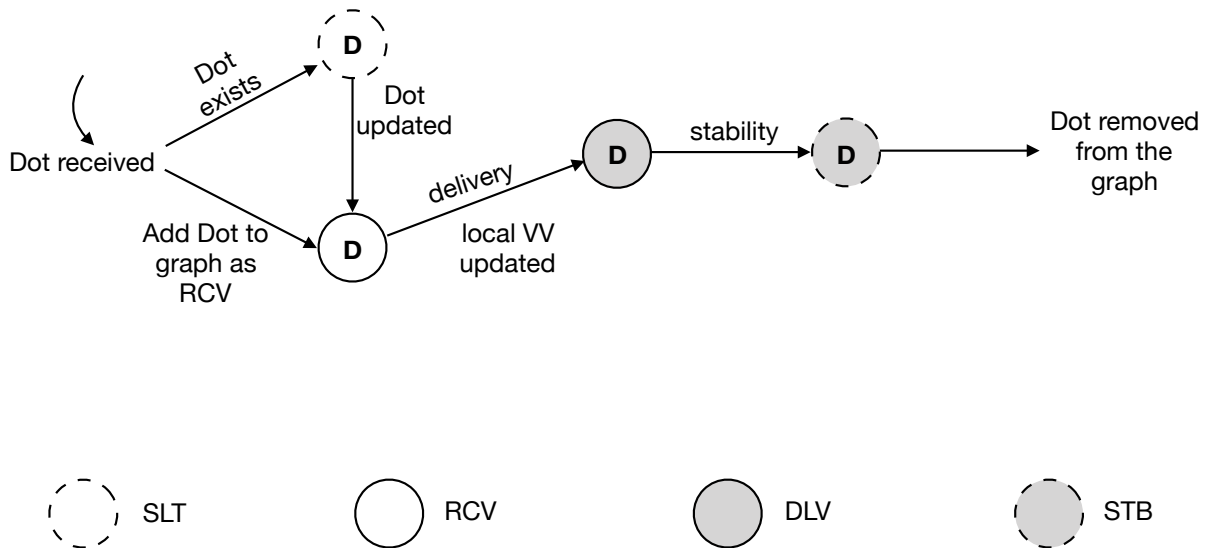


Figure 22: Stages of a dot in causal DAG (with causal stability)

In Figure 22, we tackle again the different stages of a dot in our causal DAG but now with causal stability. As we explained previously the different stages, we now add one more stage for causal stability

that we call stb. Moreover, we mentioned in the previous version without stability a dot does not stay in the graph with stage dlv. Here however, a dot stays in the graph after being delivered and the bitstring is used to compute when a dot is causally stable. Basically, whenever a dot is delivered, the algorithm updates the botstring of all its predecessors by setting to 0 the bit corresponding to the actor/node from which the delivered dot was sent originally. For instance, when a dot D_5 which represents the 5th message sent from node D is delivered all its predecessors of D_5 in the causal DAG at the receiving node update their bitstring values by setting the bit corresponding to node D to 0. When the bitstring value of delivered dot (stage dlv) is 0, it can be stabilized and its stage becomes stb. After a dot is stb it is ready to be stabilized by the client process. When the client process dequeues the stabilize message of a certain dot, the middleware can safely delete the stable dot from the causal DAG.

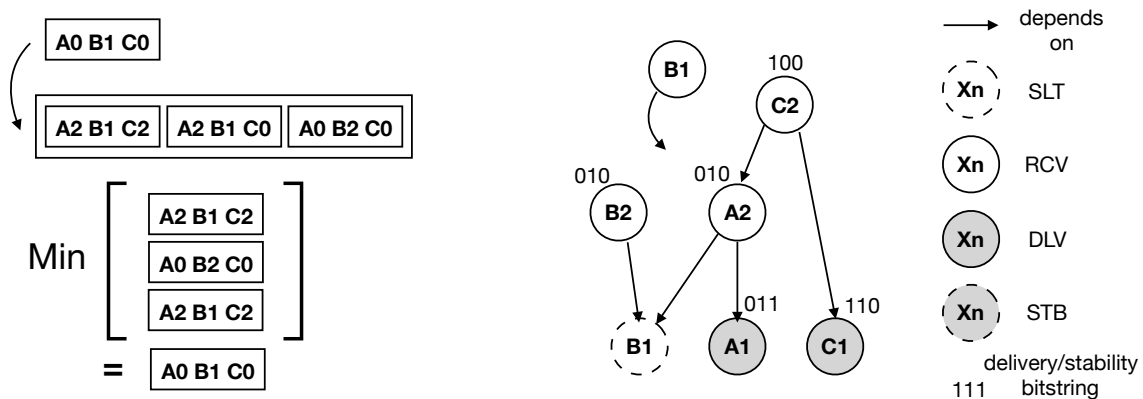


Figure 23: Graphical representation of causal delivery using a delivery queue and a causal DAG (with causal stability)

Figure 22 is also similar to Figure 20 from the previous chapter. Here we add to the previous explanation, the part related to causal stability. When using version vectors to represent and track causal dependency among messages, causal stability can be calculated by calculating the greatest lower bound of the version vectors in the stability matrix seen in Figure 22. The stability matrix holds at each entry/row the last delivered version vector of the node corresponding to that row. In the figure, the first row corresponds to node A , the second row to node B and the third to node C . The greatest lower bound allows to compute a stable version vector (SVV). The calculation of the SVV and causal stability in general is local to the node and not global, meaning that a message can be stable at A but not necessarily stable at B . At each node, when the SVV is computed, we know that it is less or equal to every last version vector delivered from every node. This means that any message that will be delivered after the SVV is computed at that node will be greater than the SVV. A message can be stabilized at a node if its version vector is less than the SVV at that node. As for the causal DAG-based version, the bitstring functionality changes whenever a dot is delivered to be used for stability. In the previous version where we did not use causal stability, the bitstring was only used to check if a dot is ready to be delivered when the bitstring is equal

to 0. Now that causal stability is used, the bitstring is used to check delivery when the stage is rcv and once delivered used similarly to check causal stability when the stage of the dot is dlv.

5.2 Causal Stability for VV-based Algorithm

5.2.1 Client process

The algorithm for the client process can be seen in **Algorithm 6**. The algorithm is very similar to the previous version without stability presented earlier in **Algorithm 2** with some incremental changes for causal stability. Therefore, we will only address the changes without repeating the previous explanation.

We notice that dequeue_i^{mc} is not only for dequeuing messages to be delivered. Instead it expects a type based on which it can either result in delivering a message stabilizing a message. If it is a delivery, the sender's entry on V_i is incremented before invoking the tcdeliver callback. Otherwise, if it is a stable message the tcstable callback is invoked.

ALGORITHM 6: Vector-based tagged causal delivery algorithm with causal stability at client process for node $i \in \mathbb{I}$

```

1 state:
2 |  $V_i : \mathbb{I} \rightarrow \mathbb{N}$  /* delivered version vector */
3 proc  $\text{init}_i()$ 
4 |  $V_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
5 proc  $\text{tcbcast}_i(\text{msg})$ 
6 |  $V_i := V_i[i] + 1$ 
7 |  $\text{tcdeliver}_i(i, V_i, \text{msg})$ 
8 |  $\text{enqueue}_i^{cm}(\text{msg}, V_i, \text{msg})$ 
9 on  $\text{dequeue}_i^{mc}(\text{type}, i, V_m, \text{msg})$ 
10 | if  $\text{type} = \text{dlv}$ 
11 | |  $V_i[j] := V_m[j]$ 
12 | |  $\text{tcdeliver}_i(j, V_m, \text{msg})$ 
13 | else if  $\text{type} = \text{stb}$ 
14 | |  $\text{tcstable}_i(j, V_m, \text{msg})$ 
15 fun  $\text{tcdeliver}_i(j, V_m, \text{msg})$ 
16 | | /* delivery callback */
17 fun  $\text{tcstable}_i(j, V_m, \text{msg})$ 
18 | | /* stabilization callback */

```

5.2.2 Middleware process

The algorithm for the *Middleware* process is shown in **Algorithm 7**. Also, this algorithm shares parts in common with its previous version without causal stability, presented in **Algorithm 3**. In this subsection,

we will only explain the new notations, data structures, functions and procedures that have not been explained previously.

ALGORITHM 7: Vector-based tagged causal delivery algorithm with causal stability at middleware process for node $i \in \mathbb{I}$

```

1  state:
2   $V_i : \mathbb{I} \rightarrow \mathbb{N}$  /* delivered VV *A5
3   $R_i : \mathbb{I} \rightarrow \mathbb{N}$  /* received VV *A6
4   $DQ_i :$  /* delivery queue *A7
5   $M_i : \mathbb{I} \rightarrow V_i$  /* stability matrix *A8
6   $SV_i : \mathbb{I} \rightarrow \mathbb{N}$  /* stable VV *A9
7   $SMap_i : \mathcal{D} \leftrightarrow \mathcal{M}$  /* map of stable dots *A50
8   $ctr_i : \mathbb{N}$  /* counter *A51

9  on  $init_i()$ 
10  $V_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
11  $R_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
12  $DQ_i := \emptyset$ 
13  $M_i := \{j \mapsto V_i \mid j \in \mathbb{I}\}$ 
14  $SV_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
15  $SMap_i := \emptyset$ 
16  $Min_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
17  $ctr_i := 0$ 

18 on  $dequeue_i^m(\langle msg, V, m \rangle)$ 
19  $V_i[i] := V_i[i] + 1$ 
20  $updatestability_i(i, V, m)$ 
21  $broadcast_i(\langle msg, i, V, m \rangle)$ 

22 on  $receive_i(\langle j, V_m, m \rangle)$ 
23 if  $R_i[j] < V_m[j]$  then
24  $R_i[j] := R_i[j] + 1$ 
25 if  $V_m[j] = V_i[j] + 1$ 
26  $\wedge (V_m[k] \leq V_i[k] \mid \forall k \neq j)$  then
27  $enqueue_i^m(\langle dlv, j, V_m, m \rangle)$ 
28  $updatestability_i(j, V_m, m)$ 
29  $deliver_i()$ 
30 else
31  $DQ_i := DQ_i + (j, V_m, m)$ 

32 proc  $updatestability_i(j, V_m, m)$ 
33  $M_i[i] := V_i$ 
34 if  $i \neq j$  then
35  $M_i[j] := V_m$ 
36  $ctr_i := Ctr_i + 1$ 
37  $SMap_i[(j, V_m[j])] := (V_m, m, ctr_i)$ 
38 if  $j \in Min_i$  then
39  $NewSV = calculateSV(j)$ 
40 if  $NewSV \neq SV_i$  then
41  $StableDots = NewSV - SV_i$ 
42  $stabilize_i(StableDots)$ 
43  $SV_i := NewSV$ 

44 proc  $deliver_i()$ 
45  $from := 0$ 
46  $to := 0$ 
47 while true do
48 if  $from \geq len(DQ_i)$  then
49 if  $to \geq from$  then
50  $break$ 
51  $truncate(DQ, to)$ 
52 if  $len(DQ_i) = 0$  then
53  $break$ 
54  $from := 0$ 
55  $to := 0$ 
56 else
57  $(j, v, v_m) = DQ[from]$ 
58 if  $v_m[j] = V_i[j] + 1 \wedge V_m[k] \leq V_i[k] \mid \forall k \neq j$  then
59  $V_i[j] = V_i[j] + 1$ 
60  $enqueue_i^m(\langle dlv, j, V_m, m \rangle)$ 
61  $updatestability_i(j, V_m, m)$ 
62 else
63  $DQ[from] := DQ[to]$ 
64  $to := to + 1$ 
65  $from := from + 1$ 

66 proc  $stabilize_i(SD)$ 
67  $L = \{SMap_i[d] \mid d \in SD\}$ 
68  $S = sort(L, |x, y| x.2 < y.2)$ 
69 for  $(v, m, c) \in S$  do
70  $enqueue_i^{mc}(\langle stb, v, m \rangle)$ 
71  $SMap_i := SD \triangleleft SMap_i$ 

72 fn  $calculateSV(j)$ 
73  $newSV := SV_i$ 
74 for  $col = 0..n$  do
75 if  $Min_i[col] = j$  then
76  $min := M_i[0][col]$ 
77  $min_{row} := 0$ 
78 for  $row = 1..n$  do
79 if  $M[row][col] < min$  then
80  $min := M_i[row][col]$ 
81  $min_{row} := row$ 
82  $newSV[col] := min$ 
83  $Min_i[col] := min_{row}$ 
84 return  $newSV$ 

```

For determining causal stability, the middleware has a stability matrix M_i , a stable version vector SV_i , a map of stable dots $SMap_i$, a vector Min_i and a counter ctr_i . For a group with N peers, each middleware

has a $N \times N$ matrix M_i , where row j is the version vector of the last delivered message from node j . SV_i is a vector with N entries, where each position is the minimum of the same column in the matrix M_i . With the minimum of each column it is possible to know what is stable (from the perspective of i that has M_i) by comparing the message's version vector with these minimums. The Min_i is vector where each entry is the row with minimum of the corresponding column of M_i . Delivered but not yet stable message are saved in the $SMap_i$. The ctr_i field is an integer that is used to order stable messages before enqueueing them to the *Client*. When a message m_j is delivered, row $M_i[j]$ is updated with its version vector, and it is also added to the map $SMap_i$. Calculating the SV_i vector every time M_i is updated can become costly, specially when dealing with large groups. To overcome this problem the Min_i vector was created, by checking if the sender's id is in it. If it is not, then the minimums of the columns are the same and SV_i has not changed. Otherwise, it means that the row where the minimum was has changed, and so the could the minimum have also changed. Therefore, for each entry k on Min_i where $Min_i[k]$ is the same as the sender's id, the column k of M_i is traversed to check for the minimum and a row with that minimum. Then every entry in the current SV_i with the sender's id is updated and if the new vector is not the same as the previous, it means that there are new stable messages. The entries that are different when comparing these stable vectors are the new stable messages. For example, for a group with 4 peers, if the previous and new SV_i vector were, respectively, [15, 9, 10, 12] and [16, 9, 10, 13], then the message with id 16 from peer 0 and the message with id 13 from peer 3 became stable. By having the rows with the minimums instead of calculating them at each update reduces overhead and time on calculations.

5.3 Causal Stability for graph-based TCB Algorithm

5.3.1 Client Process

ALGORITHM 8: Graph-based tagged causal delivery algorithm with causal stability at client process for node $i \in \mathbb{I}$

```

1 state:
2    $d_i \in \mathcal{D}$ 
3    $ctxt_i \in \mathcal{C}$ 
4 proc  $init_i()$ 
5    $d_i := (i, 0)$ 
6    $ctxt_i := \emptyset$ 
7 proc  $tcbroadcast_i(msg)$ 
8    $tcdeliver_i(d_i, ctxt_i, msg)$ 
9    $enqueue_i^{cm}(msg, d_i, ctxt_i, msg)$ 
10   $d_i := (i, d_i.1 + 1)$ 
11   $ctxt_i := d$ 
12 on  $dequeue_i^{mc}(type, d, ctxt, msg)$ 
13  if  $type = dlv$ 
14     $ctxt_i := \{d\} \cup ctxt_i \setminus ctxt$ 
15     $tcdeliver_i(d, ctxt, msg)$ 
16  else if  $type = stb$ 
17     $tcstable_i(d, ctxt, msg)$ 
18     $enqueue_i^{cm}(stb, d)$ 
19 fun  $tcdeliver_i(d, ctxt, msg)$ 
20    $/* delivery callback */$ 
21 fun  $tcstable_i(d, ctxt, msg)$ 
22    $/* stabilization callback */$ 

```

The algorithm for the client process can be seen in **Algorithm 8**. The algorithm is very similar to the previous version without stability presented earlier in **Algorithm 4** with some incremental changes for causal stability. Therefore, we will only address the changes without repeating the previous explanation.

$dequeue_i^{mc}(type, dot, ctxt, msg)$ checks the arrival of any messages at the delivery/stability queue. Whenever this queue has a message ready, if the message is a delivery message then the callback $tcdeliver_i(dot, ctxt, msg)$ is invoked to deliver the message to the application. Also the context C_i is updated by removing any dot in C_i that already exists in C_j of the message to be delivered, and adding $\{dot_j\}$. This is what makes the context at each client process hold only the immediate predecessor without any transitive predecessors. In the case where the message dequeued is a stability message, $tcstable_i(dot, ctxt, msg)$ callback is invoked and the dot is stabilized at the application. Here the context is not updated. Finally, a message is enqueued back to the middleware process signaling that the dot has been stabilized and it can be deleted from the graph safely.

5.3.2 Middleware Process

ALGORITHM 9: Graph-based tagged causal delivery algorithm with causal stability at middleware process for node $i \in \mathbb{I}$

```

1  inputs:
2  | bit :  $\mathbb{I} \rightarrow \mathbb{N}$           /* bit mask given node id */
3  state:
4  |  $G_i : \mathcal{D} \hookrightarrow \mathcal{M}$           /* message graph */
5  |  $V_i : \mathbb{I} \rightarrow \mathbb{N}$       /* delivered version vector */
6  on  $\text{init}_i()$ 
7  |  $G_i := \emptyset$ 
8  |  $V_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
9  on  $\text{dequeue}_i^m(\langle \text{msg}, d, P, m \rangle)$ 
10 |  $\text{broadcast}_i(\langle \text{msg}, d, P, m \rangle)$ 
11 |  $V_i[d.0] := d.1$ 
12 |  $P' = \{p \in P \mid \neg \text{stable}_i(p)\}$ 
13 | for  $p$  in  $P'$  do
14 |   |  $G_i[p].\text{succ} := G_i[p].\text{succ} \cup \{d\}$ 
15 |   |  $G_i[d] := \{\text{stage} : \text{dlv}, \text{bits} : \sim \text{bit}(i),$ 
16 |   |   |  $\text{pred} : P', \text{succ} : \emptyset, \text{msg} : m\}$ 
17 |   |  $\text{updatestability}_i(\sim \text{bit}(i), d)$ 
18 on  $\text{receive}_i(\langle \text{msg}, d, P, m \rangle)$ 
19 | if  $V_i[d.0] < d.1 \wedge \neg(G_i[d].\text{stage} = \text{rcv})$  then
20 |   |  $P' = \{p \in P \mid \neg \text{stable}_i(p)\}$ 
21 |   | for  $p$  in  $P'$  do
22 |     | if  $p \notin \text{dom}(G_i)$  then
23 |       |  $G_i[p] := \{\text{stage} : \text{slt}, \text{succ} : \emptyset\}$ 
24 |       |  $G_i[p].\text{succ} := G_i[p].\text{succ} \cup \{d\}$ 
25 |       |  $b = \sum \{\text{bit}(p.0) \mid p \in P' \wedge G_i[p] \neq \text{dlv}\}$ 
26 |       |  $S = \text{if } d \in \text{dom}(G_i) \text{ then } G_i[d].\text{succ}$ 
27 |       |   |  $\text{else } \emptyset$ 
28 |       |  $G_i[d] := \{\text{stage} : \text{rcv}, \text{bits} : b$ 
29 |       |   |  $\text{pred} : P', \text{succ} : S, \text{msg} : m\}$ 
30 |       | if  $b = 0$  then
31 |         |  $\text{deliver}_i(d)$ 
32 on  $\text{dequeue}_i(\langle \text{stb}, d \rangle)$ 
33 |  $\text{deletestable}_i(d)$ 
34 fn  $\text{stable}_i(d)$ 
35 |  $d.1 \leq V_i[d.0] \wedge (d \notin \text{dom}(G_i) \vee G_i[d].\text{stage} = \text{stb})$ 
36 proc  $\text{deliver}_i(d)$ 
37 |  $\text{enqueue}_i^m(\langle \text{dlv}, d, G_i[d].\text{pred}, G_i[d].\text{msg} \rangle)$ 
38 |  $(j, n) = d$ 
39 |  $V_i[j] := n$ 
40 |  $G_i[d] := G_i[d] \{\text{stage} : \text{dlv},$ 
41 |   |  $\text{bits} : \sim(\text{bit}(i) \mid \text{bit}(j))\}$ 
42 |  $\text{updatestability}_i(\sim \text{bit}(j), d)$ 
43 | for  $s$  in  $G_i[d].\text{succ}$  do
44 |   |  $G_i[s].\text{bits} := G_i[s].\text{bits} \& \sim \text{bit}(j)$ 
45 |   | if  $G_i[s].\text{bits} = 0$  then
46 |     |  $\text{deliver}_i(s)$ 
47 proc  $\text{updatestability}_i(b, d)$ 
48 | for  $p$  in  $G_i[d].\text{pred}$  do
49 |   | if  $G_i[p].\text{stage} \neq \text{stb}$  then
50 |     |  $b' = G_i[p].\text{bits} \& b$ 
51 |     | if  $b' \neq G_i[p].\text{bits}$  then
52 |       |  $G_i[p].\text{bits} := b'$ 
53 |       | if  $b' = 0$  then
54 |         |  $\text{stabilize}_i(p)$ 
55 |       | else
56 |         |  $\text{updatestability}_i(b, p)$ 
57 proc  $\text{stabilize}_i(d)$ 
58 |  $P = G_i[d].\text{pred}$ 
59 | for  $p$  in  $P$  do
60 |   | if  $G_i[p].\text{stage} \neq \text{stb}$  then
61 |     |  $\text{stabilize}_i(p)$ 
62 |   |  $G_i[d].\text{stage} := \text{stb}$ 
63 |   |  $\text{enqueue}_i^m(\langle \text{stb}, d, P, G_i[d].\text{msg} \rangle)$ 
64 proc  $\text{deletestable}_i(d)$ 
65 | for  $s$  in  $G_i[d].\text{succ}$  do
66 |   |  $G_i[s].\text{pred} := G_i[s].\text{pred} \setminus \{d\}$ 
67 |   |  $G_i := d \triangleleft G_i$ 

```

The algorithm for the *Middleware* process is shown in **Algorithm 9**. Also, this algorithm shares parts in common with its previous version without causal stability, presented in **Algorithm 5**. In this subsection, we will only explain the new notations, data structures, functions and procedures that have not been explained previously. As mentioned before, the causal relations between messages are maintained through a graph, specifically a *Directed Acyclic Graph* or *DAG*. The middleware has as state a version vector V_i and a *DAG* G_i .

The number of entries on V_i is the same as the number of peers in the group and each of them represent the *cntr* field from the dot of the last delivered message from the peer corresponding to that

position. Initially, V_i starts with 0 in all positions and an empty G_i . A node in G_i represents a message and has five parameters: *stage*, *bits*, *pred*, *succ* and *msg*.

The *stage* is an *enum* of the node's current state and can either be as a slot *SLT*, received *RCV*, delivered *DLV* or stable *STB*. A *SLT* node is a message that has not yet been received but it is a predecessor of another received message that has added it to the graph to serve as a placeholder. A node marked as *RCV* means that the message has been received but not delivered; if it is marked as *DLV* then it has been delivered but it is not stable and only it is considered as such when it has stage *STB*.

Every node has a *bits* field that is a string of bits with the same size as the number of peers in the group. Depending on the node's stage these bits are used either to determine if a message can be delivered or considered stable. If a bit is 1 while the node has stage *RCV* it means that a causal predecessor from the peer in that position has not been delivered. When a predecessor is delivered, the bit position of the sender in their successors are set to 0. Only when all the bits of a node are 0 can the message be delivered. Otherwise, when a message is marked as *DLV* this string is used to determine causal stability. After delivery, the *bit* field starts with all elements as 1 (except the local and sender peers positions) and the delivery of a message successor to the node sets a bit to 0 in the respective position of the sender peer. The bits of peer i and the sender peer are 0 after delivery because the node's message already is in the past of their next message. Besides these, a node also has a *pred* and *succ* field. These are, respectively, the dots of the message's causal predecessors and successors. Furthermore, the *msg* is the message that a peer sent.

When the *Client* process calls *tcbcast*, a message m is enqueued to *tcbcast*, along with a dot d and a list of predecessor P as a context. The *Middleware* then calls the $dequeue_i^m$ callback and dequeues from *tcbcast*. First the message is sent to be broadcast with the *broadcast* callback. Then the peer's entry on V_i is updated with the *cntr* of the message's dot. After this the message's predecessors that are not stable are calculated. As shown in the *stable* function of the algorithm, a message is stable if it has been delivered (its sender's entry on V_i is equal or less than the *cntr* from its dot) and either its stage on the graph is *STB* or it has already been removed from the graph. The message's dot is then added to the *succ* field of its predecessors that are not stable. After that the message with stage *DLV*, with the i bit as 0 and all the others as 1 is added to G_i , calling in the end *updatestability_i*.

If the *Middleware* receives a message from a peer in the group *receive_i* is called. A message can only be considered as received if the *cntr* field on its dot is greater than the sender's entry on V_i and if its stage on the DAG is not received. With this only messages that are not in G_i or have a placeholder node with stage *SLT* can be added to the graph. Then this function calculates from the predecessors P of the message the ones that are not stable. The received message's dot is added to its predecessors' *succ* field and if a predecessor has not yet been received then a node with stage *SLT* is added to the graph as a placeholder. A bit string with entry 0 as a delivered predecessor (or not a predecessor at all) from that peer or 1 as a predecessor that has not been delivered is created. In the algorithm, S is the list of the successors' dots of the received message.

In case the message's node previously had stage *SLT* then the list remains the same as the *succ* field,

otherwise S is an empty list. The message's fields are updated and added to the graph (if not already there) and if bit has all its bits as 0 then the message can be delivered. Upon delivery, the *Middleware* sends the message to the *Client*, its dot and the predecessors' dots through the $tcbdelivery$ queue. The node's stage is updated to *DLV* marking the message as delivered and the bit string is used for calculating stability. The sender's and the i entries are set to 0 while the remaining are set to 1. Then the stability of the node is updated. After that the bit in the bit string in sender's position of all the node's successors is set to 0 and if the successor's bit field has all the bits to 0 then its message can be delivered. This means that a delivery can cause other deliveries and $deliver_i$ is called recursively to traverse the graph and ensure that.

When a message is broadcast or delivered, the stability of its predecessors is updated. Starting from the node, the graph is traversed in the direction of its predecessors and setting recursively the bit in the sender's position to 0. This indicates that a message ahead of the node has been delivered. If that node's bit mask already had the bit in that position as 0 then the recursive call stops. Otherwise, if the bit field only has bits 0 then the message is considered stable because of the previous delivery and $stabilize$ is called. If not, $updatestability$ is called to each predecessor of the node. Upon calling $stabilize$ on a message, its predecessors are also marked as stable $stabilize$ is called on them and these are sent to the *Client* as stable first.

If the *Middleware* dequeues a stable dot from the *Client* it means that a message was acknowledged as stable. Therefore it can be removed from the graph by calling $delestable$. This function removes the stable dot from the $pred$ field of its successors in the graph and its node is also removed from the graph.

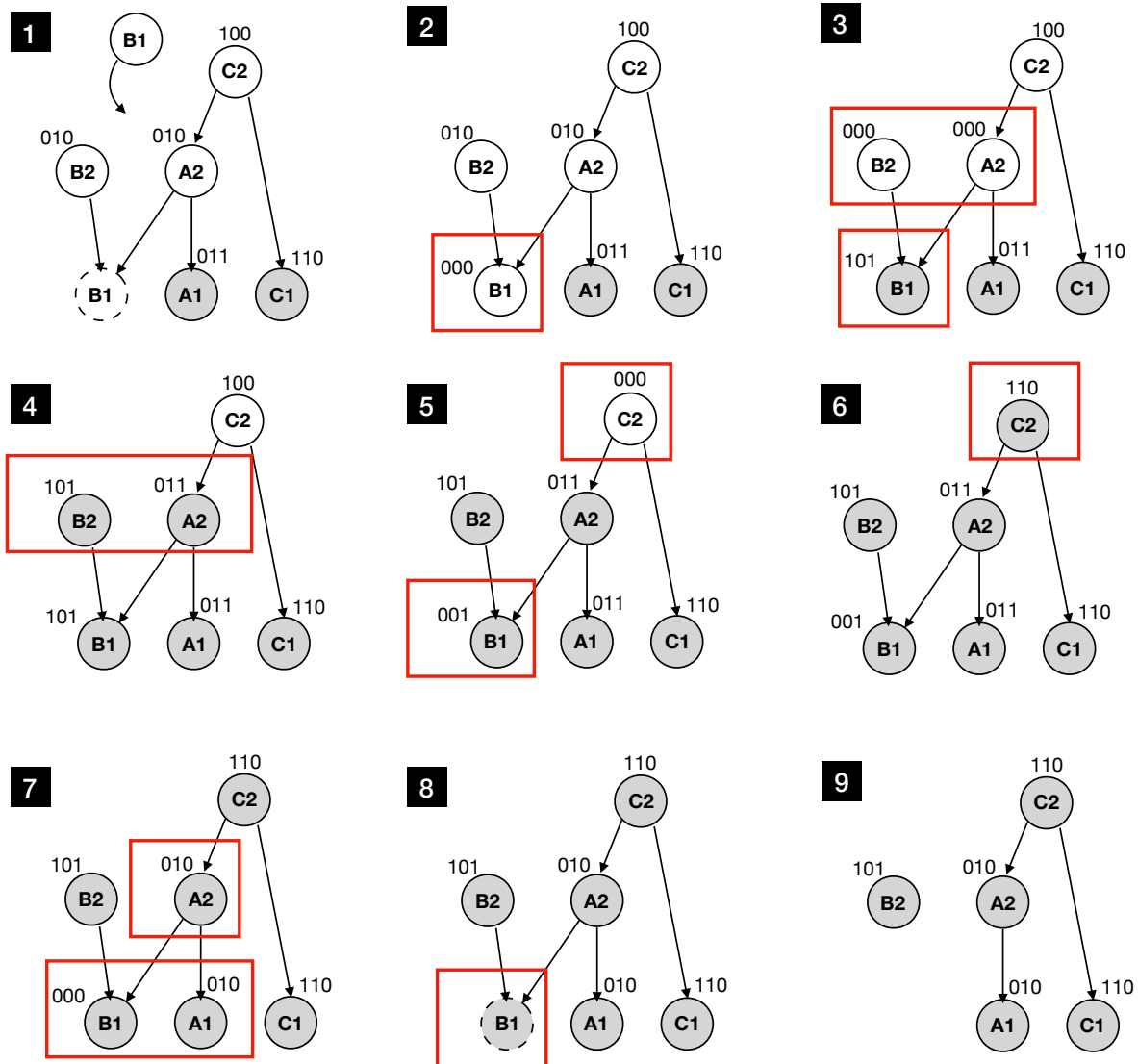


Figure 24: Evolution of the causal DAG (with causal stability)

We show in Figure 24 an example illustrating the evolution of the causal DAG. We start at the first quadrant, with previously shown example. Dot $B1$ is received at the node with the current causal DAG. We can see that a $B1$ placeholder exists with the stage slt because $B2$ was received before $B1$ and therefore has created this placeholder for its dependency $B1$. Also, the bitstrings for the dots having stage rcv are used for checking delivery ($A2$, $B2$, $C2$) and for checking stability for the dots having the stage dlv ($A1$ and $C1$). In every bitstring the most significant bit corresponds to node A , the second to node B and the least significant to node C . An consequently we can see that dot $B2$ for example having stage rcv is waiting for the delivery of a message from node B because its bitstring holds the value 010 indicating that. Same for $C2$ waiting for $A2$ to be delivered which is reflected in its bitstring 100 reflecting that a message from node A needs to be delivered. Similarly for dots having stage dlv , the dot $A1$ having bitstring 011 means that $A1$ can be stabilized when a causally succeeding dot from node B and another from node C need to be delivered first. At the second quadrant, when $B1$ is received it is added to the graph. Having

no dependencies, it can be delivered immediately and allows the delivery of its successors $A2$ and $B2$ to be delivered as well. At quadrant 4 we notice that the bitstrings of the delivered dots $A2$ and $B2$ now are updated for checking causal stability. In quadrant 5, the delivery of $A2$ triggers the delivery of its successor $C2$. And the bitstring of $B1$ changes from 101 to 001 due to the delivery of a causally succeeding dot from node A , $A2$. In quadrant 6, $C2$ is delivered and its delivery updates the bitstring values of $B2$, $A2$ and $B1$ setting to 0 the least significant bit corresponding to C . Now that $B1$ is delivered and has a bitstring of value 0, it is stabilized as seen in quadrant 8. When the client receives the stable message of $B1$ it asks the middleware process to delete $B1$ from the DAG and update it correspondingly.

5.4 Phantom Dots: An Optimisation for Active/Passive Node

In the previous sections we presented a new version of the algorithm that uses causal stability. As explained causal stability uses the causality information sent in messages as metadata to infer what message can become causally stable and therefore compact the causal graph. This improves the performance of our middleware by reducing the memory needed to store the causal graphs and the time needed to traverse and update the causal DAG. As this knowledge is inferred from messages sent, the rate and number of messages sent affects the the speed at which dot in the causal DAG become causally stable. Moreover, causal stability is done locally and therefore it needs information from all the nodes in the group. All this means that when nodes become less active, slow or completely passive, causal stability is slow and therefore the performance of our system drops. In the case where it's a small group of node and most of them are active at least at some point then the performance is not affected too much. However, when the number of nodes grows most of the nodes are readers and only a few are senders. If a node is not sending and only receiving, then stability cannot advance because that node is not sending the metadata information about what it has delivered to other nodes, which required for causal stability. In this section we address this problem by adding the option of explicitly send the metadata required for causal stability. We call those messages phantom messages as they are not added to the causal DAG and do not really affect the original algorithm. We present the algorithms at both the client process and middleware process for this version.

5.4.1 Client Process

We start from the algorithm seen in **Algorithm 10** and extend it to include sending, receiving and updating the causal graph based on the knowledge propagated using the phantom dots. At the client process, the algorithm is mostly the same as the previous version with causal stability presented earlier in **Algorithm 8**. One additional procedure `stabPhantomi()` is added which would get the current client state comprised of the current dot and the context and enqueue it on the communication queue from the client to the middleware. This procedure basically allows the application to send causality information to the

middleware process without any other payload to update the stability information in the causal dependency graph.

ALGORITHM 10: Graph-based tagged causal delivery algorithm with causal stability and phantom messages at client process for node $i \in \mathbb{I}$

```

1 state:
2    $d_i \in \mathcal{D}$ 
3    $ctxt_i \in \mathcal{C}$ 
4 proc  $init_i()$ 
5    $d_i := (i, 0)$ 
6    $ctxt_i := \emptyset$ 
7 proc  $tcbroadcast_i(msg)$ 
8   /* deliver/apply the msg */
9    $tcdeliver_i(d_i, ctxt_i, msg)$ 
10   $enqueue_i^{cm}(msg, d_i, ctxt_i, msg)$ 
11   $d_i := (i, d_i.1 + 1)$ 
12   $ctxt_i := d$ 
13 proc  $stabPhantom_i()$ 
14  $enqueue_i^{cm}(phantom, d_i, ctxt_i)$ 
15 on  $dequeue_i^{mc}(d, ctxt, msg)$ 
16    $ctxt_i := \{d\} \cup ctxt_i \setminus ctxt$ 
17    $tcdeliver_i(d, ctxt, msg)$ 
18 on  $dequeue_i^{mc}(d, ctxt)$ 
19    $tcstable_i(d_j, ctxt_j, msg)$ 
20    $enqueue_i^{cm}(stb, d_i)$ 
21 fun  $tcdeliver_i(d, ctxt, msg)$ 
22   /* delivery callback */
23 fun  $tcstable_i(d, ctxt)$ 
24   /* stabilization callback */

```

5.4.2 Middleware Process

Similarly, the algorithm for the *Middleware* process is shown in **Algorithm 11**. Also, this algorithm shares most parts in common with its previous version with causal stability, presented in **Algorithm 9**. In this subsection, we will only focus on functions and procedures that have been added for the phantom dots and causal stability updates. The main change done here is adding $dequeue_i(\langle phantom, d, P \rangle)$ to handle receiving a phantom dot request and update the local causal stability information based on the metadata in (d, P) . As this was not part of the algorithm in the previous version, we created the procedure $checkstability_i(d, P)$ that takes the information sent and updates the causal stability information in dot d and all its predecessors as well as mark dots as causally stable dots when it is the case. To avoid redundancy of checking and updating stability in our algorithm, some changes were made in the part of

our algorithm which is common with the previous version. For example, calling $\text{checkstability}_i(d, P)$ at line 30, as well as updating $\text{updatestability}_i(d, P)$ and $\text{stabilize}_i(d)$ accordingly.

ALGORITHM 11: Graph-based tagged causal delivery algorithm with causal stability and phantom messages at middleware process for node $i \in \mathbb{I}$

```

1  inputs:
2  | bit :  $\mathbb{I} \rightarrow \mathbb{N}$           /* bit mask given node id */
3  state:
4  |  $G_i : \mathcal{D} \leftrightarrow \mathcal{M}$           /* message graph */
5  |  $V_i : \mathbb{I} \rightarrow \mathbb{N}$           /* delivered version vector */
6  on  $\text{init}_i()$ 
7  |  $G_i := \emptyset$ 
8  |  $V_i := \{j \mapsto 0 \mid j \in \mathbb{I}\}$ 
9  on  $\text{dequeue}_i^m(\langle \text{msg}, d, P, m \rangle)$ 
10 |  $\text{broadcast}_i(\langle \text{msg}, d, P, m \rangle)$ 
11 |  $V_i[d.0] := d.1$ 
12 |  $P' = \{p \in P \mid \neg \text{stable}_i(p)\}$ 
13 | for  $p$  in  $P'$  do
14 |    $G_i[p].\text{succ} := G_i[p].\text{succ} \cup \{d\}$ 
15 |    $G_i[d] := \{\text{stage} : \text{dlv}, \text{bits} : \sim 0,$ 
16 |      $\text{pred} : P', \text{succ} : \emptyset, \text{msg} : m\}$ 
17 |    $\text{updatestability}_i(\sim \text{bit}(i), d)$ 
18 on  $\text{receive}_i(\langle \text{msg}, d, P, m \rangle)$ 
19 | if  $V_i[d.0] < d.1 \wedge \neg(G_i[d].\text{stage} = \text{rcv})$  then
20 |    $P' = \{p \in P \mid \neg \text{stable}_i(p)\}$ 
21 |   for  $p$  in  $P'$  do
22 |     if  $p \notin \text{dom}(G_i)$  then
23 |        $G_i[p] := \{\text{stage} : \text{slt}, \text{succ} : \emptyset\}$ 
24 |        $G_i[p].\text{succ} := G_i[p].\text{succ} \cup \{d\}$ 
25 |        $b = \sum\{\text{bit}(p.0) \mid p \in P' \wedge G_i[p] \neq \text{dlv}\}$ 
26 |        $S = \text{if } d \in \text{dom}(G_i) \text{ then } G_i[d].\text{succ}$ 
27 |         else  $\emptyset$ 
28 |        $G_i[d] := \{\text{stage} : \text{rcv}, \text{bits} : b$ 
29 |          $\text{pred} : P', \text{succ} : S, \text{msg} : m\}$ 
30 |        $\text{checkstability}_i(d, P')$ 
31 |       if  $b = 0$  then
32 |          $\text{deliver}_i(d)$ 
33 on  $\text{dequeue}_i(\langle \text{phantom}, d, P \rangle)$ 
34 |  $\text{checkstability}_i(d, P)$ 
35 on  $\text{dequeue}_i(\langle \text{stb}, d \rangle)$ 
36 |  $\text{deletestable}_i(d)$ 
37 proc  $\text{deliver}_i(d)$ 
38 |  $\text{enqueue}_i^m(\langle \text{dlv}, d, G_i[d].\text{msg} \rangle)$ 
39 |  $(j, n) = d$ 
40 |  $V_i[j] := n$ 
41 |  $G_i[d] := G_i[d]\{\text{stage} : \text{dlv},$ 
42 |    $\text{bits} : \sim \text{bit}(i)\}$ 
43 |  $\text{updatestability}_i(\sim \text{bit}(j), d)$ 
44 | for  $s$  in  $G_i[d].\text{succ}$  do
45 |    $G_i[s].\text{bits} := G_i[s].\text{bits} \& \sim \text{bit}(j)$ 
46 |   if  $G_i[s].\text{bits} = 0$  then
47 |      $\text{deliver}_i(s)$ 
48 fn  $\text{stable}_i(d)$ 
49 |  $d.1 \leq V_i[d.0] \wedge (d \notin \text{dom}(G_i) \vee G_i[d].\text{stage} = \text{stb})$ 
50 proc  $\text{checkstability}_i(d, P)$ 
51 | if  $d.1 = 1 + V_i[d.0]$  then
52 |   for  $p \in P$  do
53 |      $\text{updatestability}_i(\sim \text{bit}(d.0), p)$ 
54 proc  $\text{updatestability}_i(b, d)$ 
55 | if  $G_i[d].\text{stage} = \text{dlv}$  then
56 |    $b' = G_i[d].\text{bits} \& b$ 
57 |   if  $b' \neq G_i[d].\text{bits}$  then
58 |      $G_i[d].\text{bits} := b'$ 
59 |     if  $b' = 0$  then
60 |        $\text{stabilize}_i(d)$ 
61 |     else
62 |       for  $p$  in  $G_i[d].\text{pred}$  do
63 |          $\text{updatestability}_i(b, p)$ 
64 proc  $\text{stabilize}_i(d)$ 
65 | for  $p$  in  $G_i[d].\text{pred}$  do
66 |   if  $G_i[p].\text{stage} \neq \text{stb}$  then
67 |      $\text{stabilize}_i(p)$ 
68 |    $G_i[d].\text{stage} := \text{stb}$ 
69 |    $\text{enqueue}_i^m(\langle \text{stb}, d \rangle)$ 
70 proc  $\text{deletestable}_i(d)$ 
71 | for  $s$  in  $G_i[d].\text{succ}$  do
72 |    $G_i[s].\text{pred} := G_i[s].\text{pred} \setminus \{d\}$ 
73 |    $G_i := d \triangleleft G_i$ 

```

Dynamic Membership

In the previous chapters, we discussed our Tagged Causal Delivery Algorithms considering static membership. This means that in our system model so far, the group membership of our system is a fixed-sized set of all the nodes in the system. However, realistic use cases require distributed systems to allow new nodes to join, and existing nodes to leave. Obviously, the correctness of the system and its computation should not be affected. In this chapter, we explain how we extended our protocol to support dynamic environments, where new nodes can join, and existing nodes can leave, all maintaining the correctness of tagged causal delivery and stability.

6.1 Causal Stability in Dynamic Membership

In the previous chapters, we explained what causal stability is and how it provides the system when an operation or a message is stable at a certain node or multiple nodes. We use causal stability in the same fashion to extend our system from a static group membership to dynamic group membership. This allows the existing nodes in the system to continue working normally without blocking whenever a new node join or an existing one leaves. Therefore causal stability allows us to provide a non-blocking dynamic membership to our Tagged Causal Delivery middleware.

We consider every join request and every leave request as every other operation in the system. It has a dot as an identifier, it is issued by a node in the group membership and it is added to the causal DAG. Certainly, there some conditions that apply specifically to join or leave operations in addition to that, as they are special operations than the rest. We explain in this section the logic behind using causal stability to ensure the non-blocking join and leave operations. We defer the detailed explanation of the dynamic version of our algorithm to Section [6.2](#).

6.1.1 Causal Stability and Join Requests

Our system model requires that every node in the group membership has the knowledge of the full group membership. This is mainly needed for causal stability to work correctly. When a new node joins, the group

membership will eventually change and all the existing nodes need to know about the new membership. In addition to that, the joining node needs to have a correct and up-to-date state before starting to send and receive messages for the existing peers. One way to do that is to block the system until the new node joins, receives the the updates needed to have a correct state and inform every existing node of the new node. This might not seem costly if joins are rare. However, in dynamic systems, many nodes might be joining or leaving continuously which makes blocking the system very costly. Causal Stability allows new nodes to join without blocking the existing nodes from continuing their distributed computation or sending and receiving requests.

Every new node that wants to join the group send a request to an existing node in the group membership along with its unique ID. The contacted node handles this request and is responsible of making sure that the new node becomes a member of the group safely. Upon receiving the request, the contacted node assigns a dot (of his own) to the join request, its context containing the causal dependencies and adds it to its local causal DAG as delivered. It also updates its membership locally to include the new node. Next, it collects the application data needed as well as causality related meta data that causally precedes the join request and sends it back to the new joining node to initialize its state. The new node initializes its state and spawns a middleware process. Until this point the new node cannot safely send or broadcast any messages to the group, and it only receives messages from the contacted node. Now the contacted node broadcasts the join request as any normal operation to every node in the membership. It also keeps forwarding for the new node, all the previous messages it has received and that were concurrent to the join request. It keeps forwarding so the new node can receive all the messages and stay up-to-date. When the other nodes deliver the join request, they update their membership to include the new node which will starts receiving the broadcasts. Until this point, the new node only receives the messages but cannot send or broadcast yet. When the join request is stable at the contacted node, the latter knows that every node in the group membership has delivered the join request and therefore all the new broadcasts will also be received by the new node. Thus, there is no need to forward messages anymore. Only then the new node is included in the membership at all nodes, and receives every message. It also can safely start to send or broadcast messages yet. Causal stability allows new nodes to safely join the group without blocking the system or breaking causal consistency.

6.1.2 Causal Stability and Leave Requests

The logic is similar when an existing node decides to leave the group membership, with a few changes. When a node of the group decided to leave the group, it broadcasts to all membership a leave request. The leave request is also added to the local DAG at every node as normal message. Every node that delivers the leave request stops sending messages to the leaving node. The leave request is also added and delivered locally at the leaving node. The leaving node cannot yet leave safely. When the leave request is causally stable at the leaving node, the latter knows that all the other nodes have delivered the leave request and can safely exit the group membership. As for the other nodes, each of them can only delete

the meta data related to the to the leaving node and from their membership when the leave request is causally stable for them locally. The reason is that until the leave request is causally stable locally, a node can still receive messages concurrent to the leave request, and that require the meta data including the leaving node. Causal stability allows existing nodes to safely leave the group without blocking the system or breaking causal consistency.

6.2 Algorithm

In this section, we present the dynamic version of our Tagged Causal Delivery middleware in Algorithm 12 for the client process, and Algorithm 13 for the middleware process. We start by extending the latest algorithms presented previously in Algorithms 10 and 11.

6.2.1 System Startup

At the client process, we add a boolean called $sendstatus_i$. If $sendstatus_i$ is set to 1, the node can broadcast messages $tcbroadcast_i()$. Otherwise, it can only receive messages as it is temporarily the case for new joining nodes. In fact, the system can be bootstrapped with a number of nodes in the same group membership. Those nodes are initialized using $init_i(S)$ where ($S = active$). They can send and receive messages to every node in the group and have a middleware process that is responsible of managing causal meta data such as the causal DAG and ensuring Tagged Causal Delivery. As for new joining nodes, they are initialized with ($S = inactive$) and can only spawn a middleware process (line 42 in Algorithm 12) and start broadcasting messages (line 34 in Algorithm 12) when the join request is stable at the contacted node.

At the middleware process, J_i , a list of joining nodes is added to the state. When contacted by a new new joining node, the contacted node i stores the former's ID in J_i . It is used by contacted nodes to keep temporarily forwarding messages to joining nodes (lines 25, 54-55, 103-104 in Algorithm 13). Also, $init_i(B, G, V, J)$ takes 4 arguments. This is needed for when new joining nodes spawn a middleware process and need to initialize it with meta data transfered by the contacted node (line 19 in Algorithm 13 and 40, 42 in Algorithm 12).

ALGORITHM 12: Dynamic tcb client process for node $i \in \mathbb{I}$

```

1 state:
2    $d_i \in \mathcal{D}$ 
3    $ctxt_i \in \mathcal{C}$ 
4    $sendstatus_i \in \{0, 1\}$ 
5 proc  $init_i(S)$ 
6    $d_i := (i, 0)$ 
7    $ctxt_i := \emptyset$ 
8    $sendstatus_i = 0$ 
9   if  $(S = \text{active})$  then
10     $t = \{\{i \mapsto 1\}, \emptyset, \{j \mapsto 0 \mid j \in \mathbb{I}\}, []\}$ 
11     $\text{spawn}(\text{Middleware}, t)$   $sendstatus_i = 1$ 
12 proc  $tcbcast_i(msg)$ 
13   if  $sendstatus_i$  then
14      $\text{tcdeliver}_i(d_i, ctxt_i, msg)$ 
15      $\text{enqueue}_i^{cm}(msg, d_i, ctxt_i, msg)$ 
16      $ctxt_i := d_i$ 
17      $d_i := (i, d_i.1 + 1)$ 
18 proc  $\text{stabPhantom}_i()$ 
19    $\text{enqueue}_i^{cm}(\text{phantom}, d_i, ctxt_i)$ 
20 proc  $\text{leave}_i()$ 
21    $\text{enqueue}_i^{cm}(\text{leave}, d_i, ctxt_i)$ 
22    $ctxt_i := d_i$ 
23    $d_i := (i, d_i.1 + 1)$ 
24 on  $\text{dequeue}_i^{mc}(\text{deliver}, d, type, ctxt, msg)$ 
25    $ctxt_i := \{d\} \cup ctxt_i \setminus ctxt$ 
26    $\text{tcdeliver}_i(d, ctxt, msg)$ 
27 on  $\text{dequeue}_i^{mc}(\text{stable}, d, type, ctxt, msg)$ 
28    $\text{tcstable}_i(d, ctxt)$ 
29    $\text{enqueue}_i^{cm}(\text{stable}, d)$ 
30   if  $msg = i$  then
31     if  $type = \text{leave}$  then
32        $\text{exit}_i()$ 
33     else if  $type = \text{join}$  then
34        $sendstatus_i = 1$ 

```

```

(35) on  $\text{dequeue}_i^{mc}(\text{join}, uid)$ 
(36)    $Data := \text{getData}()$ 
(37)    $\text{enqueue}_i^{cm}(\text{join}, d_i, ctxt_i, uid, Data)$ 
(38)    $ctxt_i := d_i$ 
(39)    $d_i := (i, d_i.1 + 1)$ 
(40) on  $\text{receive}_i(\text{joinInit}, B, G, V, J, D)$ 
(41)    $\text{setData}(D)$ 
(42)    $\text{spawn}(\text{Middleware}, \{B, G, V, J\})$ 
(43) fn  $\text{tcdeliver}_i(d, ctxt, msg)$ 
(44)    $\text{/* delivery callback */}$ 
(44) fn  $\text{tcstable}_i(d, ctxt)$ 
(44)    $\text{/* stabilization callback */}$ 

```

ALGORITHM 13: Dynamic tcb middleware process for node $i \in \mathbb{I}$

```

(1) state:
(2)  $\text{bit}_i : \mathbb{I} \hookrightarrow \mathbb{N}$ , bit mask given node id
(3)  $G_i : \mathcal{D} \hookrightarrow \mathcal{M}$ , message graph
(4)  $V_i : \mathbb{I} \hookrightarrow \mathbb{N}$ , delivered version vector
(5)  $J_i : \mathbb{I}^*$ , list of ids

(6) on  $\text{init}_i(B, G, V, J)$ 
(7)  $\text{bit}_i := B$ 
(8)  $G_i := G$ 
(9)  $V_i := V$ 
(10)  $J_i := J$ 

(11) on  $\text{dequeue}_i^{cm}(\langle \text{msg}, d, P, m \rangle)$ 
(12)  $\text{local}_i(\langle \text{msg}, d, P, m \rangle)$ 

(13) on  $\text{dequeue}_i^{cm}(\langle \text{stable}, d \rangle)$ 
(14)  $\text{deletestable}_i(d)$ 

(15) on  $\text{dequeue}_i^{cm}(\langle \text{join}, d, P, j, \text{Data} \rangle)$ 
(16)  $\text{local}_i(\langle \text{join}, d, P, j \rangle)$ 
(17)  $VV = \text{generatevv}_i(d)$ 
(18)  $\{\text{past}, \text{conc}\} = \text{filter}_i(d)$ 
(19)  $\text{send}_{ij}(\text{joinInit}, \text{bit}_i, \text{past}, VV, [], \text{Data})$ 
(20) for  $k \in \text{dom}(\text{bit}_i), k \neq j$  do
(21)  $\text{send}_{ik}(\langle \text{join}, d, P, j \rangle)$ 
(22) for  $\{d' \text{ in } \text{conc} \mid G_i[d'].\text{stage} \neq \text{slt}\}$  do
(23)  $m = G_i[d'].\text{msg}$ 
(24)  $\text{send}_{ij}(\langle G_i[d'].\text{type}, d', G_i[d'].\text{pred}, m \rangle)$ 
(25)  $J_i := J_i + [j]$ 

(26) on  $\text{dequeue}_i^{cm}(\langle \text{leave}, d, P \rangle)$ 
(27)  $\text{local}_i(\langle \text{msg}, d, P, d.0 \rangle)$ 

(28) proc  $\text{local}_i(\langle \text{type}, d, P, m \rangle)$ 
(29) if  $\text{type} \neq \text{join}$  then
(30)  $\text{broadcast}_i(\langle \text{type}, d, P, m \rangle)$ 
(31)  $V_i[d.0] := d.1$ 
(32)  $P' = \{p \in P \mid \neg \text{stable}_i(p)\}$ 
(33) for  $p \text{ in } P'$  do
(34)  $G_i[p].\text{succ} := G_i[p].\text{succ} \cup \{d\}$ 
(35)  $G_i[d] := \{\text{stage} : \text{dlv}, \text{bits} : \sim 0, \text{type} : \text{type}$ 
(36)  $\quad \text{pred} : P', \text{succ} : \emptyset, \text{msg} : m\}$ 
(37)  $b = \sim \text{bit}_i(i)$ 
(38) if  $\text{type} = \text{join}$  then
(39)  $\text{updatemembership}_i(\text{type}, j)$ 
(40)  $b = \sim(\text{bit}_i(i) \mid \text{bit}_i(m))$ 
(41)  $\text{updatestability}_i(b, d)$ 

(42) on  $\text{receive}_i(\langle \text{type}, d, P, m \rangle)$ 
(43) if  $V_i[d.0] < d.1 \wedge \neg(G_i[d].\text{stage} = \text{rcv})$  then
(44)  $P' = \{p \in P \mid \neg \text{stable}_i(p)\}$ 
(45) for  $p \text{ in } P'$  do
(46) if  $p \notin \text{dom}(G_i)$  then
(47)  $G_i[p] := \{\text{stage} : \text{slt}, \text{succ} : \emptyset\}$ 
(48)  $G_i[p].\text{succ} := G_i[p].\text{succ} \cup \{d\}$ 
(49)  $b = \sum \{\text{bit}_i(p.0) \mid p \in P' \wedge G_i[p] \neq \text{dlv}\}$ 
(50)  $S = \text{if } d \in \text{dom}(G_i) \text{ then } G_i[d].\text{succ}$ 
(51) else  $\emptyset$ 
(52)  $G_i[d] := \{\text{stage} : \text{rcv}, \text{bits} : b, \text{type} : \text{type}$ 
(53)  $\quad \text{pred} : P', \text{succ} : S, \text{msg} : m\}$ 
(54) for  $j \in J_i$  do
(55)  $\text{send}_{ij}(\langle \text{type}, d, P, m \rangle)$ 
(56) if  $b = 0$  then
(57)  $\text{deliver}_i(d)$ 
(58)  $\text{checkstability}_i(d, P')$ 

(59) on  $\text{receive}_i(\langle \text{phantom}, d, P \rangle)$ 
(60)  $\text{checkstability}_i(d, P)$ 

(61) on  $\text{receive}_i(\langle \text{join}, j \rangle)$ 
(62) if  $j \notin \text{dom}(\text{bit}_i)$  then
(63)  $\text{enqueue}_i^{mc}(\langle \text{join}, j \rangle)$ 

(64) fn  $\text{stable}_i(d)$ 
(65)  $d.1 \leq V_i[d.0] \wedge (d \notin \text{dom}(G_i) \vee G_i[d].\text{stage} = \text{stb})$ 

(66) proc  $\text{deliver}_i(d)$ 
(67)  $\text{type} = G_i[d].\text{type}$ 
(68)  $\text{enqueue}_i^{mc}(\langle \text{deliver}, d, \text{type}, G_i[d].\text{msg} \rangle)$ 
(69)  $(j, n) = d$ 
(70)  $V_i[j] := n$ 
(71)  $G_i[d] := G_i[d] \{\text{stage} : \text{dlv},$ 
(72)  $\quad \text{bits} : \sim \text{bit}_i(i)\}$ 
(73)  $b = \sim \text{bit}_i(j)$ 
(74) if  $\text{type} = \text{join}$  then
(75)  $m = G_i[d].\text{msg}$ 
(76)  $\text{updatemembership}_i(\text{type}, m)$ 
(77)  $b = \sim(\text{bit}_i(j) \mid \text{bit}_i(m))$ 
(78)  $\text{updatestability}_i(b, d)$ 
(79) for  $s \text{ in } G_i[d].\text{succ}$  do
(80)  $G_i[s].\text{bits} := G_i[s].\text{bits} \& \sim \text{bit}_i(j)$ 
(81) if  $G_i[s].\text{bits} = 0$  then
(82)  $\text{deliver}_i(s)$ 

```

```

(83) proc checkstabilityi(d, P)
(84)   if d.1 = 1 + Vi[d.0] then
(85)     for p ∈ P do
(86)       | updatestabilityi(~biti(d.0), p)

(87) proc updatestabilityi(b, d)
(88)   if Gi[d].stage = dlv then
(89)     | b' = Gi[d].bits & b
(90)     | if b' ≠ Gi[d].bits then
(91)       | Gi[d].bits := b'
(92)       | if b' = 0 then
(93)         | stabilizei(d)
(94)       | else
(95)         | for p in Gi[d].pred do
(96)           | updatestabilityi(b, p)

(97) proc stabilizei(d)
(98)   for p in Gi[d].pred do
(99)     | if Gi[p].stage ≠ stb then
(100)      | stabilizei(p)
(101)   Gi[d].stage := stb
(102)   type = Gi[d].type
(103)   if type = join ∧ d.0 = i then
(104)     | Ji := tail(Ji)
(105)     | enqueueimc(⟨stable, d, type, Gi[d].msg⟩)

(106) proc deletestablei(d)
(107)   for s in Gi[d].succ do
(108)     | Gi[s].pred := Gi[s].pred \ {d}
(109)     | Gi := d ≪ Gi
(110)   if Gi[d].type = leave ∧ d.0 ≠ i then
(111)     | updatemembershipi(leave, d.0)

(112) proc updatemembershipi(type, j)
(113)   I = if type = join then
(114)     | sort(dom(biti) ∪ {j})
(115)     | else if type = leave then
(116)       | sort(dom(biti) \ {j})
(117)     | biti := {k → 2pos(k,I)} | k ∈ I}
(118)   for d ∈ Gi | Gi[d].stage ∈ {rcv, dlv} do
(119)     | f = if Gi[d].stage = rcv then 0 else 1
(120)     | Gi[d].bits := shifti(type, Gi[d].bits, 2pos(j,I), f)

(121) fn shifti(type, b, p, f)
(122)   if type = join then
(123)     | (((b ≫ p) ≪ 1) | f) ≪ (p) | (b & (2p - 1))
(124)   else if type = leave then
(125)     | (((b ≫ (p + 1)) ≪ p) | (b & (2p - 1)))

```

6.2.2 Joining Nodes

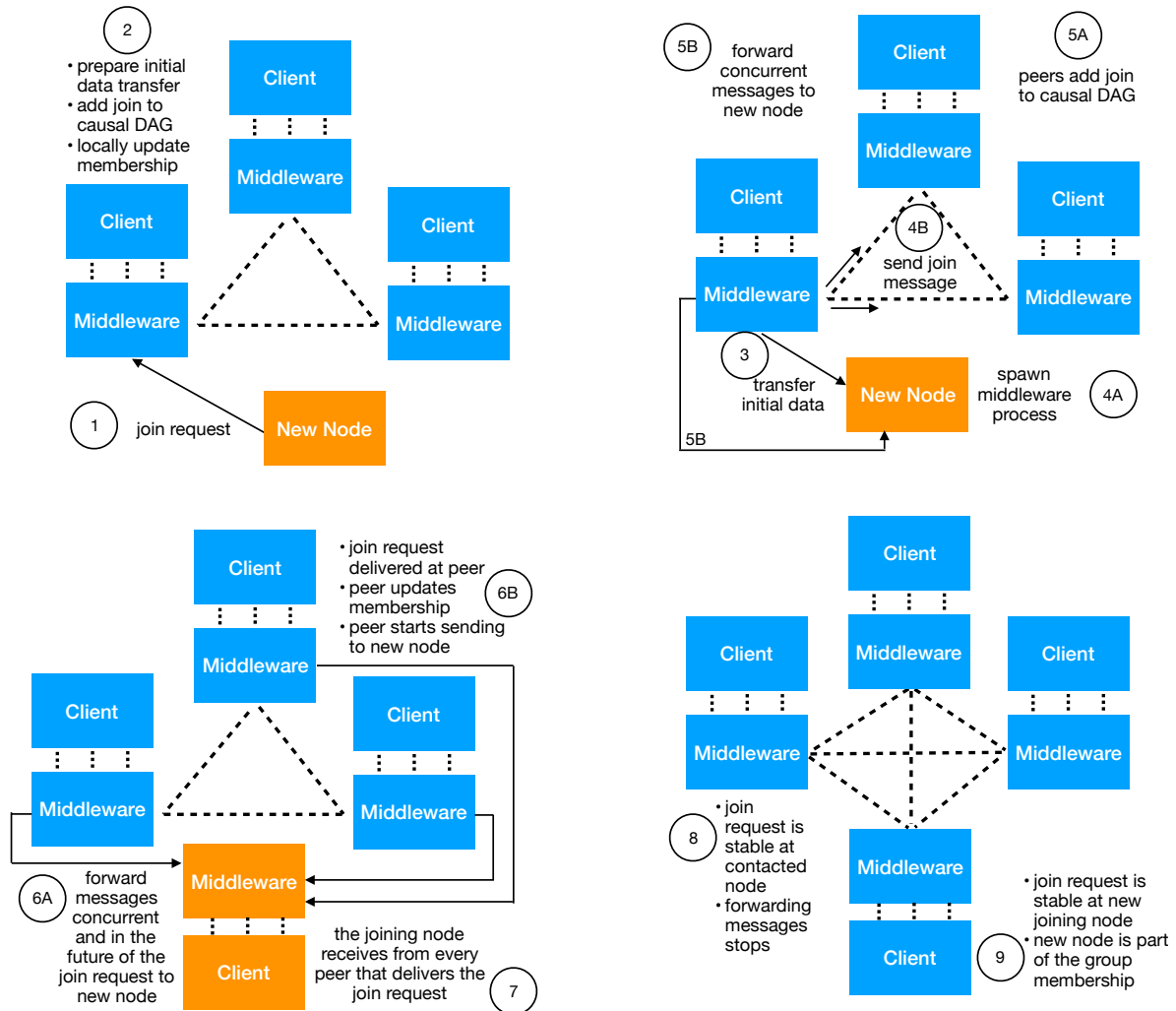


Figure 25: Evolution of a new node joining process

To join the group, a new node sends a join request and its unique ID to an existing node. The contacted node receives the join request (line 61 in Algorithm 13). It checks if the new node is part of the membership and if not enqueues the request to its client process (line 62, 63).

The client process (line 35) receiving the join request, prepares the application data/state to be transferred (line 36), enqueues a new request to the middleware process with a dot, a context and the application state (line 37), and finally updates its local dot and context (lines 38-39).

The middleware process (line 15) dequeue the request. It starts by adding the the join request to its local causal DAG (line 16). In $local_i$ the dot representing the join request is added locally as delivered at the contacted node (lines 28-41) as any other message. Notice, however, that for join requests, the contacted node updates its local membership to include the new node (lines 38-40). At line 18, the contacted node filters all causality meta data in the causal DAG in two parts: *past* that includes all dots

causally preceding the join request, and *conc* that includes the dots concurrent to it. Now the contacted node can transfer the initial state to joining node (line 19). It sends the join request to the membership (lines 20-21), and forwards the concurrent messages to the joining node (lines 22-24).

When the middleware process at other nodes (excluding the contacted node) deliver the join request, they also update their group membership to include the new node (lines 74-77). The new node now can start receiving messages from them too.

When the join request is causally stable at the contacted node i (line 103), the latter can remove the new node from J_i (line 104), and stops forwarding messages received from peers to the new node (as it used to do in lines 54-55). The initial state transferred to the new node was the same as the contacted node. Then all messages received by the contacted node were also forwarded to the joining node. This means that whenever a message is delivered or becomes stable at the contacted node, it also becomes stable at the joining node. When the join request becomes stable at the joining node (same time as at the contacted node), the middleware process informs the client process. The client process checks that the join request (line 33) is causally stable locally (line 30), and sets its *sendstatus_i* to 1 (line 34). Now the new node is part of the new group membership, and can safely send and receive messages to the group.

6.2.3 Leaving Nodes

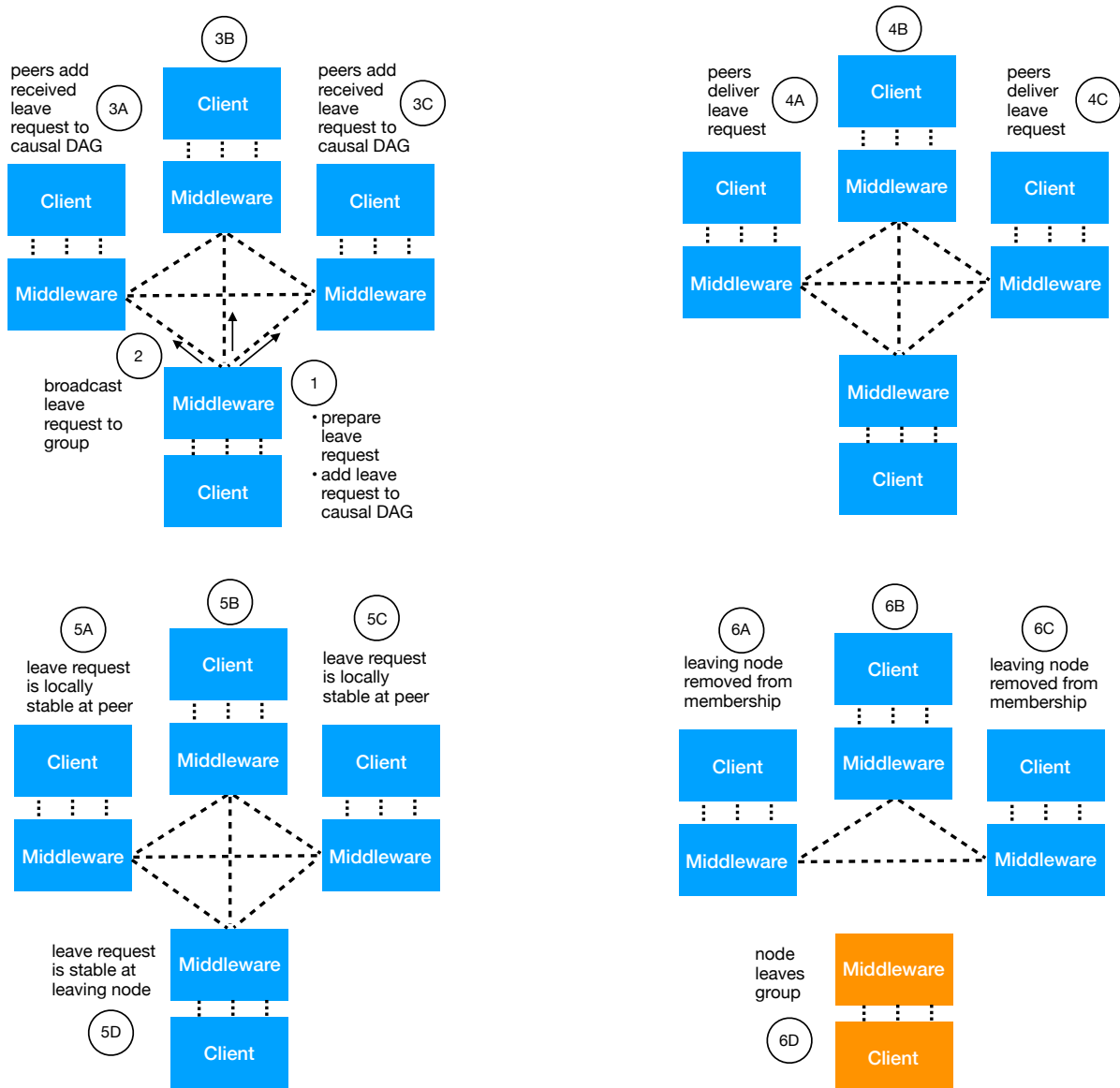


Figure 26: Evolution of an existing node leaving process

To leave the group, an existing node prepares a leave request, tagged with a dot and context and enqueues it to the middleware process (lines 20-23 in Algorithm 12). When the leave request is dequeued at the middleware process (lines 26-27), the latter creates a dot for it in its causal DAG as delivered (lines 28-41). The leave request is also broadcast to the group (lines 29-30).

At every node, the leave request is added to the local causal DAG as any other message, waiting to be delivered, stable and then removed. For each of these nodes, when the leave request is stabilized and deleted from the causal DAG, the middleware process updates its membership by removing meta data related to the leaving node (lines 110-111).

As for the leaving node, it can leave the group when the leave request is locally stable (lines 30-32 in Algorithm 12). The leaving node does not have to wait more than that to leave. It does not need to receive any messages from its peers or send any messages later than the leave request. The other peers do not expect to receive any messages from a leaving node. However, they cannot remove meta data related to the leaving node, as it could affect the causal stability computation related to messages concurrent to the leave request. This is the reason the other nodes have to wait for the leave request to become causally stable before the membership update.

6.2.4 Updating Group Membership

In this section we explain how updating the meta data such the causal DAG and the bitmask bit_i , to reflect the changes in group membership. We focus on the procedure $\text{updatemembership}_i()$ (lines 112-120) and the function $\text{shift}_i()$ (lines 121-125) in Algorithm 13.

$\text{updatemembership}_i()$ takes two arguments, *type* and *j*. *type* has for value join for updating the membership by adding a new node, or leave for removing an existing node from the groups membership. *j* is the globally unique identifier (GUID) assigned to the corresponding node. We use the IP address and port to generate a globally unique ID, but any mechanism that generates GUIDs can be used instead. Those IDs are also comparable. The bitmask bit_i , as explained earlier in previous chapters, is map from GUIDs to an integer (a power of two). The GUIDs are sorted in lexicographic order. The value associated to a GUID in the bitmask is 2^{pos} , where *pos* is the position of the GUID in lexicographic order. For instance, three IDs *A*, *B* and *C* will have the values $2^0 = 1$, $2^1 = 2$, and $2^2 = 4$ respectively. Notice that as the values are powers of two integers, they represent unique bits in a bitstring (001, 010, 100). This is very useful for representing causal delivery or causal stability for every dot. The bitstring is a very memory efficient representation to mark the set of nodes that have causally delivered or stabilized a dot. In our example 1 or its equivalent bitstring (001) represent the node *A*, 2 or 010 represent *B*, 6 or 110 represent *A* and *B*, and so on. Every dot in the causal DAG stores a bitstring, which is used to represent/compute causal delivery or causal stability. Therefore, when the group membership changes, the bitmask bit_i needs to be updated as well as every bitstring in the causal DAG. For adding new nodes, the GUIDs represented by $\text{dom}(\text{bit}_i)$ need to be sorted again including the GUID *j* of the joining node (lines 113-114). Similarly for removing nodes, $\text{dom}(\text{bit}_i)$ is sorted excluding the existing GUID *j* of the leaving node (lines 115-116). Then, every bitstring in the causal DAG is shifted accordingly (lines 118-120). $\text{shift}_i()$ is responsible of updating every bitstring by shifting the bits and adding a new bit for the joining node without affecting the values of the existing bits (lines 122-123), or to remove the bit corresponding to the leaving node (lines 124-125).

Causality Checker

Due to the complexity of the algorithms so far, we developed our own verification tool that we called Causality Checker. This Causality Checker is a standalone service that verifies that the different versions of our Tagged Causal Delivery Algorithm, ensures causal delivery and causal stability. The Causality Checker also gives insight to the developer to what might be the problem when the verification fails by printing the failed step and the state of the corresponding nodes. It makes it easier to for developers and implementors to add changes and updates to their algorithms, test them, debug them and finally verify their correctness.

The tool works by reading the serialized events logged by the middleware in test mode. The serialized events are a sequence of *Send* (broadcast), *Deliver* and *Stable* events from every peer in the group membership. The order of events in a sequence follow the same order they were logged in by the peers in runtime. Every sequence corresponds for the serialized events at one peer. Every entry in a sequence is a dot and its meta data (d, val). The meta data val stores the type of event ($d.stage$) which takes as value *SEND*, *DLV* or *STB* for send, deliver and stable events respectively.

The causality checker verifies if these dot sequences respect the following rules:

- A *Send* with a dot only appears in the sender's sequence
- A *Deliver* with a dot appears in every peer sequence except the sender's
- A *Stable* with a dot appears in both the sender and the receiving peer sequences

The logged sequences are passed to the causality checker which implements a backtracking algorithm designed to verify causal order by looping through them. The causal history of each dot is built while iterating over these sequences. Version vectors are used to represent causality. Each dot is associated with a version vector that represents its causal dependencies. A version vector is associated for every peer to represent causal delivery. While looping through the sequences, the peer version vectors are updated and the dot version vectors are computed accordingly. A causal delivery check is a comparison between the version vectors of a peer and the dot being delivered in the sequence. A causal stability check is a comparison between the version vector associated with the dot being stabilized and the peer's causal

stability matrix where the dot is being stabilized. While looping through the sequence of peer i in order, checking the correctness of an event e_i at peer i might require branching to another sequence j and verify other events e_j , then backtrack to e_i at i and so on.

7.1 Causal Check Algorithm

We explain in this section the causal check algorithm presented in **Algorithm 14**. We start by explaining the causal data structures used in it global **state**:

$G : \mathcal{D} \hookrightarrow \mathcal{M}$, represents the causal DAG $PVV : \mathbb{I} \hookrightarrow V$ where $V : \mathbb{I} \hookrightarrow \mathbb{N}$, maps peers to their version vectors $DDV : \mathcal{D} \hookrightarrow V$, maps a dot to its equivalent version vector $PMAT : \mathbb{I} \hookrightarrow M_x$ where $M_x : \mathbb{I} \hookrightarrow V$, maps peers to stability matrices $IND : \mathbb{I} \hookrightarrow \mathbb{N}$, maps peers to a sequence index

- G : a causal DAG that stores every dot d and its corresponding val . The dot is stored in the causal DAG when its *Send* event is handled by the algorithm.
- PVV : a map associating a peer id to its corresponding local version vector. The version vector at each peer represent the causal delivery meta data at that peer.
- DDV : a map associating a dot to its equivalent version vector. This version vector is computed when when the *Send* event of that dot is handled by the algorithm.
- $PMAT$: a map associating a peer id to its corresponding causal stability matrix. The matrix is used, at each peer, to verify the correctness of a *Stable* event of a dot at that peer.
- IND : a vector of counters marking, for each peer, the last position in its dot sequence handled by the causality checker;

As we explain in details in the following subsections, to verify causal delivery and stability while iterating over the peer sequences recursively, the following rules are set:

- When a *Send* event happens at a peer i , the dot associated with event must have it's ID ($d.0$), which represents the peer that created to send event in the runtime, equal to i .
- A dot associated with a *Send* event is delivered immediately at the sender when the send event is handled by the algorithm.
- A dot's version vector is only calculated when it appears in the sender's sequence through a *Send*.
- The same dot cannot appear more than once in the sender's sequence.
- In a *Deliver* event, the peer's and dot's version vectors are compared. If the causal delivery check passes, the peer's version vector is updated.

ALGORITHM 14: Causal check algorithm

```

(1) state:
(2)  $G : \mathcal{D} \hookrightarrow \mathcal{M}$ , represents the causal DAG
(3)  $PVV : \mathbb{I} \hookrightarrow V$  where  $V : \mathbb{I} \hookrightarrow \mathbb{N}$ , maps peers to their
    version vectors
(4)  $DVV : \mathcal{D} \hookrightarrow V$ , maps a dot to its equivalent version
    vector
(5)  $PMAT : \mathbb{I} \hookrightarrow Mx$  where  $Mx : \mathbb{I} \hookrightarrow V$ , maps peers to
    stability matrices
(6)  $IND : \mathbb{I} \hookrightarrow \mathbb{N}$ , maps peers to a sequence index

(7) proc init()
(8)  $G := \emptyset$ 
(9)  $vv := \{i \mapsto 0 \mid i \in \mathbb{I}\}$ 
(10)  $PVV := \{i \mapsto vv \mid i \in \mathbb{I}\}$ 
(11)  $DVV := \emptyset$ 
(12)  $PMAT := \{i \mapsto PVV \mid i \in \mathbb{I}\}$ 
(13)  $IND := vv$ 

(14) fn causalcheck(seq)
(15) for  $i = 0$  to  $\text{len}(seq)$  do
(16)    $seq_i := seq[i]$ 
(17)   for  $j = IND[i]$  to  $\text{len}(seq_i)$  do
(18)      $(d, val) := seq_i[j]$ 
(19)     switch  $val.stage$  do
(20)       case SEND do
(21)         if  $d.0 = i$  then
(22)           handleenderdot( $d, val$ )
(23)         else
(24)           return error
(25)       case DLV do
(26)         if  $d \notin G$  then
(27)           handlepeerdot( $d, seq$ )
(28)           handledelivered( $d, i$ )
(29)         case STB do
(30)           handlestable( $d, PMAT[i]$ )
(31)            $IND[i] := IND[i] + 1$ 

(32) fn handlestable( $d, M$ )
(33)    $vv_d := DVV[d]$ 
(34)   for  $vv$  in  $M$  do
(35)     if  $vv_d < vv$  then
(36)       return error

(37) fn handleenderdot( $d, val$ )
(38)   if  $d \notin G$  then
(39)      $PVV[d.0][d.0] := PVV[d.0][d.0] + 1$ 
(40)      $G[d] := val$ 
(41)      $DVV[d] := PVV[d.0]$ 
(42)      $PMAT[d.0][d.0] := PVV[d.0]$ 
(43)   else
(44)     return error

(45) fn handlepeerdot( $dot, seq$ )
(46)    $seq_i := seq[dot.0]$ 
(47)   for  $j = IND[dot.0]$  to  $\text{len}(seq_i)$  do
(48)      $(d, val) := seq_i[j]$ 
(49)     switch  $val.stage$  do
(50)       case SEND do
(51)         if  $d.0 = dot.0$  then
(52)           handleenderdot( $d$ )
(53)           if  $d = dot$  then
(54)              $IND[dot.0] := IND[dot.0] + 1$ 
(55)             return
(56)           else
(57)             return error
(58)       case DLV do
(59)         if  $d \notin G$  then
(60)           handlepeerdot( $d, seq$ )
(61)           handledelivered( $d, dot.0$ )
(62)       case STB do
(63)         handlestable( $d, PMAT[dot.0]$ )
(64)          $IND[dot.0] := IND[dot.0] + 1$ 

(65) fn handledelivered( $d, i$ )
(66)    $vv_d := DVV[d]$ 
(67)    $vv_p := PVV[i]$ 
(68)   if  $vv_p[d.0] = vv_d[d.0] + 1 \wedge vv_p[j] \geq vv_d[j] \forall j \neq$ 
      $d.0, j \in \mathbb{I}$  then
(69)      $PVV[i][d.0] := PVV[i][d.0] + 1$ 
(70)      $PMAT[i][d.0] := vv_d$ 
(71)      $PMAT[i][i] := PVV[i]$ 
(72)   else
(73)     return error

```

- For every *Send* and *Deliver* events, the peer's version matrix is updated.
- When a *Stable* event is handled, the peer's version matrix rows are compared with the dot's version vector to check causal stability.
- If any of the checks fail, an error is returned with separate logs for each peer containing the meta data and the even at which the causal check. The causal check succeeds when the algorithm goes through all the sequences without failing.

7.1.1 causalcheck()

The `causalcheck` function takes as argument `seq` which stores the local sequence of events seq_i for every peer i . It starts by iterating through every peer's sequence (lines 15-16) and handling the events incrementally (line 17). The $IND(i)$ stores an index to the last handled event for the sequence ($seq[i]$) corresponding to the peer i . Every dot and its meta data, containing the event type ($val.stage$), are retrieved from the sequence (line 18). For *Send* events (lines 20-24), the function checks if the dot's ID ($d.0$) is equal to the sender's id (i) that is trying to handle the send event (line 21). If the check passes, `handlesenderdot()` is called to handle the send event and update the corresponding data structures for dot d and peer i (line 22). After that $IND[i]$ is incremented (line 31) and the loop moves to the next entry in $seq[i]$. If the check fails, however, the algorithm return an error (line 24). For *Deliver* events (lines 25-28), the function checks if the dot to be delivered at i is already in the global causal DAG G (line 26). G stores all dots form all peers. A dot is stored in G at the *Send* event associated with it. When a dot d is not in G , and a *Deliver* event is associated with it at some peer i , this means that the *Send* event associated with it have not been handled yet by the algorithm. The algorithm then looks for the dot's sender ID ($d.0$), and calls `handlepeerdot()` (line 27). The function `handlepeerdot()` goes through the sequence associated with that dot, $seq[d.0]$. Until the *Send* event of dot d is handled. Then the algorithm backtracks to the first sequence and handles the *Deliver* event of $d.0$ at i . A branching can lead to more branchings by calling `handlepeerdot()` for other peers, and then backtracks to the first call. If the d was in G , then the algorithm handles the *Deliver* event of d by calling `handledelivered()` (line 28). For *Stable* events (lines 29-30), no previous checks are needed and `handlestable()` is called to check causal stability for the dot d at i (line 30). At the end of each iteration in $seq[i]$, the corresponding index $IND[i]$ is incremented.

7.1.2 handlesenderdot()

This function takes as arguments a dot d and its meta data val . It is responsible of handling *Send* events. A dot d is added to the causal DAG G for the first time when this function is called. The function checks if the dot to be handled is in G (line 38). If it was an error is returned and the test fails (line 44). If it was not in G , it adds it with the corresponding val (line 40). The function in this case also updates the sender's

peer version vector $PVV[d.0]$ (line 39). It increments by 1 the entry corresponding to the sender in that version vector, $PVV[d.0][d.0]$. The dot d becomes associated with the incremented version vector $PVV[d.0]$ and stored in $DVV[d]$. This version vector can be used later for checking causal delivery of this dot at other peers. Finally, The peer's entry ($PMAT[d.0][d.0]$) in the in the causal stability Matrix associated to that peer ($PMAT[d.0]$) is updated (line 42).

7.1.3 handledelivered()

This function takes as arguments a dot d to be delivered and i , the peer where it is being delivered at. It is responsible of handling *Deliver* events by checks if the dot d can be causally delivered at i . vv_d stores the version vector associated with the dot d in $DVV[d]$. vv_p stores the local version vector associated with peer i in $PVV[i]$. The two version vectors vv_d and vv_p will be used to check if d can be delivered at i (line 68). The condition requires that 1) every entry in $vv_p[j]$ to be more recent or as recent as its corresponding entry in $vv_d[j]$ and 2) that for the entry $d.0$, $vv_d[d.0]$ has delivered one more event than $vv_p[d.0]$. If this condition fails, then the dot d can not be delivered at i and the causal check fails returning error (line 73). Otherwise, the peer i 's version vector $PVV[i][d.0]$ is incremented by one at the entry $d.0$ to mark the delivery (line 69). Finally the causal stability matrix corresponding to peer i , $PMAT[i]$ is updated. $PMAT[i][d.0]$, the vector (row) in $PMAT[i]$ corresponding to the dot sender $d.0$ takes the value vv_d (line 70). $PMAT[i][i]$, the vector (row) in $PMAT[i]$ corresponding to the peer delivering the dot d takes the updated value of local version vector $PVV[i]$ (line 71).

7.1.4 handlestable()

This function takes as arguments a dot d to be stabilized and the causal stability matrix M used to check causal stability. It is responsible of handling *Stable* events. vv_d stores the version vector associated with the dot d in $DVV[d]$ (line 33). Then for every row vv in M , the version vector vv_d corresponding to the dot d must be less recent than vv (line 34-35). Which means that every peer has delivered at least one event more recent to the event correspond to the dot d . If the test fails for one vv , the function returns an error.

7.1.5 handlepeerdot()

This function takes as arguments a *dot* and the sequences for all peers seq . This function handles branching to a different sequence when the causal checker tries to handle a *Deliver* event for a *dot* that is not in the causal DAG G yet. This requires the algorithm to jump or branch to the *dot* sender's sequence $seq[dot.0]$ and handle the *Send* event corresponding to it. Only then, it can backtrack to the previous sequence (line 53-55) and handle the *Deliver* event there. These branches/jumps to the dot's sender can occur multiple times (lines 58-60). But if N is the number of peers then the maximum number of jumps is $N - 1$. The function mainly has the same functionality of function `causalcheck()`, except that it

does it for one sequence per branching, and until it handles the *Send* event of the *dot* from the previous sequence (line 53-55).

Evaluation

This chapter focuses on the evaluation of the graph based and version vector middleware implementations. The purpose of the experiments is to check how the graph implementation compares against the version vector approach, namely in scaling with the number of nodes and with the number of pending messages to be delivered, for different degrees of concurrency, given by the number of direct predecessors of a message. The configuration of the experiments is detailed in Section 8.1.

Most of the experiments were executed in a real distributed system by deploying the middlewares with Docker and Kubernetes. This deployment is further explained in Section 8.2. Some relevant metrics for comparing the middleware and evaluating the experiments results will be introduced on Section 8.3. Finally, the experiments will be presented in Section 8.4. Here, each experiment details its purpose and the configuration parameters that were used.

8.1 Configuring the experiments

Regarding the execution of experiments, the objective was to simulate different broadcast scenarios. In an ideal situation, a peer message would arrive instantaneously to its destination. But in a real group communication, there's a latency that does not allow this to happen. Therefore, some parameters were specifically added to the configuration file to be used during the evaluation.

8.1.1 Send interval

A peer's send interval can either be a fixed value or a sample from a Poisson distribution. After calculating the time until the next send, the peer will use this interval as a timeout for delivering messages. During this time the peer delivers messages (if there are any) and when it ends a send occurs.

With a fixed interval, the peers send messages at the same rate, but this time can also be calculated using a Poisson distribution, which is a model for a series of discrete events where the average time between events is known, but the exact timing of events is random [81]. In other words, knowing the average interval of sends we can calculate the time when the next message should be sent.

To use a Poisson distribution, an average time between events, which in this case are broadcasts, is needed. According to [82], to determine these intervals, a random value U between 0 and 1 from a uniform distribution has to be generated. This number is a value from the y-axis and to locate the corresponding time value on the x-axis, the inverse of the exponential function is used:

$$T = \frac{-\ln U}{\lambda}$$

The λ is the average rate of events, and because the average time interval is passed through the configuration file, it has to be transformed into the equivalent rate. Since the period is the reciprocal of the frequency, λ can be calculated for an average interval A by using $\lambda = \frac{1}{A}$.

Using a Poisson distribution can sometimes return samples much higher than the desired average. To avoid these, the calculated value from this distribution is truncated at a maximum of 4 times the average from the configuration file.

8.1.2 Network latency

The reason for simulating network latency is to add more control over the experiments executed when simulating the delay a message has when travelling over a network. The network latency can either be a fixed value or a fixed value plus another component calculated through a Weibull distribution.

The network latency is simulated by adding to the sender's clock a value in microseconds calculated through the network latency parameters.

If the configuration file only has the baseline, then the network latency is a fixed value for all the links in the group. However, to specify a different fixed latency between the peers, a path to a matrix can be set through the configuration file. This is an $N \times N$ matrix, where N is the group size, the row is the sender and the column is the receiver. For example, the latency of the link from peer 3 to peer 2 is on row 3 and column 2 of this matrix. In this file the matrix is represented as N rows, each with N integers separated by commas. At least one of these previous parameters is mandatory.

To use a baseline value plus another from a Weibull distribution, both *scale* and *shape* parameters have to be specified. A Weibull distribution is a continuous probability distribution and has two parameters: scale and shape. This is also used as a model to characterise end-to-end network delay measurements [83]. This baseline, as the previous one, is passed through the same fields.

The network latency calculation using the Weibull distribution was based from [83–85]. To achieve three standard deviations from the baseline, using *shape* = 2 and a *scale* = 0.15 had to be used. The mean of a Weibull distribution [86] can be calculated by using the gamma function Γ , as $\lambda \times \Gamma(1 + \frac{1}{k})$, where λ and k are, respectively, the scale and shape parameters. The latency is calculated with this distribution as follows:

$$A = B \times (1 + W(\lambda, k))$$

A and B are the calculated latency and baseline value, whereas $W(\lambda, k)$ is a sample from the Weibull distribution. Using the previous expression to calculate the mean of a Weibull distribution, for a scale $\lambda = 0.15$ and shape $k = 2$, this is ≈ 0.1329 . By using the preceding expression we have:

$$A = 1.1329 \times B$$

With this we can calculate the needed baseline value for a given average latency. For example, if we wanted an average network latency A of 5 seconds, the baseline value B to use together with the Weibull distribution to achieve it would be 4.4135. However, sometimes the calculated delay might be very large which in turn could affect the results. Therefore, the calculated value is truncated at a maximum given by $B \times (1 + 3 \times \lambda)$.

8.1.3 Simulating slow links

Regarding the network latency between peers, the experiments can be divided into symmetric and asymmetric network topologies. In the first, every link between the peers has the same latency. For the asymmetric scenario there's a link with a higher network latency that further delays the arrival of messages. For the experiments the slow link is always between peer 0 and 1. This slow link scenario aims to test the middleware's robustness to transient network partitions. However, for the experiments a fixed value is used for the delay in the slow link, as to avoid having large fluctuations in the calculated latency and have more control over the buffered messages by the peers between runs.

Network partitions occur when groups of nodes have difficulties communicating with other nodes outside of their group. This happens either because of higher latencies or unavailable peers. These partitions affect information availability because messages take longer to arrive at their destination, and other messages that depend on them cannot be delivered and have to be buffered until then. This causes a buildup of received messages waiting to be delivered at the middleware. Imagine a group with 5 peers on a full mesh topology, where everyone has a network latency of 1 second between them, except the link between 0 and 1 that has a latency of 20 seconds as show on Figure 27.

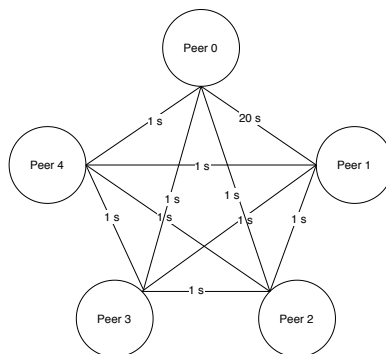


Figure 27: Full mesh topology with a slow link between 0 and 1.

Messages are broadcast from the peers that have as dependencies messages from other peers. Since they are part of the slow link, messages between 0 and 1 take a longer time to arrive at their destination. At peer 0 or 1, messages with a dependency from 1 or 0 have to wait because it takes longer for that dependency to arrive and be delivered.

8.2 Deploying environment

This section details the on deployment of the experiments. First, it explains what type of architecture was used for the deployment in order to simulate a network with multiple peers interacting with each other. After this, an introduction to Docker, some of its advantages and how it fits on the experiments will be given. Then the last part of deploying the experiments with Kubernetes will also be explained.

8.2.1 Architecture

For the deployment architecture, an Erlang program was written so that the middleware is used for broadcasting messages. Through configurations parameters this program spawns multiples nodes and each of these is either a master or a worker. Each deployment only has one master node but there can be multiple workers. A worker node locally spawns peers that broadcast messages with peers spawned by other worker nodes (depending on the configuration).

The master is the first node to be deployed by Kubernetes and after it the workers start. After it finishes setting up, it waits for all the workers to connect to it. The purpose of the master is to receive information from the workers, namely port numbers and IP addresses of the peers they spawn and, after the run has ended, their logged information.

Upon receiving all the necessary information to establish connections between the peers in the nodes, the master processes these addresses and sends them over to the workers, so they can have a view of the network layout. Furthermore, the master also assigns a globally unique id to each worker node and to the peers they spawn. With this information the peers spawned by the workers can start their middlewares, connect to each other and begin sending and delivering messages.

In this scenario, the worker nodes run a local logging service where their peers log the events to. The difference from before is that when they finish broadcasting, the causal check cannot be executed locally, since only a subset of all logs is available.

To overcome this, when the peers stop executing, their worker node sends the logged information to the master. Only after this will the master have a complete view of the logging events at each peer and the post-processor is then executed, checking also the delivery and stability order of messages. From this results the JSON file with the information collected during the experiments from the peers at every node.

8.2.2 Docker

Docker [87] is a tool that allows packaging applications in *containers*, including their dependencies, by using OS-level virtualization. These containers run applications isolated from the host system they're on and are designed to make it easy and fast to provide a consistent experience as software is replicated and moved between environments. Containers are similar to virtual machines, but instead of creating a complete virtual operating system, they only replicate the components configured to them which leads to an important boost on performance. A Docker image can be shared by different operating systems and the software inside of it can also be executed, as long as the host has Docker installed.

For the deployment of the experiences, a Docker image was created from a container that has Erlang installed and the software for spawning the peers. Since images can easily be shared, the master and worker nodes both result from them. The master and worker nodes execute this image and because an Erlang environment was configured in the container, there is no need for them to have it installed.

8.2.3 Kubernetes

The other component for deploying the experiments is Kubernetes [88], which is a tool for automating software deployment. Kubernetes provides a framework for running code on clusters and also manage its execution. From running a Docker image on Kubernetes, a *pod* is created.

Kubernetes was used to deploy the broadcast experiment image on a cluster hosted on Emulab [89] and the pods will be the master and worker nodes.

The Emulab nodes where Kubernetes deploys the images runs on different machines and these nodes have a delay between them of around 1 millisecond. Because the clocks are not fully synchronized, the latencies used during the experiments were high enough so that a 1 millisecond difference would not affect the results.

For every experiment the same type of node with the same hardware was used. The deployments ran on machines with 8 CPU cores. Because there were limitations on the number of available nodes, each pod was limited to only using 4 cores. Furthermore, each pod spawned a single local *peer* and by using this configuration, by having 64 Emulab nodes it was possible to run experiments with groups of up to 128 peers.

8.3 Comparing experiments

Non-causal latency is used to compare the results of the experiments. Any causal delivery mechanism has an algorithm induced message delay. For example, this happens when a message is received by a process before all its predecessors have also been delivered. So that the algorithm can maintain causal delivery order of messages, this delay is necessary and is known as causal latency [90]. Furthermore,

this latency is not the algorithm's fault because outside factors can delay the receiving of a message, e.g., network latency.

Some algorithms will delay delivery without knowledge that there is a causally prior message to deliver that has not yet been received. Rather the algorithm delays delivery based on lack of knowledge that such message does not exist. That is, the delay on the delivery of messages even though all their causal predecessors have already been delivered. This is known as non-causal latency [90]. The optimum causal delivery algorithm would have no non-causal latency.

During the experiments, when a received message can be immediately delivered it might trigger other deliveries. The time between a receive and the triggered delivery of a message is the non-causal latency of this message. Therefore, for both middleware implementations, when this occurs, the elapsed time between the receive of the last causal predecessor and the delivery of its successors is measured and sent to the log service. The lower this non-causal latency is the lower the delay induced in delivering messages due to the algorithm.

The time for the messages to become causally stable is another metric to compare both approaches. Like for non-causal latency, the time, after the receive that made it possible, that it takes for a message to be deemed stable is another useful measurement. This stability time for the messages at every peer is also sent to the logging service.

8.3.1 Memory Metadata

The memory allocated during the experiments is calculated by counting the words required for the algorithm being used by the middleware (either graph based and version vectors). A word is a fixed-sized piece of data and also serves as a unit. A word is 8 Bytes for 64-bit Erlang, and this is what we used for measuring the memory metadata. Since both middleware implementations require different data structures, the method for counting their allocated words differ. The purpose of this metric is to have an idea and compare the memory required by the two algorithms. Further below are the methods used for counting the words in each middleware version. It is noteworthy to mention that for this metric the size of the message's payload was ignored and only considered the necessary words for tracking causality.

8.3.1.1 Graph based

For the graph based implementation each message is tagged with a pair (*id*, *ctr*) called *dot*, where *id* is the sender's globally unique identifier and *ctr* is a monotonically increasing counter. Besides this, each message also has a list *context* of dots that represent its immediate causal predecessors. For the graph based algorithm, it was considered that the *id* and *ctr* each represent one word. Therefore, each dot is two words.

In this approach, a directed acyclic graph is used for tracking the causal dependency between messages. Each message is a node in this graph, all outgoing edges of this node are successors of that message and every incoming edges are predecessors. In Erlang, we used a map to represent the graph,

where each entry is a dot, mapped to a data structure representing the causality information. Namely, a set of dots for the predecessors, a set of dots for the successors, a bitstring for causal stability and an enum for the stage to mark if it is received, delivered, etc.

The number of words in this middleware implementations is the result of the following sum for every node in the graph:

- 2 words for the dot
- 2 words for each predecessor
- 2 words for each successor
- 1 word for the stage
- 1 word for the bitstring

8.3.1.2 Version vectors

In the version vector implementation of the middleware, each message has a vector with same length as the number of peers in the network. Each position is considered to be one word. Therefore, a version vector of length N requires N words. Just like in the previous graph based approach, the size for the message's payload is ignored.

Each middleware has two version vectors to track, respectively, the received and delivered messages: R and V . Therefore, if N is the number of peers in the network, these version vectors require $2N$ words in total. Furthermore, there is the matrix that tracks causal stability. In this matrix, row $_i$ is the version vector of the last message sent by peer $_i$. In this case, the number of words for this matrix is N^2 .

This implementation approach also has a queue DQ for received messages that could not be delivered because of causal dependencies. Each element in this queue is a message that has a payload, version vector and the sender's globally unique identifier id . Since the size of the payload is ignored, only the version vector and id are taken into account. As already mentioned, a version vector requires N words, considering a system of N peers. On the other hand, the id is only an integer. Therefore, it is considered that this id is one word.

The last data structure in this implementation were counting words is required to measure the size is the stability map $SMap$. This object maps a pair of ids (the sender's and message's) to a stable message. The pair of ids used here is similar to the dot from the graph based approach. Therefore, it is considered that it requires two words. The stable message has the message's payload and version vector. Since the payload is ignored, only the N words from the version vector are considered. For the $SMap$ with K elements and N peers, the word number is $K(2 + N)$.

In summary, for the version vector middleware implementation, the number of words, for a group with N peers and K stable messages, is the sum of the following calculations:

- N words from the received messages version vector
- N words from the delivered messages version vector
- N^2 words from the stability version matrix
- $K(2 + N)$ words from the stable messages map

8.4 Broadcast Experiments

This section focuses on comparing results of the experiments using a graph based and version vector approach to deliver messages in a causal order and determine causal stability. Between runs either the number of peers in the group or send interval of peers are updated in order to further demonstrate the behaviour of the middleware implementation. Each subsection details the configurations used while running the experiment.

8.4.1 Experiment 1: Classical end-to-end vector-based VV vs our end-to-end graph-based TCB

For this experiment the peers are in a full mesh topology where all the links have the same network latency. The purpose of this is to see how the middleware implementations behave when we modify send intervals and gradually increase the group size. The effect on performance due to calculating causal stability is also measured by doing runs with and without having to calculate it.

The following experiment was done with the same send interval for every peer and the interval between broadcasts is calculated following a Poisson distribution, and each peer sends 100 messages. The send intervals chosen for the peers in this experiment were, in milliseconds: 10, 100 and 1000. Moreover, the network latency is calculated via the Weibull distribution, as to make the peers more desynchronized from each other. Runs were executed for groups with 4, 8, 16, 32, 64 and 128 peers. The network latency chosen for the links in the runs of this experiments was 10 milliseconds.

In this experiment we compare a version vector based approach where the application uses an additional version vector to be able to provide an end-to-end happens-before against our E2E TCB approach. We call it classical E2E VV. In this version, we use a classical VV-based causal delivery algorithm using version vectors and a delivery queue for causal delivery. Plus, the application uses an additional version vector to provide an end-to-end happens-before.; This is what we called previously a duplication of effort. Our aim in this experiment is to show the overhead in terms of transmission on the network and the amount of causality metadata stored in memory between both approaches.

In Figure 28a, we can clearly see that the number of causal predecessors considered as dependencies in the graph approach is smaller than the size of version vectors needed in the classical E2E VV-based approach. At 128 nodes, the number of predecessors needed is at worst 80 for the E2E TCB approach

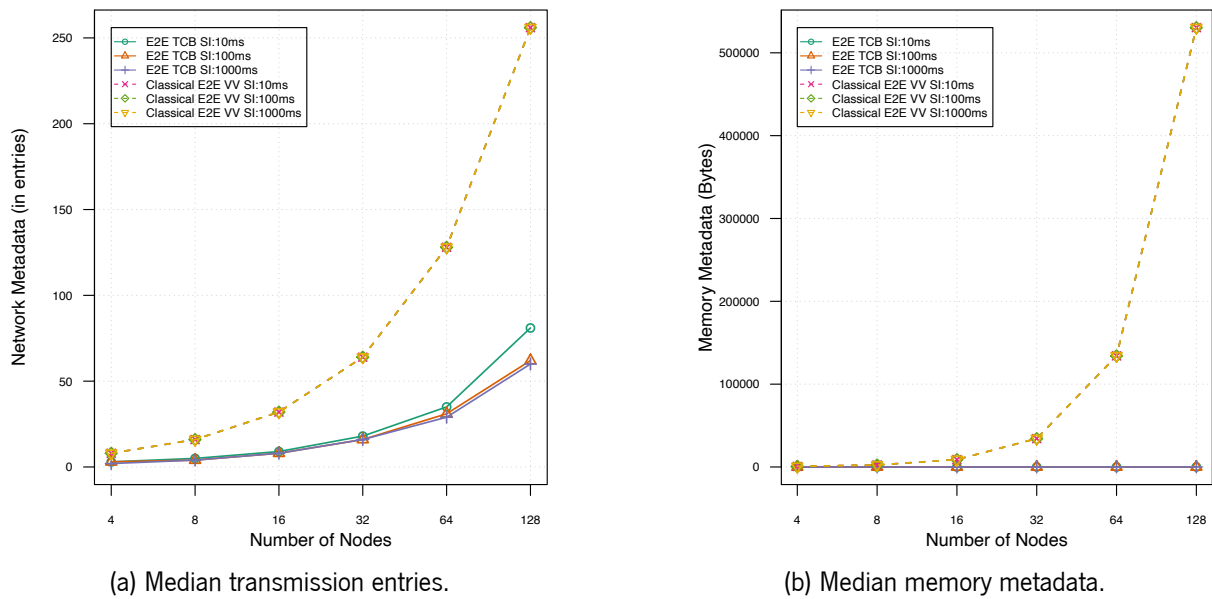


Figure 28: Experiment 1: transmission and memory, without stability.

to provide end-to-end happens-before. However, in the classical E2E VV-based approach, we need twice the size of nodes which is 256 entries.

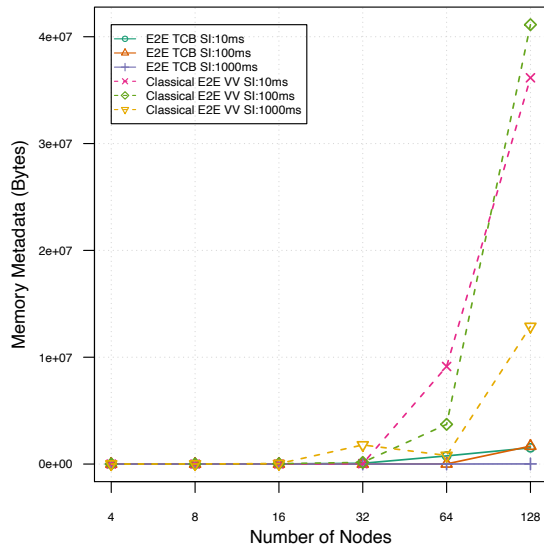
The size of those vectors also affects the size of metadata needed to be stored locally to ensure causal delivery (and stability as we show later). In Figure 28b, we only show the causality metadata stored in memory when stability is not being calculated. Meaning, less data structures are used to store causality metadata. Those data structures will use the dependencies in case of the E2E TCB approach, and the version vectors for the classical E2E VV approach. Therefore, the size of metadata stored in memory will be affected as we see in Figure 28b. In the case of the vector-based approach the data stored can get to more than 500KB for 128 nodes, where it is negligible for the E2E TCB approach.

When stability is calculated, we notice in Figure 29a that the memory size grows a lot for the classical E2E VV approach and reaches more than 40MB for 128 peers. This is due to the larger size of version vectors for this experiment, the additional data structures used for stability such as the stability matrix and the stable map. The size of metadata in memory for the E2E TCB approach also increases compared to the experiment without stability in Figure 28b, but remains significantly smaller than the classical E2E VV approach.

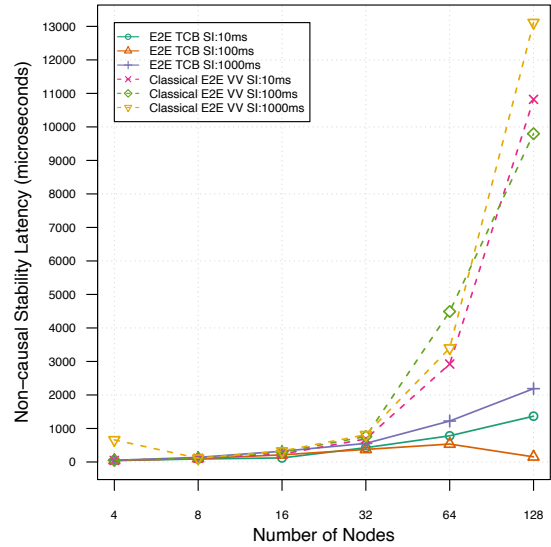
In Figure 29b we can see that the E2E TCB approach takes significantly less time (2-3 milliseconds for 128 peers) for non causal stability compared to the classical E2E VV approach (above 10 milliseconds for 128 peers).

8.4.2 Experiment2: Classical vector-based VV vs graph-based

For this experiment the peers are in a full mesh topology where all the links have the same network latency. The purpose of this is to see how the middleware implementations behave when we modify send interval



(a) Median memory metadata.



(b) Median Non-causal Stability latency (in microseconds).

Figure 29: Experiment 1: Memory and Non-causal Stability latency, with stability.

and gradually increase the group size. The effect on performance due to calculating causal stability is also measured by doing runs with and without having to calculate it.

The following experiment was done with the same send interval for every peer and the interval between broadcasts is calculated following a Poisson distribution, and each peer sends 100 messages. The send intervals chosen for the links in the runs of this experiments were, in milliseconds: 10, 100 and 1000. Moreover, the network latency is calculated via the Weibull distribution, as to make the peers more desynchronized from each other. Runs were executed for groups with 4, 8, 16, 32, 64 and 128 peers. The network latency chosen for the links in the runs of this experiments is 10 milliseconds.

In this experiment, we compare our E2E TCB approach against a correct implementation of the end-to-end vector-based approach. This way we can show the performance between an efficient implementation of the VV-based causal middleware against our graph-based implementation in terms transmission overhead on the network and memory size for causality metadata.

In Figure 30a, we see similar results to the previous experiment. However, the size for the optimized E2E VV-based approach here is half of the one seen previously in the classical E2E. The E2E TCB approach provides an end-to-end happens-before with less causal dependencies transmitted on the network. At 128 nodes, the number of predecessors needed is at worst 80 for E2E TCB to provide end-to-end happens-before. However, in Optimized E2E VV, we need twice the size of nodes which is 128 entries.

As the size of version vectors is reduced to half, compared to the previous experiment, we see that reflected in the memory size. Figure 30b. In the case of the Optimized E2E VV approach the data stored can get to more than 250KB for 128 nodes, where it is negligible for the E2E TCB approach.

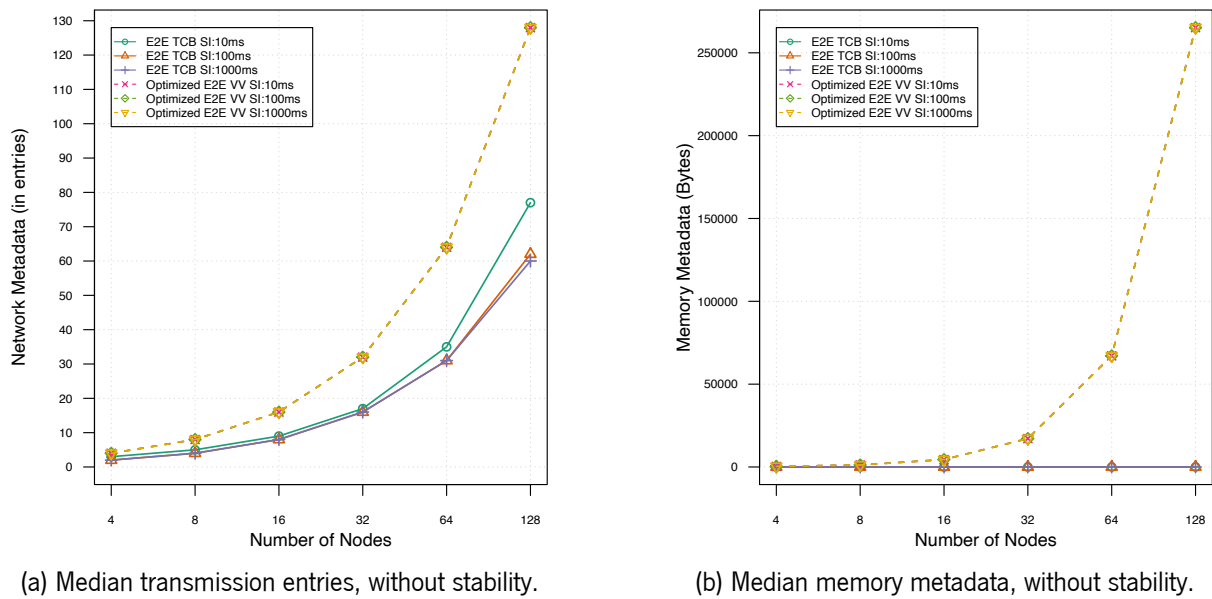


Figure 30: Experiment 2: transmission and memory metadata, without stability.

8.4.3 Experiment3: Constant amount of work for vector-based VV vs graph-based

Another experiment performed over a symmetric topology was adjusting the send interval to the group size so that the number of messages sent by all peers, per second, was constant. The motivation behind this is to see what would happen if a service's workload became more and more distributed by adding peers and dividing the work between them. Having 64 peers with a send interval of 500 milliseconds generates (on average) the same amount of messages as 128 broadcasting every second. The network latency with a Weibull distribution was set to 31 milliseconds for all group sizes. For groups with 4, 8, 16, 32, 64 and 128 peers, the send intervals were, respectively, 31, 63, 125, 250, 500 and 1000 milliseconds.

The same results from the previous experiment are shown in this experiment as well. Our graph-based algorithm scales better in terms of causality metadata transmitted on the network as shown in Figure 31a and in terms of causality metadata stored in memory as in Figure 31b.

8.4.4 Causal Stability

In this section, we revisit experiments 2 and 3 from earlier, but we focus here on causal stability. In the previous experiments shown earlier, causal stability was not calculated. Here, we show how calculating causal stability affects the size of metadata stored in memory. We also show the non-causal stability time required by each of the two algorithm.

We first start with showing for each of the two previous experiments how the metadata size in memory is affected when the algorithm are running with causal stability on.

For Experiment 2, we see in Figure 32a that the memory size is half the one in Figure 28b (from

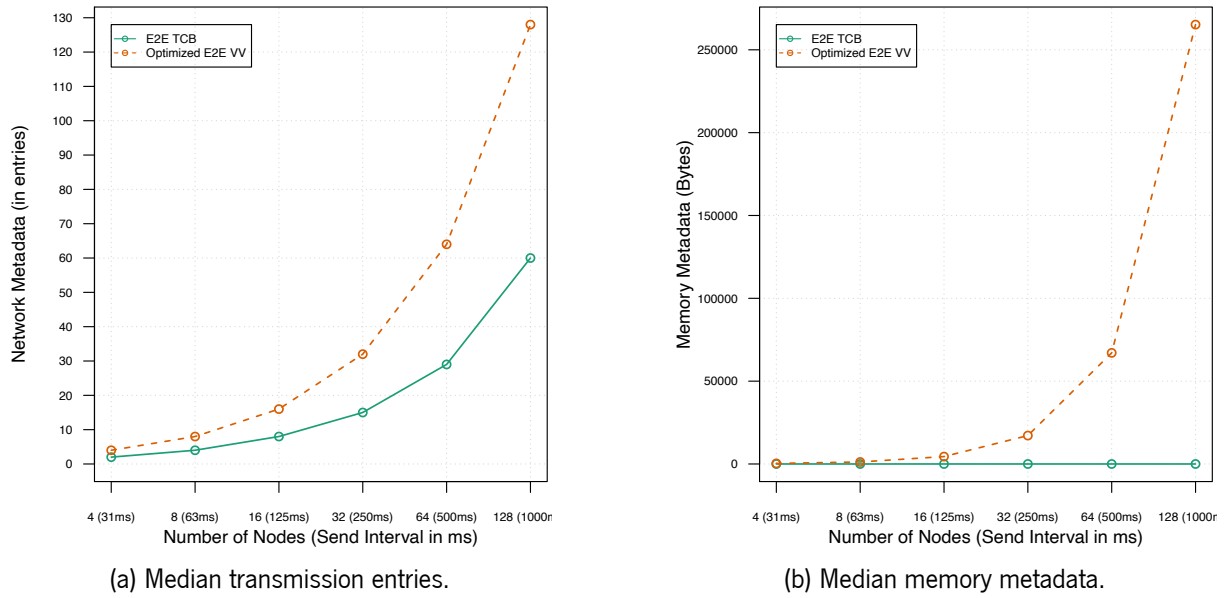


Figure 31: Experiment 3: transmission and memory, without stability.

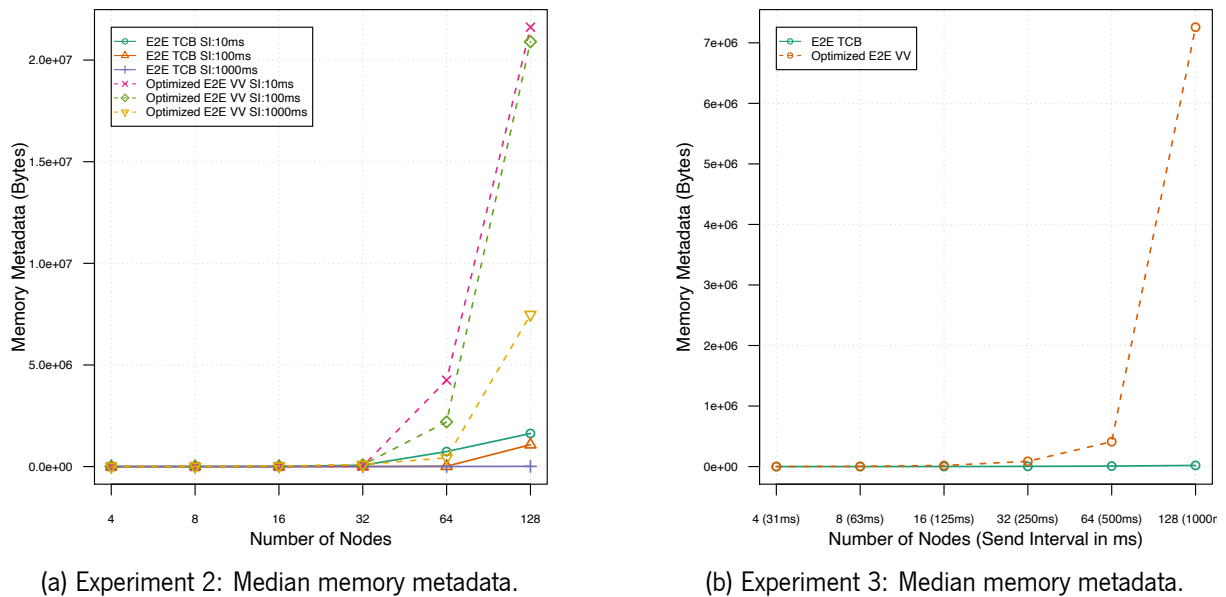
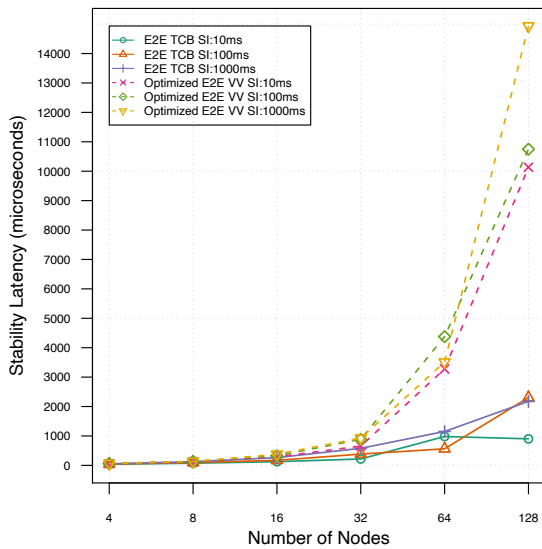


Figure 32: Experiment 2 and 3: Median memory metadata, with stability.

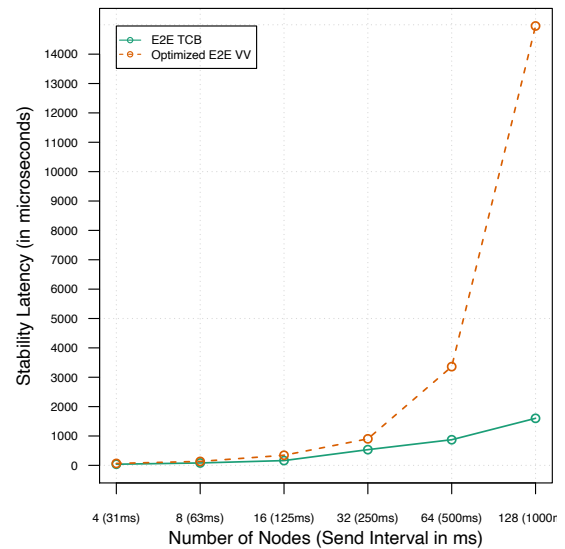
previously) due to the version vectors being half the size. The optimized E2E VV-based algorithm implements the correct end-to-end happens-before and does not require a duplication of effort and metadata sent over the network. Also, similarly to the previous results, the E2E TCB algorithm has a memory size significantly smaller (more than 20 MB for 128 peers) than the E2E VV-based algorithm (less than 2.5 MB for 128 peers).

For Experiment 3, we see in Figure 32a that the memory size gets significantly larger for the E2E VV-based approach when the group size is larger than 32 peers. It reaches around 7MB for 128 peers. However, it is almost negligible for the E2E TCB approach.

The reason for this is that for causal stability, the E2E VV-based algorithm stores a matrix of N^2 , where N is the group size. This Matrix is used to calculate the stable version vector that denotes which messages can become stable. Also, the E2E VV-based algorithm stores a StableMap for the messages that are stable in order to stabilize them in causal order. For TCB, the causal DAG stores a bitstring (one word) for every dot representing a message to compute stability.



(a) Experiment 2: Median non-causal stability latency (microseconds)



(b) Experiment 3: Median non-causal stability latency (microseconds).

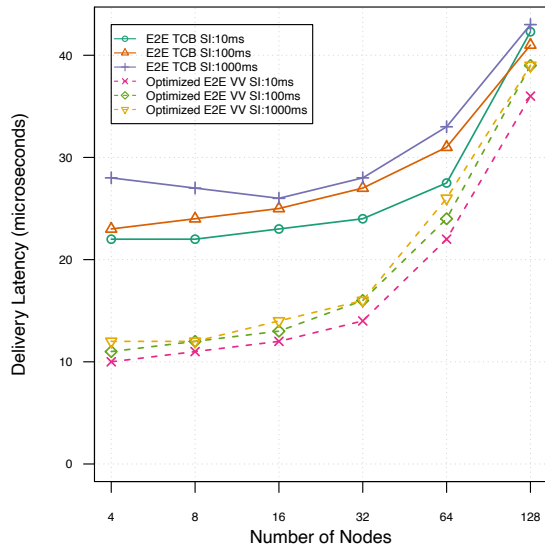
Figure 33: Experiment 2 and 3: non-causal stability latency, with stability.

In Figures 33a and 33b we can see that the TCB algorithm takes significantly less time (2-3 milliseconds for 128 peers) for non-causal stability compared to the vector-based approach (above 10 milliseconds for 128 peers). Even though the delays are not very large, we can clearly notice how the non-causal latency for stability grows much faster for the VV algorithm, which makes TCB a more scalable algorithm for larger groups.

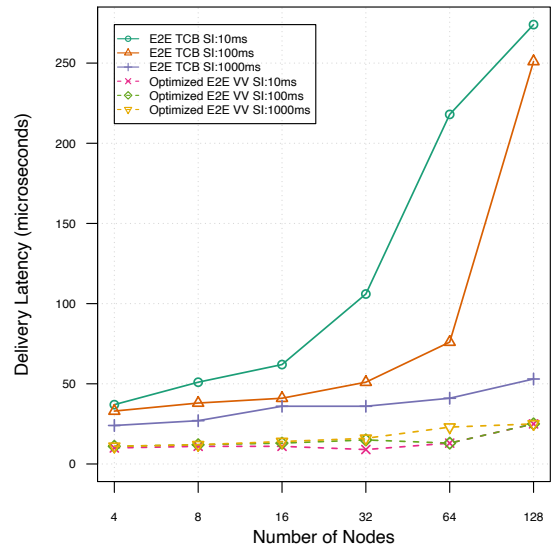
8.4.5 Causal Delivery

In this section, we revisit Experiments 2 and 3 from earlier, but we focus here on causal delivery. Specifically, the non-causal delivery latency of each of the algorithms. We start by Experiment 2 for both cases where stability is and is not being calculated. We then do the same for Experiment 3.

For Experiment 2, we see in Figure 34a that the non-causal delivery latency values are lower for the optimized E2E VV-based algorithm than for the E2E TCB algorithm. However, the results for both algorithms are close and in the order of 40 microseconds for 128 peers, which could be considered negligible. We notice a similar conclusion for Experiment 2 when the causal stability mechanism is running. In Figure 34b, the non-causal delivery latency values are also lower for the optimized E2E VV-based algorithm than for the E2E TCB algorithm. The difference is bigger here: at 128 peers, the latency for VV is around



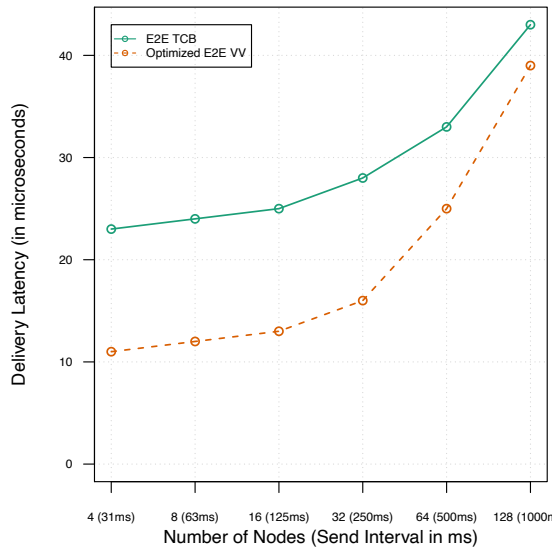
(a) Median non-causal delivery latency (microseconds), without stability.



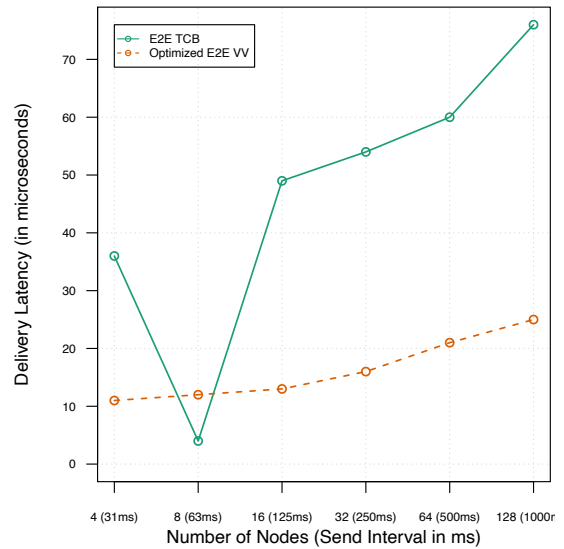
(b) Median non-causal delivery latency (microseconds), with stability.

Figure 34: Experiment 2: non-causal delivery latency.

25-50 microseconds and the latency for TCB ranges between 50 and 270 microseconds. The results are still in the order of microseconds.



(a) Median non-causal delivery latency (microseconds), without stability.



(b) Median non-causal delivery latency (microseconds), with stability.

Figure 35: Experiment 3: non-causal delivery latency.

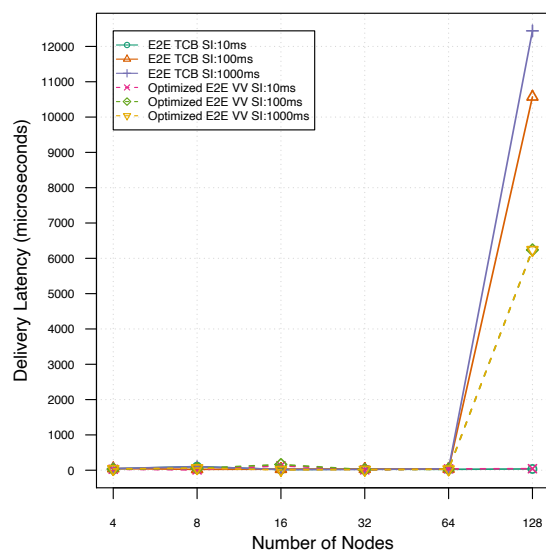
For Experiment 3, we see in Figure 35a that the non-causal delivery latency values are lower for the optimized E2E VV-based algorithm than for the E2E TCB algorithm. However, the results for both algorithms are close and in the order of 40 microseconds for 128 peers, which could be considered negligible. We also notice the same behaviour for Experiment 3 when the causal stability mechanism

is running. In Figure 35b, the non-causal delivery latency values are also lower for the optimized E2E VV-based algorithm than for the E2E TCB algorithm. The difference is a bit bigger here: at 128 peers, the latency for VV is around 25 microseconds and the latency for TCB around 75 microseconds. The results are still in the order of microseconds.

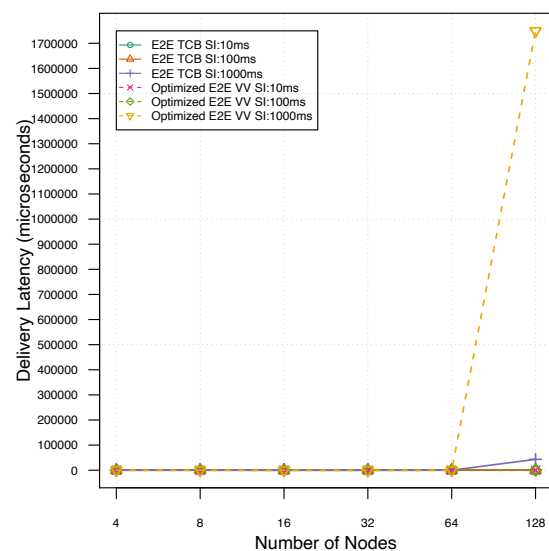
The reason for this is that when no delays happen, messages normally arrive in causal order. This means that not many messages are being queued due to missing causal dependencies. Moreover, in this simple scenario messages arrive mostly in causal order and the experiment did not run for a long time like it usually is the case for systems used in production and in real scenarios. The algorithm for TCB is more complex than the algorithm for VV, and therefore more work is being done in the former. This is why we notice a higher non-causal latency in a simple scenario.

8.4.6 Slow Links

In the previous subsection 8.4.5, we showed results for non-causal delivery latency in simple scenarios that were not very interesting. Worst case scenarios were delays and partitions occur, similarly to real world scenarios are more interesting to test the performance of the two algorithms. The aim of this experiment is to see how the TCB and VV middleware implementations perform when simulating a network partition by adding a slow link. We simulated a slow link for experiments 2 and 3, and only the median results of the peers with the slow link are presented since the other peers are not affected by it and their results are very similar to the ones from the experiments where no slow links were added. Each peer sends 100 messages, with an average send interval of 10, 100 and 1000ms milliseconds between broadcasts. The slow link is 10 times slower than the network latency.



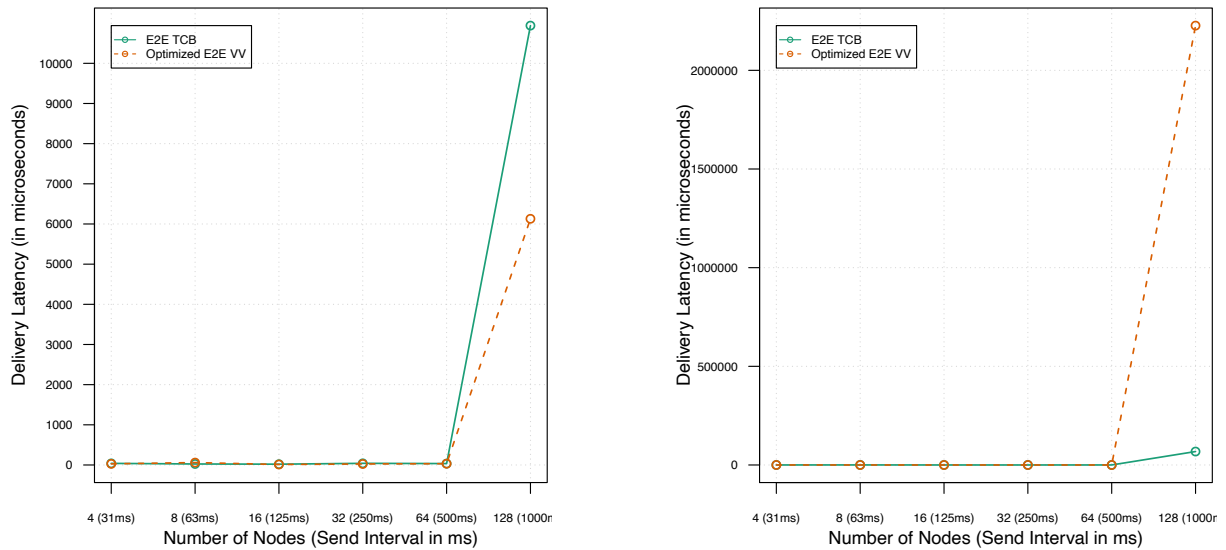
(a) Median non-causal delivery latency (microseconds) on slow link, without stability.



(b) Median non-causal delivery latency (microseconds) on slow link, with stability.

Figure 36: Experiment 2: non-causal delivery latency on slow link.

For Experiment 2, we see in Figure 36a that when a slow link is added, the non-causal delivery latency values are now higher for the optimized E2E VV-based algorithm than for the E2E TCB algorithm. And, the results are in the order of 10-12 milliseconds for 128 peers for the VV-based algorithm. We notice higher latency values for Experiment 2 when the causal stability mechanism is running. In Figure 36b, the non-causal delivery latency values are also higher for the optimized E2E VV-based algorithm than for the E2E TCB algorithm. And the difference is bigger here: at 128 peers, the latency for VV is around 1.7 seconds while the latency for TCB is around 50 milliseconds at most. This clearly shows how the results shifted from the ones presented in subsection 8.4.5, where no slow links were added.



(a) Median non-causal delivery latency (microseconds) on slow link, without stability.

(b) Median non-causal delivery latency (microseconds) on slow link, with stability.

Figure 37: Experiment 3: non-causal delivery latency on slow link.

Similarly to the results in Experiment 2 above, we notice that TCB performs and scales better than VV for Experiment 3 in terms of non-causal delivery latency when a slow link is added. Figure 37a shows that the non-causal delivery latency for VV is around 11 milliseconds for 128 peers, while it is only around 6 milliseconds for TCB for the same group size. When the causal stability mechanism is running in both versions, the TCB performs a lot better and the gap between both algorithm becomes bigger. For instance for 128 peers, the non-causal delivery latency for TCB is around 70 milliseconds which a lot smaller than 2.2 seconds for VV for the same group size.

Use Cases

This chapter focuses on showing how the work done during this thesis was used in existing software, data stores and other research as well.

9.1 ASPAS

As Secure as Possible Available Systems [91] (ASPAS) is a Byzantine resilient AP system. It follows an optimistic approach to maintain a single round-trip response time, and allows the detection of Byzantine replicas in the background, i.e., off the critical path of clients requests. My contribution to this work was in using causal stability and operation-based CRDTs to calculate the common partially-ordered log (polog) of operations and produce a certificate for BFT under EC.

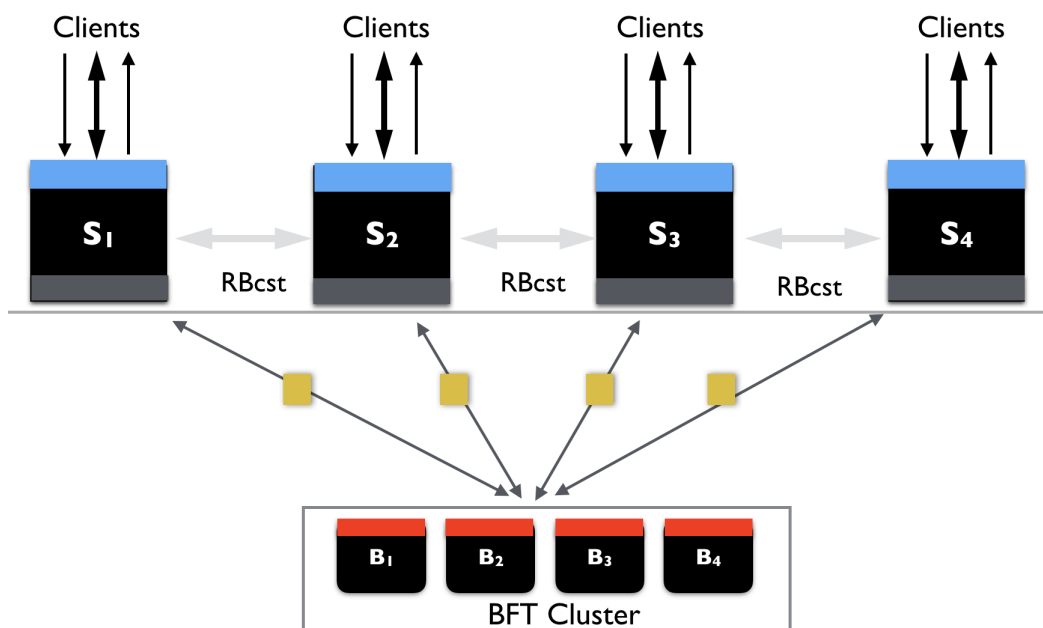


Figure 38: ASPAS architecture.

9.1.1 Data Types for backup and recovery

The possibility to guarantee Byzantine Fault Tolerant reads in the system without forfeiting system's availability is based on the ability to provide certificates for operations by the BFT cluster. For that to work correctly we provide a mechanism that allows the system to rebuild "safe" states i.e. free of operations initiated by byzantine nodes. We present below two versions of such mechanism: a naive one using "undo" and "redo", and a more elegant, generic, and safe alternative that we adopt, using the "clone" functionality.

9.1.1.1 Undo and Redo functionality

This mechanism is made possible through the use of Undo and Redo functionality for the Data Types in use. For every instance of a data type, the received operations that mutate the state of that instance are stored in a partially-ordered log, `polog`. This allows updating the state of the data types at the application servers without delaying it with the byzantine fault tolerance protocol running in the BFT cluster. The BFT cluster protocol, running in parallel to the updates on the application server, provides information on which operations are safe and which ones were initiated by a byzantine node. The information provided is through the version vectors that act as a timestamp for each operation. In addition to the `polog` logging all the operations of an instance, each data type supports an *undo* (*redo* respectively) functionality, that given the `polog`, an unsafe (safe respectively) timestamp and the state of the instance, undoes (redoes respectively) the effect of that operation from the `polog`. The undo and redo functionality allows to rebuild the state of a data type, from the original `polog`, allowing to filter unsafe operations and redoing the safe ones. However, this functionality is data type specific and requires extra care in specifying the "undo" for every new data type. We opt for a more elegant and generic approach using the "clone" functionality, that we explain next.

9.1.1.2 Data Types with clone functionality

A more efficient way to provide byzantine fault tolerant state convergence would be by using the clone functionality. This functionality is equivalent to the undo/redo functionality, but it provides a simpler, more generic and elegant approach, independently of the data type used. In this version, the state of each data type is specified as a `polog` mapping timestamps (version vectors) to the relative operation (as in Pure op-based CRDTs [7]). This means that the `polog` used "outside" the data type in undo/redo version is not needed anymore. The clone functionality receives two timestamps and the old safe state as input: a lower bound t as the last safe version vector (bottom \perp by default), and the upper bound t' as the new safe version vector calculated by the BFT cluster and s , the previously calculated safe state of the data type (by default the state is the initial empty state the data type). Instead of redoing all the operations as previously, the clone functionality "clones" the state of the current data type by copying all operations having timestamps t'' such that $t \leq t'' \leq t'$ (as per the Lamport's happens-before relation). As all

data types in this versions have a `polog` as their state, the clone functionality is generic for all of them. However, we illustrate this with an example of the Set data type.

$$\begin{aligned}
 \Sigma = T \hookrightarrow O \quad \sigma_i^0 &= \{\} \\
 \text{effect}(o, t, s) &= s \cup \{(t, o)\} \\
 &\text{(with } o = \text{ either } [\text{add}, v] \text{ or } [\text{rmv}, v]) \\
 \text{clone}(t, t', s, s') &= s' \cup \{(t'', o) \in s \mid t \leq t'' \leq t'\} \\
 \text{eval}(\text{rd}, s) &= \{v \mid (t, [\text{add}, v]) \in s \wedge \\
 &\quad \nexists (t', [\text{rmv}, v]) \in s \cdot t < t'\}
 \end{aligned}$$

Figure 39: `polog` based Add-Wins Set with Clone functionality

Add-Wins Set Figure 39 presents the specs of the Add-Wins Set with the clone functionality. The state of the `AWSet` is a `polog` mapping timestamps $t \in T$ to operations $o \in O = \{[\text{add}, v], [\text{rmv}, v]\}$. `effect` updates the state by adding the timestamped operation (t, o) to the `polog`. Finally, `clone` creates a new state (`polog`) by copying all timestamped operations between t and t' and merges it with the previously calculated safe state s' . As the application server guarantees causal consistency, all replicas have seen the same operations between the previously calculated lower bound and the upper bound version vector provided by the BFT cluster in the same causal order. It is trivial to note that cloning the state, through copying operations having timestamps between the two provided bounds, preserves the convergence and thus provides a byzantine consistent state among all replicas.

Remark on the Counter Data Type In this approach, the state `polog`-based Counter increases with the number of operations (`inc`, `dec`). This is an overhead in terms of space when compared to an integer state. We do not address this here, but compaction techniques for a `polog`-based Counter has been proposed in [92] to solve this problem.

9.2 Lasp

Another contribution done during my Ph.D. was participating in the Google Summer of Code'16 program where I saw an interesting opportunity to improve upon existing software and contribute to the open source community. I submitted a proposal to implement the operation-based CRDTs framework in Lasp [93], a programming model for building correct distributed applications. The proposal was accepted and the work was rewarded with a certificate from Google. The code is now public and available on GitHub (Section 9.6). I also had the opportunity to present the work in a poster session at INESC TEC, Porto for "HASLAB OPEN DAY 2016".

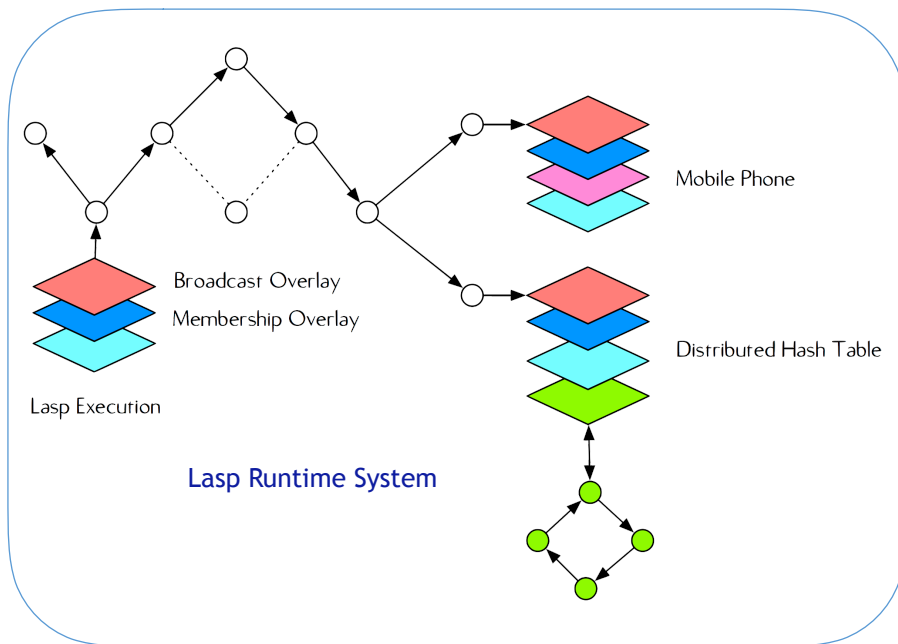


Figure 40: Lasp runtime system.

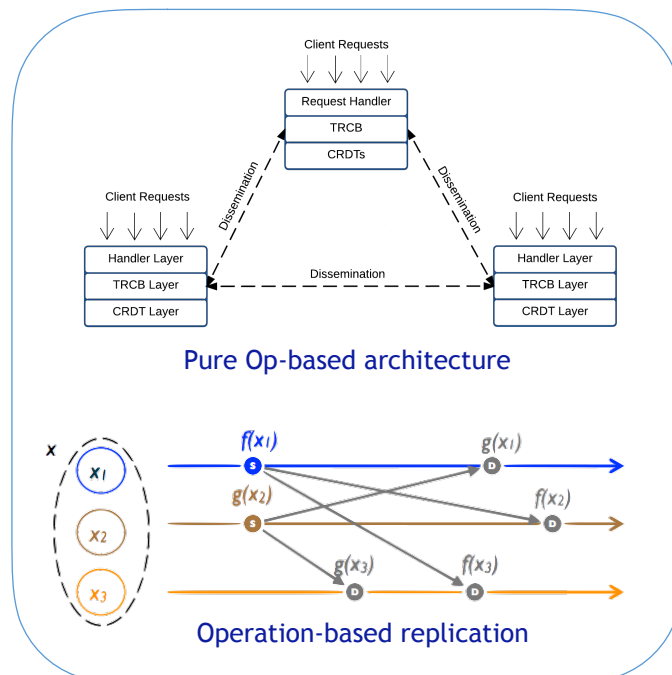


Figure 41: Lasp pure op-based CRDTs.

$$\begin{aligned}
\Sigma &= \mathbb{N} \times \mathcal{P}(T) & \sigma^0 &= (0, \{\}) \\
\text{prepare}(o, (n, s)) &= o & & \text{(with } o \text{ either inc or reset)} \\
\text{effect}(\text{inc}, t, (n, s)) &= (n, s \cup \{t\}) \\
\text{effect}(\text{reset}, t, (n, s)) &= (0, s \setminus \{t' \in s \mid t' < t\}) \\
\text{stabilize}(t, (n, s)) &= (n + 1, s \setminus \{t\}) \\
\text{eval}(\text{value}, (n, s)) &= n + |s|
\end{aligned}$$

Figure 42: Concrete Resettable Counter Implementation

I improved the design of the reliable causal broadcast middleware implemented for Lasp. This middleware is an essential component of the op-based platform and, for that reason, improving it is a main contribution that would lead to optimizing the op-based platform. The current implementation of this middleware presents a more scalable and dynamic version.

9.3 AntidoteDB

As part of the Syncfree project [94], a FP7 EU project on large-scale replicated computation without synchronization. One of the major results of this project has been the geo-replicated highly available data store Antidote [95]. A central component of this data store is a library for conflict-free replicated data types. As a contribution to this project, I was responsible of replacing the initial implementation of CRDTs with a new optimized version. I improved the design of existing data types (add-wins set, remove-wins set, pn-counter) and adapted the design of further op-based types such as maps and flags for Antidote. I have also been contributing to Antidote under the EU project Lightkone [96] by implementing a mechanism that allows compressing operations before sending to other replicas. The code is available on GitHub (Section 9.6) in Erlang with unit and property-based tests.

Moreover, while working on the map data type in Antidote, I tackled the problem of embedding counters inside maps that led to a paper entitled “Compact Resettable Counter through Causal Stability” [92] that was submitted to PAPOC/EUROSYS’17. In that paper, I presented the problem, proposed a new design for a counter that solves it and showed how causal stability could be used to garbage collect meta-data and lead to more compact state.

9.4 Redis with relaxed consistency

The popularity of Redis [76] stems from its speed, rich semantics, and stability. It was primarily designed and generally used in a single server deployment model. A single Redis instance is a remote centralized solution, and not a distributed system. However, Redis provides distributed solutions known as Redis

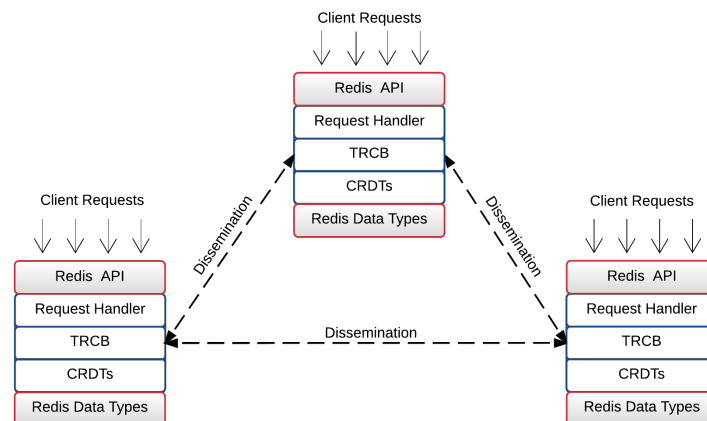


Figure 43: The general architecture of our multi-master proposed solution.

Sentinel and Redis Cluster which allow using multiple instances, asynchronous master-slave replication, and automatic fail over process.

We contributed in implementing pure op-based CRDTs to build a multi-master replication feature in Redis [76], the famous in-memory cache system. We choose Redis because: (1) it helps us demonstrate how to integrate pure op-based CRDTs in an existing popular system; (2) we contribute with the community in building a crucial multi-master feature that is currently missing in Redis. Our experience reveals that implementing pure op-based CRDTs for many data types is straightforward, whereas integrating them in an existing system is somehow challenging if the aim is to preserve the legacy API and code-base intact. The code is available on GitHub (Section 9.6) in C.

For the implementation of the multi-master replication feature using pure op-based CRDTs, we integrated three layers into Redis Server's code. We present below the three layers, shown in Figure 43, with a brief description of each layer.

Request handler The handler layer is the intermediate layer between Redis Client API and the TRCB and CRDT layers. Every client request is redirected by the original Redis Client API to this layer. Then, this layer prepares a client object containing the operation and arguments (if any), serializes it and ships it to the TRCB layer for dissemination.

Tagged Reliable Causal Broadcast The second layer (TRCB) is used to broadcast client requests (operations and arguments) to all nodes in the cluster. This layer tags the client objects received from the handler with a timestamp needed to guarantee causal delivery at each node. Also, the TRCB is implemented in a way to guarantee exactly-once delivery of each client object to each node in the cluster.

CRDT layer The CRDT layer is where we implement the pure op-based CRDTs and their related structures, such as the PLOG (Partially-Ordered Log), a map where the key is a timestamp and the value is the

operation and arguments, following the designs and specifications in [7]. In addition to that, we perform a two-phase POLog compaction to make CRDTs even more efficient. The first phase removes obsolete and redundant information from the POLog in a way that does not affect the result of the queries, keeping only relevant operations. The second one is by using *causal stability* information from the middleware to discard the timestamps of causally stable operations and move them from the POLog to the Redis data types.

Challenges We had to figure out solutions for the challenges we faced. The challenges are in terms of architecture design, preserving original API with minor changes, reusability of Redis code, configurability of the system, choice of messaging pattern and implementation of causal stability. A more detailed version of this work is published in [97].

9.5 Minidote

Minidote [98] is a lightweight, replicated key-CRDT store. The anticipated use cases for Minidote are those where the managed data fits well into the memory of each node in the cluster. The data is automatically replicated on each node, while concurrent updates are resolved using CRDTs. Minidote provides causal consistency with atomic batch-reads and batch-updates.

Compared to Antidote, Minidote includes the following simplifications, which makes it more lightweight and therefore allow it to run well on less powerful devices:

1. There is no sharding of data onto multiple machines within a data center. Each replication site only consists of a single machine.
2. There is (currently) no support for interactive transactions.

Figure 44 sketches the high-level architecture of Minidote . The components drawn with dashed lines are reusable components that are shared with Antidote.

My contribution was in replacing the inter-dc replication service used in Antidote with a different causal broadcast service provided by Camus. Camus is a *CAusal MULTicast Service*, that provides different back-ends to guarantee a reliable dissemination and delivery respecting causal order at all replicas using the service. The back-end used in Minidote is an implementation of tagged casual broadcast (TCB) protocol. In short, TCB guarantees that messages will be delivered respecting the end-to-end happens-before relation as seen by the application. Moreover, it uses smaller size timestamps than vector clocks to encode the causality between messages, a causal graph to store the dependencies between messages and an efficient algorithm for causal delivery and stability. The code is available on GitHub (Section 9.6) in Erlang with unit and property-based tests.

The `minidote_server` process, at every Minidote instance, stores locally in its state a dot, i.e. a pair of $(nodeId, counter)$, which serves as a unique identifier for a message to be broadcast, and a context,

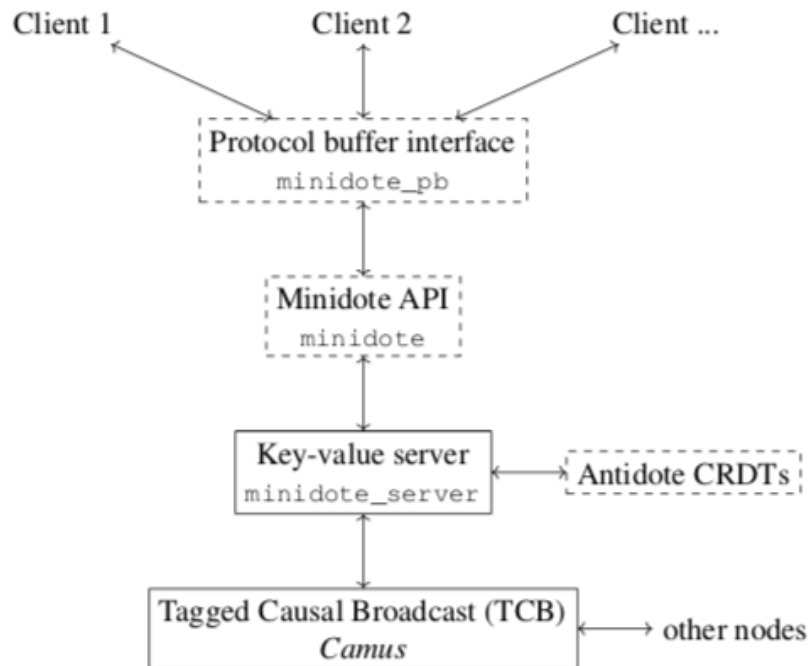


Figure 44: High-level architecture of Minidote.

i.e. a set of dots, which stores the ids/dots of the most recently delivered messages. The context serves as the causal dependency of the next message to be broadcast.

To explain the integration of TCB in Minidote, we explain below the API of Camus:

- `tcbcast(Msg, State)`: broadcasts a message to every process in the group membership. The client process (here `minidote_server`) will be responsible for tagging the message to be sent with a new dot and the causal context.
- `tcdeliver(Msg, Timestamp, State)`: delivers a message to the application with the tag. The client process or application (here: `minidote_server`) also implements message delivery. For instance, in the case of an operation-based AW-Set S_a , `tcdeliver` would add the operation (add, e , and respective timestamp t') to the state S_a .
- `tcstable(Msg, Timestamp, State)`: stabilizes a delivered message.

The `minidote_server` process also needs to handle messages coming from the TCB service in the format `camus, Opaque`. Upon receiving such a message, the function `camus:handle(Opaque, State)` is called, where `State` comprises the local dot and context. The TCB process will update the state and return it along with the type of message (`deliver, stable`), and the payload. The `minidote_server` process updates the state and based on the type of message calls `tcdeliver` or `tcstable`.

Minidote does not fully benefit from all the features of the TCB protocol. The reason is that Minidote uses Antidote's classical operation-based CRDTs as data types which use random unique tokens as a unique identifier and do not provide meta data information on the provenance of the operations. This

means that currently Minidote does its own causal tagging of operations (using random tokens) to be used for comparing operations, and the dot and context tagging provided by TCB to provide causal delivery order. An improved implementation should use the dot and context as the only tag to provide end-to-end causal delivery, without a need of duplication of effort (tagging) and extra metadata. With such modifications to the CRDTs, Minidote could benefit from an efficient garbage collection of CRDT metadata that is provided by causal stability and that is not possible with the current CRDT library. Finally, the current version works in the scope of static systems and therefore does not allow nodes to join and leave. As the anticipated use cases for Minidote are targeting deployments closer to the edge, nodes are likely to join and leave more frequently as with Antidote's data center deployment. We therefore plan to extend the current version to cover the scope of dynamic systems by extending the API of Minidote to deal with joining, leaving, state-transfer and using the dynamic version of TCB.

9.6 Software, libraries and artifacts

- <https://github.com/gyounes/camus>, Erlang implementation of different causal delivery and stability middleware including: Classical WV-based, Classical E2E WV-based, Optimised E2E WV-based, E2E TCB algorithm, Dynamic E2E TCB.
- https://github.com/gyounes/camus_exp, Erlang implementation for deployment, orchestration and evaluation of Camus using docker and kubernetes.
- <https://github.com/gyounes/minidote>, Erlang implementation of Minidote, a lightweight CRDT data store using TCB as causal delivery middleware.
- https://github.com/AntidoteDB/antidote_crdt, Erlang implementation of Op-based Conflict-free Replicated Data Types (CRDTs) in Erlang.
- <https://github.com/gyounes/RCB>, Erlang implementation of reliable causal broadcast.
- <https://github.com/lasp-lang/types>, Prototype implementation of Conflict-free Replicated Data Types (CRDTs) in Erlang.
- <https://github.com/lasp-lang/ishikawa>, Erlang implementation of tagged reliable causal broadcast over partisan/hyparview.
- <https://github.com/haslab/EC/tree/master/topic1/redis>, PoC: Implementation of multimaster replication in Redis).

Conclusions and Future Perspectives

The work in this thesis leveraged the importance of giving another thought to causal consistency techniques designed for systems of an old era, back three decades. My work revisited the end to end causal delivery stack from the application to the causal middleware API, including its internal logic. This led to novel class Dynamic End-to-End Reliable Causal Delivery Middleware for Geo-Replicated Services, tailored for modern causality systems.

My research started by analyzing the causality requirements in modern applications that are complex and multi-threaded. I explained how distributed systems practitioners could try to achieve end-to-end happens-before incorrectly by identifying the pitfalls in trying to achieve that. This work affirms the importance of going to fundamentals while designing complex systems before engineering them. To this end, I introduced Tagged Causal Delivery (Broadcast) that ensure an end-to-end happens-before. Tagged Causal Delivery redefines the happens-before relation to reflect application level dependencies and provide knowledge about the causal relations and concurrency between operations to the application.

On the other hand, studying the interaction between applications and the underlying causal broadcast middlewares was helpful to inspire new functionalities that makes consistency at the application semantics correct and efficient. For this, I introduced a new causal broadcast architecture and API that grants the application layer access to causality information for timestamping as well as for garbage collection, using causal stability. I have shown that these new middleware functionalities reduce the bandwidth and memory overhead significantly without compromising latency. Nevertheless, systems in which causal stability is not required may not take advantage of this optimization, while paying a little cost in causal delivery. Indeed, the delivery latency overhead in my work is negligible overhead (less than millisecond) over classical systems, while improving the causal stability one order of magnitude.

One more aspect addressed in the work was the internal abstractions used in causal middlewares. I have shown that the classical causal broadcast algorithms, based on version clocks, are not efficient in memory and stability as the number of replicas increase. This led to the introduction of dependency dots and a causal DAG as data structures to represent the causal dependencies and metadata. I designed and implemented an efficient graph-based Tagged Causal Delivery middleware. In the experiments, I showed how the Tagged Causal Delivery middleware scales better than an optimized E2E VV-based algorithm in

terms of transmission of meta data on the network, size of metadata in memory and the non-causal stability latency. I also showed that the algorithm is more resilient in case of unreliable networks, where delays and partitions occur.

Dynamic membership was another focus in my thesis, inspired by the advent of elastic systems, enabled through virtualization techniques. I introduced the first causal delivery middleware that tolerates dynamic membership while maintaining the causal consistency guarantees. I showed how the new graph-based TCB stability allows for the system operation in a high churn dynamic environment and in a non-blocking fashion.

During the course of my thesis, I applied these concepts to many use cases presented in the last chapter. The work was part of the EU H2020 projects, LightKone and Syncfree, as well as other national projects. Part of the work was also funded through the Google Summer Code grant. All these works obviously consumed significant time of my thesis time; however, this paid off in two main ways (in addition to the funding support). The first is digging in the engineering of the middleware which helped in realizing the theoretical pitfalls as well as resulting in a well structured and reliable TCB codebase in *Erlang*. The second benefit was increasing the impact of my work through other works, like Lasp and Antidote, that have seen high adoption in the industry.

10.1 Future Work

One direction would be extending our current middleware. More work can be done on batching operations that would improve the performance in terms of non-causal latency and stability. Also, work on causal stability required the delivery of operations to the entire system. It would be interesting to study the usefulness and technique of partial stability in a quorum-based fashion, e.g., K-stability, where the system is causality stable considering quorum of K replicas out of the total membership. Making our causal middleware more adaptive by allowing it to dynamically pick the optimal send rates to be used based on the network delays. An important part to work on is to improve the mechanism used in the TCB middleware to compare operations at the client; the mechanism used now is still slower than comparing two version vectors. Finally, we would like to work on finding solutions for node migration, persistence and mostly extend the middleware to allow partial replication.

Another direction would be extending and improving the existing pure operation-based CRDTs design and implementation. This novel CRDTs library and the current middleware could be used together to develop a highly-available and scalable data store that could offer interesting trade-offs. Moreover, it would be also interesting to work on new CRDTs like sequences that could be used for collaborative editing, without the need to synchronise often.

Bibliography

- [1] João M. Lourenço. *The NOVAthesis L^AT_EX Template User's Manual*. NOVA University Lisbon. 2021. url: <https://github.com/joaomlourenco/novathesis/raw/master/template.pdf> (cit. on p. ii).
- [2] J. Brzezinski, C. Sobaniec, and D. Wawrzyniak. "From session causality to causal consistency". In: *12th Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2004. Proceedings*. 2004, pp. 152–158. doi: [10.1109/EMPDP.2004.1271440](https://doi.org/10.1109/EMPDP.2004.1271440) (cit. on pp. 1, 9).
- [3] Prince Mahajan, Lorenzo Alvisi, and Mike Dahlin. "Consistency, Availability, and Convergence". In: (May 2012) (cit. on pp. 1, 8, 9, 13).
- [4] Kenneth Birman, André Schiper, and Pat Stephenson. "Lightweight Causal and Atomic Group Multicast". In: *ACM Trans. Comput. Syst.* 9.3 (Aug. 1991), pp. 272–314. issn: 0734-2071. doi: [10.1145/128738.128742](https://doi.org/10.1145/128738.128742). url: <http://doi.acm.org/10.1145/128738.128742> (cit. on pp. 2, 37–39, 55, 62).
- [5] Sebastian Burckhardt. "Principles of Eventual Consistency". In: *Found. Trends Program. Lang.* 1.1-2 (Oct. 2014), pp. 1–150. issn: 2325-1107. doi: [10.1561/2500000011](https://doi.org/10.1561/2500000011). url: <http://dx.doi.org/10.1561/2500000011> (cit. on pp. 2, 37).
- [6] Hyun-Gul Roh et al. "Replicated Abstract Data Types: Building Blocks for Collaborative Applications". In: *J. Parallel Distrib. Comput.* 71.3 (Mar. 2011), pp. 354–368. issn: 0743-7315. doi: [10.1016/j.jpdc.2010.12.006](https://doi.org/10.1016/j.jpdc.2010.12.006). url: <http://dx.doi.org/10.1016/j.jpdc.2010.12.006> (cit. on pp. 2, 63).
- [7] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. "Making Operation-Based CRDTs Operation-Based". In: *Proceedings of the 14th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems - Volume 8460*. Berlin, Heidelberg: Springer-Verlag, 2014, pp. 126–140. isbn: 978-3-662-43351-5. doi: [10.1007/978-3-662-43352-2_11](https://doi.org/10.1007/978-3-662-43352-2_11). url: https://doi.org/10.1007/978-3-662-43352-2_11 (cit. on pp. 2, 38, 42, 62, 110, 115).

-
- [8] Daniel Abadi. “Consistency tradeoffs in modern distributed database system design: CAP is only part of the story”. In: *Computer* 45.2 (2012), pp. 37–42 (cit. on pp. 6, 7, 9).
- [9] Eric A. Brewer. “Towards Robust Distributed Systems (Abstract)”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: Association for Computing Machinery, 2000, p. 7. isbn: 1581131836. doi: [10.1145/343477.343502](https://doi.org/10.1145/343477.343502). url: <https://doi.org/10.1145/343477.343502> (cit. on pp. 6–8, 37).
- [10] Seth Gilbert and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: *SIGACT News* 33.2 (June 2002), pp. 51–59. issn: 0163-5700. doi: [10.1145/564585.564601](https://doi.org/10.1145/564585.564601). url: <https://doi.org/10.1145/564585.564601> (cit. on pp. 6–8, 37).
- [11] L. Lamport. “Proving the Correctness of Multiprocess Programs”. In: *IEEE Transactions on Software Engineering* SE-3.2 (1977), pp. 125–143. doi: [10.1109/TSE.1977.229904](https://doi.org/10.1109/TSE.1977.229904) (cit. on p. 7).
- [12] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. “Impossibility of Distributed Consensus with One Faulty Process”. In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382. issn: 0004-5411. doi: [10.1145/3149.214121](https://doi.org/10.1145/3149.214121). url: <https://doi.org/10.1145/3149.214121> (cit. on p. 7).
- [13] Wyatt Lloyd et al. “Don’t Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS”. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. SOSP '11. Cascais, Portugal: Association for Computing Machinery, 2011, pp. 401–416. isbn: 9781450309776. doi: [10.1145/2043556.2043593](https://doi.org/10.1145/2043556.2043593). url: <https://doi.org/10.1145/2043556.2043593> (cit. on pp. 10, 30).
- [14] Peter Bailis et al. “The potential dangers of causal consistency and an explicit solution”. In: *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM. 2012, p. 22 (cit. on pp. 10, 38, 47).
- [15] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. “Towards a Scalable, Distributed Metadata Service for Causal Consistency under Partial Geo-Replication”. In: *Proceedings of the Doctoral Symposium of the 16th International Middleware Conference*. Middleware Doct Symposium '15. Vancouver, BC, Canada: Association for Computing Machinery, 2015. isbn: 9781450337281. doi: [10.1145/2843966.2843971](https://doi.org/10.1145/2843966.2843971). url: <https://doi.org/10.1145/2843966.2843971> (cit. on p. 10).
- [16] Peter Bailis et al. “Highly Available Transactions: Virtues and Limitations”. In: *Proc. VLDB Endow.* 7.3 (Nov. 2013), pp. 181–192. issn: 2150-8097. doi: [10.14778/2732232.2732237](https://doi.org/10.14778/2732232.2732237). url: <https://doi.org/10.14778/2732232.2732237> (cit. on p. 11).
- [17] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: a correctness condition for concurrent objects”. In: *ACM Transactions on Programming Languages and Systems* 12 (1990), pp. 463–492 (cit. on pp. 12, 31).

- [18] George V. Neville-Neil. “Time is an Illusion Lunchtime Doubly So”. In: *Commun. ACM* 59.1 (Dec. 2015), pp. 50–55. issn: 0001-0782. doi: [10.1145/2814336](https://doi.org/10.1145/2814336). url: <https://doi.org/10.1145/2814336> (cit. on p. 12).
- [19] Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs”. In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691. doi: [10.1109/TC.1979.1675439](https://doi.org/10.1109/TC.1979.1675439) (cit. on p. 12).
- [20] Mustaque Ahamad et al. “Causal Memory: Definitions, Implementation, and Programming”. In: *Distrib. Comput.* 9.1 (Mar. 1995), pp. 37–49. issn: 0178-2770. doi: [10.1007/BF01784241](https://doi.org/10.1007/BF01784241). url: <https://doi.org/10.1007/BF01784241> (cit. on p. 13).
- [21] Hagit Attiya, Faith Ellen, and Adam Morrison. “Limitations of Highly-Available Eventually-Consistent Data Stores”. In: *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing*. PODC '15. Donostia-San Sebastián, Spain: Association for Computing Machinery, 2015, pp. 385–394. isbn: 9781450336178. doi: [10.1145/2767386.2767419](https://doi.org/10.1145/2767386.2767419). url: <https://doi.org/10.1145/2767386.2767419> (cit. on pp. 13, 37).
- [22] Werner Vogels. “Eventually Consistent”. In: *Commun. ACM* 52.1 (Jan. 2009), pp. 40–44. issn: 0001-0782. doi: [10.1145/1435417.1435432](https://doi.org/10.1145/1435417.1435432). url: <https://doi.org/10.1145/1435417.1435432> (cit. on p. 14).
- [23] Leslie Lamport. “Time, Clocks, and the Ordering of Events in a Distributed System”. In: *Commun. ACM* 21.7 (July 1978), pp. 558–565. issn: 0001-0782. doi: [10.1145/359545.359563](https://doi.org/10.1145/359545.359563). url: <https://doi.org/10.1145/359545.359563> (cit. on pp. 16, 25, 39).
- [24] Kenneth P. Birman and Thomas A. Joseph. “Reliable Communication in the Presence of Failures”. In: *ACM Trans. Comput. Syst.* 5.1 (Jan. 1987), pp. 47–76. issn: 0734-2071. doi: [10.1145/7351.7478](https://doi.org/10.1145/7351.7478). url: <https://doi.org/10.1145/7351.7478> (cit. on pp. 16, 22, 25–28, 39).
- [25] Bernadette Charron-Bost. “Concerning the Size of Logical Clocks in Distributed Systems”. In: *Inf. Process. Lett.* 39.1 (July 1991), pp. 11–16. issn: 0020-0190. doi: [10.1016/0020-0190\(91\)90055-M](https://doi.org/10.1016/0020-0190(91)90055-M). url: [https://doi.org/10.1016/0020-0190\(91\)90055-M](https://doi.org/10.1016/0020-0190(91)90055-M) (cit. on p. 16).
- [26] D. S. Parker et al. “Detection of Mutual Inconsistency in Distributed Systems”. In: *IEEE Trans. Softw. Eng.* 9.3 (May 1983), pp. 240–247. issn: 0098-5589. doi: [10.1109/TSE.1983.236733](https://doi.org/10.1109/TSE.1983.236733). url: <https://doi.org/10.1109/TSE.1983.236733> (cit. on pp. 16, 20).
- [27] Michel Raynal and Mukesh Singhal. “Logical Time: Capturing Causality in Distributed Systems”. In: *Computer* 29.2 (Feb. 1996), pp. 49–56. issn: 0018-9162. doi: [10.1109/2.485846](https://doi.org/10.1109/2.485846). url: <https://doi.org/10.1109/2.485846> (cit. on pp. 16, 18).

- [28] Reinhard Schwarz and Friedemann Mattern. “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail”. In: *Distrib. Comput.* 7.3 (Mar. 1994), pp. 149–174. issn: 0178-2770. doi: [10.1007/BF02277859](https://doi.org/10.1007/BF02277859). url: <https://doi.org/10.1007/BF02277859> (cit. on pp. 16, 18).
- [29] Avinash Lakshman and Prashant Malik. “Cassandra: A Decentralized Structured Storage System”. In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. issn: 0163-5980. doi: [10.1145/1773912.1773922](https://doi.org/10.1145/1773912.1773922). url: <https://doi.org/10.1145/1773912.1773922> (cit. on p. 17).
- [30] C. J. Fidge. “Timestamps in message-passing systems that preserve the partial ordering”. In: *Proceedings of the 11th Australian Computer Science Conference* 10.1 (1988), pp. 56–66. url: <http://sky.scitech.qut.edu.au/~fidgec/Publications/fidge88a.pdf> (cit. on p. 19).
- [31] Friedemann Mattern. “Virtual Time and Global States of Distributed Systems”. In: *PARALLEL AND DISTRIBUTED ALGORITHMS*. North-Holland, 1988, pp. 215–226 (cit. on pp. 19, 40).
- [32] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. “Preserving and Using Context Information in Interprocess Communication”. In: *ACM Trans. Comput. Syst.* 7.3 (Aug. 1989), pp. 217–246. issn: 0734-2071. doi: [10.1145/65000.65001](http://doi.acm.org/10.1145/65000.65001). url: <http://doi.acm.org/10.1145/65000.65001> (cit. on p. 25).
- [33] Rivka Ladin et al. “Providing High Availability Using Lazy Replication”. In: *ACM Trans. Comput. Syst.* 10.4 (Nov. 1992), pp. 360–391. issn: 0734-2071. doi: [10.1145/138873.138877](http://doi.acm.org/10.1145/138873.138877). url: <http://doi.acm.org/10.1145/138873.138877> (cit. on p. 25).
- [34] André Schiper, Jorge Egli, and Alain Sandoz. “A New Algorithm to Implement Causal Ordering”. In: *Proceedings of the 3rd International Workshop on Distributed Algorithms*. London, UK, UK: Springer-Verlag, 1989, pp. 219–232. isbn: 3-540-51687-5. url: <http://dl.acm.org/citation.cfm?id=645946.675010> (cit. on pp. 26, 39).
- [35] Michel Raynal, André Schiper, and Sam Toueg. “The Causal Ordering Abstraction and a Simple Way to Implement It”. In: *Inf. Process. Lett.* 39.6 (Oct. 1991), pp. 343–350. issn: 0020-0190. doi: [10.1016/0020-0190\(91\)90008-6](http://dx.doi.org/10.1016/0020-0190(91)90008-6). url: [http://dx.doi.org/10.1016/0020-0190\(91\)90008-6](http://dx.doi.org/10.1016/0020-0190(91)90008-6) (cit. on p. 26).
- [36] Yair Amir et al. “Transis: A Communication Subsystem for High Availability”. In: *Proceedings of the 22nd International Symposium on Fault Tolerant Computing*. Boston, Massachusetts: IEEE Computer Society Press, 1992, pp. 76–84 (cit. on p. 26).
- [37] Yair Amir and Jonathan Stanton. “The Spread Wide Area Group Communication System”. In: 2007 (cit. on p. 26).
- [38] JGroups. *a toolkit for reliable multicast communication*. 2002. url: <http://www.jgroups.org> (cit. on p. 26).

- [39] Michael Stonebraker University and Michael Stonebraker. "The Case for Shared Nothing". In: *Database Engineering* 9 (1986), pp. 4–9 (cit. on p. 26).
- [40] José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero. "Bounded Version Vectors". In: *Distributed Computing*. Ed. by Rachid Guerraoui. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 102–116. isbn: 978-3-540-30186-8 (cit. on p. 26).
- [41] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. "Interval Tree Clocks". In: *Proceedings of the 12th International Conference on Principles of Distributed Systems*. OPODIS '08. Luxor, Egypt: Springer-Verlag, 2008, pp. 259–274. isbn: 9783540922209. doi: [10.1007/978-3-540-92221-6_18](https://doi.org/10.1007/978-3-540-92221-6_18). url: https://doi.org/10.1007/978-3-540-92221-6_18 (cit. on p. 27).
- [42] Mukesh Singhal and Ajay Kshemkalyani. "An Efficient Implementation of Vector Clocks". In: *Inf. Process. Lett.* 43.1 (Aug. 1992), pp. 47–52. issn: 0020-0190. doi: [10.1016/0020-0190\(92\)90028-T](https://doi.org/10.1016/0020-0190(92)90028-T). url: [http://dx.doi.org/10.1016/0020-0190\(92\)90028-T](http://dx.doi.org/10.1016/0020-0190(92)90028-T) (cit. on pp. 27, 28).
- [43] Punit Chandra, Pranav Gambhire, and Ajay Kshemkalyani. "Performance of the Optimal Causal Multicast Algorithm: A Statistical Analysis". In: *Parallel and Distributed Systems, IEEE Transactions on* 15 (Feb. 2004), pp. 40–52. doi: [10.1109/TPDS.2004.1264784](https://doi.org/10.1109/TPDS.2004.1264784) (cit. on p. 28).
- [44] Pat Stephenson and Kenneth Birman. "Fast Causal Multicast". In: *SIGOPS Oper. Syst. Rev.* 25.2 (Apr. 1991), pp. 75–79. issn: 0163-5980. doi: [10.1145/122120.122127](https://doi.org/10.1145/122120.122127). url: <http://doi.acm.org/10.1145/122120.122127> (cit. on p. 28).
- [45] Ravi Prakash, Michel Raynal, and Mukesh Singhal. "An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments". In: *J. Parallel Distrib. Comput.* 41.2 (Mar. 1997), pp. 190–204. issn: 0743-7315. doi: [10.1006/jpdc.1996.1300](https://doi.org/10.1006/jpdc.1996.1300). url: <http://dx.doi.org/10.1006/jpdc.1996.1300> (cit. on pp. 28, 55).
- [46] Ajay D. Kshemkalyani and Mukesh Singhal. "Necessary and Sufficient Conditions on Information for Causal Message Ordering and Their Optimal Implementation". In: *Distrib. Comput.* 11.2 (Apr. 1998), pp. 91–111. issn: 0178-2770. doi: [10.1007/s004460050044](https://doi.org/10.1007/s004460050044). url: <http://dx.doi.org/10.1007/s004460050044> (cit. on pp. 28, 55).
- [47] Saul Pomares Hernandez et al. "Causal Broadcast Protocol for Very Large Group Communication Systems." In: Jan. 2001, pp. 175–188 (cit. on pp. 28, 29).
- [48] Hein Meling et al. "Jgroup/ARM: a distributed object group platform with autonomous replication management". In: *Software: Practice and Experience* 38.9 (2008), pp. 885–923. doi: [10.1002/spe.853](https://doi.org/10.1002/spe.853). eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.853>. url: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.853> (cit. on p. 28).

- [49] Antonio Fernández, Ernesto Jiménez, and Vicent Cholvi. “On the Interconnection of Causal Memory Systems”. In: *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '00. Portland, Oregon, USA: Association for Computing Machinery, 2000, pp. 163–170. isbn: 1581131836. doi: [10.1145/343477.343540](https://doi.org/10.1145/343477.343540). url: <https://doi.org/10.1145/343477.343540> (cit. on p. 28).
- [50] A. Álvarez et al. “On the interconnection of message passing systems”. In: *Information Processing Letters* 105.6 (2008), pp. 249–254. issn: 0020-0190. doi: <https://doi.org/10.1016/j.ipl.2007.09.006>. url: <https://www.sciencedirect.com/science/article/pii/S0020019007002700> (cit. on p. 28).
- [51] L. E. T. Rodrigues and P. Verissimo. “Causal Separators for Large-Scale Multicast Communication”. In: *Proceedings of the 15th International Conference on Distributed Computing Systems*. ICDCS '95. USA: IEEE Computer Society, 1995, p. 83 (cit. on p. 28).
- [52] Luis Rodrigues and Paulo Verissimo. *Causal Separators and Topological Timestamping: An Approach to Support Causal Multicast in Large-Scale Systems*. Tech. rep. 1995 (cit. on p. 28).
- [53] Roberto Baldoni, Roy Friedman, and Robbert van Renesse. “The Hierarchical Daisy Architecture for Causal Delivery”. In: *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS '97)*. ICDCS '97. USA: IEEE Computer Society, 1997, p. 570. isbn: 0818678135 (cit. on p. 28).
- [54] S. Johnson, F. Jahanian, and J. Shah. “The inter-group router approach to scalable group composition”. In: *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*. 1999, pp. 4–14. doi: [10.1109/ICDCS.1999.776500](https://doi.org/10.1109/ICDCS.1999.776500) (cit. on p. 28).
- [55] A. Mostefaoui et al. “From static distributed systems to dynamic systems”. In: *24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)*. 2005, pp. 109–118. doi: [10.1109/RELDIS.2005.19](https://doi.org/10.1109/RELDIS.2005.19) (cit. on p. 29).
- [56] R. Baldoni et al. *Broadcast with Time and Causality Constraints for Multimedia Applications*. Tech. rep. Proc. of the 22nd. EUROMICRO Conference (cit. on p. 29).
- [57] Abderrahim Benslimane and Abdelhafid Abouaissa. “Dynamical Grouping Model for Distributed Real Time Causal Ordering”. In: *Comput. Commun.* 25.3 (Feb. 2002), pp. 288–302. issn: 0140-3664. doi: [10.1016/S0140-3664\(01\)00361-9](https://doi.org/10.1016/S0140-3664(01)00361-9). url: [https://doi.org/10.1016/S0140-3664\(01\)00361-9](https://doi.org/10.1016/S0140-3664(01)00361-9) (cit. on p. 29).
- [58] Raj Yavatkar and K Lakshman. “Communication support for distributed collaborative applications”. In: *Multimedia Systems* 2.2 (1994), pp. 74–88. issn: 1432-1882. doi: [10.1007/BF01274182](https://doi.org/10.1007/BF01274182). url: <https://doi.org/10.1007/BF01274182> (cit. on p. 29).

- [59] M.H. Kalantar and K.P. Birman. “Causally ordered multicast: the conservative approach”. In: *Proceedings. 19th IEEE International Conference on Distributed Computing Systems (Cat. No.99CB37003)*. 1999, pp. 36–44. doi: [10.1109/ICDCS.1999.776504](https://doi.org/10.1109/ICDCS.1999.776504) (cit. on p. 29).
- [60] Wyatt Lloyd et al. “Stronger Semantics for Low-Latency Geo-Replicated Storage”. In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*. nsdi’13. Lombard, IL: USENIX Association, 2013, pp. 313–328 (cit. on p. 31).
- [61] Jiaqing Du et al. “Orbe: Scalable Causal Consistency Using Dependency Matrices and Physical Clocks”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing*. SOCC ’13. Santa Clara, California: Association for Computing Machinery, 2013. isbn: 9781450324281. doi: [10.1145/2523616.2523628](https://doi.org/10.1145/2523616.2523628). url: <https://doi.org/10.1145/2523616.2523628> (cit. on p. 31).
- [62] Jiaqing Du et al. “GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks”. In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC ’14. Seattle, WA, USA: Association for Computing Machinery, 2014, pp. 1–13. isbn: 9781450332521. doi: [10.1145/2670979.2670983](https://doi.org/10.1145/2670979.2670983). url: <https://doi.org/10.1145/2670979.2670983> (cit. on p. 32).
- [63] Deepthi Devaki Akkoorath et al. “Cure: Strong Semantics Meets High Availability and Low Latency”. In: *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*. 2016, pp. 405–414. doi: [10.1109/ICDCS.2016.98](https://doi.org/10.1109/ICDCS.2016.98) (cit. on p. 33).
- [64] Sérgio Almeida, João Leitão, and Luís Rodrigues. “ChainReaction: A Causal+ Consistent Datastore Based on Chain Replication”. In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys ’13. Prague, Czech Republic: Association for Computing Machinery, 2013, pp. 85–98. isbn: 9781450319942. doi: [10.1145/2465351.2465361](https://doi.org/10.1145/2465351.2465361). url: <https://doi.org/10.1145/2465351.2465361> (cit. on p. 33).
- [65] Robbert van Renesse and Fred B. Schneider. “Chain Replication for Supporting High Throughput and Availability”. In: *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*. OSDI’04. San Francisco, CA: USENIX Association, 2004, p. 7 (cit. on p. 33).
- [66] Manuel Bravo, Luís Rodrigues, and Peter Van Roy. “Saturn: A Distributed Metadata Service for Causal Consistency”. In: *Proceedings of the Twelfth European Conference on Computer Systems*. EuroSys ’17. Belgrade, Serbia: ACM, 2017, pp. 111–126. isbn: 978-1-4503-4938-3. doi: [10.1145/3064176.3064210](https://doi.org/10.1145/3064176.3064210). url: <http://doi.acm.org/10.1145/3064176.3064210> (cit. on p. 34).
- [67] Syed Akbar Mehdi et al. “I Can’t Believe It’s Not Causal! Scalable Causal Consistency with No Slowdown Cascades”. In: *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*. NSDI’17. Boston, MA, USA: USENIX Association, 2017, pp. 453–468. isbn: 9781931971379 (cit. on p. 35).

- [68] Robert Escriva et al. “Kronos: The Design and Implementation of an Event Ordering Service”. In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. Amsterdam, The Netherlands: Association for Computing Machinery, 2014. isbn: 9781450327046. doi: [10.1145/2592798.2592822](https://doi.org/10.1145/2592798.2592822). url: <https://doi.org/10.1145/2592798.2592822> (cit. on p. 36).
- [69] Mustaque Ahamad et al. “Causal memory: Definitions, implementation and programming”. In: *IEEE Transactions on Parallel and Distributed Systems* 1 (1990), pp. 6–16 (cit. on pp. 37, 40).
- [70] Prince Mahajan, Lorenzo Alvisi, Mike Dahlin, et al. “Consistency, availability, and convergence”. In: *University of Texas at Austin Tech Report* 11 (2011) (cit. on p. 37).
- [71] Douglas B. Terry et al. “Session Guarantees for Weakly Consistent Replicated Data”. In: *Proceedings of the Third International Conference on Parallel and Distributed Information Systems*. PDIS '94. Washington, DC, USA: IEEE Computer Society, 1994, pp. 140–149. isbn: 0-8186-6400-2. url: <http://dl.acm.org/citation.cfm?id=645792.668302> (cit. on p. 37).
- [72] David R Cheriton and Dale Skeen. *Understanding the limitations of causally and totally ordered communication*. Vol. 27. 5. ACM, 1994 (cit. on pp. 38, 40, 47).
- [73] Marc Shapiro et al. “Conflict-free Replicated Data Types”. In: *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*. SSS'11. Grenoble, France: Springer-Verlag, 2011, pp. 386–400. isbn: 978-3-642-24549-7. url: <http://dl.acm.org/citation.cfm?id=2050613.2050642> (cit. on pp. 38, 41, 42).
- [74] J. H. Saltzer, D. P. Reed, and D. D. Clark. “End-to-end Arguments in System Design”. In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984), pp. 277–288. issn: 0734-2071. doi: [10.1145/357401.357402](https://doi.org/10.1145/357401.357402). url: <http://doi.acm.org/10.1145/357401.357402> (cit. on p. 41).
- [75] Microsoft Azure. *Multi-master at global scale with Azure Cosmos DB*. 2018. url: <https://docs.microsoft.com/en-us/azure/cosmos-db/multi-region-writers> (cit. on p. 42).
- [76] Redis Labs. *Under the Hood: Redis CRDTs*. 2017? url: <https://redislabs.com/docs/active-active-whitepaper/> (cit. on pp. 42, 113, 114).
- [77] Basho. *Riak KV Concepts: Data Types*. url: <http://docs.basho.com/riak/kv/2.2.3/learn/concepts/crdts/> (cit. on p. 42).
- [78] Atom Editor. *Code together in real time with Teletype for Atom*. 2017. url: <http://blog.atom.io/2017/11/15/code-together-in-real-time-with-teletype-for-atom.html> (cit. on p. 42).
- [79] Protocol Labs. *Decentralized Real-Time Collaborative Documents - Conflict-free editing in the browser using js-ipfs and CRDTs*. 2017. url: <https://ipfs.io/blog/30-js-ipfs-crdts.md> (cit. on p. 42).

- [80] Edsger W. Dijkstra. “Self-stabilizing Systems in Spite of Distributed Control”. In: *Commun. ACM* 17.11 (Nov. 1974), pp. 643–644. issn: 0001-0782. doi: [10 . 1145 / 361179 . 361202](https://doi.org/10.1145/361179.361202). url: <http://doi.acm.org/10.1145/361179.361202> (cit. on p. 62).
- [81] *The Poisson Distribution and Poisson Process Explained*. <https://towardsdatascience.com/the-poisson-distribution-and-poisson-process-explained-4e2cb17d459> (cit. on p. 93).
- [82] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. USA: Addison Wesley Longman Publishing Co., Inc., 1997. isbn: 0201896834 (cit. on p. 94).
- [83] Jose A. Hernandez Gutierrez and Iain Phillips. *Weibull mixture model to characterise end-to-end Internet delay at coarse time-scales*. Jan. 2006. url: <https://hdl.handle.net/2134/2121> (cit. on p. 94).
- [84] T. Holleczeck, V. Venus, and S. Naegele-Jackson. “Statistical Analysis of IP Delay Measurements as a Basis for Network Alert Systems”. In: *2009 IEEE International Conference on Communications* (2009). doi: [10.1109/icc.2009.5199487](https://doi.org/10.1109/icc.2009.5199487) (cit. on p. 94).
- [85] 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. doi: [10.1016/j.jpdc.2015.02.003](https://doi.org/10.1016/j.jpdc.2015.02.003) (cit. on p. 94).
- [86] *Weibull Distribution*. https://en.wikipedia.org/wiki/Weibull_distribution (cit. on p. 94).
- [87] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2 (cit. on p. 97).
- [88] *Kubernetes Manual*. [pleaseputtheurl](https://kubernetes.io/docs/). [Online; accessed 04-Dec-2017]. 2017 (cit. on p. 97).
- [89] Brian White et al. “An Integrated Experimental Environment for Distributed Systems and Networks”. In: Boston, MA, Dec. 2002, pp. 255–270 (cit. on p. 97).
- [90] Paul A. S. Ward. “Algorithms for Causal Message Ordering in Distributed Systems”. In: 2007 (cit. on pp. 97, 98).
- [91] Houssam Yactine, Ali Shoker, and Georges Younes. “ASPAS: As Secure as Possible Available Systems”. In: *Distributed Applications and Interoperable Systems*. Ed. by Miguel Matos and Fabiola Greve. Cham: Springer International Publishing, 2021, pp. 57–73. isbn: 978-3-030-78198-9 (cit. on p. 109).
- [92] Georges Younes, Paulo Sérgio Almeida, and Carlos Baquero. “Compact Resettable Counters through Causal Stability”. In: *Proceedings of the 3rd International Workshop on Principles and Practice of Consistency for Distributed Data*. PaPoC '17. Belgrade, Serbia: Association for Computing Machinery, 2017. isbn: 9781450349338. doi: [10 . 1145 / 3064889 . 3064892](https://doi.org/10.1145/3064889.3064892). url: <https://doi.org/10.1145/3064889.3064892> (cit. on pp. 111, 113).

- [93] Christopher Meiklejohn and Peter Van Roy. “Lasp: A Language for Distributed, Coordination-Free Programming”. In: *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*. PPDP '15. Siena, Italy: Association for Computing Machinery, 2015, pp. 184–195. isbn: 9781450335164. doi: [10.1145/2790449.2790525](https://doi.org/10.1145/2790449.2790525). url: <https://doi.org/10.1145/2790449.2790525> (cit. on p. 111).
- [94] *Large-scale computation without synchronisation*. <https://syncfree.lip6.fr> (cit. on p. 113).
- [95] *Antidotedb*. <http://syncfree.github.io/antidote/> (cit. on p. 113).
- [96] Ali Shoker et al. “Lightkone: Towards general purpose computations on the edge”. In: *White Paper published on http://www.lightkone.eu* 40 (2016) (cit. on p. 113).
- [97] Georges Younes et al. “Integration Challenges of Pure Operation-Based CRDTs in Redis”. In: *First Workshop on Programming Models and Languages for Distributed Computing*. PMLDC '16. Rome, Italy: Association for Computing Machinery, 2016. isbn: 9781450347754. doi: [10.1145/2957319.2957375](https://doi.org/10.1145/2957319.2957375). url: <https://doi.org/10.1145/2957319.2957375> (cit. on p. 115).
- [98] *Minidote*. <https://github.com/LightKone/Minidote> (cit. on p. 115).