Causal Broadcast algorithms for dynamic distributed systems

Daniel Wilhelm

Thèse présentée pour l'obtention du grade de

Doctorat de Sorbonne Université

Discipline: Informatique

Laboratoire d'Informatique de Paris 6

École Doctorale Informatique, Télécommunications et Électronique

Causal Broadcast algorithms for dynamic distributed systems

Par : Daniel Wilhelm

Sous la direction de:
Pierre Sens — Professeur, Sorbonne Université, LIP6, France
Et encadrée par:
Luciana Arantes — Maîtresse de conférence, Sorbonne Université, LIP6, France

Rapporteurs:
Achour Mostéfaoui — Professeur, Université de Nantes, LS2N, France
Matthieu Roy — Chargé de recherche, HDR, CNRS, LAAS, France
Examinateurs:
Colette Johnen — Professeur, Université de Bordeaux, LaBRI, France
Maria Potop-Butucaru — Professeur, Sorbonne Université, LIP6, France

Président du jury:
Maria Potop-Butucaru — Professeur, Sorbonne Université, LIP6, France

2023

Acknowledgements

Je voudrais remercier mes directeurs et directrices de thèse pour leur soutien inconditionnel, leur expertise et leur engagement tout au long de cette aventure de recherche. Leur accompagnement a été essentiel pour la réussite de ce projet.

Je tiens à remercier également les membres de mon jury de thèse pour le temps et l'attention qu'ils m'ont accordé. Votre lecture attentive de mon manuscrit de thèse et vos commentaires constructifs ont été d'une grande aide dans l'amélioration de mon travail.

Je suis également reconnaissant(e) envers mes collègues et mes amis qui m'ont soutenu(e) tout au long de ce parcours. Vos encouragements et votre soutien ont été des facteurs clés de ma réussite.

Enfin, je tiens à exprimer ma gratitude envers ma famille pour leur amour, leur soutien et leur compréhension. Sans leur soutien, je n'aurais pas pu arriver jusqu'ici.

Merci encore à tous pour votre soutien et votre contribution à mon travail de recherche.

Abstract

Causal broadcast is a fundamental building block of many distributed or parallel applications such as distributed databases, pub-sub, or social networks, which all need to share information among all participants (e.g., processes, machines, etc. . .) while respecting the causality among exchanged messages.

Existing causal broadcast algorithms for distributed systems either do not scale, or they do not tolerate the system dynamics caused by processes which, during execution, join and leave the system, fail, or change their set of communication channels. Some works append to messages all the information required to causally order them at destination. However, it has been proved that to track causality when broadcasting messages, the minimal required structure has one entry per process in the system. Hence, causal broadcast algorithms using such a structure to track causality do not scale. Some other works scale by making assumptions on the system, such as the network topology or the FIFO property of communication channels. On the other hand, they do not tolerate all possible dynamics of distributed systems.

In this thesis, we propose causal broadcast algorithms that both scale and tolerate the dynamics of distributed systems.

The first proposed algorithm provides a causal broadcast for Mobile Networks, composed of mobile hosts and support stations. Mobile hosts are connected to support stations through a wireless network, and support stations are connected with each other by wired channels. Mobile networks have specific features: limited capacities of mobile nodes (computation, storage, energy), unreliable communication channels, and the dynamics of connections due to the mobility of node, failures, and join/leave operations.

The second part of this thesis addresses causal broadcast implemented with clocks of M ($M \leq N$) entries, where N corresponds to the number of processes in the system. Such clocks tolerate process churn and scale since their size is independent to the number of processes in the system. However, they do not characterize causality and algorithms using them may deliver messages out of causal order.

We first propose an error detector, based on hashes, which analyzes the clock of messages before delivering them in order to detect those messages that are not causally ordered. We then propose an algorithm that ensures that messages tagged by the error detector as not causally ordered are delivered in causal order. Second, we propose *Dynamic Clock Sets* (*DCS* clocks), a new logical clock composed of a set of clocks with $M \leq N$ entries (where N is the number of process of the system), and whose size can be changed during execution.

All the proposed algorithms were implemented in C++ and executed on the OMNeT++ simulator. The causal broadcast algorithm for Mobile Networks was implemented on the framework INET of OMNeT++ which provides a realistic network simulation such as interferences on the wireless network, network layers and node mobility among others. Results confirm that our broadcast algorithm for Mobile Networks outperforms existing ones while making realistic network assumptions. Concerning the hash-based error detector, results show that it detects all not causally ordered messages. Finally, experiments conducted with a causal broadcast implemented with *DCS* clocks demonstrate that *DCS* clocks adapt their size well and fast to the number of concurrent messages of the system.

Résumé

La diffusion causale est un élément fondamental de nombreux applications distribuées ou parallèles, telles que les bases de données distribuées, les publications-abonnements, ou les réseaux sociaux, qui ont tous besoin de partager des informations entre tous les participants (par exemple, les processus, machines, etc. . .), tout en respectant la causalité entre les messages échangés.

Les algorithmes de diffusion causale existants ne passent soit pas à l'échelle, soit ne tolèrent pas toute la dynamicité introduite par les processus qui rejoignent ou quittent le système, qui échouent pendant l'exécution, ou qui modifient leur ensemble de canaux de communication. Certains travaux ajoutent aux messages toutes les informations nécessaires pour les ordonner causalement à la réception. Cependant, il a été prouvé que caractériser la causalité de messages de diffusion nécessite une structure avec au minimum une entrée par processus dans le système. Par conséquent, les algorithmes qui utilisent une telle structure pour caractériser la causalité ne passent pas à l'échelle. D'autres travaux passent à l'échelle en faisant des hypothèses sur le système, comme la topologie du réseau ou la propriété FIFO des canaux de communication. En revanche, ils ne tolèrent pas toutes les dynamiques possibles des systèmes distribués.

Dans cette thèse, nous proposons des algorithmes de diffusion causale qui passent à l'échelle et tolèrent les dynamiques des systèmes distribués.

Le premier algorithme fournit une diffusion causale pour les réseaux mobiles, composés de nœuds mobiles connectés par un réseau sans fil et de stations de support connectées par des canaux filaires. Ces réseaux ont des caractéristiques spécifiques: des nœuds mobiles avec des capacités limitées (calcul, stockage, énergie), des canaux de communication non fiables, et de la dynamicité de connexions dues à la mobilité et aux pannes de nœuds, ainsi qu'à leur opérations d'entrée et sortie du système.

La deuxième partie de la thèse aborde la diffusion causale mise en œuvre avec des horloges de M ($M \leq N$) entrées, où N correspond au nombre de processus. Ces horloges tolèrent la variation du nombre de processus dans le système car leur taille ne dépend pas du nombre de processus dans le système. Cependant, elles ne caractérisent pas la causalité et les algorithmes les utilisant ne peuvent assurer l'ordre causal qu'avec une forte probabilité. Nous proposons tout d'abord un détecteur d'erreurs, basé sur des hachages, qui analyse l'horloge des messages avant de les délivrer afin de détecter les messages qui ne sont pas causalement

ordonnés. Nous proposons ensuite un algorithme qui garantit que les messages marqués par le détecteur d'erreurs sont livrés dans l'ordre causal. Ensuite, nous proposons les *Dynamic Clock Sets* (*DCS* clocks), une nouvelle horloge logique qui est composée d'un ensemble d'horloges avec $M \leq N$ entrées (avec N étant le nombre de processus du système), et dont la taille peut être adaptée au nombre de messages concurrents dans le système.

Tous les algorithmes ci-dessus ont été implémentés en C++ et exécutés sur le simulateur OMNeT++. L'algorithme de diffusion causale pour les réseaux mobiles a été implémenté sur le framework INET d'OMNeT++, qui fournit une simulation de réseau réaliste, telle que des interférences sur le réseau sans fil, des couches réseau et la mobilité des nœuds, entre autres. Les résultats confirment que notre algorithme de diffusion pour les réseaux mobiles est plus performant que les algorithmes existants tout en faisant des hypothèses moins forte et réalistes sur le réseau. Les résultats expérimentaux sur le détecteur d'erreurs basé sur les hachages montrent qu'il détecte tous les messages dont les dépendances causales n'ont pas encore été livrées. Enfin, les expériences menées avec une diffusion causale implémentée avec les *DCS* clocks montrent que les *DCS* clocks s'adaptent bien et rapidement au nombre de messages concurrents dans le système.

Table of contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

Distributed systems are composed of many processes that cooperate to provide a service. To that end, they share information, either through message passing or by using shared memory. In the message passing model, processes exchange messages by using the *send* and *receive* primitives. Many distributed systems provide a group communication service by using these two primitives to implement the primitives *broadcast* and *deliver*, which respectively sends a message to all processes of the system and delivers a message that have been previously broadcasted.

Many distributed and parallel applications also require that messages are delivered following a given order to ensure the consistency of information. Hence, events - in this case the broadcast and delivery of messages - must be ordered even if taking place at different processes. Humans usually use physical time to order events. However, physical clocks usually cannot be used to order events in distributed systems, because they are subject to clock drift [KO87][PR94], meaning that physical clocks of different processes that are initially synchronized might eventually de-synchronize.

Lamport introduced in 1978 the concept of logical time [Lam78], which has been since then used to timestamp events in distributed systems. Several ordering approaches (e.g. causal, FIFO, total order) have been proposed in the literature to

order events following logical time. Among them, Causal Order is a partial order that allows to track causality among events. It is defined by Lamport's *happened before* relationship [Lam78]: for any two operations e_1 and e_2, e_1 is said to causally precede e_2, which is denoted as $e_1 \rightarrow e_2$, if (1) e_1 and e_2 occur on the same process and e_1 occurs before e_2, (2) $e_1 = send(m)$ and $e_2 = receive(m)$ or (3) $\exists\ e_3$ such that $e_1 \rightarrow e_3$ and $e_3 \rightarrow e_2$ (transitivity). Two events e_1 and e_2 are said to be concurrent when no causal relation exists between them ($e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$).

Causal broadcast ensures that broadcast messages are causally ordered, i.e., for any two messages m_1 and m_2 if the broadcast of m_1 causally precedes the broadcast of m_2, then all processes deliver m_1 before m_2. Introduced by Birman et al. [Bir85] in 1985, causal broadcast has been extensively investigated. Innumerable applications use it, such as publish-subscriber systems [de +17][LSB06], multimedia applications [Bal+96][Ple+06], online games [GE12], systems that provide distributed replicated causal data consistency [AHJ91], distributed databases [Ter+95][Sha+11], social networks [Bor13], distributed collaborative editing [Hei+12][NMM16], among others.

Many causal broadcast approaches have been proposed in the literature. They either piggyback information on messages in order to causally order them at reception [SES89][BSS91][KKS18][PRS96][AN96][TA99][MW17b], or they make assumptions on the system topology and communication channels to implicitly order messages [FM04][NMM18a][BCD17], thus ensuring that they are received by processes already causally ordered, and that they can therefore be causally delivered upon reception without any control. Charron-Bost proved in [Cha91] that in a distributed system without assumptions on the network topology and communication channels, the causality of broadcast events can only be characterized by a structure with one entry per process in the system. Hence, in a system with N processes, causality of broadcasted messages can only be characterized with a structure whose size is in $\mathcal{O}(N)$. Consequently, solutions that attach causal information on messages often do not scale with the number of processes. To circumvent this limitation, some works propose structures whose size is independent to the number of processes [TA99][MW17b], but algorithms using them might deliver messages out of causal order.

Several distributed systems are also dynamic (e.g. wireless or P2P networks) and are subject to failures, either of processes or communication channels. Processes might join and leave the system during execution making it difficult to maintain a system view. Processes can also be subject to transient failures, whose duration is bounded in time, as well as permanent failures, also said crash-failures, which last forever. Finally, communication channels over which processes communicate can also fail, or be unreliable, i.e., messages can be lost, corrupted, created, or altered.

Approaches that append causal information of size $\mathcal{O}(N)$, where N corresponds to the number of processes in the system, do not scale. Approaches that organize the network into an overlay usually make assumptions on the network topology, which are often incompatible with the dynamics that characterize many distributed systems. They are thus not suited to such systems, or require extra handling to adapt them to such dynamics.

Therefore, the conception of causal broadcast algorithms require a trade-off between network assumptions, the size of causal information required to causally order messages, as well as other metrics such as delivery delays. Existing algorithms often do not tolerate the dynamics of distributed systems or the failure of processes neither of communication channels, and they often make unrealistic assumptions about communication channels (e.g., FIFO, reliability).

1.1 Contributions

The aim of this thesis is to propose new causal broadcast algorithms for dynamic distributed systems. Nowadays, such systems are composed of an ever-growing number of processes. Hence, we look for algorithms that scale well with the number of processes of the system. Moreover, the algorithms should tolerate the dynamics of distributed systems, by allowing topology changes, mobility of processes, as well as processes that join and leave the system during execution.

The first contribution of the thesis focuses on causal broadcast for mobile networks. Such networks are highly dynamic: processes can move, leave or join the system, and fail. Moreover, the wireless network over which they communicate is subject to interferences and is therefore unreliable. Finally, processes present many constraints that must be taken into account, such as limited battery, computation, or memory capacities. The proposed algorithm tolerates the dynamics of mobile networks, takes into account the constraints of processes, and is scalable since it uses only a few causal information to track the causality of messages.

The second contribution of this thesis focuses on causal broadcast based on clocks with $M \leq N$ entries, where N corresponds to the number of processes of the system. Such clocks scale well and tolerate process churn, but algorithms using them might deliver messages out of causal order. The proposition consists of two proposals that aim to increase the probability that processes deliver messages in causal order. We first propose an error detector which analyzes the clock value appended on broadcast messages before delivering them, in order to detect out of causal order deliveries. We then present an algorithm, that used in conjunction with error detectors, ensures that messages tagged as not causally ordered are

delivered in causal order. The second proposal proposes the *DCS* clocks, a new clock based on clocks of M ($M \leq N$) entries, and whose size can be adapted dynamically, which is particularly interesting in systems with a varying number concurrent messages.

1.1.1 A causal broadcast algorithms that tolerates the dynamics of Mobile Networks

Mobile networks are mainly composed of Mobile Support Stations, connected between each other through wired channels, and Mobile hosts, connected to stations through the wireless network. A causal broadcast algorithm for Mobile Networks should tolerate the many constraints inherent to mobile hosts and the wireless network, and it should also scale to handle a high number of Mobile Hosts as well as Mobile Support Stations. This contribution proposes a causal broadcast algorithm for Mobile Networks that is based on FIFO dissemination, which, by forwarding messages in FIFO order, ensures that there exists no path between two machines where messages travel out of causal order.

The algorithm is scalable and is designed for mobile networks. Hosts can join/leave the network and fail, permanently or transiently, at any time. They move freely and can be temporarily disconnected from the network when out of range of any station. We assume no reliable connection protocol, and the algorithm handles multiple concurrent connections by the same host. Resource limitations of hosts (computational and memory power, battery life) are handled by keeping causal information at stations, while hosts only keep very little control information. Messages piggyback only a few integers as control information. Obsolete messages cached by stations are discarded following a decentralized approach: a station discards a message once all hosts connected to it acknowledged the message. Consequently, stations only cache necessary messages. Furthermore, a high message traffic within a wireless cell, which leads to delivery delays as well as storage and communication overheads, only has a local impact.

We also present a second causal broadcast algorithm which extends the first one to tolerate the failure of stations. The algorithm gathers stations into groups, and the causal information stored at a station is replicated at the other stations of the group.

Both algorithms have been implemented on a realistic simulator, the INET framework of the simulator OMNeT++. Our evaluations show that both algorithms are scalable in both the number of hosts and stations, have a low message traffic and storage overhead, while handling mobile network dynamics without the constraining assumptions of existing causal multicast approaches [CK04][BB08].

1.1.2 Enhancing the accuracy of clocks with $M \leq N$ entries

Clocks with M entries [TA99][MW17b][GP03][MK21] are usually much smaller than N, the number of processes in the system. Algorithms using such clocks scale well with the number of processes of the system since their size is independent of the number of processes. However, they do not characterize causality. Hence, algorithms that use them to causally order messages might deliver messages out of causal order, even though messages are usually delivered in causal order with a high probability.

The second contribution of this thesis works on M-entry clocks in order to reduce the number of messages that processes deliver out of causal order when implementing causal broadcast using them. It is divided in two proposals:

Error detectors We first determine the conditions required to provide a reliable error detector which detects all out of causal order deliveries. We show that these conditions are not realistic and that such an error detector can thus not be implemented under realistic assumptions.

Second, we propose an error detector. Processes call it before delivering a message m. It analyzes the clock of m to determine if m can be delivered in causal order. The proposed error detector is based on hashes: a process hashes the causal dependencies of a message m it wants to broadcast and appends this hash on m when broadcasting m. Processes that receive m then try to determine whether they delivered all of m's causal dependencies by building dependency sets and computing their hash. A theoretical analysis shows that the hash-based error detector misses out of causal deliveries only in case of hash collisions, which has a low probability to occur. During experiments, the proposed error detector missed no out of causal order delivery.

The error detector tags messages as causally ordered or not, but it does not inform which are the causal dependencies that have not been delivered locally. Hence, we propose an algorithm to identify missing causal dependencies. It has to be used in conjunction with an error detector. Our algorithm ensures that messages tagged as not causally ordered by the error detector are delivered in causal order.

Logical clock composed of a set of clocks with $M \leq N$ entries We propose a new clock, denoted Dynamic Clock Set (DCS), which consists of a set of probabilistic clocks. The conception of the DCS clock results from the observation that the size M of an M-entry clock should be chosen based on the number of concurrent messages inside the system, because this metric determines the efficiency of clocks of size $M \leq N$ to causally order messages. However, existing clocks of size

$M \leq N$ require fixing M at initialization, and M cannot change during execution. A wrong choice of M may imply an oversized clock or in many messages delivered out of causal order. The size of *DCS* clocks can dynamically vary during execution, by adding or removing clocks of size M. We provide the operations required to modify the size of the clock as well as to compare them. We also propose an implementation of a causal broadcast algorithm using *DCS* clocks. Experimental results show that it achieves a higher accuracy in delivering messages in causal order than those using existing clocks of size $M \leq N$ and depending on the message load, also present a lower message overhead.

1.2 Publications

The following articles were published during the thesis.

Work on causal broadcast in Mobile Networks:
A scalable causal broadcast that tolerates dynamics of mobile networks. ICDCN 2022: 9-18, Daniel Wilhelm, Luciana Arantes, Pierre Sens

Work on M-entry clocks:
Improving accuracy of probabilistic-based causal broadcast. COMPAS 2022, Daniel Wilhelm, Luciana Arantes and Pierre Sens
A probabilistic Dynamic Clock Set to capture message causality. ALGOTEL 2023, Daniel Wilhelm, Luciana Arantes and Pierre Sens

1.3 Organization of the Manuscript

The rest of the thesis is organized as follows.

Chapter 2 presents some concepts and definitions relevant to this thesis, in particular, those related to distributed systems, time and causal order in distributed systems, the broadcast delivery primitives, and Mobile Networks.

Chapter 3 gives a summary of the related work relevant to this thesis. It comprises existing causal broadcast algorithms for distributed systems as well as the causal multicast algorithms (which can easily be adapted to causal broadcast) for Mobile Networks.

Chapter 4 presents two causal broadcast algorithms for Mobile Networks, based on the FIFO-dissemination approach. Both algorithms scale well and are tailored to the features of Mobile Networks.

Chapter 5 gathers the two proposals related to clocks of size $M \leq N$ where N corresponds to the number of processes in the system. We first propose an error detector based on hashes and which aims to detect out of causal order deliveries. Then, we propose an algorithm that ensures the causal delivery of messages tagged by the error detector. The second proposal consists of *DCS* clocks, a new clock based on probabilistic clocks, and whose size can vary dynamically.

Chapter 6 concludes the thesis and proposes future research directions.

Chapter 2

Background

Contents

2.1 Introduction

This chapter presents the concepts relevant to this thesis. Distributed systems are composed of processes and communication channels, which can both fail. Processes cooperate in order to execute some common task. To that end, they often need a common time measure, which they achieve by using logical time, introduced by Lamport in 1978. Distributed events must also often be ordered. Causal order, defined by the happened-before relation, is one of such orders. To share information with all processes of the system, processes use the broadcast primitive. Causal broadcast consists of broadcast messages ordered following causal order. It is used in many systems, such as Mobile Networks, which are composed of mobile hosts and mobile support stations, and which have specific constraints.

The following of the chapter is organized as follows: Section 2.2 introduces the concept of distributed systems. Section 2.3 defines the concept of time and causal order in distributed systems. Section 2.4 explains the broadcast primitive, and Section 2.5 gives some background about Mobile Networks.

2.2 Distributed systems

The first section introduces the concept of distributed systems. These systems are mainly composed of processes that communicate with each other either over communication channels or shared memory. In the distributed systems considered in this thesis processes communicate through communication channels by message passing, and shared memory is therefore not further considered. Both processes and communication channels are prone to failures. This section first presents the characteristics of processes and communication channels, then it introduces the failures that can occur on them.

2.2.1 Processes

Distributed systems are composed of many types of machines such as personal computers, laptops, mobile devices, cars, servers, etc... Processes are an abstraction of the machines on which a distributed system runs. Processes can be seen as threads: a process executes on a machine, and a machine might run several processes.

Distributed systems usually contain many processes. The set of processes of the system is denoted $\Pi = \{p_1, .., p_N\}$. Processes usually know Π as well as the number of processes of the system. Processes are said to have a partial view of the system when they only partially know Π. For example, in very large systems, maintaining

a view of Π might be too costly in terms of memory consumption. Moreover, when processes join and leave the system it might be difficult, or even impossible, to maintain an up-to-date view of Π.

In a distributed system, each process executes its local algorithm(s) independently of the other processes. Most works assume processes to be asynchronous: each process executes its local algorithm(s) without making assumptions on the relative speed of other processes. Assuming an asynchronous model allows to make no assumption on the devices on which processes execute (computational power) or on the network connecting the devices (transmission delays). On the other hand, some works assume a synchronous model: a bound exists on the relative speed of processes, as well as on the communication delays between processes.

A *distributed algorithm* is composed of a set of local algorithms distributed over the processes of the system. Processes might execute different sets of local algorithms, such as in a client / server approach. Each process executes its algorithm(s) sequentially, i.e., it executes algorithm instructions one by one [Ray13]. Hence, instructions occurring on the same process are totally ordered following their execution order. Each instruction is considered to be atomic, which means that it is executed completely and cannot be interrupted in the middle. An instruction either produces an internal or external event [Ray13]. An example of internal event is the update of a local variable, while an external event is the sending or receiving of a message. Internal events only affect the process itself, and are, therefore, often omitted in a distributed computation. Conversely, external events also involve other processes, and result in an information exchange between processes [RS96], such as the sending or receiving of a message, or a read/write operation on a variable stored in shared memory.

2.2.2 Communication model

Processes often cooperate to work jointly. To this end, they must communicate with each other to exchange information. The two main communication models are shared memory and message passing. Shared memory consists of a segment of memory shared among processes, and processes communicate by reading and writing variables stored in the shared memory segment. This thesis and the related work consider a model based on message passing. Hence, the following describes only the message passing model.

In the message passing model, processes communicate exclusively through messages sent over communication channels [Ray13]. A communication channel connects two processes. A distributed system can be represented by the graph $G = (V, E)$, where V and E respectively consist of the set of processes and communication

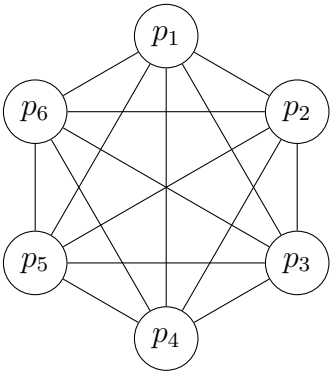


Figure 2.1: Example of complete graph

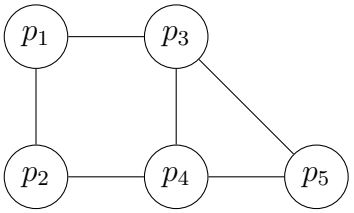


Figure 2.2: Example of graph with paths

channels (links). Communication channels can be either bidirectional or directed, i.e., messages travel in both or only one given direction. In the following of this thesis communication channels are considered to be bidirectional.

The graph G is usually assumed to be connected, meaning that any two processes of the system can communicate with each other. Some works in the literature consider G to be complete, i.e., that a communication channel connects each pair of processes, as in Figure 2.1. Other works assume that each pair of processes can communicate either directly or through intermediate processes. For example, in Figure 2.2 process p_1 can communicate with process p_4 through intermediate processes like p_3 or p_2 then p_4. The sequence of communication channels a message takes to travel from its source process p to its destination process p' is called a *path*. For example, a message m sent by p_1 to p_5 can take the paths $\{p_1, p_3\}, \{p_3, p_5\}$ or $\{p_1, p_2\}, \{p_2, p_4\}, \{p_4, p_5\}$.

Communication channels are usually assumed to be reliable. A reliable communication channel does not lose, alter or create messages. Therefore, messages sent over reliable channels are received by the destination exactly once without modification of the data they contain.

A communication channel might reorder messages sent over it, because the transmission delays on a communication channel can vary from one message to another. A communication channel can for example be a logical channel, and no physical

channel might directly connect the channel's two endpoints, or several physical channels connect the two endpoints. Therefore, two messages sent over the same communication channel might take different physical paths, which leads to different communication delays. Moreover, messages sent over the same physical path might have different transmission delays because of messages losses and retransmissions. Usually, a communication channel is either considered to be FIFO ordered or no assumption is made on the arrival order of messages sent over it. On a FIFO (First In First Out) channel messages are received in the same order as they were sent. For example, if a bidirectional communication channel connecting two processes p and p' is FIFO, then p' (resp. p) receives messages in the same order as p (resp. p') sent them to p' (resp. p).

Distributed systems are considered to be connected (each pair of processes can communicate with each other), either at any given moment or over time. On the other hand, in a non-connected system processes would not be able to communicate with all the other processes, and the system would therefore be composed of several independent smaller connected groups of processes. In a system connected at any given moment there exists a path, at any given moment, which connects any pair of processes. In a system connected over time, such as Evolving graphs [XFJ03] or some classes of Time Varying Graphs [Cas+12] (TVG), processes are connected infinitely often over time by a path of communication channels that allows them to exchange messages.

2.2.3 Failure Models

Distributed systems are prone to failures, either on processes or on communication channels. Works in the literature assume processes and communication channels to be reliable or not. This section presents the different type of failure models, as well as their effects.

Processes execute on devices which can fail. The following failure models are defined in the literature:

- **Omission model** [Ray13]: An omission fault occurs when a process does not send (or receive) a message it is supposed to send (or receive) according to its algorithm. Omission faults result in message losses. Thus, an omission fault results in a process deviating from its algorithm.
- **Crash failure model/ fail-silent crash** [BBG83][SS83a]: A process that fails stops executing instructions for the rest of the execution. A crashed process stops sending and receiving messages.

- **Crash/recovery model** [SS83b]: A process that fails stops executing instructions, but it potentially recovers afterwards and resumes taking instructions. A process that fails temporarily loses the information stored in volatile memory, and only recovers the information stored on a stable support.
- **Byzantine failure model** [LSP82]: A process that fails has an arbitrary behavior. It might stop executing instructions then resumes afterwards, as in the crash/recovery model. Or it might never stop executing instructions but executes some arbitrary instructions. It can even execute instructions against the algorithm and try to provoke a wrong execution of the algorithm.

A process that fails during an execution is said to be faulty. A process can be considered faulty during its failure, or starting from its failure to the end of the execution. A process that is not faulty during the whole execution is said to be correct. During an execution a process can fail and recover several times.

Communication channels are also prone to failures, which are:

- **Message loss**: Messages might be lost. Usually communication channels are assumed as fair-lossy or with no message losses. Fair-lossy communication channels ensure that if a message is sent an infinite amount of time, then it will be received an infinite amount of times.
- **Message corruption**: Messages might be corrupted, i.e., the information they contain might be altered. Corrupted messages are usually detected and handled by network protocols.
- **Message creation**: Messages might be created or duplicated.
- **Channel failure**: A communication channel might fail. Messages currently transiting over a failing communication channel are lost. A failed communication channel might recover, but messages which were in transit over it prior to its failure are lost.

Communication channels can thus be classified as: reliable, fair-lossy, or unreliable. A communication channel that is subject to none of the above failures is said to be reliable. A fair-lossy communication channel assumes that messages might be lost, but that a message that is sent an infinite number of times will be received by its destination an infinite number of times. An unreliable communication channel does no assumption on the reception of messages sent over it.

2.3 Time and causality in distributed systems

This section presents the notion of time and causality in distributed system. Time is a fundamental notion of the human way of thinking: we use it to plan the execution of tasks by associating to them a given duration and starting time. Therefore, providing a time measure in distributed systems renders the elaboration of algorithms for them much easier.

Three types of events occur on a process: internal events, the sending of messages, and the reception of messages. Internal events are naturally ordered following the execution order of the program, and they only impact other processes through the sending and reception of messages[RS96]. Hence, we omit internal events and only consider the sending and reception of messages.

Physical clocks usually cannot be used to timestamp events in a distributed system because of clock drift [KO87][PR94]: Except for atomic and GPS clocks, which are expensive and bulky, physical clocks may have some ticks which are slightly faster or slower than the ticks of other clocks. Hence, even physical clocks that are synchronized at initialization might de-synchronize at an arbitrary (even if slow) rate over time. Consequently, the timestamp of an event on a device d might not be valid for processes executing on other devices than d, and, thus, cannot be used to order events in distributed systems.

Events in distributed systems can, therefore, usually not be timestamped with physical clock values. Nevertheless, Lamport introduced in 1978 both the concept of logical time [Lam78], which is since used to timestamp events in distributed systems, and the *happened-before relation* used to order events timestamped with logical clocks. The *happened-before relation*, denoted $\rightarrow$, is defined as follows:

Definition 2.1. Happened-before relation: Considering two events e_1 and e_2, $e_1 \rightarrow e_2$, if and only if one of the three following conditions holds [Lam78]:

(a) e_1 and e_2 occur on the same process and e_1 precedes e_2.

(b) for a message m e_1=send(m) and e_2=deliver(m).

(c) there exists an event e_3 such that $e_1 \rightarrow e_3$ and $e_3 \rightarrow e_2$.

The *happened-before relation* is shown in Figure 2.3, which represents the timeline of two processes, p_1 and p_2, as well as the three conditions of the relation. Note that condition **(c)** is transitive: $e_1 \rightarrow e_n$ if there exists a sequence of events $e_1, ..e_n$ such that $\forall i \in [1, n-1], e_i \rightarrow e_{i+1}$. Consider two events e_1 and e_2. If $e_1 \rightarrow e_2$, then e_1 is said to causally precede e_2. Likewise, if $e_1 \not\rightarrow e_2$ and $e_2 \not\rightarrow e_1$, then none of both events causally precedes the other, and they are said to be

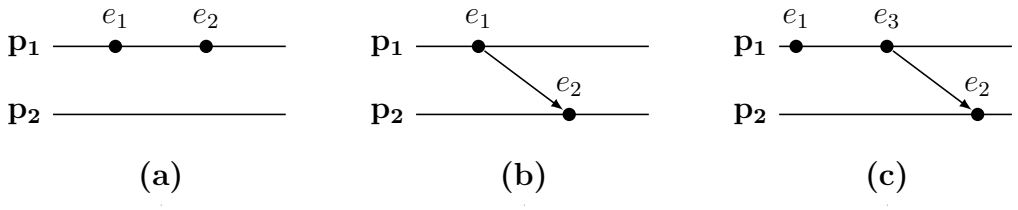


Figure 2.3: Conditions of the happened-before relation

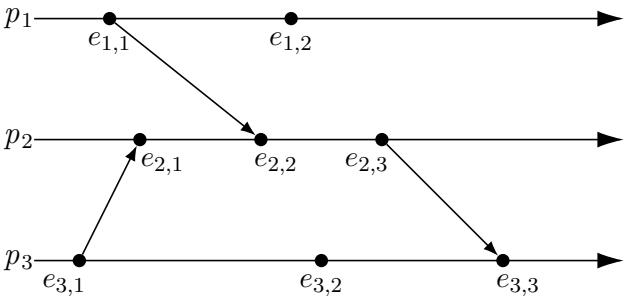


Figure 2.4: Causal order of messages

concurrent ($e_1||e_2$) [Ray13]. Without global clock, it cannot be determined which of two concurrent events happened first [SM94]. Hence, the *happened-before relation* defines a partial order, since it does not order concurrent events.

Figure 2.4 illustrates the ordering of some messages following the *happened-before relation*. The figure represents the timeline of three processes. Time increases from left to right, but no hypothesis is made on the relative execution speed of processes. Indeed, even though $e_{3,1}$ happens before $e_{1,2}$ on the figure, $e_{1,2}$ might happen on p_1 before $e_{3,1}$ happens on p_3. The arrows represent the sending and reception of messages. For example, following condition **(a)**, $e_{1,1} \rightarrow e_{1,2}$. Following condition **(b)**, $e_{3,1} \rightarrow e_{2,1}$. Finally, following condition **(c)**, $e_{3,1} \rightarrow e_{2,2}$, and by transitivity we also have $e_{1,1} \rightarrow e_{3,3}$. Generally, an event e_1 causally precedes an event e_2 if there exists a path from e_1 to e_2 [Mat80].

2.3.1 Causal order

Many distributed applications require ordering events to ensure that applications have a consistent view of the system. Causal order ensures that processes handle messages while respecting the causal relation between them, as defined by the happened-before relation [Lam78]. Therefore, a message is only delivered once all messages that causally precede it are delivered. However, communications between processes might not be causally ordered. Hence, processes must distinguish between the reception of a message on the network layer and the delivery of that message

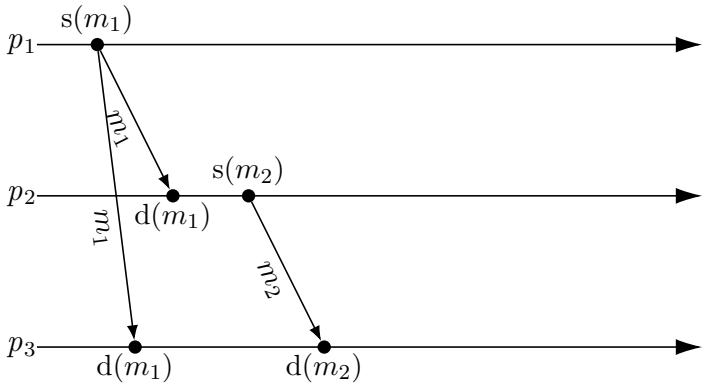


Figure 2.5: Example of message delivered in causal order

to the application to ensure that messages are delivered in causal order [BSS91]. For example, in Figure 2.5, p_2 receives and delivers m_1, then it sends m_2, thus $m_1 \rightarrow m_2$. p_3 receives m_1 and delivers it. Then it receives m_2 and delivers it. Thus, it delivered m_2 in causal order. In Figure 2.6, p_3 receives m_2 before m_1. Hence, it delivers m_2 out of causal order. The formal definition of causal order is as follows:

Definition 2.2. Causal order. Consider two messages m and m' sent to the same process p, and $send(m)$ and $deliver(m)$ the events that correspond to the sending and delivery events of m respectively. If the send event of m causally precedes the send event of m', then p must deliver m before m', or more formally [RST91]:

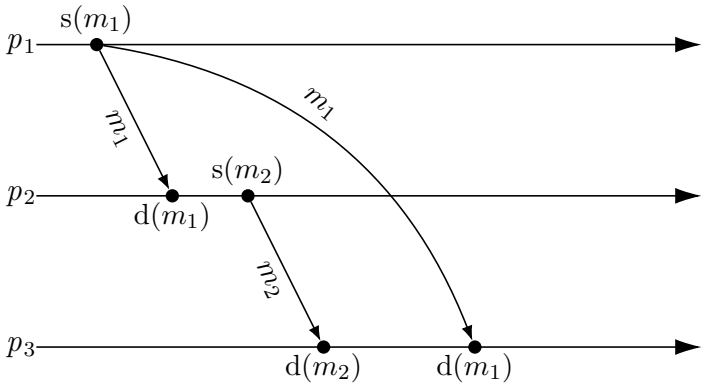
$$send(m) \rightarrow send(m') \Rightarrow deliver(m) \rightarrow deliver(m')$$

2.3.2 Logical clocks

Several works propose to use logical clocks to capture time in distributed systems [Mat80] [Fid88] [SM94].

Logical clocks have first been introduced by Lamport in 1978 [Lam78]. A logical clock is basically a timestamp, which is used to order events.

In a system using logical clocks, each event is timestamped with a logical clock value. A function $C : e \rightarrow T$ attributes a logical clock to each event e, and the set of logical clocks $C(e)$ form a time domain T. The time domain is partially ordered by the happened-before relation. The function C timestamps events such that the following property holds:



s(m): send(m) d(m): deliver(m)

Figure 2.6: Example of message delivered out of causal order

$$\text{For each two distinct events } e_i \text{ and } e_j,\ e_i \rightarrow e_j \Rightarrow C(e_i) < C(e_j).$$

The system is said to *capture* causality when it satisfies the above property. On the other hand, it is said to *characterize* causality [SM94] when it satisfies the following, stronger, property:

$$\text{For each two distinct events } e_i \text{ and } e_j,\ e_i \rightarrow e_j \iff C(e_i) < C(e_j).$$

Systems using logical clocks differ in the function C they use, as well as the algorithm to update logical clocks. The function C determines the time domain T. The algorithm called by a process to update logical clocks consists of two rules: one to update the logical clock when producing an event (send event or internal event), and a second one to update the logical clock when receiving a message.

2.3.3 Scalar clocks

In 1978, Lamport proposed scalar clocks which consists of a scalar, initialized at 0. Algorithm 1 describes the algorithm executed by processes in a system using scalar clocks. Every process p_i maintains a scalar value C_i as logical clock, and updates it at each event (internal event, or the sending or reception of a message).

Scalar clocks capture causality. Indeed, for any two events e_i and e_j, if $e_i \rightarrow e_j$, then $C(e_i) < C(e_j)$. However, scalar clock do not characterize causality. Indeed, for any two events e_i and e_j, we can have $C(e_i) < C(e_j)$ while no causal relation exists between e_i and e_j. Hence, scalar clocks might order events which do not have

Algorithm 1: Scalar clock algorithm executed by p_i

When sending a messge m

1: $C_i = C_i + d, d > 0$
2: send(m, C_i)

Upon reception of (m, C_m)

3: $C_i = max(C_i, C_m)$
4: $C_i = C_i + d, d > 0$
5: deliver(m)

Upon execution of an internal event e

6: $C_i = C_i + d, d > 0$
7: execute(e)

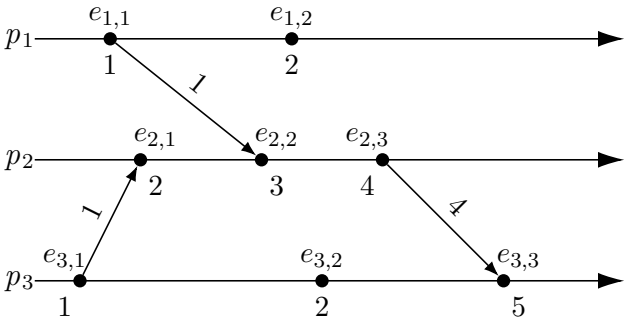


Figure 2.7: Example of a system using scalar clocks

any causal relation, and it is impossible to deduce whether e_i causally precedes e_j when $C(e_i) < C(e_j)$.

Figure 2.7 is a system composed of three processes. The figure shows the scalar clock value after the execution of each event, as well as the scalar clock piggybacked on each message. This example shows that scalar clocks do not characterize causality: $C(e_{3,2}) < C(e_{2,3})$, while $e_{3,2}$ and $e_{2,3}$ are concurrent.

2.3.4 Vector clocks

Vector clocks, proposed independently by Fidge [Fid88] and Mattern [Mat80], use a vector clock of N integers as time domain, where N is the number of processes inside the system. Each process p_i maintains a vector V_i of size N. $V_i[i]$ and $V_j[j]$ with $i \neq j$ respectively represent p_i's local time and p_i's knowledge of p_j's local time. Thus, V_i consists of p_i's view of the global logical time, and p_i uses it to timestamp events. Initially, all entries of vector clocks are set to 0. Algorithm 2 describes the algorithm executed by processes in a system using vector clocks.

Algorithm 2: Vector clock algorithm executed by p_i

When sending a messge m

1: $V_i[i] = V_i[i] + d, d > 0$
2: send(m, V_i)

Upon reception of (m, V_m)

3: $\forall j \in [0, N-1], V_i[j] = max(V_i[j], V_m[j])$
4: $V_i[j] = V_i[j] + d, d > 0$
5: deliver(m)

Upon execution of an internal event e

6: $V_i[j] = V_i[j] + d, d > 0$
7: execute(e)

Schwarz and Mattern [SM94] defined the following relations to compare two vector clock timestamps: Let V_i and V_j be the vector clocks associated to the events e_i and e_j. We have:

- $V_i \leq V_j \iff \forall k \in [1..N-1], V_i[k] \leq V_j[k]$
- $V_i < V_j \iff V_i \leq V_j \wedge \exists k \in [1..N-1], V_i[k] < V_j[k]$
- $V_i || V_j \iff \neg(V_i < V_j) \wedge \neg(V_j < V_i)$

Schwarz and Mattern [SM94] also proved that vector clocks characterize causality. For two events e_i and e_j of vector clock V_i and V_j respectively, we have:

- $e_i \rightarrow e_j \iff V_i < V_j$
- $e_i || e_j \iff V_i || V_j$

Figure 2.8 shows the same execution as Figure 2.7 with processes using vector clocks. Contrarily to scalar clocks, the comparison of the vector clocks of $e_{3,3}$ and $e_{1,3}$ shows that both events are concurrent.

Charron-Bost [Cha91] proved that a structure of size N, where N is the number of processes of the system, is the minimal structure required to characterize causality. Hence, is it not possible to characterize causality with less information than that contained in vector clocks.

2.4 Broadcast

This section presents some background on the broadcast primitive. Many distributed applications such as client-replicated servers [Cac+01] or parallel applications [TKB92] require a group communication service which allows a process to

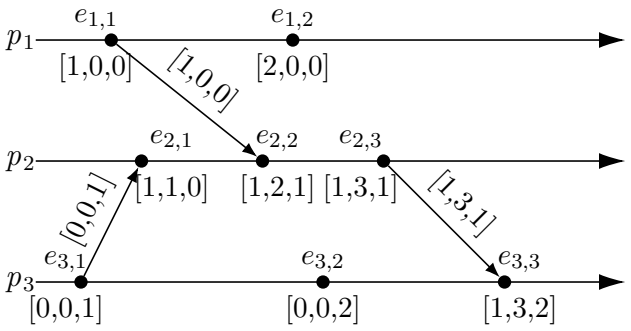


Figure 2.8: Example of a system using vector clocks

send a message to all (broadcast) processes of a group. The broadcast primitive can be provided at the network layer, or it can be implemented at the application layer. We present the broadcast specifications, as well as several message orderings of broadcast messages. We also describe the broadcast primitive in systems prone to failures, since the distributed system of the first work in this thesis contains processes that might fail.

2.4.1 Broadcast specification

Two primitives are offered to the processes:

- BROADCAST(M): called by a process to broadcast a message to all processes of a group. The function returns directly and does not block, i.e., it does not wait till all processes acknowledged the delivery of m.
- DELIVER(M): called by a process to deliver m to the application. A process calls it once the delivery conditions of m are satisfied.

In a system without failures and in which all channels are reliable, a broadcast primitive must ensure the following properties:

- **Validity:** If a process delivers a message m from a process p, then p previously broadcasted m (no creation).
- **Integrity:** A process delivers a message m at most once (no duplication).
- **Termination:** a message broadcasted by a process is eventually delivered by all processes.

2.4.2 Broadcast message ordering

Many applications require broadcast messages to be ordered. However, as seen in Figure 2.5, processes can receive messages in any order, due to network delays for example. Hence, a broadcast primitive that orders messages must implement a mechanism to store messages and delay their delivery to the application till their delivery conditions are satisfied. The literature gives mechanisms to ensure three orders of broadcast messages [KS08]:

- **FIFO order:** Messages broadcasted by the same process are delivered in their broadcast order by all processes [KS08]. Formally, if a process broadcasts m_1 then m_2, then all processes will deliver m_1 before m_2.
- **Total order:** Processes deliver messages in exactly the same order [Lam78]. Formally, if a process delivers m_1 before m_2, then no process delivers m_2 before m_1. Note that Total order does not imply Causal order. For example, if $m_1 \rightarrow m_2$ and if all processes deliver m_2 before m_1, then they respect Total but not Causal order.
- **Causal order:** Processes deliver messages while respecting the causal relation between them [Lam78]. Formally, if the broadcast of m_1 causally precedes the broadcast of m_2, then processes deliver m_1 before m_2.

In a system without failures and in which all channels are reliable, ordered broadcast primitives must ensure the properties of **Validity**, **Integrity** and **Termination** defined in Section 2.4.1, as well as one of the ordering properties given above.

2.4.3 Reliability of the broadcast primitive

In a distributed system prone to failures some processes might not receive all broadcast messages. For example, a process might fail while executing the broadcast primitive. *Reliable broadcast* requires the following properties to be satisfied:

- **Validity:** If a correct process broadcasts a message m, then it eventually delivers m.
- **Integrity:** A correct process delivers a message m at most once (no duplication).
- **Termination:** A message broadcasted by a correct process is eventually delivered by all correct processes.

- **Agreement:** If one correct process delivers a message m, then all correct processes eventually deliver m.

Reliable broadcast does not make any assumptions on behalf of faulty processes, contrarily to *Uniform reliable broadcast* which respectively replaces the **Agreement** and **Integrity** properties by the **Uniform Agreement** and **Uniform Integrity** properties:

- **Uniform Agreement:** If a process (including faulty processes) delivers a message m, then all correct processes eventually deliver m.
- **Uniform Integrity:** A process (including faulty processes) delivers a message m at most once (no duplication).

2.5 Mobile Networks

This section presents Mobile Networks. A chapter of this thesis focuses on Mobile networks with a core infrastructure, also called Cellular Networks [Mia+16]. Completely decentralized networks such as Ad-hoc networks or WSN networks are out of the scope of this thesis. In recent years, there is a fast increasing number of small devices called *hosts* with the development of mobile devices and the Internet of Things (IoT). Hosts communicate with each other through a core network, composed of a set of servers called *stations*. Hosts communicate with stations through wireless channels. The network composed of stations and hosts is called a Mobile Network. The characteristics of Mobile Networks with a core infrastructure make most of the causal broadcast algorithms not suitable to them. This section first presents the characteristics of Mobile Networks, then it presents the specifications of causal broadcast in Mobile Networks.

2.5.1 Network characteristics

Processes communicate through two networks, namely a wired and a wireless network, whose characteristics are very different [FZ94].

In the wireless network, processes communicate by using wireless antennas. Such antennas have a given range which defines the distance a message can travel. The characteristics of wireless network are:

- **Unreliability:** Messages might be lost due to interferences. Hence, a control mechanism must be implemented to re-transmit messages and acknowledge their reception.

- **Low bandwidth:** Wireless networks have a low bandwidth, especially when compared to the much higher bandwidth of wired networks. Hence, the number of messages sent over the wireless network, as well as the control information attached to them, should be reduced as much as possible.
- **Shared medium:** The wireless network is shared among all processes: a message m sent by process p_1 to a process p_2 might be lost due to a message m' sent by a process p_3 to any other process. Contrarily, a wired communication channel is shared only by its two endpoints. Therefore, the collision of messages is handled much easier and is much lower on wired communication channels than on wireless ones.
- **Broadcasting:** Broadcasting a message to all processes over the wireless network is very easy: a message sent over a wireless antenna is inherently received by all processes which are in the antenna's communication range.

The wireless network is used for communication between hosts and stations. Hosts communicate with stations exclusively through the wireless network. A host only communicates with the station to which it is connected [IB93]. The characteristics of wireless networks impose several constraints: the number of messages sent over it should be limited, as well as the size of the control information messages contain. Moreover, a mechanism should be implemented to handle message retransmission and message acknowledgment. Such characteristics are well suited to the UDP transport protocol, which provides a broadcast primitive, and which does not make any assumptions (such as reliability) on messages sent over the network.

The characteristics of wired networks are:

- **High bandwidth:** Wired networks are composed of high speed communication channels and have therefore a high bandwidth.
- **Reliability:** Wired networks are considered to be reliable. Hence, messages sent over them are eventually received (as long as the destination is a correct process).
- **FIFO:** Communication channels are considered to be FIFO ordered: if processes p_1 and p_2 that are connected through a direct communication channel and p_1 sends a message m_1 then a message m_2 to p_2, then p_2 will receive m_1 before m_2.
- **Point-to-Point communication:** Wired networks are inherently based on point-to-point communications: they are composed of wired channels whose two endpoints communicate through it point-to-point.

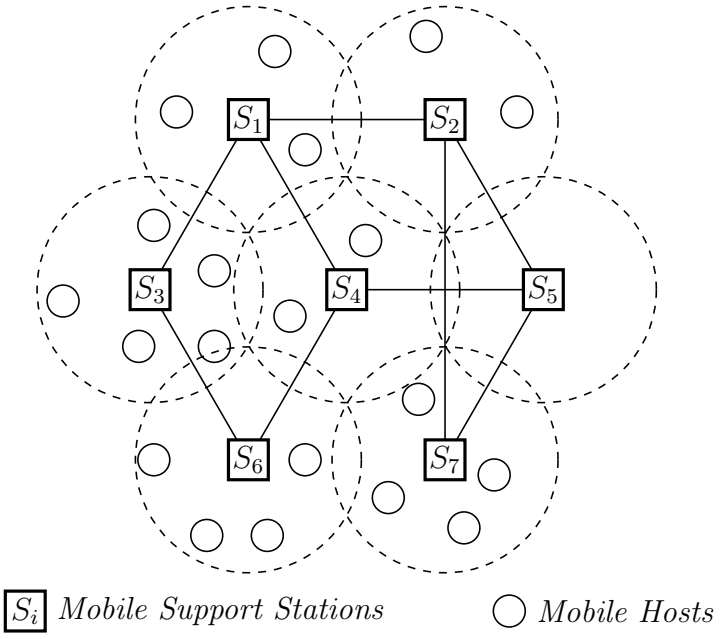


Figure 2.9: Example of Mobile Network

The wired network is exclusively used by stations, which communicate with each other exclusively through it. The characteristics of wired networks make them much more suited to a high number of messages and control information attached on messages [IB93]. The TCP protocol is well suited to wired networks since it provides the guarantees they give: reliability, FIFO order of messages, and point-to-point communications.

To conclude, the throughput of mobile networks is often limited by the wireless network which has lower capacities than the wired network. Hence, algorithms for mobile networks usually put as much of the message load and control information on the wired network in order to relieve the wireless network.

2.5.2 Mobile Support Station and Mobile Host characteristics

Hosts and stations have very different characteristics. Basically, hosts have much more constraints and limitations than stations [FZ94][BAI94]. Figure 2.9 shows an example of a Mobile Network, with stations forming the core of the network and hosts the endpoints.

2.5.2.1 Mobile Support Stations

The core of Mobile Networks is composed of mobile support stations, called stations. Mobile Networks contain much more hosts than stations [IB93]. Each station is equipped with an antenna. The area covered by the transmission range of a station's antenna is called the station's *cell*. A station is at the center of its cell, and communicates with the hosts inside its cell through the wireless network.

The characteristics of stations are as follows:

- **Fixed location:** A station has a fixed position and acts as a relay for hosts in its cell. Hence, the localization of stations is usually chosen to have a maximum area coverage.
- **Reliability:** Stations are usually considered to be reliable through hardware replication, because a failing station would disconnect the area covered by it. Ensuring area coverage could also be achieved with cell overlapping, but interferences on the wireless network should be avoided, and cell overlapping would increase interferences significantly.
- **High energy capacity:** Stations are connected to an energy source and have, therefore, no energy limitation.
- **High computational and memory capacity:** Stations have a large storage and computational capacity, or at least a much higher one than hosts, since they are not constrained by their size and energy consumption.

To conclude, stations are the backbone of Mobile Networks: they are usually reliable, do not move, and have large capacities. Hence, they can execute heavy algorithms.

2.5.2.2 Mobile Hosts

Mobile Networks usually contain many Mobile Hosts, denoted hosts. They are the active actors: they are the source and destinations of messages, while stations ensure that messages are delivered to the destination(s). Hosts can either be connected to a station at the beginning of the execution, or they can join, and also leave, the system during execution [IB93].

The characteristics of hosts are as follows:

- **Dynamics:** A host might join and leave the network at any moment. Most models assume that a host leaves the network only after it successfully notified the station to which it is connected.
- **Mobility:** Hosts move freely. A host might leave the cell of the station to which it is connected, and moves to another cell. The host must then connect to the station of its new cell. It might also move temporarily to an area which is not covered by any station, and might lose, therefore, temporarily the connection with the rest of the system.
- **Limited energy:** Hosts are mobile devices and have, therefore, a limited energy capacity.
- **Limited computational and memory capacity:** Hosts have a limited computational and memory capacity. They are restricted by their size, as well as by their energy capacities.
- **Low reliability:** Hosts are subject to transient and permanent failures. For example, a host is temporarily faulty until its battery is recharged, or is permanently faulty if it has a hardware failure. A faulty host is fail-silent and stops sending, receiving, processing messages, and loses all variables stored in volatile memory. Hence, a host that recovers might first need to obtain its lost information.

Hosts are the active actors of Mobile Networks. They can execute the functions:

- **Join():** Called by a host to join the system. The host connects itself to a station in range of its wireless antenna. A host is considered to have joined the system once the station receives its join request.
- **Leave():** Called by a host to leave the system. A host is considered to have left the system once the station receives its leave request.
- **Connect():** Called by a host when it moves into a new cell in order to connect itself to the new cell's station.
- **Broadcast(m):** Called by a host to broadcast a message m.
- **Deliver(m):** Called by a host to deliver a message m. A host is notified when it buffers such a message m, and calls $Deliver(m)$ in the notification-handler.

To conclude, hosts are the active actors of Mobile Networks: they broadcast and deliver messages, and join or leave the system during execution, while stations ensure that hosts can communicate with each other. Their capacity limitations (energy, computational, storage) should be taken into account when conceiving algorithms for mobile networks.

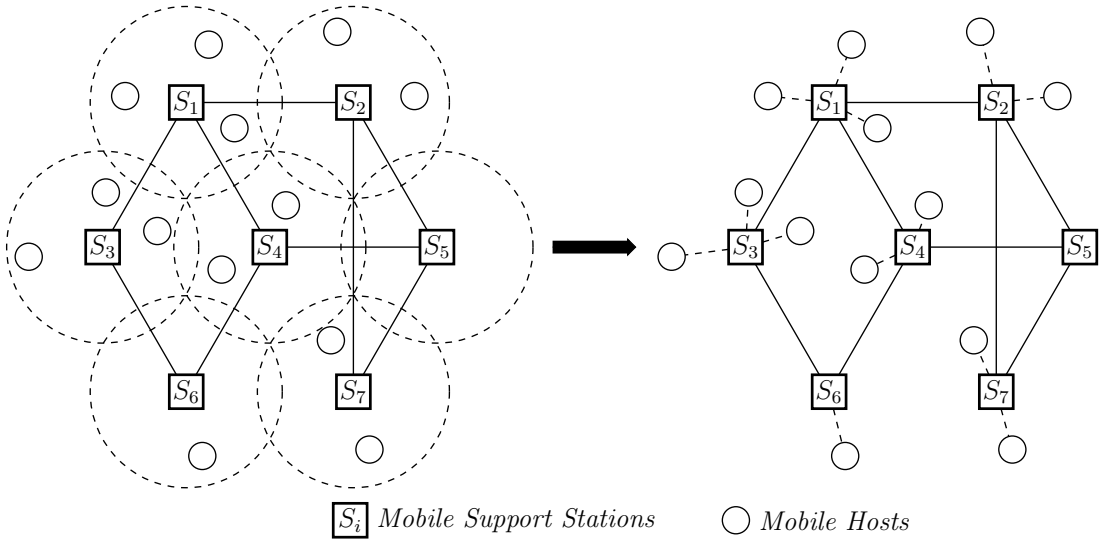


Figure 2.10: A Mobile Network and its graph representation

2.5.3 Ordering messages in Mobile Networks

Many distributed applications require to order application messages (FIFO, causal or total order) [KS08]. To this end, messages are usually buffered at reception till their respective delivery conditions are satisfied. A control mechanism then orders messages using control information attached on messages. Algorithms are divided in two parts: one executed by hosts and another executed by stations.

As previously discussed, stations have much higher capacities than hosts [KS08]. Moreover, they are usually reliable, while hosts might fail. Hence, most of the algorithm to order messages is usually put on stations. A station usually holds the ordering information on behalf of hosts connected to it. Hosts then only need to maintain a small set of information to ensure the causal delivery of messages.

A host is connected to at most one station at a given moment, which is the station that holds its ordering information. A host discards messages from stations other than the one to which it is connected. Hence, even though a host can receive messages from multiple stations through the wireless network, it discards those that do not come from the station to which it is connected. A Mobile Network, as the one on the left of Figure 2.10, can then be represented as the one on the right of Figure 2.10 where dashed lines represent the connections of hosts with stations, and the solid lines the wired connections between stations.

Keeping information on stations to order messages creates a new challenge: when a host moves to another cell, that information must be transferred to the station of the host's new cell. In this case, the station of the host's previous cell should exchange messages with the station of the host's new cell. This message exchange is called a **Handoff** [CK04].

Hosts can also join the system during execution. A host which joins the system during execution might not deliver those application messages broadcasted prior to its arrival, because those messages might already be discarded (the alternative would be to never discard any message) by the stations. Hence, a host that joins the system during execution will not deliver those messages discarded prior to its arrival. The *Termination* property of causal broadcast is therefore modified as follows:

- **Termination:** A message m sent by a host is eventually delivered by all hosts that *joined* the system when m was sent and which did not *leave* the system.

To conclude, stations hold most of the information and execute the majority of the algorithm to order messages. This usually requires stations to exchange messages when a host moves from one cell to another, in order to pass that information from the host's previous cell's station to the host's new cell's station (Handoff). Moreover, the termination property of causal broadcast must be adapted to Mobile Networks.

2.6 Conclusion

This chapter presented the background required to understand the following of this thesis. The first section presented the model of distributed systems, on which the work of this thesis is based. The second section introduced the concept of time as well as causal order, while the third section presented the broadcast primitives. The fourth section introduced the model of Mobile Networks, for which the first part of this thesis provides a causal broadcast primitive.

Chapter 3

Related Work

Contents

3.1 Introduction

This chapter presents the work of the literature relevant to this thesis.

Causal broadcast algorithms for distributed systems can be classified into six categories, based on the causal order structure they use and the assumptions they make. We present and compare the algorithms of each category. Finally, we compare the categories and give the advantages and disadvantages of each of them.

Most distributed algorithms are not adapted to the specific characteristics of Mobile Networks which are composed of Mobile Hosts and Mobile Support Stations. Most of the control and storage is done on stations, since hosts have much lower resources.

The first causal multicast algorithms for mobile networks were proposed in [Ala95], and other causal multicast algorithms for mobile networks can be seen as variations of these algorithms. Some algorithms were also proposed to tolerate failures [AB94][PKV96][ARV93][ABL04].

The first section describes existing causal broadcast algorithms for distributed systems in general. The second section describes existing causal order algorithms for Mobile Networks, for which most causal broadcast algorithms are not adapted due to the specific features of such networks.

3.2 Causal Broadcast

Birman et al. [Bir85] implemented the first causal broadcast algorithm in 1985 inside the ISIS system [Bir85][BJ87][BV93]. The algorithm causally orders messages by attaching to them the log of all messages that causally precede them. ISIS is fault-tolerant and ensures that a broadcasted message is either received by all or no correct process. However, attaching on messages the log of their causal precedences is not sustainable because the logs eventually become very large.

Many approaches have since been proposed to ensure causal broadcast with fewer causal information and/or assumptions on the system. Causal broadcast algorithms for distributed systems can be classified in six categories, based on the causal order structure they use and the assumptions they make:

1 - The algorithms of the first category attach to messages all the causal information required to causally order them.

2 - The algorithms of the second category organize the network in an overlay and disseminate messages over that overlay more efficiently, i.e., by using less information than vector clocks.

3 - The algorithms of the third category causally order messages with a high probability by using vector clocks whose size is independent to the number of processes inside the system.

4 - The algorithms of the fourth category disseminate messages through FIFO channels thus ensuring that messages are implicitly causally ordered upon reception. Therefore, no causal information is appended on messages.

5 - The algorithms of the fifth category append physical clock value timestamps on messages, and use those timestamps to order messages. They require synchronized physical clocks.

6 - The algorithms of the sixth category define causal order on the application level. Those approaches can be combined with the algorithms of the other categories.

3.2.1 Tracking causal order with structures of size $\mathcal{O}(N)$

Algorithms of the first category attach on messages all information required to causally order messages. We first describe the approach using vector clocks with one entry per process. Second, we present optimizations of vector clocks to reduce their size or enable processes joining and leaving the system during execution. Third, we describe prime clocks which encode vector clocks into one scalar number.

3.2.1.1 Vector clocks with one entry per process

Schiper et al. propose in [SES89] to heavily reduce the size of causal information attached on broadcast messages by using vector clocks [Fid88][Mat80] instead of a log of causal precedences [Bir85]. Birman et al. [BSS91] proposed in 1991 a causal broadcast algorithm using vector clocks (see Algorithm 3) which was also implemented in the ISIS system.

The algorithm uses a causal order structure composed of a vector clock with one entry per process inside the system. Each process p_i maintains a local vector clock V_i whose entries are initialized to 0. Before broadcasting a message m, p_i increments $V_i[i]$ and attaches V_i to m. When p_i receives a message m of vector clock V_m from a process p_j, it caches m until the following condition is satisfied: $\forall k \in [0, N-1] \backslash \{j\}, V_m[k] \leq V[k] \wedge V_m[j] = V_i[j] - 1$. This condition ensures that p_i delivers all messages that causally precede m before delivering m. When p_i delivers m, it also increments the entry $V_i[j]$ to register its delivery of m.

The advantage of vector clocks is that the size of causal information does not depend on the number of messages but on the number of processes, which is often much smaller. However, vector clocks thus do not scale with the number of processes, since one entry of the vector clock is associated to each process. Moreover, adding or removing processes during execution requires an additional algorithm to modify the size of the vector clock.

Charron-Bost [Cha91] proved that the minimal data structure required to characterize causality when broadcasting in a system with N processes has a memory complexity of $\mathcal{O}(N)$. Hence, vector clocks with one entry per process are the minimal data structure that characterize causality. Some algorithms presented in the following have an average memory complexity lower than $\mathcal{O}(N)$, but still have a memory complexity of $\mathcal{O}(N)$ in the worst case. Algorithms which have a lower memory complexity need to include assumptions about the system, such as FIFO channels or a given system topology.

Algorithm 3: Vector clock algorithm [SES89] executed by p_i

p_i **broadcasts a messge** m

1: $V_i[i] = V_i[i] + 1$
2: broadcast(m, V_i)

Upon reception of (m, V_m) **from** p_j **at** p_i

3: $\forall k \in [0, N-1] \backslash \{j\}, waitUntil(V_m[k] \leq V[k] \wedge V_m[j] = V_i[j] - 1)$
4: $V_i[j] = V_i[j] + 1$
5: deliver(m)

Golden et al. [RGI] proposes a vector clock whose size can vary during execution, thus tolerating process churn. It requires FIFO channels as well as some additional data structures to keep track of terminated processes. In order to handle process churn, Wang et al. [Wan+06] proposes to organize processes in a logical ring topology where channels are FIFO.

Almeida et al. have proposed Interval Tree Clocks (ITC) [ABF08] in 2008, which are vector clocks that allow processes to join and leave the system. Therefore, they are adapted to dynamic systems. An ITC is a tree structure, whose size adapts dynamically to the number of nodes inside the tree. The set of messages as well as the set of processes are each represented by an ITC. Authors provide primitives to create, remove and reuse ITC entries (i.e., provide unique message and process IDs). Experiments show that in static systems the memory consumption of ITCs is lower than the one of vector clocks, and, in dynamic systems the memory consumption of ITCs is slightly higher than the one of an algorithm which does a mapping of IDs to counters. Hence, ITCs are an alternative to vector clocks in dynamic systems.

Table 3.1 summarizes the vector-based causal broadcast approaches. Some approaches do assumptions on the network topology, Wang et al. [Wan+06] and

Almeida and al [ABF08]. All approaches require reliable communication channels and two of them additionally require them to be FIFO. All vector-based approaches have a message and local memory complexity in $\mathcal{O}(N)$. Three of the vector based approaches tolerate processes to join and leave the system during execution.

Table 3.1: Summary of vector-based causal broadcast approaches

Paper	Network Topology	Channels	Dynamics	Message memory	Local memory
Schiper et al. [SES89]		Reliable	✗	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Golden et al. [RGI]		FIFO Reliable	✓	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Wang et al. [Wan+06]	Ring-overlay	FIFO Reliable	✓	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Almeida et al. [ABF08]	Tree-overlay	Reliable	✓	$\mathcal{O}(N)$	$\mathcal{O}(N)$

N : Number of processes inside the system

3.2.1.2 Compressed and bounded vector clocks

Most vector clock based algorithms do not scale because they use vector clocks whose size grows with the number of processes. Several algorithms have been proposed to reduce the size of vector clocks.

Baldoni [Bal98] pointed out that traditional vector clock algorithms do not reset vector clock entries, thus causing them to grow indefinitely. Authors propose to bound the values of vector clock entries by using the sequence number of message acknowledgments, which are commonly used by protocols such as TCP. The algorithm uses a vector clock with one entry per process, with each entry being bounded by k, i.e., $\forall i, 0 \leq V[i] < k$. A process delays the broadcast of new messages such that it has at most k simultaneous broadcasted messages that are not acknowledged by other processes. A process that delivers a message m sends an acknowledgment to the sender p_i of m, and increments the entry $V[p_i]$ as follows: $V[p_i] = (V[p_i] + 1)\%k$. Bounding the values of vector clock entries avoids their overflowing and reduces the number of bits on which they are encoded. However, those vectors still require one entry per process, thus making them not scalable with the number of processes.

Singhal and Kshemkalyani [SK92] and Birman et al. [BSS91] proposed an algorithm based on compressed vector clocks, where a process only attaches on messages the entries it modified since its last broadcast. Each process p_i maintains a vector clock V_i of N entries, as well as a data structure of size N to keep track of modified entries of V_i since p_i's last broadcast. This approach can greatly reduce the size of causal information attached on messages when only a few processes broadcast most of the messages. However, in the worst case messages carry N

tuples $(entry, value)$, i.e., the causal information overhead is in $\mathcal{O}(2 * N)$. Hence, the compression effectiveness depends on the locality and temporality closeness of message broadcasts, as shown by Lee et al. [LKS11]. Moreover, this approach is not adapted to dynamic systems.

Table 3.2: Summary of compressed vector-based causal broadcast approaches

Paper	Channels	Dynamics	Message memory	Local memory
Baldoni [Bal98]	FIFO Reliable	✗	$\mathcal{O}(N/k)$	$\mathcal{O}(N/k)$
Birman et al. [BSS91], Singhal [SK92]	FIFO Reliable	✗	$\mathcal{O}(b)$	$\mathcal{O}(N)$

N : Number of processes inside the system
k : Constant number of bits to store an entry, constant fixed at initialization
b : Number of messages locally delivered since last local broadcast

3.2.1.3 Prime numbers

Kshemkalyani et al. introduced the Encoded Vector Clocks (EVC), which are vector clocks encoded in one scalar number by using primes [KKS18]. EVCs use the property that each number has a unique prime factorization [CP06]: $\forall k \in N, \exists! n_1, .., n_k$ prime numbers and $v_1, ..v_k \in \mathbb{N}$ such that $k = \prod_i n_i^{v_i}$. For example, $10 = 2^1 * 5^1$ or $1530 = 2^1 + 3^2 + 5^1 + 17^1$, and there exists no other prime factorization of 10 and 1530.

Causal information is encoded in EVCs. Instead of associating to each process a unique vector clock entry, authors associate to each process p_i a unique prime number n_i. Process p_i also maintains a number k_i initialized to 1. When broadcasting a message m, p_i multiplies k_i by n_i and attaches the result to m. When p_i receives m with attached number k, it first computes the prime factorization of k. Let's consider the prime factorization of $k = \prod_j n_j^{v_j}$. The vector clock corresponding to this prime factorization is $V = [v_0, v_1, ..v_n]$, with n_i being the prime number associated to process p_i. Hence, p_i must first deliver, $\forall k, v_k$ messages from process p_k before delivering m. Authors provide the operations required to encode, compare, and merge EVCs.

The first advantage of EVCs is that they tolerate process churn, since each new process simply takes the next lowest unassigned prime number. However, a mechanism must ensure that two processes do not take the same prime number. On the other hand, encoded vector clocks have a high space and operation complexity [KV19]. In fact, the smallest prime number is 2. Hence, each broadcasted messages multiplies k by at least 2. Therefore, a message that causally depends on

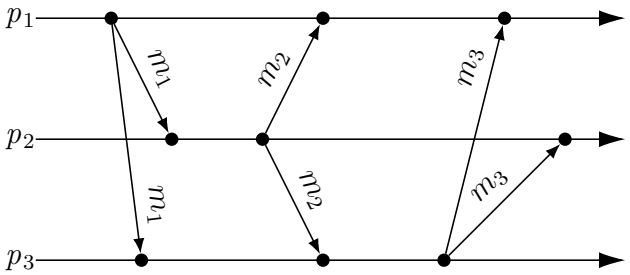


Figure 3.1: Example of direct dependencies

n messages has an attached number $k > 2^n$, i.e., k grows exponentially. Thus, the number of bits required to store k grows with each broadcasted message, as well as the operation complexity to factorize k. Simulation results [KSV20] show that EVCs grow very fast.

Pozzetti and Kshemkalyani [PK21] propose the Resettable Encoded Vector Clock (REVC) to bound the size of EVCs. The growth of an EVC can be bounded by assigning to it a given number of bits, and resetting it whenever it overflows. Each process keeps an EVC history in which it stores previous EVCs and to which it adds its EVC before resetting it. A process appends its EVC on messages it broadcasts. Hence, RECV still have an unbounded linear growth with the number of messages in the system. To tackle this problem, the authors propose to remove obsolete EVCs by regularly cleaning the EVC history.

3.2.1.4 Direct dependencies (Causal barrier)

Prakesh et al. [PRS96] showed that a message's direct dependencies are sufficient to ensure its causal delivery. The direct dependencies of a message m are the messages m' such that $m' \rightarrow m$ and no message m" exists such that $m' \rightarrow m$" $\rightarrow m$. For example, Figure 3.1 shows the broadcast of three messages m_1, m_2 and m_3, with $m_1 \rightarrow m_2 \rightarrow m_3$. In this example, m_2 is a direct dependency of m_3, since no message m' exists such that $m_2 \rightarrow m'$ and $m' \rightarrow m_3$. Conversely, m_1 is not a direct dependency of m_3, and is said to be an indirect dependency, because $m_1 \rightarrow m_2$ and $m_2 \rightarrow m_3$.

The algorithm presented in [PRS96] uses direct dependencies to track causality. It mainly focuses on causal multicast but is easily adaptable to causal broadcast, as explained in the paper. Hence, only the broadcast case is detailed here. Each process p_i keeps a counter seq_i to uniquely identify messages it broadcasts with the id (p_i, seq_i). Each process also keeps two vectors: one of N entries to track which messages have been delivered by other processes, and a dynamic one called CB, which stores the IDs (p_k, seq_k) of direct dependencies of the next message to

broadcast. The size of CB is upper bounded by N, the number of processes, but is often much smaller. When process p_i broadcasts a message, it increments seq_i then broadcasts $(m, CB_i, (p_i, seq_i))$. Moreover, it resets CB_i to $\{p_i, seq_i\}$. When p_i delivers a message $m = (CB_k, (p_k, seq_k))$, it removes from CB_i the entries that are contained in CB_k ($CB_i = CB_i \backslash CB_k$), since they become indirect dependencies through m ($\forall m_k \in CB_k, m_k \rightarrow m \rightarrow m$", with m" being the next message that p_i broadcasts). Therefore, p_i reduces the size of CB_i when broadcasting a message, or when delivering a message which has direct dependencies contained in CB_i.

The size of causal information attached on messages depends on the number of messages broadcasted per second inside the system, and not the number of processes. Hence, it scales well with the number of processes. Moreover, it tolerates process churn and is therefore adapted to dynamic systems. Cai et al. [CLZ02] further reduce the size of causal information attached on messages when using the algorithm for causal multicast. Chandra et al. [CGK04] analyze the performance of the algorithm under various network conditions and show that it scales well with the number of processes.

3.2.2 Organizing processes in hierarchical overlays

Organizing processes in a hierarchical overlay to provide causal broadcast was first proposed by Adly and Nagi [AN96]. Taguchi et al. present a similar approach in [TET04]. In [AN96] processes are organized in a logical multilevel hierarchy, called HARP [ANB93], which allows processes to send and receive messages from only a few processes. Processes are organized into clusters, and clusters are organized into a tree. Each cluster is connected to its parent cluster through a father process in its parent cluster. A process uses a vector clock to causally order messages inside its cluster and with its parent/child processes in other clusters. Hence, a process keeps a vector clock to track causality in regard to only a (small) subset of processes inside the system. Therefore, a process maintains a vector clock whose size is only equal to the size of that subset of processes. The algorithm is implemented over an unreliable network and processes may fail, leading to temporary network partitions. Processes store causal information on stable support. A restructuring algorithm is provided to expand and reorganize the network. Processes can also join and leave the network. However, they cannot move from one cluster to another. Moreover, concurrent messages are reordered at processes which connect clusters.

Evropeytsev et al. [Evr+16][Evr+17] proposed an algorithm for Peer-to-Peer networks. Processes either belong to a cluster or not. Each cluster has a super-peer. Processes inside a cluster communicate with processes outside it through the cluster's super-peer. Super-peers and processes that do not belong any cluster form

a cluster. Inside a cluster, processes store direct dependencies in bit vectors, which have a maximum size of g bits, where g corresponds to the number of processes inside the cluster. The advantage of this algorithm is that the causal information carried by messages depends on the number of peers inside the cluster, and not the total number of peers, which is usually much higher. Moreover, the algorithm uses direct dependencies and not a structure with one entry per process inside the system. However, no mechanism is provided to handle failures, and concurrent messages are reordered at the super-peer level. Taguchi and Takizawa [TT03] and Hsiao and Liao [HL11] provide similar algorithms, which use vector clocks instead of direct dependencies to track causality.

De Araujo et al. presented VCube-PS, a causal multicast algorithm [Ara+18] built on top of a VCube [DBK14], which organizes processes in a hypercube-like topology. Authors extend the algorithm to causal broadcast in [de +18a][de +18b]. Authors observe that building a single tree to disseminate messages induces overheads in terms of tree maintenance in presence of process churn, as well as delays for message propagation. Moreover, the message load is not well-balanced in the case of multicast, since messages transit more over some processes, like the root. The algorithms use direct dependencies [PRS96] to ensure causal order. Messages are propagated through multiple spanning trees, which are built dynamically and whose construction by processes only uses local information. A process that broadcasts a message becomes the root of the spanning-tree used to propagate that message. Therefore, no global tree is maintained. The VCube has logarithmic properties and ensures, for example, that a path of $log(N)$ hops exists between each pair of processes. Since messages are propagated over different paths, they might be received out of causal order. A process can infer the spanning trees built by other processes. It takes advantage of the path intersections of the different spanning trees, by delaying to its children in a message's propagation tree, the forwarding of the messages for which it knows that they did not deliver all causal dependencies yet. Hence, processes group messages without inducing any overhead. The advantage of VCube-PS is that the overlay structure is maintained dynamically, and the VCube properties ensures a $log(N)$ path between each pair of processes, thus keeping the delays introduced by the overlay low. However, the system must implement a VCube, and the algorithm does not tolerate process churn nor failures.

Santos and Rodrigues [SR19] introduced localized causal broadcast, where each process keeps information of only processes at up to $2f + 1$ hops, where f is the maximum number of simultaneous tolerated failures. Processes track causal order with a vector clock with one entry for each process at up to $2f + 1$ hops. Process failures, which cause multiple receptions of the same message, are handled by appending on messages the identifier of the communication channels they might follow. Communication channels are assumed to be FIFO. The algorithm organizes processes in a tree, and avoids partitioning by adding for each process a

communication channel with $f+1$ distinct other processes. When process j broadcasts a message m, it attributes to m an identifier $(source, target, seq)$ for each of its communication channels. Each process that receives m also adds an identifier $(source, target, seq)$ for each of its communication channels. Processes remove the tuples in a message's identifiers of processes at a distance higher than $2f+1$ hops. However, $f+1$ tuples are added to the message's identifier at each hop, leading to a causal information overhead of $\mathcal{O}(f^2)$. Those identifiers are used in case of a process failure to handle multiple reception and avoid multiple deliveries. The path identifiers and causal information are attached on messages, leading to causal information of size $\mathcal{O}(f^2)$ attached on messages. The advantage of the algorithm is that it tolerates up to $f+1$ simultaneous failures. However, processes must maintain causal information of processes at up to $2 * f + 1$ hops.

Several hierarchical approaches [PS97] have been tailored for Mobile Networks, which have inherently a hierarchical two-layered composition: the core is composed of Mobile Support Stations, to which Mobile Hosts are connecting themselves (see Section 2.5). Since the first contribution of this thesis focuses on causal broadcast in mobile networks, the algorithms for such networks are described in a dedicated section (Section 3.3).

Table 3.3: Summary of hierarchical-based causal broadcast approaches

Paper	C.O.	Channels	Dynamics	Memory overhead	Overlay
Adly and Nagi [AN96]	V.C.	Unreliable	✓	$\mathcal{O}(c)$	Tree
Taguchi and al. [TET04]	V.C.	Unreliable	✓	$\mathcal{O}(c)$	Tree
Evropeystev et al. [Evr+16]	D.D.	Reliable	✗	$\mathcal{O}(c)$	Tree
Hsiao and Liao [HL11]	V.C.	Reliable	✗	$\mathcal{O}(c)$	Tree
De Arajo et al. [de +17]	D.D.	Reliable	✗	$\mathcal{O}(N)$	Hypercube
Santos and Rodrigues [SR19]	V.C.	FIFO Reliable	✓	$\mathcal{O}(f^2)$	each process has f+1 outgoing links

V.C. : Vector Clocks
D.D. : Direct Dependencies
N : Number of processes inside the system
c : Number of processes inside a cluster
f : Number of tolerated simultaneous failures

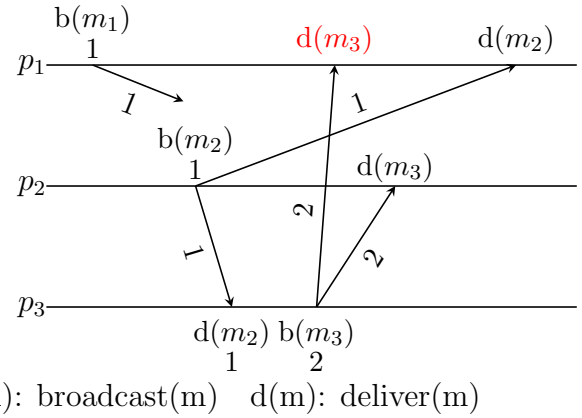


Figure 3.2: Message delivered out of causal order when using Plausible Clocks

3.2.3 Vector clocks of size M≤N

Torres-Rojas and Ahamad introduced in 1998 the concept of logical clocks of constant size [TA99]. Instead of using a vector clock with one entry per process, constant size clocks are clocks whose number of entries M is smaller or equal to N, the number of processes inside the system. Usually $M \ll N$. Charron-Bost proved that a vector clock with fewer than N entries cannot characterize causality of broadcast messages [Cha91]. Hence, these clocks do not characterize causality and algorithms using them to causally order messages might deliver messages out of causal order.

Torres-Rojas and Ahamad proposed Plausible Clocks (PC) [TA99], which associates each process to one vector clock entry. Since Plausible clocks have $M \leq N$ entries, each vector clock entry might be associated to several processes. The algorithm using PCs is otherwise similar to the one using vector clocks of N entries [SES89]. Processes which use PCs to causally order broadcast messages might deliver some of them out of causal order. For example, in the execution of Figure 5.1 the message m_3 is delivered out of causal order. In the example, PCs have one entry, and each process is associated to this unique entry. First, process p_1 increments its PC when broadcasting m_1. We assume that other processes receive m_1 later in the execution. Process p_2 broadcasts m_2 and increments its PC. Process p_3 broadcasts m_3 after receiving m_2, i.e., $m_2 \rightarrow m_3$. However, p_1 receives m_3 prior to receiving m_2, and since $PC_1 = 1$, it delivers m_3 out of causal order. Although PCs do not characterize causality, the algorithm delivers messages in causal order with a high probability, as shown by a theoretical and experimental analysis [TA99][Tor01].

Gidenstam and Papatriantafilou [GP03] observed that PCs order some concurrent messages, and introduced Non-Uniformly Mapped R-Entries Vector (NUREV) clocks. The idea behind NUREV clocks is that the mapping of the vector clock entries to processes has an impact on the number of concurrent messages that are ordered. For example, if mostly two processes broadcast messages, then they should not be associated to the same vector clock entry. The purpose of NUREV clocks is therefore to dynamically adapt the mapping of clock entries to processes. The algorithm locally maintains a vector of N entries but still only attaches a PC on messages. Authors show that accuracy basically depends on the communication patterns and the value-differences of compared PCs. Two adaptive mapping strategies based on these observations are given and evaluated through simulations. NUREV clocks are particularly effective if only some processes broadcast messages, since those processes can dynamically be associated to an exclusive clock entry. NUREV clocks also reduce the probability that a message is delivered out of causal order.

Mostéfaoui and Weiss [MW17b] proposed Probabilistic clocks (PrC). PrCs use a function f to associate one or several vector clock entries to each process. $|f|$ is determined at initialization and is the same for each process, i.e., each process has the same number of vector clock entries associated to it. Authors provide the formulas to determine the best value of $|f|$, depending on the number of concurrent messages in the system. Authors also provide a formula to compute the probability that two processes concurrently increment the same clock entry (which leads to messages that are delivered out of causal order). They show that the effectiveness of PrCs depends on the number of concurrent messages, through experiments measuring the number of messages delivered out of causal order for different message loads. Experiments also show that PrCs ensure causal order with a higher probability than PCs.

Misra and Kshemkalyani introduced Bloom clocks [Ram19][MK21][KM21]. Bloom clocks ensure causal order with Bloom filters [Blo70][TEL12]. Introduced in 1970 by Bloom, such filters are vectors of fixed size used in conjunction with hash function(s). Instead of using one hash function that associates 1 to M clock entries to each process, Bloom clocks use 1 to M hash functions that each associate one clock entry to each process. Hence, they have the same advantages and disadvantages than Probabilistic clocks.

The main advantage of vector clocks of size $M \leq N$ are their size which is independent of the number of processes, making them scalable in regard to the number of processes in the system. Moreover, they are well adapted to process churn, since vector entries are not associated to only one process and can therefore be shared and reused. However, they do not characterize causality and algorithms which use

them might deliver messages out of causal order. Hence, they are adapted to systems where causal order only impacts performance and not correctness, such as, for example, the fairness of mutual exclusion algorithms.

Table 3.4: Summary of constant clock-based causal broadcast approaches

Paper	Entries Function	Channels	Dynamics	Message memory	Local memory
Torres-Rojas and Ahamad [TA99]	$\|f\| = 1$	Reliable	✓	$\mathcal{O}(M)$	$\mathcal{O}(M)$
Gidenstam and Papatriantafilou [GP03]	f dynamic	Reliable	✓	$\mathcal{O}(M)$	$\mathcal{O}(N)$
Mostéfaoui and Weiss [MW17b]	$1 \leq \|f\| \leq M$	Reliable	✓	$\mathcal{O}(M)$	$\mathcal{O}(M)$
Misra and Kshemkalyani [MK21]	several f with $\|f\| = 1$	Reliable	✓	$\mathcal{O}(M)$	$\mathcal{O}(M)$

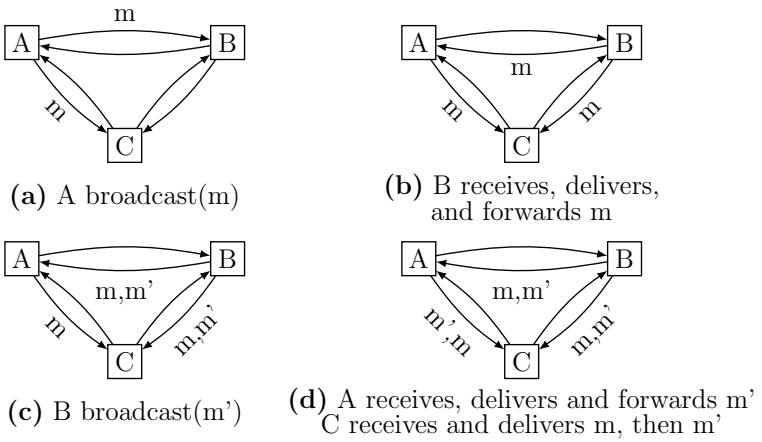
N : Number of processes inside the system
M : Constant clock size, constant fixed at initialization
f : Function that returns the entrie(s) associated to a process

3.2.4 FIFO-based approaches

Friedman and Manor [FM04] first formalized and proposed an algorithm that ensures causal order through flooding in a static overlay network composed of FIFO communication channels. Flooding through FIFO channels ensures that no path exists over which messages travel out of causal order. For example, in Figure 3.3 *A* broadcasts *m*, which causally precedes *m'* broadcasted by *B*. All processes receive *m* before *m'* since upon reception all processes forward *m* on all their communication channels. Therefore, processes deliver messages upon reception, since they receive messages already causally ordered.

The algorithm in [FM04] is executed over a hypercube overlay topology [DBK14], which is well adapted to flooding, since it ensures that any two processes are at a maximal distance of $\mathcal{O}(log(n))$ hops. Blessing et al. [BCD17] propose to implement causal broadcast based on flooding over a tree overlay, thus removing the message overhead introduced by cycles (which cause processes to receive messages several times).

Causal broadcast through flooding has two main advantages: (1) Processes receive messages in causal order and can thus deliver them at reception. (2) Messages carry no causal information.



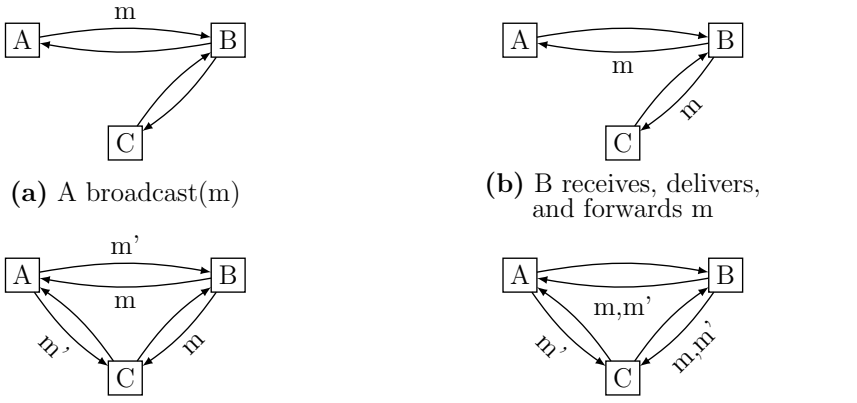
(a) A broadcast(m)

(b) B receives, delivers, and forwards m

(c) B broadcast(m')

(d) A receives, delivers and forwards m'
C receives and delivers m, then m'

Figure 3.3: Causal broadcast through flooding [FM04]



(a) A broadcast(m)

(b) B receives, delivers, and forwards m

(c) Adding channel between A and C
A broadcast(m')

(d) B receives, delivers and forwards m'
C might receive m' before m

Figure 3.4: Causal broadcast through flooding [FM04] in a dynamic network

However, this approach also has disadvantages:

- Messages must be sent over each communication channel, inducing a high message overhead depending on the communication graph. This issue can be addressed by organizing processes into a logical topology, as do the authors of [FM04][BCD17].
- Adding a new communication channel creates a shortcut over which processes can temporarily receive messages out of causal order. For example, in Figure 3.4, process A first broadcast a message m. After broadcasting m, it adds a new communication channel with C, then broadcasts m'. The path $A - C$ is then a shortcut that m' can take, but shouldn't till C receives and delivers m. Hence, the algorithms of [FM04][BCD17] only work in static networks.

Nédelec et al. [NMM18a][NMM18b] extend [FM04] to some dynamic networks. Authors observe that messages can only travel out of causal order temporarily over a newly added communication channel, as in the example above. More precisely, adding a communication channel creates a shortcut that newly broadcasted messages can take. Thus, they can be received by processes while some of the messages that causally precede them are still transiting over the network. Authors provide a procedure to initialize communication channels before using them to broadcast messages. The procedure consists of a message exchange between the two endpoints of the added communication channel, ensuring that messages transiting over it have no causal dependencies that are still transiting, i.e., not received yet by the other endpoint. This message exchange is done using communication channels which have already been initialed through this procedure, or which are present since the beginning of the execution (and are considered to be initialized). Messages from added communication channels are discarded until they are initialized. Therefore, each pair of processes must always be connected by a path of initialized communication channels. One disadvantage of [NMM18a] is that messages must be buffered to be sent to the other endpoint at the end of the initialization procedure. The buffer containing these messages may grow infinitely. Authors provide a simple solution in [NMM18b] which consists of sending the messages contained in the buffer over the network when it exceeds a given size, thus clearing the buffer. The advantages and disadvantages of this approach are the same as [FM04][BCD17], while authors make this approach suited to dynamic networks in which there always exist a path of initialized channels between each pair of processes. Therefore, an endpoint such as a client cannot migrate from one server to another by disconnecting itself from the server in between.

Table 3.5: Summary of FIFO-based causal broadcast approaches

Paper	Network Topology	Channels	Dynamics	Message memory	Local memory
Friedman and Manor [FM04]	hypercube overlay	FIFO Reliable	✗	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Blessing and al. [BCD17]	tree overlay	FIFO Reliable	✗	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Nédelec and al. [NMM18a]	network always connected with initialized links	FIFO Reliable	✓	$\mathcal{O}(1)$	$\mathcal{O}(1)$

3.2.5 Physical time based approaches

In distributed systems, physical clocks cannot be simply used to timestamp messages because clocks are drifting apart [KO87][PR94]: even if clocks are perfectly synchronized at initialization, they might slowly de-synchronize over time because they might tick slightly slower or faster than the other clocks. Nevertheless, protocols exist to synchronize physical clocks, such as the Network Time Protocol (NTP) [85][Mar+10] or the Precision Time Protocol (PTP) [08]. These protocols have a precision going on average from 10 to 100ms. The literature contains several algorithms that use those protocols to causally order messages, by assuming physical clocks synchronized with a maximum bounded clock skew: they assume that for each pair of processes p and p' of physical clocks $pc(p)$ and $pc(p')$, $|pc(p) - pc(p')| < \epsilon$.

Nishimura et al. [Nis+05] combined a hierarchical one-level structure with synchronized clocks. Processes are logically divided into clusters, and each cluster has a gateway process responsible for the communication with other clusters. Each pair of gateway processes is connected by a logical communication channel (i.e., one-level structure). Authors assume that clusters are synchronous, i.e., the physical clocks of processes in the same cluster are synchronized with an upper bound ϵ_c. Moreover, authors assume an upper bound delay ϵ_d on the communications between gateway processes. Messages carry a physical clock timestamp and are ordered by taking into account ϵ_c and ϵ_d. The algorithm has the advantage that it requires only one integer representing physical time to order messages causally. However, messages might be delivered out of causal order if one bound, ϵ_c or ϵ_d, is not respected. Moreover, no mechanism is provided to handle failures, and particularly failures of gateway processes.

Hybrid Logical Clocks (HLC) [Kul+14][Kul+][Roo+19][FJ19] and Physical clocks With Causality (PWC) [KAN22] ensure causal order by combining physical clocks and logical clocks. HLC clocks are used in systems such as MongoDB [Tyu+19], CausalSpartan [RMK17] or GentleRain [Du+14][RK16]. Both approaches assume physical clocks to be synchronized with a maximum clock skew of ϵ. HLC and PWC are composed of a physical clock pc and a logical scalar clock lc. The logical clock is used to order events which have an identical value of pc (if $m.lc = 3$, then the process knows that there are two messages which causally precede m and which have the same pc value). Hence, for each event e and e', $e \rightarrow e' \iff pc(e) < pc(e') \vee (pc(e) = pc(e') \wedge lc(e) < lc(e'))$. lc is reset each time a process receives a message or updates its physical clock, thus ensuring that lc is bounded. Authors of both clocks provide an analysis of the required size of lc to avoid its overflow. They argue that HLC and PWC can be stored in the lower bits of 64 bits integer, because applications do not need the precision provided by 64 bits integers. The two clocks scale very well, and they tolerate process churn, since neither the physical nor the logical clock entries are associated to particular processes. However, to ensure

causal order, HLC and PWC require both the maximum clock skew and the interval between send(m) and receive(m) to be higher than ϵ. Thus, messages might be delivered out of causal order if one of the two conditions does not hold.

3.2.5.1 Δ-causal order

The notion of $\Delta-$*causal order* was introduced by Yavatkar in 1992 [Yav92] and later formalized by Baldoni et al. [BRM96]. $\Delta-$*causal order* is defined for systems in which messages have real-time constraints. In such systems, messages have a limited time validity after which they are obsolete. A message sent at time t becomes obsolete at $t + \Delta$ and is therefore discarded if received after that delay. Δ-*causal order* delivers the maximum number of messages before their expiration, while ensuring that messages are delivered while respecting causal order. Such an approach is for example meaningful in multimedia real-time applications, such as teleconferencing [Wak93]. Processes synchronize their physical clocks with protocols like NTP [Mar+10] and messages carry their sending time as causal information. Therefore, Two causally related messages are assumed to have timestamps t_1 and t_2 such that $|t_1 - t_2| > \epsilon$, where ϵ corresponds to the error bound of clock synchronization protocols. Higaki et al. [THT98] observe that communication delays between processes might greatly vary. Hence, the authors modify Δ-*causal order* to also take into account the varying communication delays between processes. Rodrigues et al. [Rod+00] generalize Δ-*causal order* by allowing each message to have a given and distinct deadline.

Baldoni et al. [BRM96] ensure causal order with an algorithm similar to the traditional vector clock algorithm [Mat80]. Each process has an associated vector clock entry, but instead of scalars, the vector clock contains physical time values. A process delivers a message m once all undelivered messages that causally precede m are expired. Baldoni et al. [Bal+96][Bal+98] improve [BRM96] by replacing the vector clock with direct dependencies [PRS96], thus heavily reducing the average (in the worst case $\mathcal{O}(N)$) size of causal information maintained on processes and attached to messages.

Hernández et al. [PDG10] ensure Δ-*causal order* without a physical global clock, by using application specific knowledge to estimate the lifetime of messages. Authors use direct dependencies [PRS96] to track causality. However, they observe that direct dependencies are not sufficient to track causal order when messages are lost. For example, if $m_1 \rightarrow m_2 \rightarrow m_3$ and if m_2 is lost/expires, then it is impossible to know through direct dependencies that $m_1 \rightarrow m_3$. To circumvent this problem and maintain causal information, authors propose to append some indirect dependencies when the message loss rate increases.

Δ-*causal order* is only adapted to systems where messages become obsolete after a given time duration. Moreover, algorithms either require clock synchronization for correctness or use vector clocks of N entries, and both conditions are neither scalable nor adapted to dynamic systems.

3.2.6 Application-based causal order

Causal order algorithms usually do not take into account application specific information. Considering such information allows to causally order messages more efficiently, i.e., with fewer data. For example, two messages m_1 and m_2, related respectively to the variables v_1 and v_2, might not be causally dependent, even if they are sent by the same process. The following describes two algorithms which respectively use application-specific knowledge and the partial replication of data to causally order messages more efficiently. Both approaches are orthogonal to causal order algorithms and can therefore be used in conjunction with them.

Bailis et al. [Bai+12] observed that most algorithms reorder concurrent messages in real world applications. In fact, when a process p broadcasts a message m, most algorithms consider that all messages that p delivered prior to broadcasting m also causally precede m. More formally, for most algorithms:

$$p \text{ broadcasts } m \Rightarrow \forall m' \text{ delivered by } p \text{ prior to broadcasting } m,\ m' \rightarrow m.$$

However, this assumption is false for most real world applications. Consider for example databases that store values v_k. A message sent by a client to update a value v_i has no causal relation with messages related to other values $v_k \neq v_i$. Reordering concurrent messages delays their delivery. In fact, consider two concurrent messages m and m', reordered such as $m \rightarrow m'$ for the algorithm. The algorithm will then unnecessarily delay the delivery of m' till m is delivered. Based on such behavior, the authors of [Bai+12] proposed to explicitly define causality at the application level (which they call *explicit causality*), to track only relevant causal relations between messages. Authors argue that many applications already express semantic dependencies in their APIs, which can be used to explicitly define causality. They give as example Twitter conversations whose length average approximately 11 Tweets (a makes a tweet, to which b reacts with a tweet, to which c reacts with a tweet, etc...) [RCD10][YW10], but whose potential causal chain is several orders of magnitude higher. In this example, explicit causality heavily decreases the number of message dependencies, which in its turn increases throughput, reduces causal information, and improves concurrency.

Bravo et al. introduced SATURN [BRV17], a distributed metadata service that takes into account the partial replication of data. Authors observe that in geo-replicated databases data is often replicated at only a subset of datacenters. However, causal information about a data d is often sent to all datacenters, even to those which do not store d. The propagation of causal information to all databases introduces unnecessary message delivery latencies, as well as a decreased data throughput (because of unnecessary messages). The storage system is responsible for providing labels that are causally consistent and SATURN propagates them only to the databases holding a copy of those labels, while respecting their causal relationship. SATURN is therefore a mechanism complementary to existing databases. It allows them to take advantage of partial replication at a cost of a small message overhead. Hsu et Kshemkalyani. [HK17][HKS18] and Crain et al. [CS15] also provide algorithms to ensure causal order in partially replicated systems.

3.2.7 Summary

Table 3.6 summarizes the causal broadcast implementations presented in this section. We divided the implementations in six categories corresponding to the data structure they use to ensure causal broadcast.

Vector clock approaches are based on the traditional vector clock [Fid88][SES89] proposed by Mattern and Fidge independently, as well as algorithms extending vector clocks in order to make their size dynamic, making them tolerant to processes which join and leave the system during execution. They attach on messages all the causal information required to causally order them. However, such approaches do not scale.

Compressed and bounded vector clock approaches aim to reduce the size of vector clocks on average, by either reducing the number of bits on which each entry is encoded, or by only piggybacking the vector clock entries that have been modified since the last broadcast. However, those approaches still have a space complexity in $\mathcal{O}(N)$ and do therefore not scale.

Prime clocks encode vector clocks into a number, by using the fact that each number has a unique decomposition of prime factors. However, the numbers into which prime clocks are encoded grow exponentially, as do the number of bits required to store them. Therefore, authors of [PK21] propose a procedure to reinitialize prime clocks to bound their size. Note that the prime clock approach is independent to the number of processes inside the system, but it depends on the number of messages that have been broadcasted. Prime clocks tolerate process churn, but they grow very fast with a space complexity in $\mathcal{O}(k)$ where k corresponds to the number of sent messages since the beginning or since the last resetting of the clock.

Direct dependencies only piggyback on a message m the causal information about messages that directly precede m. Therefore, this approach is also independent to the number of processes but depends on the pattern of message broadcasting. Nevertheless, it requires processes to locally store a vector with one entry per process inside the system.

Hierarchical structures divide processes into subgroups. They ensure causal order inside and between groups by using vector clocks or direct dependencies. Therefore, they require less causal information since a process only keeps causal information about the processes of its group and proxy processes of other groups. However, hierarchical structures require a specific network topology which often does not tolerate process churn.

Constant size clock approaches ensure causal order by using vector clocks whose size is constant, independently of the number of processes inside the system. Therefore, those approaches do scale and allow processes to join and leave the system. However, they do not characterize causality and processes therefore might deliver messages out of causal order, even though they usually deliver them in causal order with a very high probability.

FIFO-based approaches ensure causal order by forwarding messages through FIFO communication channels, thus ensuring that messages are causally ordered at reception. Therefore, they do not require any causal information to be appended on messages. However, they require FIFO communication channels as well as assumptions on the network topology.

Approaches using physical clocks require a bounded clock skew between processes, which is ensured by network protocols. However, those approaches require that the clock skew between processes is always below a given known value in order to ensure the causal delivery of messages.

Δ-causal order approaches observe that in some applications the data has a limited time duration validity. They deliver messages once their validity is about to expire. These approaches are restricted to applications where data has a limited time duration validity.

Some works propose to use application specific knowledge to causally order messages. Application specific knowledge can for example be used to order separately independent messages sent by the same process. Often data is also only partially replicated in a distributed system, and propagating updates about a data d only to the processes that store d can significantly reduce the amount of sent information.

Table 3.6: Summary of causal broadcast approaches

Data structure	Channels	Dynamics	Message memory	Local memory	Causality guarantees
Vector clocks	FIFO Reliable	✓	$\mathcal{O}(N)$	$\mathcal{O}(N)$	✓
Compressed vector clocks	FIFO Reliable	✗	$\mathcal{O}(N/k)$	$\mathcal{O}(N/k)$	✓
Prime clocks	Reliable	✓	$\mathcal{O}(b)$	$\mathcal{O}(b)$	✓
Direct dependencies	Reliable	✓	$\mathcal{O}(b)$	$\mathcal{O}(b)$	✓
Hierarchical structure	Reliable	✗	$\mathcal{O}(c)$	$\mathcal{O}(c)$	✓
Constant size clocks	Reliable	✓	$\mathcal{O}(M)$	$\mathcal{O}(M)$	Probabilistic
FIFO-based	FIFO Reliable	✓	$\mathcal{O}(1)$	$\mathcal{O}(1)$	✓
Physical clocks	Reliable	✓	$\mathcal{O}(1)$	$\mathcal{O}(1)$	Physical time
Δ-causal order	Reliable	✓	$\mathcal{O}(N)$	$\mathcal{O}(N)$	Physical time

N : Number of processes inside the system
k : Constant number of bits to store an entry, fixed at initialization
b : Number of messages locally delivered since last local broadcast
c : Cluster size
M : Size of M-entry clock. Fixed at initialization

3.3 Mobile Networks

Most distributed algorithms are not suitable to the features of Mobile Networks composed of Mobile Hosts (MHs) and Mobile Support Stations (MSSs). MHs have a limited computational, memory, and battery capacity, and they communicate through the wireless network which is not reliable and has a low memory bandwidth compared to the wired network. On the other hand, MSSs have much higher capacities and communicate through the wired network. Therefore, causal order algorithms for mobile networks should concentrate information and execution on MSSs. They must also take into account the dynamics of MHs. In the literature, causal order algorithms for mobile networks have generally been proposed for multicast. Nevertheless, they are easily adapted to causal broadcast since the latter is a simplified case of multicast. Therefore, we present the broadcast versions of those algorithms.

3.3.1 First causal order algorithms for Mobile Networks

The first causal order algorithms for Mobile Networks were proposed by Alagar [Ala95][AV97] who proposed three algorithms which use vector clocks to track causality. The algorithms ensure causal order of messages, but they are easily

adapted to causal broadcast. The author copes with the limited capacities of hosts by managing the causal information at the station level: each station manages the causal information on behalf of the hosts connected to it by keeping a vector clock to track causal order for every host connected to it. Each station manages its cell which is defined by the hosts connected to it. Causal information is handled at two levels: the intra-cell level and the inter-cell level. The first is composed of a station plus the hosts connected to it and the second is composed of all stations.

By assumption, communication channels of the intra-cell level are FIFO. A station sends a message m to a host connected to it once the host delivered all messages that causally precede m. Thus, stations send messages to hosts in causal order, and hosts deliver them causally since intra-cell communication channels are FIFO.

On the inter-cell level, messages are causally ordered with vector clocks, as in the traditional vector clock algorithm [BSS91]. A station maintains a vector clock for each host connected to it. The three algorithms presented in the paper differ in the vector clock used at the inter-cell level.

In the first algorithm (**AV-1**), stations use a vector clock of N entries to causally order messages, where N corresponds to the number of hosts in the system. This algorithm does not scale with the number of hosts.

In the second algorithm (**AV-2**), stations use a vector clock of M entries, where M corresponds to the number of stations in the system, which is much smaller than the number of hosts in the system ($M \ll N$). The algorithm is based on the observation that all messages broadcasted by a host pass through the station to which the host is connected. Thus, messages are timestamped at the station level. *AV-2* scales much better than *AV-1*, since the number of stations is much lower than the number of hosts. However, messages are reordered at the station level. For example, two messages broadcasted by two hosts connected to the same station will be reordered by that station, even if the two messages are concurrent.

The third algorithm (**AV-3**) is a tradeoff between *AV-1* and *AV-2*. Each station has S entries associated to it to order messages from hosts of its cell. Thus, in *AV-3* stations use a vector clock of $S * M$ entries, where S corresponds to the number of entries associated to each station, and M corresponds to the number of stations in the system. Associating S entries to each station reduces the number of concurrent messages that are reordered, but it increases the size of the vector clock.

AV-1 uses a vector clock whose size depends on the number of hosts, which is prohibitive. *AV-2* and *AV-3* use a vector clock whose size depends on the number of stations. Thus, they scale much better, since the number of stations is much lower than the number of hosts. However, the number of stations can still be high, and the vector clock size of both algorithms can therefore still be prohibitive.

The three algorithms maintain the causal information of a host at the station to which the host is connected. A mechanism, called **Handoff**, is therefore provided for moving the causal information related to a host when it connects to a new station. A Handoff basically consists of a message exchange between stations to transmit the causal information of a host. *AV-1* requires two messages per handoff, while *AV-2* and *AV-3* require M messages per handoff, where M corresponds to the number of stations in the system. The handoffs require two assumptions to ensure correctness: (1) a host always succeeds to connect to a cell's station before leaving that cell, (2) it first sends a disconnect message to a station before leaving its cell. However, it is impossible to ensure these assumptions because of the unreliability of the wireless network.

Other causal order algorithms for Mobile Networks are variants of these three algorithms.

3.3.2 Algorithms with causal information of size $\mathcal{O}(N)$

Like *AV-1*, some algorithms maintain causal information of size $\mathcal{O}(N)$, where N corresponds to the number of hosts in the system.

Prakash et al. [PRS97] ensure causal order using direct dependencies [PRS96] instead of vector clocks (See Section 3.2.1.4). Depending on the broadcast pattern, direct dependencies use much less space but have, in the worst case, a space complexity of $\mathcal{O}(2 * N)$. A station mss_i only sends a message m to a host mh_i only after mh_i delivered and acknowledged all messages that causally precede m, which increases the delivery delays of messages. Dominguez et al. [DPG10] avoids such delays by attaching a bit vector of size N on messages sent over the wireless network, thus enabling hosts to causally order messages by themselves. However, attaching causal information of size $\mathcal{O}(N)$ on messages sent over the wireless network is prohibitive.

Chi et al. [Chi+00] proposed an algorithm that maintains causal information related to a host independently of the host's physical location, thus easily managing them, especially during handoffs. A host mh_i is attached during the whole execution to the first station mss_b to which it connected, regardless of mh_i's location. Messages from and for mh_i are first forwarded to mss_b, which disseminates and causally orders them for mh_i. When changing cells, mh_i only needs to forward its new location to mss_b. Hence, handoffs are very easy and do not require the assumptions made by [Ala95][AV97]. However, the algorithm has a high communication overhead since messages must all be routed through the initial stations instead of being handled locally. Moreover, the algorithm requires that a FIFO path exists through each pair of stations.

Skawratananond et al. [SMG98] proposed an algorithm similar to [RST91], but which implements handoffs more efficiently than [PRS96][AV97] by not requiring that messages exchanged between stations are causally ordered.

3.3.3 Algorithms with causal information of size $\mathcal{O}(\mathcal{M})$

Like *AV-1* and *AV-2*, some algorithms use causal information of size $\mathcal{O}(M)$, where M corresponds to the number of stations of the system.

Li and Huang [LH99] proposed a causal multicast algorithm similar to *AV-2*, but which handles handoffs without requiring that a host always succeeds to connect to a station before leaving the station's cell. On the other hand, the algorithm still requires the assumption that a host notifies the station of a cell before leaving it.

Prakash and Singhal present in [PS97] an algorithm that uses vector clocks with one entry per station like *AV-2*, as well as the concept of *dependency sequences* in order to reduce the number of concurrent messages that are reordered when using vector clocks with one entry per station. A dependency sequence is a range of sequence numbers appended to a message m. Its purpose is to identify concurrent messages to m. The size of dependency sequences is not bounded, even though it is usually small, because they represent ranges. Moreover, they can be periodically discarded through check pointing. However, the algorithm requires the assumption that a host unregisters itself when leaving a cell, which might not always be possible.

Yen et al. proposed an algorithm [YHH97] which is a tradeoff between *AV-1* and *AV-2*. However, the algorithm is only applicable to causal multicast and is therefore not described here. Furthermore, Skawratananond [SMG98] showed that the algorithm violates the liveness property.

Anastasi et al. [ABS99][ABS01] proposed a causal multicast algorithm that uses two additional components, coordinators, and one super-coordinator, both chosen among stations at initialization. The algorithm tracks causal order through vector clocks of C entries, where C corresponds to the number of coordinators, which is much smaller than the number of stations. Hosts order messages causally by themselves: they maintain a vector clock of C entries, which they use to causally order messages at reception and timestamp messages they broadcast. A host mh_i is attached during the whole execution to its coordinator station mss_c, regardless of mh_i's location. Stations forward to mss_c the messages from and for mh_i. mss_c then disseminates messages from mh_i in causal order. Moreover, it causally orders messages for mh_i, and forwards them in FIFO order to mh_i. When changing cells, mh_i only needs to forward its new location to mss_c. Therefore, handoffs are very easy and do not require the assumptions made by [Ala95][AV97]. On

the other hand, the algorithm has a high communication overhead since messages must all be routed through the initial stations instead of being handled locally. The algorithm also requires that a FIFO path exists through each pair of stations, and that the wireless network is FIFO. Moreover, concurrent messages are ordered with a vector clock of C entries, with C being even smaller than the number of stations. Consequently, messages are even more reordered than with *AV-2*. Finally, the coordinators and the super-coordinator are a performance and failure bottleneck.

Chandra and Kshemkalyani [CK04] extended the algorithm in [KS96] to Mobile Networks. The core algorithm is similar to *AV-2*. Authors add to the core of the algorithm the deletion of obsolete messages. A message becomes obsolete once all hosts have delivered it. Each station mss_i maintains a sequence counter seq_i to timestamp with (mss_i, seq_i) messages broadcasted by hosts in its cell. A host acknowledges its delivery of each message. A station forwards an acknowledgement from a host for message (mss_i, seq_i) to mss_i, which sends a *Delete* message for (mss_i, seq_i) to all stations, after receiving an acknowledgement from each host. A station discards message (mss_i, seq_i) upon reception of the corresponding *Delete* message. The advantage of the algorithm is that it handles the deletion of obsolete messages, contrarily to other algorithms. The two main disadvantages of the algorithm are: (1) Station mss_i only sends a message m to a host mh_i once mh_i delivered all of m's causal dependencies. Therefore, m is delayed at mss_i until the host acknowledged m's causal dependencies to mss_i; (2) A lot of acknowledgements must be forwarded on the wired network.

Benzaid and Badache proposed a causal broadcast protocol [BB08] tailored to causal broadcast, while other presented algorithms were conceived for multicast. The algorithm tracks causal dependencies through direct dependencies, introduced by Prakash in [PRS96]. Otherwise, the algorithm is similar to [CK04].

3.3.4 Hierarchical approach

Hsiao and Liao [HL11] proposed an algorithm where hosts that belong to the same cell communicate directly without passing through their cell's station. On the other hand, messages from a host to hosts in other cells still pass through stations. The algorithm tracks causal order with two vector clocks, one for the intra-cell level, and one for the inter-cell level. The intra-cell vector has one entry for each host inside the cell (M), while the inter-cell vector has one entry per station in the system. The handoff requires the assumption that a host succeeds to connect to a cell before leaving. Moreover, each host must know the identity of all the hosts inside its cell, and this cell-membership must be updated each time a host joins or leaves a cell. This algorithm enables hosts of the same cell to communicate with each other without passing through a station. The drawback of the algorithm is

that messages sent over the wireless network must carry a vector of M entries, and hosts must know the identity of all the hosts inside their cell.

3.3.5 Mobile Network failures

Traditional algorithms to handle process failures cannot be applied to Mobile Networks, due to their specific constraints. Hosts have limited memory capacities and might not have stable memory capacities because of thefts or device losses. Therefore, checkpoints should be stored at stations. Moreover, hosts should send as few data and perform as few actions as possible to save battery. Hosts are also mobile and move between cells. Determining the station to which they are or were connected is not immediate. Finally, message losses on the wireless network requires additional control. This section presents algorithms to handle host and station failures.

3.3.5.1 Mobile Host failures

Acharya and Badrinath [AB94] give rules to decide when hosts should do checkpoints when considering uncoordinated check pointing, i.e., each host does local checkpoints without aiming to compute global ones. Authors also give a set of rules to do local checkpoints that can be combined into a consistent global checkpoint.

Krishna et al. [KVP93] propose an algorithm to handle host failures. Stations are assumed to be fault-tolerant. A host mh_i does checkpoints, and send them to the station mss_i to which it is connected. Two strategies are presented: (1) mh_i does a checkpoint at each write event, and sends it to mss_i; (2) mh_i does a checkpoint regularly. mss_i first logs messages before mh_i processes them, thus ensuring that mh_i will not process a message that mss_i cannot recover. Upon recovery, mh_i connects itself to a station to recover its last checkpoint, as well as its logs. It then restores the state stored in the last checkpoint, and delivers the message stored in its logs.

Pradhan et al. [PKV96] consider two checkpointing strategies similar to [KVP93], for which they propose three handoff strategies. The memory of hosts is considered as not reliable, even the stable memory, because of thefts or device losses. Conversely, the memory of stations is considered to be reliable. In the *pessimistic* strategy a host that changes cell sends its checkpoint (and logs if used) to the station of its new cell. Such a strategy ensures that each station stores a checkpoint for hosts connected to it. However, transferring a checkpoint at each handoff consumes data. In the *lazy* strategy no checkpoint is transferred during handoffs. Instead, a host regularly creates and sends a checkpoint to the station to which

it is connected. During a handoff, a host only sends to its new station a linked list containing the stations to which it was connected. When recovering, a host contacts its station, which will look up the most recent checkpoint by contacting the stations in the linked list. In the *Trickle* strategy, a host ensures that at least the previous station to which it was connected stores a checkpoint for it. This avoids a costly message exchange to find a host's checkpoint in the case where a host changes cells often. Authors conclude through experiments that there is no optimal strategy, and that the best recovery strategy depends on several parameters which are the bandwidth of the wireless network, mobility of hosts, and the failure rate of hosts.

3.3.5.2 Mobile Support failures

Alagar et al. [ARV93] propose an algorithm that tolerates station failures. Each station has $k+1$ backup stations, where k corresponds to the maximum number of stations that can be simultaneously down. Consider a host h_i which sends a message m to the station mss_i to which it is connected. mss_i first sends m to its backup stations $Backup_i$ before disseminating m, thus ensuring that if it fails, then stations of $Backup_i$ will take in charge the retransmission of m. The authors suggest to select backup stations for mh_i either based on locality (stations that are close to mss_i) or stations to which mh_i was connected before connecting to mss_i.

Anastasi et al. [ABL04] give a causal multicast algorithm tolerant to station failures. The authors keep the checkpoints at hosts, i.e., they assume that hosts have a reliable stable storage. Therefore, handoffs do not require exchanging information related to checkpoints. The algorithm tolerates the failure of hosts and stations. Stations are grouped as in the approach in [ARV93] and there is one coordinator per group. Stations forward received messages to the coordinator which will send them to all stations of the group. Coordinator failures can be tolerated by using several coordinators per group instead of one.

3.3.6 Summary and discussion

Table 3.7: Summary of Causal order approaches in mobile networks

Paper	C.O.	Wired message overhead	Wireless message overhead	Realistic handoff assumptions	Handoff messages
Alagar - AV-1 [Ala95]	VC	$\mathcal{O}(N)$	$\mathcal{O}(1)$	✗	$\mathcal{O}(1)$
Alagar - AV-2 [Ala95]	VC	$\mathcal{O}(M)$	$\mathcal{O}(1)$	✗	$\mathcal{O}(M)$
Alagar - AV-3 [Ala95]	VC	$\mathcal{O}(M * k)$	$\mathcal{O}(1)$	✗	$\mathcal{O}(M)$
Prakash et al. [PRS97]	VC	$\mathcal{O}(N)$	$\mathcal{O}(1)$	✗	✗
Dominigez et al. [DPG10]	VC	$\mathcal{O}(N)$	$\mathcal{O}(N)$	✗	✗
Chi et al. [Chi+00]	VC	$\mathcal{O}(N)$	$\mathcal{O}(N)$	✓	✓
Skawratananond et al. [SMG98]	VC	$\mathcal{O}(N)$	$\mathcal{O}(N)$	✓	✓
Li and Huang [LH99]	DD	$\mathcal{O}(M)$	$\mathcal{O}(1)$	✗	$\mathcal{O}(1)$
Prakash and Singhal [PS97]	DD	$\mathcal{O}(M)$	$\mathcal{O}(1)$	✗	$\mathcal{O}(1)$
Anastasi et al. [ABS99]	VC	$\mathcal{O}(C)$	$\mathcal{O}(1)$	✓	$\mathcal{O}(1)$
Chandra and Kshemkalyani [CK04]	DD	$\mathcal{O}(N)$	$\mathcal{O}(1)$	✗	$\mathcal{O}(1)$
Hsiao and Liao [HL11]	VC	$\mathcal{O}(N)$	$\mathcal{O}(N)$	✓	$\mathcal{O}(1)$
Benzaid and Badache [BB08]	DD	$\mathcal{O}(N)$	$\mathcal{O}(1)$	✓	$\mathcal{O}(1)$

N: Number of hosts **M**: Number of stations **k**: Constant **C**: Number of coordinators
VC: Vector Clocks **DD**: Direct Dependencies

Table 3.7 summarizes the causal multicast approaches in mobile networks. The column *C.O.* represents the Causal Order structure. The algorithms either use vector clocks or direct dependencies. They make either realistic handoff assumptions, represented by a ✓or unrealistic ones, represented by a ✗. During a handoff, the new and previous station of a host exchange messages. The complexity in terms of number of messages exchanged during handoffs are shown in the *Handoff messages* column. Algorithms that do not handle handoffs are marked with a ✗.

Alagar et al. [Ala95] proposed the first causal multicast algorithms for mobile networks. We denote those algorithms *AV-1*, *AV-2* and *AV-3*. The other causal multicast algorithms for mobile networks are derived from these three algorithms. Only the algorithms proposed by Chi et al. [Chi+00] and Anastasi et al. [ABS99] tolerate handoffs under realistic assumptions. Chandra and Kshemkalyani [CK04] proposed the first algorithm that discards messages once they become obsolete, i.e., once all hosts have delivered them. Hsiao and Liao [HL11] proposed an algorithm where hosts of the same cell communicate directly with each other without passing through the cell's station.

Overall, only some algorithms do realistic handoff assumptions. All algorithms use causal order structures on the wired network with a space complexity either in terms of the number of mobile hosts or the number of stations (or coordinators which can become a bottleneck) in the system. Only one algorithm [CK04] discards the causal information of obsolete messages.

Finally, some works consider the failure of hosts or stations. Algorithms that do tolerate host failures do it through checkpoints that hosts send regularly to the station to which they are connected. Those that tolerate station failures do it by replicating the information stored at stations on backup stations.

3.3.7 Conclusion

This chapter presented the related work of the literature relevant to this thesis. First, we presented the causal broadcast algorithm approaches by grouping them in six categories. Second, we presented causal multicast algorithms for mobile networks, as well as algorithms that tolerate failures in mobile networks.

Causal broadcast approaches are classified in categories corresponding to the data structure they use to ensure causal order.

Some approaches append on messages the causal information required to causally order them. Those approaches use vector clocks with one entry per process, bounded and compressed vector clocks, direct dependencies (causal barrier) or vector clocks of size $M < N$ where N corresponds to the number of nodes in the system. The advantage of those approaches is that they require no or few assumptions on the system. Their main disadvantage is that the causal information attached to messages has in the worst case a space complexity of $\mathcal{O}(N)$ where N corresponds to the number of processes in the system. Hence, they do not scale.

Other approaches append less or no causal information on messages by making assumptions on the network. These approaches are hierarchical structures and algorithms based on dissemination through FIFO communication channels. The main advantage of those algorithms is that they either append no causal information to messages, or they append causal information whose size is depending on a local subset of processes. Hence, they do scale. The main disadvantage is that they require assumptions on the network, which are complicated to maintain in presence of process churn or process failures. Hence, they mostly do not tolerate process churn or process failures, or they only do it under specific conditions.

Some approaches use physical time to causally order messages. These approaches use physical clock timestamps and/or associate a given lifetime to messages. The main advantage of these approaches is that they append few causal information

on messages and do therefore scale. Moreover, they tolerate process failures and process churn. However, they require the physical clocks of processes to be synchronized to at most an upper error bound, and this condition might not be ensured all the time even when using network protocols to synchronize physical clocks.

Finally, some works propose to use application specific knowledge to causally order messages. These approaches can be implemented above the presented causal broadcast algorithms.

The second part presented existing algorithms that ensure causal multicast in mobile networks. Alagar [Ala95] proposed the first three causal multicast algorithms for mobile networks. They order messages either at the host level, thus requiring a vector with one entry per host, or at the stations level, thus requiring a vector with one entry per station, or in between the station and host level, thus requiring a vector with a number of entries between the number of hosts and stations. Other causal multicast algorithms for mobile networks of the literature derive from the three algorithms proposed by Alagar. Most of them propose a handoff procedure to allow a host to change the station to which it is connected. However, most of them require unrealistic network assumptions.

Finally, some works consider the failure of hosts or stations. Algorithms that tolerate host failures do it through checkpoints that hosts send regularly to the stations to which they are connected. Those that tolerate station failures replicate data on backup stations.

Chapter 4

Causal broadcast in Mobile Networks

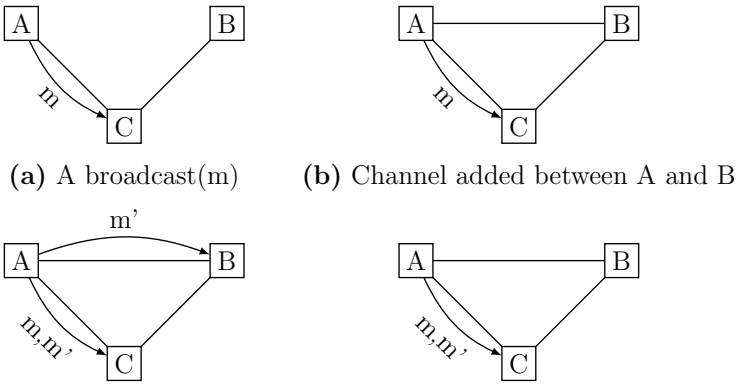
Contents

4.1 Introduction

The first part of this thesis addresses causal broadcast in Mobile Networks, which are mainly composed of Mobile Support Stations and many Mobile Hosts. A causal broadcast algorithm for Mobile Networks should tolerate the many constraints inherent to those networks (see Section 2.5), and it should also scale. This chapter presents two causal broadcast algorithms for Mobile Networks that are based on the FIFO-dissemination approach on an overlay network [FM04]. Both algorithms make realistic assumptions considering the characteristics of mobile networks.

Approaches that piggyback causal information on messages are not suitable to Mobile Networks because they do not scale. The causal information they piggyback on messages either grows with the number of nodes [Fid88][Mat80], or with the message load [MW17b][PRS96]. On the other hand, approaches based on dissemination of messages over FIFO channels [FM04][NMM18a] append no causal information on messages and scale therefore well. They organize the network in an overlay over which messages are disseminated through FIFO channels, ensuring that messages are causally ordered upon reception. Such approaches ensure that messages are causally ordered upon reception, and processes can therefore deliver them directly after receiving them without using a mechanism to order messages at reception. However, existing FIFO-based approaches require assumptions on the overlay that cannot be provided in all systems, like in Mobile Networks. More particularly, FIFO-based approaches require that there exists no path over which messages can travel out of causal order. However, adding a communication channel between two processes temporarily creates a shortcut over which messages can be received by the two endpoints out of causal order. Let's consider for instance the example of Figure 4.1. At (a), A broadcast m. At (b), a communication channel is added between A and B. At (c), A broadcasts a second message m'. We see that m' can take the shortcut $A \rightarrow B$ over which m was not sent. Hence, at (d) we see that B receives m' before m, i.e., it receives m' out of causal order.

Implementing a FIFO-based approach is challenging in Mobile Networks because of the dynamics in the network topology caused by Mobile Hosts that change locations, leading to communication channels that are removed and added regularly. The algorithm must also take into account the constraints of Mobile Networks. Mobile Hosts as well as Mobile Support Stations might fail. Moreover, Mobile Hosts have limitations such as limited memory, computational power, battery life or reliability issues [FZ94]. For example, a mobile host is subject to a transient failure when its battery becomes flate and to a hardware failure when its battery dies or a hardware failure occurs. Finally, the wireless network that connects hosts and stations is unreliable because of interferences that cause message losses.



(a) A broadcast(m) **(b)** Channel added between A and B

(c) A broadcast(m') **(d)** B receives and delivers m' before m

Figure 4.1: Path over which a message can travel out of causal order

To our knowledge, no work proposed causal broadcast that takes into account the constraints of mobile networks. On the other hand, several works address causal multicast in mobile networks [CK04][PRS96][ABS99][BB08][LH99]. However, they usually make unrealistic assumptions such as reliable [CK04][PRS96] and/or FIFO [BB08][LH99] channels, or reliable host connection protocols. For example, they do not address the problem that hosts might fail to connect to a cell's station before moving to another cell. Furthermore, they usually consider that hosts are reliable, i.e., they never fail, and the proposed solutions do not scale. Note that causal multicast algorithms usually require additional information useless for causal broadcast.

We consider that hosts can join/leave the network and fail, permanently or transiently, at any time. They move freely and might be temporarily disconnected from the network when out of range of any station. We assume no reliable connection protocol, and the algorithm handles multiple concurrent connections by the same host. Resource limitations of hosts are handled by keeping causal information at stations, while hosts keep very little control information. Messages piggyback only a few integers as control information.

This work also proposes a decentralized deletion of obsolete messages. Stations handle interferences on the wireless network by caching messages for retransmission. Those messages become obsolete once hosts delivered them. Existing discarding solutions of obsolete information are centralized [CK04][BB08], inducing a high overhead of message traffic (acknowledge messages) and memory storage (a message is cached at all stations even if only one station requires it). The presented algorithms discard obsolete messages cached at stations in a decentralized way: a station discards a message once all hosts connected to it acknowledged the message. Consequently, stations only cache necessary messages, and no extra messages are exchanged for message discarding. A high message traffic inside a cell, which might

lead to delivery delays, as well as storage and communication overheads, only has a local impact.

Summing up, the proposed causal broadcast algorithm is scalable in both the number of hosts and stations, has a low message traffic and storage overhead, while handling mobile network dynamics without the constraining assumptions of FIFO-based [FM04][NMM18a][NMM18a] or causal multicast approaches [CK04][BB08].

Section 4.2 presents the system model, Section 4.3 presents the first causal broadcast algorithm where stations are assumed to be reliable, and Section 4.4 gives the experimental results for this algorithm collected through experiments conducted over the OMNeT++/INET [Var01] simulator. Section 4.5 presents the second algorithm which extends the first algorithm to tolerate station failures.

4.2 Model

The mobile networks considered in this work are composed of Mobile Hosts, denoted *host*(s), and Static Support Stations, denoted *station*(s). Hosts and stations exclusively communicate through message passing. Hosts are the sources and destinations of application messages, and stations ensure that hosts receive and deliver messages causally. Applications running on hosts use a group communication service to join and leave the network, as well as to broadcast messages to all hosts and deliver messages in causal order.

The features of stations, hosts, and network are the following:

- Stations:
  - Each station is at the center of a cell, corresponding to the area covered by its antenna's transmission range.
  - Each station holds the causal information hosts connected to it require to deliver application messages in causal order.
  - Stations are supposed to be static. For the first algorithm, they are supposed to be reliable through hardware replication. For the second algorithm they can fail.
  - Stations have no energy limitations and have a much higher storage and computational capacity than hosts.
- Hosts:
  - A host is connected to at most one station (generally the closest one) at any given moment and communicates with the system through that station, by sending messages on the wireless network.

  - A host can join and leave the network at any moment. A host that joins the system will not deliver those application messages that the station to which it connects the first time has discarded prior to its connection.
  - Hosts move freely inside and outside cells.
  - Hosts have computational, storage, and energy limitations.
  - Hosts are subject to transient and permanent failures. For example, a host is temporarily faulty until its battery is recharged, or it is permanently faulty if it has a hardware failure. A faulty host stops sending, receiving, processing messages, and it loses all variables it stored in volatile memory.

    Hosts have two states: *up* and *down*. Station control the state of hosts connected to them: A station considers a host connected to it as *down* after not receiving any messages from the host for a given interval of time. Otherwise, the station considers the host as *up*.

- Network:
  - The wireless network is considered unreliable due to interferences. The bandwidth of the wireless network is much lower than the bandwidth of a wired network. Hosts communicate with stations exclusively through the wireless network.
  - The wired network is composed of FIFO communication channels without message losses. Nevertheless, wired channels can fail. Moreover, stations communicate over the wired network by using the algorithm proposed by Mostéfaoui [NMM18a]. Therefore, wired channels can be added and removed, as long as there exists a path of communication channels initialized by the algorithm [NMM18a].

Every application message is uniquely identified by (h_i, seq_h), where seq_h is the sequence number that host h_i attributes to the application message. Moreover, since cells may overlap, every message broadcasted over the wireless network piggybacks the ID of the cell in which it is broadcasted. Stations and hosts verify the cell ID of messages they receive over the wireless network, and only take into account those broadcasted inside their cell.

The *termination* condition of causal broadcast must be modified to adapt it to the dynamics of mobile networks. Consider s_k to be the first station which acknowledges the join of host h_i. The latter will not deliver the application messages that s_k discarded prior to receiving the join message of h_i. The termination condition of **causal broadcast** therefore becomes:
Termination. Consider s_{j,h_i} to be the first station which acknowledges to host h_i the system join of h_i. A message m co-broadcasted by an up host is co-delivered by all up hosts h_i for which s_{j,h_i} did not discard m prior to h_i's connection.

4.3 Causal broadcast with reliable stations

This section presents *WAS*, a causal broadcast algorithm which extends the FIFO dissemination approach [FM04] to mobile networks, where hosts move freely, are subject to failures, and communicate through unreliable wireless channels. Stations communicate with each other through reliable wired channels. *WAS* tolerates the failure of hosts and handles the move/join/leave operations of hosts.

The description of *WAS* is divided into three parts: (1) dissemination of application messages, (2) join/leave operations, and (3) handoff procedure to handle hosts moving between cells. The proof of the algorithm can be found in Section 4.6.

4.3.1 Dissemination of application messages

The FIFO dissemination approach achieves causal broadcast by ensuring that no path exists over which messages can travel out of causal order. Friedman and Manor [FM04] showed that ensuring that all communication channels are FIFO ordered is sufficient to ensure causal broadcast in static networks. However, in mobile networks new communication channels can be added. These communication channels create paths over which messages can temporarily take a shortcut and be received by processes out of causal order, as explained in Section 3.2.4. *WAS* provides a FIFO dissemination-based causal broadcast for mobile networks, by ensuring that those new paths are not used to disseminate messages as long as messages can take a shortcut and be received out of causal order by passing through them. This section explains how *WAS* ensures causal order in a static mobile network, while the following sections explain how *WAS* handles the dynamics of mobile networks.

In Mobile Networks, hosts are the source of application messages, and stations ensure that all hosts deliver application messages while respecting the causal relations between the application messages. We divide the network in two modules: the intra-cell module, composed of a station and hosts connected to it, and the inter-cell module, composed of all stations.

4.3.1.1 Intra-cell module

An intra-cell module, called a cell, is composed of a station s_i and the hosts connected to s_i. A Host of cell c_i communicates with the system by sending messages to s_i through the wireless network. The FIFO dissemination approach requires reliable FIFO channels. They are implemented on wireless communication channels by

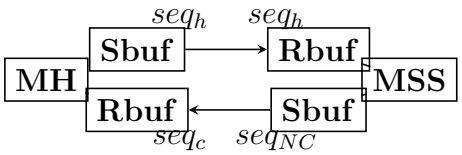


Figure 4.2: Dissemination of application messages in the intra-cell module

means of sequence numbers, retransmissions, and buffering until acknowledgment, as shown in Figure 4.2. A host uses the sequence number seq_h to order messages it sends, and its station keeps seq_h to order application messages it receives from the host. A station uses seq_C to order application messages it broadcasts, and hosts use seq_{NC} to order application messages they receive from their station. Both stations and hosts maintain two local buffers: one buffer to FIFO-order messages at reception (*Sbuf*), and another one to retransmit sent/broadcasted messages until they are acknowledged (*Rbuf*).

4.3.1.2 Inter-cell module

The inter-cell module is composed of all the stations and the wired communication channels connecting them. The FIFO dissemination of messages is ensured on the wired network with the algorithm proposed by Nédelec and all [NMM18a], which tolerates dynamics under certain conditions. The authors define initialized communication channels, which are either communication channels initially present, or communication channels which were initialized through the algorithm. The algorithm notably requires that there always exists a path of initialized communication channels between each pair of nodes. Hence, we assume that each pair of stations is always connected by a path of initialized communication channels. The algorithm also requires reliable FIFO communication channels. Wired communication channels reliability can be easily achieved through protocols like TCP. The algorithm presented by Nédelec and all [NMM18a] ensures that messages travel in causal order over the wired network, i.e., between stations in our case.

4.3.1.3 Message acknowledgment

Hosts and stations keep messages in their *SBuf* for retransmission. A host keeps an application message in its *Sbuf* until its cell's station acknowledged the message. A station of group G keeps an application message m in its *Sbuf* until, $\forall s_k \in G$, all hosts connected to s_k acknowledged m, i.e., m is acknowledged by all hosts connected to stations of G.

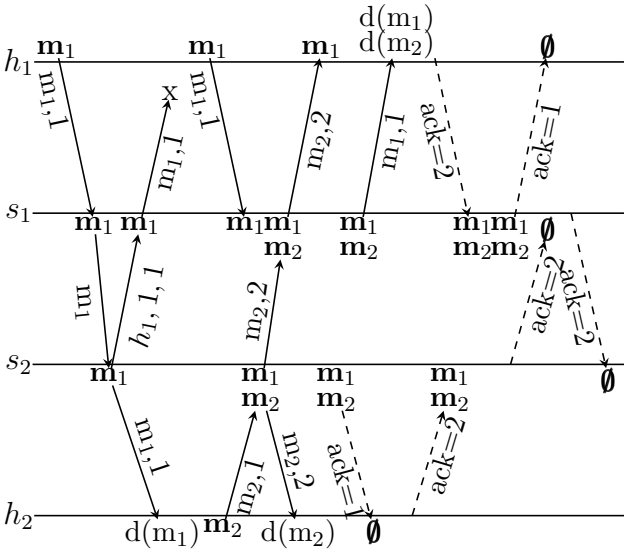


Figure 4.3: Broadcast of m_1 and m_2

A host regularly acknowledges application messages by broadcasting an acknowledge message containing a set of sequence numbers ranges $[seq_k, seq_{k+1}]$ which acknowledge the messages $m, m.seq \in [seq_k, seq_{k+1}]$. A station also regularly broadcasts such an acknowledge message in its cell to acknowledge application messages to hosts connected to it.

Stations should discard obsolete application messages from their *SBuf*. Existing discarding approaches are centralized, which induces a high message and memory overhead. *WAS* implements a decentralized discarding of obsolete messages, where each station deletes an application message m once all hosts connected to it have delivered m. Therefore, stations store fewer messages in *SBuf* when using the *WAS* algorithm.

Hosts do not send acknowledgments during handoffs, since they would acknowledge application messages received from another station, which might order application messages differently.

4.3.1.4 Broadcast example

Figure 4.13 shows the broadcast and delivery of two causally related application messages. Host h_1 (resp. h_2) is connected to station s_1 (resp. s_2), and s_1 and s_2 are connected by a wired channel. Hosts (resp. stations) piggybacks seq_h (resp. seq_C) on application messages broadcasted over the wireless network. The sending buffers *SBuffers* are represented in bold.

First, h_1 broadcasts m_1. Upon reception, s_1 attributes the sequence number $seq_C = 1$ to m_1, broadcasts m_1 in its cell, which contains h_1, and broadcasts m on the

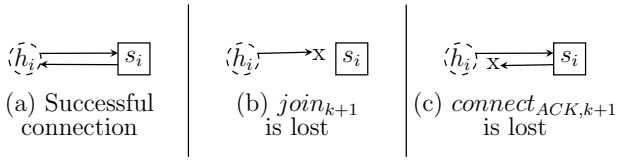
wired network to forward it to s_2. Upon reception, s_2 attributes the sequence number $seq_C = 1$ to m_1, broadcasts m_1 in its cell, which contains h_2, and also broadcasts m on the wired network. h_2 receives and delivers m_1, then broadcasts m_2 (co-broadcast(m_1) $\rightarrow$ co-broadcast(m_2)). s_2 receives m_2, attributes $seq_C = 2$ to it, then broadcasts m_2 on the wired network.h_2 (resp. s_2) stops transmitting m_1 (resp. m_1 and m_2) upon reception of the acknowledgment message regularly broadcasted by s_2 (resp. h_2). h_1 receives neither m_1 the first time s_1 broadcast it due to interferences, nor its acknowledgment. Hence, h_1 retransmits m_1. s_1 ignores the second reception of m_1 since it already received m_1. Upon reception of m_2, s_1 broadcasts it. Then h_1 receives and buffers m_2 because the sequence number that s_1 attached to m_2 is equal to 2, and h_1 awaits a message with seq=1. Eventually, s_1 broadcasts m_1 again and, upon reception, h_1 delivers m_1 then m_2. Finally, h_1 (resp. s_1) acknowledges m_1 and m_2 (resp. m_1). s_1 (resp. s_2) discards m_1 and m_2 from its sending buffer *SBuf* after h_1 (resp. h_2) acknowledged them. Hence, m_1 and m_2 are completely discarded from the network, i.e., removed from the buffers of all nodes.

4.3.2 Join/leave the network

WAS tolerates host churn during execution: a host can join and leave the network at any moment. Joining and leaving the network is not as immediate as it seems, because of the unreliability of the wireless network.

Joining the network. A host h_i joins the network by regularly sending a $join_{k+1}$ message to station s_i until s_i replies with the corresponding $connect_{ACK,k+1}$ message. The $connect_{ACK,k+1}$ message contains seq_C, the sequence number of the oldest message s_j caches in *SBuf*. Upon reception of the $connect_{ACK,k+1}$ message, h_i completes the connection initialization by updating seq_{NC} to seq_C. h_i can now begin to deliver application messages, based on seq_{NC}. Moreover, due to the unreliability of the wireless network, s_i must do some additional work to ensure that it is the only station that has an open connection with h_i.

In fact, several stations might receive the $join_{k+1}$ message of h_i, and each of these stations will then open a connection with h_i. When h_i sends a $join_{k+1}$ message to s_i through the unreliable wireless network, one of the three following scenarios presented in Figure 4.4 occurs: (1) both the $join_{k+1}$ and $connect_{ACK,k+1}$ messages are received; or interferences cause the loss of either (2) the $join_{k+1}$ message, or (3) the $connect_{ACK,k+1}$ message. Due to scenarios (2) and (3), h_i cannot decide if a station from which it receives no $connect_{ACK,k+1}$ has received its $join_{k+1}$ message or not. Moreover, h_i might move between several cells before it succeeds to connect to a station and thus join the system. h_i will then send $join_{k+1}$ messages to several stations. Therefore, several stations might receive and open a connection with h_i



(a) Successful connection | (b) $join_{k+1}$ is lost | (c) $connect_{ACK,k+1}$ is lost

Figure 4.4: Host connection scenarios

before h_i receives its first $connect_{ACK,}$ message. However, h_i should be connected to only one station, which is the one to which h_i sent the last $join_{k+1}$ message.

Stations handle such multiple connections by identifying each connection with a tuple (h_i, k), with h_i being the host that connects to the station and k being the k^{th} connection of h_i to a station. When s_i receives a $join_{k+1}$ message from h_i, it broadcasts a *Delete* message on the wired network containing the tuple $(h_i, k+1)$. Stations that maintain a connection (h_i, x) with $x < k+1$ unregister it upon reception of the *Delete* message. This ensures that eventually only the connection between s_i and h_i remains.

Leaving the network. Host h_i leaves the network by broadcasting a *leave* message, until a station, regardless of which one, acknowledges it. The station which receives the *leave* message broadcasts a *Delete* message, and stations that have a registered connection with h_i remove it upon reception of the *Delete* message.

4.3.3 Handoff procedure

The algorithm described in Section 4.3 ensures causal broadcast in networks where hosts always stay connected to the same station, i.e., in which hosts do not change cells. This section provides a handoff which enables hosts to change cells while delivering broadcast messages in causal order. Basically, when a host h_i changes the cell, by leaving the cell of station s_k, entering the cell of station s_{k+1} and connecting itself to station s_{k+1}, then s_{k+1} executes the handoff with s_k in order to ensure that h_i delivers messages causally. The handoff has three objectives:

(1). A station maintains the causal information related to hosts connected to it. Hence, when h_i changes cell, its causal information must be transferred from s_k to s_{k+1}.

(2). The handoff ensures that the wireless channel that h_i establishes with s_{k+1} when changing cell does not create a path over which h_i receives or sends messages out of causal order.

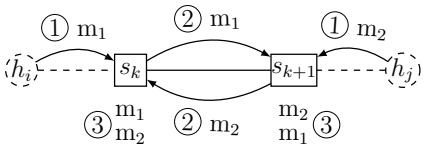


Figure 4.5: Sequence number assignation by stations

(3). The decentralized deletion of application messages by stations implies that s_{k+1} might already discarded some application messages that h_i did not deliver yet, and vice versa. s_{k+1} compares its cached application messages with s_k to ensure that h_i delivers each application message exactly once.

Note that the handoff of *WAS* makes no assumption on the movement of hosts, contrarily to some existing handoff procedures [CK04][ABS99][BB08] which make several unrealistic assumptions: they assume that a moving host always succeeds to connect to station s_k before connecting to station s_{k+1}, or/and that h_i is able to send a disconnect message to s_k. Both of these assumptions are not realistic because of the unreliability of the wireless network and the movement of hosts. *WAS* makes no assumptions on the movement of hosts, and tolerates simultaneous handoffs for the same host h_i, which might occur when h_i changes rapidly cells. The correctness of the handoff requires no assumption about the success of connection attempts. Moreover, failures of hosts and stations are tolerated, as well as the failure and addition of wired communication channels connecting stations.

We start the handoff section by discussing the challenges that the handoff procedure faces (handoff challenges), then we describe the handoff procedure algorithm itself, and finally, we give an example of handoff execution.

4.3.3.1 Handoff challenges

The three main challenges that handoff procedures faces are: host connections, the causal ordering of messages, and the discarding of obsolete application messages.

Host connections. When h_i sends a $connect_{k+1}$ message to a station s_i but receives no reply, then it cannot conclude whether s_i did or did not receive that $connect_{k+1}$ message, i.e., h_i cannot distinguish between the cases (b) and (c) of Figure 4.4. Because of such situations, h_i cannot always determine which station keeps its causal information. Furthermore, h_i might connect to several stations in a short time interval, and several of those stations might receive h_i's *connect* message, i.e., several stations execute a handoff for h_i simultaneously.

Therefore, a handoff for h_i must locate h_i's causal information, while tolerating simultaneous handoffs for h_i, and it must also ensure that the last station that started a handoff for h_i eventually maintains h_i's causal information.

Message ordering. The sequence number seq_{NC} that h_i uses to deliver application messages is only valid in the connection identified by Ses_{LC}, which stores the connection sequence number of the last connection in which h_i received a $connect_{ACK,}$ message. In fact, a station assigns a sequence number seq_C to application messages as it receives them. Hence, stations might order application messages differently, since they might receive application messages in a different order. For example, in Figure 4.5 the stations s_k and s_{k+1} order application messages differently: ① h_i (resp. h_2) broadcasts m_1 (resp. m_2). ② s_k (resp. s_{k+1}) receives m_1 (resp. m_2) and forwards it to s_{k+1} (resp. s_k). ③ s_k (resp. s_{k+1}) orders m_1, m_2 as *[m_1,m_2]* (resp. *[m_2,m_1]*). Consequently, if h_i delivers m_1 while connected to s_k and then connects itself to s_{k+1}, then s_{k+1} cannot assign to h_i the sequence number seq_{NC}*=1*, because h_i would then never deliver m_2 and deliver m_1 twice, i.e., seq_{NC} is valid at s_k but not at s_{k+1}. Hence, s_{k+1} must exchange messages with s_k to compare the ordering of application messages in their respective *SBuf*.

Discarding obsolete application messages. As seen above, s_{k+1} must compare the application messages it caches with s_k, because it might order them differently. However, a station discards an application message once all hosts connected to it have acknowledged the application message. This can lead to different *SBuf* states between s_{k+1} and s_k. For instance, in Figure 4.5, assume that h_i connects to s_{k+1} before delivering m_2, but that s_{k+1} already discarded m_2 when h_i connects to it. s_{k+1} must then recover m_2 from s_k, and h_i should deliver m_2 before delivering messages currently broadcasted by s_{k+1} (m_1). Moreover, if h_i already delivered m_1 when connected to s_k, then s_{k+1} must identify it.

However, stations do not maintain any information about messages they discard. For example, s_k cannot inform s_{k+1} that h_i already delivered m_1 if h_i delivers m_1 before connecting to s_{k+1}, because s_k does not keep any information about m_1.

Therefore, s_k and s_{k+1} must compare their cached application messages, while considering that they do not necessarily assign the same sequence number to application messages, as well as that they might respectively have some application messages that the other station has not received yet or that the other station has already discarded. They must do that without keeping information about discarded application messages.

Task of h_i :

Connecting to station s_i

broadcast(<$connect_{k+1}$,id_h,seq_{NC},k+1,Ses_{LC}>) to s_i
k = k+1

receive <$connect_{ACK,k+1}$,seq_h,seq_{NC},Ses>:

seq_{NC_i}=seq_{NC};seq_{h_i}=seq_h;Ses_{LC_i}=Ses;

Task of stations $s_i \in G_{s_{k+1}}$:

receive <$connect_{k+1}$,h_i,seq_{NC},k+1,Ses_{LC}> at s_i:

if $\nexists c \in Connections, c.id = (h_i, k+1) \wedge$ c not in handoff **then**
  Register(h_i)
  if c.s=id **then**
    broadcast(<connect,h_i,seq_{NC},Ses,Ses_{LC}>)
    broadcast(<$Req_{1,k+1}$,id_h,seq_{NC},Ses_{LC},Ses>)
else
  if Ses_{LC}=c.Ses **then**
    $c.seq_C = seq_C$
  c.Ses = Ses
  if c.Md = $\emptyset \wedge$ c.s = s_i **then**
    broadcast(<connect_{ACK},h_i,seq_{C_i},Ses>) to h_i
    broadcast(<Delete, h_i, Ses>) on the wired network

receive <Rsp_1,id_h,seq_h,m_{nd},Ses> for connection c at s_i:

if c.s = s_i **then**
  msg_{req}=message of m_{nd} that s_i has discarded
  broadcast(<$Req_{2,k+1}$,id_h,msg_{req},Ses>)

receive <Rsp_2,id_h,msg,msg_{rcv},Ses> for connection c at s_i:

$\forall m \in SBuf_{s_i}, h_i$ delivered m, m.Md = m.Md$\cup\{h_i\}$
if msg = $\emptyset$ **then**
  seq_S = min(m.seq, $\forall$m stored by s_i that h_i did not deliver)
  broadcast(<connect_{ACK},id_h,seq_S,Ses>)
else
  $\forall m \in msg$, broadcast(m) on the wireless network
if c.s = s_i **then**
  broadcast(<Delete,h_i,Ses>)

Task of s_k :

receive<$Req_{1,k+1}$,**id**$_h$,**seq**$_{NC}$,**Ses**$_{LC}$,**Ses**> **for connection c at** s_i**:**

1: **if** c.Ses < Ses∧ c in handoff **then**
2: **if** Ses_{LC} == c.Ses **then**
3: $\text{c.seq}_S = \text{seq}_{NC}$
4: m_{nd}={(id_h,seq_h) of messages h_i has not delivered}
5: c.Ses=Ses
6: **if** c.s = s_i **then**
7: broadcast(<Rsp_1,id_h,seq_{h_i},m_{nd},Ses>)

receive <$Req_{2,k+1}$,**id**$_h$,**msg**$_{req}$,**Ses**> **for connection c at** s_i**:**

8: msg= messages requested in msg_{req}
9: m_{rcv}= messages s_k received since receive($Req_{1,k+1}$)
10: broadcast(<Rsp_2,id_h,msg,m_{rcv},Ses>)
11: unregister(h_i)

4.3.3.2 Handoff description

The pseudo-code of the handoff procedure is given by the Tasks h_i, s_k, and s_{k+1}, which respectively give the pseudo-code executed by host h_i and the stations s_k and s_{k+1}. Figure 4.6 shows the handoff procedure, composed of four phases whose purpose are:

- Phase 1: h_i connects itself to s_{k+1} by broadcasting $connect_{k+1}$.
- Phase 2: s_{k+1} determines the application messages it discarded that h_i has not delivered. s_k and s_{k+1} exchange the messages $Req_{1,k+1}$ and $Rsp_{1,k+1}$ during this phase.
- Phase 3: s_{k+1} determines which, among the application messages it caches, h_i has not delivered. s_k and s_{k+1} exchange the messages $Req_{2,k+1}$ and $Rsp_{2,k+1}$ during this phase.
- Phase 4: Initialization of the connection between s_{k+1} and h_i.

The execution of handoff $H_{i,k+1}$ is said to be valid if:

- $H_{i,k+1}$ is aborted, which happens when before processing the handoff $H_{i,k+1}$, s_k processes a handoff $H_{i,k+n}$, with $n > 1$, until the end of Phase 3, i.e., handoff $H_{i,k+1}$ is aborted by a more recent handoff $H_{i,k+n}$.
- Exactly s_{k+1} and stations of $G_{s_{k+1}}$ eventually hold the causal information of h_i, provided that s_{k+1} does not fail till the end of Phase 3.

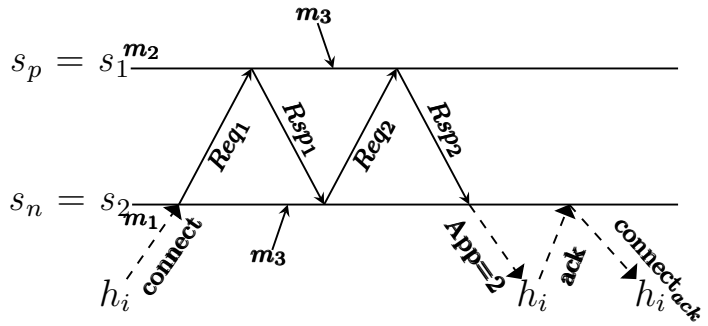


Figure 4.6: Handoff procedure

- Otherwise, exactly s_k and stations of G_{s_k} eventually hold the causal information of h_i if s_{k+1} fails before the end of Phase 3.

Phase 1. When h_i moves to the cell of s_{k+1}, it sends a $connect_{k+1}$ message to s_{k+1} that contains: the connection sequence number Ses, the connection sequence number of its last initialized connection Ses_{LC}, and the sequence number seq_{NC} of the last message it delivered.

Upon the reception of the $connect_{k+1}$ message from h_i, s_{k+1} verifies if it has already registered a connection with h_i(line $s_{k+1}.1$). s_{k+1} registers the connection with h_i if none is registered (lines $s_{k+1}.9$ – $s_{k+1}.10$) and starts the handoff procedure. On the other hand, h_i is already registered if either (1) (a) h_i is connected to s_{k+1} (b) h_i moves to another cell, but its $connect_{k+1}$ message is lost (Figure 4.4.b) (b) h_i changes cell and connects again to s_{k+1} ; (2) s_{k+1} already received the $connect_{k+1}$ message, but its $connect_{ACK,k+1}$ message was lost (Figure 4.4.c). In case (1) (if $Ses_{LC} = c.Ses$), seq_{NC} is valid for s_{k+1}. Hence, s_{k+1} uses it to acknowledge application messages that h_i might be delivered without acknowledging them. In both cases, due to the discarding mechanism of buffered application messages at stations, h_i might have to deliver some application messages that s_{k+1} has already discarded (we see later in this section how s_{k+1} determines which are those application messages). If h_i has no such application messages to deliver, then s_{k+1} sends a $connect_{ACK,k+1}$ message to initialize the connection on h_i's side ($s_{k+1}.6$).

Phase 2. We denote $M_{deleted}$ the set of messages that s_{k+1} discarded but that h_i has not delivered yet. The purpose of Phase 2 is to identify the messages of $M_{deleted}$.

s_{k+1} starts Phase 2 upon reception of the $connect_{k+1}$ message, by broadcasting a $Req_{1,k+1}$ message (line $s_{k+1}.12$).

Messages of $M_{deleted}$ are received by s_k prior to the $Req_{1,k+1}$ message. s_k uses seq_{NC} attached on the $Req_{1,k+1}$ message to build M_{ndel}, the set of application messages

not delivered by h_i among the application messages it received prior to the $Req_{1,k+1}$ message, and sends M_{ndel} to s_{k+1} in $Rsp_{1,k+1}$.

Upon reception of the $Rsp_{1,k+1}$ message, the network FIFO property ensured by the algorithm used for communication between stations [NMM18a] ensures that s_{k+1} received all messages contained in M_{ndel}. Therefore, the messages whose ID is contained in M_{ndel} but that s_{k+1} does not buffer are the messages of $M_{deleted}$.

s_{k+1} requests the message of $M_{deleted}$ in the $Req_{2,k+1}$ message. s_k piggybacks these messages onto the $Rsp_{2,k+1}$ message, by ordering them as in its *SBuf* (line s_k.10), i.e., by ordering them causally. Hence, s_{k+1} will recover those messages at the end of Phase 3.

Phase 3: We denote $M_{delivered}$ the set of messages that s_{k+1} caches but that h_i has already delivered. The purpose of Phase 3 is to identify at s_{k+1} the messages of $M_{delivered}$.

s_{k+1} starts Phase 3 upon reception of $Rsp_{1,k+1}$, by broadcasting $Req_{2,k+1}$(line s_{k+1}.15).

The application messages that h_i delivered prior to connecting to s_{k+1}, i.e., the application messages in $M_{delivered}$, are received by s_{k+1} prior to $Rsp_{1,k+1}$. We denote M_{cache} the set of application messages that s_{k+1} received prior to $Rsp_{1,k+1}$ and that it still caches. We also denote M_{ndel} the set of application messages $m \in M_{cache}$, h_i did not deliver m. We have $M_{cache} = M_{ndel} \cup M_{delivered}$. s_{k+1} determines $M_{delivered}$ by first determining M_{ndel} then deducing $M_{delivered} = M_{cache} \backslash M_{ndel}$.

The algorithm used for the communication between stations [NMM18a] ensures that application messages s_{k+1} receives before sending $Req_{1,k+1}$ that h_i has not delivered, are identified by s_k in the list $m_{nd} \in Rsp_{1,k+1}$ (line s_k.6). The algorithm used for the communication between stations [NMM18a] ensures that application messages s_{k+1} receives between send($Req_{1,k+1}$) and receive($Rsp_{1,k+1}$) are received by s_k between receive($Req_{1,k+1}$) and receive($Req_{2,k+1}$). s_k stores the list of ids of the application messages it receives between receive($Req_{1,k+1}$) and receive($Req_{2,k+1}$) in m_{rcv} and attaches m_{rcv} on $Rsp_{2,k+1}$(line s_k.11). Therefore, the application messages that h_i did not deliver among those s_{k+1} received before $Rsp_{1,k+1}$are the application messages $m \in m_{nd} \cup m_{rcv}$, i.e., $M_{ndel} = m_{nd} \cup m_{rcv}$. Therefore, we have $M_{delivered} = M_{cache} \backslash (m_{nd} \cup m_{rcv})$.

Phase 4: The purpose of Phase 4 is to initialize the connection between s_{k+1} and h_i.

First, s_{k+1} begins to send the application messages of $M_{deleted}$ to h_i, thus ensuring that h_i also delivers the application messages s_{k+1} has already discarded. Moreover,

s_{k+1} will not discard application messages it currently caches unless h_i delivered them, thus ensuring that h_i delivers every application message at least once.

Second, s_{k+1} ensures that h_i will not deliver twice any application message. To this end it tags the application messages that h_i already delivered (messages of $M_{delivered}$), because seq_{NC} is not sufficient. In fact, take the example of Figure 4.5. If h_i delivers m_1 but not m_2 before connecting to s_{k+1}, then s_{k+1} must give to h_i the sequence number $seq_{NC} = 0$, so that h_i will deliver m_2. However, without additional control, h_i would then also deliver m_1 again. To prevent multiple delivery of application messages by h_i, s_{k+1} adds h_i's ID to the application messages h_i already delivered but which s_{k+1} still caches in *SBuf* (line s_{k+1}.14), and, upon receiving them, h_i only increments seq_{NC} and does not deliver them again. Therefore, h_i does not deliver again an application message on which it is tagged, thus ensuring that h_i does not delivery any application message twice.

Thirdly, application messages of $M_{deleted}$ causally precede the application messages currently broadcasted by s_{k+1}. Therefore, h_i delivers no application message currently broadcasted by s_{k+1} unless receiving the $connect_{ACK,k+1}$ message from s_{k+1}, and s_{k+1} sends that $connect_{ACK,k+1}$ message to h_i only after that h_i acknowledged all messages of $M_{deleted}$ (or if it has no such messages to deliver) (line s_{k+1}.16). Moreover, s_{k+1} sends the application messages of $M_{deleted}$ in causal order as defined in msg of the $Rsp_{2,k+1}$ message, thus ensuring that h_i delivers them in causal order. Other application messages are delivered by h_i following the sequence number s_{k+1} attributes to them. Therefore, h_i delivers application messages in causal order.

The $connect_{ACK,k+1}$ message contains seq_{C_i}, the sequence number of the oldest application message that s_{k+1} buffers and that h_i has not delivered (line s_{k+1}.18). At its side, h_i ends the handoff by setting seq_{NC}, seq_h, and Ses_{LC} (line h_i.4).

Finally, s_{k+1} broadcasts a *Delete* message to end connections of h_i that might result from h_i's previous unsuccessful connection attempts (Figure 4.4(b)), thus ensuring that eventually h_i is only registered at s_{k+1}.

Simultaneous handoffs for h_i are handled sequentially: During handoff $Req_{1,k+1}$, s_k discards all handoff messages that do not belong to handoff $H_{i,k+1}$. A station s_{k+n} that starts a handoff $H_{i,k+n}$ regularly re-broadcasts its $Req_{1,k+n}$ message. Therefore, if $H_{i,k+n}$ is more recent than handoff $H_{i,k+1}$, then station s_{k+1} will eventually process the $Req_{1,k+n}$ once it receives it and that handoff $H_{i,k+1}$ is finished.

4.3.4 Handoff example

Figure 4.6 describes the handoff between s_k and s_{k+1} when h_i connects itself to s_{k+1}. The system configuration is as in Figure 4.5. For better readability, we assume that no other handoff for h_i executes simultaneously. At the beginning of the handoff, s_k has discarded m_1 and s_{k+1} has discarded m_2. h_i has broadcasted m_1 and h_j has broadcasted m_2. h_i has also delivered m_1. Both stations receive m_3 broadcasted by h_j during the handoff.

h_i connects itself to s_{k+1} by sending a $connect_{k+1}$ message to s_{k+1} containing seq_{NC}= *2*, since h_i has delivered m_1.

Upon reception of the $connect_{k+1}$ message, s_{k+1} broadcasts $Req_{1,k+1}$ containing seq_{NC}=2.

Upon reception of the $Req_{1,k+1}$ message, s_k determines that h_i has not delivered m_2. s_k broadcasts $Rsp_{1,k+1}$ containing the ID of m_2, as well as $seq_h = 2$ since h_i broadcasted m_1.

Upon reception of the $Rsp_{1,k+1}$ message, s_{k+1} requests m_2 in $Req_{2,k+1}$, because it has already discarded m_2.

Upon reception of the $Req_{2,k+1}$ message, s_k replies with $Rsp_{2,k+1}$ containing the application message m_2 that s_{k+1} requested in $Req_{2,k+1}$, as well as the ID of the message m_3 that s_k received between $Rsp_{1,k+1}$ and $Rsp_{2,k+1}$. Finally, s_k unregisters h_i.

Upon reception of the $Rsp_{2,k+1}$ message, s_{k+1} identifies which application messages of its $SBuf$=$\{m_1, m_3\}$ h_i has delivered. s_{k+1} received m_1 before $Rsp_{1,k+1}$, and s_k did not identify m_1 as not delivered by h_i. Hence, h_i already delivered m_1 and s_{k+1} attaches h_i's ID on m_1. In addition, h_i must first deliver m_2 before delivering m_3. Thus, s_{k+1} broadcasts m_2 received from s_k with seq_C=1 to h_i. Finally, s_{k+1} discards m_2 once h_i acknowledged it, assigns seq_{C_i}=3 to h_i, since h_i delivered m_1 and m_2, and broadcasts $connect_{ACK,k+1}$. Upon reception of $connect_{ACK,k+1}$, h_i sets seq_{NC}=3. $h_i$can now deliver m_3 as well as following application messages that s_{k+1} will broadcast.

4.3.5 Failure resilience

The algorithm tolerates the failure of hosts. Hosts can fail temporarily or permanently. The failure of hosts is handled at the station level.

A temporary failure of a host that exceeds a given time limit leads its station to disconnect the host, in order to limit the impact of the faulty host on the cell's

wireless network. In fact, a station broadcasts an application message until all hosts connected to it acknowledged the message. However, a host that is faulty obviously acknowledges no application message. Therefore, the station that registers the faulty host will eventually broadcast more and more application messages as it receives new ones to broadcast. However, the wireless network is a shared medium and has a maximal throughput because of interferences. Therefore, the duration of a temporary host failure should be bounded in time, because the application messages broadcasted due to the faulty host will eventually heavily impact the wireless network, to the point where it can even overload it. Thus, a station controls its cell's collision rate and unregisters a faulty host once the collision rate inside its cell exceeds a given limit. A station considers that a host is permanently faulty when it does not receive any message from the host during a given time interval, or if the message collision rate becomes too high due to the absence of acknowledgments from that host.

A host saves and restores few variables on persistent local storage to handle temporary failures: seq_h, seq_{NC}, Ses, and Ses_{LC}, and unacknowledged application messages broadcasted by the host before its failure. A host saves seq_h (resp., seq_{NC}) when broadcasting an application (resp., acknowledge) message, Ses when changing cell, and Ses_{LC} when the host receives the confirmation that the station to which it connects registered it. A host saves application messages when broadcasting them until they are acknowledged. Upon recovering, the host restores these variables and broadcasts a *recover* message to the station of its cell.

The station receiving the *recover* message might not hold the host's causal information, either because the host's failure duration was too long and the station discarded the host's causal information, or because the host moved during its failure. If the station still registers the host, it replies to the host with a $connect_{ACK}$, message. Otherwise, it broadcasts a $recovery_{req}$ message, to which each station replies either with the host's causal information, or a message that notifies that the station does not store the host's causal information. The host is reinitialized if no station maintains its causal information, and the end of the recovery procedure is then similar to the system join of a host. If a station maintains the host's causal information, then the recovery procedure is similar to a handoff.

Permanent failures are handled through timeouts. If a station does not receive any message from a host for a given duration, then the station considers that the host has a permanent failure and simply deletes its connection with the host.

Summing up, our algorithm tolerates permanent and transient failures of hosts, while requiring few persistent information.

4.4 Performance Evaluation

Experimental setup. Experiments were conducted on INET, a network simulator implemented on OMNeT++ [Var01]. INET offers an implementation of communication layers (e.g., TCP/UDP/Ethernet/IPv4/MAC), node mobility, node failures, and network interferences in wired and wireless networks.

We compare our algorithm, denoted *WAS*, with the one proposed by Chandra and Kshemkalyani [CK04], denoted *CK* and described in Chapter 3, which is the algorithm with the best performances among causal multicast algorithms in mobile networks [CK04]. We adapted *CK* to causal broadcast, by removing the structure log_i, since the structure $RECD_i$ is sufficient to track causal order in causal broadcastC. Second, [CK04] uses point-to-point communication on the wireless network. Nodes use the broadcast feature of UDP on the wireless network, i.e., they only send one broadcast message on the wireless network instead of doing point-to-point communications.

Stations are placed to ensure a complete area coverage with a minimum intersection of cells. Hosts are placed randomly in each cell at initialization. Antennas have a communication range of 120m and a bandwidth of 20Mb/s. Stations are connected by a wired network organized into a tree of degree 3. Wired links have a bandwidth of 100Mb/s and a delay of 10ms. Application messages have a size of 100 bytes and are encapsulated in IPv4/MAC packets, whose header has 8 (resp., 20) bytes for UDP (resp., TCP). Therefore, an UDP (resp., TCP) packet has an overhead of 20(IPv4)+20(MAC)+8(UDP)=48 (resp., 60) bytes. UDP is used for communication on the wireless network, while TCP on the wired network. Each host broadcasts application messages following a Poisson distribution. Hosts move in a straight line with a speed of 5km/h≈1.39m/s inside the area covered by stations, and change direction every 5 seconds.

The experiments aim to :

- Analyze the scalability of *WAS* and *CK* and compare them with each other.
- Compare the centralized and decentralized message discarding approaches.
- Analyze the behavior of *WAS* in a scenario with faulty hosts.

4.4.1 Scalability

Throughput and delivery delay. The first experiment, whose results are shown in Figure 4.7, evaluates the maximal throughput when the number of hosts per

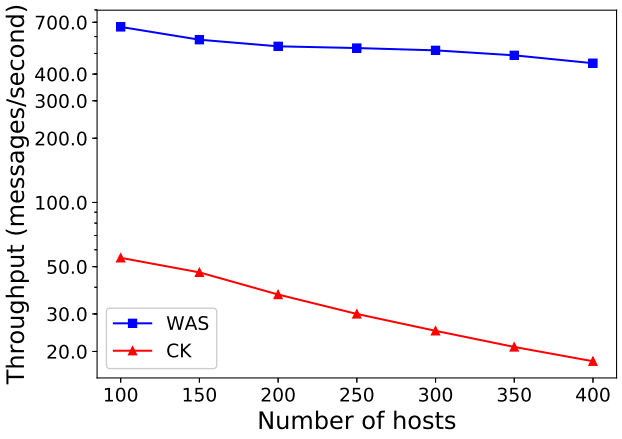


Figure 4.7: Throughput in function of hosts per cell

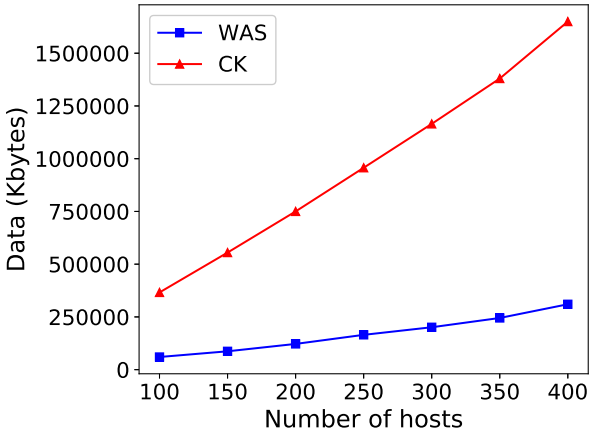


Figure 4.8: Sent data (Kbytes)

cell increases. The experiment contains 10 stations and a total number of hosts that varies from 100 to 400 (x-axis). Results, presented in a logarithmic scale, show that *WAS* has a much higher (x10-20) throughput than *CK*, and that the throughput of *CK* decreases faster than *WAS*. In a system containing 400 hosts, the maximal throughput of *WAS* is more than 20x higher than *CK*. The throughput of *CK* is bounded mostly by the fact that a station only sends an application message to a host once the latter has acknowledged all the message's dependencies. Consequently, hosts send acknowledge messages very frequently, which negatively impacts performance because of a higher message collision rate on the wireless network. Moreover, *CK* has a delivery delay - the delay between broadcast(m) and deliver(m) - 2 times higher than *WAS*, because a station waits that a host acknowledges a message's dependencies before sending it to the host.

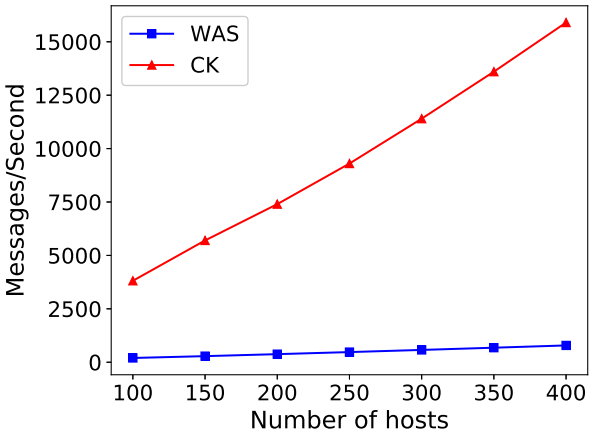


Figure 4.9: Messages sent over the wireless network

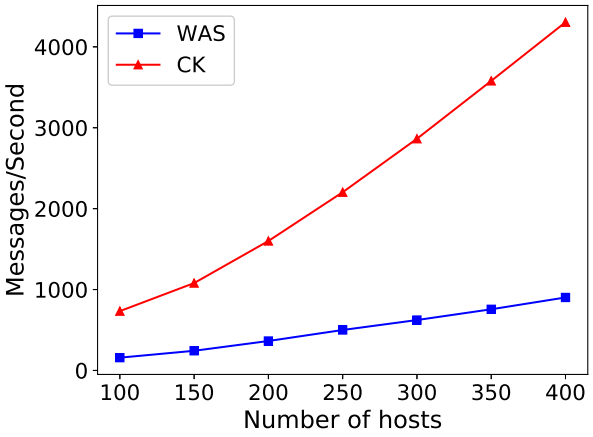


Figure 4.10: Messages sent over the wired network

The second experiment measures the size and number of messages, evaluated at the IPv4 level, when the number of hosts increases. Figure 4.9 and Figure 4.10 show the number of messages sent over the wireless and wired network respectively. Figure 4.8 shows the total amount of data sent in *WAS* and *CK* during experiments. The number of stations is adapted to keep a ratio of 20 hosts/cell. 20 messages are broadcasted per second in the system.

Number of sent messages. Figure 4.9 and Figure 4.10 show that *WAS* sends much fewer messages on both the wired and wireless networks. Moreover, the number of messages sent by *CK* increases much faster than the number of messages sent by *WAS*. *WAS* sends fewer messages on the wireless network because (1) hosts acknowledge messages less often than *CK* (20x), (2) hosts buffer messages at reception following seq_{NC} piggybacked on messages, contrarily to *CK* where hosts do not buffer messages since stations only send a message m to a host once the

host can causally deliver m. Hence, stations must retransmit messages less often than with *CK* (x5). On the wired network, *CK* sends more messages due to its centralized approach to discard obsolete messages: Every application message m from a host has an associated MSS_{init} station, and every acknowledgment related to m is forwarded by the host's cell's station to MSS_{init}, which, in its turn, broadcasts a *delete* message to all stations once it has received an acknowledgment related to m from every host. Hence, *CK* sends a lot of acknowledge and *delete* messages over the wired network. Moreover, those messages must travel along the wired network, often through several stations, until reaching the corresponding MSS_{init} station. *WAS* implements a decentralized mechanism which requires no message exchange to discard obsolete messages. We point out that *CK* sends much fewer messages on the wired network than theoretically expected, because acknowledge messages of *CK* are small, and TCP groups many of them in a single packet.

Amount of sent data. Figure 4.8 shows that *WAS* sends a lower amount of data than *CK*. On the wireless network, this is mostly due to acknowledge messages of *CK*. Even though these messages contain only a few integers, they have an additional size of 48 bytes because they are encapsulated in UDP/IPv4/Mac packets, and only a few acknowledge messages can be grouped into one single packet since the station will not send the next messages to deliver until the current ones are acknowledged. On the wired network, acknowledge and *delete* messages are grouped by TCP, which mostly removes the encapsulation overhead. However, those many acknowledge and *delete* messages scale up fast. Moreover, with *CK* stations piggyback a vector of size N (N=number of stations) on application messages sent over the wired network, and that vector rapidly takes much space when the number of stations increases. *WAS* only piggybacks a few integers on application messages.

To conclude, in terms of hosts per cell, total number of hosts (acknowledge and *delete* messages), and stations (size of vector clocks piggybacked on application messages), *WAS* scales much better than *CK*.

4.4.2 Decentralized discard mechanism

This section compares our decentralized discard approach, used by *WAS*, with the centralized discard approach [CK04].

In the third experiment, 200 hosts are distributed over 10 cells, the wireless network has a bitrate of 1Mb/s, and 35 messages are broadcasted per second for 300s. Figure 4.11 shows the number of messages that stations store in their respective sending buffers *SBuf*. Curve *Max* shows the maximum number of messages cached in a stations' *SBuf*, i.e., approximately the number of messages each station would store with a centralized discard approach. Curve *Avg* shows the average number

of messages a station stores in its sending buffer, and curve *Deviation* gives the standard deviation between the average and the number of messages each station caches. The curves *Avg* and *Deviation* do not take into account the *SBuf* of the station that stores the most messages, in order to compare both curves with the *Max* curve.

The comparison of curves *Avg* and *Max* of Figure 4.11 shows that the number of messages cached by stations can vary a lot. Such a variation depends on the message loss rate in the station's cell: the higher the message loss rate, the longer a station caches a message, since lost messages must be retransmitted. The message loss rate depends on the number of messages to broadcast as well as the position of hosts in the network. The probability of message collision is higher in areas where two cells overlap because the respective stations send messages over their cells that might collide. Similarly, areas with a high density of hosts have a higher message loss rate. The *standard deviation* is low, mostly around 10 messages, except for a short period around 70s where a heavily loaded cell degrades its adjacent cells. Hence, the number of messages a station stores in *SBuf* is mostly close to the average for all stations, except for some stations whose local characteristics make their send buffer grow temporarily. In a decentralized message discard approach, message loss rate and failing hosts only have a local impact. The comparison of curves *Avg* and *Max* shows that with a decentralized message discard approach, stations store up to 4 times fewer messages than with a centralized one, and that, on average, stations store 40-50% fewer messages.

Finally, a host that fails stops acknowledging messages. Hence, the station to which the host is connected will stop discarding application messages. Figure 4.12 shows that, in presence of a host failure, the station to which the faulty host is connected caches many more messages than the other stations (8-10x more). In the decentralized discard approach, only the station to which the faulty host is connected will cache all those messages. Hence, the decentralized discard approach caches up to 8-10 times fewer messages on average on stations when a host fails.

4.4.3 Transient host failures

The last experiment, whose results are presented in Figure 4.12 measures the impact of transient host failures on *WAS* in a system containing 10 stations and 200 hosts (20 hosts per station), a wireless network bitrate of 1Mb/s, and where 15 messages are broadcasted per second. The first host fails at t=10s for 5 seconds, then each 30 seconds another host fails, and the fault duration increases by 2 seconds at each failure. In total, 9 hosts fail, the first failing at t=10s for 5s, the second at t=40s for 7s, and so on. A host that fails it stops acknowledging messages. The number of messages cached in the *SBuf* of the station to which the faulty host is

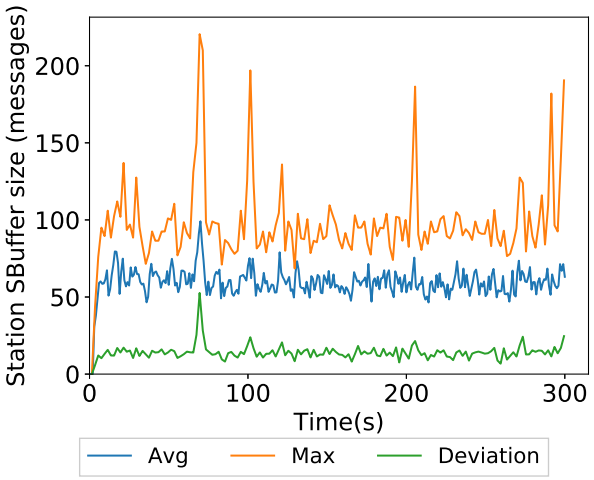


Figure 4.11: Messages in station sending buffers

connected then grows. Hence, we measure the impact of transient failures through the number of messages cached in the *SBuf* of that station.

Curve *Max* shows the maximum number of messages cached in a station's *SBuf* which is, during failures, the number of messages cached by the station at which the faulty host is registered. Curve *Avg* shows the average number of messages a station stores in its *SBuf*, and curve *Deviation* gives the standard deviation between the average and the number of messages each station caches. In order to evaluate the impact that a cell containing a faulty host has on the other cells, the former is not taken into account in the computation of *Avg* and *Deviation*. Vertical dashed lines represent a host crash.

During each failure, the number of application messages cached by the station to which the faulty host is connected linearly increases. Those application messages are also broadcasted by that station. Nevertheless, the number of cached application messages sharply decreases once the host recovers, showing that, very fast, the host receives the missing application messages and the cell rapidly reaches the same message load it had before the failure.

Curve *Avg* shows that, on average, a faulty host has a low impact in the number of messages stored by the other stations, except for the last failure occurring at t=255s.

Curve *Deviation* also shows that the increasing size of *SBuf* of the faulty host's station has no impact on other cells, as long as the *SBuf* does not become bigger than 150-200 messages. Once the *SBuf* exceeds that size, the faulty host's station begins to degrade adjacent cells that overlap with it. In fact, the station broadcasts application messages and retransmits messages not acknowledged by the faulty

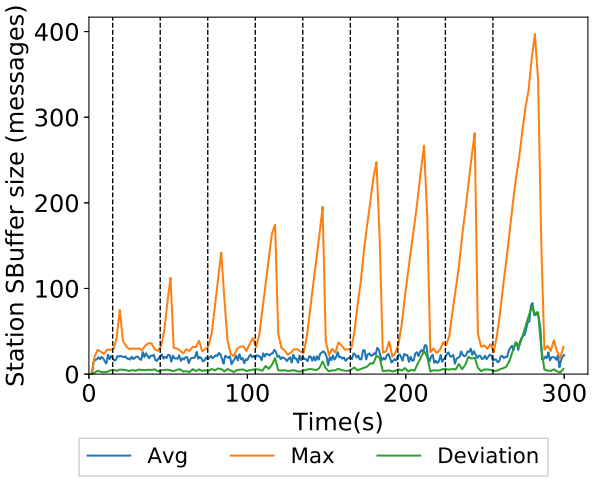


Figure 4.12: Messages cached by stations when hosts fail

host, thus increasing the number of messages sent by the station. This leads to an increasing number of message collisions in the cell, including the areas where the cell overlaps with other cells. Hosts in those overlapping areas, connected to other stations, will receive fewer messages, because of this higher message collision rate. Hence, they will deliver messages more slowly, increasing the number of messages broadcasted by the neighboring stations of the faulty host's cell. During the last crash occurring between t=250s and t=273s, the 3 adjacent cells of the faulty host's cell are impacted, explaining why the average size of the *SBuf* increases. Moreover, contrarily to previous failures where the faulty host takes less than a second to acknowledge messages, the host takes 7 seconds to acknowledge all messages. Therefore, the failure of a host first and mostly has an impact in the cell in which it occurs and afterwards in adjacent cells when its cell's station stores more than 150-200 messages. Nevertheless, such an impact rapidly disappears once the host recovers.

To conclude, experiments confirm that *WAS* scales much better than *CK* in terms of the number of hosts per cell, as well as in terms of total hosts and/or stations. *WAS* sends much fewer messages than *CK* both on the wired and wireless network, the amount of sent data is also much lower, and *WAS* has half the delivery delay of *CK*. Second, the decentralized message discard mechanism of *WAS* caches much fewer messages than *CK*, particularly during host failures. Finally, in *WAS*, the impact of a transient host failure is sharply absorbed after the faulty host recovers.

4.5 Extension to tolerate station failures

This section presents *WAS2*, an algorithm that extends *WAS* to make it tolerant to station failures. Several mechanisms are added to *WAS*, but most of the algorithm remains the same. Therefore, the following describes the modifications of *WAS* but does not describe *WAS2* completely. Each of the following subsections corresponds to the modifications added to the corresponding section of *WAS* in order to render it tolerant to station failures. For example, subsection *Model* corresponds to the modifications done in the section *Model* of *WAS*.

4.5.1 Model

Stations are subject to transient and permanent failures. A faulty station stops sending, receiving, and processing messages, and loses all data. To handle failures, stations are split into groups, and each station of a group G stores a replica of the causal information stored by the other stations of G. We define $f{=}|G|{-}1$ as the maximum number of stations of a group that can be down simultaneously.

WAS2 tolerates the failure of stations as long as the system satisfies the two following conditions :

- At most f stations of the same group fail simultaneously.
- Each pair of stations is connected by initialized links as defined in [NMM18a].

4.5.2 Dissemination of application messages

WAS2 does not modify the intra-cell module. The inter-cell module must however be modified in order to ensure that stations that belong to the same group order application messages identically.

WAS2 totally orders application messages inside each group of stations. Algorithm 7 describes the broadcast algorithm of messages. The stations of each group G elect a responsible station s_{resp} through a consensus algorithm. s_{resp} assigns an increasing sequence number seq_C to application messages as it receives them, and broadcasts their ID and seq_C on the wired network. A station delays the broadcast of an application message on its wireless network until it receives the message's sequence number seq_C through an App_{resp} message from s_{resp}.

In *WAS2*, station s_i saves a copy of the causal information of hosts connected to any station of G_{s_i}. Thus, s_i has a copy of the causal information of hosts connected

to any station of G_{s_i}, and this causal information can therefore be recovered at s_i when those other stations fail. s_i must update the causal information it stores about hosts connected to stations of G_{s_i} to keep it consistent. More particularly, it must keep track of the application messages broadcasted by hosts connected to G_{s_i}. To this end, whenever it receives an application message broadcasted by a host h_i connected to a station of G_{s_i}, it updates the seq_h associated to h_i. The algorithm used for communications on the wired network [NMM18a] ensures that s_i receives these application messages in causal order, i.e., it will receive the application message broadcasted by h_i of sequence number i before it receives the application message broadcasted by h_i of sequence number $i+1$. Moreover, [NMM18a] ensures that s_i eventually receives them and therefore updates the seq_h associated to h_i. Hence, s_i keeps track of the application messages broadcasted by hosts connected to stations of G_{s_i}.

Algorithm 7: Broadcast of application messages

broadcast at h_i **:**
1: $seq_h = seq_h + 1$
2: broadcast(<APP,h_i,seq_h>) to s_i

receive(<APP,h_i**,**seq_h**>) from** h_i **for connection** c **at** s_i**:**
3: waitUntil(c.seq$_h$ = seq_h)
4: broadcastApp(<APP,h_i,seq_h>)
5: c.seq$_h$ = c.seq$_h$+1

receive(<APP,h_i**,**seq_h**>) at** s_i**:**
5: broadcastApp(<APP,h_i,seq_h>)

broadcastApp(<APP,h_i**,**seq_h**>) at** s_i**:**
6: broadcast(<APP,h_i,seq_h>) on the wired network
7: **if** $s_i = s_{resp}$ **then**
8: $seq_C = seq_C + 1$
9: broadcast(<APP$_{resp}$,h_i,seq_h,seq_C>) on the wired network
10: **else**
11: waitUntil(receive<APP$_{resp}$, seq_C>) from s_{resp}
12: broadcast(<APP,seq_C>) on the wireless network
13: $\forall c \in Connections, c.host = h_i, c.seq = c.seq + 1$

4.5.2.1 Message acknowledgement

In *WAS2*, a station s_i continues to cache an application message m not only until all hosts of its cell delivered m, but until all hosts connected to any station of its group G_{s_i} have delivered m.

Therefore, a station s_i must acknowledge the delivery of application messages by host connected to it to the other stations of its group G_{s_i}. For this purpose, each

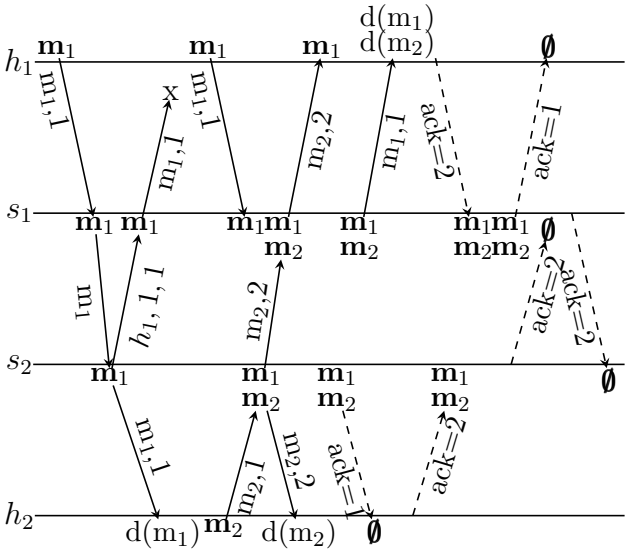


Figure 4.13: Broadcast of m_1 and m_2

station regularly sends acknowledge messages to the other stations of its group, to acknowledge application messages all hosts of its cell have delivered.

This new acknowledge mechanism introduces a new overhead, which nevertheless remains small. In fact, a station stops broadcasting an application message m once all hosts inside its cell have delivered m, and it then only keeps m for backup. Hence, stations can broadcast acknowledge messages only on a regular interval. Moreover, stations are supposed to be relatively reliable, and groups of stations should therefore be relatively small and contain only a few stations.

4.5.2.2 Broadcast example

Figure 4.13 shows the broadcast and delivery of two causally related application messages. The example is identical to the one in Section 4.3.1.4, except that the system uses *WAS2* instead of *WAS*.

Host h_1 (resp. h_2) is connected to station s_1 (resp. s_2), and s_1 and s_2 belong to the same group of stations G, and are connected by a wired channel. s_2 is the station responsible for the group G. Hosts (resp. stations) piggybacks seq_h (resp. seq_C) on application messages broadcasted over the wireless network. The sending buffers *SBufs* are represented in bold.

First, h_1 broadcasts m_1. Upon reception, s_1 forwards m_1 to s_2, and buffers m_1. Upon reception, s_2 attributes the sequence number $seq_C = 1$ to m_1, s_2 broadcasts m_1 in its cell, which contains h_2, and sends an APP_{resp} message to s_1 containing the $seq_C = 1$ it attributed to m_1. h_2 receives and delivers m_1, then broadcasts m_2 (co-broadcast(m_1) $\rightarrow$ co-broadcast(m_2)). s_2 receives m_2, attributes $seq_C = 2$ to it,

then broadcasts m_2 on the wired network with its attributed seq_C to s_1. h_2 (resp. s_2) stops transmitting m_1 (resp. m_1 and m_2) upon reception of the acknowledge message regularly broadcasted by s_2 (resp. h_2). h_1 does not receive neither m_1 the first time s_1 broadcast it due to interferences, nor its acknowledgment. Hence, h_1 retransmits m_1. s_1 ignores the second reception of m_1 since it already received m_1. Upon reception of m_2, s_1 broadcasts it. Then h_1 receives and buffers m_2 because the sequence number that s_1 attached to m_2 is equal to 2, and h_1 awaits a message with seq=1. Eventually, s_1 broadcasts m_1 again and, upon reception, h_1 delivers m_1 then m_2. Finally, h_1 (resp. s_1) acknowledges m_1 and m_2 (resp. m_1). s_1 (resp. s_2) discards m_1 and m_2 from its sending buffer *SBuf* after h_1 (resp. h_2) as well as s_2 (resp. s_1) acknowledged them. Hence, m_1 and m_2 are completely discarded from the network, i.e., removed from the buffers of all nodes.

4.5.3 Join/leave the network

Joining the network. Station s_i forwards $join_{k+1}$ messages it receives to other station of G_{s_i}, such that those stations also register the connection with h_i. Therefore, s_i broadcasts the $join_{k+1}$ message on the wired network and stations of G_{s_i} register the connection with h_i upon receiving the $join_{k+1}$ message. Otherwise, The join procedure does not change.

Leaving the network. The leave procedure does not change in *WAS2*.

4.5.4 Handoff

WAS2 has the same core handoff module than *WAS*, with some additional steps required to handle the failure of stations. During handoff $H_{i,k+1}$ between the stations s_k and s_{k+1} for host h_i, either s_k or s_{k+1} might fail.

Handoff termination. *WAS2* ensures that every handoff eventually terminates. If s_{k+1} fails, handoff $H_{i,k+1}$ is eventually cancelled; if s_k fails, s_{k+1} eventually reinitialize the handoff.

If s_{k+1} fails, then it obviously stops sending handoff messages to s_k. Upon expiration of a timeout, s_k then concludes that s_{k+1} failed and will therefore cancel the handoff. h_i will eventually connect to another station or it will reconnect itself to s_{k+1} after that s_{k+1} recovered, and s_{k+1} will then begin a new handoff.

If s_k fails, then it will obviously stop replying to handoff messages sent by s_{k+1}. Upon expiration of a timeout, s_{k+1} then concludes that s_k failed. It does then reinitialize the handoff by broadcasting a $Req_{1,k+1}$ message. Other stations of G_{s_k}

eventually also conclude that s_k failed, and one of them will then take up the responsibility of h_i. This station, denoted s_b will eventually receive the $Req_{1,k+1}$ message re-broadcasted regularly by s_{k+1}, and will then execute the handoff with s_{k+1}. Note that we make the assumption that upon detecting the failure of s_k, station s_b received all messages broadcasted by s_k before failing. Therefore, it will have received all application messages broadcasted by h_i, and seq_h associated to h_i will thus be up-to-date. If s_b fails, then another station will take up the responsibility of h_i, and so one.

Replication of h_i's causal information At the end of Phase 3, i.e. after that s_{k+1} processed the $Rsp_{2,k+1}$ message, only s_{k+1} has the causal information of h_i. Hence, it must propagate that information to the stations of $G_{s_{k+1}}$. To this end, it sends a $register_{k+1}$ message that contains all the causal information of h_i to stations of $G_{s_{k+1}}$. Stations of $G_{s_{k+1}}$ replicate the connection information between h_i and s_{k+1} upon reception of that $register_{k+1}$ message. The algorithm used for communication between stations [NMM18a] ensures that when a station s_b receives the $register_{k+1}$ message, this message contains all the causal information about h_i that s_{k+1} collected prior to sending the $register_{k+1}$ message. Therefore, the $register_{k+1}$ message contains all the relevant causal information that s_b received prior to the $register_{k+1}$ message, and s_b therefore handles messages containing causal information for h_i as it receives them after receiving the $register_{k+1}$ message.

No causal information is lost. Process s_{k+1} broadcasts a *delete* message only at the end of Phase 3 after having broadcasted a $register_{k+1}$ message. Thus, s_{k+1} will not broadcast a *delete* message without broadcasting a $register_{k+1}$ message, i.e. stations of G_{s_k} will not delete the connection with h_i unless stations of $G_{s_{k+1}}$ will register it. Consequently, it is ensured that the causal information of h_i is not lost.

Recovering. In order to recover, a station s_i must:

1. Connect itself to the wired network while not creating a path over which messages can travel out of causal order.
2. Recover the causal information of hosts connected to a station of its group.
3. Recover the application messages broadcasted by stations of its group.

Station s_i satisfies condition **(1)** in two steps: first it initializes a communication channel with one up station s_b of G_{s_i}, then in the second step it initializes its other communication channels through that first initialized communication channel. The former, denoted ch_i, is a FIFO wired communication channel whose endpoints are s_i and a station s_b of G_{s_i}. Both stations s_i and s_b initialize ch_i by exchanging

a $recovery_k$ and a $recovery_{ACK,k}$ message through ch_i. s_i begins by sending the $recovery_k$ message to s_b, which replies with a $recovery_{ACK,k}$ message containing the causal information locally stored at s_b upon its reception of the $recovery_k$ message. s_b considers ch_i as initialized after sending the $recovery_{ACK,k}$, and s_i considers ch_i as initialized after receiving the $recovery_{ACK,k}$ message. s_i discards messages it receives prior to the $recovery_{ACK,k}$ message and handles normally messages it receives afterwards. After initializing ch_i, s_i is connected to every other station through a path of initialized communication channels, and does therefore initialize its other communication channels by using the algorithm proposed by Mostéfaoui [NMM18a].

Conditions **(2)** and **(3)** are satisfied by s_i by recovering the information in the $recovery_{ACK,k}$ message: s_b appends to the $recovery_{ACK,k}$ message the causal information it stores about hosts connected to stations of G_{s_i} as well as the application messages (and their associated seq_{NC}) it stores locally. Moreover, s_b forwards each message it receives after sending the $recovery_{ACK,k}$ message to s_i. Upon reception of the $recovery_{ACK,k}$ message, s_i caches in its sending buffer the application messages contained in the $recovery_{ACK,k}$ message with their associated seq_{NC} number, and registers the causal information contained in the $recovery_{ACK,k}$ message. Therefore, s_i receives the information stored by s_b until the sending of the $recovery_{ACK,k}$ message as well as the messages s_b receives after sending the $recovery_{ACK,k}$ message. Hence, it will receive both all application messages that stations of G_{s_i} store and the messages required to ensure the causal information of hosts connected to stations of G_{s_i}.

Finally, s_i choses a new identifier when recovering, because some of the messages it sent before failing, or replies to some of those messages, might still be in transit over the network. By choosing a new identifier, s_i ensures that it will not handle any of those messages. s_i is in the up state after receiving the $recovery_{ACK,k}$ message.

4.5.5 Summary

This section presented *WAS2*, an extension of *WAS* that tolerates the failure of stations in mobile networks. *WAS2* has an overhead in complexity and messages. Stations are organized into groups to replicate the causal information they contain. Those of the same group exchange information to order both application messages identically, and to behave as backup for the causal information stored in other stations of the group. *WAS2* adapts the handoff to tolerate the failure of stations, and especially the failure of the two stations that take part in the handoff. Finally, *WAS2* provides a procedure that enables stations to recover from failures.

4.6 Proof

For the following proofs, we define the variables:

- h_i: host
- s_i: station that holds the causal information of h_i.
- $G_{s_{s_i}}$: group of stations to which s_i belongs.
- G: set containing all groups of stations.
- S: set containing all stations.
- H: set containing all hosts.
- s_{resp_i}: station responsible for the ordering of application messages for stations of G_{S_i}
- s_j: first station with which $h_i$has an up-connection.
- M_{prev}: set of messages discarded prior to the first up-connection of h_i.

Proof when nodes do not move between cells

Lemma 4.1. *WAS ensures that,* $\forall G_k \in G, \exists s_k \in G_k, s_k$ *is up* $\Rightarrow \forall h_i \in H, \exists s_i \in S,$ s_i *holds the causal information of* h_i.

Proof. Stations disseminate messages among each other by using the algorithm presented by Nédelec et al. in [NMM18a], ensuring that messages are causally ordered upon their reception at stations. Station s_{resp_i} assigns an increasing sequence number to each message m following its arrival time, and all stations of G_{S_i} assign to m that sequence number, i.e. all stations of G_{S_i} assign the same sequence number to m. Moreover, all stations of G_{S_i} have a copy of the causal information relative to h_i: s_i sends them the *connect* message, and they update the number of messages $h_i$broadcasted when receiving a message that $h_i$broadcasts.

$h_i$delivers messages from s_i following their sequence number, and saves the sequence number of the last delivered message on persistent storage. Hence $h_i$delivers messages from s_i in causal order. If s_i fails, then other up-stations of G_{S_i} still order the application messages as s_i, and they also store the causal information relative to h_i. When s_i recovers it recuperates the causal information about $h_i$by sending a recover message to s_{resp_i}, and $h_i$can then continue to deliver messages in causal order.

□

Theorem 4.1. *WAS ensures causal broadcast in mobile networks when hosts do not move between cells.*

Proof. Validity. A station disseminates messages either co-broadcasted by hosts from its cell, or received from other stations. Stations do not generate broadcast messages. Hence, stations only disseminate messages co-broadcasted by hosts. Moreover, a host only co-delivers messages that the station to which it is connected broadcasts. Hence, hosts only co-deliver messages co-broadcasted by hosts.
Integrity. First, hosts assign an increasing sequence number to each message they broadcast. A station keeps track of that sequence number for each host connected to it, and discards a message m broadcasted by $h_i$of sequence number seq, if the stored sequence number seq_{h_i} for $h_i$is $seq_{h_i} > seq$, thus avoiding that it broadcasts m twice. Following *Lemma 4.1*, that information is maintained as long as, for each group $G_S \in G, \exists s_k$ such that s_k is up.

Second, stations disseminate messages among each other by using the algorithm presented by Nédelec et al. in [NMM18a], ensuring that each station disseminates each message exactly once. Station s_{resp_i} assigns an increasing sequence number to each message m following its arrival time, and all stations of G_{S_i} assign to m that sequence number, i.e. all stations of G_{S_i} assign the same sequence number to m. $h_i$delivers messages from s_i in increasing sequence number. Thus, $h_i$will not deliver a message m again, since its local sequence number will always be greater than the one attached to m. To handle failures, $h_i$saves the sequence number of the last message it delivered on persistent storage. If s_i fails, then following *Lemma 4.1*, at least another up-station of G_{S_I} has the causal information relative to h_i, and up-stations of G_{S_i} order the application messages as s_i. When s_i recovers it will retrieve that information by sending a recover request to s_{resp_i}.
Causal order. Stations disseminate messages among each other by using the algorithm presented by Nédelec et al. in [NMM18a], ensuring that messages are causally ordered upon their reception at stations. Station s_{resp_i} assigns an increasing sequence number to each message m following its arrival time, and all stations of G_{S_i} assign to m that sequence number, i.e. all stations of G_{S_i} assign the same sequence number to m. Hence, if $m \rightarrow m'$, then s_{resp_i} receives m before m', and, therefore, $m.seq < m'.seq$. $h_i$only delivers messages broadcasted by s_i, and it delivers them in increasing sequence number. Hence, $h_i$delivers messages in causal order.

To handle host failures, $h_i$saves the sequence number of the last message it delivered on persistent storage, so it can determine which message it has not delivered when recovering. If s_i fails, then other up-stations of G_{S_i} still order the application messages as s_i and, following *Lemma 4.1*, they also have a copy of the causal

information relative to h_i. When s_i recovers, it will retrieve that information by sending a recover request to s_{resp_i}.
Termination. A host $h_i$that joins the system is not considered up until the station s_i to which $h_i$connects itself sends a $connect_{ACK}$ message to $h_i$and $h_i$receives it. Upon reception of the *join* message, s_i attributes to $h_i$the sequence number of the application message it buffers with the lowest sequence number, and forwards the join message to the other stations of G_{S_i}. Stations of G_{S_i} only delete a message m once it is acknowledged by all stations of G_{S_i}, i.e., once all hosts connected to a station of G_{S_i} have delivered m. Hence, even if s_i has a temporary failure, other stations of G_{S_i} will keep m, and when s_i recovers it will recover those messages and broadcast them until $h_i$has acknowledged (and henceforth delivered) them. Moreover, s_i retranmits the join acknowledgment as well as every buffered application message until $h_i$acknowledged each of them (i.e., after having delivered it), or s_i considers $h_i$as down. Hence, all messages that s_i did not discarded upon the reception of h_i's join message will be delivered by h_i, given that $h_i$remains an up process. □

Proof when nodes move between cells

In this section we prove that *WAS* ensures causal order when hosts move between cells without simultaneous handoffs for the same host, i.e., a host does not move to a new cell c_{k+1} unless it succeeded to connect to the station of its current cell c_k.

Lemma 4.2. *During handoff $H_{i,k+1}$, station s_{k+1} recovers the messages it discarded that h_i has not delivered and that are not in $M_{prev_{h_i}}$.*

Proof. We prove it by induction.
H0: The first station s_j to which h_i connects itself successfully has no application message to recover.
By definition, a host is up when joining the system once it received a $connect_{ACK}$ message from a station s_j to which it connects. Station s_j discards no application message after sending the $connect_{ACK}$ message to h_i, unless h_i acknowledges that message or connects to another station. Hence, s_j deletes no application message after sending $connect_{ACK}$ to h_i unless h_i acknowledges that message or connects to another station, and has, therefore, no application message to recover.

Stations of G_{s_j} also delete no application messages unless s_j acknowledges them. Hence, if s_j temporarily fails, then it will recover those messages during its recovery.

H1: When an up-host h_i connects itself to a station s_{k+1}, then s_{k+1} recovers those application messages it has already discarded, that h_i has not delivered, and that are not in $M_{prev_{h_i}}$.

Let's denote M_{recov} the set containing the application messages that s_{k+1} must recover.

s_{k+1} receives messages of M_{recov} prior to receiving the $connect_{k+1}$ message, since it discards no application message after receiving the $connect_{k+1}$ message unless that h_i acknowledges it.

In its turn, s_k receives the messages of M_{recov} before receiving the $Req_{1,k+1}$ message, since s_{k+1} broadcasts the $Req_{1,k+1}$ message after receiving the $connect_{k+1}$ message, and that the algorithm used for wired communication [NMM18a] ensures that communications on the wired network are FIFO ordered.

Upon reception of the $Req_{1,k+1}$ message, s_k which holds h_i's causal information, determines which messages h_i did not deliver among the messages s_k received prior to the $Req_{1,k+1}$ message, and attaches the list l_d of ids of those messages to $Rsp_{1,k+1}$. Therefore, the id of messages of M_{recov} are contained in l_d.

Upon reception of the $Rsp_{1,k+1}$ message, s_{k+1} determines M_{recov} by determining which messages whose id is in l_d it does not buffer. s_{k+1} then requests the messages of M_{recov} to s_k in the $Req_{2,k+1}$ message, and s_k sends them to s_{k+1} in the $Rsp_{2,k+1}$ message.

Hence, s_{k+1} recovered all messages of M_{recov} upon the reception of the $Rsp_{2,k+1}$ message.

If s_{k+1} fails, then the handoff is simply aborted at s_k's side upon timeout expiration. If s_{k+1} recovers, it will get a new id, and if h_i tries to reconnect to s_{k+1} after s_{k+1}'s recovery, then s_{k+1} will simply start another handoff.

Hence, when an up-host connects itself to a station, that station recovers the application messages it has discarded prior to the connection of the host (H1). Since the first station to which the up-host connects successfully has no message to recover (H0), we conclude that when a up-host connects itself to a station, then that station recovers the messages it discarded but that the up-host has not delivered.

□

Lemma 4.3. *Upon reception of the $Req_{1,k+1}$ message, s_k received all application messages that h_i has delivered prior to connecting to s_{k+1}.*

Proof. We prove it by induction.
H0: h_i only delivers application messages it receives from stations to which it connected successfully.

This is true by definition.

H1: We assume that upon reception of the $Req_{1,k}$ message, s_{k-1} received all application messages that h_i delivered prior to connecting to s_k. We show that upon reception of the $Req_{1,k+1}$ message, s_k received all application messages delivered by h_i prior to connecting to s_{k+1}.

s_k received the application messages delivered by h_i prior to connecting to it. By hypothesis, s_{k-1} received those application messages upon reception of the $Req_{1,k}$ message. Moreover, the algorithm used for communication between stations ensures that s_k received those application messages upon receiving the $Rsp_{1,k}$ message. Since by definition handoff $H_{i,k}$ is finished, s_k received the $Rsp_{1,k}$ message and has therefore received those application messages.

s_k received the application messages delivered by h_i while connected to it upon reception of the $Req_{1,k+1}$ message. In fact, s_{k+1} only sends the $Req_{1,k+1}$ message after receiving the $connect_{k+1}$ message, and h_i stops delivering messages from s_k after sending the $connect_{k+1}$ message. Therefore, h_i will not deliver any message from s_k that s_k receives after $Req_{1,k+1}$. Moreover, s_k obviously received the application messages that h_i delivered while connected to it.

Therefore, upon reception of the $Req_{1,k+1}$ message, s_k received all application messages that h_i delivered prior to connecting to s_k, as well as the application messages that h_i delivered while connected to s_k. Hence, upon reception of the $Req_{1,k+1}$ message, s_k received all application messages that h_i delivered prior to connecting to s_{k+1}.

Hence, upon reception of the $Req_{1,k+1}$ message, s_k received all application messages that h_i has delivered prior to connecting to s_{k+1} (H1). Since a host delivered no application message before connecting successfully to a station (H0), we conclude that upon reception of the $Req_{1,k+1}$ message, s_k received all application messages that h_i has delivered prior to connecting to s_{k+1}.

□

Lemma 4.4. *Upon receiving the $Rsp_{1,k+1}$ message, s_{k+1} received all application messages that h_i delivered before connecting to it.*

Proof. Following *Lemma 4.3*, s_k received all those messages upon receiving the $Req_{1,k+1}$ message. Moreover, s_k replies to the $Req_{1,k+1}$ message with the $Rsp_{1,k+1}$ message, and the algorithm used for the communication between stations [NMM18a]

ensures that, upon receiving the $Rsp_{1,k+1}$ message, s_{k+1} received all application messages that s_k received before broadcasting the $Rsp_{1,k+1}$ message. Therefore, upon receiving the $Rsp_{1,k+1}$ message, s_{k+1} received all messages delivered by h_i prior to connecting to it.

□

Lemma 4.5. *Application messages that h_i did not deliver among those s_{k+1} receives prior to the $Rsp_{1,k+1}$ message are either identified by s_k as not delivered by h_i ($m_{nd} \in Rsp_{1,k+1}$) or received by s_k between the $Req_{1,k+1}$ and $Req_{2,k+1}$ messages ($msg_{rcv} \in Rsp_{2,k+1}$).*

Proof. The application messages received by s_{k+1} before the $Rsp_{1,k+1}$ message are received by s_k before the $Req_{2,k+1}$ message. In fact, s_{k+1} broadcasts the $Req_{2,k+1}$ message after receiving the $Rsp_{1,k+1}$ message, and the algorithm used for communication between stations [NMM18a] ensures that s_k receives the $Req_{2,k+1}$ message after receiving the messages that s_{k+1} received before broadcasting the $Req_{2,k+1}$ message.

Among the application messages that s_k receives prior to the $Req_{1,k+1}$ message, the id of those not delivered by h_i are contained in m_{nd}.

Among the application messages that s_k receives between the $Req_{1,k+1}$ and $Req_{2,k+1}$ message, none of them are delivered by h_i. In fact, following the corollary of *Lemma 4.3*, h_i delivered no application messages that s_k receives after $Req_{1,k+1}$. The list msg_{rcv} contains the ids of application messages that s_k receives between the $Req_{1,k+1}$ and $Req_{2,k+1}$ messages.

Therefore, the list $m_{nd} \cup m_{recv}$ contains the list of ids of application messages that h_i did not deliver among the application messages that s_{k+1} received prior to the $Rsp_{1,k+1}$ message. □

Lemma 4.6. *Application messages that h_i did not deliver prior to handoff $H_{i,k+1}$ are those received by s_{k+1} prior to the $Rsp_{1,k+1}$ message and whose id is contained in $m_{nd} \cup msg_{rcv}$ ($m_{nd} \in Rsp_{1,k+1}, msg_{rcv} \in Rsp_{2,k+1}$), or those that s_{k+1} receives after the $Rsp_{1,k+1}$ message.*

Proof. Following *Lemma 4.5*, messages that h_i has not delivered among those s_{k+1} receives prior to the $Rsp_{1,k+1}$ message are those contained in $m_{nd} \cup msg_{rcv}$. Following the corollary of *Lemma 4.4*, h_i did not deliver any message that s_{k+1} receives after the $Rsp_{1,k+1}$ message. □

Lemma 4.7. *h_i must first deliver the application messages of $msg \in Rsp_{1,k+1}$ to ensure that it delivers application messages in causal order.*

Proof. The algorithm used for the communication between stations [NMM18a] ensures that stations receive messages in causal order. Moreover, stations order application messages following their reception order, i.e. if a station receives m_1 before m_2, then $seq_{m_1} < seq_{m_2}$.

Stations discard application messages in increasing sequence order, i.e. each application message that s_{k+1} buffers has a greater sequence number than application messages that s_{k+1} has discarded. Hence, h_i must first deliver the application messages that s_{k+1} discarded prior to delivering application messages that s_{k+1} currently broadcasts. Since s_{k+1} discarded the application messages $m \in msg$, h_i must first deliver them. □

Theorem 4.2. *WAS ensures causal broadcast in mobile networks where hosts move between cells.*

Proof. Following *Theorem 5.3 WAS* ensures causal broadcast in mobile networks where hosts do not move between cells, i.e., where a host always stays connected to the same station. We show that the handoff procedure of *WAS* ensures that the causal information of a host h_i that moves from cell c_i to cell c_{k+1} is transferred to the station s_{k+1} in charge of cell c_{k+1}, and that s_{k+1} therefore continues to ensure causal broadcast for h_i.

Validity. The validity of causal broadcast does not change when hosts move between cells. Hence, the validity proof of *Theorem 5.3* holds.

Integrity. Following *Theorem 5.3*, hosts co-deliver no application message twice when not changing cell. We show that when host h_i moves from cell c_k to cell c_{k+1} and connects itself to station s_{k+1}, s_{k+1} identifies the application messages that h_i already delivered, and that h_i will not deliver them again.

Following the corollary of *Lemma 4.6*, application messages that s_{k+1} buffers or receive that h_i has delivered are the application messages m that s_{k+1} receives prior to the $Rsp_{1,k+1}$ message such that $m \notin m_{nd} \cup msg_{rcv}$. s_{k+1} and the other stations of $G_{s_{k+1}}$ piggyback h_i's id on those application messages when broadcasting them, and h_i does not deliver them upon reception.

Following *Lemma 4.1*, if s_{k+1} (resp. s_k) fails, the causal information of h_i is maintained as long as a station of $G_{s_{k+1}}$ (resp. G_{s_k}) is up. Hence, the application messages that h_i already delivered are eventually identified, even if s_{k+1} or s_k fail.

Therefore, h_i will not deliver an application message again which it has already delivered, i.e., h_i delivers no application message twice.

Causal order. Following *Theorem 5.3*, *WAS* ensures causal order when hosts do not change cell. We show that handoff $H_{i,k+1}$ ensures that will deliver h_i messages in causal order when connecting to station s_{k+1}.

First we show that messages broadcasted by h_i are not disseminated out of causal order. The causal dependencies of application messages broadcasted by h_i are the application messages delivered by h_i. Following *Lemma 4.4*, s_{k+1} received those application message prior to the $Rsp_{2,k+1}$ message, i.e. before the end of handoff $H_{i,k+1}$. Since s_{k+1} broadcasts no application message from h_i prior to the end of handoff $H_{i,k+1}$, no application message m of h_i is disseminated by s_{k+1} before that s_{k+1} receives one of m's causal dependencies. Therefore, application messages of h_i are disseminated in causal order.

Second, we show that s_{k+1} identifies the application messages that $h_i$must deliver and sends them in causal order to h_i.

To begin, we show that h_i delivers in causal order application messages that s_{k+1} receives prior to the $Rsp_{1,k+1}$ message. Following *Lemma 4.7*, h_i must first deliver the application messages it has not delivered and that s_{k+1} discarded prior to h_i's connection. Following *Lemma 4.2*, s_{k+1} recovers those application messages through s_k which orders them in causal order. s_{k+1} sends them to in causal order to h_i, and h_i first delivers them before that s_{k+1} sends the $connect_{ACK,k+1}$ message to it, i.e., before that h_i can deliver application messages currently broadcasted by s_{k+1}. For the application messages received by s_{k+1} before the $Rsp_{1,k+1}$ message and that s_{k+1} did not discard, h_i will deliver them following the sequence number that s_{k+1} assigned to them, i.e., in causal order. Therefore, h_i delivers in causal order the application messages received by s_{k+1} before the $Rsp_{1,k+1}$ message.

To finish, application messages that s_{k+1} receives after the $Rsp_{1,k+1}$ message are not delivered by h_i. s_{k+1} delivers them to h_i following the sequence number associated to them. This sequence number is attributed following their reception order on the wired network, and the algorithm used for communications on the wired network[NMM18a] ensures that messages are disseminated in causal order. Therefore, those messages are delivered in causal order.

Therefore, h_i will deliver in causal order the application messages received by s_{k+1} before and after the $Rsp_{1,k+1}$ message, i.e., h_i delivers application messages in causal order. If s_{k+1} (resp. s_k) fails, then following *Lemma 4.1*, as long as other stations of $G_{s_{k+1}}$ (resp. G_{s_k}) are up, at least one of them will hold the causal information of h_i, and the faulty station will recover that causal information when recovering.

Termination. Theorem 5.3 shows that *WAS* ensures the termination when hosts do not move between cells. We show that an up host that changes cell eventually

co-delivers all application messages. Note that a host that delivers no message because it changes to often its cell would eventually be considered as down by the station registering it. Therefore, hosts are supposed to be periodically connected long enough to a station in order to deliver outstanding application messages.

Following *Lemma 4.2*, s_{k+1} recovers the application messages that it discarded prior to the connection of h_i and sends them to h_i. Hence, h_i eventually delivers all application messages that s_{k+1} receives prior to the $Rsp_{2,k+1}$ message and that s_{k+1} discarded prior to the connection of h_i. For any other message m, s_{k+1} will not discard it unless s_{k+1} considers h_i as down or unless h_i acknowledged it, and s_{k+1} will retransmit m periodically until h_i acknowledges it.

Hence, the handoff procedure ensures that s_{k+1} recovers and sends to h_i all messages it discarded that h_i has not delivered, and for all the other messages not delivered by h_i, s_{k+1} buffers them and retransmits them until h_i acknowledged them. If s_{k+1} (resp. s_k) fails, then following *Lemma 4.1*, as long as other stations of $G_{s_{k+1}}$ (resp. G_{s_k}) are up, at least one of them will hold the causal information of h_i, and the faulty station will recover that causal information when recovering.

□

Proof that *WAS* ensures causal order in presence of station failures

Lemma 4.8. *A handoff eventually executes in presence of station failures, and stations of $G_{s_{k+1}}$ eventually hold the causal information of h_i if s_{k+1} does not fail until the end of Phase 3.*

Proof. Following *Theorem 5.3*, a handoff eventually executes in a system without station failures. We prove that a handoff also eventually executes in presence of station failures.

If s_k fails, then a station s_b of G_{s_k} eventually takes up the responsibility of h_i. Moreover, s_k will then also not reply to the handoff messages of s_{k+1}, which will eventually conclude that s_k failed, and which will therefore begin again the handoff by broadcasting a $Req_{1,k+1}$ message. s_b will then receive that $Req_{1,k+1}$ message and execute the handoff with s_{k+1}. If s_b fails, then another station of G_{s_k} will eventually take up the responsibility of h_i, and so on as long as a station of G_{s_k} remains up. Therefore, the handoff eventually ends even if s_k or a station of G_{s_k} fail.

If s_{k+1} fails before it received the $Rsp_{2,k+1}$ message, i.e., before it broadcasted the $register_{k+1}$ and *delete* messages, then it will stop sending handoff messages to s_k, which will eventually cancel the handoff, thus ending it.

If s_{k+1} fails after it received the $Rsp_{2,k+1}$ message, i.e., after it broadcasted the $register_{k+1}$ and *delete* messages, then stations of $G_{s_{k+1}}$ eventually receive the $register_{k+1}$ message and register h_i, and stations of G_{s_k} eventually receive the *delete* message and unregister h_i, thus ending the handoff.

Stations other than s_k, s_{k+1} or in G_{s_k} do not participate in the handoff and their failure has therefore no impact on it. □

Lemma 4.9. *No causal information about h_i is lost during a handoff during which a station fails.*

Proof. Theorem **??** proves that no causal information about h_i is lost in during handoffs in a system without station failures. We prove this also holds in system where station failures occure during handoffs.

We prove that no causal information about h_i is lost during a handoff for h_i, i.e. that eventually either stations of G_{s_k} or $G_{s_{k+1}}$ hold the causal information of h_i. We prove it by induction.

H0: Assume that s_0 is the first station from which h_i receives a $connect_{ACK,k+1}$ message. Stations of G_{s_0} eventually hold the causal information of h_i.

Station s_0 broadcasts a $register_{k+1}$ message on the wired network when receiving a $join_{k+1}$ message from h_i. Stations of G_{s_0} eventually receive that $register_{k+1}$ message and will then register h_i. Moreover, the causal order property ensured by the algorithm used for communication on the wired network ensures that stations of G_{s_0} receive the causal information related to h_i in causal order. Therefore, they will receive the causal information related to h_i in causal order.

H1: Assume that stations of G_{s_k} eventually hold the causal information related to h_i. We show no causal information is lost when h_i moves to the cell of station s_{k+1}.

Following *Lemma 4.8*, each handoff eventually ends. We show that at the end of handoff $H_{i,k+1}$ the causal information related to h_i is not lost. By definition, the stations of G_{s_k} eventually hold the causal information related to h_i. They only delete that causal information upon receiving a *delete* message from s_{k+1}. s_{k+1} only sends that *delete* message after broadcasting to the stations of $G_{s_{k+1}}$ a $register_{k+1}$ message containing the causal information related to h_i. Moreover, the algorithm used for communications on the wired network ensures that the stations of $G_{s_{k+1}}$ receive the messages that s_{k+1} receives after broadcasting the $register_{k+1}$ message in causal order, i.e., the stations of $G_{s_{k+1}}$ receive all the causal information related to h_i and not contained in the $register_{k+1}$ message in causal order.

Therefore, stations of G_{s_k} do not receive that *delete* message and unregister h_i unless stations of $G_{s_{k+1}}$ eventually receive the causal information related to h_i. Therefore no causal information is lost during the handoff $H_{i,k+1}$.

Any handoff between two stations s_k and s_{k+1} executes without the loss of causal information (H1). Since the initial connection from h_i to a station executes without the loss of causal information (H0), we conclude that handoffs execute without the loss of causal information.

□

Lemma 4.10. *When several handoffs $H_{i,k+1}, \ldots, H_{i,k+n}$ are executing simultaneously, then the stations of $G_{s_{k+l}}$ eventually hold the causal information of h_i, with s_{k+l} being the station of the highest index $k < k+l < k+n$ that didn't fail before the end of Phase 3.*

Proof. Let's denote s_k the station initially responsible for the causal information of h_i, and that several stations s_{k+1},...,s_{k+n}, with $n > 1$, start a handoff for h_i in a short time interval. Those stations all broadcast a Req_1 message. s_k first executes the handoff $H_{i,k+p}$ corresponding to the first $Req_{1,k+l}$ message it receives.

Following *Lemma 4.8*, handoff $H_{i,k+p}$ eventually executes, and following *Lemma 4.9* s_{k+p} will then have the causal information of h_i. Stations regularly re-broadcast the Req_1 message. Moreover, station s_{k+p} only processes a $Req_{1,k+q}$ message if $k+p < k+q$. Therefore, in the worst case l handoffs are executed before that handoff $H_{i,k+l}$ is executed, i.e., handoff $H_{i,k+1}$ then $H_{i,k+2}$ etc... are executed until handoff $H_{i,k+l}$. Therefore, following *Lemma 4.8* and *Lemma 4.9* the stators of $G_{s_{k+l}}$ will eventually hold the causal information of h_i.

By definition, the stations $s_j, k+l < j < k+n$ do fail before they achieved the end of Phase 3 of handoff $H_{i,j}$. Therefore, those handoffs are eventually aborted, and the causal information of h_i is eventually maintained at stations of $G_{s_{k+l}}$.

□

Lemma 4.11. *For each broadcasted application message m, if $\exists s_i \in G$ such that s_i assigns the sequence number seq to m, then $\forall s_k \in G$ we have s_k that assigns the sequence number seq to m.*

Proof. Each group G has a designated station, denoted s_{resp}, that is responsible to assign a sequence number to application messages broadcasted by stations of G. Upon reception of an application message m, a station of G delays the broadcast of m on its wireless network until receiving the sequence number that s_{resp} assigns to m. Hence, stations of G all assign the same sequence number to m given by s_{resp}.

Only s_{resp} assigns sequence numbers to application messages. Therefore, it is sufficient to prove that stations of G continue to give the same sequence number to application messages when s_{resp} fails.

When s_{resp} fails, stations of G eventually detect it since they stop receiving messages from s_{resp}. They then elect a new responsible station s_{resp_n}. Upon electing s_{resp_n}, stations of G do not accept new messages of s_{resp}, which might still be transiting over the wired network even though s_{resp} already failed. This ensures that s_{resp_n} does not assign a sequence number seq' to an application message m to which s_{resp} already assigned a sequence number seq and that some stations assign the sequence number seq to m and other assign seq' to m. Therefore, upon being elected s_{resp_n} simply assigns sequence numbers to application messages it stores as well as to those application messages it receives, and stations of G will order application messages following the sequence number provided by s_{resp_n}, this ensuring that all stations of G assign the same sequence number to application messages than s_{resp_n}. □

Theorem 4.3. *WAS is resilient to station failures when hosts do not change cells.*

Proof. Consider a host h_i that joins the system. We show that the causal information related to h_i is resilient to station failures as long as h_i does not change cells.

Station s_i broadcasts the $join_{k+1}$ message that h_i sends to it to join the system. The algorithm used for the communication between stations [NMM18a] ensures that stations of G_{s_i} will eventually receive that $join_{k+1}$ message and therefore register the connection with h_i.

h_i uses seq_{NC} to deliver application messages, and stations of G_{s_i} use seq_h they associate to the connection with h_i to order application messages broadcasted by h_i. Hence, we must show that seq_{NC} and seq_h are valid at the stations of G_{s_i}.

Following *Lemma 4.11*, stations of G_{s_i} assign the same sequence number to application messages. If s_i fails, it will recover the sequence number assigned to application message by a station of G_{s_i} upon recovery. Therefore, seq_{NC} will remain valid in the case of a temporary failure of s_i. The case of stop failure is not considered here since s_i is assumed to not change cell. Therefore, seq_{NC} always stays valid for h_i.

seq_h that stations use to broadcast application messages of h_i is also valid at stations of G_{s_i}. When s_{resp} broadcasts an application message m of h_i on the wired network, it attaches m's id (h_i, seq_h) on m. Upon reception of m, the stations of G_{s_i} update the seq_h associated with the connection with h_i. The algorithm used for the communication between stations [NMM18a] ensures that each station

of G_{s_i} eventually receives the application messages broadcasted by s_{k+1}. Moreover, when recovering s_i broadcasts a recovery message, and the algorithm used for communication between stations[NMM18a] ensures that stations of G_{s_i} receive the $recovery_k$ message after the messages of h_i that s_i broadcasted before failing. Therefore, upon reception of the $recovery_k$ message, stations of G_{s_i} will have received all application messages from h_i that s_i broadcasted, and will therefore return a valid seq_h in the $recovery_{ACK,k}$ message. Therefore seq_h that s_i associates to h_i will be valid upon recovery.

□

Theorem 4.4. *WAS is resilient to station failures when hosts change cells.*

Proof. Following *Theorem 4.3*, *WAS* is resilient to station failures when hosts do not change cells. We show that *WAS* is also resilient to station failures when hosts do change cells. When a host h_i changes cells for the k^{th} time, it executes a handoff denoted $H_{i,k}$. It is sufficient to show that, in presence of station failures, handoff $H_{i,k}$ remains valid as described in the handoff description.

If s_k receives a $Req_{1,k+n}$ message before the $Req_{1,k+1}$ message, then it will start handoff $H_{i,k+n}$. During handoff $H_{i,k+n}$, s_k does not process the $Req_{1,k+1}$ messages, and if s_{k+n} does not fail up to the end of Phase 3 of handoff $H_{i,k+n}$, then s_k will unregister h_i and broadcast a *delete*[k+n] message, thus aborting the handoff $H_{i,k+1}$.

Following *Lemma 4.9*, no causal information is lost during handoffs when stations do fail. Moreover, following *Lemma 4.8*, a handoff eventually executes in presence of station failures and stations of $G_{s_{k+1}}$ do eventually hold the causal information related to h_i if s_{k+1} does not fail until the end of Phase 3. Otherwise the handoff is aborted, i.e., stations of G_{s_k} eventually hold the causal information of h_i.

Therefore, the three conditions of a valid are satisfied, and the handoff execution are therefore valid and *WAS* is resilient to station failures when hosts change cells. □

Theorem 4.5. *Stations broadcast application messages exactly once.*

Proof. We proof it by induction.

H0: Stations broadcast application messages from a host h_i exactly once, as long as h_i stays in its initial cell.
Assume that h_i joins the system by connecting itself to station s_k. s_k initializes the sequence number seq_h attributed to h_i to 1. h_i gives a unique increasing sequence number to application messages it broadcasts, beginning with 1. s_k increments seq_h when broadcasting the application message of h_i of sequence number seq_h,

and discards the application messages from h_i whose sequence number is lower than seq_h. h_i retransmits its application messages until s_k acknowledges them. Moreover, s_k acknowledges an application message to h_i only after broadcasting it on the wired network. Hence, s_k broadcasts each application message of h_i exactly once. The algorithm used on the wired network ensures that the other stations receive exactly once an application message broadcasted by s_k. Hence, other stations broadcast each application message exactly once.

If s_k fails, a station $s_b \in G_{s_k}$ eventually takes up the responsibility of h_i. By hypothesis, no application message from h_i is in transit when s_b takes up responsibility of h_i. Hence, s_b received all messages from h_i that s_k broadcasted. Moreover, s_b increments seq_h associated to the connection with h_i when receiving an application messages from h_i over the wired network. Hence, when s_b takes up the responsibility of h_i, the seq_h value it associates to h_i correspond to the number of application messages from h_i that s_k has broadcasted. When s_k recovers, it will recover seq_h through s_b.

H1: We assume that host h_i is first connected to station s_k and that during that connection no station broadcasted twice an application message from h_i. We show that when h_i connects itself to s_{k+1}, station broadcast exactly once application messages from h_i.

Upon the reception of the $connect_{k+1}$ message from h_i, s_{k+1} starts handoff $H_{i,k+1}$. After sending the $connect_{k+1}$ message, h_i broadcasts no application message to another station than s_{k+1}, and s_{k+1} broadcasts no application message from h_i until the end of the handoff.

During the handoff, s_{k+1} sends seq_h to s_{k+1}, and s_{k+1} does not broadcast any application message from h_i of sequence number lower than seq_h, i.e. it will broadcast no application message from h_i already broadcasted by s_k. Second, s_{k+1} increments seq_h when broadcasting a message of h_i. Hence, it will broadcast exactly once any application message of sequence number higher than seq_h. Hence, s_{k+1} broadcast application messages from h_i exactly once. The algorithm used on the wired network ensures that the other stations receive exactly once an application message broadcasted by s_{k+1}. Hence, other stations broadcast application message of h_i exactly once.

If s_k fails during the handoff, then s_{k+1} eventually does the handoff with a station of G_{s_k} after the timeout expiration. If s_{k+1} fails, then s_k cancels the handoff after the timeout expiration and s_{k+1}.

Stations broadcast application messages from h_i exactly once when h_i changes cell *(H1)*. Since stations broadcast once application messages from h_i as long as h_i stays

in its initial cell *(H0)*, we conclude that stations broadcast exactly once application messages from h_i. □

4.7 Conclusion

This chapter presented a causal broadcast algorithm tailored to the features and dynamics of mobile networks. Such networks include host mobility, dynamic host membership, unreliable dynamic wireless channels, memory and computing constraints of mobile hosts, scalability issues due to the high number of mobile hosts and stations. Both algorithms tolerate temporary and permanent failures of mobile host. The second algorithm also handles the permanent and temporary failures of stations. Messages piggyback few causal information. The algorithms scale well with hosts which have a low memory footprint while stations have a memory footprint that grows linearly with the number of locally connected hosts. Stations discard obsolete messages with only local information, removing the message exchange between stations used by centralized approaches to discard obsolete messages.

Performance results from simulations done on OMNeT++/INET show that *WAS* has a much lower message overhead, delivery delay, and caches fewer messages than a representative causal multicast algorithm for Mobile Networks adapted to provide causal broadcast [CK04]. Furthermore, the decentralized approach to discard obsolete messages of *WAS* induces much fewer messages cached by stations, as well as much fewer messages sent on the network, when compared to a centralized approach. Finally, hosts that temporarily fail rapidly catch up in delivering outstanding messages after recovering, and the decentralized discard approach of obsolete messages mostly limits the impact of host failures to the cells in which those failures occur.

WAS2 has an additional cost over *WAS*: the stations of a group need to communicate with each other to order messages and discard obsolete ones. This communication increases delivery delays as well as the causal information that is transmitted over the wired network. Future experiments will analyze the delivery delays and size of causal information depending on the size of the group of stations.

Chapter 5

Causal broadcast implemented using clocks of size $M \leq N$

Contents

5.1 Introduction

Torres-Rojas and Ahamad [TA99] introduced in 1998 *M-entry clocks*, which are clocks of $M \leq N$ entries, where N corresponds to the number of processes in the system. The second part of this thesis aims to enhance the accuracy in causally ordering messages when using such clocks to implement causal broadcast. *M-entry clocks* cannot characterize causality when doing causal broadcast, as shown by Charron-Bost who proved that at least vector clocks with one entry per process of the system are required [Cha91]. Nevertheless, Torres-Rojas and Ahamad [TA99][Tor01] showed through a theoretical and experimental analysis that an implementation of causal broadcast using M-entry clocks causally orders messages with a high probability. However, such algorithm might deliver messages out of causal order. Among *M-entry clocks*, Probabilistic clocks [MW17b] and Bloom clocks [MK21] have the best performances (see Section 3.2.3).

M-entry clocks are adapted to applications where messages that are delivered out of causal order only impact performance but not correctness. For example, applications that measure concurrency by analyzing a history of events in order to inform how many pairs of events are concurrent. Such applications would still measure the concurrency with a high precision when using *M-entry clocks*, even if some causal relationships would be missing in the event history [TA99]. Other examples are timestamped-based resource allocation or protocols that ensure the causal consistency of data through invalidation [TA99]. Moreover, in many systems messages are implicitly causally ordered. For example, in systems where the time between the generation of two causally related messages is higher than the communication delay between processes.

M-entry clocks are also adapted to applications where some errors are acceptable. For example, in social networks such as instagram, most of the comments/replies correspond to the image poster. Hence, very few replies that do not appear in the correct order make the readers misunderstand the context of the comments [HK17].

Mostefaoui and Weiss [MW17a] proposed an error detector whose purpose is to detect out of causal order deliveries when implementing causal order with probabilistic clocks. To that end, the error detector analyzes the clock attached to a message before delivering it. The error detector yields some errors, which are either false positives or false negatives. A false positive corresponds to the error detector wrongly concluding that a message is delivered out of causal order. A false negative corresponds to the error detector not detecting an out of causal order delivery.

The first contributions presented in this chapter are about error detectors that aim to detect out of causal order deliveries. We first show that it is impossible to implement an error detector that detects all out of causal order deliveries. In

the second contribution, we propose an error detector based on the hashing of messages' causal dependencies. The proposed error detector has a high accuracy of detecting out of causal order deliveries. Third, we propose an algorithm to recover the causal dependencies of messages. We propose to use it with error detectors analyzing messages before delivering them, and to retrieve the causal dependencies of tagged messages, in order to ensure that they are causally delivered. The third contribution is based on the observation that the accuracy of an M-entry clock of a given size M decreases when the number of concurrent messages increases. We therefore propose a new clock, denoted Dynamic Clock Set (*DCS* clocks), composed of Probabilistic Clocks [MW17a]. The size of *DCS* clocks can dynamically vary during execution. In particular, the size of *DCS* clocks can be adapted to the number of concurrent messages in order to keep the number of messages delivered out of causal order below a tuneable threshold.

We conducted experiments on the OMNeT++ simulator. Results show that the proposed error detector detects most -experimentally all- messages delivered out of causal order, and that a causal broadcast implementation using *DCS* clocks adapts the size of the *DCS* clock well to the message load. Moreover, retrieving the causal dependencies of messages tagged as not causally ordered allows to heavily reduce the number of messages delivered out of causal order. Hence, experimental results confirm the effectiveness of the proposed contributions.

This chapter is organized as follows: Section 5.2 gives some additional background on Probabilistic clocks [MW17b]. Section 5.3 presents the required conditions to implement a reliable error detector as well as the hash-based error detector. Section 5.4 presents the algorithm to retrieve the causal dependencies of messages and discusses its use in conjunction with the hash-based error detector. Finally, Section 5.5 presents the *DCS* clocks and an implementation of causal broadcast using them.

5.2 Background

The contributions presented in this chapter use Probabilistic clocks [MW17b] which have, with Bloom clocks [MK21], the best performances among clocks with $M \leq N$ entries. Hence, this section gives some additional background about them.

5.2.1 Probabilistic clocks

In the implementation of causal broadcast using probabilistic clocks, each process p_i keeps a local clock V_i with $M \leq N$ entries initialized to 0, where N corresponds

to the number of processes in the system. A hash function $f(p_i)$ returns the set of k clock entries assigned to p_i, with $1 \leq k \leq M$, i.e., one to several probabilistic clock entries are respectively associated to each process at initialization. $f(p_i)$ as well as k are fixed at initialization and do not change during execution. Process p_i uses the following two rules $R1$ and $R2$ to update its local probabilistic clock:

R1 - Before executing an event, p_i updates its local clock:
$$\forall x \in f(p_i), V_i[x] = V_i[x] + 1$$

R2 - Each message m carries with it the vector clock of its sender process at sending time. On the receipt of a message m, process p_i :

- Updates its local clock as follows: $\forall x, V_i[x] = max(V_i[x], m.V[x])$
- Executes $R1$,
- Delivers m

The comparison operator of two probabilistic clocks V_1 and V_2 is defined as follows:
$$V_1 < V_2, \text{ iff } \forall x, 1 \leq x \leq M, V_1[x] \leq V_2[x] \wedge \exists i, V_1[i] < V_2[x]$$

The following condition holds on probabilistic clocks:
$$send(m_1) \rightarrow send(m_2) \Rightarrow m_1.V < m_2.V$$
But the contrary is not true:
$$m_1.V < m_2.V \not\Rightarrow send(m_1) \rightarrow send(m_2)$$

5.2.2 Causal broadcast using probabilistic clocks

Causal broadcast should ensure the properties of *Validity*, *Integrity*, and *Termination* defined in Section 2.4, plus the causal delivery of messages:

Causal broadcast: Consider two broadcast messages m and m', if m causally precedes m', then all processes should deliver m before m':
$$broadcast(m) \rightarrow broadcast(m') \Rightarrow deliver(m) \rightarrow deliver(m')$$

Algorithm 8 describes the causal broadcast algorithm using probabilistic clocks presented in [MW17b]. Before broadcasting a message m, process p_i increments the entries $f(p_i)$ of its local vector clock V_i, then it broadcasts V_i with m. Upon reception of a message m from a process p_j, p_i delays the delivery of m till the two following conditions are satisfied: (1) $\forall x \in f(p_j), V_i[x] \geq m.V[x] - 1$ and (2) $\forall x \notin f(p_j), V_i[x] \geq m.V[x]$. p_i delivers m and increments the entries $k \in f(p_j)$ of V_i after both conditions are satisfied.

Authors of [MW17b] determine that the optimal number of clock entries that processes should increment when broadcasting a message is equal to: $|f| = ln(2) * \frac{|V|}{X}$,

Algorithm 8: Broadcast at process p_i

Broadcast of message m

1: $\forall x \in f(i), V_i[x] = V_i[x] + 1$
2: $m.V = V_i$
3: broadcast(m)

Upon reception of message m **from** p_j

3: waitUntil($(\forall x \in f(j), V_i[x] \geq m.V[x] - 1) \wedge \forall k \notin f(j), V_i[k] \geq m.V[k]$)
4: $\forall x \in f(j), V_i[x] = V_i[x] + 1$
5: deliver(m)

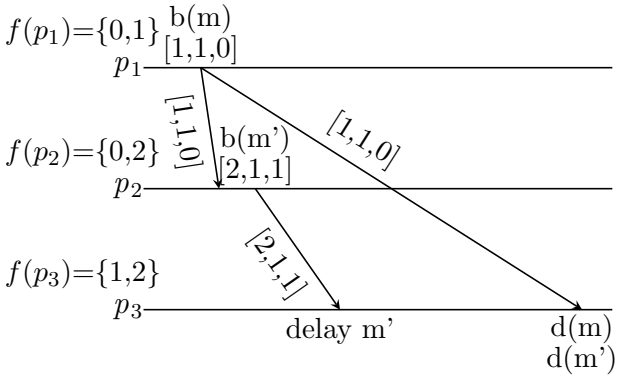


Figure 5.1: Causal broadcast using probabilistic clocks

where X corresponds to the average number of concurrent messages to m and $|V|$ the size of the used clock.

Figure 5.1 shows the broadcast of two messages m and m'. $M = 3$. The clock entries assigned to p_1, p_2, and p_3 are $f(p_1) = \{0, 1\}$, $f(p_2) = \{0, 2\}$, and $f(p_3) = \{1, 2\}$ respectively. p_i broadcasts (m, V_1) after incrementing the entries $f(p_1)$ of its local vector clock V_1. Process p_2 receives and delivers m. Then, it broadcasts (m', V_2) after incrementing the entries $f(p_2)$ of V_2. Thus, $m \rightarrow m'$. When p_3 receives m', the delivery conditions of m' are not yet satisfied since $m'.V[0] = 2 > V_3[0] = 0$. Consequently, p_3 buffers m'. Upon the reception of m, p_3 delivers m and increments the entries $f(p_1)$ of V_3. It then also delivers m' and increments the entries $f(p_2)$, since the delivery conditions of m' have been satisfied.

A causal broadcast implementation using probabilistic clocks might deliver some messages out of causal order. For example, assume that in Figure 5.1 process p_3 delivers messages concurrent to m before receiving m', such that the delivery of those messages increment $V_3[0]$ and $V_3[2]$. p_3 will then deliver m' out of causal order, because the delivery conditions of m' will be satisfied at p_3 upon reception.

Authors of [MW17b] gave the following formula to determine the probability that a process delivers a message m out of causal order: $(1 - (1 - \frac{1}{M})^{X*k})^k$, where M is the size of the clock attached on m, k the number of clock entries associated to each process, and X the number of messages concurrent to m. Two observations can be made out of that equation. First, increasing the size of the clock attached on m decreases the probability that a process delivers m out of causal order. Second, increasing the number of concurrent messages inside the system increases the probability that m is delivered out of causal order, because a concurrent message might increment the same clock entries as dependencies of m. Therefore, the higher the number of concurrent messages to m that p_i delivers before delivering m, the higher the probability that such deliveries increment the same clock entries as dependencies of m, and p_i will then wrongly satisfy m's delivery conditions, thus delivering m out of causal order.

5.2.3 Error detector

Authors of [MW17a] also proposed an error detector (see Algorithm 9) to detect out of causal order deliveries. Basically, the detector analyzes the probabilistic clock attached to a message m once the delivery conditions of m are satisfied, and returns *true* if it detects an error, and *false* otherwise. An error handler function can then further handle messages on which the error detector returned *true*. The error detector can yield false positives and negatives, i.e., it can wrongly conclude that a message will be delivered out of causal order (false positive), and it can also wrongly conclude that a message can be delivered in causal order (false negative).

Algorithm 9: Error detector executed by p_i before delivering m broadcasted by p_j

```
if ∃x ∈ f(p_j), V_{p_i}[x] = m.V[x] − 1 then
    return false # No error detected
return true # Error detected
```

Figure 5.2 shows an execution in which the error detector of Algorithm 9 detects that m_2 is not causally ordered. $M = 4$. Each process initializes its probabilistic clock to $[0, 0, 0, 0]$. The following entries $f(p_i)$ are assigning to $p_{1 \leq i \leq 4}$: $f(p_1) = \{0, 1\}$, $f(p_2) = \{1, 2\}$, $f(p_3) = \{0, 2\}$, $f(p_4) = \{1, 3\}$. Messages are sent to all processes, but only the relevant ones are shown for better readability. In this case, we can consider that the other messages are received later. $m_1 \rightarrow m_2$, since p_2 broadcasts m_2 after delivering m_1. However, p_3 receives m_2 before m_1, and upon reception, its delivery conditions are satisfied due to the delivery and broadcast of other messages. Nevertheless, the error detector of p_3 concludes that m_2 is not causally ordered: $f(p_2) = \{1, 2\}$, and $V_3[1] \neq m_2.V[1] - 1 \wedge V_3[2] \neq m_2.V[2] - 1$. Hence, the error detector returns *true*.

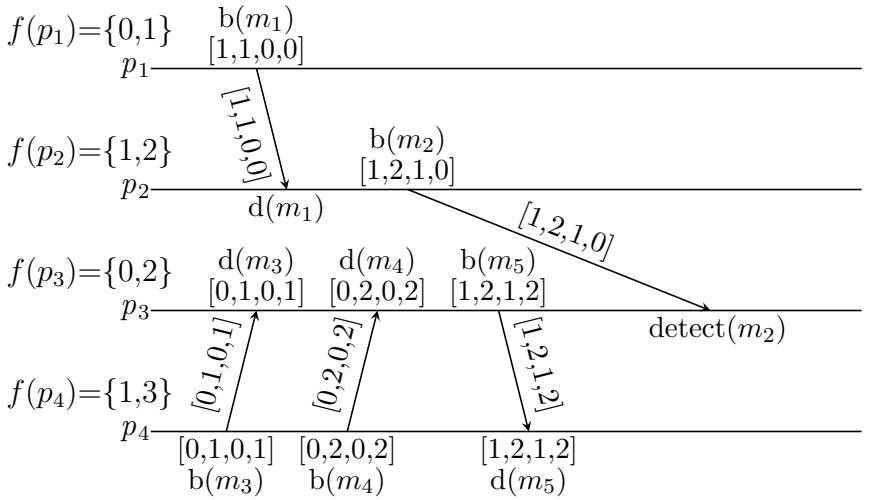


Figure 5.2: Delivery error detected by the error detection

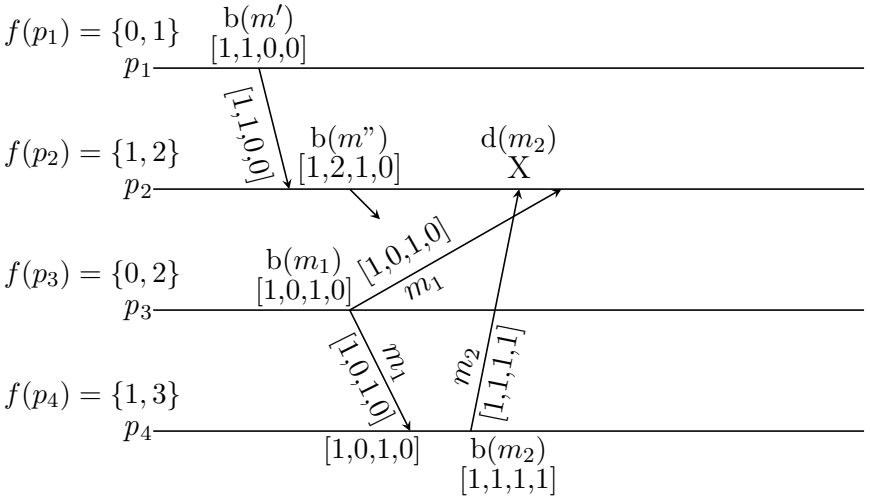


Figure 5.3: Delivery error not detected by the error detector

However, the error detector does not detect all out of causal order deliveries. For example, consider the example of Figure 5.3 , where $m_1 \rightarrow m_2$, but where p_2 receives m_2 before m_1, and the delivery conditions of m_2 are satisfied upon its reception by p_2. The error detector does not detect that p_2 cannot deliver m_2 in causal order because $f(p_4) = \{1, 3\}$ and $V_2[3] = m_2.V[3] - 1$. Hence, p_2 delivers m_2 at X without having previously delivered m_1.

5.3 Error detectors for M-entry clocks

This section presents the contributions related to error detectors.

Firstly, we define the conditions an error detector for *M-entry clocks* would require to detect all out of causal order deliveries. We then show that those conditions are unrealistic and that it is therefore impossible to implement such an error detector under realistic assumptions.

Secondly, we propose a new error detector based on hashed causal dependencies. It is based on hashing the causal dependencies of messages at sending, and determines if all the causal dependencies have been delivered at destination through the hash appended on messages. The proposed error detector has a much higher accuracy than the error detector proposed in [MW17a] in detecting out of causal order deliveries.

Thirdly, we present a short algorithm to retrieve the causal dependencies of messages. This algorithm can be used in conjunction with error detectors, by retrieving the causal dependencies of messages on which an error detector returns true, in order to ensure that the message is delivered in causal order.

5.3.1 Conditions required by a reliable error detector

This section first determines the conditions required to implement an error detector that detects all out of causal order deliveries. Second, we show that those conditions are unrealistic, and that it is therefore impossible to implement such an error detector under realistic assumptions.

In the following, and without loss of generality, we consider that the function f which returns the entries incremented by processes is chosen such that processes increment the entries of the M-entry clock V uniformly.

Lemma 5.1 determines the minimum set of messages a process must have delivered to ensure that it can causally deliver a message m.

Theorem 5.1 determines the conditions a process must satisfy to avoid being eventually deadlocked while waiting for the conditions of *Lemma 5.1* to be satisfied.

Finally, *Theorem 5.2* gives the number of messages each process must have broadcasted to avoid processes to be eventually deadlocked while waiting for the conditions of *Theorem 5.1* to be satisfied.

Lemma 5.1. *To ensure the causal delivery of a message m of M-entry clock $m.V$, process p_i must have delivered, from each process p_k, $min(\{m.V[e], e \in f(p_k)\})$ messages.*

Proof. p_i increments the entries $e \in f(p_k)$ of V_i when delivering a message from p_k. Hence, the maximum number of messages that p_i delivered from p_k prior to broadcasting a message m, on which it attaches V_i, is $m_k = min(\{m.V_i[e], e \in f(p_k)\})$. A process p_j receiving m cannot ensure that it can deliver m in causal order if it has delivered less than m_k from p_k. In fact, it cannot determine if p_i incremented the entries $f(p_k)$ of V_i by delivering m_k messages from p_k or by delivering messages from other processes. Hence, p_j must deliver m_k messages from p_k to ensure that it can deliver m in causal order.

□

One could deduce that to ensure the causal delivery of a message m, a process p_i only needs to delay the delivery of m until p_i has delivered, for each process p_k, $min(\{m.V[e], e \in f(p_k)\})$ messages. *Theorem 5.1* and *Theorem 5.2* determine the conditions to avoid a resulting deadlock.

Theorem 5.1. *A process p_i that waits to satisfy the conditions of Lemma 5.1 before delivering a message m will wait forever, i.e., will be deadlocked, unless, for each process $p_{k\in\Pi}$, p_k broadcasts at least $min_m = min(\{m.V[e], e \in f(p_k)\})$ messages prior to delivering m.*

Proof. Assume that p_k delivers m prior to broadcasting $min_m = min(\{m.V[e], e \in f(p_k)\})$ messages, then broadcasts its x^{th} message m', with $x < min_m$. We show that processes cannot deliver m before m', and vice versa, i.e., processes are deadlocked.

Following *Lemma 5.1*, to ensure that process p_i delivers a message m in causal order, it must deliver, from each process p_k, at least min_m messages before delivering m. Hence, p_i must deliver m' before m, since $x < min_m$. However, following *Lemma 5.1*, p_i must deliver m' after m, because it broadcasts m' after m and therefore $min_m < min_{m'}$. Thus, p_i cannot deliver m' before m, but it also cannot deliver m before m', i.e., it is deadlocked. □

Theorem 5.2. *Be $(M_k)_{k\in\mathbb{N}}$ a sequence of sets of messages such that $M_k \to M_{k+1}$. $\exists i_0 \in \mathbb{N}, \forall i > i_0$, such that to ensure the causal delivery of messages of M_i a process must broadcast at least $(\frac{|f|*N}{M})^{i-i_0}$ messages prior to delivering messages of M_i.*

Proof. We prove it by induction.

H0: $\exists i_o \in \mathbb{N}$, each process must broadcast at least $\frac{|f|*N}{M}$ messages before delivering messages of M_{i_0+1}.

First, $\exists i_o \in \mathbb{N}, min(\{min(m.V), m \in M_{i_0}\}) \geq 1$. In fact, a process increments the entries of its vector clock when broadcasting or delivering a message, following the entries that f returns. Since f is uniform, the entries of V are eventually all incremented, i.e. eventually $\forall p, min(p.V) > 1$.

Before delivering a message of M_{i_0+1}, a process must deliver all messages of M_{i_0} as well as the messages broadcasted prior to the messages of M_{i_0}. Following *Theorem 5.1*, each process must broadcast at least one message prior to delivering a message of M_{i_0}, because $\forall m \in M_{i_0}, min(m.V) \geq 1$. Hence, a process must deliver at least N messages prior to delivering a message of M_{i_0+1}. Each message delivery increments $|f|$ entries of the vector clock. Hence the delivery of N messages increments the vector clock $|f| * N$ times. Since $|f|$ is uniform by assumption, each entry of the vector clock is incremented $\frac{|f|*N}{M}$ times. Following *Theorem 5.1*, each process must broadcast at least $\frac{|f|*N}{M}$ messages prior to delivering a message of M_{i_0+1}, because $M_{i_0} \rightarrow M_{i_0+1}$, and therefore $\forall m \in M_{i_0+1}, min(m) > \frac{|f|*N}{M}$.

H1: We assume that processes must broadcast at least $(\frac{|f|*N}{M})^{i-i_0}$ messages prior to delivering messages of M_i. We show that processes must broadcast at least $(\frac{|f|*N}{M})^{i-i_0+1}$ messages prior to delivering messages of M_{i+1}.

Before delivering a message of M_{i+1}, a process must deliver all messages of M_i as well as the messages broadcasted prior to the messages of M_i.

Following *Theorem 5.1*, each process must broadcast at least $(\frac{|f|*N}{M})^{i-i_0}$ message prior to delivering a message of M_i. Hence, a process must deliver at least $x = (\frac{|f|*N}{M})^{i-i_0} * N$ messages prior to delivering a message of M_{i+1}. Each message delivery increments $|f|$ entries of the vector clock. Hence the delivery of N messages increments the vector clock $|f| * x$ times. Since $|f|$ is uniform, each entry of the vector clock is incremented $\frac{x}{M}$ times. Following *Theorem 5.1*, each process must broadcast at least $(\frac{|f|*N}{M})^{i-i_0+1}$ messages prior to delivering a message of M_{i+1}, because $M_i \rightarrow M_{i+1}$, and therefore $\forall m \in M_{i+1}, min(m) > (\frac{|f|*N}{M})^{i-i_0}$.

Thus, $\forall i > i_0$, processes must broadcast at least $(\frac{|f|*N}{M})^{i-i_0+1}$ messages prior to delivering messages of M_{i+1} (H_1). Since $\exists i_0$ (H_0), we conclude that $\exists i_0 \in \mathbb{N}, \forall i > i_0$, each process must broadcast at least $(\frac{|f|*N}{M})^{i-i_0}$ messages prior to delivering messages of M_i. □

Theorem 5.2 shows that the number of messages that processes must broadcast to avoid a deadlock increases exponentially. Hence, it is impossible to implement, under realistic assumptions, an error detector that detects all out of causal order deliveries. For example, consider a system with an average network delay of 100ms,

$|f| = 2$, $M = 100$, $N = 10000$. Following the formula given in *Theorem 5.2*, each process must broadcast approximately one second after the condition $min(V) = 1$ is true (very fast) $1 * (\frac{2*10000}{100})^{10} = 200^{10} = 1.024 * 10^{23}$ messages to avoid the deadlock of processes. Obviously, this does not correspond to a normal execution. Hence, implementing a reliable error detector is not possible under realistic assumptions.

5.3.2 The hash-based error detector

This section presents an error detector based on hashed causal dependencies. A process p_i calls it before delivering a message m. Its purpose is to detect out of causal order deliveries.

The section is organized as follows. Section 5.3.2.1 presents the principle of the hash-based error detector. Section 5.3.2.2 describes how the error detector chooses the message ids to hash. Section 5.3.2.3 describes how the error detector builds dependency sets to hash with the message ids it chose to hash. Section 5.3.2.4 presents another mechanism to choose the message ids to hash. Section 5.3.2.5 presents a mechanism to make the error detector resilient to a high message load, and finally Section 5.3.2.6 gives an example of execution of the error detector.

5.3.2.1 Principle of the hash-based error detector

The hash-based error detector is based on hashing the causal dependencies of messages, and appending those hashes on messages when broadcasting them. A process p_i executes the hash-based error detector on a message m once the delivery conditions of m's M-entry clock are satisfied. It builds dependency sets with the ids of messages that it has delivered, computes their hash, and compares it with the hash appended on m, in order to find the dependency set of m. Algorithm 10 describes the hash-based error detector.

Let's consider an execution from the broadcast of a message m by process p_j till its delivery by process p_i.

Process p_j broadcasts m by first computing m's hash H_m, then broadcasting m with H_m and its M-entry clock V_j. p_j builds H_m with the ids of some causal dependencies of m, which it choses as described in Section 5.3.2.2.

Upon reception of m, process p_i executes its error detector on $m.V$ once its local clock V_i satisfies the delivery conditions of m's clock. The error detector first builds *setsToHash*, the set of dependency sets it will hash (line 2). A dependency set is composed of ids of causal dependencies of m. The message ids the error detector

takes into account when building *setsToHash* is described in Section 5.3.2.2. The error detector computes the hash of each set *depSet* of *setsToHash* (line 3) till it finds one whose hash *hash(depSet)* is equal to H_m (line 4). The error detector returns *false* if it finds such a set. Otherwise, it did not succeed to find a dependency set for m with the messages p_i delivered and returns therefore *true*, i.e., maybe m cannot be delivered in causal order.

A message m on which the error detector returns *true* can be handled in several ways. Either, p_i requests m's causal dependencies Dep_m to p_j, and upon reception of m's causal dependencies, p_i delivers m once it has delivered all messages whose id is contained in Dep_m ; or p_i delivers m to the application with the information that it might not be causally ordered.

Algorithm 10: Hash-based error detector executed by p_i

```
1: Input: m: message to deliver
2: setsToHash = buildDependencySets()
3: for depSet ∈ setsToHash do
4:    if hash(depSet)==m.H_m then
5:       return false # No error detected
6: return true # Error detected
```

Collisions may occur when hashing dependency sets, i.e., two dependency sets may have the same hash value, which means that the error detector may find a set Dep'_m of hash $H_{Dep'_m} = H_m$, but $Dep'_m \neq Dep_m$. Such a situation is very unlikely to happen since a hash of x bits corresponds to a hash space of 2^x values. Nevertheless, each new computed hash increases the probability of a collision, since at each hash computation we have a given collision probability and the hash computations are independent. Hence, the probability of a collision can be bounded by computing at most a given number of hashes. Moreover, it exponentially decreases when the number of bits on which the hash is stored increases.

5.3.2.2 Choosing the message ids to hash

Process p_j computes the hash H_m of a message m it broadcasts with some of the ids of m's causal dependencies. When delivering m, p_i builds dependency sets to hash with the ids of the messages it delivered previously. This section describes how p_j and p_i choose the message ids they respectively take into account in their hash computation.

The operations described in this section require processes to keep the M-entry clock of messages after delivering them. Each process p_i keeps messages it delivered as well as their clock in the set $S_{deliv,i}$. Section 5.3.2.2.c describes how to clear $S_{deliv,i}$.

5.3.2.2.a Clock difference

Processes use the difference between the clocks of messages to determine which message ids to hash. The difference between two clocks is defined as:

Definition 5.1. Let's consider two messages m_1 and m_2 which respectively carry the clocks V_1 and V_2. The difference between V_1 and V_2 is:

$$\text{Diff}_{m_1,m_2} = \sum_x V_1[x] - \sum_x V_2[x]$$

M-entry clocks increase over time, due to the delivery of messages by processes. Hence, the average time interval between the sending of two messages m_1 and m_2 increases with their clock difference $Diff_{m_1,m_2}$. Moreover, the closer $Diff_{m_1,m_2}$ is getting to 0, the smaller the interval between the sending of m_1 and m_2. We do the following observations based on the value of Diff_{m_1,m_2}:

- $Diff_{m_1,m_2} < 0$*:* The probability that m_1 is a causal dependency of m_2 increases as Diff_{m_1,m_2} decreases.
- $Diff_{m_1,m_2} \to 0$*:* The probability that m_1 and m_2 are concurrent increases as Diff_{m_1,m_2} is getting closer to 0.
- $Diff_{m_1,m_2} > 0$*:* The probability that m_2 is a causal dependency of m_1 increases as Diff_{m_1,m_2} increases.

5.3.2.2.b Message ids considered when computing a message's hash

Charron-Bost proved in [Cha91] that a structure that characterizes the causality of an event in a system containing N processes has at least N entries when doing causal broadcast. Hashing such a structure would require $\mathcal{O}(N)$ operations, and would therefore not scale. Instead, processes hash the causal dependencies of only a subset of message ids.

Consider a process p_i that broadcasts a message m. p_i computes the hash of m with only the ids of causal dependencies m' of m whose clock difference with m is lower than a given value *Diff*. Formally, p_i computes the hash H_m of m as follows:

$$H_m = \text{hash}(\ \{m'.\text{id}, \text{Diff}_{m,m'} < \text{Diff} \wedge m' \to m\}\ )$$

The drawback of computing H_m with only those messages m' is that a process p_j that receives m, but which has not delivered yet a message m" such that $Diff_{m,m"} > Diff$, will not detect it through the hash computation. Therefore, *Diff* should be chosen such that messages that are not yet delivered by all processes are taken into account when computing the hash of messages. In addition, Section 5.3.2.4 describes a mechanism that ensures to take into account all relevant message ids, but which is more complex to implement.

Value of *Diff* The formula to determine the value of *Diff* consists of two parts. It contains the following variables:

- m_{load}: the system's message load.
- *maxDelay*: the maximum communication delay in the system.
- $|f|$: the number of entries incremented by processes when broadcasting/delivering a message.
- n_{conc}: the average number of concurrent messages.

The first part is equal to $maxDelay{*}m_{load}{*}|f|$. Process p_i should include a message m's id when computing m's hash as long as some processes did not receive and deliver m, i.e., for at least *maxDelay* seconds. During that time, p_i delivers on average m_{load} messages, and it increments $|f|$ values of its clock at each delivery.

The second part is equal to $n_{conc} * |f|$. Between the sending of m and its reception by a process p_j, p_j will potentially deliver n_{conc} concurrent messages to m, thus increasing its clock $n_{conc} * |f|$ times.

The formula to determine the value of *Diff* is:

$$Diff = \text{maxDelay}{*}m_{load}{*}|f| + n_{conc} * |f|$$

5.3.2.2.c Message ids considered when building dependency sets

This section describes how the error detector of process p_i chooses the message ids to hash when analyzing the causal delivery of a message m sent by a process p_j (Algorithm 10 line 2).

The error detector of process p_i chooses the message ids to hash by analyzing $S_{deliv,i}$. As described in Section 5.3.2.2.b, process p_j, the sender of m, computed the hash of m with the causal dependencies m' of m that verify $Diff_{m,m'} < Diff$. Hence, the error detector of p_i computes hashes with the message ids (id, seq) of messages contained in $S_{deliv,i}$ whose clock difference with m is lower than *Diff*.

Moreover, M-entry clocks capture causality: for any two messages m and m':

$$m \rightarrow m' \Rightarrow m.V < m'.V$$

Therefore, all messages m' such that $\exists x, m'.V[x] > m.V[x]$ are not dependencies of m. Hence, the error detector of p_i does not include the ids of such messages when building dependency sets to hash.

To conclude, when analyzing a message m, the error detector of p_i builds dependency sets to hash with the following message ids:

$$\mathrm{M}_{hash}=\{(m'.id, \forall m' \in S_{deliv,i},\ m'.V < m.V \wedge \mathit{Diff}_{m,m'} < \mathit{Diff}\}$$

Process p_i also uses *Diff* to clean $S_{deliv,i}$. It keeps in $S_{deliv,i}$ the messages whose clock difference with V_i is lower than *2*Diff*: It considers that in the worst case, it did not receive yet a message m' whose clock difference with V_i is at most *Diff*, and the causal dependencies of m' will have a clock difference with m' of at most *Diff*. Hence, the messages with a clock difference with V_i of more than *2*Diff* will not be used anymore in any hash computation.

5.3.2.3 Building dependency sets

M-entry clocks do not characterize causality. Hence, the set M_{hash} that process p_i identified (see Section 5.3.2.2.c) might contain the id of messages that are concurrent to m. Therefore, p_i also needs to build and hash subsets of M_{hash} in order to find one which does not contain the ids of messages concurrent to m. The number of sets p_i can build exponentially increases with the size of M_{hash}. This section therefore describes how to choose the subsets in order to increase the probability for p_i to find a dependency set whose hash is equal to the one appended to m, and by doing it with the fewest possible hashes.

Note that the probability that M_{hash} contains a message m' concurrent to a message to deliver m increases with the number of messages concurrent to m'. In fact, the probability that $m.V \geq m'.V$ increases with the number of concurrent messages to m' that the sender process of m delivers prior to broadcasting m.

M_{hash}: The set of message ids ordered following their probability to be concurrent to m. The error detector uses the clock difference between two messages to estimate their probability to be concurrent. As explained in Section 5.3.2.2.a, the probability that two messages m_1 and m_2 are concurrent increases as Diff_{m_1,m_2} is getting closer to 0. Hence, the error detector orders M_{hash} by increasing clock difference with m, i.e., $M_{hash}[0] = (m'.id,\ \mathit{Diff}_{m,m'} = min(\{\mathit{Diff}_{m,m'}, m' \in M_{hash}\})$. Note that M_{hash} does not contain messages m' such that $\mathit{Diff}_{m,m'} < 0$, because it only contains messages which verify $m'.V < m.V$.

depSet: The set of message ids sets to hash. Process p_i builds *depSet*, the dependency sets to hash, with the messages of M_{hash}. It first adds to *depSet* the list of all ids of M_{hash}. Then it builds the combinations of M_{hash} by progressively removing the messages that have the highest probability to be concurrent to m. For example, it first adds to *depSet* the combinations of M_{hash} without $M_{hash}[0]$ that contain the tuples $M_{hash}[j], j > 0$, then it adds the combinations of M_{hash} without $M_{hash}[1]$ that contain the tuples $M_{hash}[j], j > 1$, etc...

***MaxHashes*: a bound on the number of computed dependency sets.** The number of built dependency sets should be bounded. In fact, this number can be very high, while it can even be impossible to find a message m's dependency set if the process did not deliver one of m's causal dependencies. For example, even if process p_i delivered all causal dependencies of a message m, it potentially needs to compute many dependency sets if it takes into account concurrent messages of m. Moreover, p_i will not find such a set (except in case of a collision) if it did not deliver a causal dependency m' of m.

The probability that the hash of a dependency set is equal to the hash H_m of m also decreases during the hash computation, i.e., the probability that *hash(depSet[x])* = H_m is higher than *hash(depSet[x+100])* = H_m, because *depSet[x]* is built by removing messages that have a higher probability to be concurrent to m than the messages removed when building *depSet[x+100]*.

Therefore, the error detector should not compute the hash of all the possible dependency sets of m. Instead, it only computes at most a given number of *MaxHashes* hashes before returning false, i.e., concluding that m might not be causally ordered. Bounding the number of computed hashes also bounds the probability of a hash collision, since at each new computed dependency set we have a given probability of a collision.

To summarize, Algorithm 11 describes the function *buildDependencySets* of Algorithm 10. We illustrate it with the following example. Consider that process p_i executes its error detector on a message m, and that it has delivered the messages m_1, m_2, m_3, m_4 prior to receiving m. p_i first builds M_{hash}, which let's say contains the following ids ordered following their clock difference with m M_{hash}=$\{m_2, m_1, m_3\}$. p_i builds possible dependency sets for m as follows: (1) first it adds the set containing all the message ids to *depSet* (line 1), i.e., it considers that m_1, m_2, m_3 are causal dependencies of m (2) it builds dependency sets without the id of the message which has the highest probability to be concurrent to m, i.e. m_2. Hence, it adds the dependency sets $\{m_1, m_4\}$ to *depSet*. Then it continues by building dependency sets without the id of the second message that has the highest probability to be concurrent to m, i.e., $\{m_2, m_4\}, \{m_4\}$. Finally, it builds dependency sets without m_4: $\{m_2, m_3\}, \{m_2\}, \{m_1\}, \{\}$. In total it builds 7 dependency sets and sorts them as follows: $\{m_2, m_1, m_3\}$,$\{m_1, m_3\}$, $\{m_2, m_3\}, \{m_3\}$, $\{m_2, m_1\}, \{m_1\}, \{m_2\}, \{\}$.

5.3.2.4 Hash divided in a set of hashes

The hashing described in Section 5.3.2.2.b might result in a process computing a hash without a message's id that has not been delivered by all processes. The

Algorithm 11: buildDependencySets

1 **Input:** M_{hash}: ids of messages to hash
1: setsToHash = {{ m'.id, $m' \in \mathrm{M}_{hash}$}}
2: **for** $i = 0; i \leq |M_{hash}| \wedge |setsToHash| \leq MaxHashes$; i++ **do**
3: setsToHash = setsToHashes $\cup$ all combinations of M_{hash} without $M_{hash}[i]$ that contain the tuples $M_{hash}[j], j > i$
4: return setsToHash[0..$MaxHashes$]

mechanism described in this section is more complex but it ensures that the ids of messages that have not been delivered yet by all processes are taken into account when computing the hash of messages.

The mechanism works in rounds. Each message m is associated to the round k in which it was broadcasted. A process aims to associate to each round at most lim_{round} messages. The hash H_m of m is composed of a set of hashes $h_{m,k}$, with $h_{m,k}$ being the hash computed with the causal dependencies of m broadcasted in round k. Each process p_i keeps a round counter $h_{r,i}$ initialized to 0. p_i appends $h_{r,i}$ on messages it broadcasts, and updates it as follows:

(1) - Upon registering lim_{round} messages m' such that $m'.h_r = h_{r,i}$. p_i increments:

$$h_{r,i} : h_{r,i} = h_{r,i} + 1$$

(2) - Upon reception of a message m' with $m'.h_r > h_{r,i}$. p_i updates:

$$h_{r,i} : h_{r,i} = m'.h_r$$

A process increments its round counter after registering lim_{round} messages in the current round **(1)**. Moreover, it updates its local round counter if it receives a message whose round counter is higher than the local one, since this means that another process registered lim_{round} messages in lower rounds **(2)**. Hence, the hash of a round is computed with on average lim_{round} message ids.

Formally, p_i builds the hash H_m of a message m it broadcasts as follows:

$$H_m = \{h_{k \leq h_{r,i},m}, \text{ where } h_{k,m}=\text{hash}(\{m',\ m' \rightarrow m \wedge m'.h_{r,m'} = h_k\})\ \}$$

h_k, the hash of round k, becomes obsolete once all processes delivered the messages that were broadcasted in round k. An obsolete hash should be removed from the system. Hence, processes exchange messages to verify that they all delivered the messages broadcasted in round k.

To acknowledge hash $h_{k,i}$, process p_i broadcasts the dependency set Dep_k associated to $h_{k,i}$. The other processes reply with a positive or negative acknowledgment. A process positively acknowledges Dep_k if it has received and delivered exactly the same messages as described in Dep_k. Otherwise, it replies with a negative

acknowledgment. The acknowledgment round is successful provided that all processes positively acknowledge the dependency set associated to $h_{k,i}$. Processes then remove the hash $h_{k,i}$ from the hashes they store, and they also remove from S_{deliv} the messages associated to round k.

In the worst case, all processes broadcast a message at the same time, and associate those messages to round k. N messages are then associated to round k, where N corresponds to the number of processes in the system. Nevertheless, on average the number of message ids associated to a round should be close to lim_{round}, since processes change the round as soon as they receive a message from a higher round or lim_{round} messages of round k.

This mechanism ensures that all of a message m's dependencies that have not been delivered by all processes are taken into account when computing the hash of m. However, it is much more complex than the first mechanism which has, as experiments show, a very good efficiency in detecting the relevant causal dependencies of messages. Nevertheless, this mechanism is more adapted to systems where it is usual that the communication delays for some messages are much higher than the average, since the first mechanism might miss some causal dependencies in these cases.

5.3.2.5 Coping with a high number of concurrent messages

The size of the M-entry clock should be chosen following the number of concurrent messages in the system. However, that value might vary dynamically during execution, reaching high values only temporarily. Consequently, taking M based on the maximum number of concurrent messages would result in an oversized clock most of the time. Moreover, estimating the maximum number of concurrent messages might be difficult or even impossible.

On the other hand, M-entry clocks become less efficient in causally ordering messages when the number of concurrent messages increases. In fact, more concurrent messages means more concurrent messages considered when building dependency sets for a message m to deliver, thus increasing the number of dependency sets to hash to find one whose hash is equal to the hash appended to m.

The idea consists of appending on a broadcasted message m the list of ids Dep of the causal dependencies of m whose clock difference with m is lower than a given value $Diff_{conc}$. A process that receives m will then consider that all messages m', $m'.id \notin Dep \wedge Diff_{m,m'} < Diff_{conc}$ are concurrent to m. The corresponding modifications of the algorithm are as follows:

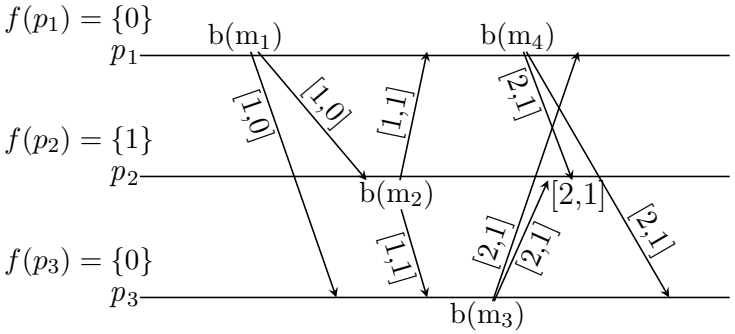


Figure 5.4: Exemple of execution of the hash-based error detector

broadcast: When process p_i broadcasts a message m, it appends on m the id of messages m' whose clock difference with m is inferior to $Diff_{conc}$: dependencies = $\{m'.id, m.V - m'.V < Diff_{conc}\}$.

Hash-based error detector: A process p_j that receives m then consider that all messages m" whose clock difference with m is smaller than $Diff_{conc}$, but whose ID is not appended to m, are concurrent to m: $m" \notin dependencies \wedge Diff_{m,m"} < Diff_{conc} \Rightarrow m||m"$.

p_i fixes the value of $Diff_{conc}$ following the average number of messages that are concurrent to each other at any given moment, since more concurrent messages means that the concurrent messages of m have a higher clock difference with m.

5.3.2.6 Example

Figure 5.4 shows an execution that uses the hash-based error detector. The Figure represents 3 processes which use a constant size clock of 2 entries. Processes p_1, p_2, and p_3 are respectively associated with the vector entries $f(p_1) = \{0\}$, $f(p_2) = \{1\}$, and $f(p_3) = \{0\}$. We consider that the error detector returns false for the messages m_1, m_2, and m_3. Consequently, processes deliver the three messages at reception (not shown in the Figure for readability's sake). The delivery conditions of m_4 are satisfied upon its reception by p_2, since $m_4.V = [2, 1]$ and $V_2 = [2, 1]$.

Let's show how the error detector of p_2 computes dependency sets for m_4. The hash-based error detector of p_2 builds dependencies sets with messages that p_2 already delivered (m_1,m_2, and m_3). The possible dependency sets are: $\{m_1\}$,$\{m_2\}$,$\{m_3\}$, $\{m_1, m_2\}$, $\{m_1, m_3\}$,$\{m_2, m_3\}$, and $\{m_1, m_2, m_3\}$.

The error detector sorts m_1, m_2 and m_3 following their clock difference with m_4. We have: $Diff_{m_4,m_1} = 3 - 1 = 2$, $Diff_{m_4,m_2} = 3 - 2 = 1$ and $Diff_{m_4,m_3} = 3 -$

$3 = 0$. Therefore, the error detector sorts them from the lowest probability of being concurrent to m_4 to the highest probability: $\{m_1, m_2, m_3\}$.

The actual dependencies of m_4 are $Dep_{m_4} = \{m_1, m_2\}$ and $h(Dep_{m_4}) = H_{m_4}$. The error detector first tries to hash the dependency set $\{m_1, m_2, m_3\}$. However, $h(\{m_1, m_2, m_3\}) \neq H_{m_4}$. Therefore, it removes the message with the highest probability of being concurrent to m_4, which is m_3, and computes the hash of $\{m_1, m_2\}$ which is equal to H_{m_4}. Therefore, p_2 delivers m_4 since it found a dependency set whose hash is equal to H_{m_4}.

5.3.3 Experimental results

Experiments were carried out on the OMNeT++ simulator. Processes generate messages on a regular interval depending on the system's message load, with a deviation following a normal distribution N(0,10)ms. Communication delays follow a normal distribution N(100,30)ms. The system contains 500 processes that use a probabilistic clock [MW17b] of *M=50* entries to causally order messages. The number of entries associated to each process $|f|$ is computed for each experiment following the formula proposed by Mostéfaoui and Weiss [MW17b] described in Section 5.2. We set *MaxHashes=200* and compute *Diff* with the formula given in Section 5.3.2.2.a. The accuracy of probabilistic clocks to causally order messages is mostly impacted by the number of concurrent messages in the system, which in its turn is determined by the system's message load and the communication delays. Since the message load is easier to set than the number of concurrent messages, we fix the communication delays and vary the message load in the experiments.

Out of causal order deliveries are controlled by a controller module that verifies the causal delivery of each message. It associates a vector clock of size N to processes and messages, with which it verifies the causal delivery of messages by processes. Processes notify the controller when broadcasting and delivering a message, but they do not use its information.

Some experiments compare the hash-based error detector with the error detector proposed by Mostéfaoui and Weiss [MW17b]. In the following we denote the first as *HD* and the second as *MW*.

Experimental results analyze the following characteristics:

(1) - Out of causal order deliveries detected by *HD* and *MW*.

(2) - Rate of false positives of *HD* and *MW*.

(3) - Cost of *HD* in terms of hash computation following *Diff* and *MaxHashes*.

(4) - Efficiency of *HD*'s mechanism to handle high message loads.

5.3.3.1 Out of causal order deliveries detected by *HD* and *MW*

The first series of experiments measured the number of out of causal order deliveries detected by *MW* and *HD* for message loads going from 10 up to 150 messages broadcasted per second. Table 5.1 gives the results. It is organized in two parts. The first part gives the number and percentage of messages that are delivered out of causal order when delivering messages at reception without any control and when implementing causal broadcast with probabilistic clocks ($M = 50$) respectively. The second part gives the number of out of causal order deliveries detected by *MW* and *HD* respectively.

First, we observe that the number of messages delivered out of causal order increases with the message load for both when delivering without control and when implementing causal broadcast with probabilistic clocks. Increasing the message load increases the number of causal dependencies of each message m, which in its turn increases the probability that at least one of them is not received upon the reception of m. Thus, increasing the message load increases the number of messages delivered out of causal order when delivering them without control. Mostéfaoui and Weiss showed in [MW17a] that the efficiency of probabilistic clocks to causal order messages decreases with the message load.

Second we observe that *HD* detects all out of causal order deliveries, while *MW* only detects a part of them. We observe that the higher the message load, the higher the percentage of out of causal order deliveries detected by *MW*. In fact, the higher the message load, the higher the probability that *MW*'s condition is false on a message m broadcasted by a process p_i ($\exists x \in f(p_i)V[x] == V_m[x] - 1$).

Message load	Out of causal order deliveries		Detected	
msg/sec	No control	PC	MW	HD
10	4552 (0.45 %)	0	0	0
25	33010 (1.3 %)	0	0	0
50	116094 (2.3 %)	211 ($4.2*10^{-3}$ %)	0	211
75	250187 (3.3 %)	2856 ($3.8*10^{-2}$ %)	10	2856
100	410802 (4.1 %)	13630 (0.14 %)	401	13630
125	603532 (4.8 %)	31472 (0.25 %)	2500	31472
150	824012 (5.5 %)	68829 (0.45 %)	7256	68829

Table 5.1: Detected out of causal order deliveries following the system's message load

5.3.3.2 Rate of returned false positives of *HD* and *MW*

An error detector returns a false positive when it wrongly concludes that the process delivered a message out of causal order. This section measures the rate of false positives for *HD* and *MW* for message loads going from 10 up to 150 messages per second.

Table 5.2 gives the percentage of true and false negatives of *HD* and *MW* for the experiments of Section 5.3.3.1. Results show that *HD* returns much less false positives than *MW*. Moreover, as expected the percentage of messages on which both error detectors return a false positive increases with the message load.

HD returns almost no false positive for a message load up to 50 messages broadcasted per second. For a message load of 75 messages broadcasted per second it returns almost as much false than true positives. When the message load increases further, *HD* returns 10 times more false positives than true positives. The false positive rate of *HD* increases because its efficiency to detect concurrent messages decreases when the message load increases. Hence, the number of concurrent messages taken into account during the hash computation increases. This in its turn increases the probability that the error detector does not find a dependency set whose hash is equal to the hash of a message in the given number of accepted hashes, even if the process has delivered all of the message's causal dependencies.

MW begins to return false positives with a message load of 25 messages broadcasted per second, with a rate comparable with the rate of false positives returned by *HD* for a message load of 75 messages broadcasted per second. The false positive rate of *MW* increases with the message load because the probability that its condition $(\exists x \in f(p_i)V[x] == V_m[x] - 1)$ is true on a message m broadcasted by a process p_i increases with the message load. In fact, the probability that all entries of $f(p_i)$ are incremented by messages concurrent to m increases with the message load.

The value returned by error detectors could be used by further handling messages following that value. A process could for example delay the delivery of messages tagged as not causally ordered and retrieve their causal dependencies, in order to ensure their causal delivery, as does the next contribution. Hence, the number of false positives should be kept as low as possible. We observe that *HD* has a better ratio of true/false positives than *MW*, especially for message loads lower or equal to 75 messages broadcasted per second.

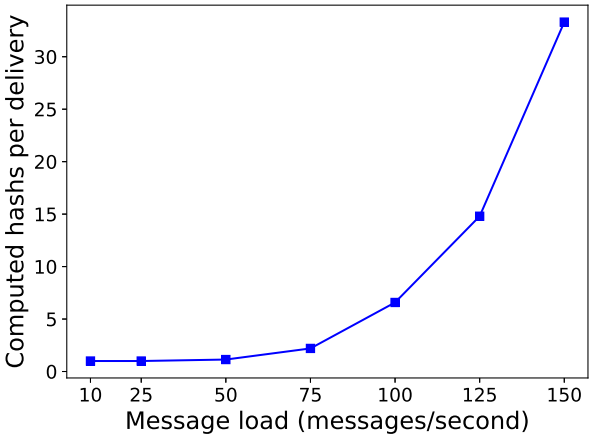


Figure 5.5: Distribution of computed hashes following the message load

Message load	MW		HD	
msg/sec	True	False	True	False
10	0	0	0	0
25	0	$1.1*10^{-2}\%$	0	0
50	0	1 %	$4.2*10^{-3}\%$	$4.2*10^{-5}\%$
75	$8.7*10^{-4}\%$	5.2 %	$4.9*10^{-2}\%$	$5.5*10^{-2}\%$
100	$3.9*10^{-2}\%$	9.2 %	0.14 %	1.4 %
125	$2.1*10^{-2}\%$	15 %	0.25 %	5%
150	0.18 %	28 %	0.85 %	9.5%

Table 5.2: True and false positives following the message load

5.3.3.3 Cost of *HD* in terms of hash computations

This section measures the cost of the hash computations for a message m done by *HD*. It is divided in two parts: (1) the average number of hashes to find a dependency set whose hash is equal to the one appended on m and (2) the cost of computing a hash. We first analyze the cost of (1), then we analyze the cost of (1) and (2) when computing hashes with the technique presented in Section 5.3.2.2.a.

5.3.3.3.a Average number of hashes computed per message delivery

Figure 5.5 gives the average number of computed hashes per message delivery for a message load that goes from 10 up to 150 messages broadcasted per second. For message loads up to 75 messages broadcasted per second, *HD* succeeds to find

the hash of messages with on average less than 5 hashes per message delivery. However, the number of required hashes begins to increase exponentially when further increasing the message load, because the number of concurrent messages taken into account in the hash computation increases.

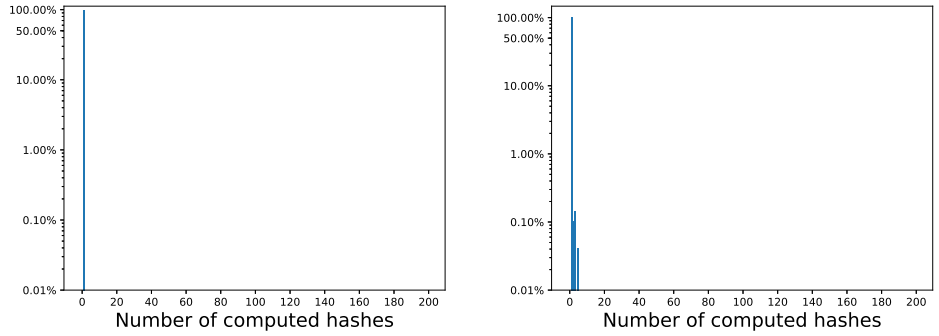
As explained in Section 5.3.2.3, processes should compute at most an upper limit of hashes, denoted *MaxHashes*, when computing hashes for a message m of hash H_m. The following experiments aim to find the best value of *MaxHashes*. The higher the value of *MaxHashes*, the higher the probability of finding a dependency set whose hash is equal to H_m. However, the higher *MaxHashes*, the higher is also the number of computed hashes. Hence, the value chosen for *MaxHashes* is a trade-off between finding a dependency set whose hash is equal to H_m and the number of computed hashes to find such a set.

The first series of experiments aims to analyze, for several messages loads, the number of hashes *HD* requires to find a dependency set for messages. To that end, we set *MaxHashes* to a high value (*MaxHashes=200*). Figure 5.6 gives, for several message loads, the distribution of the number of hashes *HD* computes. For example, Figure 5.6a shows that with a message load of 10msg/s *HD* only computes 1 hash in 100% of the hash computations.

We observe that with a message load of up to 50 messages/second, *HD* computes few hashes. The higher the message load, the higher the maximal number of hashes *HD* computes, and the higher also the number of hash computations that require a higher number of hashes. This is because detecting concurrent messages becomes less efficient when the message load increases. Thus, the percentage of hash computations in which *HD* computes 50-200 hashes increases with the message load, because *HD* takes into account more and more concurrent messages in the hash computation. Hence, the value of *MaxHashes* should be chosen following the message load.

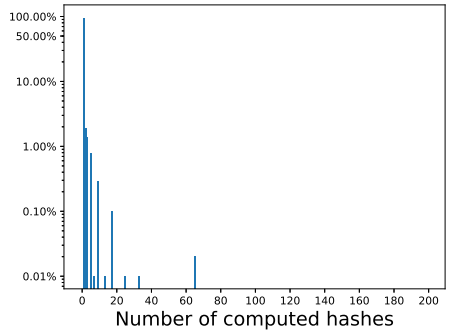
MaxHashes	Average computed hashes per delivery		False positives	
	125 msg/s	150 msg/s	125 msg/s	150 msg/s
10	4.7	6.9	17.4%	39%
25	6.7	11.7	11.4%	28.2%
50	9.1	17.9	7.3%	20%
100	12.1	26.4	4.7%	14.1%
150	14.2	33.0	2.9%	10.3%
200	15.8	38.2	2.75%	9.5%
250	17.3	43.6	2.7%	9.4%

Table 5.3: Average computed hashes per delivery and false positive rate for different values of *MaxHashes* for a message load of 125 and 150 msg/s

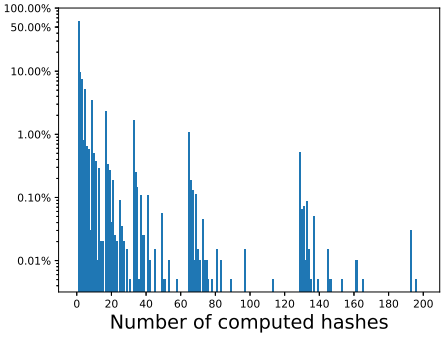


(a) Message load = 10msg/s

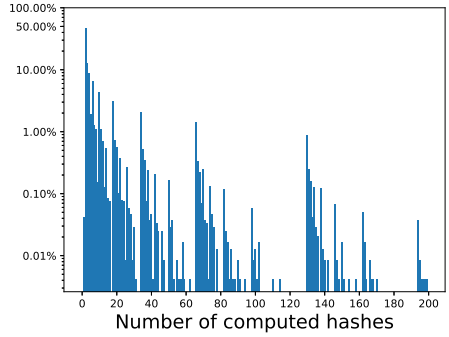
(b) Message load = 25msg/s



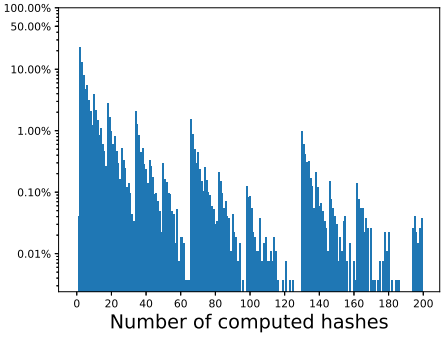
(c) Message load = 50msg/s



(d) Message load = 100msg/s



(e) Message load = 125msg/s



(f) Message load = 150msg/s

Figure 5.6: Distribution of computed hashes following the message load

The second series of experiments aims to analyze the effect of reducing *MaxHashes* when the message load is high. Reducing *MaxHashes* reduces the number of computed hashes per message delivery, but it also increases the false positive rate, i.e., messages m of hash H_m for which the error detector does not find a dependency set whose hash is equal to H_m while the process delivered all causal dependencies of m. Table 5.3 gives the results when varying *MaxHashes* from 10 up to 250 hashes for respectively a message load of 125 and 150 messages broadcasted per second.

We observe that increasing *MaxHashes* increases the average number of computed hashes per delivery, but it also decreases the rate of false positives. However, over a given limit increasing *MaxHashes* only slightly decreases the rate of false

positives while it continues to increase the average number of hashes computed per message delivery. Further increasing *MaxHashes* increases the average number of computed hashes per message delivery without decreasing the rate of false positives at the same rate, mostly because it increases the number of hashes processes try in unsuccessful hash computations. In other words, increasing *MaxHashes* increases the number of messages for which processes find a dependency set, but many of the additional computed hashes are used in unsuccessful hash computations. For example, in Table 5.3 increasing *MaxHashes* from 200 to 250 for a message load of 150 msg/s results in computing on average 5.4 hashes more per message delivery, while the error detector only returns 0.1% less false positives.

The lower the value of *MaxHashes* the faster increases the rate of false positives. For example, with a message load of 150 messages per second, decreasing *MaxHashes* from 150 to 100 hashes results in a false positive rate that increases of 3.8%, while decreasing *MaxHashes* from 25 to 10 hashes results in a false positive rate that increases of 10.8%. A low value of *MaxHashes* results in a high false positive rate, like the false positive rate of 39% when *MaxHashes* is set to 10 hashes for a message load of 150 messages/s. This is consistent with the hash distribution observed in Figure 5.6.

The best value for *MaxHashes* for both experiments seems to be between 150 and 200 hashes. Over 200, the rate of false positive only decreases slightly. Under 150, the rate of false positives increases fast. In any case, it should not be set too low because this leads to a high rate of false positives. To conclude, *MaxHashes* should be set following the message load: with a low message load *MaxHashes* can be set very low, as for example *MaxHashes=2* when 10 messages are broadcasted inside the system per second. On the other hand, *MaxHashes* should be set to a higher value when the message load increases and the detection of concurrent messages becomes less efficient, as for example *MaxHashes* should be set between 150 and 200 hashes in the examples of Table 5.3. The value of *MaxHashes* can also be set lower at the cost of a higher false positive rate, but it should not be set too low because the false positive rate will then be high, as for example 39% when *MaxHashes=10* and the message load is of 150 messages broadcasted per second.

5.3.3.3.b Cost of computing a hash

As explained in Section 5.3.2.2.a, a process p_i computes the hash of a message m it broadcasts/delivers with only the causal dependencies of m whose clock difference with m is lower than a given value *Diff*. This section analyzes the number of detected out of causal order deliveries following the chosen value of *Diff*. We also analyze the cost of computing a hash and the average number of hash computations per message delivery.

We determine the base value of *Diff* following the formula given in Section 5.3.2.2.a, consider a message load m_{load} of 150 messages broadcasted per second and $|f| = 4$ following the formula proposed in [MW17b]. The maximal network delay is assumed to be of 150ms and $n_{conc} = 10$. Those parameters give the following value of *Diff*: *Diff*=maxDelay*m_{load}*$|f| + n_{conc} * |f|$=0.15*150*4 + 10*4 = 130

Table 5.4 gives the results when varying *Diff* from a multiple of 1.25 (162) down to 0.1 (13). The table shows the percentage of detected out of causal order deliveries, as well as the average number of hashes computed per message delivery and the average number of operations done to compute a hash. It also gives the average cost of a hash computation, which corresponds to the average number of hashes per message delivery multiplied by the average number of operations done to compute a hash.

We observe that out of causal order deliveries are all detected for a *Diff* value down to 117. Decreasing *Diff* further results in out of causal order deliveries that are not detected. First, only a few of them are not detected, but their number rapidly increases when further reducing *Diff*. The reason is that most not received causal dependencies of an out of causal order delivered message m have a clock difference with m that is in a given window. In fact, the higher the value of $Diff_{m,m'}$ the higher the probability that processes have delivered m'. Thus, decreasing *Diff* first removes from the hash computation the ids of messages that are most probably delivered by processes.

The cost of computing the hash of a dependency set is equal to the number of message ids that the considered dependency set contains. Fore example, when *Diff=130*, processes take on average into account 16.7 message ids to compute a hash, and computing a hash therefore requires on average 16.7 operations. The cost per hash increases with *Diff*. As expected, the higher the value of *Diff*, the higher the number of message ids taken into account in the hash computation, thus increasing the cost of computing a hash. We note that with *Diff=85*, only 2.67 message ids are on average taken into account in the hash computation, while *HD* still detects 91% of out of causal order deliveries.

The average number of computed hashes per message delivery is stable (25.5 hashes/delivery) till *Diff=117*. It begins to decrease when *Diff=98*, and it accelerates when further decreasing *Diff*. The main reason is that decreasing *Diff* reduces the number of messages that are taken into account when computing the dependency set of messages. Hence, a process will compute the hash of fewer dependency sets, thus decreasing the number of hashes per message delivery. For example, processes consider on average only 5.5 message ids per hash computation when *Diff=98*,

while they consider on average 12.7 message IDs when *Diff=117*. Processes consider almost no message ids when *Diff=65* or lower, thus giving a very low number of computed hashes per message delivery.

Diff	Detected out of causal order deliveries	Hashes per delivery	Operations per hash	Cost per hash in operations
162(*1.25)	100%	25.5	24.7	629
130	100%	25.6	16.7	427
117(*0.9)	100%	25.4	12.7	322
98(*0.75)	99.9% (-4)	24.2	8.9	215
85(*0.65)	99.7%(-187)	16.6	5.5	91
65(*0.5)	91%(-5613)	4	2.67	10.5
26(*0.2)	11.9%(-60953)	1	1.03	1
13(*0.1)	0.2%(-69028)	1	1.005	1

Table 5.4: Effect of varying *Diff* on the percentage of detected out of causal order deliveries and cost of hash computations

To conclude, the formula to compute *Diff* is accurate and *HD* detects 100% of out of causal order deliveries even when computing the hash of messages with only the ids of a subset of causal dependencies. The cost per hash is low or similar to the cost of comparing M-entry clocks, with only 16.7 operations required to compute a hash with a message load of 150 messages broadcasted per second. However, the parameters considered when computing *Diff* might change over time, such as the maximum communication delay, and a too low value for *Diff* rapidly results in many not detected out of causal order deliveries. Hence, *Diff* should be chosen sufficiently high to tolerate a variation of the communication delays and the number of concurrent messages in the system.

5.3.3.4 Handling a high message load

This section analyzes the mechanism presented in Section 5.3.2.5 to handle high message loads, which consists of appending to messages the ids of some of their causal dependencies. We analyze the rate of *true* and *false* positives as well as the number of computed hashes per message delivery when using this mechanism. Finally, we analyze the cost of this mechanism which is determined by the size of the set of message ids appended to messages.

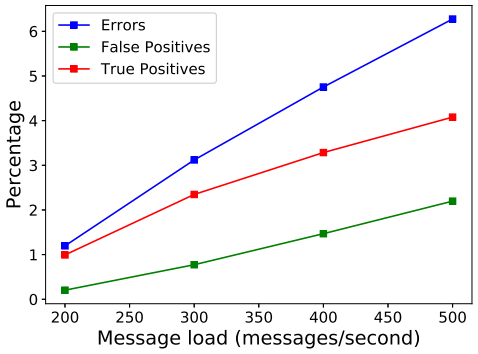


Figure 5.7: Rate of *true* and *false* positives following the message load

True and false positives Figure 5.7 shows the rate of *true* and *false* positives as well as the combined error rate when increasing the message load from 200 to 500 messages broadcasted per second.

We first observe that the rate of *true* positives increases with the message load, which is explained in Section 5.3.3.1. The rate of *false* positives also increases with the message load. Nevertheless, it increases at the same rate as the number of *true* positives, as we can see in Figure 5.7 where the curves of *true* and *false* positives are almost parallel. Finally, with a message load of 500 messages per second *true* positives still represent more than 50% of the detected errors.

We conclude that the mechanism handles high message loads well. When using it, the error detector yields a low or acceptable rate of *false* positives. However, this mechanism also has its limits since it eventually yields a substantial amount of false positives, as for example when the message load is of 500 messages broadcasted per second, even though it stays lower than the yielded rate of *true* positives.

Computed hashes per delivery Figure 5.8 shows the average number of computed hashes per message delivery. We observe that it increases with the message load, going from ≈ 3.2 hashes/delivery for a message load of 200 messages/second up to 13.8 hashes/delivery for a message load of 500 messages/second. A more precise analysis shows that the increase comes from unsuccessful hash computations. In fact, for all the message loads, the successful hash computations are done ine 1 hash. However, Figure 5.7 shows that the rate of *false* positives increases with the message load. For each *false* positive, the error detector computes up to 200 hashes. Hence, the increase in terms of *false* positives explains the increase of the average number of computed hashes per message delivery. Nevertheless, it stays acceptable with 13.8 hashes per messages delivery for a message load of 500 messages broadcasted per second.

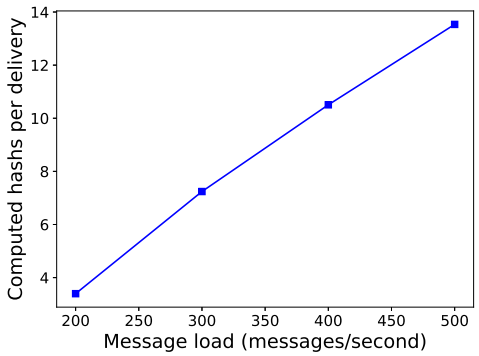


Figure 5.8: Average number of computed hashes per message delivery following the message load

We conclude that the number of computed hashes per message delivery is acceptable when using the mechanism to handle high message loads, with below 15 hashes per message delivery computed for a message load going up to 500 messages broadcasted per second. However, the number of computed hashes per message delivery increases linearly with the message load and would eventually become prohibitive.

Size of the message ids set appended to messages Figure 5.9 shows the number of message ids appended to messages following the message load. As expected, the number of ids appended to messages linearly increases with the message load. With a message load of 500 messages/second the set contains on average 34 message ids. A message id is composed of two integers, and the additional causal information appended to messages therefore has a size of 68 integers, which seems to be acceptable.

We conclude that the size of the set of message ids seems to be acceptable for up to 500 messages broadcasted per second. However, the size of the set grows linearly with the message load and will eventually become prohibitive.

5.4 Retrieve the causal dependencies of messages

This section presents a short algorithm to be executed in conjunction with an error detector, and which retrieves the causal dependencies of a message. The idea is that a process executes the error detector before delivering a message m, and if it returns *true*, i.e., m has some causal dependencies that are not delivered locally, then the process executes the algorithm to retrieve the causal dependencies of m, in order to ensure its delivery in causal order.

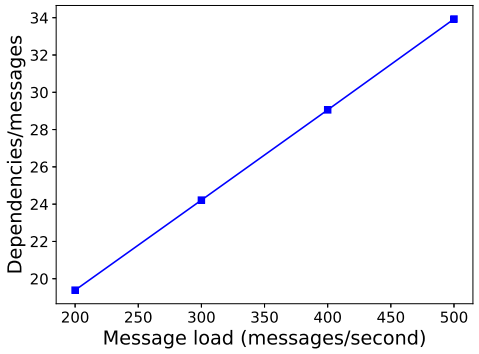


Figure 5.9: Size of the set of dependency IDs appended to messages following the message load

Process p_i either keeps the causal dependencies of a message m it broadcasts whose clock difference with m is smaller than *Diff*, i.e., similar to Section 5.3.2.2.b, or based on the hash round in which they were sent, i.e., similar to Section 5.3.2.4. p_i maintains two lists to store the causal dependencies of its broadcasted messages:

- Dep_{next}: contains the ids of the causal dependencies of its next broadcasted message.
- S_{dep_i}: list containing, for each of its unacknowledged broadcasted messages, the tuple (seq, Dep), where seq is the sequence number p_i associated to the message m and Dep contains the ids of the causal dependencies of m.

Broadcast When broadcasting a message m, p_i stores in S_{dep_i} the list Dep_{next} containing the ids of m's causal dependencies, in order to send them to processes that request them later. Moreover, p_i sets $Dep_{next} = \{m.id\}$, since m can resume them as a direct dependency [PRS96] of the next message p_i will broadcast.

Request of causal dependencies A process p_j requests the causal dependencies of m if its error detector returns *true* on m. p_j then sends a REQ message to p_i containing the id of m, to which p_i replies with a RSP message containing the ids of the causal dependencies Dep_m of m. Upon reception of the RSP message, p_j delivers m after it has delivered all messages whose id is contained in Dep_m.

Example Figure 5.10 shows a scenario where process p_2 requests the causal dependencies of m_4. The system contains three processes using a probabilistic clock of two entries. Processes p_1, p_2, and p_3 are respectively associated with the vector entries $f(p_1) = \{0\}$, $f(p_2) = \{1\}$, and $f(p_3) = \{0\}$. Let's consider that the error

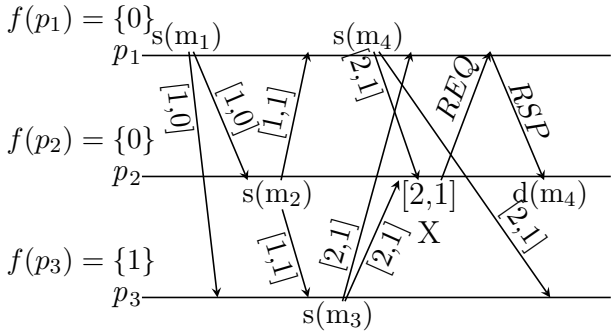


Figure 5.10: Request of message m_4's dependencies

detector of p_2 returns *false* on the messages m_1, m_2 and m_3, and that they are therefore delivered at reception (not shown in the Figure for readability's sake). Therefore, when p_2 receives m_4, the value of its clock is $V_2 = [2, 1]$ and the value of m_4's clock is $m_4.V = [2, 1]$. Let's assume that p_2 executes the error detector described in Algorithm 9 on m_4, which returns *true*, since $\nexists i \in f(p_1), V_2[i] = m_4.V[i] - 1$. Consequently, p_2 requests the causal dependencies of m_4 by sending a *REQ* message to p_1, which replies with a *RSP* message that contains m_4's causal dependencies, i.e., $\{m_1, m_2\}$. Process p_2 delivers m_4 at reception of *RSP*, since it has already delivered $\{m_1, m_2\}$.

5.4.1 Domino effect

Requesting a message's causal dependencies can lead to a domino effect by inducing the request of the causal dependencies of messages in cascade. In fact, when process p_i requests the causal dependencies of a message m, it sends a *REQ* message to the sender of m, then waits for the sender's reply message *RSP*. During that message exchange, p_i might receive several messages m' of which m is a causal precedence ($m \rightarrow m'$). The delivery conditions of some of those m' messages might be satisfied before the end of the *REQ/RSP* message exchange for m.

However, in the case of the hash-based error detector, p_i's error detector will not include m when building dependency sets of m', because p_i has not delivered m yet. Since $m \rightarrow m'$, the error detector will only find a dependency set whose hash value is equal to the one attached to m' in case of a collision. However, since the probability of collisions is very low, there is a high probability that p_i will request the causal dependencies of m'. Similarly, during the *REQ/RSP* message exchange related to m', p_i might receive another message m'' of which m' is a causal dependency, leading to another *REQ/RSP* message exchange, and so on.

To avoid such a domino effect, processes execute the request of causal dependencies sequentially, i.e., a process begins a dependency request provided that the previous one is finished. At the end of a *REQ/RSP* message exchange, the requesting process p_i has most probably delivered all causal dependencies of m, and will therefore deliver m. Hence, m will be taken into account when computing dependency sets for m', henceforth avoiding the request of the causal dependencies of m'. Processes should request causal dependencies rarely, and very few causal dependencies requests should occur simultaneously on the same process. Consequently, the overhead of sequential requests is negligible.

When using the hash-based error detector, a domino effect can occur when process p_i broadcasts a message m' during the *REQ/RSP* message exchange related to m. In fact, during the message exchange p_i will deliver fewer messages since it has a high probability not to deliver the messages of which m is a causal dependency. Therefore, the clock difference between m' and newly broadcasted messages will be higher and m' has therefore a higher probability to be considered as a causal dependency of messages to which it is concurrent. This in its turn will lead to a request of m' causal dependencies, and so on... Hence, processes delay the broadcast of messages until the end of dependency requests.

5.4.2 Liveness proof

This section gives the liveness proof of the algorithm retrieving the causal dependencies of messages used in conjunction with the hash-based error detector when implementing causal broadcast with M-entry clocks.

Theorem 5.3. *A well-formed message is eventually delivered by all processes.*

Proof. We assume that a well-formed message eventually delivered when using constant size clocks to causally order messages. For example, Mostéfaoui et Weiss proved in [MW17b] that the delivery conditions of probabilistic clocks attached to well-formed messages are always satisfied. Thus, we only need to prove that the liveness property holds when using an error detector and retrieving the causal dependencies of messages for which the error detector returns that they might not be causally ordered. The proof is done by induction.

H0: Any message m generated on the initial state is eventually delivered.

A process p_i delivers m if its error detector returns *false*. Otherwise, p_i requests m's causal dependencies to the sender of m, which replies by sending a *RSP* message containing an empty set, since m was generated on the initial state and

has, therefore, no causal dependencies. Hence, p_i delivers m upon reception of the RSP message. Hence, all processes eventually deliver m.

H1: We assume a set S of messages that are eventually delivered by all processes. A message m generated after the delivery of messages of the set S will be eventually delivered by all processes.

A process p_i delivers m if its error detector returns *false*. Otherwise, p_i requests m's causal dependencies to p_j, the sender of m, which replies by sending a RSP message containing the set S, since p_j generated m after delivering the messages of the set S. Hence, p_i delivers m upon reception of the RSP message, since p_i delivered the messages of the set S by hypothesis. Hence, all processes eventually deliver m.

Thus, a message generated after a set of eventually delivered messages is eventually delivered by all processes (H1). As processes eventually deliver messages generated in the initial state (H0), we conclude that processes eventually deliver all messages.

□

5.4.3 Removing obsolete causal information

This section presents an acknowledgment mechanism to remove obsolete causal information about broadcasted messages. A process keeps the causal information of messages it broadcasts in order to send it to processes that request it. The causal information of a message m becomes obsolete once all processes delivered m.

A process p_i can only remove the causal information about a message m it broadcasted once all processes acknowledged m's delivery. However, M-entry clocks do not characterize causal order and cannot, therefore, contain such information. Hence, the acknowledgment of the delivery of messages requires an additional mechanism.

A trivial acknowledgment mechanism consists of each process periodically broadcasting a vector of N entries to acknowledge the messages it delivered, where N corresponds to the number of processes in the system. However, this solution does not scale. The following two presented acknowledgment mechanisms work in rounds and scale much better. The first acknowledge mechanism requires only one process to send a message of size $\mathcal{O}(N)$, while other processes send a small message containing a few integers. The second mechanism requires processes to send only one small messages containing a few integers, by computing the causal dependencies of messages locally with the hash-based error detector.

5.4.4 First acknowledge mechanism

Algorithm 12 describes the first acknowledge mechanism. Process p_i starts the acknowledgment round by broadcasting a message ACK containing the id (p_i, seq) of one of the messages m it broadcasted, as well as the vector V_m containing the causal dependencies of m (*line 1*). The other processes reply to the ACK message with an ACK_{rep} message to acknowledge or not the delivery of m, by using the function *hasDelivered*(p_i, *seq*), which returns true provided that the process has delivered m and false otherwise.

The acknowledgment round ends at a process p_j once p_j received the ACK and all ACK_{rep} messages. The round is successful if all processes sent a positive acknowledgment for m (*line 6*). Since V_m resumes the causal dependencies of m, if $(p_j, seq_j) \in V_m$, then messages broadcasted by p_j with a sequence number $seq' \leq seq_j$ are also causal dependencies of m, and are therefore also acknowledged by this round. Hence, process p_j removes from S_{dep_j} the messages with a sequence number $seq_m < seq_j$. The acknowledgment round fails if one process does not acknowledge the delivery of m. No process can then clear its S_{dep}.

Any process can start an acknowledgment round, provided that it broadcasted at least one message. A process should choose a message m to acknowledge such that the probability is high that all processes have delivered m.

Algorithm 12: Acknowledge round for message $m = (p_i, seq)$ of causal dependencies V_m

p_i **starts the acknowledge round**
1: broadcast(ACK,(p_i, seq),V_m)
2: recvAck((p_i, seq),V_m)
Upon reception of (ACK,(p_j, seq),V_m)
3: broadcast(ACK_{rep},p_j, hasDelivered(p_j,seq))
4: recvAck((p_i, seq),V_m)
recvAck((p, seq)**,**V_m**)**
5: waitUntil(received ACK_{rep} from all $p_k \neq p$)
6: **if** allProcDelivered (p, seq) **then**
7: S_{dep_i}=S_{dep_i}\{$(seq', dep'), seq' \leq seq'' : (p_i, seq'') \in V_m$}

Figure 5.11 shows a successful acknowledgment round. Process p_1 starts the round by broadcasting an ACK message containing the id of m_4 $(p_1, 2)$ and m_4's causal dependencies $\{(p_1, 1), (p_2, 1)\}$ which correspond to m_1 and m_2. Processes p_2 and p_3 reply to the ACK message with an ACK_{rep} message containing $(p_1, 2, true)$, since both delivered m_4. After receiving the ACK_{dep} messages from all other processes confirming the delivery of m_4, p_1 deletes the causal information about m_1 and m_4 from S_{dep_1}. In its turn, upon reception of all ACK_{dep} messages, p_2 deletes from

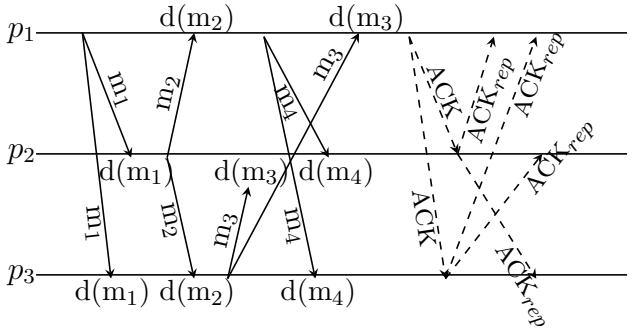


Figure 5.11: Successful acknowledgment round

S_{dep_2} the causal information of m_2, since $m_2 \in \{(p_1, 1), (p_2, 1)\}$. However, even though all processes delivered m_3, p_3 cannot delete the causal information of m_3 from S_{dep_3}, because m_3 does not causally precede m_4.

The acknowledgment round fails in a slightly modified execution of Figure 5.11, where p_3 receives m_4 after the ACK message. In this case, upon the reception of the ACK message, p_3 has not delivered m_4 yet. Hence, it does not acknowledge the delivery of m_4. Thus, not all processes acknowledge the delivery of m_4, and no causal information can be removed from processes' S_{dep} during this round. Nevertheless, a new acknowledgment round can always be started right after the end of a failed one if necessary.

The size of the ACK message can be reduced by not appending on it the whole vector V_m. The acknowledgment round will then only acknowledge messages from processes p_j such there is an entry (p_j, seq) in V_m. For example, if only some processes p_j broadcast messages, then only the tuples (p_j, seq_j) should be included in V_m. Another example would be to include in V_m only the tuple $(p_j seq_j)$ once p_j has k messages to acknowledge.

5.4.5 Second acknowledge round mechanism

Algorithm 13 describes the second acknowledgment mechanism. It uses the hash-based error detector. Process p_i starts the acknowledgment round by broadcasting a message ACK containing the id (p_i, seq) of one of the messages m it broadcasted, as well as the hash H_m of m's causal dependencies (*line 1*). The other processes reply to the ACK message with an ACK_{rep} message to acknowledge or not the delivery of m, by using the function *hasDelivered(p_i, seq)*, which returns true provided that the process has delivered m and false otherwise.

The acknowledgment round ends at a process p_j once p_j received the ACK and all ACK_{rep} messages. The round is successful if all processes sent a positive acknowledgment for m (*line 6*). Process p_j then uses H_m to compute the causal dependencies set Dep_m of m. These messages are also acknowledged by the acknowledgment of m. Hence, if a message m' of id $(p_j, seq_j) \in Dep_m$, then p_j removes from S_{dep_j} the messages with a sequence number $seq_m < seq_j$. The acknowledgment round fails if one process does not acknowledge the delivery of m. No process can then clear its S_{dep}.

Any process can start an acknowledgment round, provided that it broadcasted at least one message. A process should choose a message m to acknowledge such that the probability is high that all processes have delivered m.

Algorithm 13: Acknowledge round for message $m = (p_i, seq)$ of hash H_m

p_i **starts the acknowledge round of** m
1: broadcast(ACK,(p_i, seq),H_m)
2: recvAck((p_i, seq),H_m)
Upon reception of (ACK,(p_j, seq)**,**H_m**)**
3: broadcast(ACK$_{rep}$,p_j, hasDelivered(p_j,seq))
4: recvAck((p_i, seq),H_m)
recvAck((p, seq)**,**H_m**)**
5: waitUntil(received ACK$_{rep}$ from all $p_k \neq p$)
6: **if** allProcDelivered (p, seq) **then**
7: Compute Dep_m with H_m
8: S_{dep_i}=$S_{dep_i} \backslash \{(seq', dep'), seq' \leq seq'' : (p_i, seq'') \in Dep_m\}$

5.4.6 Experiments

This section evaluates the algorithm to retrieve causal dependencies used in conjunction with the hash-based error detector (see Section 5.3.2) in a system implementing causal broadcast with M-entry clocks.

Experiments were carried out on the OMNeT++ simulator. Processes generate messages on a regular interval depending on the system's message load, with a deviation following a normal distribution N(0,10)ms. The communication delays follow a normal distribution N(100,30)ms. The system contains 500 processes that use a probabilistic clock [MW17b] of *M=50* entries to causally order messages. The number of entries associated to each process $|f|$ is computed for each experiment following the formula proposed by Mostéfaoui and Weiss [MW17b] described in Section 5.2. We set *MaxHashes=200* and compute *Diff* with the formula given in Section 5.3.2.2.a. The accuracy of probabilistic clocks to causally order messages is mostly impacted by the number of concurrent messages in the system, which in its

turn is determined by the system's message load and the communication delays. Since the message load is easier to set than the number of concurrent messages, we fix the communication delays and vary the message load in the experiments.

Out of causal order deliveries are controlled by a controller module that verifies the causal delivery of each message. It associates a vector clock of size N to processes and messages, with which it verifies the causal delivery of messages by processes. Processes notify the controller when broadcasting and delivering a message, but they do not use its information.

We evaluate the following characteristics of the algorithm to retrieve the causal dependencies of messages:

- Number of messages delivered out of causal order when retrieving the causal dependencies of tagged messages.
- Limits of the algorithm.

5.4.6.1 Messages delivered out of causal order

The first series of experiments measured the number of out of causal order deliveries when retrieving the causal dependencies for a message load going from 10 up to 150 messages broadcasted per second. We executed the same experiments as in Section 5.3.3.1. Table 5.5 gives the results.

We observe that the hash-based error detector detected all messages that are not causally ordered. The process requested the causal dependencies of the tagged messages m, and upon receiving them only delivered m after it had delivered all of m's causal dependencies. We observe in Table 5.5 that processes delivered no message out of causal order for a message load up to 75 messages broadcasted per second. For higher message loads, marked by an X, we observe that processes request the causal dependencies of many messages, thus continuously delaying the delivery of messages. Hence, messages will be delayed more and more, rendering the mechanism not effective. In fact, the algorithm does a message exchange in order to retrieve the causal dependencies of messages m tagged by the error detector. During that message exchange, the process will most probably not deliver messages of which m is a causal dependency. Hence, when the message load is too high, processes will be recovering the causal dependencies of messages all the time, delaying more and more the delivery of messages.

We conclude that the algorithm to retrieve causal dependencies can effectively be used with the hash-based error detector to deliver all messages in causal order when implementing causal broadcast with probabilistic clocks. However, it tolerates a

maximum message load above which it requests all the time the causal dependencies of messages, thus delaying more and more the delivery of other messages, making it not usable for high message loads.

Message load	Out of causal order deliveries		Detected
msg/sec	PC	Retrieve dependencies	HD
10	0	0	0
25	0	0	0
50	211 ($4.2*10^{-3}$ %)	0	211
75	2856 ($3.8*10^{-2}$ %)	0	2856
100	13630 (0.14 %)	X	13630
125	31472 (0.25 %)	X	31472
150	68829 (0.45 %)	X	68829

Table 5.5: Out of causal order deliveries following the system's message load

5.4.6.2 Limits of the algorithm

This section analyzes, for several clocks, the maximum tolerated message load upon which processes are all the time recovering the causal dependencies of messages. Table 5.6 presents the maximum message load tolerated when retrieving the causal dependencies of messages tagged by the error detector in an implementation of causal broadcast using probabilistic clocks. Results show that the maximal message load increases with the size of the clock. In our implementation, a process jumps its scheduled broadcast if at the scheduled time it is in recovery of the causal dependencies of a message. This has an influence on the measured maximal tolerated message load.

Theoretically, the maximal tolerated message load should depend on the rate of positives the error detector returns, i.e., on the number of message recoveries processes need to perform. The rate of positives of the hash-based error detector decreases when increasing the clock size, which increases the maximal tolerated message load. Future experiments will compare the maximal tolerated message load with the positive rate of the error detector.

Clock size	Maximal message load (msg/s)
25	75
50	85
75	95
100	108
125	125
150	150

Table 5.6: Maximal tolerated message load following the system's clock size

To conclude, experiments showed that the mechanism to retrieve the causal dependencies of messages allows to deliver all messages in causal order. However, the mechanism only tolerates a limited message load, over which processes are retrieving all the time causal dependencies of messages, which in its turn delays more and more the delivery of messages.

5.4.7 Conclusion

This section proposed a new error detector to detect out of causal order deliveries.

First, we defined the required assumptions to implement an error detector that detects all out of causal order deliveries. We then showed that those assumptions are not realistic and that such an error detector can therefore not be implemented under realistic assumptions.

Second, we presented a hash-based error detector for M-entry clocks which experimentally detected all out of causal order deliveries of a causal broadcast implementation using probabilistic clocks. We keep the cost of the hash-based error detector small by using the clock difference between messages to only hash a subset of message ids.

Finally, we proposed an algorithm to retrieve the causal information of messages tagged as not causally ordered by error detectors, as well as a mechanism to remove this causal information once all processes have delivered them.

Experimental results and a theoretical analysis show that our error detector misses very few -experimentally none- out of causal order deliveries, that it adapts well to high message loads, and induces a low overhead. Moreover, using it with the algorithm to retrieve the causal dependencies of messages allowed to deliver all

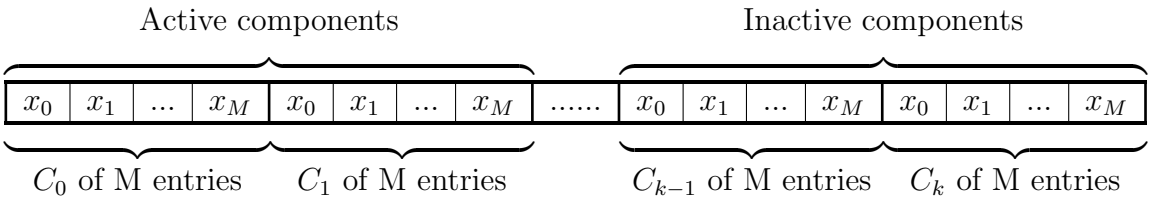


Figure 5.12: Representation of a *DCS* clock

messages in causal order in an implementation of causal broadcast using probabilistic clocks up to a given message load depending on the size of the system's probabilistic clock.

5.5 Dynamic Constant Size clocks

Mostéfaoui and Weiss [MW17b] showed that the efficiency of a probabilistic clock PC of size M to causally order a message m depends on the number of concurrent messages to m: the probability to causally order m with PC decreases when the number of concurrent messages to m inside the system increases. Moreover, the authors showed that increasing the size of PC also increases the probability that it causally orders m. The size of probabilistic clocks should, therefore, be chosen following the average number of concurrent messages inside the system. In other words, the size of probabilistic clocks should dynamically vary with the number of concurrent messages, increasing (resp. decreasing) whenever the number of concurrent messages is above (resp. below) a given value.

On the other hand, the size of Probabilistic clocks is fixed at initialization and cannot vary during execution. A wrong choice in the clock size either leads to an oversized clock or, if the size is too small, in many messages that are delivered out of causal order. Determining beforehand the number of concurrent messages in the system might be difficult and is often impossible.

This section presents a new logical clock, denoted Dynamic Clock Set (*DCS*), composed of a set of probabilistic clocks and whose size can dynamically vary during execution. We also give the operations to compare *DCS* clocks and change their size during execution. A Dynamic Clock Set (*DCS*) is composed of a set of Probabilistic clocks, denoted components, which all have the same number of entries (M). Figure 5.12 gives the representation of a *DCS* clock. The size of a *DCS* clock changes by varying the number of its components. *DCS* clocks *capture* causality but do not *characterize* it, meaning that for two messages m_1 (resp. m_2) of DCS clocks D_1 (resp. D_2), $m_1 \rightarrow m_2 \Rightarrow D_1 < D_2$, but $D_1 < D_2 \not\Rightarrow m_1 \rightarrow m_2$.

5.5.1 Definition of *DCS* clock components

A component C_k of a *DCS* clock is uniquely identified by its index k. C_k is either *active* or *inactive*. As shown in Figure 5.12, a *DCS* clock D is composed of one or several *active* components - with C_0 always being *active* - followed by none or several *inactive* components. In other words, C_0 is always *active*, followed by none or several *active* components C_i with $1 < i \leq |D|$, followed by none or several *inactive* components C_j with $i < j \leq |D|$.

A process increments one or several *active* components of its *DCS* clock to keep track of the causality of events, and only attaches the *active* components of its *DCS* clock on messages. The set $S_{incr,i}$ contains the set of indexes of the components that process p_i increments when executing an event, and before each event p_i increments the entries $f(p_i)$ of the components whose index is contained in $S_{incr,i}$.

5.5.2 Update of a DCS clock

Process p_i uses the following two rules $R1$ and $R2$ to update its local *DCS* clock:

- $R1$: Before executing an event, it updates its local clock D:
  $\forall x \in f(p_i), \forall k \in S_{incr}, D_i.C_k[x] = D_i.C_k[x] + d \ (d > 0)$
- $R2$: Each message m carries with it the vector clock D_m of its sender process at sending time. On the receipt of a message (m, D_m), process p_i:
  - Updates its local clock as follows:
    (1) If $|D_i| < |D_m|$, p_i calls $Add()$, defined below, till $|D_i| = |D_m|$.
    (2) $\forall k \in [1, |D_m|], \forall x \in [1, M], D_i.C_k[x] = max(D_i.C_k[x], D_m.C_k[x])$
  - Executes $R1$, Deliver(m)

5.5.3 Comparison of two *DCS* clocks

The components of *DCS* clocks are independent probabilistic clocks. Hence, the comparison operator "$<$" of *DCS* clocks is based on the comparison operator of probabilistic clocks. As a reminder, the comparison operator "$<$" of two probabilistic clocks C_1 and C_2 is defined as follows:

$$C_1 < C_2 \text{ iff } \forall x, C_1[x] \leq C_2[x] \wedge \exists k, C_1[k] < C_2[k]$$

The "$<$" operator of two *DCS* clocks D_1 and D_2 is defined as follows :

(1) $|D_1| \leq |D_2|$

(2) Each component C_k of D_1 is smaller or equal to the corresponding component C_k of D_2, and at least one component C_j of D_1 is strictly smaller than the component C_j of D_2: $\forall k \in [1, |D_1|], D_1.C_k \leq D_2.C_k \wedge \exists C_j, D_1.C_j < D_2.C_j$.

Two causally related messages m_1 and m_2 of respective DCS clocks $m_1.D$ and $m_2.D$ verify the following condition:

$$send(m_1) \rightarrow send(m_2) \Rightarrow m_1.D < m_2.D.$$

Note that two messages m_1 and m_2 whose DCS clock comparison does not satisfy the above two conditions are said to be concurrent, denoted as $m_1||m_2$. Formally:

$$m_1.D \not\leq m_2.D \wedge m_2.D \not\leq m_1.D \Rightarrow m_1||m_2$$

Theorem 5.4. *For any two messages m and m' of respective DCS clocks $m.D$ and $m'.D$, if $m \rightarrow m'$ then we have : $send(m) \rightarrow send(m') \Rightarrow m.D < m'.D$*

Proof. Consider that process p_i of DCS clock D_i sends a message m of causal dependencies Dep_m. We prove that $\forall m' \in Dep_m, m'.D < m.D$, by showing that when p_i sends m, we have $\forall m' \in Dep_m, m'.D < D_i$.

A process p_j updates its DCS clock D_j when delivering a message m: p_j adds components to D_j in order to ensure that D_j has at least as many components than $m.D$. Therefore, we have $|D_j| \geq |m.D|$. Second p_j updates D_j : $\forall x \in [1, M], \forall k \in [1, |D_j|], D_j.C_k[x] = max(D_j.C_k[x], m.D.C_k[x])$. Therefore, we have $\forall x \in [1, M], \forall k \in [1, |D_j|], D_j.C_k[x] \geq m.D.C_k[x]$. Hence, $m.D \leq D_j$ after p_j delivered m.

For all messages $m' \in Dep_m$, either p_i delivered m', or another process p_j delivered m' and broadcasted a message m" such that $m' \rightarrow m$" $\rightarrow m$ and p_i delivered m". If p_i delivered m', then $m'.D \leq D_i$ as showed above. Otherwise, (1) a process p_j has delivered m' and therefore $m'.D \leq m$".D (2) p_i has delivered m" and therefore m".$D \leq D_i$. Therefore, $m'.D \leq D_i$. Hence, we have $\forall m' \in Dep_m, m'.D \leq D_i$.

When p_i sends m, it first updates its DCS clock by incrementing at least one entry x of at least one component C_k before appending D_i on m. Thus, $\forall m' \in Dep_m, \exists x \in [1, M], \exists k \in [1, |D_j|], m.D.C_k[x] > m'.D.C_k[x]$.

Therefore, $\forall m' \in Dep_m, m'.D < m.D$.

□

5.5.3.1 Operations to modify the size of *DCS* clocks

The size of a *DCS* clock can be dynamically adjusted during execution by adding and removing components to it. In particular, its size can be adapted to the number of concurrent messages in the system, since the efficiency of probabilistic clocks to causally order messages depend on that parameter (see Section 5.2). A *DCS* clock should increase its size when observing an increase in the number of concurrent messages and should decrease it in the opposite case. The size of *DCS* clocks can be chosen following the desired accuracy of causal message ordering.

Process p_i modifies its local *DCS* clock D_i through the following operations:

- **Activate():** Activates the component of D_i with the lowest index among D_i's inactive components.
- **Deactivate():** Deactivates the component of D_i with the highest index among D_i's active components.
- **Add():** Creates a new component, sets its entries to 0, and adds the component at the end D_i.
- **Remove():** Removes the component of D_i with the highest index.

Activate() Process p_i calls the operation *Activate()* to activate the inactive component of D_i with the lowest index, provided that D_i has at least one inactive component. The call to *Activate()* immediately returns *false* if D_i has no inactive component. Otherwise, the inactive component of D_i with the lowest index is activated.

Deactivate() Process p_i calls the operation *Deactivate()* to deactivate the active component of D_i with the highest index. A *DCS* clock has at least one active component. Thus, the call to *Deactivate()* immediately returns *false* if D_i has only one active component. Otherwise, the active component of D_i with the highest index is deactivated.

To illustrate the *Activate()* and *Deactivate()* operations, let's consider a process p_i whose *DCS* clock D_i has four components: $D_i = \{C_0, C_1, C_2, C_3\}$. If p_i wants to deactivate components, then it will deactive them in decreasing order, i.e, first C_3, then C_2, and finally C_1. On the other hand, if p_i wants to re-activate them, then it will first activate C_1, then C_2, and finally C_3.

A process keeps each deactivated component C_d locally, because it might receive a message whose *DCS* clock contains C_d, and it will then require the local C_d to ensure that the delivery conditions of the message's component C_d are satisfied.

Add() A process decides locally to add a new component to its *DCS* clock, i.e., without communicating with other processes. When p_i calls the *Add()* operation, it first creates a new component C_k in active state, sets its entries to 0, and appends C_k to the end of D_i. Therefore, C_k will be the component of D_i with the highest index.

After adding a component to D_i, p_i also activates all components of D_i, since the inactive components of a *DCS* clock have a strictly higher index than the active components. Therefore, adding a component at the end of D_i implicates that all components of D_i must be active.

Remove() Process p_i calls *Remove()* to remove the component of D_i with the highest index. It returns *false* if D_i has only one component, since by definition a *DCS* clock has at least one component.

5.6 Causal broadcast algorithm using DCS clocks

This section presents an implementation of causal broadcast using *DCS* clocks. We first present the causal broadcast algorithm, then we describe the implementation of the operations to modify the size of *DCS* clocks.

5.6.1 Model

The system contains a set $\Pi = \{p_1, p_2, \ldots, p_N\}$ of N processes. Processes are reliable and communicate through message passing. Each pair of processes is connected by a reliable communication channel. Local events induce no interactions with other processes and are therefore omitted. Each application message is broadcasted to all processes of the system.

5.6.2 Definition of the algorithm

Algorithm 14 describes the *DCS* clock-based causal broadcast algorithm. Each process p_i keeps:

- D_i: its local *DCS* clock.
- $S_{incr,i}$: a set containing the indexes of the components of D_i that p_i increments when broadcasting a message.

Broadcast of a message m Process p_i first updates its *DCS* clock by executing the Rule $R1$ given in Section 5.5.2 with $d = 1$:

$$\forall x \in f(p_i), \forall k \in S_{incr,i}, D_i.C_k[x] = D_i.C_k[x] + 1.$$

Then, p_i broadcasts m with $S_{incr,i}$ and the *active* components of D_i.

Reception of a message m Process p_i calls the function *prepareComparison()* upon receiving a message m of *DCS* clock D_m and of set of incremented component indexes $S_{incr,m}$. *prepareComparison()* prepares D_i to the comparison with D_m in three steps:

- First, it calls *Add()* till $|D_i| = |D_m|$, since D_i must satisfy the delivery conditions of each component of D_m. Note that D_i might have more components than D_m. It is then sufficient to ensure that D_i satisfies the delivery conditions of the components of D_m.
- Second, it activates the inactive components C_d of D_i for which $\exists x, D_i.C_d[x] < D_m.C_d[x]$, since those components contain new causal information. The components $D_{k<d}$ are then also activated to maintain the *DCS* clock property that active components always have lower indexes than inactive ones.
- Third, if it has activated components, then it sets $S_{incr,i}$ to a new set of randomly chosen indexes of active components, to ensure that active components are on average incremented by the same number of processes.

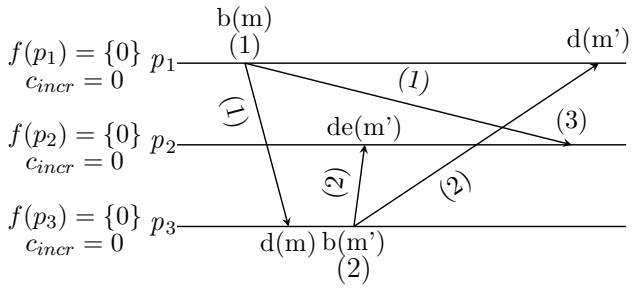
After calling *prepareComparison()*, p_i waits till D_i satisfies the following delivery conditions of D_m:

- For each component $C_{k \notin S_{incr,m}}$: p_j did not increment C_k when broadcasting m. Hence, the entries of $D_i.C_k$ should be equal or greater than those of $D_m.C_k$:

$$\text{waitUntil}(\ \forall C_{k \notin S_{incr,m}}, D_m.C_k[x] \leq D_i.C_k[x])$$

- For each component $C_{k \in S_{incr,m}}$: p_j incremented the entries $f(p_j)$ of C_k when broadcasting m. Hence, the entries $x \in f(p_j)$ of $D_i.C_k$ should be equal or greater than those of $D_m.C_k$ minus one, and the entries $x \notin f(p_j)$ of $D_i.C_k$ should be equal or greater than those of $D_m.C_k$:

$$\text{waitUntil}(\forall C_{k \in S_{incr,m}}, \forall x \in f(p_j), D_m.C_k[x] - 1 \leq D_i.C_k[x])$$
$$\text{waitUntil}(\forall C_{k \in S_{incr,m}}, \forall x \notin f(p_j), D_m.C_k[x] \leq D_i.C_k[x])$$



(1): [{[1],[0]},0] (2): [{[1],[1]},1] (3): d(m), d(m')
b(m): broadcast(m) **d(m)**: deliver(m) **de(m)**:cache(m)

Figure 5.13: Causal broadcast using DCS clocks

Process p_i executes Rule $R1$ given in Section 5.5.2 with $d = 1$ once D_i satisfies the delivery conditions of D_m:

$$\forall x \in f(p_i), \forall k \in S_{incr}, D_i.C_k[x] = D_i.C_k[x] + 1$$

Finally, p_i delivers m (line 7).

Algorithm 14: Broadcast at process p_i

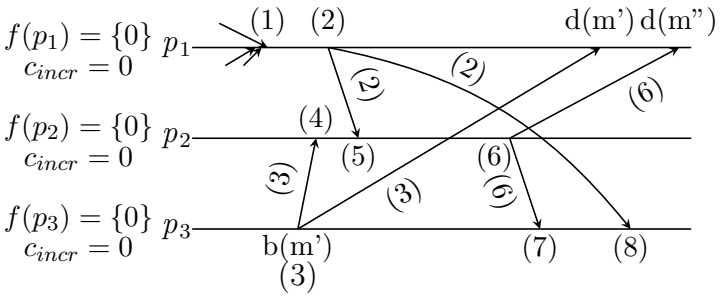
Broadcast of message m

1: $\forall x \in f(p_i), \forall k \in S_{incr,i}, D_i.C_k[x] = D_i.C_k[x] + 1$
2: broadcast(m,D_i,$S_{incr,i}$)

Upon reception of message $(m, D_m, S_{incr,m})$ **from** p_j

3: prepareComparison()
4: waitUntil($\forall C_{k \notin S_{incr,m}}, D_m.C_k[x] \leq D_i.C_k[x]$)
5: waitUntil($\forall C_{k \in S_{incr,m}}, \forall x \in f(p_j), D_m.C_k[x] - 1 \leq D_i.C_k[x] \wedge \forall x \notin f(p_j), D_m.C_k[x] \leq D_i.C_k[x]$)
6: $\forall x \in f(p_j), \forall k \in S_{incr,m}, D_i.C_k[x] = D_i.C_k[x] + 1$
7: deliver(m)

Figure 5.13 shows the broadcast of two messages. The system is composed of three processes p_1, p_2, and p_3. Each process maintains two components with each component having one entry. Processes p_1 and p_2 have $S_{incr} = \{0\}$, i.e., they increment component C_0 when broadcasting a message while p_3 increments component C_1 when broadcasting a message. In the scenario, p_1 first broadcasts m whose causal information is represented by (1). Upon reception of m, p_3 delivers it since its delivery conditions are satisfied. Then, p_3 broadcasts m', i.e., $m \rightarrow m'$. The causal information of m' is represented in (2). Process p_2 receives m' before m. Thus, it postpones the delivery of m since its delivery conditions are not satisfied, because $p_2.D_2.C_0[0] < m'.D_{m'}.C_0[0]$. When it eventually receives m at (3), it delivers m and then m'.



(1): expand() (2): [{[0],[1]},1] (3): [{[1],[0]},1]
(4): d(m') (5): expand(), d(m), D_2=[{[1],[1]},0]
(6): b(m"), m"=[{[2],[1]},0] (7): delay(m")
(8): d(m),d(m"),D_3=[{[2],[1]},1]}

Figure 5.14: Example of *DCS* clocks expansion

5.6.2.1 Expansion of the local DCS clock

The algorithm expands *DCS* clocks in order to reduce the number of processes that increment the same entries when broadcasting a message. Process p_i decides to increment its local *DCS* clock D_i without communicating with other processes. p_i decides to expand D_i spontaneously, when observing for example a high message load. To expand its *DCS* clock, p_i:

- First, calls *Activate()* described in Section 5.5.3.1, which activates an inactive component of D_i if one is available, and which returns *false* otherwise.
- Second, calls *Add()* described in Section 5.5.3.1 if *Activate()* returns false, i.e., p_i adds a new component to D_i if D_i has no inactive component.
- Third, p_i sets $S_{incr,i}$ to a new set of randomly chosen indexes of active components, to ensure that active components are on average incremented by the same number of processes.

Figure 5.14 shows a scenario where processes expand their *DCS* clock. The system is composed of three processes. Initially, each process maintains a *DCS* clock of one component, namely C_0, and each process increments C_0 when broadcasting a message. The notation of causal information is as follows: [*DCS* clock, $S_{incr,}$], i.e., [{[0],[1]},1] means the *DCS* clock {[0],[1]} with $S_{incr,}$={1}.

At (1), p_1 decides to expand its *DCS* clock D_1 (e.g., detection of high message load) by adding a new component to D_1, since D_1 has no deactivated component. Moreover, it re-assigns itself to a random component of D_1 which gives $S_{incr,1}$ =

$\{random()\%2\} = \{1\}$. At (2) and (3) p_1 and p_3 broadcast a message m of *DCS* clock $\{\{[0],[1]\}\}$ and m' of *DCS* clock $\{\{[1]\}\}$ respectively. Process p_2 first receives m' at (4) and delivers it. At (5), p_2 receives m, adds a component to its *DCS* clock, since $D_2 < |m.D|$, and sets $S_{incr,2} = \{random()\%2\} = \{0\}$, then it delivers m.

At (6), p_2 broadcasts message m" with *DCS* clock [{[2],[1]},0]. p_3 receives m" at (7), expands its *DCS* clock by adding a new component since $|D_3| < |m".D|$ and sets $S_{incr,3} = \{random()\%2\} = \{1\}$. It postpones the delivery of m" since $m".D.C_0[0] = 1$ and $D_3.C_0[0] = 0$. At (8), p_3 receives m, and delivers both m then m'.

5.6.2.2 Deactivate DCS clock components

This section describes an implementation to deactivate *DCS* clock components without loss of causal information. A process should only deactivate the components of its *DCS* clock that do not contain causal information that is still useful to some other processes.

Processes should deactivate components whenever possible, as for example when the message load decreases, because deactivated components are not sent with messages and are only kept locally. Consequently, deactivating components reduces the causal information carried by messages.

Process p_i deactivates the component of D_i with the highest index among the active ones, i.e., if k components of D_i are active, then p_i first deactivates C_{k-1}, then C_{k-2}, etc. up to C_1. C_0 cannot be deactivated.

Component C_k of D_i provides causal information to at least one other process as long as C_k's delivery conditions are not satisfied by all processes, i.e., as long as $\exists p_j, \exists x, p_j.D_j.C_k[x] < D_i.C_k[x]$.

We add two additional conditions to ensure that p_i will not activate C_k again shortly after deactivating it:

(1) No process currently increments C_k.

(2) No process currently delays the delivery of a message m with $k \in m.S_{incr}$, because the delivery of m might violate the above condition.

5.6.2.2.a Deactivation round

Process p_i verifies the satisfaction of the above conditions through a two phase exchange of messages with the other processes. In *Phase 1*, p_i sends the component C_k to deactivate to all processes, which reply with a positive or negative acknowledgement. In *Phase 2*, p_i confirms or not the deactivation of C_k to all processes.

Phase 1. Process p_i starts *Phase 1* by broadcasting a *Deactivate* message containing C_k and C_k's index k. The other processes reply with an *AckDeactivate* message containing a positive or negative acknowledgement of C_k, depending on whether they locally satisfy the deactivation conditions of C_k or not. Moreover, they freeze the dynamics operations of their *DCS* clock till the end of the *Deactivation round*, i.e., till they receive the *DecisionDeactivate* message from p_i.

Phase 2. p_i broadcasts a *DecisionDeactivate* message once it received the *AckDeactivate* message from all processes. The *DecisionDeactivate* message contains C_k's index k and a boolean that confirms or not the success of the round, i.e., the deactivation or not of C_k. p_i sets this boolean to *true* if all processes positively acknowledged the deactivation of C_k and *false* otherwise. Upon reception of the *DecisionDeactivate* message, a process unfreezes the dynamics operations of its *DCS* clock, and deactivates C_k provided that the boolean is set to *true*.

Any process can start a *Deactivation round*. The process can be chosen probabilistically or, for example, in a predefined way based on the process identifier. Several processes could start a *Deactivation round* for the same component or for different components simultaneously, but this should be avoided since acknowledging the deactivation of the same components several times is useless.

5.6.2.2.b Complexity analysis of Deactivation rounds

A *Deactivation round* has a message complexity in $\mathcal{O}(N)$: N *Deactivate* messages, N *AckDeactivate* messages and N *DecisionDeactivate* messages. A *Deactivation round* has a message memory complexity in $\mathcal{O}(M)$: *Deactivate* messages contain one integer and a component of M integers, while *AckDeactivate* and *DecisionDeactivate* messages contain some integer and boolean values and have therefore a space complexity in $\mathcal{O}(1)$. We assume bounded integers encoded on 32 bits.

5.6.2.3 Removal of *DCS* clock components

This section describes an implementation to remove components from *DCS* clocks. Processes keep inactive components of their *DCS* clock locally, and should eventually remove them in order to free up memory space.

Processes must coordinate the removal of inactive components with each other, because if one process removes a component C_k from its *DCS* clock, then all processes should remove C_k from their *DCS* clock. In fact, assume that one process p_i removes C_k from its *DCS* clock, and that another process p_j then broadcasts a message m without having removed C_k from its *DCS* clock. The *DCS* clock appended on m will then contain C_k. Upon reception of m, p_i will have lost the causal information of C_k since it deleted C_k. Thus, a process should only delete a component C_k once it is ensured that it will receive no other message containing this C_k. Before removing C_k, p_i should therefore verify that :

(1): C_k is inactive at all processes of the system.

(2): It will receive no message containing C_k after removing C_k from its *DCS* clock.

5.6.2.3.a Remove round

Process p_i verifies the satisfaction of the above conditions through a two phase exchange of messages with the other processes. *Phase 1* synchronizes processes to ensure that they all have delivered the messages containing the component C_k to be removed and that they will broadcast no new message containing C_k. *Phase 2* propagates the deletion decision of C_k, which depends on the satisfaction by all processes of the two above conditions.

Phase 1. Process p_i broadcasts a *Remove* message containing a set with a tuple $< p_k, seq_k >$ for each process p_k that broadcasted a message since the last *Remove* round, with seq_k corresponding to the number of messages p_k broadcasted since the last *Remove* round.

Upon reception of the *Remove* message, process p_j freezes the dynamics operations of its *DCS* clock. Moreover, it verifies that C_k is locally inactive and that for each tuple $< p_k, seq_k >$ it delivered seq_k messages from p_k since the last *Remove* round. Finally, p_j replies with an *AckRemove* message containing k, the index of C_k, as well as a boolean set to *true* if both conditions are satisfied, and *false* otherwise.

Phase 2. p_i sends a *DecisionRemove* message after it received the *AckRemove* message from all processes. The *DecisionRemove* message contains C_k's index k

and a boolean that confirms or not the success of the round, i.e., the removal or not of C_k. p_i sets this boolean to *true* if all processes positively acknowledged the removal of C_k and *false* otherwise. Upon reception of the *DecisionRemove* message, a process unfreezes the dynamics operations of its *DCS* clock, and removes C_k provided that the boolean is set to *true*.

5.6.2.3.b Complexity analysis

A *Remove round* has a message complexity in $\mathcal{O}(N)$: N *Remove* messages, N *AckRemove* messages and N *DecisionRemove* messages. A *Remove round* has a message memory complexity in $\mathcal{O}(N)$: *Remove* messages contain a vector with up to N integers, while *RepRemove* and *Remove* messages contain some integer and boolean values. We assume bounded integers encoded on 32 bits.

Remove rounds should not be executed often because of the memory complexity which is in $\mathcal{O}(N)$ messages. Nevertheless, several components can be acknowledged at once. Moreover, a *DCS* clock is usually composed of a few components, and keeping them locally without sending them with messages only represents a small local memory overhead.

5.6.2.4 Termination proof of *DCS* clocks

This section gives the proof of termination of the causal broadcast algorithm using *DCS* clocks. The proof is divided in two parts. *Theorem 5.5* proves the termination property for static *DCS* clocks. *Theorem 5.6* proves that the termination property holds when adding and removing components to *DCS* clocks.

Theorem 5.5. *A well-formed message broadcasted with an algorithm using a static DCS clock to causally order messages is eventually delivered by all processes.*

Proof. We prove it by induction. We assume that each process p_i has a *DCS* clock D_i of $l \geq 1$ components.

H_0: Messages generated on the initial state are eventually delivered by all processes.

Processes initialize the entries of components to 0. Hence, a message m generated by p_i in the initial state carries a *DCS* clock with $\forall x \in f(p_i), \forall k \in S_{incr,i}, m.C_k[x] = 1$ and for all the other component entries values equal to 0. Since all processes initialize the entries of components to 0, their *DCS* clock satisfies both delivery conditions upon reception of m. Thus, messages generated in the initial state are eventually delivered by all processes.

H_1: We assume a set of messages M that are eventually delivered by all processes. We show that any message generated by a process after it delivered the messages of M will be eventually delivered by all processes.

Let's consider a message m of *DCS* clock D_m generated by a process p_i after it has delivered all messages of M.

By hypothesis, all processes eventually deliver the messages of M and increment their *DCS* clock accordingly. Moreover, p_i only increments the entries $x \in f(p_i)$ of the components $C_k, k \in S_{incr,i}$ when broadcasting m. Hence, the entries of the *DCS* clock appended on m and the *DCS* clock of processes after they delivered m only differs by one $\forall x \in f(p_i), \forall k \in S_{incr,i}$. Therefore, processes will satisfy the delivery conditions of m once they delivered the messages of M, which they do by definition. Therefore, processes eventually deliver m.

Any message generated after a set of eventually delivered messages will eventually be delivered by all processes (*see H1*). Since all messages generated on the initial state are eventually delivered by all processes (*see H0*), we conclude that any message is eventually delivered by all processes.

□

Lemma 5.2. *The termination property of the causal broadcast algorithm using DCS clocks holds when processes add or activate components of their DCS clock.*

Proof. Adding a new component to a *DCS* clock is equivalent to activate an inactive component not yet contained in the *DCS* clock of any other process. Therefore, it is sufficient to show that the termination property holds when a process p_i activates an inactive component C_k of its *DCS* clock D_i. We consider that p_i broadcasts a message m after activating C_k.

p_i reaffects itself to new components when activating C_k, and stores the index of those components in $S_{incr,i}$. Any process p_j that receives m first adds and activates C_k to its *DCS* clock D_j if D_j has no such component yet. Moreover, p_j knows which components p_i incremented when broadcasting m, since m carries $S_{incr,i}$. Therefore, p_j adds and activates C_k to its *DCS* clock, and by using S_{incr} it will also increment the entries of the right components when delivering m. □

Lemma 5.3. *The termination property of the causal broadcast algorithm using DCS clocks holds when processes remove components of their DCS clock.*

Proof. The *Remove round* ensures that processes only remove a component C_k provided that all processes have delivered all messages whose *DCS* clock contains C_k. Moreover, the *Remove round* ensures that no new message containing C_k will

be broadcasted (even though a process might add a new component of index k after C_k was removed). Therefore, after a successful *Remove round*, C_k will not be used in any delivery of message, and its deletion will therefore impact no message delivery. □

Lemma 5.4. *The termination property of the causal broadcast algorithm using DCS clocks holds when processes deactivate components of their DCS clock.*

Proof. Consider that process p_i deactivates component C_k. p_i does not increment deactivated components, i.e., C_k. Hence, C_k will contain no new causal information. Moreover, deactivated components are not sent with messages, but are kept locally by processes. Therefore, deactivating components only removes delivery conditions of a message without loosing causal information. The deactivation of components does therefore impact no message delivery. □

Theorem 5.6. *A well-formed message is eventually delivered by all processes when broadcasting messages with the causal broadcast algorithm using DCS clocks.*

Proof. Following *Theorem 5.5*, processes eventually deliver messages when using a causal broadcast algorithm using static *DCS* clocks to causally order messages. Following *Lemma 5.2*, *Lemma 5.3* and *Lemma 5.4*, the termination property holds when the dynamics operations of *DCS* clocks are considered (*Add(),Activate(), Remove(), Deactivate()*). The termination property of *DCS* clocks holds therefore also when considering dynamic *DCS* clocks.

□

5.7 Experimental results

Experiments were carried out on the OMNeT++ simulator. Processes generate messages on a regular interval plus a deviation computed according to a normal distribution $N(10, 0)$. The propagation delays of messages follows a normal distribution $N(100, 20)$. An independent controller module detects out of causal order deliveries. The number of concurrent messages in the system depends on the system's message load and the distribution of propagation delays of messages. In the following we keep a propagation delay of messages that follows a normal distribution $N(100, 20)$, and vary the message load instead of the number of concurrent messages, since the former is easier to set up.

The first experiments aim to determine the required *DCS* clock size to achieve a given accuracy of causal message ordering, depending on the system's message

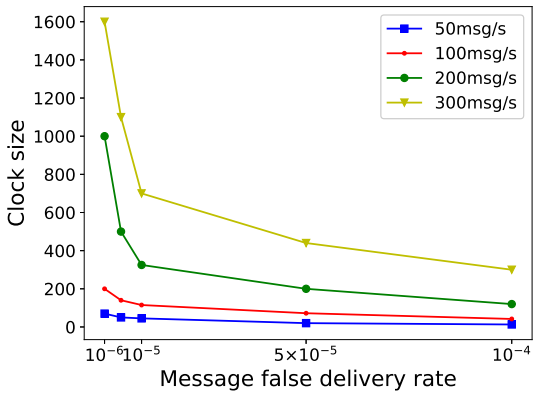


Figure 5.15: Clock size following the message load to achieve a given causal ordering accuracy

load. The second set of experiments compares *DCS* clocks to probabilistic clocks for two different message load patterns. The third experiment evaluates the load balancing ability of *DCS* clocks.

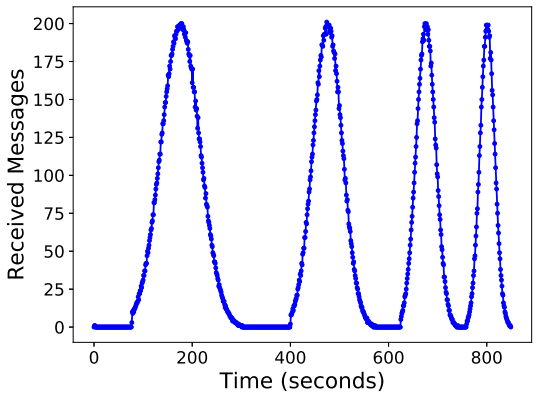
5.7.1 Clock size following the message load

The size of *DCS* clocks should be set following the message load and the accepted probability that a message is delivered out of causal order. Hence, the first experiment aims to determine the required *DCS* clock size following the message load and the accepted probability that a message is delivered out of causal order.

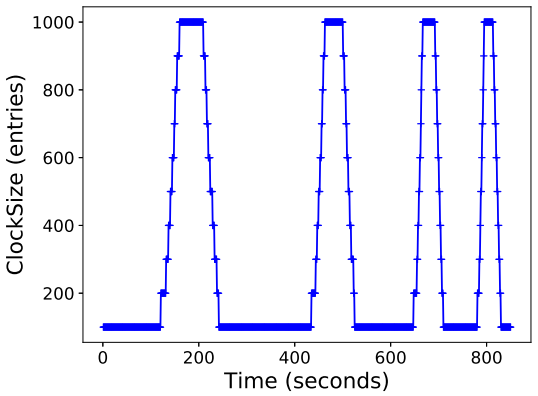
The system contains 2000 processes broadcasting messages at a frequency determined by the system's message load. The hash function returns two entries, i.e., processes increment two entries when broadcasting a message. Figure 5.15 presents the required probabilistic clock size, depending on the message load, to have $10^{-2}\%, 5.10^{-3}\%, 10^{-3}\%, 5.10^{-4}\%$ and $10^{-4}\%$ out of causal order deliveries.

Results show that the required size of the clock increases with the causal ordering accuracy, i.e., to have a probability to deliver a message out of causal order of $10^{-3}\%$ requires a bigger clock than a probability of only $10^{-2}\%$.

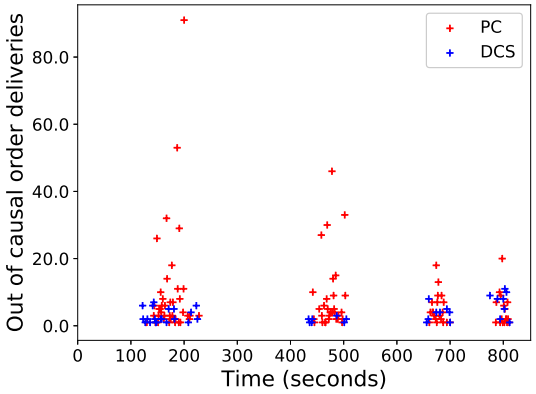
The clock size required to causally order messages increases faster than linearly with the message load. We explain it by analyzing the formula presented in [MW17b] and described in Section 5.2 that gives the probability that a message is delivered out of causal order: $(1-(1-\frac{1}{M})^{X*k})^k$, where X corresponds to the number of concurrent messages, which is directly affected by message load, and M corresponds to the clock size. The formula confirms that an increase in message load (resp., clock size) has an exponential (resp., division) impact in the formula result, thus



(a) Message load pattern



(b) Size of clocks attached to messages



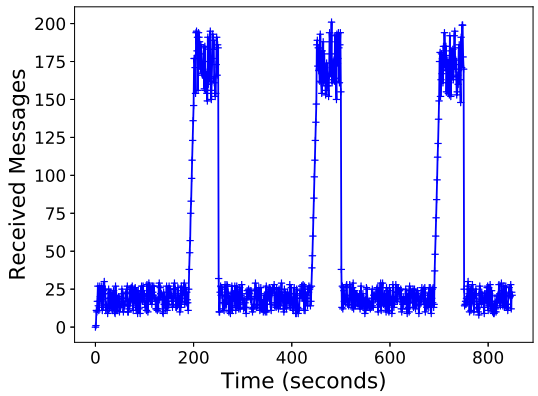
(c) Out of causal order deliveries

Figure 5.16: Size of *DCS* clocks and out of causal order deliveries of the first message load pattern

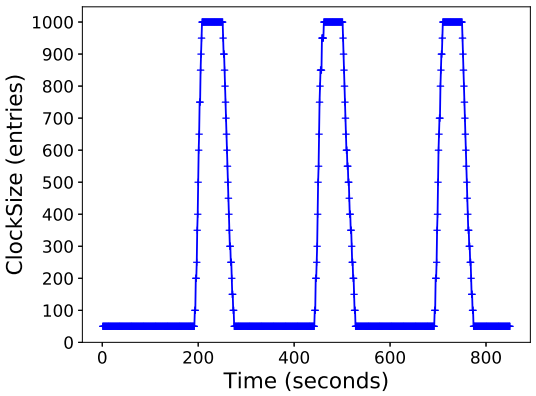
explaining why the clock size increases faster than linearly. This observation accentuates the importance of adjusting dynamically the clock size to the number of concurrent messages inside the system, instead of choosing a size following the highest expected number of concurrent messages, since the clock risks being much bigger than required.

5.7.2 Behavior following different message load patterns

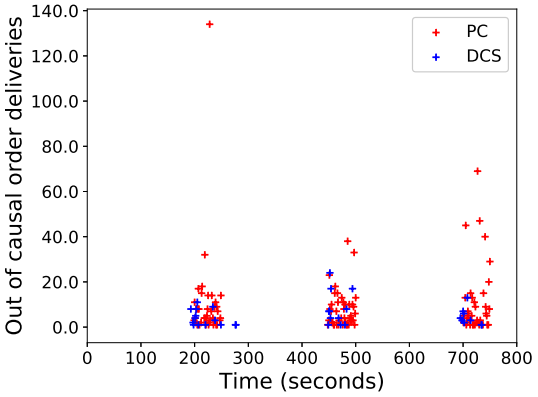
The second set of experiments compares *DCS* clocks and probabilistic clocks for two different message load patterns. The first pattern consists of intervals in which the message load goes from 10 to 200 messages broadcasted per second following a normal distribution. The second pattern consists of random messages loads either in an interval between 10 and 30 messages broadcasted per second or an interval between 150 and 200 messages broadcasted per second. Figures 5.16 and Figure 5.17 show the message loads of the experiment using each pattern. The



(a) Message load pattern



(b) Size of clocks attached to messages



(c) Out of causal order deliveries

Figure 5.17: Size of *DCS* clocks and out of causal order deliveries of the second message load pattern

system contains 2000 processes which increment two entries when broadcasting a message.

5.7.2.1 Bell message load pattern

Figure 5.16 shows the results of the experiments conducted with the first message load pattern, represented in Figure 5.16a. It consists of 4 bells obtained following a normal distribution of respective variance $\sigma^2 = 40, \sigma^2 = 30, \sigma^2 = 20$ and $\sigma^2 = 15$. Figure 5.16b shows the size of the *DCS* clock attached to messages. The *DCS* clocks in this experiment have an average size of 304 entries. Hence, we conducted an experiment with the same message load using a probabilistic clock of 304 entries. Figure 5.16c gives the number of out of causal order deliveries that occurred when using *DCS* clock and probabilistic clocks.

We first observe in Figure 5.16b that *DCS* clocks rapidly adapt themselves to the message load pattern. Second, Figure 5.16c shows that the causal broadcast

algorithm using *DCS* clocks delivers fever messages out of causal order than the causal broadcast algorithm using probabilistic clocks. In total, with the algorithm using *DCS* clocks, we observed 154 out of causal order deliveries, while we observed 854 out of causal order deliveries with the algorithm using probabilistic clocks. Hence, with the algorithm using probabilistic clocks we observe ≈ 5.5 more out of causal order deliveries. The out of causal order deliveries are for both algorithms concentrated around the message load peaks.

Following the theoretical analysis, we would have expected that the algorithm using *DCS* clocks delivers fewer messages out of causal order. By further analyzing we found that the out of causal order deliveries are coming from the *DCS* clock whose size is not increasing fast enough with the message load. The current metric a process uses to determine when to increase its *DCS* clock is the number of messages it received in the last second. Hence, a process might not adapt fast enough when the message load increases abruptly.

To conclude, the results of the first experiment shows that *DCS* clocks have better performances in causally ordering messages than probabilistic clocks in systems with a bell message load pattern as presented in Figure 5.16. Moreover, *DCS* clocks require a further analysis to determine which metric/mechanism to use to determine when to increase the size of the clock.

5.7.2.2 Random message load pattern

Figure 5.16 shows the results of the experiments conducted with the second message load pattern, represented in Figure 5.17a. It is divided in intervals of low message loads chosen every second and going from 10 to 30 messages broadcasted per second, followed by high message loads going from 150 to 200 messages broadcasted per second. Figure 5.16b shows the size of the *DCS* clock attached to messages. The *DCS* clocks in this experiment have an average size of 255 entries. Hence, we conducted an experiment with the same message load using a probabilistic clock of 255 entries. Figure 5.16c gives the number of out of causal order deliveries that occurred when using *DCS* clock and probabilistic clocks.

Results confirm that the size of *DCS* clocks is rapidly adapted to the message load: they remain small most of the time and grow fast during message load peaks. Second, Figure 5.16c shows that the causal broadcast algorithm using *DCS* clocks delivers fever messages out of causal order than the causal broadcast algorithm using probabilistic clocks. In total, the algorithm using *DCS* clocks delivered 187 messages out of causal order, while the one using probabilistic clocks delivered 1126 messages out of causal order ($\approx$ 6x). The out of causal order deliveries are for both algorithms again concentrated around the message load peaks.

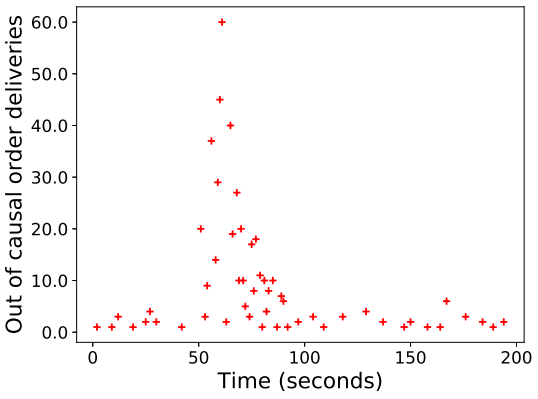


Figure 5.18: Out of causal order deliveries with load balancing

To conclude, the results of the second experiment shows that *DCS* clocks have better performances in causally ordering messages than probabilistic clocks in systems with a message load pattern consisting of low message loads followed by peaks, as presented in Figure 5.17.

5.7.3 Load balancing

Clocks with M entries are the most accurate when their entries are incremented uniformly. Processes should therefore be affected to clock entries such that clock entries are incremented uniformly. However, a process might vary the number of messages it broadcasts during execution, thus rendering an initially uniformly incremented clock non uniformly incremented.

Algorithms using *DCS* clocks can dynamically change the clock entries associated to processes. The last experiment measures the ability of *DCS* clocks to affect processes to new clock entries in order to increment *DCS* clocks more uniformly.

The system consists of 1000 processes, which increment two entries when broadcasting a message (i.e., $|f| = 2$). The system's message load is of 100 messages broadcasted per second, and processes maintain a *DCS* clock of 200 entries. Till t=50s processes increment the *DCS* clock entries uniformly. At t=50s, we modify the clock entries associated to processes in order to have 75% of message broadcasts that increment component C_0, i.e., component C_0 is incremented 3 times more after t=50s. Figure 5.18 shows the number of messages that are delivered out of causal order.

We observe that the number of messages delivered out of causal order increases a lot after t=50s. At t=70s, processes that increment component C_0 detect that component C_0 is much more incremented than other components. In order to balance,

they assign themselves to other components with a probability of 50%. Consequently, the number of messages delivered out of causal order drops. Eventually, the remaining processes that increment component C_0 detect that C_0 is still more incremented than other components, and reassign themselves therefore to other components with a probability of 50% till the *DCS* clock is again incremented uniformly, which happens at t=90s. Therefore, processes eventually incremented their *DCS* clock again uniformly.

5.7.4 Summary

This section presented a causal broadcast implementation using Dynamic Clock Sets (*DCS*), a new logical type of clocks based on Probabilistic clocks. The main feature of *DCS* clocks over constant size ones is that their size can be dynamically adjusted during execution. This is particularly important since the optimal size of *DCS* and constant size clocks usually depend on the number of concurrent messages inside the system, which can drastically vary and whose knowledge beforehand is difficult or even impossible to determine.

Experimental results confirm that *DCS* clocks have a higher accuracy of causal message ordering when compared to probabilistic clocks. Moreover, depending on the system's message load pattern, *DCS* clocks also require less memory than probabilistic clocks. Finally, *DCS* clocks can efficiently be used to increment clock entries more uniformly.

5.8 Conclusion

This chapter presented the work of this thesis done on M-entry clocks. Such clocks scale well with the number of processes N, since their size M is independent and usually much smaller than N. However, algorithms using M-entry clocks might deliver some messages out of causal order. Therefore, the first contribution of this chapter aims to detect messages that are delivered out of causal order when causally ordering messages by using M-entry clocks. The second contribution aims to avoid the out of causal order delivery of messages tagged as having not delivered causal dependencies. The third contribution of this chapter defines a new clock, based on probabilistic clocks, and whose size is dynamically adjustable during execution.

First, we defined the assumptions required to implement an error detector for M-entry clocks that detects all messages that are delivered out of causal order. Moreover, we showed that these assumptions are not realistic and that such an error detector can therefore not be implemented under realistic assumptions. We

then proposed an error detector for probabilistic clocks based on hashes. The proposed error detector detects most of the messages that are delivered out of causal order. We gave some mechanisms to increase its efficiency and decrease its costs. We implemented our error detector on the OMNeT++ simulator, tested its efficiency, and compared it to another error detector for M-entry clocks.

Second, we proposed an algorithm to retrieve the causal dependencies of messages, in order to ensure the causal delivery of messages tagged as not causally ordered by error detectors. We tested our algorithm on an implementation done on the OMNeT++ simulator.

Third, we proposed the Dynamic Clock Sets (*DCS* clocks), a new logical clock based on probabilistic clock. The main feature of *DCS* clocks over M-entry clocks is that their size can be dynamically adjusted during execution. This is particularly important since the optimal size of *DCS* and M-entry clocks depends on the number of concurrent messages in the system, which can drastically vary and whose knowledge beforehand is difficult or even impossible to determine. We provided an implementation of causal broadcast using *DCS* clocks and analyzed its behavior through experiments done on the OMNeT++ simulator.

Chapter 6

Conclusion

Contents

Causal broadcast has been extensively investigated, and innumerable applications use it. Causal broadcast algorithms either piggyback information on messages in order to causally order them at reception or they organize the system and make assumptions on the network topology to ensure that messages are implicitly ordered at reception, thus ensuring that they can be causally delivered upon reception without any control.

This thesis aimed to provide causal broadcast for large dynamic distributed systems. Hence, the algorithms, presented in two parts, scale and tolerate processes that join and leave the system during execution.

6.1 A causal broadcast algorithms that tolerates the dynamics of Mobile Networks

We proposed a causal broadcast algorithm tailored to the features and dynamics of mobile networks. These features are mobile host mobility, dynamic host membership, unreliable dynamic wireless channels, memory and computing constraints of mobile hosts, as well as scalability issues due to the high number of mobile hosts and stations. The algorithm tolerates the failure of mobile hosts. We also proposed an extension to the algorithm that tolerates the failure of stations.

Messages piggyback few causal information. The algorithm scales well with both hosts and stations. Hosts have a low memory footprint while stations have a memory footprint that grows linearly with the number of locally connected hosts. Contrarily to existing centralized approaches, stations discard obsolete messages using only local information, thus requiring no message exchange.

Experimental results showed that both algorithms use much less causal information and messages than existing algorithms for mobile networks, while making fewer assumptions on the network and devices.

6.2 Causally order messages using M-entry clocks

We showed that an error detector that detects all out of causal order deliveries cannot be implemented under realistic conditions. Moreover, we proposed an error detector based on hashes. Basically, it consists of hashing the causal dependencies of messages, and use this hash to retrieve the causal dependencies of messages at destination. Experimental results confirmed that the error detector has a high accuracy in detecting out of causal order deliveries. We also gave optimizations to reduce the costs of the hash-based error detector and measured their efficiency experimentally.

Second, we proposed a short algorithm to retrieve the causal dependencies of messages and showed its limitations. Experiments confirmed that using this algorithm in conjunction with a causal broadcast implemented with probabilistic clocks heavily reduces the number of messages that are delivered out of causal order.

Finally, we proposed the Dynamic Clock Sets (*DCS* clocks), a new logical clock composed of a set of probabilistic clocks. The main feature of *DCS* clocks is that their size can be dynamically adjusted during execution. Such an elasticity is particularly interesting because the size of *DCS* and M-entry clocks depends on the number of concurrent messages in the system and to have knowledge about the latter beforehand is difficult or even impossible. We have defined *DCS* clocks as well as the operations required to change their size. We have also presented a causal broadcast implementation using them. Experimental results show that a causal broadcast algorithm implemented with *DCS* clocks delivers messages in causal order with a higher accuracy than one implemented with probabilistic clocks. Moreover, *DCS* clocks can adapt to increment themselves more efficiently, which increases their accuracy in causally ordering messages.

6.3 Future directions

This section discusses some future directions.

In the near future, we intend to conduct experiments to evaluate the extension of the causal broadcast algorithm for mobile networks that we proposed in Section 4 and that tolerates the failure of stations. To that end, stations are gathered in groups, and share causal information. We aim to evaluate the impact of the size of these groups in the number of messages exchanged between stations as well as the overhead due to the additional causal information and the delivery delays introduced by those message exchanges.

Another research direction will be how to detect concurrent messages when using causal broadcast implemented with M-entry clocks. For example, Torres-Rojas and Ahamad [TA99] use a scalar to distinguish some concurrent messages when using plausible clocks. Concerning the error detector, we would like to propose a handling procedure for messages tagged as not causally ordered by error detectors.

With respect to DCS clocks, we will investigate some heuristics to decide when to increase or reduce the size of the clock. Such heuristics would be particularly useful in systems where the message load varies abruptly.

Bibliography

[08] "IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems". In: *IEEE Std 1588-2008 (Revision of IEEE Std 1588-2002)* (2008), pp. 1–269.

[85] *Network Time Protocol (NTP)*. Request for Comments RFC 958. Internet Engineering Task Force, 1985. (Visited on 09/16/2022).

[AB94] A. Acharya and B.R. Badrinath. "Checkpointing Distributed Applications on Mobile Computers". In: *Proceedings of 3rd International Conference on Parallel and Distributed Information Systems*. 1994, pp. 73–80.

[ABF08] Paulo Almeida, Carlos Baquero, and Victor Fonte. "Interval Tree Clocks: A Logical Clock for Dynamic Systems". In: vol. 5401. 2008, pp. 259–274.

[ABL04] Giuseppe Anastasi, Alberto Bartoli, and Flaminia L. Luccio. "Fault-Tolerant Support for Reliable Multicast in Mobile Wireless Systems: Design and Evaluation". In: *Wireless Networks* 10.3 (2004), pp. 259–269. (Visited on 09/13/2022).

[ABS01] Giuseppe Anastasi, Alberto Bartoli, and Francesco Spadoni. "A Reliable Multicast Protocol for Distributed Mobile Systems: Design and Evaluation." In: *IEEE Trans. Parallel Distrib. Syst.* 12 (2001), pp. 1009–1022.

[ABS99] G. Anastasi, A. Bartoli, and F. Spadoni. "Group Multicast in Distributed Mobile Systems with Unreliable Wireless Network". In: *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*. 1999, pp. 14–23.

[AHJ91] M. Ahamad, P.W. Hutto, and R. John. "Implementing and Programming Causal Distributed Shared Memory". In: *[1991] Proceedings. 11th International Conference on Distributed Computing Systems*. 1991, pp. 274–281.

[Ala95] Sridhar Alagar. "Causally Ordered Message Delivery in Mobile Systems". In: 1995, pp. 169–175.

[AN96] Noha Adly and Magdy Nagi. "Maintaining Causal Order in Large Scale Distributed Systems Using a Logical Hierarchy". In: (1996).

[ANB93] Noha Adly, Magdy Nagi, and Jean Bacon. "A Hierarchical Asynchronous Replication Protocol for Large Scale Systems". In: 1993, pp. 152–157.

[Ara+18] João Paulo de Araujo et al. "A Communication-Efficient Causal Broadcast Protocol". In: *ICPP 2018 - 47th International Conference on Parallel Processing*. 2018. (Visited on 04/01/2023).

[ARV93] Sridhar Alagar, Ramki Rajagopalan, and S. Venkatesan. "Tolerating Mobile Support Station Failures". In: *Of the University of Texas at Dallas*. 1993, pp. 225–231.

[AV97] S. Alagar and S. Venkatesan. "Causal Ordering in Distributed Mobile Systems". In: *IEEE Transactions on Computers* 46.3 (1997), pp. 353–361.

[Bai+12] Peter Bailis et al. "The Potential Dangers of Causal Consistency and an Explicit Solution". In: *Proceedings of the Third ACM Symposium on Cloud Computing - SoCC '12*. San Jose, California: ACM Press, 2012, pp. 1–7. (Visited on 09/25/2022).

[BAI94] B.R. Badrinath, A. Acharya, and T. Imielinski. "Structuring Distributed Algorithms for Mobile Hosts". In: *14th International Conference on Distributed Computing Systems*. 1994, pp. 21–28.

[Bal+96] R. Baldoni et al. "Broadcast with Time and Causality Constraints for Multimedia Applications". In: *Proceedings of EUROMICRO 96. 22nd Euromicro Conference. Beyond 2000: Hardware and Software Design Strategies*. 1996, pp. 617–624.

[Bal+98] Roberto Baldoni et al. "Efficient Δ-Causal Broadcasting". In: *International Journal of Computer Systems Science and Engineering* 13 (1998), pp. 263–271.

[Bal98] R. Baldoni. "A Positive Acknowledgment Protocol for Causal Broadcasting". In: *IEEE Transactions on Computers* 47.12 (1998), pp. 1341–1350.

[BB08] Chafika Benzaid and N. Badache. "BMobi_Causal: A Causal Broadcast Protocol in Mobile Dynamic Groups". In: 2008, p. 421.

[BBG83] Anita Borg, Jim Baumbach, and Sam Glazer. "A Message System Supporting Fault Tolerance". In: *ACM SIGOPS Operating Systems Review* 17.5 (1983), pp. 90–99. (Visited on 09/14/2022).

[BCD17] Sebastian Blessing, Sylvan Clebsch, and Sophia Drossopoulou. "Tree Topologies for Causal Message Delivery". In: 2017, pp. 1–10.

[Bir85] Kenneth P. Birman. "Replication and Fault-Tolerance in the ISIS System". In: *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*. SOSP '85. New York, NY, USA: Association for Computing Machinery, 1985, pp. 79–86. (Visited on 09/11/2022).

[BJ87] Kenneth P. Birman and Thomas A. Joseph. "Reliable Communication in the Presence of Failures". In: *ACM Transactions on Computer Systems* 5.1 (1987), pp. 47–76. (Visited on 09/11/2022).

[Blo70] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Communications of the ACM* 13.7 (1970), pp. 422–426. (Visited on 09/22/2022).

[Bor13] Dhruba Borthakur. "Petabyte Scale Databases and Storage Systems at Facebook". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 1267–1268. (Visited on 04/06/2023).

[BRM96] Roberto Baldoni, Michel Raynal, and Achour Mostéfaoui. "Causal Delivery of Messages with Real-Time Data in Unreliable Networks". In: *Real-Time Systems* 10 (1996), pp. 245–262.

[BRV17] 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*. Belgrade Serbia: ACM, 2017, pp. 111–126. (Visited on 09/11/2022).

[BSS91] Kenneth Birman, André Schiper, and Pat Stephenson. "Lightweight Causal and Atomic Group Multicast". In: *ACM Transactions on Computer Systems* 9.3 (1991), pp. 272–314. (Visited on 09/11/2022).

[BV93] Ken Birman and Robbert Van Renesse. *Reliable Distributed Computing with the ISIS Toolkit*. 1993.

[Cac+01] Christian Cachin et al. "Secure and Efficient Asynchronous Broadcast Protocols". In: *Advances in Cryptology — CRYPTO 2001*. Ed. by Gerhard Goos et al. Vol. 2139. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 524–541. (Visited on 11/14/2022).

[Cas+12] Arnaud Casteigts et al. "Time-Varying Graphs and Dynamic Networks". In: *International Journal of Parallel, Emergent and Distributed Systems* 27.5 (2012), pp. 387–408. (Visited on 09/11/2022).

[CGK04] 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 (2004), pp. 40–52.

[Cha91] Bernadette Charron-Bost. "Concerning the Size of Logical Clocks in Distributed Systems". In: *Information Processing Letters* 39.1 (1991), pp. 11–16. (Visited on 09/11/2022).

[Chi+00] KH Chi et al. "A Causal Multicast Protocol for Mobile Distributed Systems". In: *IEICE Transactions on Information and Systems* E83D (2000), pp. 2065–2074.

[CK04] P. Chandra and A.D. Kshemkalyani. "Causal Multicast in Mobile Networks". In: *The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004. (MASCOTS 2004). Proceedings.* Volendam, The Netherlands, EU: IEEE, 2004, pp. 213–220. (Visited on 09/29/2022).

[CLZ02] Wentong Cai, Bu-Sung Lee, and Junlan Zhou. "Causal Order Delivery in a Multicast Environment: An Improved Algorithm". In: *Journal of Parallel and Distributed Computing* 62.1 (2002), pp. 111–131.

[CP06] Richard Crandall and Carl B. Pomerance. *Prime Numbers: A Computational Perspective*. Springer Science & Business Media, 2006.

[CS15] Tyler Crain and Marc Shapiro. "Designing a Causally Consistent Protocol for Geo-Distributed Partial Replication". In: *W. on Principles and Practice of Consistency for Distributed Data (PaPoC)*. Bordeaux, France: ACM, 2015. (Visited on 10/14/2022).

[DBK14] Elias Duarte Jr, Luis Carlos Bona, and Vinicius Kwiecien Ruoso. "VCube: A Provably Scalable Distributed Diagnosis Algorithm". In: 2014.

[de +17] João Paulo de Araujo et al. "A Publish/Subscribe System Using Causal Broadcast over Dynamically Built Spanning Trees". In: 2017, pp. 161–168.

[de +18a] João Paulo de Araujo et al. "A Communication-Efficient Causal Broadcast Protocol". In: *ICPP 2018: Proceedings of the 47th International Conference on Parallel Processing*. 2018, pp. 1–10.

[de +18b] João Paulo de Araujo et al. "VCube-PS: A Causal Broadcast Topic-Based Publish/Subscribe System". In: *Journal of Parallel and Distributed Computing* 125 (2018).

[DPG10] Eduardo Domínguez, Saul Pomares Hernandez, and Gustavo Gómez. "MOCAVI: An Efficient Causal Protocol for Cellular Networks". In: (2010).

[Du+14] Jiaqing Du et al. "GentleRain: Cheap and Scalable Causal Consistency with Physical Clocks". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. New York, NY, USA: Association for Computing Machinery, 2014, pp. 1–13. (Visited on 09/11/2022).

[Evr+16] Grigory Evropeytsev et al. "An Efficient Causal Group Communication Protocol for Free Scale Peer-to-Peer Networks". In: *Applied Sciences* 6.9 (2016), p. 234. (Visited on 09/12/2022).

[Evr+17] Grigory Evropeytsev et al. "An Efficient Causal Group Communication Protocol for P2P Hierarchical Overlay Networks". In: *Journal of Parallel and Distributed Computing* 102 (2017), pp. 149–162. (Visited on 09/23/2022).

[Fid88] Colin J. Fidge. "Timestamps in Message-Passing Systems That Preserve the Partial Ordering,"" in: *Proc. 11th Austral. Comput. Sci. Conf. (ACSC '88)*. 1988, pp. 56–66.

[FJ19] Denys Flores and Arshad Jhumka. "Hybrid Logical Clocks for Database Forensics: Filling the Gap between Chain of Custody and Database Auditing". In: 2019.

[FM04] Roy Friedman and Shiri Manor. *Causal Ordering in Deterministic Overlay Networks*. 2004.

[FZ94] G.H. Forman and J. Zahorjan. "The Challenges of Mobile Computing". In: *Computer* 27.4 (1994), pp. 38–47.

[GE12] John S. Gilmore and Herman A. Engelbrecht. "A Survey of State Persistency in Peer-to-Peer Massively Multiplayer Online Games". In: *IEEE Transactions on Parallel and Distributed Systems* 23.5 (2012), pp. 818–834.

[GP03] Anders Gidenstam and Marina Papatriantafilou. "Adaptive Plausible Clocks". In: *In ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04*. IEEE, 2003, pp. 86–93.

[Hei+12] Matthias Heinrich et al. "Exploiting Single-User Web Applications for Shared Editing: A Generic Transformation Approach". In: *Proceedings of the 21st international conference on World Wide Web* (2012), pp. 1057–1066. (Visited on 04/06/2023).

[HK17] Ta-Yuan Hsu and Ajay Kshemkalyani. "Value the Recent Past: Approximate Causal Consistency for Partially Replicated Systems". In: *IEEE Transactions on Parallel and Distributed Systems* PP (2017), pp. 1–1.

[HKS18] Ta-Yuan Hsu, Ajay D. Kshemkalyani, and Min Shen. "Causal Consistency Algorithms for Partially Replicated and Fully Replicated Systems". In: *Future Generation Computer Systems* 86 (2018), pp. 1118–1133. (Visited on 09/13/2022).

[HL11] Chih-Ming Hsiao and Yi-Pin Liao. "Domain-Based Causal Ordering Group Communication in Wireless Hybrid Networks". In: *Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication*. ICUIMC '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 1–6. (Visited on 09/23/2022).

[IB93] Tomasz Imielinski and B. Badrinath. "Mobile Wireless Computing: Solutions and Challenges in Data Management". In: *Communications of The ACM - CACM* 37 (1993).

[KAN22] Sandeep S Kulkarni, Gabe Appleton, and Duong Nguyen. "Achieving Causality with Physical Clocks". In: *23rd International Conference on Distributed Computing and Networking*. ICDCN 2022. New York, NY, USA: Association for Computing Machinery, 2022, pp. 97–106. (Visited on 09/13/2022).

[KKS18] Ajay D. Kshemkalyani, Ashfaq Khokhar, and Min Shen. "Encoded Vector Clock: Using Primes to Characterize Causality in Distributed Systems". In: *Proceedings of the 19th International Conference on Distributed Computing and Networking*. Varanasi India: ACM, 2018, pp. 1–8. (Visited on 09/11/2022).

[KM21] Ajay D. Kshemkalyani and Anshuman Misra. "The Bloom Clock to Characterize Causality in Distributed Systems". In: *Advances in Networked-Based Information Systems*. Ed. by Leonard Barolli et al. Vol. 1264. Cham: Springer International Publishing, 2021, pp. 269–279. (Visited on 09/22/2022).

[KO87] Hermann Kopetz and Wilhelm Ochsenreiter. "Clock Synchronization in Distributed Real-Time Systems". In: *IEEE Transactions on Computers* C-36.8 (1987), pp. 933–940.

[KS08] Ajay D. Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. First. USA: Cambridge University Press, 2008.

[KS96] Ajay D. Kshemkalyani and Mukesh Singhal. "An Optimal Algorithm for Generalized Causal Message Ordering". In: *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. PODC '96. New York, NY, USA: Association for Computing Machinery, 1996, p. 87. (Visited on 10/01/2022).

[KSV20] Ajay D. Kshemkalyani, Min Shen, and Bhargav Voleti. "Prime Clock: Encoded Vector Clock to Characterize Causality in Distributed Systems". In: *Journal of Parallel and Distributed Computing* 140 (2020), pp. 37–51.

[Kul+] Sandeep Kulkarni et al. "Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases". In: (), p. 13.

[Kul+14] Sandeep S. Kulkarni et al. "Logical Physical Clocks". In: *Principles of Distributed Systems*. Ed. by Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro. Vol. 8878. Cham: Springer International Publishing, 2014, pp. 17–32. (Visited on 09/14/2022).

[KV19] Ajay D. Kshemkalyani and Bhargav Voleti. "On the Growth of the Prime Numbers Based Encoded Vector Clock". In: *Distributed Computing and Internet Technology*. Ed. by Günter Fahrnberger, Sapna Gopinathan, and Laxmi Parida. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2019, pp. 169–184.

[KVP93] P. Krishna, N.H. Vaidya, and D.K. Pradhan. "Recovery in Distributed Mobile Environments". In: *Proceedings 1993 IEEE Workshop on Advances in Parallel and Distributed Systems*. 1993, pp. 83–88.

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (1978), pp. 558–565. (Visited on 09/11/2022).

[LH99] CHAO-PING LI and TING-LU HUANG. "A Mobile-Support-Station-Based Causal Multicast Algorithm in Mobile Computing Environment". In: 23 (1999).

[LKS11] Sangyoon Lee, Ajay D. Kshemkalyani, and Min Shen. "Performance Evaluation of Incremental Vector Clocks". In: *2011 10th International Symposium on Parallel and Distributed Computing*. 2011, pp. 117–124.

[LSB06] Cristian Lumezanu, Neil Spring, and Bobby Bhattacharjee. "Decentralized Message Ordering for Publish/Subscribe Systems". In: *Middleware 2006*. Ed. by Maarten van Steen and Michi Henning. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 162–179.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine Generals Problem". In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401. (Visited on 09/11/2022).

[Mar+10] Jim Martin et al. *Network Time Protocol Version 4: Protocol and Algorithms Specification*. Request for Comments RFC 5905. Internet Engineering Task Force, 2010. (Visited on 09/16/2022).

[Mat80] Friedemann Mattern. *Virtual Time and Global States of Distributed Systems*. 1980.

[Mia+16] Guowang Miao et al. *Fundamentals of Mobile Data Networks*. Cambridge: Cambridge University Press, 2016. (Visited on 02/17/2023).

[MK21] Anshuman Misra and Ajay Kshemkalyani. "The Bloom Clock for Causality Testing". In: 2021, pp. 3–23.

[MW17a] Achour Mostefaoui and Stéphane Weiss. "A Probabilistic Causal Message Ordering Mechanism". Report. LS2N, Université de Nantes, 2017, p. 11. (Visited on 01/11/2023).

[MW17b] Achour Mostéfaoui and Stéphane Weiss. "Probabilistic Causal Message Ordering". In: 2017, pp. 315–326.

[Nis+05] T. Nishimura et al. "Causally Ordered Delivery with Global Clock in Hierarchical Group". In: vol. 2. 2005, 560–564 Vol. 2.

[NMM16] Brice Nédelec, Pascal Molli, and Achour Mostéfaoui. "CRATE: Writing Stories Together with Our Browsers". In: *WWW '16 Companion: Proceedings of the 25th International Conference Companion on World Wide Web*. 2016, pp. 231–234.

[NMM18a] Brice Nedelec, Pascal Molli, and Achour Mostéfaoui. "Breaking the Scalability Barrier of Causal Broadcast for Large and Dynamic Systems". In: 2018, pp. 51–60.

[NMM18b] Brice Nédelec, Pascal Molli, and Achour Mostefaoui. "Causal Broadcast: How to Forget?" In: *The 22nd International Conference on Principles of Distributed Systems (OPODIS)*. Hong Kong, China, 2018. (Visited on 09/14/2022).

[PDG10] Saul Pomares Hernandez, Eduardo Domínguez, and Gustavo Gómez. "An Efficient Delta-Causal Distributed Algorithm for Synchronous Cooperative Systems in Unreliable Networks". In: *Computación y Sistemas* 14 (2010), pp. 31–44.

[PK21] Tommaso Pozzetti and Ajay D. Kshemkalyani. "Resettable Encoded Vector Clock for Causality Analysis With an Application to Dynamic Race Detection". In: *IEEE Transactions on Parallel and Distributed Systems* 32.4 (2021), pp. 772–785.

[PKV96] D.K. Pradhan, P. Krishna, and N.H. Vaidya. "Recoverable Mobile Environment: Design and Trade-off Analysis". In: *Proceedings of Annual Symposium on Fault Tolerant Computing*. 1996, pp. 16–25.

[Ple+06] C. Plesca et al. "A Coordination-Level Middleware for Supporting Flexible Consistency in CSCW". In: *14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP'06)*. 2006, 6 pp.-.

[PR94] Boaz Patt-Shamir and Sergio Rajsbaum. "A Theory of Clock Synchronization (Extended Abstract)". In: *Proceedings of the Twenty-Sixth Annual ACM Symposium on Theory of Computing*. STOC '94. New York, NY, USA: Association for Computing Machinery, 1994, pp. 810–819. (Visited on 09/11/2022).

[PRS96] R. Prakash, M. Raynal, and M. Singhal. "An Efficient Causal Ordering Algorithm for Mobile Computing Environments". In: *Proceedings of 16th International Conference on Distributed Computing Systems*. 1996, pp. 744–751.

[PRS97] Ravi Prakash, Michel Raynal, and Mukesh Singhal. "An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments". In: *Journal of Parallel and Distributed Computing* 41.2 (1997), pp. 190–204. (Visited on 09/28/2022).

[PS97] Ravi Prakash and Mukesh Singhal. "Dependency Sequences and Hierarchical Clocks: Efficient Alternatives to Vector Clocks for Mobile Computing Systems". In: *Wireless Networks* 3.5 (1997), pp. 349–360. (Visited on 09/23/2022).

[Ram19] Lum Ramabaja. *The Bloom Clock*. 2019. arXiv: `arXiv:1905.13064`. (Visited on 09/22/2022).

[Ray13] Michel Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer Publishing Company, Incorporated, 2013.

[RCD10] Alan Ritter, Colin Cherry, and Bill Dolan. "Unsupervised Modeling of Twitter Conversations". In: *Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational Linguistics*. Los Angeles, California: Association for Computational Linguistics, 2010, pp. 172–180. (Visited on 09/26/2022).

[RGI] Golden Richard, Prof Golden, and G. Richard Iii. *Efficient Vector Time with Dynamic Process Creation and Termination*.

[RK16] Mohammad Roohitavaf and Sandeep Kulkarni. "GentleRain+: Making GentleRain Robust on Clock Anomalies". In: (2016).

[RMK17] Mohammad Roohitavaf, Demirbas Murat, and Sandeep Kulkarni. "CausalSpartan: Causal Consistency for Distributed Data Stores Using Hybrid Logical Clocks". In: 2017.

[Rod+00] L. Rodrigues et al. "Deadline-Constrained Causal Order". In: *Proceedings Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2000) (Cat. No. PR00607)*. Newport, CA, USA: IEEE Comput. Soc, 2000, pp. 234–241. (Visited on 09/13/2022).

[Roo+19] Mohammad Roohitavaf et al. "Session Guarantees with Raft and Hybrid Logical Clocks". In: *ICDCN '19: Proceedings of the 20th International Conference on Distributed Computing and Networking*. 2019, pp. 100–109.

[RS96] M. Raynal and M. Singhal. "Logical Time: Capturing Causality in Distributed Systems". In: *Computer* 29.2 (1996), pp. 49–56.

[RST91] Michel Raynal, André Schiper, and Sam Toueg. "The Causal Ordering Abstraction and a Simple Way to Implement It". In: *Information Processing Letters* 39.6 (1991), pp. 343–350. (Visited on 09/11/2022).

[SES89] André Schiper, Jorge Eggli, and Alain Sandoz. "A New Algorithm to Implement Causal Ordering". In: *Proceedings of the 3rd International Workshop on Distributed Algorithms*. Berlin, Heidelberg: Springer-Verlag, 1989, pp. 219–232. (Visited on 09/11/2022).

[Sha+11] Marc Shapiro et al. "A Comprehensive Study of Convergent and Commutative Replicated Data Types". Report. Inria – Centre Paris-Rocquencourt ; INRIA, 2011, p. 50. (Visited on 04/06/2023).

[SK92] Mukesh Singhal and Ajay Kshemkalyani. "An Efficient Implementation of Vector Clocks". In: *Information Processing Letters* 43.1 (1992), pp. 47–52. (Visited on 09/11/2022).

[SM94] Reinhard Schwarz and Friedemann Mattern. "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail". In: *Distributed Computing* 7.3 (1994), pp. 149–174. (Visited on 09/11/2022).

[SMG98] Chakarat Skawratananond, Neeraj Mittal, and Vijay K. Garg. *A Lightweight Algorithm for Causal Message Ordering in Mobile Computing Systems*. Tech. rep. In Proc. of 12th ISCA Intl. Conf. on Parallel and Distributed Computing Systems (PDCS, 1998.

[SR19] Valter Santos and Luis Rodrigues. "Localized Reliable Causal Multicast". In: *2019 IEEE 18th International Symposium on Network Computing and Applications (NCA)*. Cambridge, MA, USA: IEEE, 2019, pp. 1–10. (Visited on 09/11/2022).

[SS83a] Richard D. Schlichting and Fred B. Schneider. "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems". In: *ACM Transactions on Computer Systems* 1.3 (1983), pp. 222–238. (Visited on 10/02/2022).

[SS83b] D. Skeen and M. Stonebraker. "A Formal Model of Crash Recovery in a Distributed System". In: *IEEE Transactions on Software Engineering* SE-9.3 (1983), pp. 219–228.

[TA99] Francisco J. Torres-Rojas and Mustaque Ahamad. "Plausible Clocks: Constant Size Logical Clocks for Distributed Systems". In: *Distributed Computing* 12.4 (1999), pp. 179–195. (Visited on 09/21/2022).

[TEL12] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. "Theory and Practice of Bloom Filters for Distributed Systems". In: *Communications Surveys & Tutorials, IEEE* 14 (2012), pp. 131–155.

[Ter+95] D. B. Terry et al. "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System". In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. New York, NY, USA: Association for Computing Machinery, 1995, pp. 172–182. (Visited on 04/06/2023).

[TET04] K. Taguchi, Tomoya Enokido, and Makoto Takizawa. "Causally Ordered Delivery for a Hierarchical Group". In: 2004, pp. 453–460.

[THT98] Takayuki Tachikawa, Hiroaki Higaki, and Makoto Takizawa. "Group Communication Protocol for Realtime Applications." In: 1998, pp. 40–47.

[TKB92] A.S. Tannenbaum, M.F. Kaashoek, and H.E. Bal. "Parallel Programming Using Shared Objects and Broadcasting". In: *Computer* 25.8 (1992), pp. 10–19. (Visited on 11/14/2022).

[Tor01] Francisco J. Torres-Rojas. "Performance Evaluation of Plausible Clocks". In: *Euro-Par 2001 Parallel Processing*. Ed. by Rizos Sakellariou et al. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2001, pp. 476–481.

[TT03] Kojiro Taguchi and Makoto Takizawa. "Two-Layered Protocol for a Large-Scale Group of Processes." In: *J. Inf. Sci. Eng.* 19 (2003), pp. 451–465.

[Tyu+19] Misha Tyulenev et al. “Implementation of Cluster-wide Logical Clock and Causal Consistency in MongoDB”. In: 2019, pp. 636–650.

[Var01] András Varga. “The OMNET++ Discrete Event Simulation System”. In: *Proc. ESM'2001* 9 (2001).

[Wak93] I. Wakeman. “Packetized Video—Options for Interaction between the User, the Network and the Codec”. In: *The Computer Journal* 36.1 (1993), pp. 55–67. (Visited on 09/26/2022).

[Wan+06] Xinli Wang et al. “An Efficient Implementation of Vector Clocks in Dynamic Systems.” In: 2006, pp. 593–599.

[XFJ03] B. Bui Xuan, A. Ferreira, and A. Jarry. “COMPUTING SHORTEST, FASTEST, AND FOREMOST JOURNEYS IN DYNAMIC NETWORKS”. In: *International Journal of Foundations of Computer Science* 14.02 (2003), pp. 267–285. (Visited on 11/14/2022).

[Yav92] R. Yavatkar. “MCP: A Protocol for Coordination and Temporal Synchronization in Multimedia Collaborative Applications”. In: *[1992] Proceedings of the 12th International Conference on Distributed Computing Systems*. 1992, pp. 606–613.

[YHH97] Li-Hsing Yen, Ting-Lu Huang, and Shu-Yuen Hwang. “A Protocol for Causally Ordered Message Delivery in Mobile Computing Systems”. In: *Mobile Networks and Applications* 2.4 (1997), pp. 365–372. (Visited on 09/14/2022).

[YW10] Shaozhi Ye and S. Felix Wu. “Measuring Message Propagation and Social Influence on Twitter.Com”. In: *Social Informatics*. Ed. by Leonard Bolc, Marek Makowski, and Adam Wierzbicki. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2010, pp. 216–231.