On Multicast Primitives in Large Networks and Partial Replication Protocols

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Nicolas Schiper

under the supervision of
Fernando Pedone

October 2009

Dissertation Committee

Antonio Carzaniga	University of Lugano, Switzerland
Matthias Hauswirth	University of Lugano, Switzerland
Lorenzo Alvisi	University of Texas at Austin, United States
Robbert van Renesse	Cornell University, United States
Willy Zwaenepoel	EPFL, Switzerland

Dissertation accepted on 19 October 2009

Research Advisor	PhD Program Director
Fernando Pedone	**Fabio Crestani**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Nicolas Schiper
Lugano, 19 October 2009

To Françoise, André, and Isabelle,
for being such a wonderful family.

Abstract

Recent years have seen the rapid growth of web-based applications such as search engines, social networks, and e-commerce platforms. As a consequence, our daily life activities rely on computers more and more each day. Designing reliable computer systems has thus become of a prime importance. Reliability alone is not sufficient however. These systems must support high loads of client requests and, as a result, scalability is highly valued as well.

In this thesis, we address the design of fault-tolerant computer systems. More precisely, we investigate the feasibility of designing scalable database systems that offer the illusion of accessing a single copy of a database, despite failures. This study is carried out in the context of large networks composed of several *groups* of machines located in the same geographical region. Groups may be data centers, each located in a local area network, connected through high-latency links. In these settings, the goal is to minimize the use of inter-group links. We mask failures using data replication: if one copy of the data is not available, a replica is accessed instead. Guaranteeing data consistency in the presence of failures while offering good performance constitutes the main challenge of this thesis.

To reach this goal, we first study fault-tolerant multicast communication primitives that offer various message ordering guarantees. We then rely on these multicast abstractions to propose replication protocols in which machines hold a subset of the application's data, denoted as partial replication. In contrast to full replication, partial replication may potentially offer better scalability since updates need not be applied to every machine in the system. More specifically, this thesis makes contributions in the distributed systems domain and in the database domain.

In the distributed systems domain, we first devise FIFO and causal multicast algorithms, primitives that ease the design of replicated data management protocols, as we will show. We then study atomic multicast, a basic building block for synchronous database replication. Two failure models are considered: one in which groups are correct, i.e., groups contain at least one process that is always

up, and one in which groups may fail entirely. We show a tight lower bound on the minimum number of inter-group message delays required for atomic multicast in the first failure model. When an arbitrary number of processes may fail and process failures may not be predicted, we demonstrate that erroneous process failure suspicion cannot be tolerated. We then present atomic multicast protocols for the case of correct and faulty groups and empirically compare their performance. The majority of the proposed algorithms are latency-optimal.

In the database domain, we extend the database state machine (DBSM), a previously proposed full replication technique, to partial replication. In the DBSM, transactions are executed locally at one database site according to the strict two-phase locking policy. To ensure global data consistency, a certification protocol is triggered at the end of each transaction. We present three certification protocols that differ in the communication primitives they use and the amount of information related to transactions they store. The first two algorithms are tailored for local area networks and ensure that sites unrelated to a transaction T only permanently store the identifier of T. The third protocol is more generic since it is not customized for any type of network. Furthermore, with this protocol, only sites that replicate data items read or updated by T are involved in T's certification.

Acknowledgements

I am very thankful to a number of people who contributed in their own way to this thesis. First of all, I wish to thank Fernando Pedone for advising me with great care and expressing his confidence in me.

I thank Rodrigo Schmidt and Lásaro Camargos for challenging my opinions on various topics of distributed computing, especially Paxos. I also wish to express my gratitude to Pierre Sutra, for his collaboration in the experimental evaluation of atomic multicast algorithms, and Sam Toueg, for the enlightening and passionate discussions we had.

I am grateful to the dissertation committee members, Antonio Carzaniga, Matthias Hauswirth, Lorenzo Alvisi, Robbert van Renesse, and Willy Zwaenepool for their feedback and the time they spent examining this thesis.

Last but not least, I would like to thank the following graduate students at the University of Lugano for all the good times we shared and for their friendship: Giovanni Ansaloni, Paolo Bonzini, Lásaro Camargos, Giovanni Ciampaglia, Leonardo Fernandes, Anna Förster, Shima Gerani, Cyrus Hall, Mostafa Kheika, Mircea Lungu, Amirhossein Malekpour, Parvaz Mahdabi, Parisa Marandi, Mehdi Mir, Vaidė Narváez, Edgar Pek, Jeff Rose, Rodrigo Schmidt, Aliaksei Tsitovich, and Marcin Wieloch.

Preface

This thesis concerns the Ph.D. work done under the supervision of Prof. Fernando Pedone at the University of Lugano, from 2005 to 2009. It focuses on communication primitives for large networks and protocols for partial replication. The result of this work appears in several papers and a technical report:

N. Schiper, R. Schmidt, and F. Pedone. Optimistic algorithms for partial database replication. In *Proceedings of OPODIS'06*, pages 81–93. Springer, 2006. Brief Announcement in *Proceedings of DISC'06*, pages 557–559. Springer, 2006.

N. Schiper and F. Pedone. Optimal atomic broadcast and multicast algorithms for wide area networks. Brief Announcement in *Proceedings of PODC'07*, pages 384–385. ACM Press, 2007.

N. Schiper and F. Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. In *Proceedings of ICDCN'08*, pages 147–157. Springer, 2008.

N. Schiper and F. Pedone. Solving atomic multicast when groups crash. In *Proceedings of OPODIS'08*, pages 481–495. Springer, 2008.

N. Schiper and P. Sutra and F. Pedone. Genuine versus Non-Genuine Atomic Multicast Protocols for Wide Area Networks: An Empirical Study. To appear in *Proceedings of SRDS'09*, IEEE Computer Society, 2009.

N. Schiper and F. Pedone. Fast, Scalable, Flexible, and Highly Resilient Fifo and Causal Multicast Algorithms. Technical Report 2009/003, University of Lugano, 2009.

Outside of the main scope of this research, I also worked on the implementation and evaluation of a leader election service. Leader election plays an important role in the field of fault-tolerance: it may be used to solve consensus, a basic building-block at the core of data replication.

N. Schiper and S. Toueg. A robust and lightweight stable leader election service for dynamic systems. In *Proceedings of DSN'08*, pages 207–216. IEEE Computer Society, 2008.

Contents

Figures

Tables

Chapter 1

Introduction

If you know exactly what you are going to do,
what is the point of doing it?

Pablo Picasso

Demand for cheap and reliable computer systems has dramatically increased over the years. Internet-scale applications rely on machines located in data centers spread around the globe to provide low response times and high availability to clients. In this context, machines failures may have various causes ranging from defective hardware to data center fires. A natural way of masking these failures is by means of replication: as soon as one machine fails and cannot offer its service anymore, one of its replicas takes over. The main challenge in achieving replication is to keep the state of the replicas consistent at all times, despite machine crashes and the system's asynchrony.

1.1 Motivation

To tackle the complexity of data replication, several fault-tolerant communication primitives have been defined [10; 32]. All of these primitives allow to multicast messages to a set of processes, possibly located on different machines, and ensure agreement on the set of messages delivered, despite failures and unpredictable message delays. In addition, these communication abstractions offer various message ordering guarantees, ranging from first-in-first-out order, referred to as FIFO order, where messages originating from the same process are delivered in the order they were multicast, to total order, where processes

agree on the message delivery order. Atomic broadcast and multicast, primitives that ensure total ordering of messages, have been used in numerous replication schemes, notably in the context of distributed databases [2; 45; 33; 34; 47; 55; 62; 39; 46]. Thanks to the adequacy of the properties they offer, most of the complexity involved in synchronizing database replicas is handled by the group communication layer.

Previous work on group communication-based database replication has focused mainly on full replication. However, full replication might not always be adequate. First, sites might not have enough disk or memory resources to replicate the database fully. Second, full replication provides limited scalability since every update transaction is executed by each replica. Third, when access locality is observed, full replication is pointless. Replication in large networks, e.g., wide area networks, is another interesting and related question. Two main reasons motivate large area network replication: First, if clients accessing the database are spread over a large geographical region, it is a good idea to place replicas as close to the clients as possible so as to minimize the clients' access latencies. Second, to tolerate natural disasters such as earthquakes or hurricanes, database replicas must be placed as far as possible from each other.

Efficient replication protocols offering these advantages require multicast algorithms tailored for large networks. In general, these networks are composed of several *groups* of machines located in the same geographical region. Groups may be data centers, each located in a local area network (LAN), connected through continental or even inter-continental links. The latency of intra-group and inter-group communication links is often separated by several orders of magnitude. Protocols should thus use inter-group communication links sparingly. In the replicated systems we envision, high availability is provided inside groups by tolerating machine crashes and across the system by coping with group crashes.

The main challenge addressed by this thesis is to provide partial replication protocols that are scalable, use inter-group links sparingly, and are as resilient to process failures as possible. Achieving all of these goals simultaneously is not an easy task. In fact, existing protocols either (i) assume full replication, and thus exhibit poor scalability, (ii) do not tolerate group crashes, or (iii) exhibit prohibitive latencies in wide area networks. Motivated by these observations, this thesis first studies multicast communication primitives that are at the core of partial replication protocols. We then show how to build partial replication protocols on top of these primitives.

1.2 Research Contributions

This thesis provides four major contributions. In the distributed systems domain, we study and devise multicast algorithms offering different message ordering guarantees. In the database domain, we show how partial replication protocols can be built on top of the adequate group communication primitives.

FIFO and Causal Multicast. FIFO and causal multicast help the programming of distributed applications in various domains such as global snapshot construction [6], fair resource allocation [37], and replicated data management [57]. These group communication primitives have been extensively studied in the literature. However, previously proposed solutions either do not tolerate quasi-reliable networks, in which a message sent can be lost because of the crash of its sender, or disallow sending messages to groups the sender does not belong to. In contrast, we propose algorithms that tolerate quasi-reliable networks, allow messages to be multicast to any subset of groups, and tolerate an arbitrary number of process failures.

Atomic Multicast. We devise atomic multicast algorithms in large scale networks and seek to minimize the number of inter-group message delays between the multicast of a message and its delivery. We study the minimal cost of multicast in the case where groups are *correct*, i.e., groups contain at least one process that is always up, and in the case of data center disasters, where groups may crash entirely. In particular, we are interested in the consequences of requiring multicast algorithms to be *genuine*: to deliver a message m, only m's sender and addressees may participate in the protocol. In the context of correct groups, we show that genuine atomic multicast requires an extra inter-group communication step compared to its non-genuine counterpart. When failures cannot be predicted and groups may crash entirely, we demonstrate that genuine multicast demands perfectly accurate information about process failures: processes may not erroneously suspect other processes to have crashed. As an alternative, we devise a non-genuine protocol that weakens the failure detection requirements and allows optimal message delivery latency.

Evaluation of Multicast Protocols. In the context of correct groups, genuine multicast is more expensive in terms of latency than its non-genuine counterpart. When choosing a multicast algorithm, it seems natural to question under which circumstances a genuine algorithm is to be preferred over a non-genuine

algorithm. We answer this question by experimentally evaluating the performance of latency-optimal genuine and non-genuine algorithms proposed in this thesis. As part of the empirical study, we assess the scalability of the protocols by varying the number of groups, the proportion of global messages, and the load, i.e., the frequency at which messages are multicast. To complete our study, we measure the overhead of a disaster-tolerant multicast protocol presented in this dissertation. We also identify a *convoy effect* in multicast algorithms that delay the delivery of messages and propose techniques to reduce this effect.

Partial Replication. Partial replication allows database sites to replicate a subset of the application's data. Compared to full replication, higher transaction loads can potentially be supported since updates need not be applied to all replicas. We show how partial replication protocols can be built on top of multicast primitives by extending the database state machine (DBSM), a replication technique initially proposed in the context of full replication [47]. In the DBSM, transactions are executed locally on database sites according to the two-phase locking policy. To guarantee global consistency, a certification protocol is triggered at the end of each transaction. We present three different certification protocols. The first two are tailored for local area networks and ensure that sites unrelated to a transaction T only permanently store the identifier of T. The third protocol is more generic since it is not customized for any type of network. Furthermore, with this protocol, only sites that replicate data items read or updated by T are involved in T's certification.

1.3 Thesis Outline

This thesis is organized as follows. Chapter 2 describes the system model, defines fault-tolerant multicast and related problems, and provides database definitions and notations used throughout this thesis. Chapter 3 presents latency-optimal FIFO and causal multicast algorithms that tolerate an arbitrary number of process failures. Chapter 4 addresses the problem of atomic multicast in large networks composed of several groups of machines, where intra-group and inter-group communication latency are separated by several orders of magnitude. We first consider that each group contains at least one correct process and then relax this assumption by allowing groups to crash entirely. Chapter 5 empirically compares genuine and non-genuine latency-optimal protocols proposed in Chapter 4. Chapter 6 extends the database state machine to partial replication and proposes three protocols that are built on top of group communication

primitives. Finally, Chapter 7 concludes this thesis and proposes future research directions.

Chapter 2

System Model and Definitions

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> *Leslie Lamport*

A system model describes precisely and concisely all the hypotheses about the system. We here define the system model assumed throughout this thesis, define some fault-tolerant problems of interest to this work, and present important database definitions.

2.1 Processes and Links

We consider a system $\Pi = \{p_1, ..., p_n\}$ of processes which communicate through message passing and do not have access to a shared memory or a global clock. Processes may however access failure detectors [15]. We assume the benign crash-stop failure model: processes may fail by crashing, but do not behave maliciously. A process that never crashes is *correct*; otherwise it is *faulty*. The maximum number of processes that may crash is denoted by f.

The system is asynchronous, i.e., messages may experience arbitrarily large (but finite) delays and there is no bound on relative process speeds. Furthermore, the communication links do not corrupt nor duplicate messages and are quasi-reliable, more precisely:

- *uniform integrity:* For any process p and message m, p receives m at most once, and only if m was previously sent to p.

- *quasi-reliability:* For any two *correct* processes p and q, and any message m, if p sends m to q, then q eventually receives m.

We define $\Gamma = \{g_1, ..., g_m\}$ as the set of process groups in the system. Groups are disjoint, non-empty, and satisfy $\bigcup_{g \in \Gamma} g = \Pi$. For each process $p \in \Pi$, group(p) identifies the group p belongs to. A group g that contains at least one correct process is correct; otherwise g is faulty.

2.2 Fault-Tolerant Multicasts and Related Problems

Group communication was initially proposed as part of the V kernel [17]. This seminal paper introduced the idea of multicasting a message to a group of processes hosted on different machines. Several message ordering semantics such as causal and total order were later introduced in [10; 32]. All of these group communication primitives ensure *reliability*—agreement on the set of messages delivered—but offer various message ordering semantics. For example, causal order stipulates that the message delivery order does not violate causality, i.e., if the multicast of a message m *causally precedes* the multicast of a message m', then processes do not deliver m' unless they delivered m previously. With total order, all processes deliver messages in the same order but not necessarily in causal order.

These communication abstractions ease the design of distributed applications. For instance, causal multicast may be used to construct consistent global snapshots of a distributed system [6]; total order multicast, also referred to as *atomic multicast*, supports data replication by ensuring that all client operations are executed in the same order at all replicas [37].

In Figure 2.1, we illustrate how an application interacts with the group communication layer. To *multicast* a message m to a set of groups, the application invokes the group communication abstraction. When a message m is ready to be delivered, the application is notified through a *deliver(m)* callback. To implement the reliability and various message ordering guarantees despite process failures, the group communication layer uses the physical network to exchange *protocol messages* with the other machines in the system.

Below, we define several reliable group communication primitives and one agreement problem, namely consensus. These definitions, except for the uniform prefix order property of atomic multicast, are borrowed from [32]. Let $\mathscr{A}$ be an algorithm solving a problem. We define $\mathscr{R}(\mathscr{A})$ as the set of all admissible runs of $\mathscr{A}$.

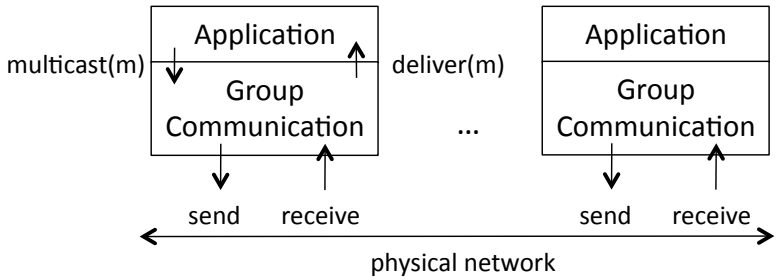


Figure 2.1. The application and group communication layers.

Reliable Multicast Our algorithms use a *uniform reliable multicast* primitive that allows to multicast messages reliably to a subset of the groups in Γ. For each message m, $m.dst$ denotes the groups to which the message is reliably multicast. Let p be a process. By abuse of notation, we write $p \in m.dst$ instead of $\exists g \in \Gamma : g \in m.dst \wedge p \in g$. Uniform reliable multicast is defined by primitives *R-MCast(m)* and *R-Deliver(m)*, and satisfies the following properties:

- *uniform integrity:* For any process p and any message m, p R-Delivers m at most once, and only if $p \in m.dst$ and m was previously R-MCast.
- *validity:* If a correct process p R-MCasts a message m, then eventually all correct processes $q \in m.dst$ R-Deliver m.
- *uniform agreement:* If a process p R-Delivers a message m, then eventually all correct processes $q \in m.dst$ R-Deliver m.

FIFO Multicast FIFO multicast ensures that the delivery order of messages multicast by some process q follows the order in which these messages were multicast. More precisely, uniform FIFO multicast is defined by primitives *F-MCast(m)* and *F-Deliver(m)* and satisfies the uniform integrity, validity, and uniform agreement properties of reliable multicast as well as the following property:

- *uniform FIFO order:* If a process p F-MCasts a message m before F-MCasting a message m', then no process in $m.dst \cap m'.dst$ F-Delivers m' unless it has previously F-Delivered m.

Causal Multicast Causal multicast is defined by primitives *C-MCast(m)* and *C-Deliver(m)*, and satisfies the uniform integrity, validity, and uniform agreement properties of reliable multicast as well as the following property:

- *uniform causal order:* For any two messages m and m', if C-MCast(m) $\rightarrow$ C-MCast(m'), then no process $p \in m.dst \cap m'.dst$ C-Delivers m' unless it has previously C-Delivered m.[1]

Atomic Multicast Atomic multicast allows messages to be A-MCast to any subset of groups in Γ. For every message m, $m.dst$ denotes the groups to which m is multicast. A message m is multicast by invoking A-MCast(m) and delivered with A-Deliver(m). Consider the set of messages processes A-Deliver. We define the relation $<$ on this set as follows: $m < m'$ iff there exists a process that A-Delivers m before m'. Atomic multicast satisfies the uniform integrity, validity, and uniform agreement properties of reliable multicast as well as:

- *uniform prefix order:* For any two messages m and m' and any two processes p and q such that $\{p,q\} \subseteq m.dst \cap m'.dst$, if p A-Delivers m and q A-Delivers m', then either p A-Delivers m' before m or q A-Delivers m before m'.

- *uniform acyclic order:* The relation $<$ is acyclic.

Informally, the uniform acyclic order property ensures that processes agree on the order of messages they A-Deliver and forbids cycles from occurring in the delivery sequence. Suppose three groups exist in the system, g_x, g_y, and g_z, and three messages $m_{x,z}$, $m_{y,x}$, $m_{z,y}$ are multicast such that message $m_{i,j}$ is addressed to groups $\{g_i, g_j\}$. Without the uniform acyclic order property, we could have $m_{x,z} < m_{y,x}$, $m_{y,x} < m_{z,y}$, and $m_{z,y} < m_{x,z}$. This scenario is problematic for the following reason: suppose that group g_i replicates object i and that message $m_{i,j}$ writes i and reads j. Hence, with the message delivery order specified above, there exists no serial order in which messages can *logically* be placed such that it represents the global execution of these operations. Indeed, since message $m_{i,j}$ writes i and reads j, the serial order *SO* must follow the message delivery order *MO*. Since *MO* is cyclic, *SO* cannot be serial.

The uniform acyclic property alone is not sufficient however. Indeed, without uniform prefix order, atomic multicast allows holes to appear in the message delivery sequence. Consider a run R where three messages m_1, m_2, and m_3 are addressed to a group g. A process p of g A-Delivers m_1, m_2, and m_3 in that order. Another faulty member q of g A-Delivers m_1 directly followed by m_3. In

[1] The relation $\rightarrow$ is Lamport's transitive happened before relation on events [37]. Here, events can be of two types, C-MCast or C-Deliver. The relation is defined as follows: $e_1 \rightarrow e_2 \Leftrightarrow e_1, e_2$ are two events on the same process and e_1 happens before e_2 or e_1 = C-MCast(m) and e_2 = C-Deliver(m) for some message m.

R, the delivery sequence of q contains a hole: q does not A-Deliver m_2. Note that none of the properties of atomic multicast, apart from uniform prefix order, are violated in R. In particular, uniform agreement does not force q to A-Deliver m_2 since q is faulty. However, if we apply the uniform prefix order property to this run, q is forced to A-Deliver m_2 before m_3. Indeed, by this property, either p A-Delivers m_3 before m_2 or q A-Delivers m_2 before m_3. The first case is impossible since p would A-Deliver m_3 twice, contradicting uniform integrity. Hence, only the second case is possible and q A-Delivers m_2 before m_3.

Protocols solving reliable, FIFO, causal, and atomic multicast may be *genuine*, i.e., to deliver a message m, only the addressees of m participate in the protocol [31]. More precisely:

- *Genuineness*: An algorithm $\mathscr{A}$ solving multicast is said to be *genuine* iff for any run $R \in \mathscr{R}(\mathscr{A})$ and for any process p, in R, if p sends or receives a message then some message m is multicast and either p is the process that multicasts m or $p \in m.dst$.

Broadcasts Reliable, FIFO, Causal, and Atomic broadcast satisfy the same properties as their respective multicast counterpart. With broadcast however, all messages m are such that $m.dst = \Gamma$, i.e., messages are always broadcast to all groups in the system. Note that for atomic broadcast, the uniform acyclic order property is superfluous: the uniform prefix and integrity properties imply the uniform acyclic order property.

In local-area networks, some implementations of reliable broadcast can take advantage of network hardware characteristics to deliver messages in total order with high probability [49]. We call such a primitive Weak Ordering Reliable Broadcast, WOR-Broadcast.

Consensus We also assume the existence of a *uniform consensus* abstraction. In the *consensus* problem, processes propose values and must reach agreement on the value decided. Uniform consensus is defined by the primitives propose(v) and decide(v) and satisfies the following properties:

- *uniform integrity:* If a process decides v, then v was previously proposed by some process.
- *uniform agreement:* If a process decides v, then all correct processes eventually decide v.
- *termination:* Every correct process eventually decides exactly one value.

2.3 Failure Detectors

Fischer, Lynch, and Patterson showed in [27] that consensus is not solvable in an asynchronous system even if only one process is allowed to crash. To circumvent this impossibility result, the research community proposed a number of approaches: randomized algorithms [8; 50], partially synchronous systems [26; 16], and failure detector oracles [15]. These oracles provide possibly inaccurate information about process failures. For example, a failure detector may suspect a process that has not failed or it may never suspect a process that crashed. Many different failure detectors have been introduced in the literature, usually differing on what properties they ensure with regards to correct processes (accuracy) and crashed ones (completeness).

Perhaps the most famous among all of them is the leader failure detector Ω, which allows to solve consensus when a majority of processes are correct [14]. At each process p, this failure detector outputs the identity of a process $leader_p$ and ensures the following property: if there exists a correct process, then there exists a correct process l and a time after which, for every correct process p, $leader_p = l$.

A natural question to ask is what is the minimal amount of information a failure detector needs to provide to solve a given agreement problem P, such as consensus or atomic multicast for example. Such a "minimal" failure detector is the *weakest* for solving P. More formally, a failure detector $\mathscr{D}_1$ is at least as strong as a failure detector $\mathscr{D}_2$, denoted as $\mathscr{D}_1 \succeq \mathscr{D}_2$, if and only if there exists an algorithm that implements $\mathscr{D}_2$ using $\mathscr{D}_1$, i.e., the algorithm emulates the output of $\mathscr{D}_2$ using $\mathscr{D}_1$. A failure detector $\mathscr{D}^*$ is the weakest failure detector for P if two conditions are met: we can use $\mathscr{D}^*$ to solve P (sufficiency) and any failure detector $\mathscr{D}$ that can be used to solve P is at least as strong as $\mathscr{D}^*$, i.e., $\mathscr{D} \succeq \mathscr{D}^*$ (necessity) [14].

When a majority of processes are correct, the weakest failure detector for solving consensus is Ω [14]. When the number of process failures is not bounded, Ω is not sufficient anymore. Solving consensus in this context requires Ω for liveness and the quorum failure detector Σ for safety [24]. Informally, Σ outputs a set of trusted processes at each process such that: any two sets output at any times and by any process intersect, and eventually every set output at correct processes contain only correct processes.

In this thesis, we consider *realistic* failure detectors only, i.e., those that cannot predict the future [22]. Delporte *et al.* show in [22] that, when an arbitrary number of processes may fail, the weakest *realistic* failure detector for solving consensus cannot make any mistakes about the alive status of processes, i.e., it

may not stop trusting a process before it crashes. Additionally, it must eventually stop trusting all crashed processes. In the literature, this failure detector is denoted as the perfect failure detector $\mathscr{P}$. This result seems to contradict [24]. However, it does not: when we consider realistic failure detectors only and the number of process failures is not bounded, $\mathscr{P}$ and Σ are equivalent [23].

2.4 Database Definitions

A database $\mathscr{D} = \{x_1, .., x_n\}$ is a finite set of data items. Database sites have a partial copy of the database. For each site s_i (or process) in Π, $Items(s_i) \subseteq \mathscr{D}$ is defined as the set of data items replicated on s_i. A transaction T_i is a sequence of read and write operations on data items followed by a commit or abort operation. A read and a write of transaction T_i on some data item x are respectively denoted as $r_i[x]$ and $w_i[x]$; c_i and a_i respectively denote the commit and abort of T_i. For simplicity, we represent a transaction T as a tuple (id, rs, ws, up), where id is the unique identifier of T, rs is the readset of T, ws is the writeset of T and up contains the updates of T. More precisely, up is a set of tuples (x, v), where, for each data item x in ws, v is the value written to x by T. For every transaction T, $Items(T)$ is defined as the set of data items read or written by T. Two transactions T and T' are said to be conflicting if there exists a data item $x \in Items(T) \cap Items(T') \cap (T.ws \cup T'.ws)$. We define $Site(T)$ as the site on which T is executed. Furthermore, we assume that for every data item $x \in \mathscr{D}$, there exists a correct site s_i which replicates x, i.e., $x \in Items(s_i)$. Finally, we define $Replicas(T)$ as the set of sites that replicate at least one data item written by T, i.e., $Replicas(T) = \{s_i \mid s_i \in \Pi \wedge Items(s_i) \cap T.ws \neq \emptyset\}$.

Database replication protocols may ensure different data consistency criteria. In this thesis, we consider *one-copy serializability* (1-SR) [9]. Let H be a replicated data history consisting of committed transactions only. History H is 1-SR iff H is *view-equivalent* to some one-copy serial history $1H$, where H and $1H$ are view-equivalent iff the following holds:

1. H and $1H$ are defined over the same set of transactions,
2. H and $1H$ have the same *read-x-from* relationships on data items: $\forall T_i, T_j \in H$ (and hence, $T_i, T_j \in 1H$): T_j *read-x-from* T_i in H iff T_j *read-x-from* T_i in $1H$, and,
3. For each final write $w_i[x]$ in $1H$, $w_i[x_A]$ is also a final write in H for some copy x_A of x.

Chapter 3

FIFO and Causal Multicast

> First things first, but not necessarily in that order.
>
> *Doctor Who*

Multicast abstractions ensure a similar *reliability* guarantee—agreement on the set of messages delivered—but offer various message *ordering* properties. Two of these properties, FIFO and causal order, are of special interest: they ensure that a message m is not delivered at a process p that does not know m's *context*, where the notion of context is defined differently for each order property. With FIFO order, the context of m at p is the messages that were previously multicast by m's sender and addressed to p. Causal order extends the notion of context to all messages that causally precede m, i.e., messages that are causally linked to m through a chain of multicast and delivery events. FIFO and causal order help the programming of distributed applications in various domains such as global snapshot construction [6], fair resource allocation [37], and replicated data management [57].

FIFO and causal broadcast protocols have been extensively studied in the literature. In this chapter, we propose FIFO and causal multicast protocols for systems composed of a set of disjoint groups (e.g., server racks or data centers), each containing several processes. In particular, we show that mechanisms devised for FIFO and causal broadcast protocols are not applicable to multicast protocols. As our main contribution, we propose FIFO and causal multicast algorithms that offer several desirable properties. To the best of our knowledge, these algorithms are the first to be simultaneously *fast*, *scalable*, *flexible*, and *highly resilient*, in a precise sense, as we now explain.

First, they are *fast*: messages can be delivered in two communication steps; and we further show that this is optimal. Second, these protocols are *scalable*: (i) to deliver a message m only the sender and the addressees of m participate in the protocol, a property referred to as *genuineness*; and (ii) if n and m denote, respectively, the number of processes and groups in the system, messages only carry m counters to ensure FIFO order, and $n \times m$ counters to ensure causal order. Since n is within a constant factor of m, this is optimal for causal multicast [36]. Third, the algorithms are *flexible* in the sense that a process p may multicast messages to groups p does not belong to, that is, groups are "open". Finally, our algorithms are *highly resilient*: they tolerate an arbitrary number of process failures and can cope with quasi-reliable links.

This is in contrast to several multicast protocols [42; 44; 51; 53; 36], which depend on reliable links—message delivery is guaranteed as long as the receiver is correct, regardless of the correctness of the sender. Reliable links are not a realistic assumption: to send a message m, the machine M_p hosting process p typically inserts m into one (or more) local buffer before m is sent over the wire. Hence, even though p *thinks* that m was successfully sent, m may still be lost in case M_p crashes before m hits the wire.

3.1 Tolerating Quasi-reliable Networks

Devising multicast protocols that tolerate quasi-reliable links introduces difficulties that were not discussed elsewhere. Figure 3.1 illustrates the problem. Consider some process p that multicasts a message m_1 to some group g_2. Later, p multicasts a message m_2 to groups g_1 and g_2 and crashes. Message m_2 is received by processes in g_1, and since m_2 is the first message multicast from p to g_1, m_2 is delivered by processes in g_1. On the contrary, all messages sent from p to members of g_2 are lost. Note that this can happen because p crashes and links are quasi-reliable.

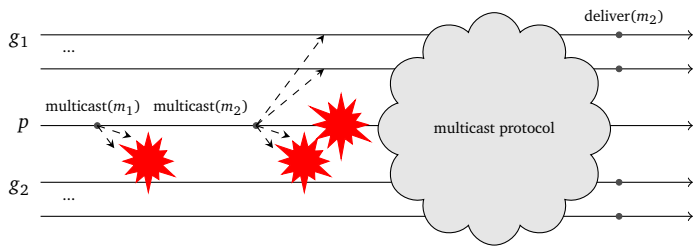


Figure 3.1. The message delivery order violation problem.

From the reliability guarantees of multicast, correct processes in g_2 must eventually deliver m_2. However, if they do so, the ordering guarantees of FIFO and causal multicast will be violated: members of g_2 cannot deliver m_1 before m_2 since m_1 was lost. If messages were broadcast, then m_1 would also be addressed to g_1, and thus, g_1 could help g_2 by forwarding m_1 to g_2. With multicast however, g_1 does not even know about the existence of m_1, since m_1 was not addressed to g_1. In this chapter, we propose a mechanism to cope with this problem despite an arbitrary number of process failures; the resulting FIFO and causal multicast algorithms are as latency-efficient as their broadcast counterparts.

3.2 Related Work

FIFO and causal broadcast were originally specified as part of the Isis system [10]. In [32], FIFO broadcast is implemented by reliably broadcasting messages along with a sequence number and by delivering messages in the sequence number order.

The first implementation of causal broadcast uses a simple strategy [10]: the causal history of delivered messages is piggybacked on each message to be broadcast. The amount of information contained in messages is thus unbounded. In [51], causal order is ensured differently: messages carry control information in the form of a matrix of counters, where each entry (p, q) denotes the number of messages that were multicast from process p to q in the causal history. This control information is used to know when messages can be safely delivered. This algorithm does not tolerate process failures. A fault-tolerant algorithm that ensures causal order using a similar technique appears in [32]. Although [32] specifies both causal broadcast and multicast, the algorithm given considers the broadcast case only.

In [11], processes may belong to several groups at the same time but messages sent from a process p cannot be multicast to groups p is not a member of. Using the terminology of [20], the protocol in [11] is closed-group. In this algorithm, each message carries a vector of counters, and this for every group in the system. Messages may be large if the number of groups is high. In contrast, [44] only requires processes to append a vector of counters to messages, where the size of the vector is equal to the number of groups. However, this protocol is not fault-tolerant. In [53], the topology of the underlying network is used to reduce the amount of information that must be appended to messages.

The necessary conditions on how much information should be stored at pro-

Algorithm	order	type	speed	scalability	flexibility	resilience	
			latency	requires message piggybacking?	open/closed group	processes	requires reliable network?
[32]	FIFO	broadcast	2	no	-	crash-stop	no
$\mathscr{A}_{fifo}$	FIFO	multicast	2	no	open	crash-stop	no
[42]	causal	unicast	1	no	-	no failures	yes
[32]	causal	broadcast	2	no	-	crash-stop	no
[44]	causal	multicast	1	no	closed	no failures	yes
[51]	causal	multicast	1	no	open	no failures	yes
[53]	causal	multicast	topology dependent	no	open	no failures	yes
[36][1]	causal	multicast	1	no	open	crash-stop	yes
[10]	causal	multicast	2	yes	closed	crash-stop	no
[11]	causal	multicast	2	no	closed	crash-stop	no
$\mathscr{A}_{causal}$	causal	multicast	2	no	open	crash-stop	no

Table 3.1. Comparison of the FIFO and causal multicast algorithms.

cesses and appended to messages to ensure causal order are presented in [36]. This paper also provides an information-optimal algorithm that does not need any a priori knowledge of the communication network. The algorithm in [42] does not append any information on messages but only considers the unicast case and postpones the sending of messages until after all the previous messages sent were acknowledged.

In this chapter, we present fault-tolerant and latency-optimal FIFO and causal multicast protocols, respectively denoted as $\mathscr{A}_{fifo}$ and $\mathscr{A}_{causal}$. To the best of our knowledge, these are the first algorithms that are at the same time open-group and tolerate quasi-reliable networks. As discussed above, and later in Section 3.3, implementing these primitives with a quasi-reliable communication medium raises a problem that was not discussed elsewhere.

Table 3.1 provides a comparison of the algorithms. The last five columns respectively indicate: the best-case latency of the algorithms, measured in the number of communication delays; whether the algorithms resort to message piggybacking or not; whether an algorithm $\mathscr{A}$ allows messages to be multicast from a process p to groups p does not belong to, in which case we say that $\mathscr{A}$ is an open-group algorithm, or not, in which case we say that $\mathscr{A}$ is a closed-group algorithm; and the process as well as network failure resilience.

[1]The algorithm in [36] tolerates process crashes and has a latency of 1 message delay. This does not contradict the lower bound of two message delays we show in this chapter. Indeed, two message delays is minimal for algorithms that tolerate quasi-reliable links. However, the algorithm in [36] requires reliable links.

3.3 FIFO Multicast

In this section, we present a genuine FIFO multicast algorithm that tolerates an arbitrary number of failures. This protocol is latency-optimal, as Section 3.5 shows.

In Algorithm $\mathscr{A}_{fifo}$, every message m is tagged with a sequence number, denoted as $m.seq$. Messages multicast by some process q are F-Delivered in the sequence number order. To do so, every process p keeps track of the next message F-MCast by q to be F-Delivered by p. This information is stored in a variable denoted as $nextFDel[q]_p$. So far, this is like the FIFO broadcast algorithm in [32]. We now explain how $\mathscr{A}_{fifo}$ differs from [32]. First, since messages may be addressed to a subset of the system's groups, messages do not carry a single sequence number, as in [32], but an array of sequence numbers, one for each group (see Algorithm $\mathscr{A}_{fifo}$, lines 5-9).

Algorithm $\mathscr{A}_{fifo}$
Genuine FIFO Multicast - Code of process p

1: **Initialization**
2: $nbCast[g] \leftarrow 0$, for each group g ▷ nb. of msgs. F-MCast to g
3: $nextFDel[q] \leftarrow 1$, for each process q ▷ next msg. F-MCast from q to be F-Delivered
4: $msgSet \leftarrow \emptyset$ ▷ set of messages received but not yet F-Delivered

5: **To F-MCast** message m *{Task 1}*
6: **foreach** $g \in m.dst$ **do**
7: $nbCast[g] \leftarrow nbCast[g] + 1$
8: $m.seq \leftarrow nbCast$
9: send(m) to $m.dst$

10: **When** receive(m) or receive(m, OK)
11: **if** $m \notin msgSet$ **then**
12: **if** $m.seq[group(p)] = nextFDel[m.sender]$ **then**
13: send(m, OK) to $m.dst$
14: **else**
15: send(m) to $m.dst$
16: $msgSet \leftarrow msgSet \cup \{m\}$

17: **When** $\exists m \in msgSet$: $\forall g \in m.dst$: received (m, OK) from all processes in $\Theta_g \wedge$
$m.seq[group(p)] = nextFDel[m.sender]$
18: F-Deliver(m)
19: $nextFDel[m.sender] \leftarrow nextFDel[m.sender] + 1$
20: **if** $\exists m' \in msgSet$: $m'.seq[group(p)] = nextFDel[m'.sender]$ **then**
21: send(m', OK) to $m'.dst$

Second, recall the aforementioned problematic scenario specific to multicast:

some process p F-MCasts a local message m_l to some group g_2; later, p F-MCasts a global message m_g to groups g_1 and g_2 and crashes. Message m_g is received by processes in g_1, and since m_g is the first message multicasts from p to g_1, m_g is delivered by processes in g_1. On the contrary, all messages sent from p to members of g_2 are lost. Note that this can happen because p crashes and links are quasi-reliable. From the uniform agreement property of FIFO multicast, correct processes in g_2 must eventually deliver m_g. However, if they deliver m_g, the FIFO order property of FIFO multicast will be violated: members of g_2 cannot deliver m_l before m_g since m_l was lost.

To solve this problem, before F-Delivering a message m, a process $p \in m.dst$ announces the addressees of m that it F-Delivered all messages $m.sender$ previously F-MCast. To do so, p sends m's addressees an *OK* message (lines 13 or 21). Message m is then F-Delivered by p when p received an *OK* message from at least one correct process of every correct destination group of m. This is implemented by relying on failure detector Θ [4]. At each process, this oracle outputs a list of processes that are trusted to be alive such that:

- *Completeness:* There is a time after which processes do not trust any process that crashes.
- *Accuracy:* If some process never crashes then, at every time, every process trusts at least one correct process that never crashes.

To ensure that p received an *OK* message from at least one correct process of every correct destination group g of m, for every such group g, p waits to receive an *OK* message from all processes trusted by Θ_g, i.e., the failure detector Θ whose output is restricted to members of g (line 17).

This mechanism is also used to ensure uniform agreement: if there exists a correct addressee of m, when p received an *OK* message from all processes trusted by Θ_g, and this for every group g in $m.dst$, process p knows m was received by at least one correct addressee of m. Hence, all correct processes in $m.dst$ will eventually receive m.

When a group g contains fewer than a majority of faulty processes, Θ_g can easily be implemented as follows: a process p that wishes to query Θ_g periodically receives *heartbeat* messages from members of g along with a monotonically increasing sequence number; the i-th output of Θ_g at p is any majority of *heartbeat* messages with sequence number i. In the context of Algorithm $\mathcal{A}_{fifo}$, Θ_g can be implemented without adding extra messages: when p queries Θ_g at line 17, p waits to receive a majority of *OK* messages from processes in g. We note, however, that when the universe of failure detectors is restricted to realistic ones

and the number of process failures is not bounded, Θ is equivalent to the perfect failure detector $\mathscr{P}$ [23].

3.4 Causal Multicast

We now present the first open-group causal multicast algorithm that tolerates quasi-reliable communication links. This algorithm tolerates an arbitrary number of failures and is latency-optimal (c.f. Section 3.5).

The causal multicast algorithm $\mathscr{A}_{causal}$ relies on FIFO multicast and is blocking, that is, processes may delay the delivery of a message m for a later time even though all the protocol messages to deliver m have been received.

In Algorithm $\mathscr{A}_{causal}$, every process p keeps track of how many messages, multicast by some process q, it has C-Delivered. This bookkeeping is done for every process q of the system. At p, this information is stored in a vector denoted as $nbDel_p$, indexed by process id. This is like in the causal broadcast algorithm in [32]. In this algorithm, to broadcast a message m, p F-BCasts m along with $nbDel_p$. Upon F-Delivering m, p inserts m in a list of messages $msgLst_p$ and C-Delivers m as soon as it is the first message in $msgLst_p$ such that $nbDel_p \geq m.nbDel$.[2] It is not hard to see why this algorithm works: since $m.nbDel[q]$ denotes the number of messages originating from q that causally precede the multicast of m, C-Delivering m when it is the first message in $msgLst_p$ such that $nbDel_p \geq m.nbDel$, ensures that causal order will not be violated.

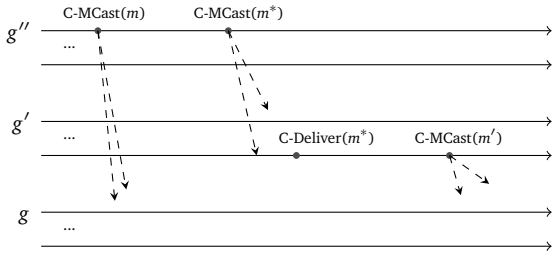


Figure 3.2. A causal relation between m and m' that is blind for g.

In the multicast case, however, this algorithm does not work. Consider the following causal relation between two messages m and m', C-MCast(m) $\rightarrow$ C-MCast(m'), both addressed to some group g that is denoted as *blind* for g. Figure 3.2 illustrates the scenario. Messages m and m' are such that the causal

[2]Given any two vectors v_1 and v_2, we write $v_1 \geq v_2$ instead of $\forall q \in \Pi : v_1[q] \geq v_2[q]$ for simplicity.

chain linking the events C-MCast(m) and C-MCast(m') does not contain any events of type C-Deliver of some message addressed to g, and let m' be C-MCast by a process different from $m.sender$. Intuitively, this causal relation is problematic because processes in g may C-Deliver m and m' in different orders. Indeed, since the causal chain linking the events C-MCast(m) and C-MCast(m') does not contain any events of type C-Deliver of some message addressed to g, it is impossible to distinguish m from m' by only comparing the number of messages addressed to g that were C-Delivered in the causal history of events C-MCast(m) and C-MCast(m').[3]

Algorithm $\mathscr{A}_{causal}$
Genuine Causal Multicast - Code of process p

1: **Initialization**
2: $nbCast[g][q] \leftarrow 0$, for each group g and process q ▷ nb. msgs. q C-MCast to g in causal history
3: $nbDel[q] \leftarrow 0$, for each process q ▷ nb. msgs. q C-MCast that p C-Delivered
4: $msgLst \leftarrow \epsilon$ ▷ list of messages F-Delivered but not yet C-Delivered

5: **To C-MCast** message m *{Task 1}*
6: **foreach** $g \in m.dst$ **do**
7: $nbCast[g][p] \leftarrow nbCast[g][p] + 1$
8: $m.nbCast \leftarrow nbCast$
9: F-MCast (m) to $m.dst$

10: **Function** IsDeliverable(m)
11: return $\forall q \in \Pi \setminus \{m.sender\} : m.nbCast[group(p)][q] \leq nbDel[q]$

12: **When** F-Deliver(m)
13: $msgLst \leftarrow msgLst \oplus m$ ▷ add m at the tail of $msgLst$
14: **while** $\exists m' \in msgLst$: IsDeliverable(m')
15: Let m' be the first message in $msgLst$ s.t. IsDeliverable(m')
16: C-Deliver(m')
17: $nbDel[m'.sender] \leftarrow m'.nbCast[group(p)][m'.sender]$
18: **foreach** $g \in \Gamma$ **do**
19: $nbCast[g] \leftarrow \max(m'.nbCast[g], nbCast[g])$
20: $msgLst \leftarrow msgLst \ominus m'$

To be able to distinguish messages m and m' in the example above, processes keep track of the number of messages C-MCast in the causal history instead of the number of C-Delivered messages. This accounting is done on a group basis.

Hence, in addition to maintaining vector $nbDel$, each process p keeps track of the number of messages addressed to any group g, originating from any pro-

[3] An event e is in the causal history of an event e' iff $e \rightarrow e'$.

cess q, that were C-MCast in its causal history, denoted as $nbCast[g][q]_p$. This variable is piggybacked on every C-MCast message m. Message m is then C-Delivered at p as soon as it is the first message in $msgLst_p$ such that for all processes q different from $m.sender$, $m.nbCast[group(p)][q] \leq nbDel[q]$, i.e., p C-Delivered all messages addressed to $group(p)$ that were C-MCast in the causal history of event C-MCast(m). The delivery condition does not involve $m.sender$ since FIFO multicast ensures that messages multicast by the same process will be delivered in the order they were multicast.

We now present the causal multicast algorithm $\mathscr{A}_{causal}$ in detail. To C-MCast a message m, for any group $g \in m.dst$, p increments $nbCast[g][p]_p$ and F-MCasts m along with the $nbCast$ variable (lines 6-9). As soon as some process q F-Delivers this message, q adds m at the end of $msgLst$ (line 13) and checks whether a message can be C-Delivered (line 14). If it is the case, the first C-Deliverable message of $msgLst_p$, m', is C-Delivered. Before removing m' from $msgLst$, $nbDel[m'.sender]$ is updated and for all group g and processes q of the system, $nbCast[g][q]$ is set to the maximum between $m'.nbCast[g][q]$ and $nbCast[g][q]$ so that $nbCast[g][q]$ represents the number of messages originating from q and addressed to g that were C-MCast in the causal history of C-MCast(m') (line 19).

3.5 Latency Optimality

We show that for any message m there exists no uniform reliable multicast algorithm $\mathscr{A}$ that tolerates quasi-reliable links and delivers m in one message delay, whatever the destination groups of m are. This result is independent of the genuineness of $\mathscr{A}$ and shows the optimality of our uniform FIFO and causal multicast algorithms. Indeed, if it were not the case we could get a more efficient uniform reliable multicast algorithm by reducing it to causal or FIFO multicast, a contradiction. Moreover, this result also applies to uniform reliable broadcast. To see why, suppose there would exist a uniform reliable broadcast algorithm $\mathscr{A}_{urb}$ that could deliver messages in one message delay. We could then devise a non-genuine uniform reliable multicast algorithm that could deliver messages in one message delay by relying on $\mathscr{A}_{urb}$, a contradiction.

We show this result in the synchronous round-based model which we briefly recall now (see Chapter 2 in [40] for a formal description). Each process p has a buffer, $buffer_p$, that represents the set of messages that have been sent to p but not yet received; p receives the message when it removes it from its buffer. In any run of an algorithm, until it crashes, each process p repeatedly performs the

following two steps, which define one round:

1. In the first step, p generates the (possibly $null$) messages to be sent to each process based on its current state, and puts these messages in the appropriate process buffers. If p crashes in round r, only a subset of the messages created in r by p are put in the buffers.

2. In the second step, p determines its new state based on its current state and on the messages received, and removes all messages from its buffer.

Proposition 3.5.1 *In any system with $n \geq 3$, $f \geq 2$, and quasi-reliable links, for any uniform reliable multicast algorithm $\mathscr{A}$ and any message m addressed to at least two processes, there does not exist a run R of $\mathscr{A}$ in which m is R-MCast in some round r and R-Delivered by some process q at the end of r.*

Proof: Suppose, by way of contradiction, that such an algorithm $\mathscr{A}$ and run R of $\mathscr{A}$ exist. In some round r of run R, some process p R-MCasts m and q R-Delivers m at the end of round r. We build a run R' that is indistinguishable from R to q up to and including round r. In R', p crashes in r and m is only received by q. Moreover, q crashes just after R-Delivering m. Hence, in run R', no correct process in $m.dst$ R-Delivers m, violating the uniform agreement property of $\mathscr{A}$. □

3.6 Discussion

FIFO and causal multicast ensure that a message m is not delivered at a process p that does not know m's *context*, where the notion of context is defined differently for each order property. With FIFO order, the context of m at p is the messages that were previously multicast by m's sender and addressed to p. Causal order extends the notion of context to all messages that causally precede m, i.e., messages that are causally linked to m through a chain of multicast and delivery events. These communication primitives are used in domains such as global snapshot construction [6] and fair resource allocation [37]. Moreover, we demonstrate in Chapter 4 how causal multicast can be used to implement atomic multicast.

In this chapter, we showed that implementing these primitives in quasi-reliable networks presents a challenge that did not appear elsewhere. We proposed FIFO and causal multicast algorithms that are open-group and tolerate an arbitrary number of process failures. These protocols are as latency-efficient as their broadcast counterpart. In fact, they are latency-optimal.

3.7 Proofs of Correctness

In the proofs below, we denote the value of a variable V on a process p at time t as V_p^t. Furthermore, for events of the type C-MCast and C-Deliver, we sometimes add a subscript to denote on which process this event occurred.

3.7.1 The Proof of Algorithm $\mathscr{A}_{fifo}$

Proposition 3.7.1 (Uniform Integrity) *For any process p and any message m, (a) p F-Delivers m at most once, and (b) only if $p \in m.dst$ and (c) m was previously F-MCast.*

Proof:

- (a) After p F-Delivers m, p increments $nextFDel[m.sender]$. Thus, the condition of line 17 can never evaluate to true for m anymore.
- (b) Follows directly from the uniform integrity property of links and the algorithm.
- (c) Process p F-Delivers m only if p received m. From the uniform integrity property of links, m was sent by some process. Consequently, m was F-MCast. □

Proposition 3.7.2 Uniform FIFO Order *If a process p F-MCasts a message m before F-MCasting a message m', then no process in $m.dst \cap m'.dst$ F-Delivers m' unless it has previously F-Delivered m.*

Proof: Let q be any process in $m.dst \cap m'.dst$ that F-Delivers m', we show that q F-Delivers m before. If q F-Delivers m', then there is a time t before q F-Delivers m' at which $nextFDel[m.sender]_q^t = m'.seq[group(q)]$. From the definition of m, $m.seq[group(q)] < m'.seq[group(q)]$. From lines 17-19, q must have F-Delivered m before t, and thus before q F-Delivers m'. □

Definition 3.7.1 *We define the binary relation $pred$ on messages as follows, m_1 $pred$ m_2 iff:*

1. *$m_1.sender = m_2.sender$,*
2. *$m_1.sender$ F-MCasts m_1 before m_2, and*
3. *There exists at least one correct process in $m_1.dst \cap m_2.dst$*

Moreover, let $\mathcal{G}_{pred(m)} = (V, E)$ be a finite DAG constructed as follows:

1. *add vertex m to V*
2. *while $\exists m_1 \in V : \exists m_2 \notin V : m_2\ pred\ m_1$ do:*
   add m_2 to V and add directed edge $m_2 \rightarrow m_1$ to E

For any message m' in $\mathcal{G}_{pred(m)}$, we say that m' is at distance k of m iff the longest path from m' to m is of length k. We let $\mathcal{M}_k$ be the subset of messages in $\mathcal{G}_{pred(m)}$ that are at distance k of m.

Lemma 3.7.1 *For any message m, if for all messages m' in $\mathcal{G}_{pred(m)}$ all correct processes in $m'.dst$ receive m', then all correct processes $p \in m.dst$ eventually F-Deliver m.*

Proof: Assume that for all messages m' in $\mathcal{G}_{pred(m)}$ all correct processes in $m'.dst$ receive m'. We prove that, for any $k \geq 0$, all messages in $\mathcal{M}_k$ are eventually F-Delivered by all their correct addressees. Since $\mathcal{M}_0 = \{m\}$, this shows the claim. Let x be the largest integer such that $\mathcal{M}_x \neq \emptyset$. We proceed by induction on k, starting from $k = x$.

- Base step ($k = x$): Let m_x be any message in $\mathcal{M}_x$ and q be any correct process in $m_x.dst$. From the definition of x, (*) there exists no message m_{x+1} such that $m_{x+1}\ pred\ m_x$. Since for all messages m' in $\mathcal{G}_{pred(m)}$, all correct processes in $m'.dst$ eventually receive m', q eventually receives m_x. By (*), m_x is the first message F-MCast by $m.sender$ such that $q \in m_x.dst$, and hence, $m_x.seq[group(q)] = 1$. Therefore, all correct processes in $m_x.dst$ eventually send(m_x, OK) and by the reliability property of links, q eventually receives these messages. By the completeness property of Θ, there exists a time after which q does not trust any process that crashes. Hence, by the condition of line 17, q eventually F-Delivers m_x.
- Induction step: Suppose the claim holds for k ($0 < k \leq x$), we show it holds for $k - 1$. Let m_{k-1} be any message in $\mathcal{M}_{k-1}$, g be any correct group in $m_{k-1}.dst$, and q be any correct process in g. We first show that (*) there exists a time t at which $nextFDel[m.sender]_q^t = m_{k-1}.seq[group(q)]$. Either (a) m_{k-1} is the first message F-MCast to g or (b) not.
  - In case (a), $m_{k-1}.seq[g] = 1$. Since $nextFDel[m.sender]$ is initialized to 1, (*) holds.

– In case (b), there exists a message $m_{k'}$ in $\mathscr{M}_{k'}$ ($x \geq k' > k-1$) such that $m_{k'}\ pred\ m_{k-1}$, $g \in m_{k'}.dst \cap m_{k-1}.dst$, and $m_{k'}.seq[g] = m_{k-1}.seq[g]-1$. By the induction hypothesis, q F-Delivers $m_{k'}$. Therefore, (*) holds.

From the algorithm, since for all messages m' in $\mathscr{G}_{pred(m)}$, all correct processes in $m'.dst$ eventually receive m', all correct processes r in $m_{k-1}.dst$ eventually receive m_{k-1}. Consequently, from (*), all r send (m_{k-1}, OK), either at line 13 or at line 21. By the quasi-reliability property of links, q eventually receive these messages. By the completeness property of Θ, there exists a time after which q does not trust any process that crashes. Consequently, by the condition of line 17, q eventually F-Delivers m_{k-1}. □

Lemma 3.7.2 *For any message m and any process p, if p sends (m, OK), then p F-Delivered all messages m' such that $p \in m'.dst$ and $m.sender$ F-MCast m' before m.*

Proof: If m is the first message $m.sender$ F-MCasts to $group(p)$, the claim holds trivially. Otherwise, let m_x be the message such that $p \in m_x.dst$ and $m.sender$ F-MCasts m_x just before m. Since p sends (m, OK), there exists a time t at which $nextFDel[m.sender]_p^t = m.seq[group(p)]$. From lines 17-19, p must have F-Delivered m_x before t. By applying Proposition 3.7.2 multiple times, before t, p also F-Delivered all messages addressed to $group(p)$ that $m.sender$ F-MCast before m_x. □

Proposition 3.7.3 (Uniform Agreement) *If a process p F-Delivers a message m, then all correct processes $q \in m.dst$ eventually F-Deliver m.*

Proof: Let $\mathscr{M}_k$ be the subset of messages in $\mathscr{G}_{pred(m)}$ that are at distance k of m. We first show that, for any $k \geq 0$ and any message m' in $\mathscr{M}_k$ such that $\mathscr{M}_k \neq \emptyset$: (1) m' is received by all correct processes in $m'.dst$ and (2) for each correct group $g \in m'.dst$, there is a correct process q in g that sends (m', OK). We proceed by simultaneous induction on (1) and (2).

- Base step ($k = 0$):

  – (1) Since p F-Delivers m, from the condition of line 17, p received an (m, OK) message from all processes trusted by Θ_g, and this for every group $g \in m.dst$. If there are no correct processes in $m.dst$, then the base step of (1) holds trivially. Otherwise, by the accuracy property

of Θ, p received an (m, OK) message from a correct addressee q of m. Since q is correct, by the quasi-reliability property of links, every correct process in $m.dst$ eventually receives m from q.

- (2) Since p F-Delivers m, from the condition of line 17, for every group g in $m.dst$, p received a message (m, OK) from all processes trusted by Θ_g. By the accuracy property of Θ, for all correct group $g \in m.dst$, p received message (m, OK) from a correct process q in g. Hence, by the uniform integrity property of links, q sent (m, OK).

- Induction step: Suppose that (1) and (2) hold for $k-1$ ($k > 0$), we show that (1) and (2) also hold for k. Let m_{k-1} be any message in $\mathscr{M}_{k-1}$ and let m_k be any message in $\mathscr{M}_k$ such that $m_k \; pred \; m_{k-1}$.

  - (1) Because $k > 0$, from the definition of m_k and the definition of the *pred* relation, $m_k.sender$ F-MCasts m_k before m_{k-1} and there exists a correct process in $m_k.dst \cap m_{k-1}.dst$. By the induction hypothesis, for each group g in m_{k-1} containing at least one correct process, there exists a correct process q in g that sends (m_{k-1}, OK). Hence, there exists a correct process q in $m_{k-1}.dst \cap m_k.dst$ such that q sends (m_{k-1}, OK). By Lemma 3.7.2, q F-Delivered m_k. If there are no correct processes in $m_k.dst$, then the induction step of (1) holds trivially. Otherwise, from the condition of line 17, q received an (m_k, OK) message from all processes trusted by Θ_g, and this for every group $g \in m_k.dst$. Hence, from the accuracy property of Θ, q received (m_k, OK) from a correct process $r \in m_k.dst$. Therefore, by the quasi-reliability property of links, every correct process in $m_k.dst$ eventually receives m_k from r.

  - (2) From the definition of m_k and the definition of the *pred* relation, $m_k.sender$ F-MCasts m_k before m_{k-1} and there exists a correct process in $m_k.dst \cap m_{k-1}.dst$. By the induction hypothesis, there exists a correct process $r \in m_k.dst \cap m_{k-1}.dst$ such that r sends (m_{k-1}, OK). By Lemma 3.7.2, r F-Delivered m_k. From the condition of line 17, r received an (m_k, OK) message from all processes trusted by Θ_g, and this for every group $g \in m_k.dst$. Hence, by the accuracy property of Θ, for all correct group $g \in m_k.dst$, r received (m_k, OK) from a correct process q in g. Therefore, By the uniform integrity property of links, q sent (m_k, OK).

Hence, from (1), all messages $m' \in \mathscr{G}_{pred(m)}$ are received by all correct processes in $m'.dst$. Therefore, by Lemma 3.7.1, all correct processes in $m.dst$ F-

Deliver m. □

Proposition 3.7.4 (Validity) *If a correct process p F-MCasts a message m, then eventually all correct processes $q \in m.dst$ F-Deliver m.*

Proof: Since p is correct, by the quasi-reliability property of links, for all messages $m' \in \mathscr{G}_{pred(m)}$, all correct processes in $m'.dst$ receive m. By Lemma 3.7.1, all correct processes $q \in m.dst$ eventually F-Deliver m. □

3.7.2 The Proof of Algorithm $\mathscr{A}_{causal}$

Proposition 3.7.5 (Uniform Integrity) *For any process p and any message m, (a) p C-Delivers m at most once, and (b) only if $p \in m.dst$ and (c) m was previously C-MCast.*

Proof:

- (a) Follows directly from the uniform integrity property of FIFO multicast and from the fact that a message is removed from $msgLst_p$ after it is C-Delivered.
- (b) Follows directly from the algorithm.
- (c) Process p C-Delivers m only if p F-Delivered m. From the uniform integrity property of FIFO multicast, m was F-MCast. Consequently, m was C-MCast. □

Lemma 3.7.3 *For any message m such that $m.nbDel$ is defined, any group g, and any integer k,$m.nbCast[g][m.sender] = k$ iff m is the k-th message $m.sender$ C-MCasts to g.*

Proof:

- ($\Rightarrow$): From the algorithm, $m.sender$ increments $nbCast[g][m.sender]_{m.sender}$ at line 7 only ($m.sender$ does not update $nbCast[g][m.sender]_{m.sender}$ at line 19). Moreover, $m.sender$ does so before every message C-MCast to g. Therefore, since $nbCast[g][m.sender]$ is initialized to 0, m is the k-th message $m.sender$ C-MCasts to g.
- ($\Leftarrow$): The same argument as in ($\Rightarrow$) is used to show that $m.nbCast[g][m.sender] = k$. □

Lemma 3.7.4 *For any two messages m and m′ such that m.nbCast and m′.nbCast are defined, and any group g, if C-MCast(m) → C-MCast(m′), then m.nbCast*$[g] \leq$ *m′.nbCast*$[g]$.

Proof: From the definition of the causal precedence relation, it is easy to see that there exist processes p_1, p_2, .., p_k and messages m_1, m_2, .., $m_k = m'$ ($k \geq 2$) such that:

- $p_1 = m.sender$
- p_i C-MCasts m_i for all $1 \leq i \leq k$
- either (a) $m = m_1$ or (b) p_1 C-MCasts m before m_1 and
- p_i C-Delivers m_{i-1} before it C-MCasts m_i, for all $2 \leq i \leq k$
- In case (a), we show that for any $1 \leq i < k$, $m_i.nbCast[g] \leq m_{i+1}.nbCast[g]$. Process p_{i+1} C-Delivers m_i before C-MCasting m_{i+1}. Thus, from line 19 and because for any process q $nbCast[g][q]_{p_{i+1}}$ is monotonically increasing with time, $m_i.nbCast[g] \leq m_{i+1}.nbCast[g]$.
- In case (b), since for any process q $nbCast[g][q]_{p_1}$ is monotonically increasing with time, $m.nbCast[g][q] \leq m_1.nbCast[g][q]$. To conclude the proof, the same argument as in (a) is used to show that for any $1 \leq i < k$, $m_i.nbCast[g] \leq m_{i+1}.nbCast[g]$.

Lemma 3.7.5 *For any processes p and q, any integer k, and any time t at which p evaluates the condition of line 11,* $nbDel[group(p)][q]_p^t = k$ *iff before t, p C-Delivered the first k messages q C-MCasts to group*(p).

Proof:

- ($\Rightarrow$) : We proceed by induction on k.
  - Base step ($k = 0$): Since $nbDel[group(p)][q]_p$ is initialized to zero and monotonically increasing with time, if at the time t at which p evaluates line 11, $nbDel[group(p)][q]_p^t = 0$ then p did not C-Deliver any message C-MCast from q before t.
  - Induction step: Suppose that the claim holds for any l such that $0 \leq l \leq k-1$, we show that it also holds for k. Process p sets $nbDel[group(p)][q]_p$ to k just after C-Delivering a message m_k originating from q such that $m.nbCast[group(p)][q] = k$. From

Lemma 3.7.3, m_k is the k-th message q C-MCasts to $group(p)$. Let t' be the latest time before t at which p evaluates line 11 such that $nbDel[group(p)][q]_p^{t'} \neq nbDel[group(p)][q]_p^{t}$. We show that (*) $nbDel[group(p)][q]_p^{t'} = k-1$.

Suppose, by way of contradiction, that $nbDel[group(p)][q]_p^{t'} \neq k-1$. Let k' be the value of $nbDel[group(p)][q]_p^{t'}$. Either (a) $k' < k-1$ or (b) $k' > k-1$. We show that both (a) and (b) lead to a contradiction.

In case (a), from the induction hypothesis, before t' p C-Delivered the first k' messages C-MCast from q. Since $k' < k-1$, either (a-i), p does not C-Deliver the k-1-th message m_{k-1} C-MCast from q or (a-ii) p C-Delivers m_{k-1} after m_k.

- In case (a-i), since p C-Delivers m_k, from the uniform FIFO order property of FIFO multicast, p F-Delivers and inserts m_{k-1} before m_k in $msgLst_p$. From Lemma 3.7.4, $m_{k-1}.nbCast[group(p)][q] \leq m_k.nbCast[group(p)][q]$. From the condition of line 11 and since $nbDel[group(p)][q]_p$ is monotonically increasing with time, p must have C-Delivered m_{k-1} before m_k, a contradiction to the fact that p does not C-Deliver m_{k-1}.
- In case (a-ii), the same argument as in (a-i) is used to obtain a contradiction.

In case (b), since $k' \neq k$ and $k' > k-1$, $k' > k$. Process p sets $nbDel[group(p)][q]_p$ to k' just after C-Delivering a message $m_{k'}$ originating from q such that $m.nbCast[group(p)][q] = k'$. From Lemma 3.7.3, $m_{k'}$ is the k'-th message q C-MCasts to $group(p)$. Since $t' < t$, (**) p C-Delivers $m_{k'}$ before m_k. Since $k < k'$, from the uniform FIFO order property of FIFO multicast, p F-Delivers and inserts m_k before $m_{k'}$ in $msgLst_p$. From Lemma 3.7.4, $m_k.nbCast[group(p)][q] \leq m_{k'}.nbCast[group(p)][q]$. From the condition of line 11 and since $nbDel[group(p)][q]_p$ is monotonically increasing with time, p C-Delivers m_k before $m_{k'}$, a contradiction to (**).

From (*), $nbDel[group(p)][q]_p^{t'} = k-1$. From the induction hypothesis, before t' p C-Delivered the first $k-1$ messages C-MCast from q. Therefore, before t, p C-Delivered the first k messages C-MCast from q.

- ($\Leftarrow$): Either (a) $k = 0$ or (b) $k > 0$.

- In case (a), if p did not C-Deliver any message C-MCast from q, since $nbDel[group(p)][q]_p$ is initialized to zero, then $nbDel[group(p)][q]_p^t = 0$.
- In case (b), let m_k be the k-th message C-MCast from q that p C-Delivers. We show that (*) the last message C-MCast from q that p C-Delivers before t is m_k. Suppose, by way of contradiction, that the last message $m_{k'}$ C-MCast from q that p C-Delivers before t is not m_k. If before t, p C-Delivers the first k messages C-MCast from q, then m'_k is the k'-th message C-MCast from q to $group(p)$ such that $k' < k$. From the uniform FIFO order property of FIFO multicast, p F-Delivers and inserts $m_{k'}$ before m_k in $msgLst_p$. From Lemma 3.7.4, $m_{k'}.nbCast[group(p)][q] \leq m_k.nbCast[group(p)][q]$. Since p C-Delivers m_k, from the condition of line 11 and since $nbDel[group(p)][q]_p$ is monotonically increasing with time, p C-Delivers $m_{k'}$ before m_k. Since p C-Delivers m_k before t, $m_{k'}$ is not the last message C-MCast from q that p C-Delivers before t, a contradiction.

  From Lemma 3.7.3, m_k is such that $m_k.nbCast[group(p)][q] = k$. Therefore, from (*) and line 17, $nbDel[group(p)][q]_p^t = k$. □

Proposition 3.7.6 Uniform Causal Order *For any messages m and m', if C-MCast(m) → C-MCast(m'), then no process $p \in m.dst \cap m'.dst$ C-Delivers m' unless it has previously C-Delivered m.*

Proof: Let q be any process in $m.dst \cap m'.dst$ that C-Delivers m', we show that q C-Delivered m before. Either (a) $m.sender = m'.sender$ or (b) not.

- In case (a), since q C-Delivers m', q F-Delivers m'. From the uniform FIFO order property of FIFO multicast, (*) q F-Delivers m before F-Delivering m'. Either (a-i) there exists a time at which m and m' are in $msgLst_q$ or (a-ii) not.
  - In case (a-i), from (*) m can only appear before m' in $msgLst_q$. From Lemma 3.7.4, $m.nbCast[group(q)] \leq m'.nbCast[group(q)]$. Since at line 15, processes C-Deliver the first message in $msgLst$ such that the condition of line 11 is satisfied, q C-Delivers m before m'.
  - In case (a-ii), from (*) m is inserted in $msgLst_q$ before m'. Since processes remove messages from $msgLst_q$ only after C-Delivering them (line 20), q C-Delivers m before m'.

- In case (b), from Lemma 3.7.4, $m.nbCast[group(q)] \leq m'.nbCast[group(q)]$. From the condition of line 11, there exists a time t before q C-Delivers m' at which q evaluates line 11 such that for any process $r \neq m'.sender$ $nbDel[r]_q^t \geq m'.nbCast[group(q)][r]$. Hence, since $m.sender \neq m'.sender$, $nbDel[m.sender]_q^t \geq m.nbCast[group(q)][m.sender]$. Let k_1 and k_2 be $nbDel[m.sender]_q^t$ and $m.nbCast[group(q)][m.sender]$ respectively. From Lemma 3.7.3, m' is the k_2-th message $m.sender$ C-MCasts to $group(q)$. From Lemma 3.7.5, before t, q C-Delivers the first k_1 messages $m.sender$ C-MCasts to $group(q)$. Therefore, since $k_1 \geq k_2$, q C-Delivers m before m'. □

Definition 3.7.2 *Let m be a message, we define the finite DAG $\mathcal{G}_{pred(m)} = (V, E)$ as follows:*

1. *add vertex m to V*
2. *while $\exists m_1, m_2$ s.t. $m_1 \in V \wedge$ C-MCast$(m_2) \rightarrow$ C-MCast$(m_1) \wedge m_1.dst \cap m_2.dst \neq \emptyset$ do:*
   add m_2 to V and add directed edge $m_2 \rightarrow m_1$ to E

For any message m' in $\mathcal{G}_{pred(m)}$, we say that m' is at distance k of m iff the longest path from m' to m is of length k. We let $\mathcal{M}_k$ be the subset of messages in $\mathcal{G}_{pred(m)}$ that are at distance k of m.

Lemma 3.7.6 *For any message m, if for all messages $m' \in \mathcal{G}_{pred(m)}$ all correct processes in $m'.dst$ F-Deliver m', then all correct processes in m.dst eventually C-Deliver m.*

Proof: Assume that for all messages m' in $\mathcal{G}_{pred(m)}$ all correct processes in $m'.dst$ F-Deliver m'. We prove that, for any $k \geq 0$, all messages in $\mathcal{M}_k$ are eventually C-Delivered by their correct addressees. Since $\mathcal{M}_0 = \{m\}$, this shows the claim. Let x be the largest integer such that $\mathcal{M}_x \neq \emptyset$. We proceed by induction on k, starting from $k = x$.

- Base step ($k = x$): Let m_x be any message in $\mathcal{M}_x$ and q be any correct process in $m_x.dst$. From the definition of m_x, (*) there exists no message m' such that C-MCast$(m') \rightarrow$ C-MCast(m_x) and $m'.dst \cap m_x.dst \neq \emptyset$. Let g be any group in $m_x.dst$ and let r be any process different from $m_x.sender$. From (*), $m_x.sender$ never updated $nbCast[g][r]_{m_x.sender}$ at line 19. Therefore, $m.nbCast[g][r] = 0$. From the algorithm, $nbDel[g][r]_q$ is monotonically increasing with time and hence $nbDel[g][r]_q \geq m.nbCast[g][r]$ is

always true. Since all correct processes in $m_x.dst$ eventually F-Deliver m_x, from the condition of line 11, all correct processes in $m_x.dst$ eventually C-Deliver m_x.

- Induction step: Suppose that for any l such that $x \geq l > k \geq 0$ the claim holds, we show the claim holds for k. Let m_k be any message in $\mathscr{M}_k$ and q be any correct process in $m_k.dst$ (if there exists no correct process in $m_k.dst$, then the claim holds trivially). We show that (*) for any process r different from $m_k.sender$ there exists a time t at which $nbDel[group(q)][r]_q^t \geq m_k.nbCast[group(q)][r]$.

  If $m_k.nbCast[group(q)][r] = 0$, then (*) holds trivially. Otherwise, from line 19, there exists a message m_r such that $m_r.sender = r$, $group(q) \in m_r.dst$, C-MCast(m_r) $\rightarrow$ C-MCast(m_k), and $m_r.nbCast[group(q)][r] = m_k.nbCast[group(q)][r]$. From the induction hypothesis, q eventually C-Delivers m_r and sets $nbDel[group(q)][r]_q$ to $m_r.nbCast[group(q)][r]$. Since $m_r.nbCast[group(q)][r] = m_k.nbCast[group(q)][r]$, (*) holds.

  Since all correct processes in $m_k.dst$ eventually F-Deliver m_k, from (*) and since $nbDel[group(q)][r]_q$ is monotonically increasing with time, from the condition of line 11, all correct processes in $m_k.dst$ eventually C-Deliver m_k. □

Proposition 3.7.7 (Uniform Agreement) *If a process p C-Delivers a message m, then all correct processes $q \in m.dst$ eventually C-Deliver m.*

Proof: Let $\mathscr{M}_k$ be the subset of messages in $\mathscr{G}_{pred(m)}$ that are at distance k of m. We first show that (*) for any k, all messages in $\mathscr{M}_k$ are eventually F-Delivered by all of their correct addressees.

- Base step ($k = 0$): Since p C-Delivers m, p F-Delivers m. From the uniform agreement property of FIFO multicast, all correct processes in $m.dst$ eventually F-Deliver m.

- Induction step: Suppose the claim holds for any l such that $k \geq l \geq 0$, we show that the claims holds for $k+1$. Let m_{k+1} be any message in $\mathscr{M}_{k+1}$ and g be any correct group in $m_{k+1}.dst$. Furthermore, let k' be the largest integer smaller than $k+1$ such that there exists a message $m_{k'}$ in $\mathscr{M}_{k'}$, $g \in m_{k'}.dst \cap m_{k+1}.dst$, and C-MCast($m_{k+1}$) $\rightarrow$ C-MCast($m_{k'}$). Either (a) $m_{k+1}.sender = m_{k'}.sender$ or (b) not.

- In case (a), by the induction hypothesis, all correct processes in g eventually F-Deliver $m_{k'}$. Therefore, from the uniform FIFO order property of FIFO multicast, all correct processes in g F-Deliver m_{k+1} before $m_{k'}$.
- In case (b), since C-MCast(m_{k+1}) $\rightarrow$ C-MCast($m_{k'}$) and m_{k+1}.*sender* $\neq$ $m_{k'}$.*sender*, from the definition of $m_{k'}$, $m_{k'}$.*sender* C-Delivers m_{k+1} before C-MCasting $m_{k'}$. Hence, from the algorithm, $m_{k'}$.*sender* F-Delivers m_{k+1}. Therefore, from the uniform agreement property of FIFO multicast, all correct processes in g eventually F-Deliver m_{k+1}.

By (*) and Lemma 3.7.6, all correct processes $q \in m.dst$ eventually C-Deliver m. □

Proposition 3.7.8 (Validity) *If a correct process p C-MCasts a message m, then eventually all correct processes $q \in m.dst$ C-Deliver m.*

Proof: Let $\mathscr{M}_k$ be the subset of messages in $\mathscr{G}_{pred(m)}$ that are at distance k of m. We show that (*) for any k, all messages in $\mathscr{M}_k$ are eventually F-Delivered by all of their correct addressees.

- Base step ($k = 0$): From the algorithm, p F-MCasts m. Since p is correct, from the validity property of FIFO multicast, all correct processes in $m.dst$ eventually F-Deliver m.
- Induction step: Suppose the claim holds for any l such that $k \geq l \geq 0$, we show that the claims holds for $k+1$. The same argument as in the induction step of Proposition 3.7.7 is used.

By (*) and Lemma 3.7.6, all correct processes $q \in m.dst$ eventually C-Deliver m. □

Chapter 4

Atomic Multicast in Large Networks: Algorithms

> The most beautiful thing we can experience is the mysterious. It is the source of all true art and all science. He to whom this emotion is a stranger, who can no longer pause to wonder and stand rapt in awe, is as good as dead: his eyes are closed.
>
> *Albert Einstein*

Fault-tolerant multicast primitives are basic building-blocks for reliable distributed systems. Among these primitives, atomic broadcast and multicast are of a particular interest as they support data replication [37].

We focus on multicast algorithms tailored for large networks, e.g., wide area networks, composed of several *groups* of machines. Groups may be data centers, each located in a local area network (LAN), connected through high-latency communication links. In this context, protocols should use inter-group links sparingly.

We first devise atomic broadcast and multicast algorithms when groups do not crash, i.e., inside each group, there is at least one process that never fails. We then investigate the same problem when groups may crash entirely. Ideally, we would like to devise protocols that use inter-group links as sparingly as possible, saving on both latency and bandwidth (i.e., number of messages).

From a problem solvability point of view, atomic multicast can be easily reduced to atomic broadcast: every message is broadcast to all the groups in

the system and only delivered by those processes the message is originally addressed to. However, this solution is not genuine since processes not addressed by the message are also involved in the protocol. Hence, non-genuine multicast (e.g., atomic broadcast) may be less scalable than genuine multicast: consider a system composed of a large number of groups where messages are rarely addressed to more than a few groups. In this context, because of its genuineness property, multicast should impose a lighter load on communication links than non-genuine multicast, thus allowing for more scalability.

Nevertheless, we shall see that genuineness is an expensive property: in the case of correct groups, it imposes a minimal message delivery latency that is higher than the one of atomic broadcast and non-genuine atomic multicast. Moreover, when groups can crash entirely, genuine multicast requires perfectly accurate failure detection, i.e., erroneously suspecting a process to have crashed cannot be tolerated.

4.1 Related Work

The literature on atomic broadcast and multicast algorithms is abundant [20]; we here review the papers most relevant to our protocols. To compare multicast protocols we consider their best-case message delivery latency and inter-group message complexity. These metrics are computed by considering a failure-free scenario where a message is multicast by some process p to k groups ($k \geq 2$), including $group(p)$. We let Δ be the inter-group message delay and assume that the intra-group delay is negligible.

4.1.1 Atomic Multicast

We identify three atomic multicast protocol categories: *timestamp-based*, *round-based*, and *ring-based*. *Timestamp-based* protocols can be viewed as variations of Skeen's algorithm [10], a multicast algorithm designed for failure-free systems. In this type of protocol, messages are assigned timestamps and two properties are ensured: (a) processes agree on the final timestamp of each message and (b) message delivery follows timestamp order. In *round-based* algorithms, processes execute an unbounded sequence of rounds and agree on the set of messages A-Delivered at the end of each round. *Ring-based* algorithms propagate messages along a predefined path, denoted as a ring, and ensure agreement on the message delivery by relying on this topology.

In addition to guaranteeing agreement on the message delivery order, protocols also need to prevent holes in the delivery sequence. Algorithms categorized as *round-based* and as *ring-based* get this property for free; for some *timestamp-based* algorithms ensuring this property requires extra work however.

In [31], the authors show the impossibility of solving genuine atomic multicast with unreliable failure detectors when groups are allowed to intersect. Hence, the algorithms cited below consider non-intersecting groups. We first review algorithms that assume that all groups contain at least one correct process, i.e., disaster-vulnerable algorithms, and then algorithms that tolerate group crashes, i.e., disaster-tolerant algorithms.

Disaster-vulnerable Algorithms

Timestamp-based. In [52], the addressees of a message m, i.e., the processes to which m is multicast, exchange the timestamp they assigned to m, and, once they receive this timestamp from a majority of processes of each group, they propose the maximum value received to consensus. Consensus and gathering timestamp proposals from a majority of processes in each group, respectively, ensure properties (a) and (b) of timestamp-based protocols. Once processes in each destination group of m decide on m's definitive timestamp, they exchange their history of messages, that is, messages that were reliably delivered or decided in consensus previously. This is to prevent holes in the delivery sequence. Because consensus is run among the addressees of a message and can thus span multiple groups, this algorithm is not well-suited for wide area networks. In the best case, global messages are A-Delivered within 4Δ.

In [28], inside each group g, processes implement a logical *clock* that is used to generate timestamps; consensus is used among processes in g to maintain g's clock. Every multicast message m goes through four stages: s_0 through s_3. In s_0, in every group g addressed by m, processes define a timestamp for m using g's clock. This is g's proposal for m's final timestamp. In s_1, groups exchange their proposals and set m's final timestamp to the maximum among all proposals to ensure property (a). In the last two stages s_2 and s_3, the clock of g is updated to a value bigger than m's final timestamp and m is delivered when its timestamp is the smallest among all messages that are in one of the four stages. This guarantees property (b). Timestamps can be implemented in different ways. For example, each group g addressed by message m can define g's timestamp by using the consensus instance number that decides on m (stage s_1). Moreover, a consensus instance may decide on multiple messages, possibly in different stages. Since every consensus instance i may decide on messages in stage s_2,

after deciding in i, g's clock is set to one plus the biggest message timestamp that i decided on. In the best case, messages are A-Delivered within 2Δ, which is optimal for genuine multicast algorithms [56].

In contrast to [28], the algorithm $\mathscr{A}_{ge}^{dv}$, presented in Section 4.2.2, allows messages to skip stages. Messages that are multicast to one group only can *jump* from stage s_0 to stage s_3. Moreover, even if a message m is multicast to more than one group, at processes belonging to the group that proposed the largest timestamp, i.e., m's final timestamp, m skips stage s_2. Since consensus instances are run inside groups, the best-case latency and the number of inter-group messages sent by $\mathscr{A}_{ge}^{dv}$ are the same as in [28]. Nevertheless, we observe in Section 5.3.2 that these optimizations allow to reduce the *average* delivery latency under a broad range of loads.

Round-based. In Section 4.2.4, we present a non-genuine algorithm denoted as $\mathscr{A}_{ng}^{dv}$. This protocol is faster than [28] and [56]: it can A-Deliver messages within Δ, which is obviously optimal. This protocol is a disaster-vulnerable version of the disaster-tolerant non-genuine algorithm $\mathscr{A}_{ng}^{dt}$, described below.

Ring-based. In [21], consensus is run inside groups exclusively. Consider a message m that is multicast to groups $g_1, ..., g_k$. Message m is first multicast to g_1. Once a consensus instance decides on m, members of g_1 A-Deliver m and hand over this message to group g_2. Every subsequent group proceeds similarly up to g_k. To ensure agreement on the message delivery order, before handling other messages, every group waits for a final acknowledgment from group g_k. Two messages addressed to groups $\{g_1, g_2, g_3\}$ and $\{g_2, g_3, g_4\}$ respectively will thus be ordered by g_2. The latency of this algorithm is $(k+1)\Delta$.

Disaster-tolerant Algorithms

In Section 4.3, we present two disaster-tolerant algorithms. The first algorithm, denoted as $\mathscr{A}_{ge}^{dt}$, is timestamp-based, genuine, and tolerates an arbitrary number of failures but requires perfect failure detection. It has a latency of 6Δ and it is an open question whether this is optimal for disaster-tolerant genuine atomic multicast. The second algorithm $\mathscr{A}_{ng}^{dt}$ is not genuine and requires a two-third majority of correct processes, i.e., $f < n/3$, but only requires perfect failure detection inside each group. Unreliable failure detection can also be tolerated but at the cost of a weaker liveness guarantee. This protocol is round-based and has a latency of 2Δ. As a corollary of [38], this protocol is optimal.

It is worth noting that non-genuine atomic multicast could also be solved with atomic broadcast. However, atomic broadcast protocols do not optimize the latency of local messages.

4.1.2 Atomic Broadcast

In [3], the authors consider the atomic broadcast and multicast problems in a publish-subscribe system where links are reliable, publishers *do not crash*, and cast infinitely many messages. Agreement on the message ordering is ensured by using the same deterministic merge function at every subscriber process. Given the cast rate of publishers, the authors give optimal algorithms with regards to the merge delay, i.e., the time elapsed between the reception of a message by a subscriber and its delivery. Both algorithms achieve a latency of Δ.

In [61], a time-based protocol is introduced to increase the probability of *spontaneous* total order in wide area networks by artificially delaying messages. Although the latency of the *optimistic* delivery of a message is Δ, the latency of its *final* delivery is 2Δ. Moreover, their protocol is non-uniform, i.e., the *agreement* and *prefix order* properties of Chapter 2 are only ensured for correct processes. A uniform protocol based on multiple sequencers is proposed in [64]. Every process p is assigned a sequencer that associates sequence numbers to the messages p broadcasts. Processes optimistically deliver a message m when they receive m's sequence number. The final delivery of m occurs when the sequence number of m has been validated by a majority of processes. The latency of this algorithm is identical to [61].

In Section 4.2.3, we present a broadcast algorithm denoted as $\mathscr{A}_{bcast}^{dv}$. This protocol relies on correct groups to allow for message delivery within only one Δ.

4.1.3 Analytical Comparison

Tables 4.1 and 4.2 respectively provide a comparison of the atomic broadcast and multicast algorithms. To compute the inter-group message complexity, we assume that there are n processes in the system. Furthermore, in the case of atomic multicast, we consider that a message is addressed to k groups of d processes. In the figures below, we also provide the resiliency and the failure detectors required by the protocols. Two comments are now in order. First, we note that the atomic broadcast algorithm in [61] is non-uniform, i.e., it guarantees the prefix order and agreement properties of Chapter 2 only for correct processes. Second, the latency of the atomic multicast algorithm in [3] does not contradict the lower bound of genuine atomic multicast. Indeed, their assumptions are different from ours, i.e., to ensure liveness of their multicast algorithm, they require that each publisher multicast infinitely many messages to *each* subscriber.

Algorithm	resiliency	failure detector	inter-group latency	latency optimal?	inter-group msgs.
[61]	$f < n/2$	system-wide Ω	2Δ	yes	$O(n)$
[64]	$f < n/2$	system-wide Ω	2Δ	yes	$O(n^2)$
$\mathscr{A}^{dv}_{bcast}$	majority correct in each group	group-wide Ω	Δ	yes	$O(n^2)$
	at least one correct in each group	group-wide $\mathscr{P}$			
[3]	only subscribers may crash	-	Δ	yes	$O(n)$

Table 4.1. Comparison of the atomic broadcast algorithms (Δ is the inter-group message latency).

4.2 Disaster-vulnerable Atomic Multicast

In the context of correct groups, atomic broadcast and multicast establish an inherent trade-off, as we now explain. For messages multicast to at least two groups, we show that no genuine atomic multicast algorithm can hope to deliver messages within fewer than two inter-group message delays. This result is proven under strong system assumptions, namely processes do not crash and links are reliable. Moreover, this lower bound is tight, i.e., the fault-tolerant algorithm $\mathscr{A}^{dv}_{ge}$ of Section 4.2.2 and the algorithm in [28] achieve this latency (as opposed to [28], $\mathscr{A}^{dv}_{ge}$ reduces the number of intra-group messages sent, see Section 4.2.2). A corollary of this result is that Skeen's algorithm, initially described in [10] and designed for failure-free systems, is also optimal—a result that has apparently been left unnoticed by the scientific community for more than 20 years.

We demonstrate that atomic multicast is inherently more expensive than atomic broadcast in terms of latency. We do so by presenting a fault-tolerant broadcast algorithm, denoted as $\mathscr{A}^{dv}_{bcast}$, that delivers messages within one inter-group message delay. To achieve such a low latency, the algorithm is proactive, i.e., it may take actions even though no messages are broadcast. Nevertheless, we show how it can be made *quiescent*: provided that a finite number of messages is broadcast, processes eventually cease to communicate. The quiescent version of $\mathscr{A}^{dv}_{bcast}$ may, however, be as expensive as atomic multicast: in runs where the algorithm becomes quiescent too early, that is, a message m is broadcast after processes have decided to stop communicating, m will not be delivered in a single inter-group message delay, but in two. We show that this extra cost is

Algorithm	genuine?	resiliency	failure detector(s)	inter-group latency	latency optimal?	inter-group msgs.
[21]	yes	majority correct in each group	group-wide Ω	$(k+1)\Delta$	no	$O(kd^2)$
[52]	yes	majority correct in each group	group-wide Ω	4Δ	no	$O(k^2d^2)$
[28]	yes	majority correct in each group	group-wide Ω	2Δ	yes	$O(k^2d^2)$
$\mathcal{A}_{ge}^{dv}$	yes	majority correct in each group	group-wide Ω	2Δ	yes	$O(k^2d^2)$
		at least one correct in each group	group-wide $\mathcal{P}$			
$\mathcal{A}_{ng}^{dv}$	no	majority correct in each group	group-wide Ω	Δ	yes	$O(n^2)$
		at least one correct in each group	group-wide $\mathcal{P}$			
[3]	yes	only subscribers may crash	-	Δ	yes	$O(kd)$
$\mathcal{A}_{ge}^{dt}$	yes	$f \leq n$	system-wide $\mathcal{P}$	6Δ	?	$O(k^3d^3)$
$\mathcal{A}_{ng}^{dt}$	no	$f < n/3$	group-wide $\mathcal{P}$	2Δ	yes	$O(n^2)$
		$f < n/2$	and system-wide $\Diamond\mathcal{P}$	3Δ		

Table 4.2. Comparison of the atomic multicast algorithms (d denotes the number of processes per group, k is the number of groups addressed by the multicast message, and Δ is the inter-group message latency).

unavoidable, i.e., no quiescent atomic broadcast algorithm can hope to always deliver messages within one inter-group delay.[1]

These two lower bound results stem from a common cause, namely the *reactiveness* of the processes at the time when the message is cast. Roughly speaking, a process p is said to be *reactive* when the next message m that p sends is in response either to a local multicast event or to the reception of another message. In Section 4.2.1, we first show that no atomic broadcast or multicast algorithm can hope to deliver the last cast message m within one inter-group delay if m

[1]This result also holds for quiescent (genuine or non-genuine) atomic multicast algorithms. The genuine case is already covered by the first lower bound result and is therefore irrelevant here.

is cast at a time when processes are reactive. To obtain the lower bounds, we then show that (i) in runs of any genuine atomic multicast algorithm where one message is multicast at time t, processes are reactive at t and (ii) in runs of any quiescent atomic broadcast or atomic multicast algorithm where a finite number of messages are cast, processes are eventually reactive forever.

To circumvent the latency lower bound of genuine atomic multicast, we provide a non-genuine multicast algorithm, denoted as $\mathscr{A}_{ng}^{dv}$, that is directly inspired by $\mathscr{A}_{bcast}^{dv}$. Similarly to $\mathscr{A}_{bcast}^{dv}$, $\mathscr{A}_{ng}^{dv}$ may deliver global messages in a single inter-group delay. Moreover, $\mathscr{A}_{ng}^{dv}$ allows local messages to be delivered with no inter-group communication.

These results help better understand the difference between genuine and non-genuine atomic multicast. In particular, they point out a trade-off between the optimal message delivery latency and message complexity of these two problems. Consider a partial replication scenario where each group replicates a set of objects. If latency is the main concern, then every operation should be multicast with a non-genuine primitive. This solution, however, has a higher message complexity than genuine multicast: every operation leads to sending messages to processes unrelated to this operation. To reduce the message complexity, genuine multicast can be used. However, no genuine multicast algorithm can deliver messages within fewer than two inter-group delays. This trade-off is explored empirically in Chapter 5.

4.2.1 The Inherent Cost of Multicast

We establish the inherent cost of the genuine atomic multicast problem for messages that are multicast to multiple groups. We also show that quiescence has a cost, i.e., in runs where a message m is cast at a time when the algorithm is quiescent, there exists no algorithm that delivers m within fewer than two inter-group message delays. We proceed in two steps. We first show that, if processes are reactive when the last message m is cast, then m cannot be delivered within fewer than two inter-group message delays. We then prove that (i) in runs of any genuine atomic multicast algorithm where one message is multicast at time t, processes are reactive at t and (ii) in runs of any quiescent atomic broadcast or atomic multicast algorithm where a finite number of messages are cast, processes are eventually reactive forever.

The proofs are done in a model identical to the model of Chapter 2, except that processes do not crash and links are reliable, i.e., they do not corrupt, duplicate, or lose messages. Moreover, we assume that the inter-group message

delay is at least Δ and the intra-group delay is negligible. We denote by A-XCast an event of type A-BCast or A-MCast.

Definition 4.2.1 *In a run R of an atomic broadcast or multicast algorithm, we say that a process p is reactive at time t iff p sends a message at time $t' \geq t$ only if either p A-XCasts a message in the interval $[t, t']$ or p received a message sent in the interval $[t, t']$.*

Proposition 4.2.1 *In a system with at least two groups, for any atomic broadcast or any atomic multicast algorithm $\mathscr{A}$, there does not exist runs R_1, R_2 of $\mathscr{A}$ in which processes are reactive at the time the last messages m_1, m_2 are A-XCast to at least two groups, such that m_1 and m_2 are both A-Delivered within less than 2Δ.*

Proof: Suppose, by way of contradiction, that there exist an algorithm $\mathscr{A}$ and runs R_i of $\mathscr{A}$, $i \in \{1, 2\}$, such that, in R_i, m_i is A-Delivered within less than 2Δ. Consider two groups, g_1 and g_2. In run R_i, process $p_i \in g_i$ A-XCasts message m_i at time t to g_1 and g_2. We first show that (*) in R_i, at or after time t, processes can only send messages m such that for a sequence of events $e_1 =$ A-XCast$(m_i), e_2, ..., e_k =$ send(m), A-XCast$(m_i) \rightarrow e_2 \rightarrow ... \rightarrow$ send(m).[2] Suppose, by way of contradiction, that there exists a process p in R_i that sends a message m at a time $t'_i \geq t$ such that the event send(m) is not causally linked to the event A-XCast(m_i). We construct a run R'_i identical to run R_i except that message m_i is not A-MCast (note that processes are also reactive at time t in R'_i). Since in R_i, there is no causal chain linking the event A-XCast(m_i) with the event send(m), runs R'_i and R_i are indistinguishable to process p up to and including time t'_i. Therefore, p also sends m in R'_i. Hence, since processes are reactive at time t and no message is A-XCast at or after t, p must have received a message m' sent at or after t by some process q. Applying the same reasoning multiple times, we argue that there must exists a process r that sends a message m'' at time t such that for some events $e_1 =$ send$(m''), e_2, ..., e_{x-1} =$ send$(m'), e_x =$ send(m), we have send$(m'') \rightarrow ... \rightarrow$ send$(m') \rightarrow$ send(m). However, r cannot send m'' because no message is A-XCast at or after t, a contradiction.

By the validity property of $\mathscr{A}$ and because there is no failure, all processes eventually A-Deliver m_i. Since m_i is A-Delivered within less than 2Δ and the inter-group latency is at least Δ, by (*), processes in g_i A-Deliver m_i before

[2] Events $e_1, ..., e_k$ can be of four kinds, either send(m), receive(m), A-XCast(m), or A-Deliver(m) for some message m. Moreover, the relation $\rightarrow$ is Lamport's transitive happened before relation on events [37]. It is defined as follows: $e_1 \rightarrow e_2 \Leftrightarrow e_1, e_2$ are two events on the same process and e_1 happens before e_2 or $e_1 =$ send(m) and $e_2 =$ receive(m) for some message m.

receiving any message from processes in g_{3-i} sent at or after time t. Let $t_i^* > t$ be the time at which all processes in g_i have A-Delivered message m_i. We now build run R_3 as follows. As in run R_i, p_i A-XCasts m_i. Runs R_i and R_3 are indistinguishable for processes in group g_i up to time t_i^*, that is, all messages causally linked to the event A-XCast(m_{3-i}) (including A-XCast(m_{3-i}) itself) sent from processes in group g_{3-i} to processes in group g_i are delayed until after t_i^*. Consequently, processes in group g_i have all A-Delivered m_i by time t_i^*. By the uniform agreement of $\mathscr{A}$, processes in g_1 eventually A-Deliver m_2 and processes in g_2 eventually A-Deliver m_1. By the uniform prefix order property of $\mathscr{A}$, either (i) processes in g_1 A-Deliver m_2 before m_1 or (ii) processes in g_2 A-Deliver m_1 before m_2. Cases (i) and (ii) violate the uniform integrity property of $\mathscr{A}$ since, in case (i), processes in g_1 A-Deliver m_2 twice, and in case (ii), processes in g_2 A-Deliver m_1 twice, a contradiction. □

Proposition 4.2.2 *For any run R of any genuine atomic multicast algorithm $\mathscr{A}$ where one message m is A-MCast at an arbitrary time t, processes are reactive at time t.*

Proof: In run R, by the genuineness property of $\mathscr{A}$, for any message m' sent, there exist events $e_1 = \text{A-MCast}(m), e_2, ..., e_x = \text{send}(m')$ such that A-MCast(m) $\rightarrow$ $e_2 \rightarrow ... \rightarrow \text{send}(m')$ (otherwise, using a similar argument as in Proposition 4.2.1, we could build a run R' identical to run R, except that no message is A-MCast in R', such that a process sends a message anyway, contradicting the fact that in R' no message is A-MCast and $\mathscr{A}$ is genuine).

Consequently, for any process p, if p sends a message m' at $t' \geq t$, then either p A-MCasts a message in the interval $[t, t']$ or p received a message sent in the interval $[t, t']$. □

Proposition 4.2.3 *For any run R of any quiescent atomic broadcast or atomic multicast algorithm $\mathscr{A}$ in which a finite number of messages are A-XCast, there exists a time t such that for all $t' \geq t$, processes are reactive at t'.*

Proof: In R, a finite number of messages are A-XCast. Because $\mathscr{A}$ is quiescent, there exists a time t at or after which no messages are sent. It follows directly that for all $t' \geq t$ processes are reactive at t'. □

Although a consequence of Propositions 4.2.1 and 4.2.3 is that if the last message m is cast when processes are reactive, then m cannot be delivered within less than 2Δ, in practice, multiple messages may bear this overhead. In fact, this might even be the case in runs where an infinite number of messages are

cast. Indeed, to ensure quiescence, processes must somehow *predict* whether any message will be cast in the future. Hence, if no message is expected to be cast, processes must stop communicating, and this may happen prematurely.

4.2.2 An Optimal Genuine Atomic Multicast Algorithm

In this section, we present a latency-optimal atomic multicast algorithm which is inspired by the one from Fritzke *et al.* [28], an adaptation of Skeen's algorithm for failure-prone systems. We first explain the basic principle of our algorithm and how it differs from [28]. We then explain our algorithm in detail.

Algorithm Overview

The algorithm associates every multicast message with a timestamp. To ensure agreement on the message delivery order, two properties are ensured: (1) processes agree on the message timestamps and (2) after a process p A-Delivers a message with timestamp ts, p does not A-Deliver a message with a smaller timestamp than ts. To satisfy these two properties, inside each group g, processes implement a logical *clock* that is used to generate timestamps; this is g's clock. To guarantee g's clock consistency across members of g, processes use consensus to maintain it. Moreover, every message m goes through the following four stages:

- *Stage* s_0: In every group $g \in m.dst$, processes define a timestamp for m using g's clock. This is g's proposal for m's final timestamp.
- *Stage* s_1: Groups in $m.dst$ exchange their proposals for m's timestamp and set m's final timestamp to the maximum timestamp among all proposals.
- *Stage* s_2: Every group in $m.dst$ sets its clock to a value greater than the final timestamp of m.
- *Stage* s_3: Message m is A-Delivered when its final timestamp is the smallest among all messages that are in one of the four stages and not yet A-Delivered.

As mentioned above, our algorithm differentiates itself from [28] in several aspects. First, when a message is multicast, instead of using a uniform reliable multicast primitive, we use a non-uniform version of this primitive while still ensuring properties as strong as in [28].[3] Second, in contrast to [28], not all

[3]Non-uniform reliable multicast ensures the same properties as its uniform counterpart defined in Chapter 2, except for uniform agreement which is replaced by *agreement:* if a correct process p R-Delivers a message m, then eventually all correct processes $q \in m.dst$ R-Deliver m.

messages go through all four stages. For messages that are multicast to only one group, our algorithm allows them to *jump* from stage s_0 to stage s_3 directly. Also, even for messages that are multicast to more than one group, on processes belonging to a group that has proposed a timestamp equal to the final timestamp of m (the biggest proposal of all), m skips stage s_2. We observe in Section 5.3.2 that the latter optimization allows to reduce the *average* delivery latency under a broad range of loads.

The Algorithm in Detail

Algorithm $\mathscr{A}_{ge}^{dv}$ is composed of two concurrent tasks. Each line of the algorithm is executed atomically. Apart from application data, messages are composed of four fields: dst, id, ts, and *stage*. For every message m, $m.dst$ indicates to which group m is A-MCast, $m.id$ is m's unique identifier, $m.ts$ denotes m's current timestamp, and $m.stage$ defines in which stage m is. On every process p, four global variables are used: K is p's copy of $group(p)$'s clock and also denotes the current consensus instance in execution or the next to be executed, *propK* forbids p to propose more than one value per consensus instance, *PENDING* is the set of messages that have not yet been A-Delivered, and *ADELIVERED* is the set of A-Delivered messages. We explain Algorithm $\mathscr{A}_{ge}^{dv}$ by describing the actions a process p takes when a message m is in one of the four possible stages. We illustrate the execution in Figure 4.1.

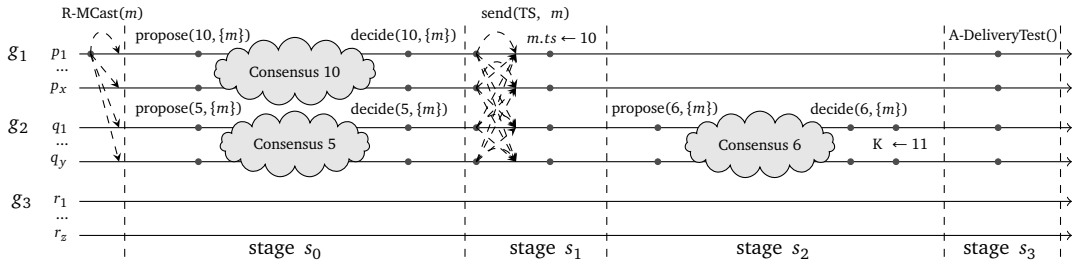


Figure 4.1. Algorithm $\mathscr{A}_{ge}^{dv}$ in the failure-free case when a message m is A-MCast to groups g_1 and g_2. At the beginning of the run above, we assume that groups g_1 and g_2 have their variable K equal to 10 and 5 respectively.

Algorithm $\mathscr{A}_{ge}^{dv}$
Genuine Atomic Multicast - Code of process p

1: **Initialization**
2: $K \leftarrow 1$, $propK \leftarrow 1$, $PENDING \leftarrow \emptyset$, $ADELIVERED \leftarrow \emptyset$

3: **procedure** ADeliveryTest()
4: **while** $\exists m \in PENDING : m.stage = s_3 \wedge$
$\forall m' \in PENDING : m' \neq m \Rightarrow (m.ts, m.id) < (m'.ts, m'.id)$ **do**
5: A-Deliver(m)
6: $ADELIVERED \leftarrow ADELIVERED \cup \{m\}$
7: $PENDING \leftarrow PENDING \setminus \{m\}$

8: **To A-MCast** message m *{Task 1}*
9: R-MCast(m) to $m.dst$

10: **When** (R-Deliver(m) $\vee$ receive(TS, m)) $\wedge$
$m \notin PENDING \cup ADELIVERED$ *{Task 2}*
11: $m.ts \leftarrow K$
12: $m.stage \leftarrow s_0$
13: $PENDING \leftarrow PENDING \cup \{m\}$

14: **When** $(\exists m \in PENDING : m.stage = s_0 \ \vee \ m.stage = s_2) \ \wedge$
$propK \leq K$
15: $msgSet = \{m \mid m \in PENDING \wedge (m.stage = s_0 \vee m.stage = s_2)\}$
16: Propose($K, msgSet$) ▷ consensus inside group
17: $propK \leftarrow K + 1$

18: **When** Decided($K, msgSet'$)
19: **foreach** $m' \in msgSet'$ **do**
20: **if** $|m'.dst| > 1$ **then**
21: **if** $m'.stage = s_0$ **then**
22: $m'.ts \leftarrow K$
23: $m'.stage \leftarrow s_1$
24: send(TS, m') to $\{q \mid q \in m.dst \wedge group(q) \neq group(p)\}$
25: **else**
26: $m'.stage \leftarrow s_3$
27: **else**
28: $m'.ts \leftarrow K$
29: $m'.stage \leftarrow s_3$ ▷ second consensus not needed
30: $PENDING \leftarrow PENDING \cup \{msgSet'\}$ ▷ add message or update its fields
31: $K \leftarrow \max(\max_{m' \in msgSet'}(m'.ts), K) + 1$
32: ADeliveryTest()

33: **When** $\exists m \in PENDING : m.stage = s_1 \ \wedge$
$\forall g \in (m.dst \setminus group(p)) \ \exists q \in g$: received(TS, m) from q
34: $TSset = \{m.ts \mid \exists q \in m.dst$: received (TS, m) from $q\}$
35: **if** $m.ts \geq \max_{ts \in TSset}(ts)$ **then**
36: $m.stage \leftarrow s_3$ ▷ second consensus not needed
37: ADeliveryTest()
38: **else**
39: $m.ts \leftarrow \max_{ts \in TSset}(ts)$
40: $m.stage \leftarrow s_2$

Stage s_0: For p to A-MCast m, p R-MCasts m to processes in $m.dst$. When p R-Delivers m, if m has not been added to *PENDING* at line 30 or A-Delivered before, p sets $m.stage$ to s_0 and adds m to the *PENDING* set. Note that p also sets m's timestamp to the current value of K to guarantee that every message in *PENDING* is associated with a timestamp. In order for processes in each group of $m.dst$ to agree on their timestamp proposal, a consensus instance inside each group is executed. Hence, p checks that $propK \leq K$ (line 14) to verify that no consensus instance is currently running and, if it is the case, p proposes m to the next consensus instance. Process p actually proposes all messages in *PENDING* that are either in stage s_0 or s_2 to share the cost of consensus instances among the set of messages proposed and to allow messages in different stages to make progress in parallel. As soon as a consensus instance k decides on m ($m \in msgSet'$ at line 18), p takes the following actions. First, if m is A-MCast to more than one group, then m transitions to stage s_1 and $group(p)$'s proposal for m's timestamp is k (lines 22-23). Otherwise, if m is A-MCast to one group only, m's final timestamp is k and m transitions to stage s_3 directly (lines 28-29). Indeed, at this point in time, processes in $m.dst$ already agree on m's timestamp because p's group is the only group in $m.dst$. Moreover, p's copy of $group(p)$'s clock, K, will be greater than $m.ts$ after p executes line 31.

Stage s_1: Process p sends its group's proposal to all the groups in $m.dst$ different from p's group (line 24).[4]

Once p receives all the required timestamps' proposals (line 33), p gathers them in a set called *TSset* and computes the maximum value, max, of that set. If the proposal of p's group is bigger than or equal to max, then m can skip stage s_2. Indeed, at this point in time, p's copy of $group(p)$'s clock, K, is already bigger than $m.ts$, because p previously executed line 31. On the other hand, if the proposal of p's group is smaller than max, p sets m's timestamp to max and m transitions to stage s_2 (line 39-40).

Stage s_2: Process p then keeps on proposing m to consensus instances until an instance k decides on m. When instance k terminates, m transitions to stage s_3 at line 26 and p sets K to a value greater than m's final timestamp at line 31.

Stage s_3: After m reaches stage s_3 (at lines 26, 29, or 36), p checks whether m can be A-Delivered by executing the procedure ADeliveryTest. This procedure A-

[4]This message also serves the purpose of propagating m. Consider a scenario where the process that A-MCasts m is faulty and m is A-MCast to multiple groups. Because the reliable multicast primitive we use is non-uniform, it is possible that only faulty processes in a group g R-Deliver m. After processes in g decide on m in consensus and send the (TS, m) message at line 24, since there is at least one correct process per group, we guarantee that correct processes in $m.dst$ receive m.

Delivers m only if m has the smallest timestamp among all messages in *PENDING* (line 4). If two messages m_1 and m_2 have the same timestamp, we break ties using their message identifier. More precisely, $(m_1.ts, m_1.id) < (m_2.ts, m_2.id)$ is true either if $m_1.ts < m_2.ts$ or if $m_1.ts = m_2.ts$ and $m_1.id < m_2.id$.

4.2.3 An Optimal Atomic Broadcast Algorithm

In this section, we present the first fault-tolerant atomic broadcast algorithm that can deliver messages in one inter-group message delay. Together with the lower bound of Section 4.2.1, this shows that in the context of correct groups, genuine atomic multicast is more costly in terms of inter-group latency than atomic broadcast. We present the main idea of this algorithm and then explain it in detail.

Algorithm Overview

To atomically broadcast a message m, a process p reliably multicasts m to the processes in p's group. In parallel, processes execute an *unbounded* sequence of rounds. At the end of each round, processes deliver a set of messages according to some deterministic order. To ensure agreement on the messages delivered in round r, processes proceed in two steps. In the first step, inside each group g, processes use consensus to define g's bundle of messages. In the second step, groups exchange their message bundles. The set of message delivered at the end of round r is the union of all bundles. Note that we also wish to ensure *quiescence*, i.e., if there is a time after which no message is broadcast, then processes eventually stop sending messages. To do so, processes try to predict when no further messages will be broadcast. When the algorithm predicts that messages will no longer be broadcast, processes stop executing rounds. Our algorithm is indulgent with regards to prediction mistakes, i.e., if processes become quiescent too early, they can restart so that liveness is still ensured. We explain below how this is done.

The Algorithm in Detail

Algorithm $\mathscr{A}_{ng}^{dv}$ is composed of four concurrent tasks. Each line of the algorithm is executed atomically. On every process p, six global variables are used: K denotes the current round number, *propK* forbids p to propose more than one value per consensus instance, *RDELIVERED* and *ADELIVERED* are the set of R-Delivered

and A-Delivered messages respectively, *Msgs* is used to store the groups' message bundles, and *Barrier* denotes the last round p currently thinks it will execute.

We explain how a message m is A-Delivered; Figure 4.2 illustrates the execution. To A-BCast m, a process p R-MCasts m to p's group (line 5). When p R-Delivers m, p adds it to *RDELIVERED*. At the beginning of every round r, p proposes, in the next consensus instance, the set of messages that have been R-Delivered but not A-Delivered yet (line 12). Note that this proposal may be the empty set. When this instance decides on a set of messages $msgSet'$ (line 14), p's group message bundle in round r, p sends $msgSet'$ to all the groups different from $group(p)$. Process p then waits to receive the message bundles of round r from all the other groups and A-Delivers the union of all bundles in some deterministic order (lines 16-20).

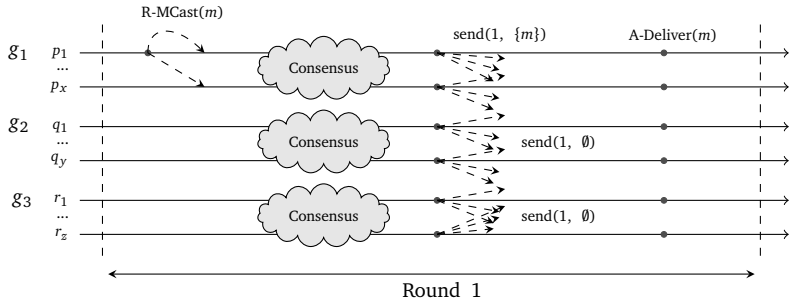


Figure 4.2. Algorithm $\mathscr{A}^{dv}_{bcast}$ when a message m is A-BCast from p_1.

In order to ensure quiescence, processes try to predict when no message will be broadcast anymore. Our prediction strategy is simple: if no message was A-Delivered in the current round (line 22), p leaves *Barrier* untouched, in which case p will not execute the next round. This follows from the fact that K is incremented by one at the end of each round (line 21), and if no message is A-BCast anymore and all R-Delivered messages were A-Delivered, line 11 never evaluates to true anymore and p does not execute further rounds.

To tolerate prediction mistakes, the algorithm proceeds as follows. Consider a process p that broadcasts a message m after all processes have become quiescent. Let r be the last round processes executed. After processes in p's group R-Deliver m, line 11 evaluates to true and they start executing round $r+1$. To allow processes in other groups to restart executing rounds as well, after receiving p's group message bundle (line 8), these processes set *Barrier* to $r+1$. Hence, line 11 evaluates to true and they start round $r+1$.

Algorithm $\mathscr{A}^{dv}_{bcast}$
Atomic Broadcast - Code of process p

1: **Initialization**
2: $K \leftarrow 1$, $propK \leftarrow 1$, $RDELIVERED \leftarrow \emptyset$, $ADELIVERED \leftarrow \emptyset$
3: $Msgs \leftarrow \emptyset$, $Barrier \leftarrow 0$

4: **To A-BCast** message m *{Task 1}*
5: R-MCast(m) to $group(p)$

6: **When** R-Deliver(m) *{Task 2}*
7: $RDELIVERED \leftarrow RDELIVERED \cup \{m\}$

8: **When** receive($x, msgSet$) from q *{Task 3}*
9: $Msgs \leftarrow Msgs \cup (x, q, msgSet)$
10: $Barrier \leftarrow \max(Barrier, x)$

11: **When** $((RDELIVERED \setminus ADELIVERED) \neq \emptyset \ \vee$
$K \leq Barrier) \ \wedge \ propK \leq K$ *{Task 4}*
12: Propose(K, $RDELIVERED \setminus ADELIVERED$) ▷ consensus inside group
13: $propK \leftarrow K + 1$

14: **When** Decided(K, $msgSet'$)
15: send(K, $msgSet'$) to $\{q \mid q \in \Pi \wedge group(q) \neq group(p)\}$
16: **wait until** $\forall g \in (\Gamma \setminus group(p)) : \exists q \in g$ s. t. received (K, -) from q
17: $Msgs \leftarrow Msgs \cup (K, p, msgSet')$
18: $msgsToADel \leftarrow \{m \mid (K, \text{-}, msgSet) \in Msgs \ \wedge \ m \in msgSet\}$
19: A-Deliver messages in $msgsToADel$ in some deterministic order
20: $ADELIVERED \leftarrow ADELIVERED \cup msgsToADel$
21: $K \leftarrow K + 1$
22: **if** $msgsToADel \neq \emptyset$ **then** ▷ stop executing rounds?
23: $Barrier \leftarrow \max(Barrier, K)$

4.2.4 Deriving a Non-Genuine Multicast Algorithm

In this section, we derive a non-genuine multicast algorithm $\mathscr{A}^{dv}_{ng}$ that can A-Deliver global messages in one message delay. This protocol is similar to Algorithm $\mathscr{A}^{dv}_{bcast}$. We explain the similarities and differences between these two protocols.

Similarly to $\mathscr{A}^{dv}_{bcast}$, to atomic multicast a message m, a process p reliably multicasts m to p's group (line 4). In parallel, processes execute an unbounded sequence of rounds, and agree on the set of messages A-Delivered in each round. To do so, in each round r, members of each group g define g's message bundle using consensus (lines 8-9), exchange their message bundle with the other groups, and processes A-Deliver the messages contained in the bundles of r that

are addressed to them (lines 17-18).

Algorithm $\mathscr{A}_{ng}^{dv}$
Non-Genuine Atomic Multicast - Code of process p

```
 1: Initialization
 2:    K ← 1, Rdel ← ∅, Decided ← ∅

 3: To A-MCast message m                                          {Task 1}
 4:    R-MCast m to group(p)

 5: When R-Deliver(m)                                             {Task 2}
 6:    Rdel ← Rdel ∪ {m}

 7: Loop                                                          {Task 3}
 8:    Propose(K, Rdel \ Decided)                  ▷ consensus inside group
 9:    wait until Decide(K, msgs)
10:    Decided ← Decided ∪ msgs
11:    lMsgs ← {m | m ∈ msgs ∧ m.dst = {group(p)}}
12:    A-Deliver messages in lMsgs in some deterministic order
13:    foreach g ∈ (Γ \ group(p))
14:       toSend ← {m | m ∈ msgs ∧ g ∈ m.dst}
15:       send(K, group(p), toSend) to all q ∈ g
16:    wait until ∀g ∈ (Γ \ group(p)) : received(K, g, msgs′)
17:    gMsgs ← {m | p ∈ m.dst ∧
                    ((m ∈ msgs ∧ |m.dst| > 1) ∨
                    (∃g ∈ Γ : received(K, g, msgs′) ∧ m ∈ msgs′))}
18:    A-Deliver messages in gMsgs in some deterministic order
19:    K ← K + 1
```

In contrast to atomic broadcast, atomic multicast allows messages to be addressed to a single group. Hence, after being decided in consensus, these local messages do not need to be propagated to the other groups. Furthermore, they can be A-Delivered directly after consensus (line 12). This optimizations allows local messages to be A-Delivered without bearing the cost of a single inter-group delay.

As is, Algorithm $\mathscr{A}_{ng}^{dv}$ sends messages forever even in runs when a finite number of messages is multicast. This algorithm can however be made quiescent using the same technique as in $\mathscr{A}_{bcast}^{dv}$. In case processes become quiescent too early, the next global message multicast will have a latency of two inter-group delays, similarly to $\mathscr{A}_{bcast}^{dv}$.

4.3 Disaster-tolerant Atomic Multicast

Mission-critical distributed applications typically replicate data in different data centers. These data centers, or groups of machines, are spread over a large geographical area to provide maximum data availability despite natural disasters. Replicating data across data centers can be achieved by means of an atomic multicast protocol that tolerates group crashes.

In this section, we devise multicast algorithms that are disaster-tolerant. These protocols rely on failure detectors that possibly provide inaccurate information about process failures. Ideally, we would like to find the *weakest* failure detector $\mathscr{D}_{amcast}$ for genuine atomic multicast.

We here consider *realistic* failure detectors only, i.e., those that cannot predict the future [22]. Moreover, we do not assume any bound on the number of processes that can crash. In this context, Delporte *et al.* showed in [22] that the weakest failure detector $\mathscr{D}_{cons}$ for consensus is the perfect failure detector $\mathscr{P}$. Obviously, atomic multicast allows to solve consensus: every process atomically multicasts its proposal; the decision of consensus is the first delivered message. Hence, the weakest realistic failure detector to solve genuine atomic multicast $\mathscr{D}_{amcast}$ when the number of faulty processes is not bounded is at least as strong as $\mathscr{P}$, i.e., $\mathscr{D}_{amcast} \succeq \mathscr{P}$. We show that $\mathscr{P}$ is in fact the weakest realistic failure detector for genuine atomic multicast when an arbitrary number of processes may fail by presenting $\mathscr{A}_{ge}^{dt}$, an algorithm that solves the problem using perfect failure detection.

As implementing $\mathscr{P}$ seems hard, if not impossible, in certain settings (e.g., wide area networks), we revisit the problem from a different angle: we consider non-genuine atomic multicast algorithms. For this purpose, atomic broadcast could be used: unreliable failure detection allows to solve atomic broadcast and cope with group crashes as long as a majority of processes are correct. This solution, however, is of little practical interest as delivering messages requires all processes to communicate, even for *local* messages. The second algorithm $\mathscr{A}_{ng}^{dt}$ we present does not suffer from this problem: local messages may be delivered without inter-group communication. Moreover, Algorithm $\mathscr{A}_{ng}^{dt}$ offers some advantages when compared to Algorithm $\mathscr{A}_{ge}^{dt}$, based on $\mathscr{P}$: wide area communication links are used sparingly, messages addressed to multiple groups can be delivered within two inter-group message delays, and perfect failure detection is only required within groups and not across the system. Although this assumption is more reasonable than implementing $\mathscr{P}$ in a wide area network, it may still be too strong for some systems. Thus, we discuss a modification to

the algorithm that tolerates unreliable failure detection, at the cost of a weaker liveness guarantee. The price to pay for the valuable features of Algorithm $\mathscr{A}_{ng}^{dt}$ is a lower process failure resiliency: group crashes are still tolerated provided that *enough* processes in the whole system are correct.

4.3.1 Solving Atomic Multicast with Perfect Failure Detection

We present the first genuine atomic multicast algorithm that tolerates an arbitrary number of process failures, i.e., $f \leq n$. We first define additional abstractions used in the algorithm, then explain the mechanisms to ensure agreement on the delivery order, and finally, we present the algorithm itself.

Additional Definitions and Assumptions

Failure Detector $\mathscr{P}$: We assume that processes have access to the perfect failure detector $\mathscr{P}$ [15]. This failure detector outputs a list of trusted processes and satisfies the following properties[5]:

- *strong completeness:* Eventually no faulty process is ever trusted by any correct process.
- *strong accuracy:* No process stops being trusted before it crashes.

Global Data Computation: We also assume the existence of a *global data computation* abstraction [25]. The global data computation problem consists in providing each process with the same vector V, with one entry per process, such that each entry is filled with a value provided by the corresponding process. Global data computation is defined by the primitives propose(v) and decide(V) and satisfies the following properties:

- *uniform validity:* If a process p decides V, then $\forall q : V[q] \in \{v_q, \perp\}$, where v_q is q's proposal.
- *termination:* If every correct process proposes a value, then every correct process eventually decides one vector.
- *uniform agreement:* If a process p decides V, then all correct processes q eventually decide V.

[5]Historically, $\mathscr{P}$ was defined to output a set of suspected processes. We here define its output as a set of trusted processes, i.e., in our definition the output corresponds to the complement of the output in the original definition.

- *uniform obligation:* If a process p decides V, then $V[p] = v_p$.

An algorithm that solves global data computation using the perfect failure detector $\mathscr{P}$ appears in [25]. This algorithm tolerates an arbitrary number of failures.

Agreeing on the Delivery Order

The algorithm associates every multicast message with a timestamp. To guarantee agreement on the message delivery order, two properties are ensured: (1) processes agree on the message timestamps and (2) after a process p A-Delivers a message with timestamp ts, p does not A-Deliver a message with a smaller timestamp than ts. These properties are implemented as described next.

For simplicity, we initially assume a multicast primitive that guarantees agreement on the set of messages processes deliver, but not causal order; we then show how this algorithm may run into problems, which can be solved using causal multicast. To A-MCast a message m_1, m_1 is first multicast to the addressees of m_1. Upon delivery of m_1, every process p uses a local variable, denoted as TS_p, to define its proposal for m_1's timestamp, $m_1.ts_p$. Process p then proposes $m_1.ts_p$ in m_1's global data computation (gdc) instance. The definitive timestamp of m_1, $m_1.ts^{def}$, is the maximum value of the decided vector V. Finally, p sets TS_p to a bigger value than $m_1.ts^{def}$ and A-Delivers m_1 when all *pending* messages have a bigger timestamp than $m_1.ts^{def}$—a message m is pending if p delivered m but did not A-Deliver m yet.

Although this reasoning ensures that processes agree on the message delivery order, the delivery sequence of faulty processes may contain *holes*. For instance, p may A-Deliver m_1 followed by m_2, while some faulty process q only A-Delivers m_2. To see why, consider the following scenario. Process p delivers m_1 and m_2, and proposes some timestamp ts_p for these two messages. As q is faulty, it may only deliver m_2 and propose some timestamp ts_q bigger than ts_p as m_2's timestamp—this is possible because q may have A-Delivered several messages before m_2 that were not addressed to p and q thus updated its TS variable. Right after deciding in m_2's gdc instance, q A-Delivers m_2 and crashes. Later, p decides in m_1 and m_2's gdc instances, and A-Delivers m_1 followed by m_2, as m_1's definitive timestamp is smaller than m_2's.

To solve this problem, before A-Delivering a message m, every process p addressed by m computes m's *potential predecessor set*, denoted as $m.pps$. This set contains all messages addressed to p that may potentially have a smaller definitive timestamp than m's (in the example above, m_1 belongs to $m_2.pps$).[6]

[6]Note that the idea of computing a message's potential predecessor set appears in the atomic

Message m is then A-Delivered when for all messages m' in $m.pps$ either (a) $m'.ts^{def}$ is known and it is bigger than $m.ts^{def}$ or (b) m' has been A-Delivered already.

The potential predecessor set of m is computed using causal multicast: To A-MCast m, m is first causally multicast. Second, after p decides in m's instance and updates its TS variable, p causally multicasts an *ack* message to the destination processes of m. As soon as p receives an *ack* message from all processes addressed by m that are trusted by its perfect failure detector module, the potential predecessor set of m is simply the set of pending messages.

Intuitively, m's potential predecessor set is correctly constructed for the two following facts: (1) any message m', addressed to p and some process q, that q causally delivers *before* multicasting m's *ack* message will be in $m.pps$ (the definitive timestamp of m' might be smaller than m's), and (2) any message causally delivered by some addressee q of m *after* multicasting m's *ack* message will have a bigger definitive timestamp than m's. Fact (1) holds from causal order, i.e., if q C-Delivers m' before multicasting m's *ack* message, then p C-Delivers m' before C-Delivering m's *ack*. Fact (2) is a consequence of the following. As p's failure detector module is perfect, p stops waiting for *ack* messages as soon as p received an *ack* from all *alive* addressees of m. Hence, since processes update their TS variable after deciding in m's global data computation instance but before multicasting the *ack* message of m, no addressee of m proposes a timestamp smaller than $m.ts^{def}$ *after* multicasting m's *ack* message.

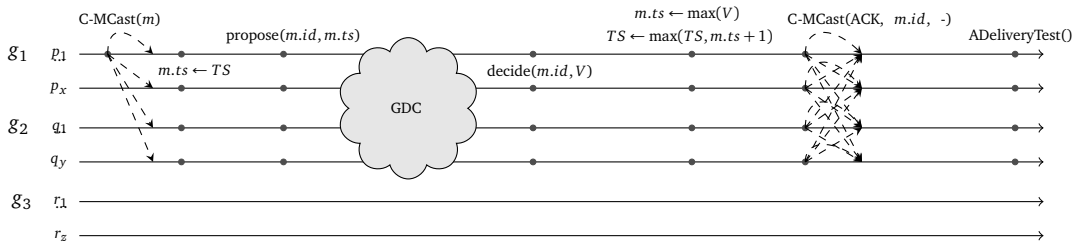


Figure 4.3. Algorithm $\mathcal{A}_{ge}^{dt}$ in the failure-free case when a message m is A-MCast to groups g_1 and g_2.

multicast algorithm of [52]. However, this algorithm assumes a majority of correct processes in every group and thus computes this set differently.

Algorithm $\mathscr{A}_{ge}^{dt}$
Genuine Atomic Multicast using $\mathscr{P}$ - Code of process p

```
1: Initialization
2:    TS ← 1, Pending ← ∅

3: procedure ADeliveryTest()
4:    while ∃m ∈ Pending : m.stage = s_2
            ∀id ∈ m.pps : ∃m' ∈ Pending : m'.id = id ⇒
                  m'.stage = s_2 ∧ (m.ts, m.id) < (m'.ts, m'.id) do
5:       A-Deliver(m)
6:       Pending ← Pending \ {m}

7: To A-MCast message m                                          {Task 1}
8:    C-MCast(m) to m.dst

9: When C-Deliver(m) atomically do                               {Task 2}
10:   m.ts ← TS
11:   m.stage ← s_0
12:   Pending ← Pending ∪ {m}

13: When ∃m ∈ Pending : m.stage = s_0                            {Task 3}
14:   m.stage ← s_1
15:   fork task ConsensusTask(m)

16: ConsensusTask(m)                                             {Task x}
17:   Propose(m.id, m.ts)          ▷ global data computation among processes in m.dst
18:   wait until Decide(m.id, V)
19:   m.ts ← max(V)
20:   TS ← max(TS, m.ts + 1)
21:   C-MCast(ACK, m.id, p) to m.dst
22:   wait until ∀q ∈ 𝒫 ∩ m.dst : C-Deliver(ACK, m.id, q)
23:   m.pps ← {m'.id | m' ∈ Pending ∧ m' ≠ m}
24:   m.stage ← s_2
25:   atomic block
26:      ADeliveryTest()
```

The Algorithm

Algorithm $\mathscr{A}_{ge}^{dt}$ is composed of four concurrent tasks. Each line of the algorithm, task 2, and the procedure ADeliveryTest are executed atomically. Messages are composed of application data plus four fields: dst, id, ts, and $stage$. For every message m, $m.dst$ indicates to which groups m is A-MCast, $m.id$ is m's unique identifier, $m.ts$ denotes m's current timestamp, and $m.stage$ defines in which stage m is. We explain Algorithm $\mathscr{A}_{ge}^{dt}$ by describing the actions a process p takes when a message m is in one of the three possible stages: s_0, s_1, or s_2. The execution is illustrated in Figure 4.3.

To A-MCast m, m is first C-MCast to its addressees (line 8). In stage s_0, p C-Delivers m, sets m's timestamp proposal, and adds m to the set of pending messages *Pending* (lines 10-12). In stage s_1, p computes $m.ts^{def}$ (lines 17-19) and ensures that all messages in $m.pps$ are in p's pending set (lines 20-23), as explained above. Finally, in stage s_2, m is A-Delivered when for all messages m' in $m.pps$ that are still in p's pending set (if m' is not in p's pending set anymore, m' was A-Delivered before), m' is in stage s_2 (and thus $m'.ts$ is the definitive timestamp of m') and $m'.ts$ is bigger than $m.ts$ (lines 4-6). Notice that if m and m' have the same timestamp, we break ties using their message identifiers, as described in Section 4.2.2.

4.3.2 Solving Atomic Multicast with Weaker Failure Detectors

The Algorithm $\mathscr{A}_{ng}^{dt}$ we present next is non-genuine but does not require system-wide perfect failure detection and delivers messages in fewer communication steps. We first define additional abstractions used by the algorithm and summarize its assumptions. We then present the algorithm itself and conclude with a discussion on how to further reduce its delivery latency and weaken its failure detection requirements.

Additional Definitions and Assumptions

Failure Detector $\Diamond\mathscr{P}$: We assume that processes have access to an eventually perfect failure detector $\Diamond\mathscr{P}$ [15]. This failure detector ensures the strong completeness property of $\mathscr{P}$ as well as the following property:

- *eventual strong accuracy:* There is a time after which no process stops being trusted before it crashes.

Generic Broadcast: Processes access a generic broadcast abstraction that ensures the same properties as atomic broadcast except that not all messages are totally ordered. More precisely, messages are taken from a set to which all messages belong. Generic broadcast depends on a user-defined symmetric and non-reflexive *conflict* relation on messages, and only orders messages that conflict. Formally, generic broadcast ensures the uniform integrity, validity, and uniform agreement properties of atomic broadcast as well as:

- *uniform generalized order:* For any two conflicting messages m and m' and any two processes p and q, if p G-Delivers m and q G-Delivers m', then either p G-Delivers m' before m or q G-Delivers m before m'.

Assumptions: To solve generic broadcast, either a simple majority of correct processes must be correct, i.e., $f < n/2$, and non-conflicting messages may be delivered in three message delays [5] or a two-third majority of processes must be correct, i.e., $f < n/3$, and non-conflicting message may be delivered in two message delays [48]. Both algorithms require a system-wide leader failure detector Ω [14], and thus the eventual perfect failure detector $\Diamond\mathscr{P}$ we assume is sufficient. Moreover, inside each group, we need consensus and reliable multicast abstractions that tolerate an arbitrary number of failures. For this purpose, among realistic failure detectors, $\mathscr{P}$ is necessary and sufficient for consensus [22] and sufficient for reliable multicast [4].[7] Note that in practice, implementing $\mathscr{P}$ within each group is more reasonable than across the system, especially if groups are inside local area networks. Despite this fact, this assumption may still be too strong for some systems. We thus discuss below how to tolerate unreliable failure detection.

Algorithm Overview

Algorithm $\mathscr{A}_{ng}^{dt}$ is inspired by the atomic multicast algorithm $\mathscr{A}_{ng}^{dv}$, which assumes that there is at least one correct process in every group. We recall its main ideas and then explain how we cope with group failures.

To A-MCast a message m, a process p R-MCasts m to p's group. In parallel, processes execute an *unbounded* sequence of rounds. At the end of each round, processes A-Deliver a set of messages according to some deterministic order. To ensure agreement on the messages A-Delivered in round r, processes proceed in two steps. In the first step, inside each group g, processes use consensus to define g's bundle of messages. In the second step, groups exchange their message bundles. The set of message A-Delivered by some process p at the end of round r is the union of all bundles, restricted to messages addressed to p.

In case of group crashes, however, this solution does not ensure liveness. Indeed, if a group g crashes, then there will be some round r after which no process receives the message bundles of g. To circumvent this problem we proceed in two steps: (a) we allow processes to stop waiting for g's message bundle, and (b) we let processes agree on the set of message bundles to consider for each round.

To implement (a), processes maintain a common *view* of the groups that are trusted to be alive, i.e., groups that contain at least one alive process. Processes then wait for the message bundles from the groups currently in the view. A group

[7] In [4], the authors present the weakest failure detector to solve reliable broadcast. Extending the algorithm of [4] to the multicast case using the same failure detector is straightforward.

g may be erroneously removed from the view if it was mistakenly suspected of having crashed. Therefore, to ensure that message m multicast by a correct process is delivered by all correct addressees of m, we allow members of g to add their group back to the view. To achieve (b), processes agree on the sequence of views and the set of message bundles between each view change. For this purpose, we use a generic broadcast abstraction to propagate message bundles and view change messages, i.e., messages to add or remove groups. Since message bundles can be delivered in different orders at different processes, provided that they are delivered between the same two view change messages, we define the message conflict relation as follows: view change messages conflict with all messages and message bundles only conflict with view change messages. As view change messages are not expected to be broadcast often, such a conflict relation definition allows for faster message bundle delivery.

Processes may also A-Deliver local messages to some group g without communicating with processes outside of g. As these messages are addressed to g only, members of g may A-Deliver them directly after consensus, and thus before receiving the groups' message bundles.

We note that maintaining a common view of the alive groups in the system resembles what is called in the literature group membership [18]. Intuitively, a group membership service provides processes with a consistent view of alive processes in the system, i.e., processes "see" the same sequence of views. Moreover, processes agree on the set of messages delivered between each view change, a property that is required for message bundles. In fact, our algorithm could have been built on top of such an abstraction. However, doing so would have given us less freedom to optimize the delivery latency of message bundles.

The Algorithm

Algorithm $\mathscr{A}_{ng}^{dt}$ is composed of five concurrent tasks. Each line of the algorithm is executed atomically. On every process p, six global variables are used: *Rnd* denotes the current round number, *Rdelivered* and *Adelivered* are the set of R-Delivered and A-Delivered messages respectively, *Gdelivered* is the sequence of G-Delivered messages, *MsgBundle* stores the message bundles, and *View* is the set of groups currently deemed to be alive.

In the algorithm, every G-BCast message m has the following format: $(rnd, g, type, msgs)$, where *rnd* denotes the round in which m was G-BCast, g is the group m refers to, *type* denotes m's type and is either *msgBundle*, *add*, or *remove*, and *msgs* is a set of messages; this field is only used if m is a message bundle.

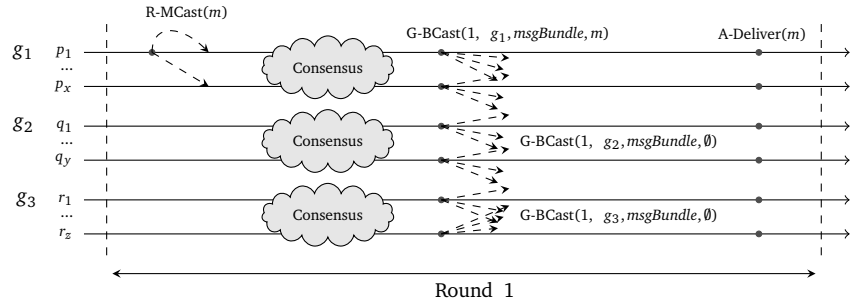


Figure 4.4. Algorithm $\mathscr{A}_{ng}^{dt}$ in the failure-free case when a message m is A-MCast to groups g_1 and g_2.

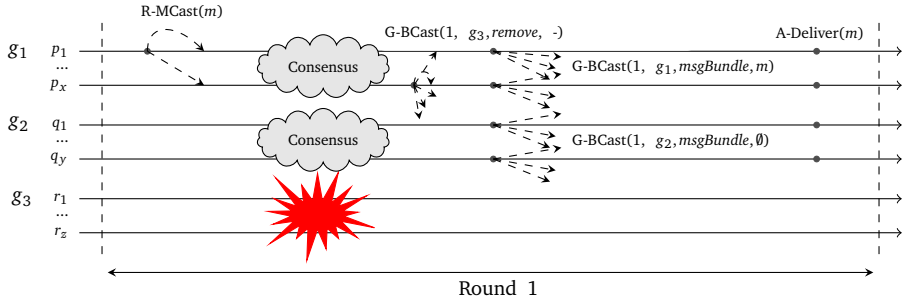


Figure 4.5. Algorithm $\mathscr{A}_{ng}^{dt}$ when group g_3 crashes and a message m is A-MCast to groups g_1 and g_2.

We explain how a message m is A-Delivered; Figures 4.4 and 4.5 respectively illustrate a failure-free run of the algorithm and a run where group g_3 crashes entirely. To A-MCast m, a process p R-MCasts m to p's group (line 5). In every round r, the set of messages that have been R-Delivered but not A-Delivered yet are proposed to the next consensus instance (line 9), p A-Delivers the set of local messages decided in this instance (line 12), and global messages are G-BCast at line 14 if $group(p)$ belongs to the view. Otherwise, p G-BCasts a message to add $group(p)$ to the view.

Algorithm $\mathscr{A}_{ng}^{dt}$
Non-Genuine Atomic Multicast - Code of process p

1: **Initialization**
2: $Rnd \leftarrow 1$, $Rdelivered \leftarrow \emptyset$, $Adelivered \leftarrow \emptyset$, $Gdelivered \leftarrow \epsilon$
3: $View \leftarrow \Gamma$, $MsgBundle[g] \leftarrow \bot$ for each group $g \in \Gamma$

4: **To A-MCast** message m *{Task 1}*
5: R-MCast(m) to $group(p)$

6: **When** R-Deliver(m) *{Task 2}*
7: $Rdelivered \leftarrow Rdelivered \cup \{m\}$

8: **Loop** *{Task 3}*
9: Propose(Rnd, $Rdelivered \setminus Adelivered$) ▷ consensus inside group
10: **wait until** Decide(Rnd, $msgs$)

11: $localMsgs \leftarrow \{m \mid m \in msgs \ \wedge \ m.dst = \{group(p)\}\}$
12: **A-Deliver** messages in $localMsgs$ in some deterministic order
13: $Adelivered \leftarrow Adelivered \cup localMsgs$

14: **if** $group(p) \in View$ **then** G-BCast($Rnd, group(p), msgBundle, msgs \setminus localMsgs$)
15: **else** G-BCast(-, $group(p), add$, -)
16: $groupsToAdd \leftarrow \emptyset$

17: **while** $\exists g \in \Gamma : MsgBundle[g] \in \{\bot, \top\}$
18: **if** $\nexists (rnd, g, type, msgs) \in Gdelivered : (rnd = Rnd \ \vee \ type = add)$ **then**
19: **wait until** G-Deliver($rnd, g, type, msgs$) $\wedge$ $(rnd = Rnd \ \vee \ type = add)$
20: $(rnd, g', type, msgs) \leftarrow$ remove first message in $Gdelivered$ s.t. $(rnd = Rnd \ \vee \ type = add)$

21: **if** $MsgBundle[g'] \in \{\bot, \top\}$ **then**
22: **if** $type = add$ **then** $groupsToAdd \leftarrow groupsToAdd \cup \{g'\}$
23: **else if** $type = remove$ **then** $MsgBundle[g'] \leftarrow \emptyset$
24: **else** $MsgBundle[g'] \leftarrow msgs$
25: $globalMsgs \leftarrow \{m \mid \exists g \in \Gamma : MsgBundle[g] = msgs \ \wedge \ m \in msgs\}$
26: **A-Deliver** messages in $globalMsgs$ addressed to p in some deterministic order
27: $Adelivered \leftarrow Adelivered \cup globalMsgs$

28: $View \leftarrow \{g \mid MsgBundle[g] \neq \emptyset\} \cup groupsToAdd$
29: **foreach** $g \in \Gamma : MsgBundle[g] \leftarrow \bot$ (if $g \in View$) or $\emptyset$ (otherwise)
30: $Rnd \leftarrow Rnd + 1$

31: **When** $\exists g \in View : MsgBundle[g] = \bot \ \wedge \ \forall q \in g : q \notin \Diamond\mathscr{P}$ *{Task 4}*
32: G-BCast(Rnd, g, $remove$, -)
33: $MsgBundle[g] \leftarrow \top$

34: **When** G-Deliver($type, m$) *{Task 5}*
35: $Gdelivered \leftarrow Gdelivered \oplus (rnd, g, type, msgs)$

Process p then gathers message bundles of the current round k using variable *MsgBundle*: Process p executes the while loop of lines 17–24 until, for

every group g, $MsgBundle[g]$ is neither $\perp$, i.e. p is not waiting to receive a message bundle from g, nor $\top$, a value whose significance is explained below. The first message m_g^k of round k related to g of type *msgBundle* or *remove* that p G-Delivers "locks" $MsgBundle[g]$, i.e., any subsequent G-Delivered message of round k concerning g is discarded (line 21). If m_g^k is of type *msgBundle*, p stores g's message bundle in $MsgBundle[g]$ (line 24). Otherwise, m_g^k was G-BCast by some process q that suspected g to have crashed entirely, i.e., failure detector $\Diamond\mathscr{P}$ at q did not trust any member of g (lines 31-33), and thus p sets $MsgBundle[g]$ to $\emptyset$ (line 23). Note that q sets $MsgBundle[g]$ to $\top$ after G-BCasting a message of the form $(k, g, remove, \text{-})$ to prevent q from G-BCasting multiple "remove g" messages in the same round.

While p is gathering message bundles for round k, it may also handle some message of type *add* concerning g, in which case p adds g to a local variable *groupsToAdd* (line 22). Note that this type of message is not tagged with a round number to ensure that messages A-MCast from correct groups are eventually A-Delivered by their correct addressees. In fact, tagging *add* messages with the round number could prevent a group from being added to the view as we now explain. Consider a correct group g that is removed from the view in the first round. In every round, members of g G-BCast a message to add g back to the view. In every round, however, processes G-Deliver message bundles of groups in the view before G-Delivering these "add g" messages, and they are thus discarded.

After exiting from the while loop, p A-Delivers global messages (line 26), the view is recomputed as the groups g such that $MsgBundle[g] \neq \emptyset$ or $g \in groupsToAdd$ (line 28), and p sets $MsgBundle[g]$ to either $\perp$, if g belongs to the new view, or $\emptyset$ otherwise, i.e., p will not wait for a message bundle from g in the next round.

Further Improvements

To weaken the failure detector required inside each group, i.e., $\mathscr{P}$ in Algorithm $\mathscr{A}_{ng}^{dt}$, we may remove a group g from the view as soon as a majority of processes in g are suspected. This allows to use consensus and reliable multicast algorithms that are safe under an arbitrary number of failures and live only when a majority of processes are correct. Hence, the leader failure detector Ω becomes sufficient. Care should be taken as when to add g to the view again: this should only be done when a majority of processes in g are trusted to be alive. This solution ensures a weaker liveness guarantee however: correct pro-

cesses in some group g will *successfully* multicast and deliver messages only if g is *maj-correct*, i.e., g contains a majority of correct processes. More precisely, the liveness guaranteed by this modified algorithm is as follows (uniform integrity and uniform prefix order remain unchanged):

- *weak uniform agreement:* if a process p A-Delivers a message m, then all correct processes $q \in m.dst$ in a maj-correct group eventually A-Deliver m.
- *weak validity:* if a correct process p in a maj-correct group A-MCasts a message m, then all correct processes $q \in m.dst$ in a maj-correct group eventually A-Deliver m.

4.4 Discussion

In this chapter, we addressed the problem of solving atomic multicast in large scale networks. In the case of correct groups, we demonstrated that no genuine atomic multicast can deliver global messages within fewer than two inter-group message delays. This bound is tight: the algorithm in [28] and $\mathscr{A}_{ge}^{dv}$ achieve a latency of two inter-group message delays. We then presented an atomic broadcast and a non-genuine atomic multicast protocol that can deliver global messages in a single message delay, thus showing that the genuineness of multicast is an expensive property.

In the case of faulty groups, we presented two algorithms. The first algorithm is genuine and tolerates an arbitrary number of process failures but it is costly in terms of latency: global messages are delivered within a minimum of six inter-group message delays. Furthermore, this algorithm requires the perfect failure detector $\mathscr{P}$. We showed that if we consider realistic failure detectors only and we do not bound the number of failures, $\mathscr{P}$ is necessary to solve this problem. The second algorithm we presented is not genuine but requires perfect failure detection inside each group only and may deliver messages addressed to multiple groups within two inter-group message delays. We showed how this latter algorithm can be modified to cope with unreliable failure detection, at the cost of a weaker liveness guarantee.

4.5 Proofs of Correctness

4.5.1 The Proof of Algorithm $\mathscr{A}_{ge}^{dv}$

In the following proofs, for brevity, we often write: "process p decides on m in instance k" instead of writing "process p decides on $msgSet'$ such that $m \in msgSet'$ in instance k". Let S_1 and S_2 be two sequences. We use the following definition of *is a prefix of*: S_1 is a prefix of S_2 iff $\exists \alpha : S_1 \oplus \alpha = S_2$.

Definition 4.5.1 *We denote as κ_p^t the sequence of values taken by variable K on process p up to time t.*

Lemma 4.5.1 *For any two processes p, q such that $group(p) = group(q)$ and any time t, either κ_p^t is a prefix of κ_q^t or κ_q^t is a prefix of κ_p^t.*

Proof: We proceed by induction on the length l of κ_p^t.

- Base step ($l = 1$) : K is initialized to 1, therefore $\kappa_p^t = \{1\}$ and 1 is the first element of κ_q^t. Therefore, κ_p^t is a prefix of κ_q^t.
- Induction step: Suppose that Lemma 4.5.1 holds for $x = l - 1$, we prove that Lemma 4.5.1 holds for $x = l$. We do so by showing that $\neg(\kappa_p^t$ is a prefix of $\kappa_q^t) \Rightarrow \kappa_q^t$ is a prefix of κ_p^t. Suppose, by way of contradiction, that (*) $\neg(\kappa_p^t$ is a prefix of $\kappa_q^t) \wedge \neg(\kappa_q^t$ is a prefix of $\kappa_p^t)$. By the induction hypothesis, either (a) $\exists \alpha : \kappa_{p_{l-1}}^t \oplus \alpha = \kappa_q^t$ or (b) $\exists \beta : \kappa_q^t \oplus \beta = \kappa_{p_{l-1}}^t$.[8] We now show that (a) and (b) lead to a contradiction.
  - In case (a), $\kappa_{p_{l-1}}^t = \{k_1, .., k_{l-1}\}$, $\kappa_p^t = \{k_1, .., k_l\}$, and $\kappa_q^t = \{k_1, .., k_{l-1}\} \oplus \alpha'$. There are two cases to consider, (a-i) $\alpha' = \epsilon$ or (a-ii) $\alpha' \neq \epsilon$.

    In case (a-i), $\alpha' = \epsilon$ and therefore κ_q^t is a prefix of κ_p^t, a contradiction to (*).

    In case (a-ii), because $\neg(\kappa_p^t$ is a prefix of $\kappa_q^t)$, (**) the first integer k_a in α' is different from k_l. By the uniform agreement property of consensus, p and q decide on the same set of messages in instance k_{l-1}. Therefore, p and q set their variable K to the same value at line 31 when $K_p = K_q = k_{l-1}$. Consequently, $k_a = k_l$, a contradiction to (**).
  - In case (b), $\exists \beta : \kappa_q^t \oplus \beta = \kappa_{p_{l-1}}^t$ and therefore $\exists \beta' : \kappa_q^t \oplus \beta' = \kappa_p^t$, a contradiction to (*). □

[8] $\kappa_{p_{l-1}}^t$ denotes the prefix of κ_p^t of length $l - 1$.

Lemma 4.5.2 *For any correct process p, any t, and any process q such that group(p) = group(q), there exists a t' such that $\kappa_p^{t'} = \kappa_q^t$.*

Proof: By Lemma 4.5.1, either (a) κ_p^t is a prefix of κ_q^t or (b) κ_q^t is a prefix of κ_p^t.

- In case (a), κ_p^t is a prefix of κ_q^t. Therefore, there exists α such that $\kappa_p^t \oplus \alpha = \kappa_q^t$. Let k be the length of α and α_x be the prefix of α of length x. We show by induction on x that for $1 \leq x \leq k$, there exists a time t' such that $\kappa_p^{t'} = \kappa_p^t \oplus \alpha_x$.
  - Base step ($x = 1$): Let k_1 and k_2 be the last and only element of κ_p^t and α_1 respectively. Because there is a time at which $K_q = k_2$, q decided in instance k_1. By the uniform agreement property of consensus p eventually decides in instance k_1 and p, q decide on the same set of messages in that instance. Therefore, p eventually executes line 31 and sets K_p to k_2.
  - Induction step: Suppose that there exists a time t' such that $\kappa_p^{t'} = \kappa_p^t \oplus \alpha_{x-1}$, we show that this also holds for x ($1 \leq x \leq k$). The same argument as in the base step is used, where k_1 is the last element of α_{x-1} and k_2 is the last element of α_x.
- In case (b), κ_q^t is a prefix of κ_p^t, therefore there exists a time t' such that $\kappa_p^{t'} = \kappa_q^t$. □

Lemma 4.5.3 *For any message m and any process p, after p adds m to $PENDING_p$ at line 13 or line 30, $m \in PENDING_p \cup ADELIVERED_p$ forever.*

Proof: Before m is removed from $PENDING_p$ at line 7, m is added to $ADELIVERED_p$ at line 6. Therefore, after m is added to $PENDING_p$ either at line 13 or line 30, $m \in PENDING_q \cup ADELIVERED_q$ forever. □

Lemma 4.5.4 *For any message m and any process p:*

- *(a) p executes at most once line 18 when $m.stage = s_0$ with $m \in msgSet'$*
- *(b) p executes at most once line 18 when $m.stage = s_2$ with $m \in msgSet'$*

Proof:

- (a) Suppose, by way of contradiction, that p executes line 18 such that $m \in msgSet' \wedge m.stage = s_0$ more than once. Let k and k' ($k < k'$) be the first and second consensus instances such that p decides on a $msgSet'$ with $m \in msgSet' \wedge m.stage = s_0$ at line 18. By the uniform integrity property of consensus, there exists a process $q \in group(p)$ such that q proposes m to instance k' with $m.stage = s_0$. By Lemma 4.5.1, q decides in instance k before proposing m in instance k'. By the uniform agreement property of consensus, q decides on m in instance k. By Lemma 4.5.3, after q finishes executing line 30 when $K_q = k$, $m \in PENDING_q \cup ADELIVERED_q$ forever. Hence, after deciding in instance k, q cannot execute line 12 to set m's stage back to s_0. Therefore, q does not propose m to consensus instance k' such that $m.stage = s_0$, a contradiction.

- (b) Notice that p can only execute line 18 such that $m.stage = s_2$ with $m \in msgSet'$ if $|m.dst| > 1$. Suppose, by way of contradiction, that p executes line 18 such that $m \in msgSet' \wedge m.stage = s_2$ more than once. Let k_1 and k_3 ($k_1 < k_3$) be the first and second consensus instances such that p decides on m with $m.stage = s_2$. By the uniform integrity property of consensus, there exists a process $q \in group(p)$ such that q proposes m to instance k_3 with $m.stage = s_2$. By Lemma 4.5.1, q decides in instance k_1 before proposing m in instance k_3. By the uniform agreement property of consensus, q decides on m in instance k_1 such that $m.stage = s_2$. Because q sets m's stage to s_3 after deciding in instance k_1 at line 26, after deciding in instance k_1 and before proposing m in instance k_3, either (b-i) q sets m's stage back to s_0 at line 12 or (b-ii) q sets m's stage back to stage s_1 at line 22. We show that (b-i) and (b-ii) lead to a contradiction.

  - In case (b-i), by Lemma 4.5.3, after q executes line 30 when $K_q = k_1$, $m \in PENDING_q \cup ADELIVERED_q$ forever. Therefore, q does not execute line 12 after deciding in instance k_1, a contradiction.

  - In case (b-ii), there exists a consensus instance k_2 ($k_1 < k_2 < k_3$) such that q decides on m with $m.stage = s_0$ in instance k_2. By the uniform integrity property of consensus, there exists a process $r \in group(q)$ such that r proposes m in instance k_2. By Lemma 4.5.1, r decides in instance k_1 before proposing m in instance k_2. By the uniform agreement property of consensus, r decides on m with $m.stage = s_2$ in instance k_1. By Lemma 4.5.3, after r executes line 30 when $K_r = k_1$, $m \in PENDING_r \cup ADELIVERED_r$ forever. Therefore, r does not set m's stage back to s_0 at line 12 after deciding in instance k_1. Consequently,

r does not propose m in instance k_2 such that $m.stage = s_0$, a contradiction. □

Lemma 4.5.5 *For any message m and any process p, on p m transitions only once to stage* s_3.

Proof: There are two cases to consider:

- (1) $|m.dst| = 1$: Follows directly from Lemma 4.5.4.
- (2) $|m.dst| > 1$: Message m transitions to stage s_3 either (i) at line 26 or (ii) at line 36. In case (i), by Lemma 4.5.4, p decides on m such that $m.stage = s_2$ only once. Therefore, m transitions to stage s_3 at line 26 only once. In case (ii), by Lemma 4.5.4, p decides on m such that $m.stage = s_0$ only once. Therefore, m transitions to stage s_1 at line 23 only once and consequently m transitions to stage s_3 at line 36 only once. □

Definition 4.5.2 *From Lemma 4.5.5 and because the timestamp of a message m does not change after m transitions to stage* s_3*, we can define* $m.ts_p^{s_3}$ *as the timestamp of m on a process p when* $m.stage = s_3$*. If m never transitions to stage* s_3 *on p, then* $m.ts_p^{s_3} = \perp$.

Proposition 4.5.1 (Uniform Integrity) *For any process p and any message m, (a) p A-Delivers m at most once, and (b) only if* $p \in m.dst$ *and m was previously A-MCast.*

Proof:

- (a) By Lemma 4.5.5, on p, m transitions to stage s_3 only once. Because m is only A-Delivered if $m.stage = s_3$ such that $m \in$ *PENDING* and because m is removed from *PENDING* just after it has been A-Delivered, p A-Delivers m at most once.
- (b) Follows directly from the algorithm. □

Lemma 4.5.6 *For any message m and any correct process p, if there exists a time at which* $m \in PENDING_p$ *such that* $m.stage = s_0$*, then for all correct processes* $q \in group(p)$ *there exists a time at which* $m \in PENDING_q$.

Proof: Process p can only add m to $PENDING_p$ such that $m.stage = s_0$ at line 13. Therefore, either m was R-Delivered or (TS, m) was received. In the first case, because p and q are correct and by the agreement property of reliable multicast, all correct processes $q \in group(p)$ eventually R-Deliver m. In the second case, because p and q are correct and links are quasi-reliable, all correct processes $q \in group(p)$ eventually receive (TS, m). Therefore q eventually adds m to $PENDING_q$ if $m \notin PENDING_q \cup ADELIVERED_q$. Note that if $m \in ADELIVERED_q$, there exists a time at which $m \in PENDING_q$. □

Lemma 4.5.7 *For any message m and any correct process p:*

- *(a) if there exists a time t at which* $m \in PENDING_p$ *such that* $m.stage_p = s_0$*, then all correct processes* $q \in group(p)$ *eventually execute line 18 such that* $m \in msgSet' \wedge m.stage = s_0$.
- *(b) if there exists a time t at which* $m \in PENDING_p$ *such that* $m.stage_p = s_1$*, then for all correct processes* $q \in m.dst$*, m eventually reaches stage* s_1 *on q.*
- *(c) if there exists a time t at which* $m \in PENDING_p$ *such that* $m.stage_p = s_2$*, then all correct processes* $q \in group(p)$ *eventually execute line 18 such that* $m \in msgSet' \wedge m.stage = s_2$.
- *(d) if there exists a time t at which* $m \in PENDING_p$ *such that* $m.stage_p = s_1$*, then for all correct processes* $q \in m.dst$*, m eventually reaches stage* s_3 *on q.*

Proof:

- (a) By Lemma 4.5.6, eventually $m \in PENDING_q$. Suppose, by way of contradiction, that there exists a correct process $r \in group(p)$ that never decides on m at line 18 such that $m.stage = s_0$. Therefore, by by Lemma 4.5.2, and by the uniform agreement and termination properties of consensus, there exists no correct process $q \in group(p)$ that decides on m with $m.stage = s_0$. Consequently, m never reaches stage s_1 on process q and no process q A-Delivers m. Therefore for all q, $m \in PENDING_q \wedge m.stage = s_0$ holds forever. Let t be the time after which all faulty processes have crashed. Therefore, after t, (*) processes q always propose a set of messages *msgSet* such that $m \in msgSet \wedge m.stage = s_0$ at line 16. By the termination property of consensus, processes q execute an infinite number of consensus instances. Therefore by (*), and by the uniform integrity and uniform agreement properties of consensus, processes q (including r) eventually decide on m such that $m.stage = s_0$, a contradiction.

- (b) Because p sets m's stage to s_1 at line 23, p decided on m in an instance k such that $m.stage = s_0$. There are two cases to consider: (b-i) $q \in group(p)$ and (b-ii) $q \notin group(p)$.

  - In case (b-i), by the uniform agreement property of consensus and by Lemma 4.5.2, q eventually decides on m in instance k such that $m.stage = s_0$. Consequently, q eventually sets m's stage to s_1 at line 23.
  - In case (b-ii), p sends a (TS, m) message to all $q \in m.dst \setminus group(p)$. Because p is correct and links are quasi-reliable, (*) q eventually receive that message.

    We now prove that for all groups in $m.dst \setminus group(p)$ there exists at least one process r such that there exists a time at which r adds m to $PENDING_r$ with $m.stage = s_0$ at line 13. By (a), this shows that m eventually transitions to stage s_1 on all correct processes $q \in m.dst \setminus group(g)$. Suppose, by way of contradiction, that there exists a group $g \in (m.dst \setminus group(p))$ in which no process adds m to *PENDING* at line 13. Consequently, because consensus instances are executed inside groups, in g, no process adds m to *PENDING* or *ADELIVERED*. Therefore, by (*) all processes in g eventually execute line 13, a contradiction.

- (c) We first prove that m eventually reaches stage s_2 on q. If $m.stage = s_2$ on p, then m reached stage s_1 on p before. Therefore by (b), m eventually reaches stage s_1 on all correct processes $r \in m.dst$. Therefore, r sends a (TS, m) message to all processes in $m.dst \setminus group(r)$ and because there is at least one correct process per group and links are quasi-reliable, all processes $q \in group(p)$ eventually receive (TS, m) messages from every group different from $group(p)$. As m reaches stage s_2 on p, on every correct process $q \in group(p)$, line 35 evaluates to false, and m reaches stage s_2 on all q.
  Now suppose, by way of contradiction, that there exists a correct process $s \in group(p)$ that never decides on m at line 18 such that $m.stage = s_2$. Consequently, by Lemma 4.5.2, and by the uniform agreement and termination properties of consensus, no process in $group(p)$ decides on m such that $m.stage = s_2$ and none A-Delivers m. Therefore for all correct processes $q \in group(p)$, $m \in PENDING_q \wedge m.stage = s_2$ holds forever. Let t be the time after which all faulty processes have crashed. Therefore, after t, (*) processes q always propose a set of messages *msgSet* such that

$m \in msgSet \wedge m.stage = s_2$ at line 16. By Lemma 4.5.2 and the termination property of consensus, processes q execute an infinite number of consensus instances. Therefore, by (*), and the uniform integrity and uniform agreement properties of consensus, processes q (including s) eventually decide on m such that $m.stage = s_2$, a contradiction.

- (d) If there exists a time at which, on p, $m \in PENDING_p$ such that $m.stage = s_1$, then by (b), on all correct processes $q \in m.dst$, m reaches stage s_1 and q executes line 24. Because there is at least one correct process per group, links are quasi-reliable, and processes p and q are correct, q eventually executes line 33. There are two cases to consider: on q, either (i) line 35 evaluates to true or (ii) not.
  - In case (i), Lemma 4.5.7-(d) trivially holds from the algorithm.
  - In case (ii), m reaches stage s_2 on q. By (c), q eventually decide on m such that $m.stage = s_2$. Therefore, m eventually reaches stage s_3 at line 26. □

Lemma 4.5.8 *For any message m and any correct process p, if there exists a time at which $m \in PENDING_p$, then m eventually reaches stage s_3 on p.*

Proof: There are two cases to consider, either (a) $|m.dst| = 1$, or (b) $|m.dst| > 1$:

- In case (a), p adds m to $PENDING_p$ (a-i) at line 13 or (a-ii) at line 30.
  - In case (a-i), by Lemma 4.5.7-(a), all correct processes $q \in \text{group}(p)$ (including p) eventually decide on a consensus instance such that $m \in msgSet' \wedge m.stage = s_0$ and on p, $m.stage$ is set to s_3 at line 29.
  - In case (a-ii), Lemma 4.5.8 trivially holds from the algorithm.
- In case (b), p adds m to $PENDING_p$ (b-i) at line 13 or (b-ii) at line 30.
  - In case (b-i), by Lemma 4.5.7-(a), all correct processes in $q \in \text{group}(p)$ eventually execute line 18 such that $m \in msgSet' \wedge m.stage = s_0$. Because there exists at least one correct process in each group, there is at least one correct process $r \in \text{group}(p)$ such that m reaches stage s_1 at line 23. Consequently, by Lemma 4.5.7-(d), m reaches stage s_3 on all correct processes in $m.dst$ (including p).
  - In case (b-ii), when p adds m to $PENDING_p$, either (b-ii-1) $m.stage = s_1$ or (b-ii-2) $m.stage = s_3$.

In case (b-ii-1), by Lemma 4.5.7-(d), on all correct processes $q \in m.dst$ (including p), m eventually reaches stage s_3.
In case (b-ii-2), Lemma 4.5.8 holds. □

Lemma 4.5.9 *For any correct process p and any message m such that there exists a time at which $m \in PENDING_p$, after m reaches stage s_3 on p, p eventually stops adding messages m' to $PENDING_p$ such that $m'.ts \leq m.ts_p^{s_3}$.*

Proof: Message m reaches stage s_3 at line 26, at line 29, or at line 36. In the three cases, from line 31, there exist a time t at which $K_p > m.ts_p^{s_3}$. After t, p can add a message m' to $PENDING_p$ either (a) at line 13 or (b) at line 30.

- In case (a), before adding m' to $PENDING_p$, m'.ts is set to K_p.
- In case (b), there are three sub-cases to consider, (c-i) $|m'.dst| > 1 \wedge m'.stage = s_1$, (c-ii) $|m'.dst| > 1 \wedge m'.stage = s_3$, or (c-iii) $|m'.dst| = 1$.
  - In cases (c-i) and (c-iii), $m'.ts$ is set to K_p.
  - In case (c-ii), suppose, by way of contradiction, that p adds an infinite number of messages m' at line 30 such that $|m'.dst| > 1$, $m'.stage = s_3$, and $m'.ts < m.ts_p^{s_3}$. Therefore, by the uniform integrity property of consensus and because $|\Pi| < \infty$, there exists a process $r \in group(p)$ that proposes messages m' such that $m'.stage = s_2$ and $m'.ts < m.ts_p^{s_3}$ an infinite number of times. After such a message m' is decided in consensus, m' transitions to stage s_3. By Lemma 4.5.4, m' can do so at most once and consequently r adds an infinite number of different messages m'' to $PENDING_r$ such that $m''.stage = s_1 \wedge |m''.dst| > 1 \wedge m''.ts < m.ts_p^{s_3}$, a contradiction to (c-i). □

Proposition 4.5.2 (Uniform Agreement) *For any message m, if a process p A-Delivers m, then all correct processes $q \in m.dst$ eventually A-Deliver m.*

Proof: We first show that eventually $m \in PENDING_q$. There are two cases to consider, either (a) $|m.dst| = 1$ or (b) $|m.dst| > 1$. In both cases, because p A-Delivers m, there exists a consensus instance k such that p decides on m in k with $m.stage = s_0$. By Lemma 4.5.2 and by the uniform agreement of consensus, all correct processes $q \in group(p)$ eventually decide on m in k. Therefore, q adds m to $PENDING_q$ at line 30. This shows the claim for case (a). In case (b), because there is at least one correct process per group, there exists at least one process $r \in group(p)$ that sends (TS, m) to all processes in $(m.dst \setminus group(p))$

at line 24. Therefore, because links are quasi-reliable, all correct processes $q \in (m.dst \setminus group(p))$ eventually receive that message and add m to $PENDING_q$ at line 13 if $m \notin PENDING_q \cup ADELIVERED_q$. Note that if $m \in PENDING_q$, then obviously there is a time at which $m \in PENDING_q$.

By Lemma 4.5.8, m eventually reaches stage s_3 on q. By Lemma 4.5.9, q eventually stops adding messages m' to $PENDING_q$ such that $m'.ts \leq m.ts_q^{s_3}$. By Lemma 4.5.8, all such messages m' eventually reach stage s_3 and are removed from $PENDING_q$. Therefore, q eventually A-Delivers m. □

Proposition 4.5.3 (Validity) *If a correct process p A-MCasts m, then all correct processes $q \in m.dst$ eventually A-Deliver m.*

Proof: We first prove that q eventually adds m to $PENDING_q$. By the properties of Reliable Multicast and because p is correct, all correct processes $q \in m.dst$ R-Deliver m and add m to $PENDING_q$ at line 13 if $m \notin PENDING_q \cup ADELIVERED_q$. Notice that if $m \in ADELIVERED_q$, then obviously q A-Delivered m and Proposition 4.5.3 holds. By Lemma 4.5.8, m eventually reaches stage s_3 on q. By Lemma 4.5.9, q eventually stops adding messages m' to $PENDING_q$ such that $m'.ts \leq m.ts_q^{s_3}$. By Lemma 4.5.8, all such messages m' eventually get to stage s_3 and are removed from $PENDING_q$. Therefore, q eventually A-Delivers m. □

Lemma 4.5.10 *For any message m and any two processes p and q such that p and q A-Deliver m, $m.ts_p^{s_3} = m.ts_q^{s_3}$.*

Proof: There are two cases to consider: either (a) $|m.dst| = 1$ or (b) $|m.dst| > 1$.

- In case (a), by the uniform agreement property of consensus and by Lemma 4.5.4, all processes in p's group decide on m in the same consensus instance k and only in k. Therefore, p and q set $m.ts$ to the same value at line 29.

- In case (b), by the uniform agreement of consensus and by Lemma 4.5.4 for all groups $g \in m.dst$, all processes q in group g decide on m such that $m.stage = s_2$ in the same and only consensus instance k and send the same timestamp at line 24. Therefore, by line 35 and line 39, $m.ts_p^{s_3}$ and $m.ts_q^{s_3}$ are set to the same value. □

Lemma 4.5.11 *For any two messages m_1, m_2, and any two processes p and q such that $\{p,q\} \subseteq m_1.dst \cap m_2.dst$, if p A-Delivers m_1, q A-Delivers m_2, and $(m_1.ts_p^{s_3}, m_1.id) < (m_2.ts_q^{s_3}, m_2.id)$, then q A-Delivers m_1 before m_2.*

Proof: Because there is at least one correct process per group and by Proposition 4.5.2, there exists a correct process $r \in group(q)$ that A-Delivers m_1 and m_2. Let k_{m_1} and k_{m_2} respectively be the first consensus instance in which $group(q)$ decides on m_1 and the last consensus instance in which $group(q)$ decides on m_2 (the decision of instance k_{m_2} is such that m_2's stage is either s_0 or s_2). We first show that when q decides in instance k_{m_2}, $m_1 \in PENDING_q$.

Suppose, by way of contradiction, that it is not the case. Hence, $k_{m_1} > k_{m_2}$ (otherwise, $m_1 \in PENDING_q$ when q decides in k_{m_2}). From the algorithm, $m_1.ts_r^{s_3} \geq k_{m_1}$. From line 31, after r decides in instance k_{m_2} $K_r > m_2.ts_r^{s_3}$. Since $k_{m_1} > k_{m_2}$, $k_{m_1} > m_2.ts_r^{s_3}$. Therefore, since $m_1.ts_r^{s_3} \geq k_{m_1}$, $m_1.ts_r^{s_3} > m_2.ts_r^{s_3}$. By Lemma 4.5.10, $m_1.ts_p^{s_3} = m_1.ts_r^{s_3}$ and $m_2.ts_q^{s_3} = m_2.ts_r^{s_3}$. Consequently, $(m_1.ts_p^{s_3}, m_1.id) > (m_2.ts_q^{s_3}, m_2.id)$, a contradiction to the fact that $(m_1.ts_p^{s_3}, m_1.id) < (m_2.ts_q^{s_3}, m_2.id)$. Hence, (*) when q decides in instance k_{m_2}, $m_1 \in PENDING_q$.

Recall that by Lemma 4.5.10, $m_1.ts_r^{s_3} = m_1.ts_p^{s_3}$ and $m_2.ts_r^{s_3} = m_2.ts_q^{s_3}$. Hence, from the algorithm, m_1's timestamp on q can never be bigger than $m_2.ts_q^{s_3}$. Indeed, otherwise at some time t, r would set $m_1.ts$ to a higher value than $m_2.ts_q^{s_3}$, a contradiction to the fact that, on r, $m_1.ts$ is monotonically increasing with time and $(m_1.ts_p^{s_3}, m_1.id) < (m_2.ts_q^{s_3}, m_2.id)$. Hence, from (*) and the condition under which a message is A-Delivered (line 4), q A-Delivers m_1 before m_2. □

Proposition 4.5.4 (Uniform Prefix Order) *For any two messages m and m' and any two processes p and q such that $\{p,q\} \subseteq m.dst \cap m'.dst$, if p A-Delivers m and q A-Delivers m', then either p A-Delivers m' before m or q A-Delivers m before m'*

Proof: Since p A-Delivers m and q A-Delivers m', $m.ts_p^{def}$ and $m'.ts_q^{def}$ are defined. Either $(m.ts_p^{s_3}, m.id) < (m'.ts_q^{s_3}, m'.id)$ or $(m.ts_p^{s_3}, m.id) > (m'.ts_q^{s_3}, m'.id)$. By Lemma 4.5.11, either q A-Delivers m before m' or p A-Delivers m' before m. □

Lemma 4.5.12 *For any two messages m_1 and m_2, $m_1 < m_2 \Rightarrow (m_1.ts^{s_3}, m_1.id) < (m_2.ts^{s_3}, m_2.id)$.*

Proof: Notice that in the proof below, we use the fact that, by definition, for any two messages m_1 and m_2, $m_1.ts^{s_3} < m_2.ts^{s_3} \Rightarrow (m_1.ts^{s_3}, m_1.id) < (m_2.ts^{s_3}, m_2.id)$. Let p be the process that A-Delivers m_1 before m_2. At the time m_1 is A-Delivered, either (a) $m_2 \in PENDING_p$ or (b) $m_2 \notin PENDING_p$.

- In case (a), $(m_1.ts^{s_3}, m_1.id) < (m_2.ts^{s_3}, m_2.id)$ holds trivially by the condition of line 4.

- In case (b), because m_2 is not in $PENDING_p$ at the time m_1 is A-Delivered, a message is removed from this set only after it has been A-Delivered (line 7), and m_2 is A-Delivered after m_1, (*) m_2 has not yet been added to $PENDING_p$ either at line 13 or at line 30. Since K increases after each consensus instance, if $|m_2.dest| = 1$, $m_1.ts^{s_3} < m_2.ts^{s_3}$. If $|m_2.dest| > 1$, before m_2 reaches stage s_3 on p, p executed line 18 such that $m_2 \in msgSet' \wedge m_2.stage = s_0$ and m_2 transitions to stage s_1 at line 23. Therefore, since K increases after each consensus instance and because of (*), at the time m_2 reaches stage s_1, $m_1.ts^{s_3} < m_2.ts$. By line 35 or line 39 we have that $m_2.ts^{s_3}$ is equal to the maximum of all timestamps received, therefore $m_1.ts^{s_3} < m_2.ts^{s_3}$. □

Proposition 4.5.5 (Uniform Acyclic Order) *The relation $<$ is acyclic.*

Proof: Suppose, by way of contradiction, that the relation is cyclic. Therefore, there exists two messages m_1 and m_2 such that $m_1 < ... < m_2 < ... < m_1$. By Lemma 4.5.12, we have $(m_1.ts^{s_3}, m_1.id) < (m_2.ts^{s_3}, m_2.id)$ and $(m_2.ts^{s_3}, m_2.id) < (m_1.ts^{s_3}, m_1.id)$. There are two cases to consider, either (a) $m_1.ts^{s_3} = m_2.ts^{s_3}$ or (b) not.

- In case (a), $m_1.id < m_2.id < m_1.id$, a contradiction.
- In case (b), $m_1.ts^{s_3} < m_2.ts^{s_3} < m_1.ts^{s_3}$, a contradiction. □

4.5.2 The Proof of Algorithm $\mathscr{A}^{dv}_{bcast}$

Definition 4.5.3 *We define $msgsToADel_p^k$ as the value of set $msgsToADel_p$ after process p executed line 18 when $K_p = k$. If process p does not execute line 18 when $K_p = k$, $msgsToADel_p^k = \perp$.*

Lemma 4.5.13 *For any k, any two processes p and q such that group(p) =group(q) and any two messages $(k, msgSet'_p)$ and $(k, msgSet'_q)$ respectively sent by p and q at line 15, $msgSet'_p = msgSet'_q$.*

Proof: Follows directly from the uniform agreement property of consensus. □

Lemma 4.5.14 *For any two processes p and q and any k, if p and q execute line 18 when $K_p = K_q = k$, then $msgsToADel_p^k = msgsToADel_q^k$.*

Proof: From the condition of line 16, p and q received a message (k, -) from a process in each group different from $group(p)$ and $group(q)$. By Lemma 4.5.13, each two messages $(k, msgSet'_r)$ and $(k, msgSet'_s)$ coming from processes r and s that are in the same group are such that $msgSet'_r = msgSet'_s$. There are two cases to consider, either (a) $group(p) = group(q)$ or (b) not.

- In case (a), by the uniform agreement property of consensus, p and q add the same set of messages to *Msgs* at line 17. Therefore, $msgsToADel^k_p = msgsToADel^k_q$.
- In case (b), let $msgSet'_p$ and $msgSet'_q$ be the set of messages that p and q respectively add to *Msgs* at line 17. By the condition of line 16, p received a message (k, $msgSet_1$) from a process in $group(q)$ and q received a message (k, $msgSet_2$) from a process in $group(p)$. Because processes send the same set of messages at line 15 that they add to *Msgs* at line 17, by Lemma 4.5.13, $msgSet'_p = msgSet_2$ and $msgSet'_q = msgSet_1$. Therefore, $msgsToADel^k_p = msgsToADel^k_q$. □

Proposition 4.5.6 (Uniform Integrity) *For any process p and any message m, (a) p A-Delivers m at most once and (b) only if m was previously A-BCast.*

Proof:

- (a) Let k be the value of K_p the first time p A-Delivers m. Consequently, $m \in msgsToADel^k_p$ (notice that m can only appear once in *msgsToADel* because it is a set). By Lemma 4.5.14, all processes q that execute line 18 when $K_q = k$ are such that $m \in msgsToADel^k_q$. Consequently, since processes add m to *ADELIVERED* at line 20 after A-Delivering m, no process proposes m to a consensus instance $k' > k$. Therefore, there exists no $k' > k$ such that $m \in msgsToADel^{k'}_p$ and p never A-delivers m again.
- (b) Follows directly from the algorithm. □

Proposition 4.5.7 (Uniform Agreement) *For any message m, if a process p A-Delivers m, then all correct processes q eventually A-Deliver m.*

Proof: If p A-Delivers m, then there exists a k such that $m \in msgsToADel^k_p$. Thus, p decided in consensus instance k. By Lemma 4.5.15, every correct process q eventually decides in consensus instance k and executes line 18 when $K_q = k$. By Lemma 4.5.14, $m \in msgsToADel^k_q$ and therefore q A-Delivers m. □

Lemma 4.5.15 *For any process p and any k,*

- *(a) if p decides in consensus instance k, then all correct processes q eventually decide in instance k.*
- *(b) p does not wait forever at line 16 when $K_p = k$.*

Proof: We proceed by simultaneous induction on (a) and (b).

- Base step ($k = 1$):
  - (a) There are two cases to consider: either (a-1) $q \in group(p)$ or (a-ii) not.

    (a-1) Variable K is initialized to 1. Therefore by the uniform agreement property of consensus, all correct processes q eventually decide in instance 1.

    (a-2) By (a-1) and because there is at least one correct process per group, there is at least one correct process $r \in group(p)$ that sends a message (1, -) to all processes $q \notin group(p)$ at line 15. Because r is correct and links are quasi-reliable, all correct processes q eventually receive that message. Thus, eventually, $Barrier_q \geq 1$. Consequently, q proposes a value in instance 1 (if q has not decided yet in instance 1) and by the termination property of consensus, q eventually decides in instance 1.
  - (b) Suppose, by way of contradiction, that p waits forever at line 16. Consequently, p is correct. By (a), all correct processes q eventually decide in instance 1. After deciding in instance 1, correct processes q send a (1, -) message to all processes not in $group(q)$. Because there is at least one correct process per group, p is correct, and links are quasi-reliable, p eventually receive this message from a process in every group different from $group(p)$ and stops waiting at line 16, a contradiction.
- Induction step: Suppose that (a) and (b) hold for $k-1$ we prove that they hold for k.
  - (a) There are two cases to consider either (a-1) $q \in group(p)$ or (a-ii) not.

    (a-1) From the induction hypotheses, q eventually decides in instance $k-1$. Thus, eventually, $K_q = k$. Therefore by the uniform

agreement property of consensus, q eventually decide in instance k.

(a-2) By (a-1) and because there is at least one correct process per group, there is at least one correct process $r \in group(p)$ that sends a message (k, -) to all processes $q \notin group(p)$ at line 15. Because r is correct and links are quasi-reliable, all correct processes q eventually receive that message. Thus, eventually, $Barrier_q \geq k$. By the induction hypotheses, q decides in instance $k-1$ and does not wait forever at line 16 when $K_q = k-1$. Consequently, q proposes a value in instance k (if q has not decided yet in instance k) and by the termination property of consensus, q eventually decides in instance k.

– (b) The same argument as in the base step of (b) is used, where every occurrence of "1" is replaced by "k". □

Lemma 4.5.16 *For any group g and any message m, if there is a time after which all correct processes p in g are such that $m \in RDELIVERED_p$, then all correct processes eventually A-Deliver m.*

Proof: Suppose, by way of contradiction, that there exists a correct process that never A-Delivers m. By Proposition 4.5.7, no correct process A-Delivers m. Therefore, $m \in RDELIVERED_p \setminus ADELIVERED_p$ eventually forever. Consequently, by the termination property of consensus and Lemma 4.5.15, processes p execute an infinite number of consensus instances. Let t be the time at which all faulty processes have crashed. After t, consensus proposals in g always contain m, and thus, by the uniform integrity and uniform agreement properties of consensus, processes p eventually decide on m in an instance k.

Consequently, (*) processes p send a $(k, msgSet'_p)$ message such that $m \in msgSet'_p$. Because there is at least one correct process in each group and links are quasi-reliable, all correct processes $r \in \Pi$ eventually receive that message and eventually $Barrier_r \geq k$. By the termination property of consensus and by Lemma 4.5.15, there exists a time at which processes r have executed line 18 when $K_r = k$. Therefore, by the condition of line 16 and (*), $m \in msgsToADel_r^k$ and thus, all correct processes eventually A-Deliver m, a contradiction. □

Proposition 4.5.8 (Validity) *If a correct process p A-BCasts m, then all correct processes eventually A-Deliver m.*

Proof: By the validity property of reliable multicast, all correct processes q in group(p) eventually R-Deliver m and add m to $RDELIVERED_q$. Therefore, by Lemma 4.5.16, all correct processes eventually A-Deliver m. □

Definition 4.5.4 *We define the round of a message m as the value k such that there exists a process p with $m \in msgsToADel_p^k$. If m is never A-Delivered by any process, $round(m) = \perp$.*[9]

Proposition 4.5.9 (Uniform Prefix Order) *For any two messages m and m' and any two processes p and q, if p A-Delivers m and q A-Delivers m', then either p A-Delivers m' before m or q A-Delivers m before m'.*

Proof: Either (a) $round(m) < round(m')$, (b) $round(m) = round(m')$, or (c) $round(m) > round(m')$. We show that in each one of the three cases, either p A-Delivers m' before m or q A-Delivers m before m'.

- In case (a), $m \in msgsToADel_p^{round(m)}$. Since K_q is monotonically increasing with time and $round(m) < round(m')$, q executes line 19 when $K_q = round(m)$ before executing the same line when $K_q = round(m')$. By Lemma 4.5.14, $msgsToADel_p^{round(m)} = msgsToADel_q^{round(m)}$. Therefore, q A-Delivers m before m'.

- In case (b), let k be $round(m) = round(m')$. From the algorithm, $m \in msgsToADel_p^k$ and $m' \in msgsToADel_q^k$. By Lemma 4.5.14, $msgsToADel_p^k = msgsToADel_q^k$. Since p and q A-Deliver the messages of round k in the same deterministic order, either p A-Delivers m' before m or q A-Delivers m before m'.

- In case (c), a similar argument as in (a) can be used to show that p A-Delivers m' before m. □

Lemma 4.5.17 *If there exists a time after which no message is A-BCast, then for any correct process p, eventually $(RDELIVERED_p \setminus ADELIVERED_p) = \emptyset$ forever.*

Proof: If there exists a time after which no message is A-BCast, then there exists a time t after which no message is R-MCast. Therefore, by the uniform integrity of reliable multicast, p only R-Delivers a finite number of messages and thus, there exists a time after which no message is added to $RDELIVERED_p$. We now prove that for any message $m \in RDELIVERED_p$, eventually $m \in ADELIVERED_p$. Since $m \in RDELIVERED_p$, p R-Delivered m. By the agreement property of reliable multicast and because p is correct, all correct processes $q \in group(p)$ eventually R-Deliver m. By Lemma 4.5.16, all correct processes eventually A-Deliver m and therefore eventually $m \in ADELIVERED_p$. □

[9] Note that by Proposition 4.5.6 and Lemma 4.5.14, for any message m, round(m) is uniquely defined.

Lemma 4.5.18 *If there exists a time after which no message is A-BCast, then there exists a $Barrier_{max}$ such that for all correct processes p, $Barrier_p < Barrier_{max}$.*

Proof: By Lemma 4.5.17, for all correct processes p, eventually ($RDELIVERED_p \setminus ADELIVERED_p$) $= \emptyset$ forever. Let t_p be the earliest time at which p executes line 20 such that after executing line 20, ($RDELIVERED_p \setminus ADELIVERED_p$) $= \emptyset$ forever, and let k_p be the value of K_p at time t_p. We first prove that there exists a k such that for all p, $k_p = k$. Suppose, by way of contradiction, that there exist correct processes q, r such that $k_q > k_r$. Let m be the last message q A-Delivers in instance k_q (such an m exists by the definition of k_q). Before A-Delivering m, q sends a message (k_q, $msgSet'_q$) such that $m \in msgSet'_q$ to all. Because q and r are correct and links are quasi-reliable, r eventually receives this message and thus eventually $Barrier_r \geq k_q$. By the termination property of consensus and Lemma 4.5.15, r eventually decides in consensus instance k_q. Consequently, r eventually executes line 18 when $K_r = k_q$. By Proposition 4.5.6, r A-Delivers m only once and therefore $k_r \geq k_q$, a contradiction.
Now suppose, by way of contradiction, that there exists a process q such that $Barrier_q$ keeps on increasing. Variable $Barrier_q$ is increased either (a) at line 23 or (b) line 10.

- From the definition of k_p, for every $k > k_p$ such that $msgsToADel_p^k \neq \perp$, $msgsToADel_p^k = \emptyset$. Therefore, q does not increase $Barrier_q$ at line 23 when $K_p > k_p$, a contradiction.
- From (a), there exists no process r such that $Barrier_r > k_p + 1$, and thus q does not receive a (k, -) message at line 8 such that $k > k_p + 1$, a contradiction. □

Proposition 4.5.10 (Quiescence) *If there exists a time after which no message is A-BCast, then eventually all processes stop sending messages.*

Proof: Messages are sent either (a) at line 5 (reliable multicast), (b) at line 12 (consensus), or (c) at line 15 (send). Obviously, only correct processes can send messages forever. Consequently, proving that eventually all correct processes stop executing these lines is enough to show that eventually processes stop sending messages.[10]

[10]Notice that we here consider consensus and reliable multicast algorithms that are *halting*, i.e., in all runs of the algorithms, there is a time after which all processes stop taking steps and thus only a finite number of messages is sent. Halting algorithms for consensus and reliable multicast can be found in [54] and [29] respectively.

- (a) If there exists a time after which no message is A-BCast, eventually no (correct) process executes line 5 anymore.

- (b) From Lemmata 4.5.17 and 4.5.18, for every process p, the condition of line 11 eventually evaluates to false forever. Therefore, p only executes a finite number of times line 12.

- (c) From (b) and the uniform integrity of consensus, p decides in only a finite number of consensus instances and therefore p executes a finite number of times line 15. □

4.5.3 The Proof of Algorithm $\mathscr{A}_{ng}^{dv}$

Definition 4.5.5 *We define $gMsgs_p^k$ as the value of variable gMsgs on p after p executes line 17 in round k. If p does not execute line 17 in round k, then $gMsgs_p^k$ is undefined.*

Lemma 4.5.19 *For any message m, any two processes p and q such that $\{p,q\} \subseteq m.dst$, and any k, if $gMsgs_p^k$ and $gMsgs_q^k$ are both defined, then $m \in gMsgs_p^k \iff m \in gMsgs_q^k$.*

Proof: Let g be the group from which m was A-MCast.

- ($\Rightarrow$) From the algorithm, there exists a members r of g that decide on m in consensus instance k and p receives m from r. From the uniform agreement property of consensus, (*) all members of g that decide in consensus instance k decide on m. Therefore, if $q \in g$, then $m \in gMsgs_q^k$. Otherwise, from (*), q receives m from some member of g, and thus, $m \in gMsgs_q^k$.

- ($\Leftarrow$) A similar argument as in $\Rightarrow$ is used.

Proposition 4.5.11 (Uniform Integrity) *For any process p and any message m, (a) p A-Delivers m at most once, and (b) only if $p \in m.dst$ and (c) m was previously A-MCast.*

Proof:

- (a) Process p A-Delivers m either (a-i) at line 12 or (a-ii) at line 18.

  - In case (a-i), m was A-MCast by a process in $group(p)$ and $m.dst = \{group(p)\}$. Let k be the round in which p A-Delivers m for the first time. By the uniform agreement property of consensus, in round k, all processes q in $group(p)$ that decide on consensus, decide on the same set of messages $msgs$. From the algorithm, $m \in msgs$. Consequently, all q A-Deliver m in round k for the first time. Moreover, all q add m to *Decided* at line 10 in round k. Therefore, no process in $group(p)$ proposes m to consensus in a round bigger than k, and p does not A-Deliver m a second time.
  - In case (a-ii), let g be the group from which m was A-MCast, and let k be the first round in which a member of g decides on m in consensus. From the uniform agreement property of consensus, all members q of g that decide in consensus instance k decide on m. Hence, from the algorithm, all q add m to variable *Decided* at line 10 in round k and no process in g proposes m to consensus in a round $k' > k$. Therefore, there exists no $k' > k$ such that $m \in gMsgs_p^{k'}$ and p does not A-Deliver m a second time.

- (b) follows directly from the algorithm.
- (c) follows directly from the algorithm. □

Proposition 4.5.12 (Uniform Prefix Order) *For any two messages m and m' and any two processes p and q such that $\{p,q\} \subseteq m.dst \cap m'.dst$, if p A-Delivers m and q A-Delivers m', then either p A-Delivers m' before m or q A-Delivers m before m'.*

Proof: Let k and k' be the rounds in which p A-Delivers m and q A-Delivers m' respectively. Either (a) $k < k'$, (b) $k = k'$, or (c) $k > k'$.

- In case (a), either p A-Delivers m (a-i) at line 12 or (a-ii) at line 18.
  - In case (a-i), $m.dst = \{group(p)\}$ and $group(p) = group(q)$. Since $k < k'$ and q A-Delivers m' in round k', q decides in instance k of consensus. Because p A-Delivers m at line 12 in round k, in consensus instance k, p decides on a set of messages $msgs$ such that $m \in msgs$. From the uniform agreement property of consensus, q decides on $msgs$ in consensus instance k. Therefore, q A-Delivers m before m'.
  - In case (a-ii), $m \in MsgBundle_p^k$. Since $k < k'$, $gMsgs_q^k$ is defined. By Lemma 4.5.19, $m \in gMsgs_q^k$. Therefore, q A-Delivers m before m'.

- In case (b), either (b-i) both m and m' are A-Delivered at line 12, (b-ii) both m and m' are A-Delivered at line 18, or (b-iii) m and m' are not A-Delivered at the same line.

  - In case (b-i), $m.dst = m'.dst = \{group(p)\}$. Moreover, in consensus instance k, p and q decide on sets $msgs$ and $msgs'$ respectively such that $m \in msgs$ and $m' \in msgs'$. By the uniform agreement property of consensus, $msgs = msgs'$. Therefore, since messages in $msgs$ are A-Delivered at line 12 in a deterministic order, either p A-Delivers m' before m or q A-Delivers m before m'.
  - In case (b-ii), $m \in gMsgs_p^k$ and $m' \in gMsgs_q^k$. By Lemma 4.5.19, $m' \in gMsgs_p^k$ and $m \in gMsgs_q^k$. Therefore, since messages are A-Delivered in a deterministic order at line 18, either p A-Delivers m' before m or q A-Delivers m before m'.
  - In case (b-iii), either p A-Delivers m (b-iii-*) at line 12 or (b-iii-**) at line 18.

    In case (b-iii-*), $m.dst = \{group(p)\}$ and in consensus instance k, p decides on a set of messages $msgs$ such that $m \in msgs$. Moreover, since q A-Delivers m' at line 18, q decides in consensus instance k. From the uniform agreement property of consensus, q decides on $msgs$. Therefore, q A-Delivers m before m'.

    In case (b-iii-**), the same argument as in (b-iii-*) is used where every occurrence of m, m', p, and q are respectively replaced by m', m, q, and p.

- In case (c), a similar argument as in (a) is used where every occurrence of p, q, m, m', k, and k' are respectively replaced by q, p, m', m, k', and k. □

Lemma 4.5.20 *For any message m, and any two processes p and q, if p and q A-Deliver m, then they do so in the same round.*

Proof: Let k be the round in which p A-Delivers m. There are two cases to consider, either (a) m is local or (b) m is global.

- In case (a), from Proposition 4.5.11, p A-Delivers m once, and thus, k is uniquely defined. From the uniform agreement property of consensus and since q A-Delivers m, q decides on m in instance k. From Proposition 4.5.11 q only A-Delivers m once, and thus k is also uniquely defined on q.

- In case (b), a similar argument as in (a) using Lemma 4.5.19 is used. □

Proposition 4.5.13 (Uniform Acyclic Order) *The relation $<$ is acyclic.*

Proof: Suppose, by way of contradiction, that the relation is cyclic. Therefore, there exist two messages m_1 and m_2 such that (*) $m_1 < ... < m_2 < ... < m_1$. Let m_a and m_b be two messages such that $m_1 < m_a$ and $m_b < m_1$. Note that $m_a = m_2 = m_b$ is possible. From the definition of $<$, there exist processes p and q such that p A-Delivers m_1 before m_a and q A-Delivers m_b before m_1. From Lemma 4.5.20, (**) p and q A-Deliver m_1 in the same round k. From the algorithm, it is obvious that for any two messages m and m', if $m < m'$, then the process r that A-Delivers m before m' A-Delivers m in some round k and m' in some round k' such that $k \leq k'$. Consequently, from (*) and (**), the process that A-Delivers m_2 does so in round k. Either (a) m_1 is local or (b) m_1 is global. We show that both cases lead to a contradiction.

- In case (a), if m_2 is local, then m_1 should appear twice in variable *lMsgs*, a contradiction to the fact that it is a set. Otherwise, if m_2 is global, then from (*), m_1 should be global as well, a contradiction.

- In case (b), m_2 cannot be local as it would contradict (*). Otherwise, if m_2 is global, then m_1 should appear twice in variable *gMsgs*, a contradiction to the fact that this variable is a set. □

Proposition 4.5.14 (Uniform Agreement) *For any message m, if a process p A-Delivers m, then all correct processes $q \in m.dst$ eventually A-Deliver m.*

Proof: Let k be the round in which p A-Delivers m and let g be the group from which m is A-MCast. Either (a) $m.dst = \{g\}$ or (b) $m.dst \neq \{g\}$.

- In case (a), in consensus instance k, p decides on a set of messages *msgs* such that $m \in msgs$. Since groups are correct, q is correct, and links are quasi-reliable, q never waits forever at line 16. Hence, from the termination property of consensus, q eventually decides in consensus instance k. From the uniform agreement property of consensus, q decides on m in instance k. Therefore, q eventually A-Delivers m.

- In case (b), from the algorithm, $m \in gMsgs_p^k$. Since q is correct, groups are correct, and links are quasi-reliable, q does not wait forever at line 16. Hence, q eventually A-Delivers the global messages of round k at line 18 and thus $gMsgs_q^k$ is defined. By Lemma 4.5.19, $m \in gMsgs_q^k$. Therefore, q eventually A-Delivers m. □

Proposition 4.5.15 (Validity) *If a correct process p A-MCasts m, then all correct processes $q \in m.dst$ eventually A-Deliver m.*

Proof: Suppose, by way of contradiction, that there exists a correct process $r \in m.dst$ that never A-Delivers m. By Proposition 4.5.14, no correct process $q \in m.dst$ A-Delivers m (otherwise r would A-Deliver m). If p A-MCasts m, then p R-MCasts m to $group(p)$. Since p is correct, by the validity property of reliable multicast, all correct processes $s \in group(p)$ eventually R-Deliver m and add m to $Rdel_s$ at line 6. Let t be the time at which all faulty processes in g have crashed. Since no correct process $q \in m.dst$ A-Delivers m, after t, $m \in Rdel_s \setminus Decided_s$ forever. Hence, there exists a round k_1 such that for all $k' \geq k_1$, processes s always propose m to consensus instance k' and thus by the uniform integrity and uniform agreement properties of consensus, (*) processes in $group(p)$ decide on a set of messages $msgs$ such that $m \in msgs$ in consensus instance k'. Either (a) $m.dst = \{group(p)\}$ or (b) $m.dst \neq \{group(p)\}$.

- In case (a), from (*), r A-Delivers m in round k_1 at line 12, a contradiction.
- In case (b), from the algorithm, $m \in gMsgs_r^{k'}$. Therefore, r A-Delivers m in round k' at line 18, a contradiction. □

4.5.4 The Proof of Algorithm $\mathscr{A}_{ge}^{dt}$

In the proofs below, we denote the value of a variable V on a process p at time t as V_p^t.

Definition 4.5.6 *For any message m, we define $m.ts_p^{def}$ as the definitive timestamp of m on a process p, i.e., it is m's timestamp after p executed line 19 in Algorithm $\mathscr{A}_{ge}^{dt}$. If p never executes line 19 for m, then $m.ts_p^{def}$ is undefined. From the uniform agreement of global data computation, it is clear that for any two processes p and q such that $m.ts_p^{def}$ and $m.ts_q^{def}$ are defined, $m.ts_p^{def} = m.ts_q^{def}$. We thus sometimes write $m.ts^{def}$ for short.*

Proposition 4.5.16 (Uniform Integrity) *For any process p and any message m, (a) p A-Delivers m at most once, and (b) only if $p \in m.dst$ and (c) m was previously A-MCast.*

Proof:

- (a) Follows directly from the uniform integrity property of causal multicast and from the fact that a message is removed from $Pending_p$ after it has been A-Delivered.

- (b) Follows directly from the algorithm.
- (c) Process p A-Delivers m only if p C-Delivered m. From the uniform integrity property of causal multicast, m was C-MCast. Consequently, m was A-MCast. □

Lemma 4.5.21 *For any correct process p and any message m, if p C-Delivers m, then m eventually reaches stage s_2 on p.*

Proof: By the uniform agreement property of causal multicast, all correct processes $q \in m.dst$ eventually C-Deliver m and fork the consensus task for m. By the termination property of global data computation, q eventually decides and C-MCasts (ACK, $m.id, q$). By the strong completeness property of $\mathscr{P}$, eventually no faulty process is ever trusted by any correct process. Therefore, by the validity property of causal multicast, q eventually C-Delivers(ACK, $m.id$,-) for all processes in $\mathscr{P} \cap m.dst$. Therefore, m eventually reaches stage s_2 on q, and in particular on p. □

Lemma 4.5.22 *For any correct process p and any message m, if p C-Delivers m, then p eventually A-Delivers m.*

Proof: By Lemma 4.5.21, m eventually reaches stage s_2 on p. Consider the *transitive* relation on messages *in-pps* defined as follows: m_1 *in-pps* m_2 if and only if $m_1 \in m_2.pps$. Let $PPS(m)$ be the set of messages m' such that m' *in-pps* m. By Lemma 4.5.21, all $m' \in PPS(m)$ eventually reach stage s_2 on p. Because the identifiers of messages are unique, the relation $<$ on messages' timestamps and identifiers defines a total order. Hence, messages in $PPS(m)$ are delivered according to the order defined by $<$ and thus, since $|PPS(m)|$ is finite, p eventually A-Delivers m. □

Proposition 4.5.17 (Uniform Agreement) *If a process p A-Delivers a message m, then all correct processes $q \in m.dst$ eventually A-Deliver m.*

Proof: If p A-Delivers m, p C-Delivered m. By the uniform agreement property of causal multicast, all correct processes $q \in m.dst$ eventually C-Deliver m. By Lemma 4.5.22, q eventually A-Delivers m. □

Proposition 4.5.18 (Validity) *If a correct process p A-MCasts a message m, then eventually all correct processes $q \in m.dst$ A-Deliver m.*

Proof: If p A-MCasts m, then p C-MCasts m. By the validity property of causal multicast, all correct processes $q \in m.dst$ eventually C-Deliver m. By Lemma 4.5.22, q eventually A-Delivers m. □

Lemma 4.5.23 *For any two messages m_1, m_2, and any two processes p and q such that $\{p,q\} \subseteq m_1.dst \cap m_2.dst$, if p A-Delivers m_1, q A-Delivers m_2, and $(m_1.ts_p^{def}, m_1.id) < (m_2.ts_q^{def}, m_2.id)$, then q A-Delivers m_1 before m_2.*

Proof: Let t_1 and t_2 be the times at which p gathered ACK messages for m_1 and q gathered ACK messages for m_2 respectively. Either (a) $t_1 \leq t_2$ or (b) $t_1 > t_2$.

- In case (a), by the strong accuracy property of $\mathscr{P}$, p C-Delivers (ACK, $m_1.id, q$). Hence, q C-Delivers m_1 before t_1 (and t_2). Consequently, q adds m_1 to $Pending_q$ before A-Delivering m_2. At the time q computes m_2's potential predecessor set (line 23), either (a-i) $m_1 \in Pending_q$ or (a-ii) not.

  - In case (a-ii), because a message is removed from *Pending* only after being A-Delivered (line 6), q A-Delivers m_1 before m_2.
  - In case (a-ii), from line 23, $m_1 \in m_2.pps$. From line 4, q does not A-Deliver m_2 before m_1 reaches stage s_2. Since q A-Delivers m_2, m_1 reaches stage s_2 on q. By the uniform agreement property of global data computation, when m_1 reaches stage s_2 on q, $m_1.ts_q^{def} = m_1.ts_p^{def}$. Since $(m_1.ts_p^{def}, m_1.id) < (m_2.ts_q^{def}, m_2.id)$, $(m_1.ts_q^{def}, m_1.id) < (m_2.ts_q^{def}, m_2.id)$. Consequently, from the condition of line 4, q A-delivers m_1 before m_2.

- In case (b), by the strong accuracy property of $\mathscr{P}$, q C-Delivers (ACK, $m_2.id, p$).

  We now show that p C-Delivers m_1 before C-MCasting (ACK, $m_2.id, p$). Suppose, by way of contradiction, that (*) p does not C-Deliver m_1 before C-MCasting (ACK, $m_2.id, p$). Since p A-Delivers m_1, p C-Delivers m_1. From (*), p does so after p executes line 20 in the consensus task of m_2. Since p C-MCasts (ACK, $m_2.id, p$), p decides in m_2's global data computation instance. By the uniform agreement property of global data computation, (**) $m_2.ts_p^{def} = m_2.ts_q^{def}$ (line 19). Since p A-Delivers m_1, p decides in m_1's global data computation instance. By the uniform obligation property of global data computation, p decides on vector V such that $V[p] = v_p$. Because p sets TS_p to $\max(m_2.ts_p^{def} + 1, TS_p)$ at line 20,

from (*) and (**), $m_2.ts_q^{def} < v_p \leq m_1.ts_p^{def}$, a contradiction to the fact that $(m_1.ts_p^{def}, m_1.id) < (m_2.ts_q^{def}, m_2.id)$.

Consequently, p C-Delivers m_1 before C-MCasting (ACK, $m_2.id, p$). Hence, C-Mcast(m_1) $\rightarrow$ C-Deliver(m_1)$_p$ $\rightarrow$ C-MCast(ACK, $m_2.id, p$)$_p$ $\rightarrow$ C-Deliver(ACK, $m_2.id, p$)$_q$. Therefore, from the causal order property of causal multicast, q C-delivers m_1 and adds m_1 to $Pending_q$, before A-Delivering m_2. A similar argument as in (a) is then used to conclude the proof. □

Proposition 4.5.19 (Uniform Prefix Order) *For any two messages m and m' and any two processes p and q such that $\{p, q\} \subseteq m.dst \cap m'.dst$, if p A-Delivers m and q A-Delivers m', then either p A-Delivers m' before m or q A-Delivers m before m'.*

Proof: Since p A-Delivers m and q A-Delivers m', $m.ts_p^{def}$ and $m'.ts_q^{def}$ are both defined. Either $(m.ts_p^{def}, m.id) < (m'.ts_q^{def}, m'.id)$ or $(m.ts_p^{def}, m.id) > (m'.ts_q^{def}, m'.id)$. By Lemma 4.5.23, either q A-Delivers m before m' or p A-Delivers m' before m. □

Lemma 4.5.24 *For any two messages m_1 and m_2, $m_1 < m_2 \Rightarrow (m_1.ts^{def}, m_1.id) < (m_2.ts^{def}, m_2.id)$.*

Proof: In the proof below, we use the fact that, by definition, for any two messages m_1 and m_2, $m_1.ts^{def} < m_2.ts^{def} \Rightarrow (m_1.ts^{def}, m_1.id) < (m_2.ts^{def}, m_2.id)$. Let p be the process that A-Delivers m_1 before m_2. At the time m_1 is A-Delivered, either (a) $m_2 \in Pending_p$ or (b) $m_2 \notin Pending_p$.

- In case (a), $(m_1.ts^{def}, m_1.id) < (m_2.ts^{def}, m_2.id)$ holds trivially by the condition of line 4.
- In case (b), because m_2 is not in $Pending_p$ at the time m_1 is A-Delivered, a message is removed from this set only after it has been A-Delivered (line 6), and m_2 is A-Delivered after m_1, (*) m_2 has not yet been added to $Pending_p$ at the time p A-Delivers m_1. Since TS_p is set to a greater value than $m_1.ts^{def}$ after deciding in the global data computation instance of m_1, from (*), p proposes a bigger timestamp than $m_1.ts^{def}$ for m_2. Since p A-Delivers m_2, p decides in the global data computation instance of m_2. From the uniform obligation property of global data computation, p decides on a vector V that contains its proposal. Therefore, since the definitive timestamp of m_2 is the biggest value contained in V, $m_1.ts^{def} < m_2.ts^{def}$. □

Proposition 4.5.20 (Uniform Acyclic Order) *The relation $<$ is acyclic.*

Proof: Suppose, by way of contradiction, that the relation is cyclic. Therefore, there exists two messages m_1 and m_2 such that $m_1 < ... < m_2 < ... < m_1$. By Lemma 4.5.24, we have $(m_1.ts^{def}, m_1.id) < (m_2.ts^{def}, m_2.id)$ and $(m_2.ts^{def}, m_2.id) < (m_1.ts^{def}, m_1.id)$. There are two cases to consider, either (a) $m_1.ts^{def} = m_2.ts^{def}$ or (b) not.

- In case (a), $m_1.id < m_2.id < m_1.id$, a contradiction.
- In case (b), $m_1.ts^{def} < m_2.ts^{def} < m_1.ts^{def}$, a contradiction. □

4.5.5 The Proof of Algorithm $\mathscr{A}_{ng}^{dt}$

In the proof below, a message of round k concerning a group g is any G-BCast message of the form $(k, g, \text{-}, \text{-})$.

Definition 4.5.7

- *We define $MsgBundle_p^k$ as the value of variable MsgBundle on p before p executes line 25 in round k. If p does not execute line 25 in round k, then $globalMsgs_p^k$ is undefined.*
- *We define $View_p^k$ as the value of variable View on p after p executes line 28 in round k. If p does not execute line 28 in round k, then $View_p^k$ is undefined.*
- *We define LM_p^k as the last message process p removes from the sequence Gdelivered at line 20 before p computes the set of global messages of round k at line 25. If p never executes line 25 in round k, then LM_p^k is undefined.*

Lemma 4.5.25 *For any two processes p and q and any k:*

1. *if $MsgBundle_p^k$ and $MsgBundle_q^k$ are both defined, then $MsgBundle_p^k = MsgBundle_q^k$.*
2. *if $View_p^k$ and $View_q^k$ are both defined, then $View_p^k = View_q^k$.*

Proof: In the proof below, we denote as $groupsToAdd_p^k$ the value of variable *groupsToAdd* on process p before p executes line 25 in round k. We proceed by simultaneous induction on 1 and 2.

- Base step ($k = 1$):

1. We show that for any group g, $MsgBundle_p^1[g] = MsgBundle_q^1[g]$. Suppose, by way of contradiction, that (*) $MsgBundle_p^1[g] \neq MsgBundle_q^1[g]$. From the condition of line 17, either (a) $MsgBundle_p^1[g] = \emptyset$ or (b) $MsgBundle_p^1[g] \neq \emptyset$.
   - In case (a), since $MsgBundle_p[g]$ is initialized to $\perp$, the first message concerning g that p removes from the *Gdelivered* sequence is a message of the form (1, g,*remove*, -). Let m_p be this message. Since $MsgBundle_q[g]$ is initialized to $\perp$, the first message concerning g that q removes from the *Gdelivered* sequence is a message of the form (1, g,*msgBundle*, -). Let m_q be this message. From the uniform generalized order property of generic broadcast, either (a-i) p G-Delivers m_q before m_p or (a-ii) q G-Delivers m_p before m_q. We show that both (a-i) and (a-ii) lead to a contradiction.

     In case (a-i), from the algorithm, $MsgBundle_p^1 \neq \emptyset$, a contradiction to hypothesis (a).

     In case (a-ii), from the algorithm, $MsgBundle_q^1 = \emptyset$, a contradiction to (*).
   - In case (b), a similar argument as in (a) is used where every occurrence of *remove*, *msgBundle*, $\neq$, and $=$ are respectively replaced by *msgBundle*, *remove*, $=$, and $\neq$.
2. From 1, $MsgBundle_p^1 = MsgBundle_q^1$. Therefore, it is sufficient to show that $groupsToAdd_p^1 = groupsToAdd_q^1$. We prove that, for any group g, $g \in groupsToAdd_p^1 \Leftrightarrow groupsToAdd_q^1$.
   - ($\Rightarrow$) Process p G-Delivers a message of the form (-, g,*add*, -). Let m_p^{add-g} be this message. Suppose, by way of contradiction, that (*) there exists a group g in $groupsToAdd_p^1$ that is not in $groupsToAdd_q^1$. From the algorithm, LM_p^1 cannot be of the form (-, -,*add*, -). Therefore, (**) p G-Delivers m_p^{add-g} before LM_p^1 and LM_p^1 is either (a) of the form (1, g',*remove*, -) or (b) of the form (1, g',*msgBundle*, -) for some group g'.

     In case (a), from 1, $MsgBundle_p^1 = MsgBundle_q^1$, therefore the first message q G-Delivers concerning g' is a message of the form (1, g',*remove*, -). Let $m_q^{remove-g'}$ be this message. From the uniform generalized order of generic broadcast, either (a-i) p G-Delivers $m_q^{remove-g'}$ before m_p^{add-g} or (a-ii) q

G-Delivers m_p^{add-g} before $m_q^{remove-g'}$.

· In case (a-i), p G-Delivers a message concerning g' before LM_p^1. Therefore, from (**), LM_p^1 cannot concern g' ($MsgBundle_p^1[g']$ is locked before p removes LM_p^1 from the *Gdelivered* sequence), a contradiction.

· In case (a-ii), $g \in groupsToAdd_q^1$, a contradiction to (*).

In case (b), a similar argument as in (a) is used where every occurrence of *remove* is replaced by *msgBundle*.

– ($\Leftarrow$) A similar argument as in ($\Rightarrow$) is used where every occurrence of p and q are respectively replaced by q and p.

- Induction step: Suppose that Lemma 4.5.25 holds for $k-1$, we show that Lemma 4.5.25 also holds for k.

  1. We show that for any group g, $MsgBundle_p^k[g] = MsgBundle_q^k[g]$. From the induction hypothesis, $View_p^{k-1} = View_q^{k-1}$, therefore when p and q execute line 29 in round $k-1$, p and q respectively set $MsgBundle_p[g]$ and $MsgBundle_q[g]$ either (a) to $\perp$ or (b) to $\emptyset$.
     – In case (a), a similar argument as in the base step of 1 is used where every occurrence of 1 is replaced by k.
     – In case (b), from the algorithm, $MsgBundle_p^k[g] = MsgBundle_q^k[g] = \emptyset$.
  2. A similar argument as in the base step of 2 is used where every occurrence of 1 is replaced by k. □

Proposition 4.5.21 (Uniform Integrity) *For any process p and any message m, (a) p A-Delivers m at most once, and (b) only if $p \in m.dst$ and (c) m was previously A-MCast.*

Proof:

- (a) Process p A-Delivers m either (a-i) at line 12 or (a-ii) at line 26.

  – In case (a-i), m was A-MCast by a process in $group(p)$ and $m.dst = \{group(p)\}$. Let k be the round in which p A-Delivers m for the first time. By the uniform agreement property of consensus, in round k, all processes q in $group(p)$ that decide on consensus, decide on the same set of messages *msgs*. From the algorithm, $m \in msgs$. Consequently,

all q A-Deliver m in round k for the first time. Moreover, all q add m to *Adelivered* at line 13 in round k. Therefore, no process in $group(p)$ proposes m to consensus in a round bigger than k, and p does not A-Deliver m a second time.

– In case (a-ii), let g be the group from which m was A-MCast. Moreover, let k be the first round in which p A-Delivers m. From the algorithm, (*) $MsgBundle_p^k[g] = msgs$ for some set of messages $msgs$ such that $m \in msgs$. By Lemma 4.5.25, for all processes q such that $MsgBundle_q^k$ is defined $MsgBundle_q^k[g] = msgs$. Consequently, all q that start round $k+1$ add m to *Adelivered* at line 27 in round k and no process in g proposes m to consensus in a round $k' > k$. Therefore, there exists no $k' > k$ such that $m \in MsgBundle_p^{k'}[g]$ and p does not A-Deliver m a second time.

- (b) follows directly from the algorithm.
- (c) follows directly from the algorithm. □

Proposition 4.5.22 (Uniform Prefix Order) *For any two messages m and m' and any two processes p and q such that $\{p,q\} \subseteq m.dst \cap m'.dst$, if p A-Delivers m and q A-Delivers m', then either p A-Delivers m' before m or q A-Delivers m before m'.*

Proof: Let k and k' be the rounds in which p A-Delivers m and q A-Delivers m' respectively. Either (a) $k < k'$, (b) $k = k'$, or (c) $k > k'$.

- In case (a), either p A-Delivers m (a-i) at line 12 or (a-ii) at line 26.

  – In case (a-i), $m.dst = \{group(p)\}$ and $group(p) = group(q)$. Since $k < k'$ and q A-Delivers m' in round k', q decides in instance k of consensus. Because p A-Delivers m at line 12 in round k, in consensus instance k, p decides on a set of messages $msgs$ such that $m \in msgs$. From the uniform agreement property of consensus, q decides on $msgs$ in consensus instance k. Therefore, q A-Delivers m before m'.

  – In case (a-ii), there exists a group g and a set of messages $msgs$ such that $m \in msgs$ and $MsgBundle_p^k[g] = msgs$. Since $k < k'$, $MsgBundle_q^k$ is defined. By Lemma 4.5.25, $MsgBundle_p^k[g] = MsgBundle_q^k[g]$. Therefore, q A-Delivers m before m'.

- In case (b), either (b-i) both m and m' are A-Delivered at line 12, (b-ii) both m and m' are A-Delivered at line 26, or (b-iii) m and m' are not A-Delivered at the same line.

  - In case (b-i), $m.dst = m'.dst = \{group(p)\}$. Moreover, in consensus instance k, p and q decide on sets $msgs$ and $msgs'$ respectively such that $m \in msgs$ and $m' \in msgs'$. By the uniform agreement property of consensus, $msgs = msgs'$. Therefore, since messages in $msgs$ are A-Delivered at line 12 in a deterministic order, either p A-Delivers m' before m or q A-Delivers m before m'.
  - In case (b-ii), there exist groups g and g' as well as sets of messages $msgs$ and $msgs'$ such that $m \in msgs$, $m' \in msgs'$, $MsgBundle_p^k[g] = msgs$, and $MsgBundle_q^k[g'] = msgs'$. By Lemma 4.5.25, $MsgBundle_p^k = MsgBundle_q^k$. Therefore, since messages are A-Delivered in a deterministic order at line 26, either p A-Delivers m' before m or q A-Delivers m before m'.
  - In case (b-iii), either p A-Delivers m (b-iii-*) at line 12 or (b-iii-**) at line 26.

    In case (b-iii-*), $m.dst = \{group(p)\}$ and in consensus instance k, p decides on a set of messages $msgs$ such that $m \in msgs$. Moreover, since q A-Delivers m' at line 26, q decides in consensus instance k. From the uniform agreement property of consensus, q decides on $msgs$. Therefore, q A-Delivers m before m'.

    In case (b-iii-**), the same argument as in (b-iii-*) is used where every occurrence of m, m', p, and q are respectively replaced by m', m, q, and p.

- In case (c), a similar argument as in (a) is used where every occurrence of p, q, m, m', k, and k' are respectively replaced by q, p, m', m, k', and k. □

Lemma 4.5.26 *For any message m, and any two processes p and q, if p and q A-Deliver m, then they do so in the same round.*

Proof: Let k be the round in which p A-Delivers m. There are two cases to consider, either (a) m is local or (b) m is global.

- In case (a), from Proposition 4.5.21, p A-Delivers m once, and thus, k is uniquely defined. From the uniform agreement property of consensus

and since q A-Delivers m, q decides on m in instance k. From Proposition 4.5.21 q only A-Delivers m once, and thus k is also uniquely defined on q.

- In case (b), a similar argument as in (a) using Lemma 4.5.25 is used. □

Proposition 4.5.23 (Uniform Acyclic Order) *The relation $<$ is acyclic.*

Proof: Suppose, by way of contradiction, that the relation is cyclic. Therefore, there exist two messages m_1 and m_2 such that (*) $m_1 < ... < m_2 < ... < m_1$. Let m_a and m_b be two messages such that $m_1 < m_a$ and $m_b < m_1$. Note that $m_a = m_2 = m_b$ is possible. From the definition of $<$, there exist processes p and q such that p A-Delivers m_1 before m_a and q A-Delivers m_b before m_1. From Lemma 4.5.26, (**) p and q A-Deliver m_1 in the same round k. From the algorithm, it is obvious that for any two messages m and m', if $m < m'$, then the process r that A-Delivers m before m' A-Delivers m in some round k and m' in some round k' such that $k \leq k'$. Consequently, from (*) and (**), the process that A-Delivers m_2 does so in round k. Either (a) m_1 is local or (b) m_1 is global. We show that both cases lead to a contradiction.

- In case (a), if m_2 is local, then m_1 should appear twice in variable *localMsgs*, a contradiction to the fact that it is a set. Otherwise, if m_2 is global, then from (*), m_1 should be global as well, a contradiction.

- In case (b), m_2 cannot be local as it would contradict (*). Otherwise, if m_2 is global, then m_1 should appear twice in variable *globalMsgs*, a contradiction to the fact that this variable is a set. □

Lemma 4.5.27 *For any correct process p and any k, p eventually A-Delivers the global messages of round k at line 26.*

Proof: We proceed by induction on k.

- Base step ($k = 1$): Suppose, by way of contradiction, that p never executes line 26 in round 1. Therefore, (*) there exists a group g such that $MsgBundle_p[g] \in \{\bot, \top\}$ forever in round 1. From the termination property of consensus, p eventually decides in consensus instance 1 and executes the while loop of lines 17-24. Hence, from (**), p never G-Delivers a message of the form (1, g, *type*, -) where *type* is equal to *remove* or *msgBundle* at line 35. Either (a) g is correct or (b) g is faulty.

- In case (a), since *View* is initialized to Γ, there exists a correct process q in g that G-BCasts a message of the form (1, g, *msgBundle*, -) at line 14. From the validity property of generic broadcast p eventually G-Delivers this message, a contradiction to (**).
- In case (b), from the strong completeness property of $\Diamond\mathscr{P}$, p eventually stops trusting processes in g and G-BCasts a message of the form (1, g, *remove*, -) at line 32. From the validity property of generic broadcast, p eventually G-Delivers this message, a contradiction to (**).

- Induction step: Suppose that Lemma 4.5.27 holds for $k-1$, we show that Lemma 4.5.27 also holds for k. From the induction hypothesis, p eventually starts consensus instance k. By the termination property of consensus, p eventually decides and executes the while loop of lines 17-24 in round k. Suppose, by way of contradiction, that (*) there exists a group g such that $MsgBundle_p[g] \in \{\bot, \top\}$ forever in round k. Hence, (**) p never G-Delivers a message of the form (k, g, *type*, -) where *type* is equal to *remove* or *msgBundle* at line 35. Either (a) $g \in View_p^{k-1}$ or (b) $g \notin View_p^{k-1}$.

  - In case (a), either (a-i) g is correct or (a-ii) g is faulty.

    In case (a-i), there exists a correct process $q \in g$. From hypothesis (a), $g \in View_p^{k-1}$. By Lemma 4.5.25, $View_p^{k-1} = View_q^{k-1}$ and thus $g \in View_q^{k-1}$. Therefore, q G-BCasts a message of the form (k, g, *msgBundle*, -) at line 14. From the validity property of generic broadcast p eventually G-Delivers this message, a contradiction to (**).

    In case (a-ii), from the strong completeness property of $\Diamond\mathscr{P}$, p eventually stops trusting processes in g and G-BCasts a message of the form (k, g, *remove*, -) at line 32. From the validity property of generic broadcast, p eventually G-Delivers this message, a contradiction to (**).
  - In case (b), p sets $MsgBundle_p[g]$ to $\emptyset$ at line 29 in round $k-1$. Therefore, there is a time at which $MsgBundle_p[g] \notin \{\bot, \top\}$ in round k, a contradiction to (*). □

Proposition 4.5.24 (Uniform Agreement) *For any message m, if a process p A-Delivers m, then all correct processes $q \in m.dst$ eventually A-Deliver m.*

Proof: Let k be the round in which p A-Delivers m and let g be the group from which m is A-MCast. Either (a) $m.dst = \{g\}$ or (b) $m.dst \neq \{g\}$.

- In case (a), in consensus instance k, p decides on a set of messages *msgs* such that $m \in msgs$. Since q is correct, by Lemma 4.5.27, q eventually A-Delivers the global messages of round $k-1$ at line 26. Consequently, q starts consensus instance k, and by the termination property of consensus, q decides in that instance. By the uniform agreement property of consensus, q decides on *msgs* in consensus instance k. Therefore, q eventually A-Delivers m.

- In case (b), from the algorithm, $MsgBundle_p^k[g] = msgs$ for some set of messages *msgs* such that $m \in msgs$. Since q is correct, by Lemma 4.5.27, q eventually A-Delivers the global messages of round k at line 26 and thus $MsgBundle_q^k$ is defined. By Lemma 4.5.25, $MsgBundle_p^k[g] = MsgBundle_q^k[g]$. Therefore, q eventually A-Delivers m. □

Lemma 4.5.28 *For any correct processes p and q, there exists a round k such that for all $k' \geq k$, $group(p) \in View_q^{k'}$.*

Proof: By the eventual strong accuracy of $\Diamond\mathscr{P}$, there is a time after which no process stops being trusted before it crashes. Since p is correct, there exists a time after which processes always trust p. Therefore, (*) there exists round k_{no-rmv} such that for all $k' \geq k_{no-rmv}$ no process G-BCasts a message of the form $(k', group(p), remove, \text{-})$. Since process p and processes q are correct, by Lemma 4.5.27, processes p and q execute an infinite number of rounds. From the algorithm, for any round k' such that $group(p) \notin View_p^{k'-1}$, p G-BCasts a message of the form $(\text{-}, group(p), add, \text{-})$. Since p is correct, by the validity property of generic broadcast all such messages are eventually G-Delivered by all correct processes. Hence, from (*), there exists a round $k \geq k_{no-rmv}$ such that $group(p)$ is in $View_p^k$ and $group(p)$ is never removed from $View_p$ anymore, i.e., for all $k' \geq k$, $group(p) \in View_p^k$. Thus, by Lemma 4.5.25, for any $k' \geq k$, $group(p) \in View_q^{k'}$. □

Proposition 4.5.25 (Validity) *If a correct process p A-MCasts m, then all correct processes $q \in m.dst$ eventually A-Deliver m.*

Proof: Suppose, by way of contradiction, that there exists a correct process $r \in m.dst$ that never A-Delivers m. By Proposition 4.5.24, no correct process $q \in m.dst$ A-Delivers m (otherwise r would A-Deliver m). If p A-MCasts m, then p

R-MCasts m to $group(p)$. Since p is correct, by the validity property of reliable multicast, all correct processes $s \in group(p)$ eventually R-Deliver m and add m to $Rdelivered_s$ at line 7. Let t be the time at which all faulty processes in g have crashed. Since no correct process $q \in m.dst$ A-Delivers m, after t, $m \in Rdelivered_s \setminus Adelivered_s$ forever. Hence, there exists a round k_1 such that for all $k' \geq k_1$, processes s always propose m to consensus instance k' and thus by the uniform integrity and uniform agreement properties of consensus, (*) processes in $group(p)$ decide on a set of messages $msgs$ such that $m \in msgs$ in consensus instance k'. Either (a) $m.dst = \{group(p)\}$ or (b) $m.dst \neq \{group(p)\}$.

- In case (a), from (*), r A-Delivers m in round k_1 at line 12, a contradiction.
- In case (b), by Lemma 4.5.28, there exists a round k_2 such that for any $k' \geq k_2$, $group(p) \in View_q^{k'}$. Hence, from (*), there exists a round $k' = \max(k_2, k_1)$ such that: (1) $group(p) \in View_q^{k'}$ and (2) processes in $group(p)$ G-BCast a message at line 14 of the form $(k', group(p), msgBundle, msgs)$ such that $m \in msgs$. Therefore, r A-Delivers m in round k' at line 26, a contradiction. □

Chapter 5

Atomic Multicast in Large Networks: Performance Evaluation

> Then the Warrior realizes that these repeated experiences have but one aim: to teach him what he does not want to learn.
>
> *Paulo Coelho*

Although most multicast algorithms proposed in the literature are genuine (e.g., [28; 21; 52]), we showed in Section 4.2 that genuineness is an expensive property: no genuine atomic multicast algorithm can deliver global messages in one inter-group message delay,[1] a limitation that is not imposed on non-genuine multicast algorithms. Therefore, when choosing a multicast algorithm, it seems natural to question *the circumstances under which a genuine algorithm is more efficient than a non-genuine algorithm*.

To answer this question, we experimentally evaluate the performance of existing latency-optimal multicast algorithms. More specifically, we select the genuine and non-genuine protocols $\mathscr{A}_{ge}^{dv}$ and $\mathscr{A}_{ng}^{dv}$. As part of the empirical study, we assess the scalability of the protocols by varying the number of groups, the proportion of global messages, and the load, i.e., the frequency at which messages are multicast. The results suggest that the genuineness of multicast is interesting only in large and highly loaded systems; in all the other considered scenarios the non-genuine protocol $\mathscr{A}_{ng}^{dv}$ outperforms the optimal genuine algorithm.

[1] This lower bound is tight since $\mathscr{A}_{ge}^{dv}$ and the algorithm in [28] can deliver messages in two inter-group delays.

To complete our study, we measure the overhead of the disaster-tolerant and latency-optimal multicast protocol $\mathcal{A}_{ng}^{dt}$ and observe that although in general it is more costly than the two other implemented protocols, it matches the performance of the genuine algorithm when there are few groups.

We start our study by identifying a *convoy effect* in multicast algorithms that may delay the delivery of local messages by as much as the latency of global messages. All multicast algorithms we are aware of suffer from this effect. We propose techniques to reduce this effect in Algorithms $\mathcal{A}_{ge}^{dv}$, $\mathcal{A}_{ng}^{dt}$ and $\mathcal{A}_{ng}^{dv}$. Although simple, these techniques decrease the delivery latency of local messages by as much as two orders of magnitude.

5.1 The Convoy Effect

The convoy effect refers to the phenomenon by which the delivery of a *local message* m_l is delayed by a *global message* m_g. This effect may happen for different reasons. In timestamp-based protocols, it may occur if m_g has a smaller timestamp than m_l's and m_g hasn't been A-Delivered yet; in round-based protocols, the phenomenon may happen if m_l cannot be proposed to consensus as the message bundles of the current round are being exchanged.

We observed that all protocols surveyed in Section 4.1 suffer from this undesired behavior more or less severely but can be modified to reduce its effects, at least partially. Below, we illustrate how the convoy effect happens in the latency-optimal algorithms $\mathcal{A}_{ge}^{dv}$, $\mathcal{A}_{ng}^{dv}$, and $\mathcal{A}_{ng}^{dt}$, and propose techniques to reduce it.

The timestamp-based algorithm $\mathcal{A}_{ge}^{dv}$. Consider the following scenario in which a global message m_g delays the delivery of a local message m_l. Messages m_g and m_l are addressed to groups $\{g_1, g_2\}$ and g_1, respectively. Processes in g_1 R-Deliver m_g and define their proposal timestamp for m_g with consensus instance k_1. Shortly after, members of g_1 R-Deliver m_l and decide on m_l in consensus instance k_2, such that $k_2 > k_1$. Message m_l cannot be delivered at this point since m_g has a smaller timestamp than m_l and m_g has not been delivered yet. To deliver the local message m_l, members of g_1 must wait to receive g_2's timestamp proposal for m_g, which may take up to two inter-group message delays, if m_g was A-MCast from within g_1.

To deliver local messages faster, global and local messages are handled differently. We refer to this optimized version of $\mathcal{A}_{ge}^{dv}$ as $\mathcal{A}_{ge}^{dv*}$. Local messages are not assigned timestamps anymore and are A-Delivered directly after consensus. More precisely, when some process wishes to A-MCast a local message m_l to a

group g, m_l is reliably multicast to g. In each group of the system, we run consensus instances to ensure agreement on the delivery order of local messages, as well as to assign timestamps to global messages as explained in Section 4.2.2. As soon as a consensus instance in g decides on m_l, members of g A-Deliver m_l.

To agree on the delivery order of global and local messages, global messages must be A-Delivered after the same consensus instance on members of the same group. To ensure this property, all global messages m_g must go through the four stages defined in Section 4.2.2, even in the group that proposed the highest timestamp for m_g. To understand why this is necessary, consider a global message m_g and a local message m_l that are respectively addressed to groups $\{g_1, g_2\}$ and g_1. Group g_1 is the group that assigned the highest timestamp to m_g. If we allow m_g to skip stage s_2 in g_1, two members p and q of g_1 may A-Deliver m_g and m_l in different orders. For example, assume p and q define m_g's proposal timestamp in a consensus instance k_1. Then, p receives g_2's timestamp for m_g, A-Delivers m_g, decides on m_l in a consensus instance k_2, and A-Delivers m_l. However, q first decides in consensus instance k_2, delivers m_l, receives g_2's timestamp proposal for m_g, and delivers m_g.

The round-based algorithms $\mathscr{A}_{ng}^{dv}$ and $\mathscr{A}_{ng}^{dt}$. In $\mathscr{A}_{ng}^{dv}$ and $\mathscr{A}_{ng}^{dt}$, local messages may be delayed by global messages as much as one inter-group delay if they are multicast while group bundles are being exchanged. We can make this scenario unlikely to happen by executing multiple consensus instances per round. The number of consensus instances per round is denoted by parameter κ.

Although this optimization decreases the average delivery latency of local messages, the delivery latency of global messages can now be increased by as many as κ - 1 consensus instances—this is because each group's bundle of messages is sent every κ consensus instances. Hence, to reduce the delivery latency of global messages, we allow rounds to overlap. That is, we start the next round before receiving the groups' bundles of messages of the current round. In other words, we execute consensus instances while the bundles are being exchanged. In our implementation, message bundles are exchanged after every η consensus instances.

To ensure agreement on the relative delivery order of local and global messages, it is necessary that processes inside the same group agree on when global messages of a given round are delivered, i.e., after which consensus instance. To summarize, processes send the message bundle of some round r after consensus instance $r \cdot \eta$ and A-Deliver messages of round r after instance $r \cdot \eta + \kappa$. Figure 5.1 illustrates a failure-free run of the algorithm. We explore the influence of parameters η and κ in Section 5.3.2.

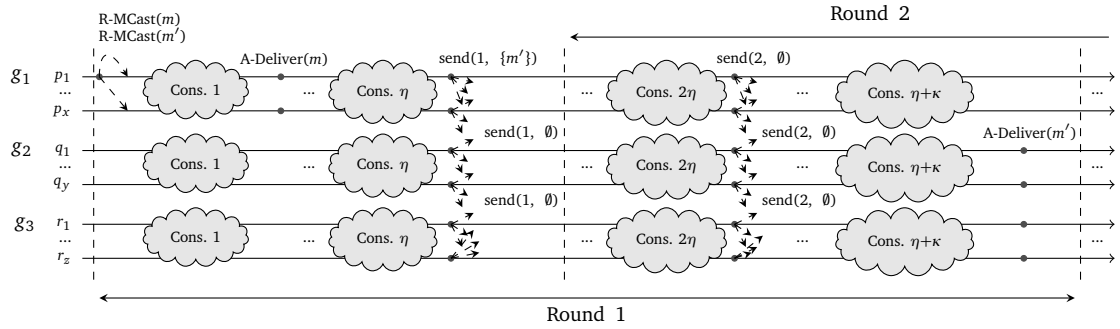


Figure 5.1. Algorithm $\mathcal{A}_{ng}^{dv}$ when messages m and m' are A-MCast from p_1 and are respectively addressed to g_1 and $\{g_2, g_3\}$.

5.2 Implementation Issues

We have implemented the three latency-optimal algorithms in Java, using a Paxos library as the consensus protocol [12]; all communication is based on TCP.

Inter-group communication represents a major source of overhead and should be used sparingly. In our implementation, these communications are handled by a dedicated layer. As we explain below, this layer optimizes the communication, reducing the number of inter-group messages.

Message Batching. Inside each group g, a special process is elected as leader [58]. Members of a group use their leader to batch and forward messages to the remote groups' leaders. When a leader receives a message m, it dispatches m to the members of its group.

Message Filtering. In each one of the presented algorithms, inter-group communication originating from processes of the same group g presents some redundancy. In the non-genuine Algorithms $\mathcal{A}_{ng}^{dv}$ and $\mathcal{A}_{ng}^{dt}$, at the end of a round r, members of g send the same message bundle. Moreover, in the genuine Algorithm $\mathcal{A}_{ge}^{dv}$, members of g send the same timestamp proposal for some message m. To avoid this redundancy, only the group leaders propagate these messages. More precisely, message bundles of Algorithms $\mathcal{A}_{ng}^{dv}$ and $\mathcal{A}_{ng}^{dt}$, and the timestamp proposals of Algorithm $\mathcal{A}_{ge}^{dv}$ are only sent by the group leaders. Messages sent by non-leader processes are discarded by the inter-group communication layer.

In the case of a leader failure, these optimizations may lead to the loss of some messages, which will be resent by the new leader.

5.3 Experimental evaluation

In this section, we evaluate experimentally the performance of the latency-optimal multicast protocols $\mathscr{A}_{ge}^{dv}$, $\mathscr{A}_{ng}^{dv}$, and $\mathscr{A}_{ng}^{dt}$. We start by describing the system parameters and the benchmark used to assess the protocols. We then evaluate the influence of the convoy effect on the algorithms; compare the genuine and non-genuine protocols by varying the load imposed on the system, the number of groups, and the proportion of global messages; and measure the overhead of tolerating disasters.

5.3.1 Experimental Settings

The system. The experiments were conducted in a cluster of 24 nodes connected with a gigabit switch. Each node is equipped with two dual-core AMD Opteron 2 Ghz, 4GB of RAM, and runs Linux 2.6.20. In all experiments, each group consists of 3 nodes; the number of groups varies from 4 to 8. The bandwidth and message delay of our local network, measured using netperf and ping, are about 940 Mbps and 0.05 ms respectively. To emulate inter-group delays with higher latency and lower bandwidth, we used the Linux traffic shaping tools.

We emulated two network setups. In setup 1, the message delay between any two groups follows a normal distribution with a mean of 100 ms and a standard deviation of 5 ms, and each group is connected to the other groups via a 125 KBps (1 Mbps) full-duplex link. In setup 2, the message delay between any two groups follows a normal distribution with a mean of 20 ms and a standard deviation of 1 ms, and each group is connected to the other groups via a 1.25MBps (10 Mbps) full-duplex link. We report the results using setup 1 and briefly comment on the behavior of the algorithms in setup 2.

The benchmark. The communication pattern of our benchmark was modeled after TPC-C, an industry standard benchmark for on-line transaction processing (OLTP) [1]. TPC-C represents a generic wholesale supplier workload and is composed of five predefined transaction types. Two out of these five types may access multiple warehouses; the other three types access one warehouse only. We assume that each group hosts one warehouse. Hence, the warehouses involved in the execution of a transaction define to which groups the transaction

is multicast. Each multicast message also contains the transaction's parameters and on average, a message contains 80 bytes of payload.

In TPC-C, about 10% of transactions involve multiple warehouses. Thus, roughly 10% of messages are global. To assess the scalability of our protocols, we parameterize the benchmark to control the proportion p of global messages. In the experiments, we report measurements for $p = 0.1$ (i.e., original TPC-C) and $p = 0.5$. The vast majority of global messages involve two groups. Note that in our benchmark, transactions are not executed; TPC-C is only used to determine the communication pattern.

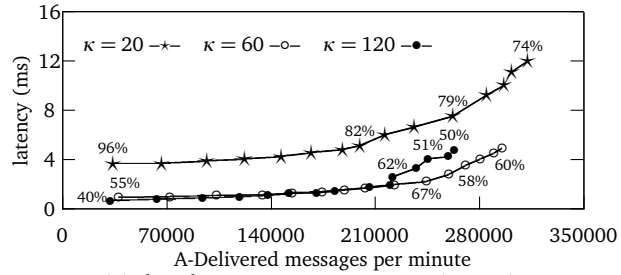
Each node of the system contains an equal number of clients executing the benchmark in a closed loop: each client multicasts a message and waits for its delivery before multicasting another message. Hence, all messages always address the sender's group; global messages also address other groups. The number of clients per node varies between 1 and 160 and in each experiment, at least one hundred thousand messages are multicast.

For all the experiments, we report either the average message delivery latency (in milliseconds) or the average inter-group bandwidth (in kilo bytes per second) as a function of the throughput, i.e., the number of messages A-Delivered per minute. We computed 95% confidence intervals for the A-delivery latency but we do not report them here as they were always smaller than 2% of the average latency. The throughput was increased by adding an equal number of clients to each node of the system.

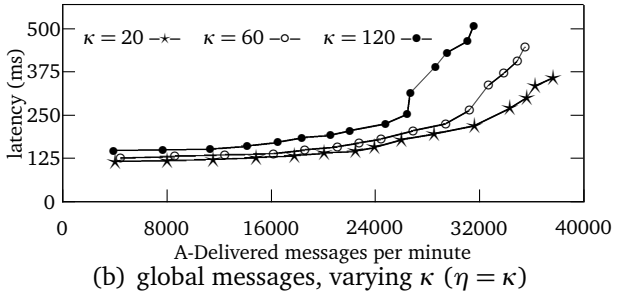
5.3.2 Assessing the Convoy Effect

The round-based Algorithms $\mathscr{A}_{ng}^{dv}$ and $\mathscr{A}_{ng}^{dt}$. We explore the influence of parameters κ, the number of consensus instances per round, and η, the number of consensus instances between two consecutive message bundle exchanges, on the convoy effect. In principle, the higher the value of κ, the less likely the convoy effect is to happen. Indeed, the more consensus instances are run per round, the less probable it is that a local message waits for the message bundles to be exchanged. However, increasing κ also increases the average latency of global messages. The optimal value of κ should be set to allow groups to execute as many local consensus instances as they can while message bundles are being exchanged, to minimize the convoy effect, without affecting the latency of global messages. If we let Δ_{max} and δ_{cons} respectively denote the maximum inter-group delay and the consensus latency, κ should be set to $\lfloor \Delta_{max}/\delta_{cons} \rfloor$. Setting parameter η is less trivial: setting it low potentially decreases the average global message latency but may also saturate the inter-group network.

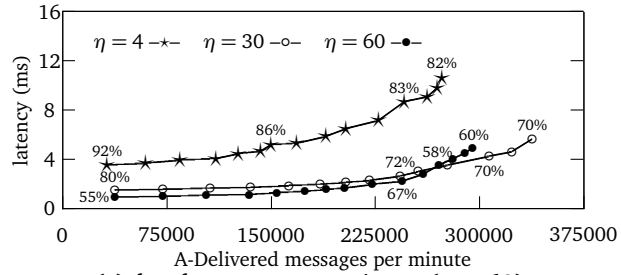
In failure-free runs, $\mathscr{A}_{ng}^{dv}$ and $\mathscr{A}_{ng}^{dt}$ only differ on how the message bundles are exchanged, we thus explore the influence of these parameters on $\mathscr{A}_{ng}^{dv}$ only. Figures 5.2(a) and 5.2(b) illustrate the impact of κ on $\mathscr{A}_{ng}^{dv}$ in a system with four groups when rounds do not overlap (i.e., $\eta = \kappa$). We report the average percentage of the local message latency that is due to the convoy effect. To make the figures easily readable, we do so only for certain loads.



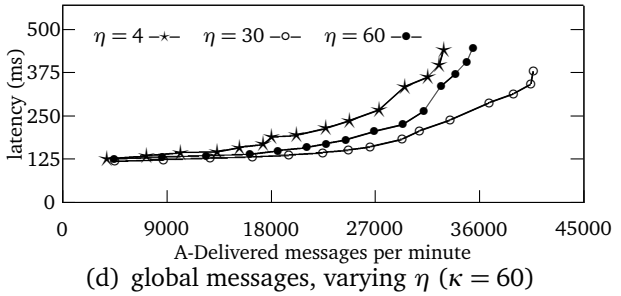
(a) local messages, varying κ ($\eta = \kappa$)



(b) global messages, varying κ ($\eta = \kappa$)



(c) local messages, varying η ($\kappa = 60$)



(d) global messages, varying η ($\kappa = 60$)

Figure 5.2. The influence of κ and η on $\mathscr{A}_{ng}^{dv}$ with four groups (percentages show convoy effect).

As explained above, using very low or high values of κ respectively increases the chances of the convoy effect to happen or the latency of global messages. Hence, we use values of κ close to the theoretical optimum of $\lfloor \Delta_{max}/\delta_{cons} \rfloor$.

We first consider local messages, and note that setting κ too low (i.e., $\kappa = 20$ in our experiments) or too high ($\kappa = 120$) respectively increases the chances of the convoy effect to happen (see percentages in Figure 5.2(a)), which impacts the latency, or harms the scalability of the protocol: as the number of global messages sent at the end of each round increased, the inter-group communication became the bottleneck since the traffic was too bursty. In our settings, $\kappa = 60$ gave the best results for local messages (see Figure 5.2(a)).

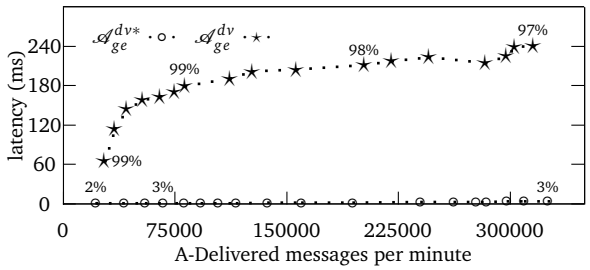
For global messages, setting κ to 60 gives worse performance than setting it to 20 (see Figure 5.2(b)). We address this problem by tuning parameter η, as illustrated in Figures 5.2(c) and 5.2(d). While setting η too low ($\eta = 4$) worsens the latency and the scalability of the protocol, and increases the convoy effect, setting it too high ($\eta = 60$) harms its scalability. When $\eta = 30$, the local message latency is similar to the one when $\kappa = \eta = 60$ (Figure 5.2(a)), while almost matching the global message latency of $\kappa = \eta = 20$ (Figure 5.2(b)). Moreover, with respect to Figures 5.2(a) and 5.2(b), the scalability of the protocol is improved for both local and global messages.

To further reduce the global message latency, we tried other values for η. However, we did not find a value that gave better performance than $\eta = 30$ nor did we reach the theoretical optimum latency of one inter-group message delay, i.e., 100 ms. We observed that this was mainly because groups do not start rounds exactly at the same time; consequently, some groups had to wait more than 100 milliseconds to receive all message bundles. Therefore, in all experiments that follow we use $\kappa = 60$ and $\eta = 30$.

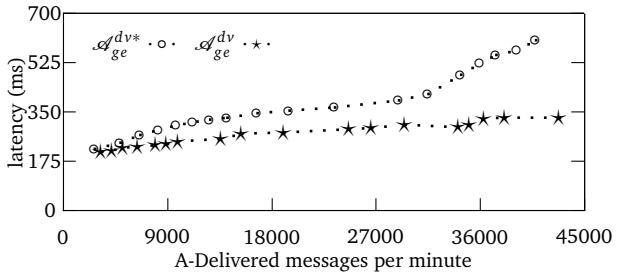
The timestamp-based Algorithm $\mathscr{A}_{ge}^{dv}$. Figures 5.3(a) and 5.3(b) evaluate the influence of the convoy effect on $\mathscr{A}_{ge}^{dv}$ in a system with four groups. Similarly as above, we report the average percentage of the local message latency that is due to the convoy effect. Algorithm $\mathscr{A}_{ge}^{dv*}$ delivers local messages much faster than its non-optimized counterpart prone to the convoy effect $\mathscr{A}_{ge}^{dv}$ (see Figure 5.3(a)): as the load increases the convoy effect happens more frequently and increases the local message latency until it reaches the latency of global messages, i.e., around 200 ms. In fact, with $\mathscr{A}_{ge}^{dv}$, at least 97% of the local message latency is due to the convoy effect. With $\mathscr{A}_{ge}^{dv*}$, as the percentages in Figure 5.3(a) show, local messages also experience a minor convoy effect: after being R-Delivered, some local messages cannot be proposed to consensus directly as the current consensus instance is still running and handling global messages.

In Figure 5.3(b), we observe that $\mathscr{A}_{ge}^{dv*}$ slows down the delivery of global messages. This phenomenon has two causes. First, all global messages now go through the four stages, thus, an increased number of consensus instances must be run for the same throughput. Second, as an effect of the first cause, global messages have a higher chance to be delayed by other global messages. This is similar to the convoy effect, but for global messages.

These observations underline the importance of allowing global messages to skip stage s_2, an optimization that is present in $\mathscr{A}_{ge}^{dv}$, but not allowed in $\mathscr{A}_{ge}^{dv*}$ (c.f. Section 5.1), and render $\mathscr{A}_{ge}^{dv*}$ interesting only when the decrease in local message latency matters more than the increase in global message latency. As we expect the proportion of local messages to be higher than the proportion of global messages, an assumption that is verified by TPC-C, we only consider $\mathscr{A}_{ge}^{dv*}$ hereafter.



(a) local messages, optimized versus non-optimized $\mathscr{A}_{ge}^{dv}$



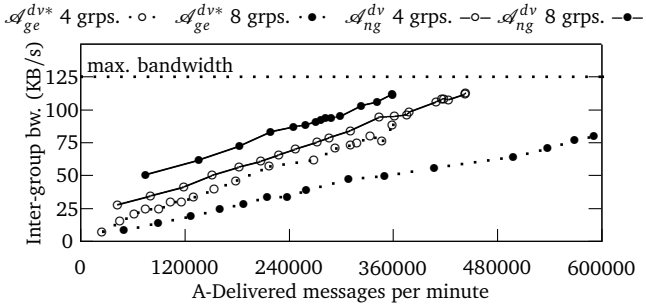
(b) global messages, optimized versus non-optimized $\mathscr{A}_{ge}^{dv}$

Figure 5.3. The convoy effect in $\mathscr{A}_{ge}^{dv}$ with four groups (percentages show convoy effect).

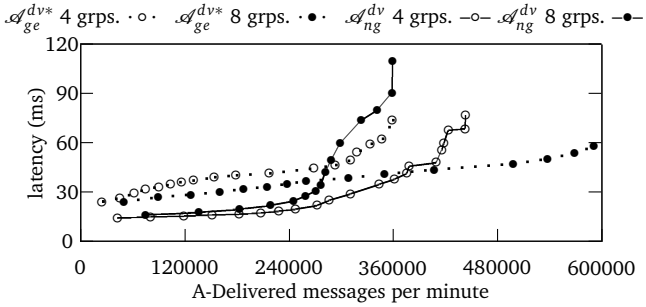
5.3.3 Genuine vs. Non-Genuine Multicast

We compare $\mathscr{A}_{ng}^{dv}$ to $\mathscr{A}_{ge}^{dv*}$ when the number of groups increases using two mixes of global and local messages. We first set the proportion of global messages to 10% and run the algorithms in a system with four and eight groups. Figures 5.4(a) and 5.4(b) respectively report the average outgoing inter-group

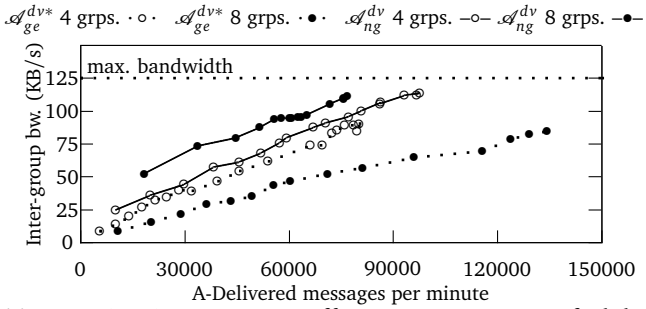
traffic per group and the average A-Delivery latency, both as a function of the throughput.



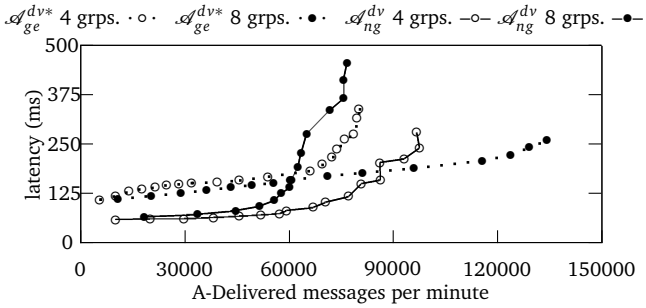
(a) outgoing inter-group traffic per group, 10% of global messages



(b) latency, 10% of global messages



(c) outgoing inter-group traffic per group, 50% of global messages



(d) latency, 50% of global messages

Figure 5.4. Genuine versus Non-Genuine Multicast.

For brevity we report the overall average delivery latency, without differentiating between local and global messages.

Although $\mathscr{A}_{ng}^{dv}$ exhibits a better latency than $\mathscr{A}_{ge}^{dv*}$ with 4 groups, $\mathscr{A}_{ng}^{dv}$ does not scale as well as $\mathscr{A}_{ge}^{dv*}$ with eight groups (Figure 5.4(b)). This is a consequence of $\mathscr{A}_{ng}^{dv}$'s higher demand on throughput: with eight groups the algorithm requires as much as 111 KBps of average inter-group bandwidth, a value close to the maximum available capacity of 125 KBps (Figure 5.4(a)).

In Figures 5.4(c) and 5.4(d), we observe that when half of the messages are global, the two algorithms compare similarly as above but do not scale as well.

As a final remark, we note that in contrast to $\mathscr{A}_{ng}^{dv}$, $\mathscr{A}_{ge}^{dv*}$ delivers messages faster and supports more load with eight groups than with four (Figures 5.4(b) and 5.4(d)). Indeed, increasing the number of groups decreases the load that each group must handle as, in our benchmark, the vast majority of global messages are addressed to two groups. This effect can be seen in Figures 5.4(a) and 5.4(c), where each group needs less inter-group bandwidth with eight groups.

Summary. Figure 5.5 provides a qualitative comparison between the genuine and non-genuine algorithms. We consider four scenarios generated by all combinations of the two following parameters: the load (high or low) and the number of groups (many or few); the proportion of global messages is not taken into account as it has no influence on the comparison. We note that $\mathscr{A}_{ng}^{dv}$ is the winner except when the load is high and there are many groups.

We also carried out the same comparison in network setup 2, i.e., a network where the message delay between any two groups follows a normal distribution with a mean of 20 ms and a standard deviation of 1 ms, and each group is connected to the other groups via a 1.25MBps (10 Mbps) full-duplex link. We briefly comment on the obtained results. With 4 groups, $\mathscr{A}_{ge}^{dv*}$ and $\mathscr{A}_{ng}^{dv}$ compare similarly as in setup 1. Because of the lower inter-group latency the performance of $\mathscr{A}_{ge}^{dv*}$ becomes closer to the one of $\mathscr{A}_{ng}^{dv}$ however. With eight groups, the non-genuine protocol scales almost as well as the genuine algorithm thanks to the extra available inter-group bandwidth.

5.3.4 The Cost of Tolerating Disasters

To evaluate the overhead of tolerating disasters, we compare $\mathscr{A}_{ng}^{dt}$ to the overall best-performing disaster-vulnerable algorithm $\mathscr{A}_{ng}^{dv}$. With $\mathscr{A}_{ng}^{dt}$ we set κ and η to 120 and 60 respectively; with $\mathscr{A}_{ng}^{dv}$, the default values are used, $\kappa = 60$ and $\eta = 30$.

In Figures 5.6(b) and 5.7(b), we observe that with four groups, $\mathscr{A}_{ng}^{dt}$ roughly

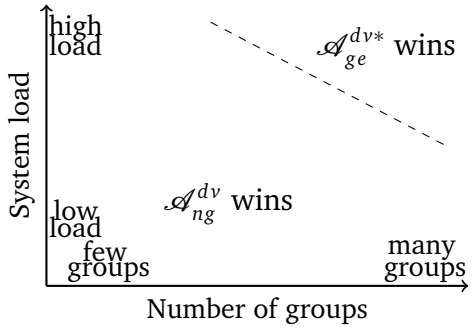
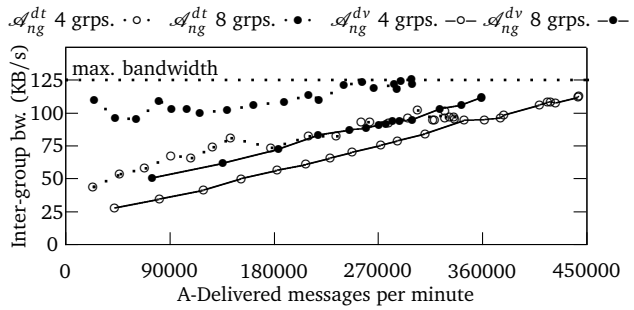
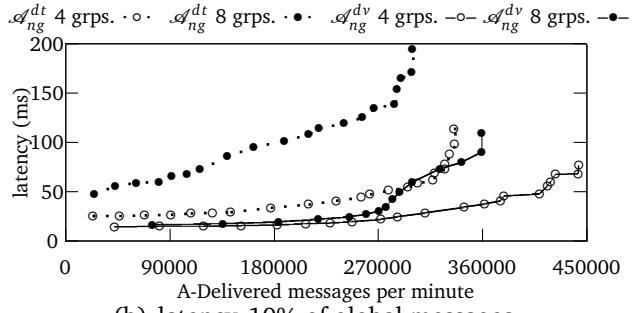


Figure 5.5. Comparing $\mathscr{A}_{ge}^{dv*}$ to $\mathscr{A}_{ng}^{dv}$.

needs twice as much time as $\mathscr{A}_{ng}^{dv}$ to deliver messages. This is expected: local messages take about the same time to be delivered with the two algorithms; global messages roughly need and additional 100 milliseconds to be delivered with $\mathscr{A}_{ng}^{dt}$. Interestingly, $\mathscr{A}_{ng}^{dt}$ matches the performance of $\mathscr{A}_{ge}^{dv*}$ in a system with four groups (Figures 5.4(b), 5.6(b), 5.4(d), and 5.7(b)).



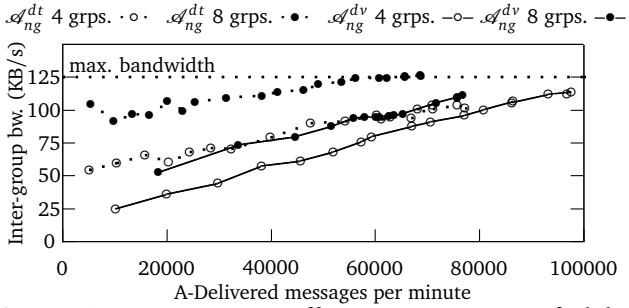
(a) outgoing inter-group traffic per group, 10% of global messages



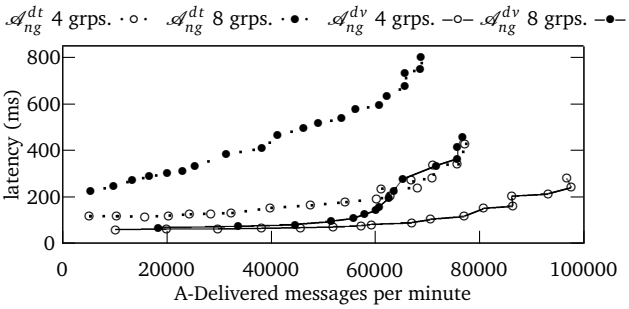
(b) latency, 10% of global messages

Figure 5.6. The cost of tolerating disasters with 10% of global messages.

With eight groups, $\mathscr{A}_{ng}^{dt}$ utilizes the entire inter-group bandwidth under almost every considered load (Figures 5.6(a) and 5.7(a)). The latency and scalability of the disaster-tolerant algorithm thus become much worse than $\mathscr{A}_{ng}^{dv}$'s.



(a) outgoing inter-group traffic per group, 50% of global messages



(b) latency, 50% of global messages

Figure 5.7. The cost of tolerating disasters with 50% of global messages.

5.4 Discussion

Chapter 4 showed that no genuine multicast can deliver global messages in fewer than two inter-group message delays, a restriction that is not imposed on non-genuine multicast protocols. To discover in which circumstances genuine multicast provides better performance than its non-genuine counterpart, we experimentally evaluated the behavior of the latency-optimal genuine and non-genuine protocols $\mathscr{A}_{ge}^{dv}$ and $\mathscr{A}_{ng}^{dv}$ respectively. We assessed the scalability of the algorithms by varying the number of groups, the proportion of global messages, and the load, i.e., the frequency at which messages are multicast.

The results suggest that the genuineness of multicast is interesting only in large and highly loaded systems; in all the other considered scenarios the non-genuine protocol $\mathscr{A}_{ng}^{dv}$ outperforms the genuine algorithm $\mathscr{A}_{ge}^{dv}$. According to our observations, Algorithm $\mathscr{A}_{ge}^{dv}$ outperforms $\mathscr{A}_{ng}^{dv}$ when communication links get saturated. Since extra-bandwidth can be purchased, we believe that $\mathscr{A}_{ng}^{dv}$ is the protocol that should be favored.

We also identified a convoy effect that delays the delivery of local messages

and proposed techniques to reduce this effect. To complete our study, we showed that the disaster-tolerant and latency-optimal protocol $\mathscr{A}_{ng}^{dt}$ does not offer the same level of performance as $\mathscr{A}_{ng}^{dv}$ but matches the performance of $\mathscr{A}_{ge}^{dv}$ when there are few groups.

Chapter 6

Partial Replication Protocols

The whole is more than the sum of its parts.

Aristotle

Database replication protocols based on group communication have received a lot of attention for more than ten years [2; 62; 34; 47; 45; 39; 46]. This comes from the fact that group communication primitives offer adequate properties, namely agreement on the messages delivered and on their order, to implement synchronous database replication. Most of the complexity involved in synchronizing database replicas is handled by the group communication layer.

Previous work on database replication based on group communication has focused mainly on full replication. However, full replication might not always be adequate. First, sites might not have enough disk or memory resources to replicate the database fully. Second, when access locality is observed, full replication is pointless. Third, full replication provides limited scalability since every update transaction must be executed by each replica. In this chapter, we extend the Database State Machine (DBSM) [47], a database replication technique based on group communication, to partial replication.

The DBSM is based on the deferred update replication model [9]. Transactions execute locally on one database site and their execution does not cause any interaction with other sites. Read-only transactions commit locally only; update transactions are atomically broadcast to all database sites at commit time for certification. The certification test ensures *one-copy serializability*: the execution of concurrent transactions on different replicas is equivalent to a serial execution on a single replica [9]. In order to execute the certification test, every database

site keeps the *writesets* of committed transactions. The certification of a transaction T consists in checking that T's *readset* does not contain any outdated value, i.e., no committed transaction T' wrote a data item x after T read x.

6.1 Extending the DBSM to Partial Replication

A straightforward way of extending the DBSM to partial replication consists in executing the same certification test as before but having database sites only process update operations for data items they replicate. But as the certification test requires storing the writesets of all committed transactions, this strategy defeats the whole purpose of partial replication since replicas may store information related to data items they do not replicate. We thus capture the legitimacy of a partial replication protocol in the following property:

- *Genuine Partial Replication:* For every submitted transaction T, database sites that do not replicate data items read or written by T do not store any information about T.

Such a strict property, however, forces the use of a genuine atomic multicast protocol as the group communication primitive to propagate transactions. Since genuine multicast protocols exhibit a higher latency than broadcast ones in large networks (c.f. Chapter 4) and in flat networks [21], this property may restrict the performance of the replication protocol if latency is more important than storage space and message complexity.

To allow broadcast-based implementations of the partially replicated DBSM, we define a weaker version of Genuine Partial Replication. To do so, we let sites receive and momentarily store transactions unrelated to the data items they replicate as long as this information is erased after a short time. Moreover, we want to make sure each transaction is handled by a site at most once. If sites are allowed to forget about past transactions completely, this constraint cannot obviously be satisfied. We capture these two requirements with the following property:

- *Quasi-Genuine Partial Replication:* For every submitted transaction T, correct database sites that do not replicate data items read or written by T permanently store not more than the identifier of T.[1]

[1]Notice that even though transaction identifiers could theoretically be arbitrarily large, in practice, 8-byte identifiers are enough to uniquely represent 2^{64} transactions.

Consider now the following modification to the DBSM, allowing it to ensure Quasi-Genuine Partial Replication. Besides atomically broadcasting transactions for certification, database sites periodically broadcast "garbage collection" messages. When a garbage collection message is delivered, a site deletes all the writesets of previously committed transactions. When a transaction is delivered for certification, if the site does not contain the writesets needed for its certification, the transaction is conservatively aborted. Since all sites deliver both transactions and garbage collection messages in the same order, they will all reach the same outcome after executing the certification test. This mechanism, however, may abort transactions that would be committed in the original DBSM. In order to rule out such solutions, we introduce the following property:

- *Non-Trivial Certification:* If there is a time after which no two conflicting transactions are submitted, then eventually transactions are not aborted by certification.

In Section 6.3, we present a generalization of the database state machine approach. The protocol in [47] is an instance of our generalization in the fully replicated context. We then present two termination protocols that satisfy Quasi-Genuine Partial Replication and one that satisfies Genuine Partial Replication in Section 6.4. All three of these protocols satisfy Non-Trivial Certification as well as the following liveness property: if a correct site submits a transaction T, then either $Site(T)$ aborts T or eventually all correct sites in $Replicas(T)$ commit T.

Both Quasi-Genuine algorithms are optimized for local area networks: they make optimistic assumptions to ensure better performance. The first algorithm is simpler and assumes *spontaneous total order*: with high probability messages sent to all servers in the cluster reach all destinations in the same order, a property usually verified in local area networks. As a drawback, it processes a single transaction at a time. Our second algorithm is able to certify multiple transactions at a time and, as explained in Section 6.4.1, assumes a weaker assumption than spontaneous total order.

The Genuine Partial Replication protocol is not optimized for a particular type of network and does not rely on total order to ensure one-copy serializability. Instead, it relies on a genuine atomic multicast primitive that only orders transactions operating on common data items. In the case of large or hierarchical networks, the disaster-vulnerable and disaster-tolerant genuine algorithms of Chapter 4 can thus be used.[2] In the context of small or flat networks, multi-

[2]When combined with a non-genuine multicast algorithm, the partial replication protocol still ensures one-copy serializability and Non-Trivial Certification, only Genuine Partial Replication is violated.

cast protocols requiring a small total number of message delays, counting both inter-group and intra-group delays, should be favored over the algorithms of Chapter 4.

6.2 Related Work

The majority of database replication protocols consider full replication [47; 45; 46; 62]. These algorithms may offer limited scalability under intensive update workloads as every update must be applied to all replicas. To improve scalability, the research community recently started investigating partial replication. Two consistency criteria have been considered: one-copy serializability [34; 60; 13; 19] and a generalized form of snapshot isolation [59; 7]. With the latter criterion, transactions read data from a possibly old committed snapshot of the database and execute without interfering with each other. A transaction T can only successfully commit if no other transaction T' updated the same data items and committed after T started (*first-committer-wins* rule). This consistency criterion never blocks nor aborts read-only transactions and update transactions are never blocked nor aborted because of read-only transactions. However, the following *write skew* anomaly may occur:

$$r_i[x], r_i[y], ..., r_j[x], r_j[y], w_j[x], c_j, ..., w_i[y], c_i$$

We here review partial replication database protocols that ensure one-copy serializability and satisfy *Quasi-Genuine* or *Genuine Partial Replication*. We start with quasi-genuine algorithms.

In [60] the authors extend the DBSM to partial replication. They use an optimistic atomic broadcast primitive and a variation of atomic commit, called resilient atomic commit. In contrast to atomic commit, resilient atomic commit may decide to commit a transaction even though some participants crash. When a transaction T is optimistically delivered, replicas certify T and execute a resilient atomic commit protocol using the result of the certification test as their vote. If the optimistic order of T corresponds to the final order, the protocol ends; otherwise when the final order is known, T is certified again and a second resilient atomic commit protocol is executed. Since [60] assumes that for each data item x, there exists a correct replica of x, resilient atomic commit is implemented in one message delay, in which all participants exchange their votes. The protocol ensures *Quasi-Genuine Partial Replication*, since only sites replicating data item written by T keep T in their committed transaction sequence.

In this chapter, we present two quasi-genuine protocols tailored for local area networks that exhibit the same latency as the algorithm in [60]. The first algorithm certifies one transaction at a time and relies on spontaneous total order: with high probability messages are received in the same order at all recipients. The second algorithm can certify multiple transactions at a time and, as explained in Section 6.4.1, assumes a weaker assumption than spontaneous total order. Moreover, in contrast to [60], which requires a separate instance of atomic commit to decide on the outcome of each transaction, with Algorithm $\mathcal{A}_{pdsm}^{qg^*}$, a single voting phase suffices to decide on the outcome of several transactions.

A genuine partial replication protocol called C-JDBC is presented in [13]. In C-JDBC, replication is synchronous: read and write operations on a data item x are respectively sent to one and all available replicas of x; databases rely on lock-based concurrency control to execute these operations. To allow database sites to recover after a failure, C-JDBC provides checkpoints and a recovery log.

A protocol optimized for local area networks is presented in [19]. In this algorithm, each transaction T is first routed to the least loaded site s that stores all data items touched by T. Each replica maintains a timestamp variable TS that is incremented every time a new transaction is submitted. When a transaction T is submitted at some site s, s assigns T its current value of TS and FIFO multicasts the transaction to all sites replicating data items touched by T. Upon delivery, T is optimistically executed. To ensure one-copy serializability, transactions are committed in their timestamp order. Hence, if the timestamp order does not follow the delivery order, all transactions with a smaller timestamp than T are aborted and resubmitted. This mechanism may thus abort many transactions under high loads. Moreover, to function properly, the algorithm requires that the message delay be bounded by a known constant.

In [34] the authors propose a database replication protocol based on atomic multicast. Every read operation on data item x is multicast to the group replicating x; writes are multicast along with the commit request. The delivered operations are executed on the replicas using strict two-phase locking and results are sent back to the client. A final atomic commit protocol ensures transaction atomicity. In the atomic commit protocol, every group replicating a data item read or written by a transaction T sends its vote to a *coordinator* group, which collects the votes and sends the result back to all participating groups.

A genuine algorithm based on atomic multicast is presented in this chapter. It requires a single multicast per transaction, and, in contrast to [19], may be deployed in both local and wide area networks.

(a) The protocols in a LAN

Algorithm	Quasi-Genuine or Genuine?	latency with/without opt. assumption	messages	remarks
[60]	quasi-genuine	$3\delta/4\delta$	$O(n^2+d^2)$	-
$\mathscr{A}^{qg}_{pdsm}$	quasi-genuine	$3\delta/4\delta$	$O(n^2+d^2)$	-
$\mathscr{A}^{qg^*}_{pdsm}$				-
[13]	genuine	$(r+w)\times 2\delta$	$O((r+w)\times d^2)$	-
[34]	genuine	$r\times 7\delta+7\delta$	$O(r\times d^2)$	-
[19]	genuine	2δ	$O(d^2)$	requires a synch. system
$\mathscr{A}^{ge}_{pdbsm}$	genuine	6δ	$O(d^2)$	-

(b) The protocols in a WAN

Algorithm	Quasi-Genuine or Genuine?	inter-group latency	inter-group msgs.
[13]	genuine	$(r+w)\times 2\Delta$	$O((r+w)\times d^2)$
[34]	genuine	$r\times 3\Delta+3\Delta$	$O(r\times d^2)$
$\mathscr{A}^{ge}_{pdbsm}$	genuine	3Δ	$O(d^2)$

Table 6.1. Comparison of the database replication protocols (d is the number of sites that replicate data items touched by transaction T, r and w are respectively the number of reads and writes performed by T, and Δ and δ are the message delays in a WAN and in a LAN respectively).

Table 6.1 compares the cost of the reviewed protocols with the three algorithms presented in this chapter. We compare the latency of the algorithms and the number of messages exchanged during the execution of a transaction T, in a local and a wide area network. In local area networks, we assume that messages have a delay of δ, and consider two cases, one where the algorithms' respective optimistic assumptions hold and one where they do not (c.f. Section 6.4); in wide area networks, groups are correct, inter-group messages have a delay of Δ, and intra-group message delays are assumed to be negligible.

We consider the best achievable latency and the minimum number of messages exchanged, when neither failures nor failure suspicions occur, the most frequent case in practical settings. For algorithms whose cost does not depend on any optimistic assumption, we report a single value. To compute the cost of the execution of T, we consider that T consists of r read and w write operation. For all the protocols, we consider that d database sites replicate data items touched by T and that n is the total number of database sites in the system.

Note that for wide area networks, we do not give the cost of Algorithms $\mathscr{A}^{qg}_{pdsm}$, $\mathscr{A}^{qg^*}_{pdsm}$, and [60; 19] as these protocols are optimized for local area networks. As a reference, we present in Table 6.2 the cost of known algorithms used by the protocols compared in this section.

Problem	LAN		WAN	
	latency	msgs.	inter-group latency	inter-group msgs.
Non-Uniform Reliable Broadcast [15]	δ	$O(k^2)$	Δ	$O(k^2)$
Uniform FIFO Multicast $\mathscr{A}_{fifo}$	2δ	$O(k^2)$	2Δ	$O(k^2)$
Uniform Consensus [54]	2δ	$O(k^2)$	2Δ	$O(k^2)$
Non-Blocking Atomic Commit [30] [3]	2δ	$O(k^2)$	2Δ	$O(k^2)$
Uniform Atomic Broadcast [15]	3δ	$O(k^2)$	3Δ	$O(k^2)$
Uniform Atomic Multicast [52]	6δ	$O(k^2)$	6Δ	$O(k^2)$
Uniform Atomic Multicast $\mathscr{A}^{dv}_{ge}$	7δ	$O(k^2)$	2Δ	$O(k^2)$

Table 6.2. Cost of different agreement problems (Δ and δ are the message delays in a WAN and in a LAN respectively, and k denotes the number of participants in the protocol).

In Table 6.1, we consider the execution of a single transaction. The cost of the protocols might however change if we considered multiple transactions. In this scenario, the following observations can be made. First, even though Algorithms $\mathscr{A}^{qg}_{pdsm}$ and $\mathscr{A}^{qg^*}_{pdsm}$ have equal costs, the overhead might be higher for Algorithm $\mathscr{A}^{qg}_{pdsm}$ when multiple transactions are submitted. This stems from the fact that in Algorithm $\mathscr{A}^{qg^*}_{pdsm}$, the cost of running consensus is shared among a set of transactions, therefore reducing the number of generated messages. Second, in [34; 60], each transaction requires a separate instance of atomic commit to decide on its outcome. In Algorithm $\mathscr{A}^{qg^*}_{pdsm}$, however, at most two voting phases are needed to decide on the outcome of the sequence of transactions decided in the same consensus instance. Therefore, the longer this sequence, the cheaper Algorithm $\mathscr{A}^{qg^*}_{pdsm}$ will be compared to [34; 60].

[3] This cost corresponds to the case where all participants spontaneously start the protocol. This assumption makes sense here because in [34] participants deliver a transaction's commit request before starting the atomic commit protocol.

6.3 The Database State Machine Approach

We now present a generalization of the Database State Machine approach (DBSM). The protocol in [47] is an instance of our generalization in the fully replicated context. For the sake of simplicity, we consider a replication model where a transaction T can only be executed on a site s_i if $Items(T) \subseteq Items(s_i)$. In Section 6.4.3 we revisit this assumption. To simplify the presentation, we consider a client c that sends requests on behalf of a transaction T to $Site(T)$. In the following, we comment on the states in which a transaction can be in the DBSM.

- *Executing:* Read and write operations are executed locally at $Site(T)$ according to the strict two-phase locking rule (strict 2PL). When c requests to commit T, it is immediately committed and passes to the Committed state if it is a read-only transaction, an event that we denote $Committed(T)_{Site(T)}$; if T is an update transaction, it is submitted for certification and passes to the Submitted state at $Site(T)$. We represent this event as $Submitted(T)_{Site(T)}$. In the fully replicated case, to submit T, sites use an atomic broadcast primitive; in a partial replication context, the algorithms of Section 6.4 are used.

- *Submitted:* When T enters the Submitted state, its read locks are released at $Site(T)$ and T is eventually certified. With full replication, the certification happens when T is delivered; Section 6.4 explains when certification happens in a partially replicated scenario. Certification ensures that if a committed transaction T' executed concurrently with T, and T read a data item written by T' then T is aborted. T' is concurrent with T if it committed at $Site(T)$ *after* T entered the Submitted state at $Site(T)$. Therefore, T passes the certification test on site s_i if for every T' already committed at s_i the following condition holds:

$$\begin{array}{c} Committed(T')_{Site(T)} \rightarrow Submitted(T)_{Site(T)} \\ \vee \\ T'.ws \cap T.rs = \emptyset, \end{array} \tag{6.1}$$

  where $\rightarrow$ is Lamport's happened before relation on events [37].

  In the fully replicated DBSM, transactions are certified locally by each site upon delivery. In the partially replicated DBSM, to ensure *Quasi-Genuine Partial Replication*, sites only store the writesets of committed transactions that wrote data items they replicate. Therefore, sites might not have enough information to decide on the outcome of all transactions. Hence,

to satisfy *Non-trivial Certification*, we introduce a voting phase where each site sends the result of its certification test to the other sites. Site s_i can safely decide to commit or abort T when it has received votes from a *voting quorum* for T. Intuitively, a voting quorum VQ for T is a set of databases such that for each data item read by T, there is at least one database in VQ replicating this item. More formally, a quorum of sites is a voting quorum for T if it belongs to $VQS(T)$, defined as follows:

$$VQS(T) = \{VQ | VQ \subseteq \Pi \wedge T.rs \subseteq \bigcup_{s \in VQ} Items(s)\} \tag{6.2}$$

For T to commit, every site in a voting quorum for T has to vote *yes*. If a site in the quorum votes *no*, it means that T read an old value and should be aborted; committing T would make the execution non-serializable. Notice that $Site(T)$ is a voting quorum for T by itself, since for every transaction T, $Items(T) \subseteq Items(Site(T))$. If T passes the certification test at s_i, it requests the write locks for the data items it has updated. If there exists a transaction T' on s_i that holds conflicting locks with T's write locks, the action taken depends on T''s state on s_i and on T''s type, read-only or update:

1. *Executing:* If T' is in execution on s_i then one of two things will happen: if T' is a read-only transaction, T waits for T' to terminate; if T' is an update transaction, it is aborted.
2. *Submitted:* This happens if T' executed on s_i, already requested commit but was not committed yet. In this case, T's updates should be applied to the database before T''s. How this is ensured is implementation specific.[4]

Once the locks are granted, T applies its updates to the database and passes to the Committed state. If T fails the certification test, it passes to the Aborted state.

- *Committed/Aborted:* These are final states.

[4]For example, a very simple solution would be for s_i to abort T'; if T' later passes certification, its writes would be re-executed. The price paid for simplicity here is the double execution of T''s write operations.

6.4 Partially-replicated DBSM

In this section, we present three algorithms for the termination protocol of the DBSM in a partial replication context. These protocols ensure both one-copy serializability [9] and the following liveness property: if a correct site submits a transaction T, then either $Site(T)$ aborts T or eventually all correct sites in $Replicas(T)$ commit T. The algorithms also satisfy either Quasi-Genuine or Genuine Partial Replication as well as Non-Trivial Certification.

6.4.1 Quasi-Genuine Algorithms

The "One-at-a-time" Algorithm

Algorithm Overview. Sites execute a sequence of *steps*. In each step, sites decide on the outcome of one transaction. A step is composed of two phases, a consensus phase and a voting phase. Consensus is used to guarantee that sites agree on the commit order of transactions. In the voting phase, sites exchange the result of their certification test to ensure that the commit of a transaction T in step K induces a serializable execution.

The naive way to implement the termination protocol is to first use consensus to determine the next transaction T in the serial order and then execute the voting phase for T. We take a different approach: Based on the observation that with a high probability messages broadcast in a local-area network are received in total order [49], we overlap the consensus phase with the voting phase to save one communication step. If sites receive the transaction to be certified in the same order, they vote for the transaction before proposing it to consensus. With luck, by the time consensus decides on a transaction T, every site will already have received the votes for T and will be able to decide on the outcome of T.

The Algorithm in Detail. Algorithm $\mathscr{A}^{qg}_{pdsm}$ is composed of three concurrent tasks. Each line of the algorithm is executed atomically. The state transitions of transactions are specified in the right margin of lines 10, 28, and 30. Notice that the state transition happens after the corresponding line has been executed. Every transaction T is a tuple $(id, site, rs, ws, up, past, order)$. We added three fields to the definition of a transaction (c.f. Chapter 2), namely *site*, *past*, and *order*: *site* is the database site on which T is executed; *past* is the order of T's submission; and *order* is T's commit order. The algorithm also uses five global variables: K stores the *step* number; *UNDECIDED* and *DECIDED* are (ordered) sequences

of, respectively, pending transactions and transactions for which the outcome is known; *COMMITTED* is the set of committed transactions; and the set *VOTES* stores the votes received, i.e., the results of the certification test. We use the operators $\oplus$ and $\ominus$ for the concatenation and decomposition of sequences. Let seq_1 and seq_2 be two sequences of transactions. Then, $seq_1 \oplus seq_2$ is the sequence of transactions in seq_1 followed by all the transactions in seq_2, and $seq_1 \ominus seq_2$ is the sequence of transactions in seq_1 that are not in seq_2. Transactions are matched using their identifiers.

To take advantage of spontaneous total order, database sites use the WOR-Broadcast primitive to submit transactions (line 10). When no consensus instance is running and *UNDECIDED* is not empty, sites first execute the *Vote* procedure for T at the head of *UNDECIDED* (line 17) and then propose T (line 18). In the *Vote* procedure, T is certified and the result of the certification is sent in a message of type *VOTE*.

Notice that even though $Site(T)$ is a voting quorum for T by itself, i.e, $Items(T) \subseteq Items(Site(T))$, in the algorithm, all sites replicating a data item read by T vote. This is done to tolerate the crash of $Site(T)$. If only $Site(T)$ voted, the following undesirable scenario could happen: $Site(T)$ submits T and crashes just after executing line 10. Databases *WOR-Deliver* T, propose T and decide on T. In this execution, sites would wait forever at line 24, as $Site(T)$ crashed before voting for T.

Two further remarks concern the *Vote* procedure. First, to be able to certify transactions, we need to implement the precedence relation $\rightarrow$ between events. For two transactions T and T', this is done by comparing the value of their *past* and *order* fields. If $T.order < T'.past$, we are sure that T committed before T' was submitted, because K is incremented after transactions commit. Second, notice that *VOTE* messages contain the *step number* K in which T was certified. This information is necessary because a transaction can be certified in different steps and the result of the certification test in steps K and K' might be different. This is precisely why sites wait for *VOTE* messages coming from step number K at line 24. Moreover, even if sites receive votes from different voting quorums, they will agree on the outcome of the transaction. Intuitively, this holds because we only take into account *voting quorums* that voted in *step* K, therefore they consider the same sequence of committed transactions. Finally, by verifying that transactions T and T' are the same at line 20, sites check if the spontaneous total order holds. If it is not the case, sites need to vote for the transaction decided by consensus.

Algorithm $\mathscr{A}^{qg}_{pdsm}$
The "One-at-a-time" algorithm - Code of database site s

```
1:  Initialization
2:    K ← 1, UNDECIDED ← ϵ, DECIDED ← ϵ,COMMITTED ← ∅, VOTES ← ∅
3:  function Certify(T)
4:    return ∀(id, order, ws) ∈ COMMITTED : order < T.past ∨ ws ∩ T.rs = ∅
5:  procedure Vote(T)
6:    if T.rs ∩ Items(s) ≠ ∅ then
7:      send(VOTE, T.id, K, Certify(T)) to all q in Replicas(T)
8:  To submit transaction T                                          {Task 1}
9:    T.past ← K
10:   WOR-Broadcast(VOTE_REQ, T)                                    {Executing → Submitted}
11: When receive(VOTE, T.id, K′, vote) from q                        {Task 2}
12:   VOTES ← VOTES ∪ (T.id, q, K′, vote)
13: When WOR-Deliver(VOTE_REQ, T) ∧ T.id ∉ DECIDED                   {Task 3}
14:   UNDECIDED ← UNDECIDED ⊕ T
15: When UNDECIDED ≠ ϵ
16:   T ← head(UNDECIDED)
17:   Vote(T)
18:   Propose(K, T)
19:   wait until Decide(K, T′)
20:   if T′.id ≠ T.id then Vote(T′)
21:   UNDECIDED ← UNDECIDED ⊖ T′
22:   DECIDED ← DECIDED ⊕ T′.id
23:   if T′.ws ∩ Items(s) ≠ ∅ then
24:     wait until ∃VQ ∈ VQS(T′) : ∀q ∈ VQ : (T′.id, q, K, −) ∈ VOTES
25:     if ∀q ∈ VQ : (T′.id, q, K, yes) ∈ VOTES then
26:       T′.order ← K
27:       COMMITTED ← COMMITTED ∪ (T′.id, T′.order, T′.ws ∩ Items(s))
28:       commit T′                                                  {Submitted → Committed}
29:     else
30:       if s = T′.site then abort T′                               {Submitted → Aborted}
31:   K ← K + 1
32:   VOTES ← {(tid, q, K′, v) ∈ VOTES | K′ ≥ K}
```

The "Many-at-a-time" Algorithm

The previous algorithm certifies transactions sequentially. Thus, if many transactions are submitted, an ever-growing chain of uncommitted transactions can be formed. Algorithm $\mathscr{A}^{qg^*}_{pdsm}$ solves that problem by allowing a sequence of trans-

actions to be proposed in consensus instances and by changing the certification test accordingly.

Algorithm Overview. Algorithm $\mathscr{A}_{pdsm}^{qg^*}$ is similar to algorithm $\mathscr{A}_{pdsm}^{qg}$, its main difference lies in the way transactions are certified. Since a sequence of transactions can be proposed to consensus, sites now optimistically certify a sequence S of transactions before each consensus instance.

A naive implementation would certify transactions in the order they appear in S, that is, transactions would be certified against previously committed transactions and transactions that appear before them in S. To guarantee quasi-genuine partial replication, sites only permanently store a subset of the committed transactions. Hence, the certification would require that, for each transaction T in S, sites vote for T considering each possible outcome of the transactions appearing before T in S. Indeed, a site s_i may vote to abort a transaction T_1 because s_i voted to commit the transaction T_2 directly preceding T_1. However, another site s_j may have voted to abort T_1, in which case T_1 would abort and T_2 could thus potentially commit.

We opt for a different approach that is simple and only requires sites to cast a single vote per transaction in S. Transactions are certified in two phases. In the first phase, each transaction in S is certified against previously committed transactions only. In the second phase, once sites have decided on the transaction sequence S', transactions are handled in the order they appear in S'. More precisely, for each transaction T in S', sites check whether T can commit by looking at the votes cast by T's voting quorum. If it is the case, T is certified against committed transactions that precede T in S'.

Algorithm in Detail. Algorithm $\mathscr{A}_{pdsm}^{qg^*}$ follows the same structure and uses the same global variables as Algorithm $\mathscr{A}_{pdsm}^{qg}$. The difference lies in Task 3 and the auxiliary procedures used. In the general case, when sites notice that there is a sequence of pending transactions that have not been committed or aborted ("$UNDECIDED \neq \epsilon$" at line 25), this sequence is voted for and proposed in consensus instance K (lines 26–27). In the *Vote* procedure, every pending transaction is certified considering only the previously committed transactions (lines 3–9). The results are gathered in a set and later sent to all sites that have data items updated by some transaction in the pending sequence (lines 10–12). The "$VOTES \neq \emptyset$" condition at line 25 is used for garbage collection: it forces the proposal of empty sequences in case there are votes for undelivered vote requests (a possible situation due to failures that would violate *Quasi-Genuine Partial*

Replication).

After the K-th instance of consensus has decided on a sequence *SEQ* of transactions (line 28), sites verify whether they have voted for all transactions in *SEQ*; if it is not the case, they vote for the sequence *SEQ* (lines 29–30). Then, sites replicating data items updated by one of the transactions in *SEQ* sequentially certify all transactions in *SEQ* following their order (lines 35–45). The certification of transaction T is divided into two parts. First, T is certified considering the transactions committed in steps lower than K by taking into account the votes of a voting quorum (line 37). Second, sites certify T considering committed transactions that have been decided in the same consensus instance (line 38). This is done by gathering committed transactions in a set called *LCOMMIT* and by verifying that there does not exist a transaction T' in this set that writes a data item read by T. If T passes both certifications and updates a data item in $Items(s)$, it is treated in exactly the same way as certified transactions in Algorithm $\mathscr{A}^{qg}_{pdsm}$ (lines 41–43). To reduce the number of aborts, transactions in the decided sequence *SEQ* could be reordered using the deterministic reordering technique introduced in [47]. We omit this optimization from the code for simplicity.

Unlike Algorithm $\mathscr{A}^{qg}_{pdsm}$, Algorithm $\mathscr{A}^{qg^*}_{pdsm}$ does not rely on spontaneous total order. This is because sequences of transactions are used when voting and proposing values to a consensus instance, and the order of transactions in this sequence does not matter when it comes to voting. Recall that the vote phase in step K consists in independently certifying undecided transactions against transactions committed in previous steps (line 11 and function *Certify* at lines 3–9). This phase does not take into consideration conflicts within the sequence itself since they are solved after the consensus instance is decided. Nevertheless, the voting mechanism is still optimistic in Algorithm $\mathscr{A}^{qg^*}_{pdsm}$ as they are sent before the consensus instance has decided on its outcome.

The optimistic assumption that allows a transaction T to be certified as soon as consensus instance K decides on a sequence containing T is that every member of at least one correct voting quorum *VQ* for T has voted for any sequence containing T before consensus instance K (line 26). Notice that the sequences considered by different members of *VQ* do not have to be the same, the only requirement is that they all contain T.

We could further relax the optimistic assumptions required at the price of a higher number of VOTE messages. In the way both algorithms are described, sites vote for a transaction only before it is proposed to the next consensus instance (line 17 of Algorithm $\mathscr{A}^{qg}_{pdsm}$, line 26 of Algorithm $\mathscr{A}^{qg^*}_{pdsm}$). Consider a scenario where the vote request for a transaction T is delivered by a site s right after s

Algorithm $\mathcal{A}^{qg^*}_{pdsm}$
The "Many-at-a-time" algorithm - Code of database site s

1: **Initialization**
2: $K \leftarrow 1$, $UNDECIDED \leftarrow \epsilon$, $DECIDED \leftarrow \epsilon$, $COMMITTED \leftarrow \emptyset$, $VOTES \leftarrow \emptyset$
3: **function** Certify(SEQ)
4: $V \leftarrow \emptyset$
5: **for all** $T \in SEQ$ **do**
6: **if** $\forall (id, order, ws) \in COMMITTED : order < T.past \ \vee \ ws \cap T.rs = \emptyset$ **then**
7: $V \leftarrow V \cup (T.id, yes)$
8: **else** $V \leftarrow V \cup (T.id, no)$
9: **return** V
10: **procedure** Vote(SEQ)
11: **if** $\exists T \in SEQ \ : \ T.rs \cap Items(s) \neq \emptyset$ **then**
12: send (VOTE, Strip(SEQ), K, Certify(SEQ)) to $\{q \mid \exists T \in SEQ \ : \ q \in Replicas(T)\}$
13: **function** Strip(SEQ)
14: $RESULT \leftarrow \epsilon$
15: **for all** $T \in SEQ$ in order **do**
16: $RESULT \leftarrow RESULT \oplus T.id$
17: **return** $RESULT$
18: **To submit transaction** T {Task 1}
19: $T.past \leftarrow K$
20: R-bcast (VOTE_REQ, T) {*Executing → Submitted*}
21: **When** receive (VOTE, $IDSEQ, K', V$) from q {Task 2}
22: $VOTES \leftarrow VOTES \cup (IDSEQ, q, K', V)$
23: **When** R-deliver (VOTE_REQ, T) $\wedge$ $T.id \notin DECIDED$ {Task 3}
24: $UNDECIDED \leftarrow UNDECIDED \ \oplus \ T$
25: **When** $UNDECIDED \neq \epsilon \vee VOTES \neq \emptyset$
26: Vote($UNDECIDED$)
27: Propose(K,$UNDECIDED$)
28: **wait until** Decide(K,SEQ)
29: **if** $\exists T \ : \ T \in SEQ \ \wedge \ T \notin UNDECIDED$ **then**
30: Vote(SEQ)
31: $DECIDED \leftarrow DECIDED \ \oplus$ Strip(SEQ)
32: $UNDECIDED \leftarrow UNDECIDED \ \ominus \ SEQ$
33: **if** $\exists T \in SEQ \ : \ T.ws \cap Items(s) \neq \emptyset$ **then**
34: $LCOMMIT \leftarrow \emptyset$
35: **for all** $T \in SEQ$ in order **do**
36: **wait until**
$\exists VQ \in VQS(T) : \forall q \in VQ : \exists (SEQ_q, q, K, V_q) \in VOTES \ : \ T \in SEQ_q$
37: **if** $(\forall q \in VQ : \exists (SEQ_q, q, K, V_q) \in VOTES : T \in SEQ_q \wedge (T.id, yes) \in V_q)$
38: $\wedge \ (\nexists T' \in LCOMMIT \ : \ T'.ws \cap T.rs \neq \emptyset)$ **then**
39: $LCOMMIT \leftarrow LCOMMIT \cup \{T\}$
40: **if** $T.ws \cap Items(s) \neq \emptyset$ **then**
41: $T.order \leftarrow K$
42: $COMMITTED \leftarrow COMMITTED \cup (T.id, T.order, T.ws \cap Items(s))$
43: commit T {*Submitted → Committed*}
44: **else**
45: **if** $s = T.site$ **then** abort T {*Submitted → Aborted*}
46: $K \leftarrow K + 1$
47: $VOTES \leftarrow \{(tid, q, K', v) \in VOTES \mid K' \geq K\}$

has proposed transaction(s) to consensus. Site s will therefore have to wait until the instance finishes to send its vote concerning T. However, T's vote request might have been delivered earlier by some other site and might even have been proposed to the current instance of consensus. If that is the case, and T is part of the consensus decision, the optimistic assumptions will not hold and the protocols might need an extra message step to certify T. This problem can be avoided if sites are allowed to vote while solving a consensus instance. In our example scenario, site s would vote for T even though it has already voted for its consensus proposal. Both votes would then be received by other sites and they would be used to decide on the outcome of T. This optimization relieves the need for spontaneous total order in Algorithm $\mathscr{A}_{pdsm}^{qg}$ and relaxes even more the optimistic assumption of Algorithm $\mathscr{A}_{pdsm}^{qg^*}$. As a secondary effect, it reduces the average latency of transaction certification since votes are sent right after the vote request is received.

6.4.2 A Genuine Algorithm

Unlike the previous quasi-genuine protocols that rely on consensus, the genuine algorithm $\mathscr{A}_{pdbsm}^{ge}$ we present next relies on atomic multicast. It assumes that each group replicates the same set of data items. To ensure genuine partial replication, Algorithm $\mathscr{A}_{ge}^{dv}$ or $\mathscr{A}_{ge}^{dt}$ can be used as the atomic multicast protocol. We first present an overview of the algorithm and then present $\mathscr{A}_{pdbsm}^{ge}$ in detail.

Algorithm Overview. When a transaction T is submitted for certification, Algorithm $\mathscr{A}_{pdbsm}^{ge}$ multicasts T to all groups that replicate data items touched by T. Upon A-Delivering T, sites certify T. Since transactions are now only partially ordered, the certification test must be implemented differently from the quasi-genuine algorithms. Recall that to certify a transaction T, we must check that each data item read by T is still up-to-date. To do so, each site maintains a version number $TS[x]$ for each data item x replicated locally. Upon submitting a transaction T, for each data item x read by T, $T.past[x]$ is set to the current version number of x. We then certify T by checking that for all data items x read by T, $T.past[x] = TS[x]$ holds.

If the number of data items is large, then this mechanism may need too much storage space. A simple way to circumvent this problem is to use a coarser granularity for data items: instead of using row granularity, table granularity could for example be used. However, this technique unnecessarily aborts some of the transactions. Indeed, with table granularity, two transactions T and T'

conflict as soon as they access the same table x and one of the transactions updates x. Hence, if T and T' are executed concurrently, one transaction will be aborted, even if T and T' access different rows of x. We discuss an alternative mechanism that does not have this drawback after the in-detail explanation of the algorithm.

Algorithm in Detail. Algorithm $\mathscr{A}^{ge}_{pdbsm}$ is composed of three concurrent tasks. Each line of the algorithm is executed atomically. The algorithm uses two global variables: *VOTES* store the same information as in the previous quasi-genuine protocols, and for each data item x replicated locally, $TS[x]$ denotes the current version number of x.

When a transaction T is submitted, for each data item x read by T, $T.past[x]$ is set to the current version number of x, and T is multicast to all sites concerned by T (lines 6-7). Upon A-Delivering T, sites that store data items read by T certify T (lines 11-12). Any replica s of data items updated by T then proceed as follows: s waits to receive votes from a voting quorum for T (lines 13-14); increments the version number of all data items written by T, if T commits (lines 15-16); and garbage collects the votes of T (line 21). If T fails the certification test, $Site(T)$ aborts T.

A certification alternative. The technique we describe next uses a single timestamp variable to certify transactions, and, unlike the technique using a coarser granularity for data items, it does not unnecessarily abort transactions. This mechanism proceeds as follows. Each site maintains a single timestamp variable denoted as *TS*. This variable is incremented every time a transaction is A-Delivered and is piggybacked on the vote message of each transaction T. Upon submitting T, $T.past$ is set to *TS*. If T commits, then *TS* is set to the maximum timestamp received in T's vote messages plus one. Sites then assign the current value of *TS* to *T.order* and execute the same certification test as in the quasi-genuine protocols.

In order to function properly, this mechanism requires each group g to either not replicate any data item in common with any other group, or if g replicates a data item that another group g' replicates, then g and g' must replicate the exact same set of data items. If this condition were not met, then sites belonging to different groups would set *T.order* to different values, thus leading to disagreement on the certification of future transactions conflicting with T.

Algorithm $\mathscr{A}^{ge}_{pdbsm}$
Genuine Partial Replication - Code of database site s

```
1:  Initialization
2:    VOTES ← ∅, TS[x] ← 1, for each x ∈ Items(s)
3:  function Certify(T)
4:    return ∀x ∈ T.rs ∩ Items(s) : T.past[x] = TS[x]
5:  To submit transaction T                                          {Task 1}
6:    foreach x ∈ T.rs do T.past[x] ← TS[x]
7:    A-MCast(VOTE_REQ, T) to all q s.t.
                  Items(q) ∩ Items(T) ≠ ∅                  {Executing → Submitted}
8:  When receive(VOTE, T.id, vote) from q                            {Task 2}
9:    VOTES ← VOTES ∪ (T.id, q, vote)
10: When A-Deliver(VOTE_REQ, T)                                      {Task 3}
11:   if T.rs ∩ Items(s) ≠ ∅ then
12:     send(VOTE, T.id, Certify(T)) to all q in Replicas(T)
13:   if T.ws ∩ Items(s) ≠ ∅ then
14:     wait until ∃VQ ∈ VQS(T) : ∀q ∈ VQ : (T.id, q, -) ∈ VOTES
15:     if ∀q ∈ VQ : (T.id, q, yes) ∈ VOTES then
16:       commit T                                       {Submitted → Committed}
17:       foreach x ∈ T.ws ∩ Items(s) do
18:         TS[x] ← TS[x] + 1
19:     else
20:       if s = T.site then abort T                     {Submitted → Aborted}
21:     VOTES ← {(tid, q, v) ∈ VOTES | tid ≠ T.id}
```

6.4.3 Handling Distributed Transactions

In the protocols above, we assume that each transaction T is executed on a site s that stores all data items read and updated by T. We discuss below how to remove this assumption.

Items updated. In the DBSM, s does not apply T's updates to the database during T's execution. However, T holds its write locks on s until T's certification. This is only done to reduce the transaction abort rate and not for correctness. Indeed, holding these locks serializes transactions that update the same data items as T on s. Hence, to allow T to execute on a site that does not replicate all data items T updates, the DBSM requires no modification. The transaction abort rate may however increase since T will not hold locks on remote data items T updated.

Items read. To allow T to read some data item x remotely, T may simply ask x from a site s that replicates x. Along with the value of x, s should send the version number of x. Sites can then certify T by checking that all data items T read have a version number equal to their current version number, similarly as in the genuine protocol $\mathscr{A}^{ge}_{pdbsm}$. This mechanism can also be implemented in the quasi-genuine protocols, provided that they use the same certification procedure as in $\mathscr{A}^{ge}_{pdbsm}$.

We here briefly argue why this modification still ensures one-copy serializability. The DBSM's correctness relies on two properties: (i) transactions do not read stale data and (ii) updates of any data item x are applied in the same order at all replicas of x. It is easy to see that these two properties guarantee one-copy serializability: the global transaction history H is *equivalent* to a one-copy sequential history $1H$ in which the transaction order follows the data items' update order; if two transactions do not update any data item in common they can appear in any order in $1H$.

With the modification proposed above, property (ii) trivially remains true as updates of x are still totally ordered across replicas of x. Property (i) also holds because of the certification test: a transaction T commits only if the version number of each data item x read by T is equal to x's current version number, guaranteeing that data items read are still up-to-date.

6.5 Discussion

Partial replication is a promising alternative to full replication since it potentially offers more scalability: update transactions must only be executed by a subset of the system's replicas.

In this chapter, we extended the database state machine approach to partial replication and presented three transaction termination protocols. The first two algorithms are quasi-genuine, i.e., sites permanently store the identifiers of all submitted transactions in addition to the data items they replicate; the third protocol is genuine, i.e., sites do not store information about transactions that do not *touch* any data item they replicate.

The quasi-genuine property is interesting for latency reasons: it allows the use of atomic broadcast, a primitive that is inherently more latency-efficient than multicast (c.f. Chapter 4). When storage space or bandwidth is in short supply however, the genuine algorithm is preferred.

6.6 Proofs of Correctness

We here only give the proofs of Algorithms $\mathscr{A}^{qg^*}_{pdsm}$ and $\mathscr{A}^{ge}_{pdbsm}$, as Algorithm $\mathscr{A}^{qg}_{pdsm}$ is a special case of Algorithm $\mathscr{A}^{qg^*}_{pdsm}$.

6.6.1 The Proof of Algorithm $\mathscr{A}^{qg^*}_{pdsm}$

Lemma 6.6.1 *No two consensus instances k, k′ ($k \neq k'$) decide on the same transaction T.*

Proof: Suppose, by way of contradiction, that two instances k and k' decide on the same transaction T. Without loss of generality, suppose that $k' > k$. Consequently, by the validity property of consensus, there exists a database site s_i that proposes T for instance k'. If s_i proposes T in instance $k' > k$, s_i decided in instance k. By the uniform agreement of consensus, s_i decided on T in instance k. Therefore, $T.id$ is added to $DECIDED_i$ at line 31 and T is removed from $UNDECIDED_i$ at line 32 when $K_i = k$. Since no transaction is removed from $DECIDED_i$, T cannot be inserted in $UNDECIDED_i$ afterwards at line 24. Therefore, T cannot be in $UNDECIDED_i$ at line 27 when $K_i > k$ and therefore s_i does not propose T in consensus instance k', a contradiction. □

Lemma 6.6.2 *For any submitted transaction T, any database site s_i, and any K_i, between two invocations of Certify(-), namely Certify(-)$_1$ and Certify(-)$_2$, such that K_i is the same at the time Certify(-)$_1$ and Certify(-)$_2$ are called, Certify(-)$_1$ reads the same COMMITTED set as Certify(-)$_2$.*

Proof: The function Certify is called indirectly either (a) from line 26 or (b) from line 30. The set *COMMITTED* is modified at line 42. Therefore, Certify(-)$_2$ reads the same *COMMITTED* set as Certify(-)$_1$ when K_i is the same at the time Certify(-)$_1$ and Certify(-)$_2$ are called. □

Lemma 6.6.3 *For any submitted transaction T and any database site s_i, in every step K_i, any two invocations of Certify(-), namely Certify(SEQ_1) and Certify(SEQ_2), return the same value for all transactions T such that $T \in SEQ_1 \cap SEQ_2$.*

Proof: By Lemma 6.6.2, the values of *COMMITTED* read in Certify(SEQ_1) and Certify(SEQ_2) are the same. By the fact that $T.past$ and $T.rs$ are constant, the returned values of function Certify(SEQ_1) and Certify(SEQ_2) are the same for all $T \in SEQ_1 \cap SEQ_2$. □

Definition 6.6.1 *From Lemma 6.6.2, we can define* $COMMITTED_i^k$ *as the value of variable COMMITTED read at line 6 on* s_i *when* $K_i = k$. *From Lemma 6.6.3, we can define* $Certify(T,k)_i$ *as the returned value of the function Certify(SEQ) for* T ($T \in SEQ$), *called on* s_i *when* $K_i = k$. *If there exists no invocation of function Certify(SEQ) on* s_i *when* $K_i = k$ *such that* $T \in SEQ$, *we say that* $Certify(T,k)_i$ *is undefined, and we write* $Certify(T,k)_i = \perp$.

Definition 6.6.2 *We define* $vote(VQ,T,k)$, *voting quorum VQ's vote for transaction* T ($VQ \in VQS(T)$), *considering the VOTE messages of all* $q \in VQ$ *cast when* $K_q = k$ *as follows:*

- $vote(VQ,T,k) = yes$ *iff* $\forall s_i \in VQ : Certify(T,k)_i = yes$
- $vote(VQ,T,k) = no$ *iff* $\forall s_i \in VQ : Certify(T,k)_i \neq \perp \wedge$ $\exists s_j \in VQ : Certify(T,k)_j = no$
- $vote(VQ,T,k) = \perp$ *iff* $\exists s_i \in VQ : Certify(T,k)_i = \perp$

Lemma 6.6.4 *For all k and every transaction T, there does not exist* $VQ_1, VQ_2 \in VQS(T)$ *such that* $vote(VQ_1,T,k) = yes$ *and* $vote(VQ_2,T,k) = no$.

Proof: The proof is by induction on k.

- Base step (k = 1): When the algorithm initializes, the set *COMMITTED* is empty. Therefore, for all s_i such that $\text{Certify}(T,1)_i \neq \perp$, we have that $\text{Certify}(T,1)_i = yes$. Therefore, for all $VQ \in VQS(T)$ such that $vote(VQ,T,1) \neq \perp$, $vote(VQ,T,1) = yes$.
- Induction step: Suppose Lemma 6.6.4 holds for all m such that $1 \leq m < k$, we show that Lemma 6.6.4 holds for k. Suppose, by way of contradiction, that there exist a transaction T and two voting quorums VQ_1, $VQ_2 \in VQS(T)$, such that $vote(VQ_1,T,k) = yes$ and $vote(VQ_2,T,k) = no$. Therefore, for every database s_i in VQ_1, $\text{Certify}(T,k)_i = yes$ and there exists a database site s_j in VQ_2 such that $\text{Certify}(T,k)_j = no$. Therefore, there exists a transaction $T' \in COMMITTED_j^k$ such that $T'.ws \cap T.rs \neq \emptyset$ and $T'.order \geq T.past$. Let k' be the consensus instance in which T' was decided ($k' < k$). By the uniform agreement property of consensus, all databases decide on T' in instance k'. Since VQ_1 is a voting quorum for T and because $T'.ws \cap T.rs \neq \emptyset$, there exists a database site s_l in VQ_1 such that $Items(s_l) \cap T'.ws \neq \emptyset$. By the induction hypothesis, for all $VQ \in VQS(T') : vote(VQ,T',k') = yes$, therefore $T' \in COMMITTED_l^{k'+1}$. Thus, $\text{Certify}(T,k)_j = \text{Certify}(T,k)_l = no$, a contradiction. □

We now show that Algorithm $\mathscr{A}^{qg^*}_{pdsm}$ ensures 1-SR. A replicated data history H is 1-SR if it is equivalent to a one-copy serial history $1H$ [9]. Notice that in these histories we only consider operations of committed transactions. Our proof is composed of two steps. We first show that every history H_i of database s_i is serializable. Second we show how to construct a serial one-copy history $1H$ that is equivalent to the replicated data history H.

Lemma 6.6.5 *Every history H_i of database site s_i is serializable.*

Proof: In the following, we use the multi-version serialization graph formalism introduced in [9]. A multi-version serialization graph of a history H_i $MVSG(H_i, \ll)$ is a directed graph, in which the nodes represent committed transactions and $\ll$ defines the version order on data items. An edge $T_a \rightarrow T_b \in MVSG(H_i, \ll)$ can be of three types:

1. Read-from: T_b reads data item x from T_a.
2. Version-order type I: T_a and T_b write x such that $x_a \ll x_b$.
3. Version-order type II: T_a reads x_c from T_c and T_b writes x_b such that $x_c \ll x_b$.

To show that history H_i is serializable, we prove that $MVSG(H_i, \ll)$ is acyclic. To prove that $MVSG(H_i, \ll)$ is acyclic, we show that for every node T_a, T_b: $T_a \rightarrow T_b \in MVSG(H_i, \ll) \Rightarrow C(T_a) \prec_i C(T_b)$.[5]

1) *Read-from edge:* Let s_j be $Site(T_b)$. Because databases use the strict 2PL locking policy, T_b cannot read uncommitted data and therefore $C(T_a) \prec_j C(T_b)$. Now there are two cases to consider, either (i) T_b is a read-only or (ii) not.

- In case (i), since T_b is read-only, T_b *touches* a single site, and thus $s_i = s_j$. Therefore, $C(T_a) \prec_i C(T_b)$.
- In case (ii), since $C(T_a) \prec_j C(T_b)$, s_j decides on T_a in consensus instance k and on T_b in instance k' such that $k \leq k'$. Note that by Lemma 6.6.1, k and k' are uniquely defined on s_j. By the uniform agreement property of consensus s_i also decides on T_a in instance k and on T_b in instance k'. If $k = k'$, since $C(T_a) \prec_j C(T_b)$, T_a appears before T_b in the consensus decision, and thus, $C(T_a) \prec_i C(T_b)$. Otherwise, if $k < k'$, it follows directly that $C(T_a) \prec_i C(T_b)$.

[5] $\prec_i$ defines the commit order at site s_i

2) *Version-order edge type I:* Since the commit order induces the version order, we have that $C(T_a) \prec_i C(T_b)$.

3) *Version-order edge type II:* Suppose, by way of contradiction, that $C(T_b) \prec_i C(T_a)$. Let $Site(T_a)$ be s_j. When T_b requests the write locks to apply its updates at s_j, T_a can either be in the Executing state or in the Submitted state. We show that both cases lead to a contradiction:

- *Executing state:* There are two cases to consider, either (i) T_a is read-only or (ii) not.
  - In case (i), T_a touches a single site, and thus, $s_i = s_j$. According to rule 1 on how locks for T_b are granted, on s_i, T_b waits for T_a to commit, contradicting the fact that $C(T_b) \prec_i C(T_a)$.
  - In case (ii), according to rule 1 on how locks for T_b are granted, T_a is aborted at s_j. Hence, T_a is never submitted, a contradiction to the fact that T_a commits in H_i.
- *Submitted state:* Since $C(T_b) \prec_i C(T_a)$, s_i decides on T_b in consensus instance k and on T_a in instance k' such that $k \leq k'$. By Lemma 6.6.1, k and k' are uniquely defined on s_i. From the uniform agreement property of consensus, all sites that decide in instances k and k' decide on T_b in instance k and on T_a in instance k'. Now either (a) $k = k'$ or (b) $k < k'$.
  - In case (a), because $C(T_b) \prec_i C(T_a)$, T_b appears before T_a in the decision of consensus instance k, and thus T_b is in variable $LCOMMIT_i$ at line 38 when s_i tries to commit T_a. Because T_b writes x and T_a reads x, $T_b.ws \cap T_a.rs \neq \emptyset$ and therefore s_i does not commit T_a, a contradiction.
  - In case (b), since on s_j, T_a is in the Submitted state at the time T_b commits, (*) $T_a.past \leq T_b.order_j = k$. Since T_a commits, there exists a voting quorum VQ_1 for T_a such that $vote(VQ_1, T_a, T_a.order) = yes$. By Lemma 6.6.4, for all voting quorums $VQ \in VQS(T_a)$ such that $vote(VQ, T_a, T_a.order) \neq \perp$, $vote(VQ, T_a, T_a.order) = yes$. We now prove that there exists a voting quorum $VQ_2 \in VQS(T_a)$ such that $vote(VQ_2, T_a, T_a.order) = no$, contradicting Lemma 6.6.4.

    Because for all transactions T, there exists a correct voting quorum for T, there exists a correct voting quorum VQ_2 for T_a. By the definition of a correct voting quorum, there exists a correct site $s_r \in VQ_2$ such

that $x \in Items(s_r)$. Since s_i commits T_b, by Lemma 6.6.4, s_r commits T_b. Because s_j and s_r decide on T_b in instance k, $T_b \in COMMITTED_r^{k+1}$ and $T_b.order_j = T_b.order_r$. Hence, from (*), $T_a.past \leq T_b.order_r$. Consequently, $certify(T_a, T_a.order)_r = no$ and $vote(VQ_2, T_a, T_a.order) = no$. □

Proposition 6.6.1 (Safety) *There exists a serial one-copy history 1H that is equivalent to H.*

Proof: From Lemma 6.6.5 and because databases use strict two-phase locking, for each s_i, there is a serial execution $1H_i$ equivalent to H_i, where no operation of different transactions are interleaved and operations of transactions follow the order of the commits. By the uniform agreement property of consensus, databases agree on the sequence of update transactions. By the uniform integrity of Reliable Broadcast, for any transaction T and any site s_i, T can only be R-Delivered once on s_i and thus T can only be added once to $UNDECIDED_i$. Therefore, no database site proposes a sequence SEQ to consensus such that a transaction T appears more than once in SEQ. Hence, by Lemma 1, for all transactions T, T can only be committed once on s_i. We thus construct *1H* in the following way:

1. A read operation $r_i[x]$ of transaction T_i in H is mapped to the same operation $r_i[x]$ of T_i in *1H*. Write operations $w_i[x_A]$, $w_i[x_B]$, ..., $w_i[x_N]$ of transaction T_i in H is mapped to a single write operation $w_i[x]$ of transaction T_i in *1H*.

2. The commit order of the updates in *1H* follow the order defined by the consensus instances.

3. A read-only transaction T_q that commits just after the commit of an update transactions T_u on $Site(T_q)$ in H, appears after T_u and before any update transaction $T_{u'}$ in *1H*.

4. In *1H*, for any two transactions T_i and T_j, their respective operations do not interleave.

We now show that *1H* is view equivalent to H. For *1H* to be view equivalent to H, two conditions have to be fulfilled [9]:

1. H and *1H* have the same *read-x-from* relationships on data items: $\forall T_i, T_j$: T_j *read-x-from* T_i in $H \iff T_j$ *read-x-from* T_i in *1H*.

2. For each final write $w_i[x]$ in *1H*, $w_i[x_A]$ is also a final write in H for some copy x_A of x.

H is view equivalent to *1H*:

1. ($\Rightarrow$) Let T_i, T_j be two transactions such that T_j *read-x-from* T_i in H. We prove that T_j *read-x-from* T_i in *1H*. There are two cases to consider, either (a) T_j is an update transaction or (b) T_j is a read-only transaction. In case (a), either (a-i) T_i and T_j are decided in the same consensus instance k or (a-ii) T_i and T_j are decided in consensus instance k and k'.
   (a-i) By the fact that databases use the strict 2PL locking policy, T_j cannot read uncommitted data, and therefore $Site(T_j)$ cannot submit T_j before T_i commits. Hence T_i and T_j cannot be decided in the same consensus instance k.
   (a-ii) In H, by Lemma 6.6.5 and the fact that databases use strict two-phase locking, there exists no transaction T_a that updates data item x that commits between instance k and k', otherwise T_j would not *read-x-from* T_i in H. Therefore, by construction step 2 of *1H*, there exists no such transaction T_a in *1H* and therefore by step 4 of *1H*, T_j *read-x-from* T_i in *1H*. In case (b), by using construction step 3 and 4 of *1H*, we conclude that T_j *read-x-from* T_i in *1H*.

   ($\Leftarrow$) Let T_i, T_j be two transactions such that T_j *read-x-from* T_i in *1H*. We prove that T_j *read-x-from* T_i in H. If T_j *read-x-from* T_i in *1H*, by construction step 4, (*) there exists no transaction T_a that commits between T_i and T_j and updates data item x in *1H*. There are two cases to consider, either (a) T_j is an update transaction or (b) T_j is a read-only transaction. In case (a), either (a-i) T_i and T_j are decided in the same consensus instance k or (a-ii) T_i and T_j are decided in consensus instance k and k'.
   (a-i) For the same reason as in ($\Rightarrow$), this case is impossible.
   (a-ii) By (*) and construction step 2 of *1H*, there exists no consensus instance k'', $k < k'' < k'$, such that T_a is decided in consensus instance k'. Therefore, by Lemma 6.6.5 and by the fact that databases use strict two-phase locking, T_j *read-x-from* T_i in H. In case (b), by using construction step 3, by Lemma 6.6.5 and by the fact that databases use strict two-phase locking, we conclude that T_j *read-x-from* T_i in H.

2. Clear from the fact that both histories contain the same update transactions and the fact that update transactions in *1H* follow the order of up-

date transactions in H. □

Lemma 6.6.6 *For any correct database site s_i, s_i does not wait forever at line 36.*

Proof: If s_i waits at line 36, s_i executed *Propose(K, -)*. By the termination property of consensus, all correct sites decide in instance K. Let *SEQ* be the decision of consensus instance K. Because there exists a correct voting quorum *VQ* for every transaction T and because channels are quasi-reliable, after all sites in *VQ* executed line 29, all correct sites eventually receive enough VOTE message for all $T' \in SEQ$ to constitute a voting quorum. Therefore, s_i eventually stop waiting at line 36. □

Lemma 6.6.7 *For any submitted transaction T, if a correct database site s_i R-Delivers(VOTE_REQ, T), then there exists a k such that consensus instance k decides on T.*

Proof: Suppose, by way of contradiction, that there exists a transaction T such that a correct database site s_i R-Delivers(VOTE_REQ, *T)* and no consensus instance decides on T. If s_i R-Delivers(VOTE_REQ, T), by the agreement property of Reliable Broadcast and because s_i is correct, every correct database s_j eventually R-Delivers(VOTE_REQ, T). Since no consensus instance k decides on T, it is always true that $T.id \notin DECIDED_j$ and therefore s_j adds T to $UNDECIDED_j$ at line 24. By the termination property of consensus, by Lemma 6.6.6, and because for all s_j, T is in $UNDECIDED_j$ forever, all s_j start infinitely many consensus instances at line 27. Let t_1 be the time after which all the faulty databases have crashed and let t_2 be the time at which the last correct site adds T to *UNDECIDED*. After t_1 and t_2, all values proposed to the consensus instances contain T, therefore by the validity and termination property of consensus, there exists a consensus instance k that decides on *SEQ* such that $T \in SEQ$, a contradiction. □

Proposition 6.6.2 (Liveness) *For any submitted transaction T and any correct database site s_i, if s_i submit(T) then eventually either all correct database sites $s_j \in Replicas(T)$ commit T or s_i aborts T.*

Proof: Since s_i is correct, s_i R-Broadcasts(VOTE_REQ, T) and by the validity property of Reliable Broadcast, s_i eventually R-Delivers(VOTE_REQ, T). By Lemma 6.6.7, there exists a k such that consensus instance k decides on T. By Lemma 6.6.6, no correct database sites $s_j \in Replicas(T)$ waits forever at line 36. Therefore, by Lemma 6.6.4, either all sites s_j commit T or s_i aborts T. □

Proposition 6.6.3 (Quasi-Genuine Partial Replication) *For every submitted transaction T, correct database sites that do not replicate data items read or written by T permanently store not more than the identifier of T.*

Proof: From the algorithm, information other than the identifier of T is stored (a) in the *UNDECIDED* sequence, (b) in the *COMMITTED* set, (c) in the consensus instances, and (d) in the *VOTES* set. Notice that in [15], the implementation of Reliable Broadcast only keeps the identifiers of messages.
(a) First, notice that because task 3 executes the last two *when blocks*, these *when blocks* are executed atomically. A correct site s_i can only add a transaction T to $UNDECIDED_i$ at line 24 if s_i R-Delivers(VOTE_REQ, T). By the integrity property of Reliable Broadcast, s_i can only R-Deliver(VOTE_REQ, T) once. If s_i adds T to $UNDECIDED_i$ at line 24, $T.id \notin DECIDED_i$. If $T.id \notin DECIDED_i$, s_i has not executed line 31 yet. By Lemma 6.6.7, eventually all correct database sites decide on T and s_i removes T from $UNDECIDED_i$ at line 32.

(b) A correct site s_i adds T to $COMMITTED_i$ only if $T'.ws \cap Items(s_i) \neq \emptyset$.

(c) We here only give an example of a consensus algorithm that satisfies this property. Consider the consensus algorithm given in [54]. It can be used in our termination protocol because it assumes the same system model, namely sites are crash-stop and channels are quasi-reliable. In this algorithm, after a correct site *delivers* and processes the *decide* message (by the termination property of consensus, this eventually happens), it stops taking part in the algorithm, and therefore all the variables of that consensus instance can be safely *garbage collected*.

(d) If for some correct site s_i $VOTES_i \neq \emptyset$, by the condition of line 25, s_i proposes a sequence of transactions to consensus. By the termination property of consensus, s_i eventually decides. By Lemma 6.6.6, s_i does not wait forever at line 36 and s_i eventually deletes all votes from the *VOTES* set that concern the current and previous consensus instances. □

Proposition 6.6.4 (Non-Trivial Certification) *If there is a time after which no two conflicting transactions are submitted, then eventually transactions are not aborted by certification.*

Proof: Let t_1 be the time after which no two conflicting transactions are submitted. Let $t_2 > t_1$ be the time after which the last transaction T submitted before t_1

commits. Let $t_3 > t_2$ be the time after which all transactions T' submitted after t_1 are such that $T.order < T'.past$. Because after t_1, no two conflicting transactions are submitted, after t_3 we have that:

1. All call of function Certify return *yes* and the condition at line 37-38 always evaluates to true,
2. No transaction is aborted by rules 1 and 2 of Section 6.3 on how locks are granted.

Thus, after t_3, no transaction is aborted by certification. □

6.6.2 The Proof of Algorithm $\mathscr{A}^{ge}_{pdbsm}$

Definition 6.6.3 *We define the binary relation $\rightsquigarrow$ on transactions as follows: $T_1 \rightsquigarrow T_2$ iff $\exists s_i \in \Pi : s_i$ A-Delivers T_1 before T_2 and $T_1.ws \cap Items(T_2) \neq \emptyset$. Moreover, let $\mathscr{G}_{\rightsquigarrow(T)} = (V, E)$ be a finite DAG constructed as follows:*

1. *add vertex T to V*
2. *while $\exists T_1 \in V : \exists T_2 \notin V : T_2 \rightsquigarrow T_1$ do:*
   add T_2 to V and add directed edge $T_2 \rightarrow T_1$ to E

For any transaction T' in $\mathscr{G}_{\rightsquigarrow(T)}$, we say that T' is at distance k of T iff the longest path from T' to T is of length k. We let $\mathscr{T}_k$ be the subset of transactions in $\mathscr{G}_{\rightsquigarrow(T)}$ that are at distance k of T.

Lemma 6.6.8 *For any submitted transaction T, $\mathscr{G}_{\rightsquigarrow(T)}$ is acyclic.*

Proof: Follows directly from the uniform acyclic order property of atomic multicast.

Definition 6.6.4 *We define $Certify(T)_i$ as the returned value of the function Certify(T) called on s_i. If there exists no invocation of function Certify(T) on s_i we say that Certify(T) is undefined, and we write $Certify(T)_i = \perp$.*

Definition 6.6.5 *We define $vote(VQ, T)$, voting quorum VQ's vote for transaction T ($VQ \in VQS(T)$), considering the VOTE messages cast of all $q \in VQ$ as follows:*

- $vote(VQ, T) = yes$ *iff* $\forall s_i \in VQ : Certify(T)_i = yes$
- $vote(VQ, T) = no$ *iff* $\forall s_i \in VQ : Certify(T)_i \neq \perp \wedge$
  $\exists s_j \in VQ : Certify(T)_j = no$

- $vote(VQ, T) = \perp$ *iff* $\exists s_i \in VQ$: $Certify(T)_i = \perp$

Definition 6.6.6 *We define TS_i^T as the value of TS on site s_i after s_i executed line 4 for transaction T. If s_i never executes line 4 for transaction T, then $TS_i^T = \perp$.*

Lemma 6.6.9 *For any submitted transaction T:*

1. *There does not exist $VQ_1, VQ_2 \in VQS(T)$ such that $vote(VQ_1, T) =$ yes and $vote(VQ_2, T) =$ no.*
2. *For any data item $x \in T.ws$, for any two sites s_i, s_j such that $x \in Items(s_i)$ and $x \in Items(s_j)$, if $TS_i^T \neq \perp$ and $TS_j^T \neq \perp$, then $TS_i^T[x] = TS_j^T[x]$.*

Proof: Let $\mathcal{T}_k$ be the subset of the transactions in $\mathcal{G}_{\rightsquigarrow(T)}$ that are at distance k of T. We show that for any k and any $T' \in \mathcal{T}_k$, 1. and 2. hold. Since $T \in \mathcal{T}_0$, this shows the two claims. Let k_{max} be the largest k such that $\mathcal{T}_k \neq \emptyset$. We proceed by simultaneous induction on 1. and 2, starting from k_{max}.

- Base step ($k = k_{max}$):

  1. From the definition of $\mathcal{T}_{k_{max}}$, there exists no transaction $T' \in \mathcal{T}_{k_{max}}$ such that a site s_i A-Delivers a transaction T'' before T' and $T''.ws \cap Items(T')$. Hence, on all sites s_i such that $Items(s_i) \cap T'.rs \neq \emptyset$ and that certify T', T' passes the certification test. Hence, for all $VQ \in VQS(T')$, $vote(VQ, T') =$ yes.
  2. From the definition of $\mathcal{T}_{k_{max}}$ and the algorithm, for any transaction $T' \in \mathcal{T}_{k_{max}}$ and any data item $x \in T'.ws$, on all sites s_i such that $x \in Items(s_i)$ and $TS_i^{T'}$ is defined, $TS_i^{T'}[x] = 1$.

- Induction step: Suppose that the two claims hold for any k such that $0 < k \leq k_{max}$, we show that they also hold for $k - 1$. Let T' be any transaction in $\mathcal{T}_{k-1}$.

  1. Suppose, by way of contradiction, that $vote(VQ_1, T') =$ yes and $vote(VQ_2, T') =$ no. Hence, there exists a site $s_i \in VQ_2$ such that $\text{Certify}(T')_i =$ no. Consequently, there exists a data item x read by T' such that $T'.past[x] \neq TS_i^{T'}[x]$. From the algorithm, $TS[x]$ is monotonically increasing with time, and thus, $T'.past[x] < TS_i^{T'}[x]$. Hence, there exists a transaction T_1 such that s_i A-Delivers T_1 just before T', T_1 commits and updates x, and $TS_i^{T_1}[x] = T'.past[x]$

(just after T_1 commits, $TS_i[x]$ is incremented and becomes greater than $T'.past[x]$). From the definition of a voting quorum, there exists a site $s_j \in VQ_1$ such that $x \in Items(s_j)$, and thus, T_1 is also multicast to s_j. From the uniform prefix order property of atomic multicast, either (i) s_i A-Delivers T' before T_1 or (ii) s_j A-Delivers T_1 before T'. Case (i) is impossible as s_i would A-Deliver T' twice, a contradiction to the uniform integrity property of atomic multicast. Therefore, s_j A-Delivers T_1 before T'. From the induction hypothesis of 1, all voting quorums for T_1 vote similarly, and thus, since s_i commits T_1, s_j commits T_1 as well. From the induction hypothesis of 2, $TS_i^{T_1}[x] = TS_j^{T_1}[x]$. Hence, since s_j A-Delivers T_1 before T', s_j increments $TS[x]$ to a greater value than $T'.past[x]$ before A-Delivering T', and Certify$(T')_j$ = no, a contradiction to the fact that $s_j \in VQ_1$ and $vote(VQ_1, T')$ = yes. □

2. Let T_1 be the last transaction that commits on s_i before T' such that T_1 updates x. Using a similar argument as in the induction step of 1, we can show that s_j A-Delivers T_1 before T'. From the induction hypothesis of 1, T_1 commits on s_j.

   We now show that (*), on s_j, T_1 is also the last transaction that updates x and commits before T'. Suppose, by way of contradiction, that there exists a transaction T_2 that commits after T_1 and before T' on s_j such that $x \in T_2.ws$. From the uniform prefix order of atomic multicast, either (i) s_i A-Delivers T_2 before T' or (ii) s_j A-Delivers T' before T_2. Case (ii) is impossible as s_j would A-Deliver T' twice, a contradiction to the uniform integrity property of atomic multicast. Hence, s_i A-Delivers T_2 before T'. Now, either (iii) s_i A-Delivers T_1 before T_2 or (iv) s_j A-Delivers T_2 before T_1. Case (iv) is impossible for the same reason as case (ii). Therefore, s_i A-Delivers, in order, T_1, T_2, and T'. From the induction hypothesis of 1, s_i also commits T_2. Consequently, the last transaction that commits before T' on s_i and updates x is T_2, a contradiction to the definition of T_1.

   By the induction hypothesis of 2, $TS_i^{T_1}[x] = TS_j^{T_1}[x]$. Therefore, from (*) and the algorithm, $TS_i^{T'}[x] = TS_j^{T'}[x]$. □

Lemma 6.6.10 *Every history H_i of database site s_i is serializable.*

Proof: The proof is similar to Lemma 6.6.5. We use the multi-version serialization graph formalism introduced in [9] to show that H_i is serializable. In order

to do that, we prove that $MVSG(H_i, \ll)$ is acyclic. To show that $MVSG(H_i, \ll)$ is acyclic, we show that for every node T_a, T_b: $T_a \rightarrow T_b \in MVSG(H_i, \ll) \Rightarrow C(T_a) \prec_i C(T_b)$.[6]

1) *Read-from edge:* Let s_j be $Site(T_b)$. Because databases use the strict 2PL locking policy, T_b cannot read uncommitted data and therefore $C(T_a) \prec_j C(T_b)$. Now there are two cases to consider, either (i) T_b is a read-only or (ii) not.

- In case (i), since T_b is read-only, T_b *touches* a single site, and thus $s_i = s_j$. Therefore, $C(T_a) \prec_i C(T_b)$.
- In case (ii), since $C(T_a) \prec_j C(T_b)$, s_j A-Delivers T_a before T_b. Hence, from the uniform prefix order property of atomic multicast, either (i) s_i A-Delivers T_a before T_b or (ii) s_j A-Delivers T_b before T_a. Case (ii) is impossible as s_j would A-Deliver T_b twice, a contradiction to the uniform integrity property of atomic multicast. Therefore, s_i A-Delivers T_a before T_b, and thus, $C(T_a) \prec_i C(T_b)$.

2) *Version-order edge type I:* Since the commit order induces the version order, we have that $C(T_a) \prec_i C(T_b)$.

3) *Version-order edge type II:* Suppose, by way of contradiction, that $C(T_b) \prec_i C(T_a)$. Let $Site(T_a)$ be s_j. When T_b requests the write locks to apply its updates at s_j, T_a can either be in the Executing state or in the Submitted state. We show that both cases lead to a contradiction:

- *Executing state:* There are two cases to consider, either (i) T_a is read-only or (ii) not.
  - In case (i), T_a touches a single site, and thus, $s_i = s_j$. According to rule 1 on how locks for T_b are granted, on s_i, T_b waits for T_a to commit, contradicting the fact that $C(T_b) \prec_i C(T_a)$.
  - In case (ii), according to rule 1 on how locks for T_b are granted, T_a is aborted at s_j. Hence, T_a is never submitted, a contradiction to the fact that T_a commits in H_i.
- *Submitted state:* Since at the time T_b commits, T_a is in the Submitted state, (*) $T_a.past[x] < TS_j^{T_a}[x]$. Since T_a commits in H_i, there exists a voting quorum VQ_1 for T_a such that $vote(VQ_1, T_a, T_a.order) = yes$. By Lemma 6.6.9, for

[6] $\prec_i$ defines the commit order at site s_i

all voting quorums $VQ \in VQS(T_a)$ such that $vote(VQ, T_a) \neq \perp$, $vote(VQ, T_a) = yes$. We now prove that there exists a voting quorum $VQ_2 \in VQS(T_a)$ such that $vote(VQ_2, T_a) = no$, contradicting Lemma 6.6.9.

Because for all transactions T, there exists a correct voting quorum for T, we let VQ_2 be a correct voting quorum for T_a. By the definition of a correct voting quorum, there exists a correct site $s_r \in VQ_2$ such that $x \in Items(s_r)$. Hence, since s_r is correct, from the uniform agreement property of atomic multicast, s_r A-Delivers T_a. From Lemma 6.6.9, $TS_j^{T_a}[x] = TS_r^{T_a}[x]$. Therefore, from (*), $T_a.past[x] < TS_r^{T_a}[x]$, and Certify$(T_a)_r =$ no. Thus, $vote(VQ_2, T_a) = no$. □

Proposition 6.6.5 (Safety) *There exists a serial one-copy history 1H that is equivalent to H.*

Proof: From Lemma 6.6.10 and because databases use strict two-phase locking, for each s_i, there is a serial execution $1H_i$ equivalent to H_i, where no operation of different transactions are interleaved and operations of transactions follow the order of the commits. We thus construct *1H* in the following way:

1. A read operation $r_i[x]$ of transaction T_i in H is mapped to the same operation $r_i[x]$ of T_i in *1H*. Write operations $w_i[x_A]$, $w_i[x_B]$, ..., $w_i[x_N]$ of transaction T_i in H is mapped to a single write operation $w_i[x]$ of transaction T_i in *1H*.

2. The commit order of update transactions in *1H* is defined using the partial order relation $<$ on the set of transactions A-Delivered. Recall that this relation is defined as follows: for any two transactions T_1 and T_2, $T_1 < T_2$ iff there exists a site that A-Delivers T_1 before T_2. If relation $<$ does not order T_1 and T_2, then these transactions can appear in any order in $1H$. From the properties of atomic multicast, transactions updating the same data item x will thus be totally ordered.

3. A read-only transaction T_q that commits just after the commit of an update transactions T_u on $Site(T_q)$ in H, appears after T_u and before any update transaction $T_{u'}$ in *1H*.

4. In *1H*, for any two transactions T_i and T_j, their respective operations do not interleave.

We now show that *1H* is view equivalent to H. For *1H* to be view equivalent to H, two conditions have to be fulfilled [9]:

1. H and *1H* have the same *read-x-from* relationships on data items: $\forall T_i, T_j$: T_j *read-x-from* T_i in $H \iff T_j$ *read-x-from* T_i in *1H*.

2. For each final write $w_i[x]$ in *1H*, $w_i[x_A]$ is also a final write in H for some copy x_A of x.

H is view equivalent to *1H*:

1. ($\Rightarrow$) Let T_i, T_j be two transactions such that T_j *read-x-from* T_i in H. We prove that T_j *read-x-from* T_i in *1H*. There are two cases to consider, either (a) T_j is an update transaction or (b) T_j is a read-only transaction. In case (a), by Lemma 6.6.10 and the fact that databases use strict two-phase locking, in H, there exists no transaction T_a that updates data item x that commits between the commit of T_i and T_j, otherwise T_j would not *read-x-from* T_i in H. Therefore, by construction step 2 of *1H*, there exists no such transaction T_a in *1H* and hence, by step 4 of *1H*, T_j *read-x-from* T_i in *1H*. In case (b), by using construction step 3 and 4 of *1H*, we conclude that T_j *read-x-from* T_i in *1H*.

   ($\Leftarrow$) Let T_i, T_j be two transactions such that T_j *read-x-from* T_i in *1H*. We prove that T_j *read-x-from* T_i in H. If T_j *read-x-from* T_i in *1H*, by construction step 4, (*) there exists no transaction T_a that commits between T_i and T_j and updates data item x in *1H*. There are two cases to consider, either (a) T_j is an update transaction or (b) T_j is a read-only transaction. In case (a), by (*) and construction step 2 of *1H*, there exists no transaction T_a that updates data item x such that there exist sites s_i and s_j ($s_i = s_j$ is possible) and s_i A-Delivers T_i before T_a and s_j A- Delivers T_a before T_j. Therefore, by Lemma 6.6.10 and by the fact that databases use strict two-phase locking, T_j *read-x-from* T_i in H. In case (b), by using construction step 3, by Lemma 6.6.10 and by the fact that databases use strict two-phase locking, we conclude that T_j *read-x-from* T_i in H.

2. Clear from the fact that both histories contain the same update transactions and the fact that update transactions in *1H* follow the order of update transactions in H (construction step 2). □

Lemma 6.6.11 *For any correct database site s_i, s_i does not wait forever at line 14.*

Proof: If s_i waits at line 14, then (*) s_i A-Delivered some transaction T. Since there is a correct voting quorum for all transactions, there is a correct voting

quorum $VQ \in VSQ(T)$. From (*) and the uniform agreement property of atomic multicast, all sites $s_j \in VQ$ eventually A-Deliver T. Since links are quasi-reliable, s_i eventually receives the votes from all processes in VQ and stops waiting at line 14. □

Proposition 6.6.6 (Liveness) *For any submitted transaction T and any correct database site s_i, if s_i submit(T) then eventually either all correct database sites $s_j \in Replicas(T)$ commit T or s_i aborts T.*

Proof: Since s_i is correct, s_i A-MCasts(VOTE_REQ, T) and by the validity property of atomic multicast, all correct sites s_j eventually wait for a voting quorum for T. By Lemma 6.6.11, sites s_j eventually stop waiting and therefore, by Lemma 6.6.9, either all sites $s_j \in Replicas(T)$ commit T or s_i aborts T. □

Proposition 6.6.7 (Genuine Partial Replication) *Using the atomic multicast algorithm $\mathscr{A}_{ge}^{dv}$ or $\mathscr{A}_{ge}^{dt}$, for every submitted transaction T, database sites that do not replicate data items read or written by T do not store any information about T.*

Proof: If a site s_i does not replicate any data item read or written by T, then T is not executed on s_i, T's vote request message is not multicast to s_i, and T's votes are not sent to s_i. When using the genuine atomic multicast algorithm $\mathscr{A}_{ge}^{dv}$ or $\mathscr{A}_{ge}^{dt}$, since to A-Deliver T, only sites addressed by T store information about T, s_i does not store any information about T. □

Proposition 6.6.8 (Non-Trivial Certification) *If there is a time after which no two conflicting transactions are submitted, then eventually transactions are not aborted by certification.*

Proof: Let t_1 be the time after which no two conflicting transactions are submitted. Let $t_2 > t_1$ be the time after which the last transaction T submitted before t_1 commits. We claim that no transaction submitted after t_2 is aborted by certification. Indeed, for any transaction T submitted after t_2, we have that:

1. The call of function Certify return *yes* and the condition at line 15 always evaluates to true. This is because on any site s_i that certifies T, and for any data item x read by T and replicated by s_i, $T.past[x] = TS_i^T[x]$.

2. No transaction is aborted by rules 1 and 2 of Section 6.3 on how locks are granted.

Chapter 7

Conclusion

> Do not seek the truth. Only cease to cherish opinions.
>
> *Seng-ts'an*

Recent years have seen the rapid proliferation of web-based applications such as e-banking, social networks, and e-commerce platforms. As a consequence, our lives depend on these systems more and more each day. In this context, these applications must provide high availability, to provide interrupted service, and scalability, to handle the ever increasing client demand.

Previous proposals suggested to provide high-availability by fully replicating the application's data. Many of these solutions were built on top of group communication primitives that provide precise and easy-to-understand guarantees. Replication protocols then relied on high-level synchronization constructs offered by this communication layer to ensure global data consistency. Despite this clear separation of concerns, these solutions suffered from an inherent scalability bottleneck as every update must be applied to all replicas.

Partial replication is a promising alternative to improve scalability: independent requests may be handled by different parts of the system in parallel. Additionally, partial replication requires less storage resources since data only needs to be replicated close to the clients accessing it.

This thesis investigates the minimal latency cost and failure detection requirements of multicast, a primitive at the core of partial replication, and presents several multicast algorithms for the case of correct and faulty groups. Partial replication protocols built on top of these primitives are also proposed.

7.1 Research Assessment

This thesis presents four contributions: (i) FIFO and causal multicast protocols, (ii) atomic multicast algorithms, (iii) the experimental performance evaluation of some of these algorithms, and (iv) partial replication protocols that are built on top of the above multicast primitives. These contributions can be classified into two domains: the first three belong to the distributed systems domain and the last one to the database domain.

Fifo and Causal Multicast. Fifo and causal multicast are powerful communication primitives that ease the programming of distributed applications. In this thesis, the latter primitive was used in the atomic multicast algorithm $\mathscr{A}_{ge}^{dt}$ to ensure a hole-free message delivery sequence.

Although these communication abstractions have been studied extensively, previously proposed solutions either did not tolerate quasi-reliable networks, in which a message sent can be lost because of the crash of its sender, or disallowed messages to be sent to groups the sender does not belong to. To address these limitations, we proposed algorithms that tolerate quasi-reliable networks, allow messages to be multicast to any subset of groups, and tolerate an arbitrary number of process failures. We showed that these protocols are latency-optimal.

Atomic Multicast. We devised atomic multicast algorithms in large scale networks that minimize the number of inter-group message delays between the multicast of a message and its delivery. Two different settings have been considered: correct groups, in which at least one member of each group is correct, and groups that may crash entirely. In the first setting, we showed that genuine atomic multicast is more expensive than its non-genuine counterpart: multicast requires a minimum of two inter-group message delays and we presented a non-genuine multicast protocol that needs a single inter-group delay. The multicast lower bound is tight, the algorithm $\mathscr{A}_{ge}^{dv}$ of Section 4.2.2 and the algorithm in [28] achieve a latency of two inter-group message delays (as opposed to [28], $\mathscr{A}_{ge}^{dv}$ reduces the number of intra-group messages sent).

In the second setting, we provided a genuine algorithm that tolerates an arbitrary number of process failures but does not tolerate erroneous process failure suspicions. As a corollary of [22], this failure detection accuracy is necessary if we disallow process crashes to be predicted. Since it seems hard, if not impossible, to ensure that no spurious failure suspicions occur in large networks, we presented a non-genuine protocol that weakens the failure detection require-

ments and allows to deliver messages in two inter-group delays. As a corollary of the hyperfast learning Lemma in [38], this is optimal. Altogether, these results revealed that the genuineness of multicast is an expensive property, both in terms of latency and failure detection accuracy.

Evaluation of Multicast Protocols. In the context of correct groups, we showed that genuine multicast is more expensive than its non-genuine counterpart: the former variant requires an extra inter-group communication step compared to the latter variant. To determine under which circumstances a genuine algorithm is more efficient than a non-genuine algorithm, we experimentally evaluated the performance of the latency-optimal and disaster-vulnerable genuine and non-genuine algorithms proposed in this thesis.

The results revealed that genuine multicast is interesting only in large and highly loaded systems; in all the other considered scenarios the non-genuine protocol $\mathscr{A}_{ng}^{dv}$ outperformed the optimal genuine algorithm. To complete our study, we measured the overhead of the disaster-tolerant and latency-optimal multicast protocol $\mathscr{A}_{ng}^{dt}$ and observed that although it is in general more costly than the two other disaster-vulnerable protocols, it matched the performance of the genuine algorithm when there are few groups. We also identified a *convoy effect* in multicast algorithms that delay the delivery of messages and proposed techniques to reduce this effect.

Partial Replication. Partial replication was proposed as an alternative to full replication to provide better scalability: database sites store a subset of the application data and, consequently, sites only need to apply updates related to the data items they replicate. We extended the database state machine (DBSM) [47] to partial replication. In the DBSM, transactions are executed locally on database sites and a certification protocol is triggered at the end of each transaction to ensure global consistency.

We proposed two properties that characterize the legitimacy of partial replication protocols. The first property, quasi-genuine partial replication, allows for broadcast-based implementations and ensures that sites unrelated to a transaction T only permanently store the identifier of T. The second property, genuine partial replication, guarantees that sites unrelated to a transaction T do not store any information about T. We presented two termination protocols tailored for local area networks that ensure quasi-genuine partial replication. We also demonstrated how atomic multicast can be used to build a genuine partial replication protocol. This protocol is more generic than the first two: it is not

optimized for any particular type of network.

7.2 Future Directions

The research conducted during this thesis has raised several problems that deserve further investigation. In the following, we describe future directions and open questions related to this research.

Genuine Atomic Multicast. The ability to tolerate disasters comes at a high price for the genuine atomic multicast algorithm $\mathscr{A}_{ge}^{dt}$. This protocol requires six inter-group message delays to deliver global messages. Is this cost minimal? Based on the fact that the non-genuine variant $\mathscr{A}_{ng}^{dt}$ only needs two inter-group delays, we believe that there is room for improvement. Orthogonally, we have observed that multicast protocols are subject to an undesired behavior we denoted as convoy effect. This behavior can lead to delay the delivery of local and global messages by as much as the latency of global messages. Although we proposed techniques to reduce this effect for local and global messages in the non-genuine algorithms $\mathscr{A}_{ng}^{dv}$ and $\mathscr{A}_{ng}^{dt}$, we only tackled this problem for local messages in the genuine protocol $\mathscr{A}_{ge}^{dv}$. In fact, it is not known whether the convoy effect of global messages in genuine atomic multicast can be avoided.

Generic Multicast. Generic broadcast permits applications to specify the desired message ordering semantics. Originally, this abstraction was introduced to reduce the latency of commutable messages by removing consensus from the critical path. In the context of large scale networks, generic multicast algorithms should reduce the number of inter-group delays for messages that do not need to be ordered. It can easily be shown that, in the case of correct groups, genuine generic multicast has the same minimal cost as genuine atomic multicast, even when all messages are commutable. Intuitively, this holds for the following reason: before delivering some messages m, any protocol must check that no other process has delivered a message m' that does not commute with m. Hence, one inter-group delay is needed to propagate m, and another is required to perform this check, leading to two inter-group message delays. Nevertheless, relaxed ordering semantics could be exploited in the case of data center disasters to offer faster message delivery.

The Efficiency of Partial Replication Protocols. We conducted experiments in various scenarios to determine how genuine and non-genuine atomic multicast

compare. The results indicated that the genuineness of multicast in interesting only in large and highly loaded systems. A similar study must be carried out to understand how the genuine and quasi-genuine partial replication protocols perform, and whether they compare similarly as the atomic multicast algorithms. Other issues such as allowing transactions to be distributed without overly impacting the transaction abort rate is also a topic to be investigated.

Partial Replication with Snapshot Isolation. Recently, partial replication protocols that offer a generalized form of snapshot isolation have been proposed [59; 7]. This consistency criterion never blocks nor aborts read-only transactions and update transactions are never blocked nor aborted due to read-only transactions. To ensure that transactions read valid snapshots, protocols either take a dummy snapshot after every commit [59] or atomically broadcast a *snapshot* message to all replicas before the transaction starts [7]. Constructing consistent snapshots more efficiently is an open problem.

Byzantine Partial Replication. Researchers recently focused on improving the performance of full replication protocols in environments where processes may behave maliciously [35; 41; 43; 63]. We have seen that atomic multicast is a primitive of choice to build partial replication protocols. A natural question is whether the atomic multicast protocols presented in this thesis can be adapted to untrusted environments. Interestingly, the notion of genuine atomic multicast may cause problems in this context: consider some application p hosted on a machine M_p that wishes to multicast a message m to groups g_1 and g_2. If M_p is malicious, M_p may behave as if m was addressed to g_1 only. In particular, M_p may omit sending m to g_2. In this scenario, the uniform agreement property of atomic multicast is violated since only members of g_1 can deliver m. A solution to this problem is to let the application p sign m. However, this adds overhead to the protocol. Devising efficient genuine atomic multicast algorithms is thus an open problem.

Bibliography

[1] Transaction processing performance council (tpc) - benchmark c. http://www.tpc.org/tpcc/.

[2] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In *Proceedings of Euro-Par'97*, pages 496–503. Springer-Verlag, 1997.

[3] M. K. Aguilera and R. E. Strom. Efficient atomic broadcast using deterministic merge. In *Proceedings of PODC '00*, pages 209–218. ACM Press, 2000.

[4] M. K. Aguilera, S. Toueg, and B. Deianov. Revising the weakest failure detector for uniform reliable broadcast. In *Proceedings of DISC'99*, pages 19–33. Springer-Verlag, 1999.

[5] M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Thrifty generic broadcast. In *Proceedings of DISC'00*, pages 268–283. Springer-Verlag, 2000.

[6] S. Alagar and S. Venkatesan. An optimal algorithm for distributed snapshots with causal message ordering. *Information Processing Letters*, 50(6):311–316, 1994.

[7] J. E. Armendáriz, A. Mauch-Goya, J. R. González de Mendívil, and F. D. Muñoz. Sipre: a partial database replication protocol with si replicas. In *Proceedings of SAC '08*, pages 2181–2185, New York, NY, USA, 2008. ACM.

[8] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of PODC'83*, pages 27–30, 1983.

[9] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[10] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[11] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[12] L. Camargos. http://sourceforge.net/projects/daisylib/.

[13] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-jdbc: Flexible database clustering middleware. In *USENIX Annual Technical Conference, FREENIX Track*, pages 9–18, 2004.

[14] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

[15] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[16] B. Charron-Bost and A. Schiper. The Heard-Of Model: Unifying all Benign Failures. *Distributed Computing*, To appear.

[17] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the v kernel. *ACM Trans. Comput. Syst.*, 3(2):77–107, 1985.

[18] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys*, 33(4):427–469, 2001.

[19] C. Coulon, E. Pacitti, and P. Valduriez. Consistency management for partial replication in a high performance database cluster. In *Proceedings of ICPADS'05*, volume 1, pages 809–815, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[20] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[21] C. Delporte-Gallet and H. Fauconnier. Fault-tolerant genuine atomic multicast to multiple groups. In *Proceedings of OPODIS'00*, pages 107–122. Suger, Saint-Denis, rue Catulienne, France, 2000.

[22] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. A realistic look at failure detectors. In *Proceedings of DSN'02*, pages 345–353. IEEE Computer Society, 2002.

[23] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Shared Memory vs Message Passing. Technical Report LPD-2003/001, EPFL, 2003.

[24] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *Proceedings of PODC'04*, pages 338–346, 2004.

[25] C. Delporte-Gallet, H. Fauconnier, J.-M. Helary, and M. Raynal. Early stopping in global data computation. *IEEE Transactions on Parallel and Distributed Systems*, 14(9):909–921, 2003.

[26] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[27] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.

[28] U. Fritzke, Ph. Ingels, A. Mostéfaoui, and M. Raynal. Fault-tolerant total order multicast to asynchronous groups. In *Proceedings of SRDS'98*, pages 578–585. IEEE Computer Society, 1998.

[29] S. Frolund and F. Pedone. Ruminations on domain-based reliable broadcast. In *Proceedings of DISC'02*, pages 148–162. Springer-Verlag, 2002.

[30] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems*, 31(1):133–160, 2006.

[31] R. Guerraoui and A. Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science*, 254(1-2):297–316, 2001.

[32] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In Sape J. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 1993.

[33] J. Holliday, D. Agrawal, and A. E. Abbadi. The performance of database replication with group multicast. In *Proceedings of FTCS'99*, pages 158–165. IEEE Computer Society, 1999.

[34] Udo Fritzke Jr. and Philippe Ingels. Transactions on partially replicated data based on reliable and atomic multicasts. In *Proceedings of ICDCS'01*, pages 284–291. IEEE Computer Society, 2001.

[35] R. Kotla, A. Clement, E. L. Wong, L. Alvisi, and M. Dahlin. Zyzzyva: speculative byzantine fault tolerance. *Communications of the ACM*, 51(11):86–95, 2008.

[36] A. D. Kshemkalyani and M. Singhal. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing*, 11(2):91–111, 1998.

[37] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[38] L. Lamport. Lower bounds for asynchronous consensus. *Distributed Computing*, 19(2):104–125, 2006.

[39] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *Proceedings of SIGMOD '05*, pages 419–430, New York, NY, USA, 2005. ACM.

[40] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[41] J.-P. Martin and L. Alvisi. Fast byzantine consensus. In *DSN'05*, pages 402–411, 2005.

[42] F. Mattern and S. Fünfrocken. A non-blocking lightweight implementation of causal order message delivery. In *Selected Papers from the International Workshop on Theory and Practice in Distributed Systems*, pages 197–213, London, UK, 1995. Springer-Verlag.

[43] M. G. Merideth and M. K. Reiter. Probabilistic opaque quorum systems. In *DISC'07*, pages 403–419, 2007.

[44] A. Mostefaoui and M. Raynal. Causal multicasts in overlapping groups: Towards a low cost approach. In *Proceedings of the 4th IEEE International Conference on Future Trends in Distributed Computing Systems*, pages 136–142, 1993.

[45] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Transactions on Computer Systems*, 23(4):375–423, 2005.

[46] F. Pedone and S. Frølund. Pronto: High availability for standard off-the-shelf databases. *Journal of Parallel and Distributed Computing*, 68(2):150–164, 2008.

[47] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Journal of Distributed and Parallel Databases and Technology*, 14(1):71–98, 2003.

[48] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing*, 15(2):97–107, 2002.

[49] F. Pedone, A. Schiper, P. Urban, and D. Cavin. Solving agreement problems with weak ordering oracles. In *Proceedings of EDCC'02*, pages 44–61. Springer, 2002.

[50] M. Rabin. Randomized Byzantine generals. In *Proceedings of FOCS'83*, pages 403–409, 1983.

[51] M. Raynal, A. Schiper, and S. Toueg. The causal ordering abstraction and a simple way to implement it. *Information Processing Letters*, 39:343–350, 1991.

[52] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *Proceedings of IC3N'98*, pages 840–847. IEEE, 1998.

[53] L. Rodrigues and P. Verissimo. Causal separators for large-scale multicast communication. In *Proceedings of ICDCS '95*, page 83, Washington, DC, USA, 1995. IEEE Computer Society.

[54] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[55] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, 1996.

[56] N. Schiper and F. Pedone. On the inherent cost of atomic broadcast and multicast in wide area networks. In *Proceedings of ICDCN'08*, pages 147–157. Springer, 2008.

[57] N. Schiper and F. Pedone. Solving atomic multicast when groups crash. In *Proceedings of OPODIS'08*, pages 481–495, 2008.

[58] N. Schiper and S. Toueg. A robust and lightweight stable leader election service for dynamic systems. In *Proceedings of DSN'08*, pages 207–216. IEEE Computer Society, 2008.

[59] D. Serrano, M. Patiño Martínez, R. Jiménez-Peris, and B. Kemme. Boosting database replication scalability through partial replication and 1-copy-snapshot-isolation. In *Proceedings of PRDC '07*, pages 290–297, Washington, DC, USA, 2007. IEEE Computer Society.

[60] A. Sousa, F. Pedone, R. Oliveira, and F. Moura. Partial replication in the database state machine. In *Proceedings of NCA'01*, pages 298–309. IEEE Computer Society, 2001.

[61] A. Sousa, J. Pereira, F. Moura, and R. Oliveira. Optimistic total order in wide area networks. In *Proceedings of SRDS'02*, pages 190–199. IEEE Computer Society, 2002.

[62] I. Stanoi, D. Agrawal, and A. E. Abbadi. Using broadcast primitives in replicated databases. In *Proceedings of ICDCS'98*, pages 148–155. IEEE Computer Society, 1998.

[63] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *Proceedings of SOSP'07*, pages 59–72, 2007.

[64] P. Vicente and L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. In *Proceedings of SRDS'02*, page 92. IEEE Computer Society, 2002.