Totally Ordered Broadcast and Multicast Algorithms: A Comprehensive Survey

Xavier Défago*
Graduate School of Knowledge Science,
Japan Advanced Institute of Science and Technology,
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan.
email: defago@jaist.ac.jp

André Schiper and Péter Urbán
Communication Systems Department,
Swiss Federal Institute of Technology in Lausanne,
CH-1015 Lausanne EPFL, Switzerland.
email: andre.schiper@epfl.ch, peter.urban@epfl.ch

Abstract

Total order multicast algorithms constitute an important class of problems in distributed systems, especially in the context of fault-tolerance. In short, the problem of total order multicast consists in sending messages to a set of processes, in such a way that all messages are delivered by all correct destinations in the same order. However, the huge amount of literature on the subject and the plethora of solutions proposed so far make it difficult for practitioners to select a solution adapted to their specific problem. As a result, naive solutions are often used while better solutions are ignored.

This paper proposes a classification of total order multicast algorithms based on the ordering mechanism of the algorithms, and describes a set of common characteristics (e.g., assumptions, properties) with which to evaluate them. In this classification, more than fifty total order broadcast and multicast algorithms are surveyed. The presentation includes asynchronous algorithms as well as algorithms based on the more restrictive synchronous model. Fault-tolerance issues are also considered as the paper studies the properties and behavior of the different algorithms with respect to failures.

1 Introduction

Distributed systems and applications are notoriously difficult to build. This is mostly due to the unavoidable concurrency of such systems, combined with the difficulty of providing a global control. This difficulty is greatly reduced by relying on communication primitives that provide higher guarantees than standard point-to-

*Most of this work was done as part of Xavier Défago's Ph.D. research at the Swiss Federal Institute of Technology in Lausanne [45].

point communication. One such primitive is called total order multicast.[1] Informally, total order multicast is a primitive that sends messages to a set of destinations and enforces a total order on the delivery of those messages (i.e., all messages are delivered in the same order). Total order multicast is an important primitive as it plays, for instance, a central role when implementing the state machine approach (also called active replication) [114, 113, 104]). It has also other applications such as clock synchronization [111], computer supported cooperative writing [7], distributed shared memory, or distributed locking [78]. More recently, it has also been shown that an adequate use of total order multicast can significantly improve the performance of replicated databases [2, 100, 75].

Literature on total order multicast There exists a considerable amount of literature on total order multicast, and many algorithms have been proposed (more than fifty!), following various approaches. It is however difficult to compare them as they often differ with respect to their actual properties, assumptions, objectives, or other important aspects. It is hence difficult to know which solution is best suited to a given application context. When confronted to a new problem, the absence of a roadmap to the problem of total order multicast has often led engineers and researchers to either develop new algorithms rather than adapt existing solutions (thus reinventing the wheel), or use a solution poorly suited to the application needs. An important step to improve the present situation is to provide a classification of existing algorithms.

Related work Previous attempts have been made at classifying and comparing total order multicast algorithms [9, 11, 38, 55, 86]. However, none is based on a comprehensive survey of existing algorithms, and hence they all lack generality.

The most complete comparison so far is due to [11] (an extended version was later published in French by [9]) who take an interesting approach based on the *properties* of the algorithms. The paper raises some fundamental questions upon which our work draws some of its inspiration. It is however a little outdated now, and slightly lacks generality. Indeed the authors only study seven different algorithms among which, for instance, none is based on a communication history approach (see Sect. 5.1).

[38] take a different approach focusing on the implementation of those algorithms rather than their properties. They study four different algorithms, and compare them using discrete event simulation. They find interesting results regarding the respective performance of different implementation strategies. Nevertheless, they fail to discuss the respective properties of the different algorithms. Besides, as they compare only four algorithms, this work is less general than Anceaume's.

[55] study the impact that packing messages has on the performance of algorithms. To this purpose, they study six algorithms among which those studied by [38]. They measure the actual performance of those algorithms and confirm the observations made by [38]. They show that packing message indeed provides an effective way to increase the performance of algorithms. The comparison also lacks generality, but this is quite understandable as this is not the main concern of that paper.

[86] defines a framework in which total order multicast algorithms can be compared from a performance point of view. The definition of such a framework is an important step towards an extensive and meaningful

[1] Total order multicast is a generalization of the well-known problem of *Atomic Broadcast*.[2] However, since the term *atomic broadcast* has also been employed abusively to designate a broadcast primitive without ordering property [51, 69, 57], we prefer to use the less ambiguous *total order*.

[2] Usually, a *multicast* puts no restriction on the set of destination processes, while a *broadcast* requires that all messages are sent to all processes. There are however situations in which this distinction is ambiguous.

comparison of algorithms. However, the paper does not go so far as to actually compare the numerous existing algorithms.

Classification A classification system is based on similarities. Entities are grouped into classes which are defined according to some criteria. Ideally, a perfect classification system would define its classes so that (1) entities with similar characteristics are grouped together, (2) every entity belongs to a class, (3) no entity belongs to two different classes unless one is a subclass of the other, and (4) entities are evenly distributed among classes. But, most importantly, the purpose of any classification is to bring a better understanding on how the various entities relate to each other. This is achieved by carefully choosing the criteria on which the classes are based. Some choices must sometimes be arbitrary, and hence rely on a subjective perception of the field. Nevertheless, any classification system is good and serves its purpose if it can bring this understanding.

Contribution In this paper, we propose a classification system based on the mechanisms of the algorithms, as well as a set of characteristics and assumptions. Based on this classification, we present a vast survey of published algorithms that solve the problem of total order multicast. We separate the presentation between algorithms which order messages using physical time and those which do not use physical time. We however restrict our study to algorithms that neither require specific hardware (e.g., unlike [21, 68, 89]) nor a specific network infrastructure or topology (e.g., unlike [32, 27, 54, 119]).

This work comes in two parts. In the first part, we present the problem of total order multicast and separate the issues concerning the ordering of messages and the questions of managing failures. In the second part, we present the numerous algorithms that appear in the literature according to the properties considered in the first part of the paper. Our classification system is essentially based on the way messages are ordered. We define five classes into which algorithms are grouped: *communication history*, *privilege-based*, *moving sequencer*, *fixed sequencer*, and *destinations agreement*.

Structure The rest of the paper is structured as follows. Section 2 introduces important concepts used in the context of this paper. In Section 3, we discuss the definition of Total Order Multicast. We first point out that there actually exist a hierarchy of specifications rather than a single one. We also mention the problem of contamination. Section 4 presents the architectural aspects of the algorithms, such as the possibility to multicast to multiple overlapping groups. In Section 5, we define five classes of total order multicast algorithms, according to the way messages are ordered. Section 6 gives a vast survey of total order broadcast and multicast algorithms found in the literature. The algorithms are grouped along their respective classes. For each algorithm, we discuss their principal characteristics. In Section 7, we talk about various other issues that are relevant to total order multicast. Finally, Section 8 concludes the paper.

2 System Models and Definitions

2.1 Basic System Models

Distributed systems are modeled as a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that interact by exchanging messages through communication channels. There exist a quantity of models that more or less restrict the behavior of these systems components. When one describes a system model, the important characteristics to consider are the synchrony and the failure mode.

2.1.1 Synchrony

The synchrony of a model is related to the timing assumptions that are made on the behavior of processes and communication channels. More specifically, one usually considers two major parameters. The first parameter is the *relative speed of processes*, which is given by the speed ratio of the slowest process with respect to the fastest process in the system. The second parameter is the *communication delay*, which is given by the time elapsed between the emission and the reception of messages. The synchrony of the system is defined by considering various bounds on these two parameters. For each parameter one usually consider the following level of synchrony.

1. There is a known upper bound which always holds.
2. There is an unknown upper bound which always holds.
3. There is a known upper bound which eventually holds forever.
4. There is an unknown upper bound which eventually holds forever.[3]
5. There is no bound on the value of the parameter.

A system in which both parameters are assumed to satisfy (1) is called a *synchronous system*. At the other extreme, a system in which relative process speed and communication delays are unbounded, i.e., (5), is called an *asynchronous system*. Between those two extremes lie the definition of various partially synchronous system models [47, 50].

2.1.2 Failure Mode

When one considers the problem of failures in a system it is important to characterize these failures. The failure mode of a system specifies the kinds of failures that are expected to occur in that system, as well as the conditions under which these failures may or may not occur. The general classes of failures are the following.

- *Crash failures.* When a process crashes, it ceases functioning forever. This means that it stops performing any activity including sending, transmitting, or receiving any message.
- *Omission failures.* When a process fails by omission, it omits performing some actions such as sending or receiving a message.
- *Timing failures.* A timing failure occurs when a process violates one of the synchrony assumptions. This type of failure is irrelevant in asynchronous systems.
- *Byzantine failures.* Byzantine failures are the most general type of failures. A Byzantine component is allowed any arbitrary behavior. For instance, a faulty process may change the content of messages, duplicate messages, send unsolicited messages, or even maliciously try to break down the whole system.

  In practice, one often considers a particular case of Byzantine failures, called *authenticated* Byzantine failures. Authenticated Byzantine failures allow Byzantine processes to behave arbitrarily. However, it

[3] There exist many other possible assumptions, such as: *There is a known upper bound that holds infinitely often for periods of a known duration.*

is assumed that processes have access to some authentication mechanism (e.g., digital signatures), thus making it possible to detect the forgery of valid messages by Byzantine processes. When mentioning Byzantine failures in the sequel (mostly in Sect. 6), we implicitly refer to *authenticated* Byzantine failures.

By contrast, a *correct* process is a process that never expresses any of the faulty behaviors mentioned above. Note that correct/incorrect are predicates over the whole execution: a process that crashes at time t is incorrect already before time t.

Similarly, communication channels can also be subject to crash, omission, timing, and Byzantine failures. For instance, a crashed channel is one that drops every message. Communication failures can for instance cause network partitions. Nevertheless, since communication failures are hardly considered in this paper, we do not describe them in further detail.

Peculiarities of Timing Failures A system is characterized by its "amount of synchrony", and by its failure modes. While the failure mode is normally orthogonal to the synchrony of the system, this is not the case with timing failures which are directly related to the synchrony of the system. Indeed, timing failures are characterized by a violation of the synchrony of the system.

2.2 Oracles

Depending on the synchrony of the system, some distributed problems cannot be solved. Yet, these problems become solvable if the system is extended with an oracle. In short, an oracle is a distributed component that processes can query, and which gives some information that the algorithm can use to guide its choices. In distributed algorithms, at least three different types of oracles are used: (physical) clocks, failure detectors, and coin flips.[4] Since the information provided by these oracles is sufficient to solve problems that are otherwise unsolvable, such oracles augment the power of the system model, and must hence be considered as a part of it.

2.2.1 Physical Clocks

A clock oracle gives information about physical time. Each process has access to its local physical clock and clocks are assumed to give a value that increases monotonically.

The values returned by clocks can also be constrained by further assumptions, such as synchronized clocks. Two clocks are ϵ-synchronized if, at any time, the difference between the values returned by the two clocks is never greater than ϵ. Two clocks are perfectly synchronized if $\epsilon = 0$. Conversely, clocks are not synchronized if there is no bound on the difference of their respective values.

Depending on the assumptions, the information returned by the clocks can or cannot be related to real-time. Synchronized clocks are not necessary synchronized with real-time. However, if all local clocks are synchronized with real-time, then they are of course synchronized with each other.

Note that, with the advent of GPS-based systems, assuming clocks that are perfectly synchronized with real-time is not unrealistic, even in large-scale systems. Indeed, [121] achieve clock synchronization in the order of a few microseconds, whereas software-based clock synchronization can at best achieve this at a precision several orders of magnitude lower.

[4] Suggested by Bernadette Charron-Bost.

2.2.2 Failure Detectors

The notion of failure detectors has been formalized by [23]. Briefly, a failure detector can be seen as a set of distributed modules, one module FD_i attached to each process p_i. Any process p_i can query its failure detector FD_i about the status of other processes (e.g., crashed or not crashed). The information returned by a failure detector FD_i can be incorrect (e.g., a non crashed process can be suspected), and inconsistent (e.g., FD_i can suspect process p_k at time t while, also at time t, FD_j does not suspect p_k). A failure detector is abstractly characterized by a property of *completeness* and by a property of *accuracy*. There exist various classes of failure detectors that are defined according to variants of these two basic properties.

Let us illustrate the properties of failure detectors with an example. The class of failure detectors $\Diamond\mathcal{S}$ is used in some total order broadcast and multicast algorithms, and it is defined by the following properties [23].

(STRONG COMPLETENESS) Eventually every faulty process is permanently suspected by all correct processes.

(EVENTUAL WEAK ACCURACY) There is a time after which some correct process is never suspected by any correct process.

An important aspect of $\Diamond\mathcal{S}$ is that it is the weakest failure detector to solve the problem of consensus (and hence total order multicast) in asynchronous systems with crash failures [22].[5] We do not describe the other classes of failure detectors, as giving a full description of all failure detectors that are presented by [23] is well beyond the scope of this paper. Their work is considerably more general than the sole detection of failures, and it can even be argued that this work can actually be extended to encompass the notion of oracle as a whole ([22] give the first hint in this direction).

2.2.3 Coin Flip

Another approach to extend the power of a system model consists in introducing the ability to generate random values. This is used by a class of algorithms called randomized algorithms. Those algorithms can solve problems such as consensus with a probability that asymptotically tends to 1 (e.g., [31]). It is however important to note that solving a problem for sure and solving a problem with probability 1 are not equivalent. We however do not take this issue further here, as we know of no total order multicast algorithm that is explicitly based on this approach.

2.3 Agreement Problems

Agreement problems constitute a fundamental class of problems in the context of distributed systems. There exist many different agreement problems that share a common pattern: processes have to reach some common decision, the nature of which depends on the problem. In this paper, we mostly consider four important agreement problems: *Consensus*, *Byzantine Agreement*, *Reliable Broadcast*, and *Atomic Broadcast*.

[5] The weakest failure detector to solve consensus is actually $\Diamond\mathcal{W}$ that differs from $\Diamond\mathcal{S}$ by satisfying a property of weak instead of strong completeness. However, [23] prove the equivalence of $\Diamond\mathcal{S}$ and $\Diamond\mathcal{W}$.

2.3.1 Consensus

Informally, the problem of Consensus is defined as follows.[6] Every process p_i begins by proposing a value v_i. Then, all processes must eventually agree on the same decision value v, which must be one of the proposed values.

2.3.2 Byzantine Agreement

The problem of Byzantine Agreement is also commonly known as the "Byzantine generals problem" [80]. In this problem, every process has an *a priori* knowledge that a particular process s is supposed to broadcast a single message m. Informally, the problem requires that all correct processes deliver the same message, which must be m if the sender s is correct.

As the name indicates, Byzantine Agreement has mostly been studied in relation with Byzantine failures. A variant of Byzantine Agreement, called *Terminating Reliable Broadcast* is sometimes studied in a context limited to crash failures.

2.3.3 Reliable Broadcast

As the name indicates, Reliable Broadcast is defined as a broadcast primitive. In short, Reliable Broadcast is a broadcast that guarantees that each broadcasted message m is delivered by all correct processes if the process $sender(m)$ is correct. If $sender(m)$ is not correct, then m must be delivered either by all correct processes or by none of them.

2.3.4 Atomic Broadcast

The problem of Atomic Broadcast is also itself an agreement problem. In short, this problem is defined as a Reliable Broadcast problem that must also ensure that all delivered messages are delivered by all processes in the same order.

2.3.5 Important Theoretical Results

There are at least four fundamental theoretical results that are directly relevant to the problem of Atomic Broadcast and Consensus. First, Atomic Broadcast and Consensus are equivalent problems [47, 23], that is, if there exists an algorithm that solves one problem, then it can be transformed to solve the other problem. Second, there is no (deterministic) solution to the problem of Consensus in asynchronous systems if just a single process can crash [53]. Nevertheless, Consensus can be solved in asynchronous systems extended with failure detectors [23], and in some partially synchronous systems [47, 50]. Finally, the weakest failure detector to solve Consensus in an asynchronous system is $\Diamond\mathcal{W}$ [22].

2.4 A Note on Asynchronous Total Order Multicast Algorithms

In many papers written about Total Order Multicast, the authors claim that their algorithm solves the problem of Total Order Multicast in asynchronous systems with process failures. This claim is of course incorrect, or

[6] Note that there exist other specifications of the Consensus problem in the literature. However, a more detailed discussion on this issue is irrelevant here.

incomplete at best. Indeed, Total Order Multicast is not solvable in the asynchronous system model if even one process can crash, as proved by Fischer, Lynch, and Paterson [53]. Or put differently, it is not possible, in asynchronous systems with process failures, to guarantee both safety and liveness.

From a formal point-of-view, most practical systems are asynchronous because it is not possible to assume that there is an upper bound on communication delays. In spite of this, why do many practitioners still claim that their algorithm can solve Consensus related problems in real systems?

To answer this question, it is first of all important to understand that real systems usually exhibit some level synchrony, and are thus not exactly asynchronous. However, practitioners often rely on the rarely explicit assumption that "most messages are likely to reach their destination within a known delay δ" [42, 40]. Whether the bound δ is known or not does not actually matter from a theoretical point of view. Indeed, the existence of such a *finite* bound means that the probabilistic behavior of the network is *known* and *stable*, thus increasing the strength of the model. In the sequel, we call such a system model an *ad-hoc asynchronous system model*. Hence, so-called asynchronous algorithms require that the system meets the (often implicit) assumptions of the ad-hoc asynchronous model.

An ad-hoc asynchronous model can be related to a synchronous model with timing failures. Indeed, assuming that messages will meet a deadline $T + \delta$ with a given probability $P[T + \delta]$ is equivalent to assuming that messages will miss the deadline $T + \delta$ (i.e., a timing failure) with a known probability $1 - P[T + \delta]$. This does not put a bound on the occurrence of timing failures, but puts a probabilistic restriction on the occurrence of such failures. At the same time, a purely asynchronous system model can also be seen as a synchronous system model with an unrestricted number of timing failures.

2.5 Process Controlled Crash

Process controlled crash is the ability given to processes to kill other processes or to commit suicide. In other words, this is the ability to artificially force the crash of a process. Allowing process controlled crash in a system model increases its power. Indeed, this makes it possible to transform severe failures (e.g., omission, Byzantine) into less severe failures (e.g., crash), and to emulate an "almost perfect" failure detector. However, this power does not come without a price, and process controlled crash should be avoided whenever possible.

Automatic transformation of failures [98] present a technique that uses process controlled crash to transform severe failures (e.g., omission, Byzantine) into less severe ones (i.e., crash failures). In short, the technique is based on the idea that processes have their behavior monitored. Then, whenever a process begins to behave incorrectly (e.g., omission, Byzantine), it is killed.[7]

However, this technique cannot be used in systems with lossy channels, or subject to partitions. Indeed, in such contexts, processes might be killing each other until not a single one is left alive in the system.

Emulation of an almost perfect failure detector In practical systems, perfect failure detectors ($\mathcal{P}$) are extremely difficult to implement because of the difficulty to distinguish crashed processes from very slow ones. Chandra and Toueg [23] show that $\mathcal{P}$ can be implemented with timeouts in a synchronous system. However, practical systems are rarely completely synchronous.

[7] The actual technique is more complicated than that, but this gives the basic idea.

A perfect failure detector ($\mathcal{P}$) satisfies both strong completeness and strong accuracy. Even in asynchronous systems, it is easy to implement a failure detector that satisfies weak completeness (e.g., [115, 26]), and hence strong completeness [23] (see Sect. 2.2). However, the difficulty is to satisfy strong accuracy: no process is suspected before it crashes [23]. Process controlled crash makes it possible to emulate the slightly weaker accuracy property that follows:

(QUASI-STRONG ACCURACY) No correct process is ever suspected by any correct process.

Given a failure detector $\mathcal{X}$ that satisfies strong completeness and any form of accuracy, it is possible to use process controlled crash to emulate a category that we call $\mathcal{AP}$ (almost perfect). The principle is simple: whenever $\mathcal{X}$ suspects a process p, then p is killed (forced to crash). As a result, $\mathcal{AP}$ never makes a mistake *a posteriori*, and thus satisfies the above "quasi-strong accuracy" property.

2.5.1 Cost of a Free Lunch

Process controlled crash is often used in practice to solve agreement problems such as Atomic Broadcast and Consensus. It is extremely powerful, but this has a cost. Indeed, process controlled crash reduces the effective fault-tolerance of the system.

To understand this price, it is first necessary to distinguish between two types of crash failures: genuine and provoked failures. *Genuine failures* are failures that naturally occur in the system, without any intervention from a process. Conversely, *provoked failures* are failures that are deliberately caused by some process (murder or suicide), that is, process controlled crash.

A fault-tolerant algorithm can only tolerate the crash of a bounded number of processes.[8] In a system with process controlled crash, this limit not only includes genuine failures, but also provoked failures. This means that each provoked failure actually *decreases* the number of genuine failures that can be tolerated. In other words, it reduces the actual fault-tolerance of the system.

It is important to stress two points. First, in systems with process controlled crash, each occurrence of a suspicion (if it leads to a provoked failure) reduces the number of genuine failures that the system can tolerate. Second, except in very tightly coupled systems, the number of suspicions is generally unpredictable, because message delays are themselves highly unpredictable (e.g., due to unexpected high load). As a result, requiring an upper bound on the number of suspicions makes the system less robust as it may be unable to survive long periods of communication instability.

2.6 Group Membership Service

A group membership service is a distributed service that is responsible for managing groups of processes. It often used as a basic building block for implementing complex group communication systems (e.g., Isis [19], Totem [94], Transis [49, 4, 83], Phoenix [85, 84]), as it allows to keep track of the membership of each process group.

In short, a group membership service is based on the notion of *view*. The view of a group is a list of processes that belong to that group. The membership service must maintain an agreement about the composition of groups. Thus, it must report changes in this composition to all members, by issuing *view change* notifications.

[8] The recovery of processes and the dynamic join of new processes are discussed in Section 2.6.

In a group membership service, processes are managed according to three basic operations. Processes can *join* or *leave* a group during the execution of the system, and processes that are suspected to have crashed are automatically *excluded* from the group. The composition of process groups can thus change dynamically, and processes are informed of these changes when they receive the view change notifications.

2.6.1 Join / Leave

A process p can dynamically *join* a process group G by issuing some "join request" message to the membership service. This triggers a view change protocol between the members of the group G, and eventually leads them to accepting a new view that includes p. The protocol normally requires that p's state be synchronized with the state of the members of G. This is done through a *state transfer*.[9]

When a process p needs to *leave* a process group, it issues a "leave request" message to the membership service. This triggers a view change protocol, with the result that p does not belong to the new view.

2.6.2 Exclusion

When the crash of a process has been detected, this process is removed (or excluded) from any groups it belongs to, thus preventing the system from being clogged by useless crashed processes. Unlike with join and leave operations, the initiative of the view change is not taken by the affected process. If a process p is suspected to have crashed by some other process q, then q can initiate a view change leading to the acceptance of a new view from which p is excluded.

However, if p is incorrectly suspected and gets excluded anyway, then p is supposed to commit suicide. What happens in practice is that process p tries to join back the group with a new identity. This means that it will obtain a new state from one of the members of the group, and hence "forget" what happened before it was excluded. This corresponds to process controlled crash, with all its limitations.

2.7 Notation

Table 1 presents the notation that is used throughout the paper. Note that the set Π is closely related to the system model. The set Π can be finite or infinite, it can be static or dynamic (varies over time), etc. These issues are interesting, but in order to keep the presentation simple, we have deliberately decided to ignore those considerations here.

3 Specification of Total Order Multicast

When talking about a problem, the first thing to discuss is its specification. Indeed, the specification of a problem *defines* that problem and hence must be considered before anything else. More specifically, the specification of a problem should be considered before considering any algorithm that solves that problem. This is particularly important given that two algorithms that meet even slightly different specifications actually solve different problems.

According to this policy, we first present the specification of the problem of total order multicast. We then situate that problem with respect to the hierarchy of multicast problems presented by [66].

[9] A state transfer is an operation in which a process transfers a copy of its state to another process. A detailed discussion on state transfer is however beyond the scope of this paper.

Table 1: Notation.

$\mathcal{M}$	set of all messages sent in the system.
Π	set of all processes in the system.
$sender(m)$	sender of message m.
$Dest(m)$	set of destination processes for message m.
Π_{sender}	set of all sending processes in the system. $\Pi_{sender} \stackrel{\text{def}}{=} \bigcup_{m \in \mathcal{M}} sender(m)$
Π_{dest}	set of all destination processes in the system. $\Pi_{dest} \stackrel{\text{def}}{=} \bigcup_{m \in \mathcal{M}} Dest(m)$

3.1 Total Order Multicast

The problem of Total Order Multicast is defined by four properties: validity, agreement, integrity, and total order.

(VALIDITY) If a correct process multicasts a message m to $Dest(m)$, then some correct process in $Dest(m)$ eventually delivers m.

(UNIFORM AGREEMENT) If a process delivers a message m, then all correct processes in $Dest(m)$ eventually deliver m.

(UNIFORM INTEGRITY) For any message m, every process p delivers m at most once, and only if (1) m was previously multicast by $sender(m)$, and (2) p is a process in $Dest(m)$.

(UNIFORM TOTAL ORDER) If processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

The well-known problem of Atomic Broadcast[10] (e.g., [66, 67]) appears as a special case of Total Order Multicast, where $Dest(m)$ is equal to Π for all messages. This distinction is discussed further in in Section 4.3.

3.2 Hierarchy of Non-Uniform Problems

The properties of agreement, integrity, and total order that are given above are uniform. This means that these properties not only apply to correct processes but also to faulty ones. Uniform properties are required by some classes of application such as atomic commitment. However, for some other applications uniformity is not necessary. Since enforcing uniformity in an algorithm often has a cost in terms of performance, it is also important to consider weaker problems specified using the following three non-uniform counterparts of the properties mentioned above.

(AGREEMENT) If a ***correct*** process delivers a message m, then all correct processes in $Dest(m)$ eventually deliver m.

(INTEGRITY) For any message m, every ***correct*** process p delivers m at most once, and only if (1) m was previously multicast by $sender(m)$, and (2) p is a process in $Dest(m)$.

[10] Note that the term *Atomic Broadcast* has been used abusively to designate a broadcast primitive without ordering property [51, 69, 57]

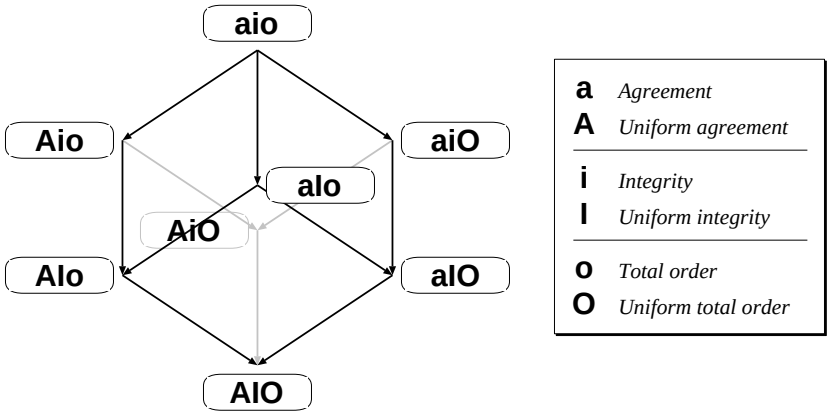


Figure 1: Hierarchy of fault-tolerant specifications for total order multicast.

(TOTAL ORDER) If two ***correct*** processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

The combinations of uniform and non-uniform properties define eight different specifications to the problem of fault-tolerant total order multicast. Those definitions form a hierarchy of problems. [124] propose a hierarchy based on two properties (total order and agreement), thus resulting in four different problems. In this paper, we complement this hierarchy by considering the property of uniform versus non-uniform integrity. We illustrate in Figure 1 the hierarchy given by the eight combinations of agreement, integrity, and total order.

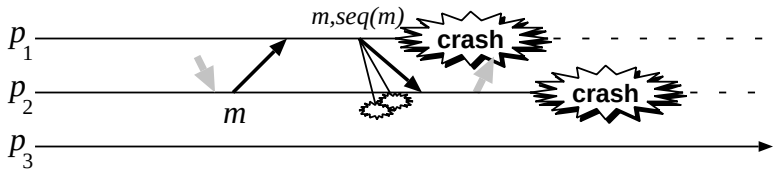


Figure 2: Violation of Uniform Agreement (example)

Figure 2 illustrates a violation of the Uniform Agreement property with a simple example. In this example, process p_2 sends a message m to a sequencer p_1, which gives a sequence number to m and then relays m to all processes. However, p_1 crashes while it is relaying m. The crash occurs in such a way that only p_2 receives m, but not p_3 and p_4. Upon reception of m and the sequence number, p_2 delivers m and then also crashes. As a result, the system is in a situation where the message m has been delivered by some process (p_2), but no correct process (e.g., p_3) will ever be able to deliver it. Indeed, m is lost to the system because every process that had a knowledge of m (p_1 and p_2) has crashed.

Uniform Agreement and message stabilization In algorithms that ensure Uniform Agreement, a process p is not allowed to deliver a message m before that message is stable. A message m is *stable* once it is certain that m will eventually be delivered by all correct processes.

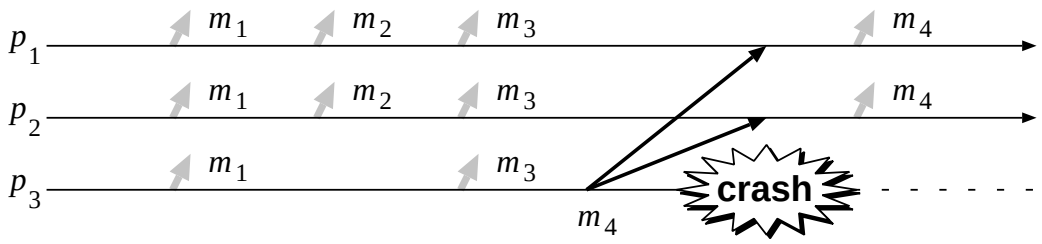


Figure 3: Contamination of correct processes (p_1, p_2) by a message (m_4) based on an inconsistent state (p_3 delivered m_3 but not m_2).

Uniform properties and Byzantine failures It is important to note that algorithms tolerant to Byzantine failures can guarantee none of the uniform properties. This is understandable as no behavior can be enforced on Byzantine processes. In other words, nothing can prevent a Byzantine process from (1) delivering a message more than once (violates integrity), (2) delivering a message that is not delivered by other processes (violates agreement), or (3) delivering two messages out of order (violates total order).

A note on Integrity Except in a system with Byzantine processes, Uniform Integrity is easily ensured.[11] For this reason, almost every algorithm found in the literature satisfies Uniform Integrity rather than its weaker counterpart. Hence, in the sequel, we do not discuss Integrity.

3.3 Problem of Contamination

The specification of total order multicast can be either uniform or not, depending whether it specifies the behavior of faulty processes or not. However, even with the strongest specification (uniform agreement, uniform integrity, and uniform total order) a faulty process is not prevented from getting an inconsistent state. This becomes a serious problem when one considers that this process can multicast a message based on this inconsistent state, and hence contaminate correct processes [61, 11, 9, 67].

Illustration Figure 3 illustrates an example [25, 67, 66] where an incorrect process contaminates the correct processes by sending a message based on an inconsistent state. Process p_3 delivers messages m_1 and m_3, but not m_2. It then has an inconsistent state when it multicasts m_4 to the other processes and then crashes. The correct processes p_1 and p_2 deliver m_4, thus getting contaminated by the inconsistent state p_3 had before it crashed (p_3 delivered m_3 but not m_2). It is important to stress again that the situation depicted in Figure 3 satisfies even the strongest specification.

Specification To exclude contamination, it is necessary to extend the specification of total order multicast. A specification can prevent contamination in two ways. One way is to forbid faulty processes from sending messages if their state is inconsistent. This is however difficult to formalize as a property. Hence the second (and stronger) solution is usually preferred, which consists in preventing any process from delivering a message that may lead to an inconsistent state [11, 74, 38].

[11] Ensuring Uniform Integrity requires (1) to uniquely identify messages, and (2) to keep track of delivered messages.

(PREFIX ORDER) For any two processes p and q, either $hist(p)$ is a prefix of $hist(q)$ or $hist(q)$ is a prefix of $hist(p)$, where $hist(p)$ and $hist(q)$ are the histories of messages delivered by p and q respectively.

Algorithms Among the numerous algorithms studied in the paper, a large majority of them ignore the problem of contamination in their specification. In spite of this, some of these algorithms avoid contamination. The mechanism of these algorithms either (1) prevents all processes from reaching an inconsistent state, or (2) prevents processes with an inconsistent state from sending messages to other processes.

Contamination and Byzantine failures Contamination is impossible to avoid in the context of arbitrary failures, because a faulty process may be inconsistent even if it delivers all messages correctly. It may then contaminate the other processes by multicasting a bogus message that seems correct to every other process [67].

4 Architectural Properties

When a total order multicast algorithm is used in some application context, the architectural properties of this algorithm are very important. For instance, there is a general distinction between *broadcast* algorithms (messages are received and delivered by all processes in the system) and *multicast* algorithms (messages are delivered only by a subset of the processes in the system). In this section, we define the following architectural properties of algorithms: single *vs* multiple destination groups, and closed *vs* open groups.

4.1 Single *vs* Multiple Destination Groups

Most algorithms present in the literature assume that there is only one destination group in the system. Since the ability to multicast messages to multiple (overlapping) groups is required by some applications, we consider it as an important architectural characteristic of an algorithm.

Single group ordering Messages are sent to only one group of destination processes and all messages that are sent to this group are delivered in the same order. This class groups a vast majority of the algorithms that are studied in this paper. Single group ordering can be defined by the following property:[12]

$$\forall m \in \mathcal{M} \ \ (Dest(m) = \Pi_{dest})$$

Even though these algorithms can be adapted fairly easily to systems with multiple *non-overlapping* groups, they often cannot be made to cope decently with groups that overlap.

Multiple groups ordering Multiple groups ordering occurs when multiple destination groups, that may overlap, exist in the system. An algorithm designed for such a system must guarantee that messages are delivered in a total order by all destination processes, including those that are at the intersection of two groups. For instance, if two messages m_A and m_B are sent to two overlapping groups A and B respectively, then m_A and m_B must be delivered in the same relative order by all processes that belong to the intersection $A \cap B$.

[12] Notation: see Table 1, p. 11.

Trivially, any algorithm that solves the problem of total order multicast in the context of a single group can solve it for multiple groups. Indeed, one can form a super-group with the union of all groups in the system. Whenever a message is multicast to a group, it is instead multicast to the super-group, and processes not in $Dest(m)$ simply discard it. To avoid such a trivial solution, we require multiple groups ordering algorithms to satisfy the following minimality property.

(WEAK MINIMALITY) The execution of the algorithm implementing the total order multicast of a message m to a destination set $Dest(m)$ involves only $sender(m)$, the processes in $Dest(m)$, and possibly an external process p_s (where $p_s \neq sender(m)$ and $p_s \notin Dest(m)$).

Genuine Multicast A genuine multicast is defined by [65] as a multiple groups ordering algorithm which satisfies the following minimality property.

(STRONG MINIMALITY) The execution of the algorithm implementing the total order multicast of a message m to a destination set $Dest(m)$ involves only $sender(m)$, and the processes in $Dest(m)$.

4.2 Closed *vs* Open Groups

In the literature, many total order multicast algorithms are designed with the implicit assumption that messages are sent *within* a group of processes. This originally comes from the fact that early work on this topic was done in the context of highly available storage systems [35]. However, a large part of distributed applications are now developed by considering more open interaction models, such as the client-server model, N-tier architectures, or publish/subscribe. For this reason, we consider that it is important for a process to be able multicast messages to a group it does not belong to. Consequently, it is an important characteristic of algorithms to be easily adaptable to accommodate open interaction models.

Closed Group Closed group algorithms require sending processes to be part of the destination processes.

$$\forall m \in \mathcal{M} \ (sender(m) \in Dest(m))$$

As a result, these algorithms do not allow external processes (processes that are not member of the group) to multicast messages to the destination group. This limitation forbids the use of such algorithms in the context of large-scale systems in which client processes are short-lived and their number is unbounded (e.g., on-line reservation systems, Internet banking).

Open Group Conversely, open group algorithms allow any arbitrary process in the system to multicast messages to a group, whether that process belongs to the destination group or not. Open group algorithms are more general than their closed group counterparts because the former can be used in a closed group architecture while the opposite is not true.

Remarks Most algorithms considered in this paper are designed for closed groups. However, in the sequel, we classify into the open group category all algorithms that can *trivially* be adapted to allow external processes to multicast to the group. By trivially adapted, we mean that such an adaptation requires (1) no additional

communication step, (2) no additional information to manage (e.g., longer vector clocks), and (3) no additional computation in the algorithm.

For instance, condition (1) rules out the following transformation from a closed group to an open group algorithm. An external sending process sends its message m to a member p of the group, which then broadcasts m within the group. This solution has the drawback that the client must monitor p, and resend m if p has crashed. This also means that m can be received multiple times by the members of the group, and thus duplicates must be detected and discarded.

Another possibility would be to include the sending process in the group and require it to discard the messages that it receives. This approach would be against condition (2) and (3).

4.3 Other Architectural Properties

4.3.1 Broadcast *vs* Multicast

A *broadcast* primitive is defined as one that sends messages to all processes in a system (i.e., single closed group).

$$\text{Broadcast: } \forall m \in \mathcal{M} \ \ (Dest(m) = \Pi)$$

In contrast, a *multicast* primitive sends messages to any subset of the processes in the system (i.e., open multiple groups). However, the distinction between broadcast and multicast is not precise enough, because it mixes two orthogonal aspects; single *vs* multiple destination groups, and closed *vs* open group.

4.3.2 Source Ordering

Some papers (e.g., [59, 70]) make a distinction between single source and multiple sources ordering. These papers define single source ordering algorithms as algorithms that ensure total order only if a *single* process multicasts every messages. This problem is actually a simplification of FIFO multicast and is easily solved using sequence numbers. In this paper, we are interested in algorithms that ensure total order and not only FIFO order. Hence we do not discuss this issue further, and all algorithms presented here provide multiple sources ordering:

$$|\Pi_{sender}| \geq 1$$

5 Ordering Mechanisms

In this section, we propose a classification of total order multicast algorithms in the absence of failures. The first question that we ask is: *"who builds the order?"* More specifically, we are interested in the entity which generates the information that is necessary for defining the order of messages (e.g., timestamp or sequence number).

We classify the processes into three distinct categories according to the role they take in the algorithm: sender process, destination process, or sequencer process. A *sender process*, or *sender*, is a process p_s from which a message originates ($p_s \in \Pi_{sender}$). A *destination process*, or *destination*, is a process p_d to which a message is destined ($p_d \in \Pi_{dest}$). Finally, an *sequencer process* is some process p_x that is not necessarily a

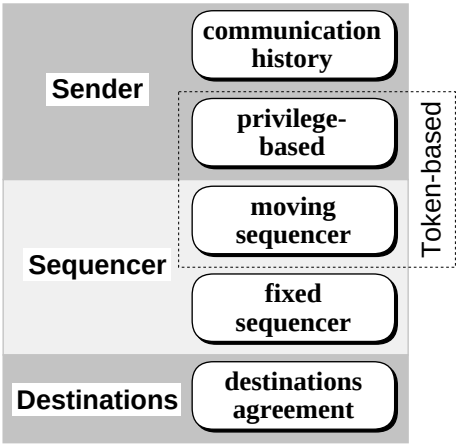


Figure 4: Classes of total order multicast algorithms.

sender nor a destination, but is nevertheless involved in the algorithm ($p_x \in \Pi$). A single process may combine these different roles (e.g., sender *and* sequencer *and* destination). However, we represent these roles separately as they are distinct from a *conceptual* viewpoint.

Following from the three different roles that processes can take, total order multicast algorithms can fall into three basic classes, depending whether the order is built by sender, sequencer, or destination processes respectively. Among those three basic classes, there exist differences in the algorithms. This allows us to define subclasses as illustrated in Figure 4. This results in the five classes illustrated on the figure: *communication history*, *privilege-based*, *moving sequencer*, *fixed sequencer*, and *destinations agreement*. Algorithms that are either privilege-based or based on a moving sequencer are commonly referred to as token-based algorithms.

The terminology used by this classification is partly borrowed from other authors. For instance, "communication history" and "fixed sequencer" was proposed by [41]. The term "privilege-based" was suggested by Malkhi. Finally, [81] group algorithms into three classes based on where the order is built. However, the definition of the classes is specific to a client-server architecture.

In this section, we present the five classes of algorithms. We illustrate each class with a simple algorithm for which we give the pseudo-code. These algorithms are presented for the sole purpose of illustrating the corresponding category. Although they are inspired from existing algorithms, they are simplified. Also, as mentioned earlier, the problem of failures are ignored in this section, thus none of these algorithms tolerate failures.

5.1 Communication History

Communication history algorithms are based on the following principle. A partial order "$<$" is defined on messages, based on the history of communications. Communication history algorithms deliver messages in a total order that is compatible with the partial order "$<$". This class of algorithms takes its name from the fact that the order is induced by the communication. The partial order "$<$" is defined as follows:

$$send(m) \longrightarrow send(m') \Rightarrow m < m'$$

where "$\longrightarrow$" denotes Lamport's relation "happened before."[13]

```
1:  Senders and destinations (code of process p):
2:    Initialisation:
3:      received_p ← ∅                                   {Messages received by process p}
4:      delivered_p ← ∅                                  {Messages delivered by process p}
5:      LC_p[p_1 ... p_n] ← {0, ..., 0}                  { LC_p[q]: logical clock of process q as seen by process p}
6:    procedure TO-multicast(m)                          {To TO-multicast a message m}
7:      LC_p[p] ← LC_p[p] + 1
8:      ts(m) ← LC_p[p]
9:      send (m, ts(m)) to all
10:   end
11:   when receive (m, ts(m))
12:     LC_p[p] ← max(ts(m), LC_p[p]) + 1
13:     LC_p[sender(m)] ← ts(m)
14:     received_p ← received_p ∪ {m}
15:     deliverable ← ∅
16:     for each message m' in received_p \ delivered_p do
17:       if ts(m') < min_{q∈Π} LC_p[q] then
18:         deliverable ← deliverable ∪ {m'}
19:       end if
20:     deliver all messages in deliverable, according to the total order ⟹ (see Sect. 5.1)
21:     delivered_p ← delivered_p ∪ deliverable
22:   end when
```

Figure 5: Simple communication history algorithm.

Figure 5 illustrates a typical algorithm of this class, inspired by [78] and based on his logical clocks.[14] In short, Lamport's logical clocks are defined according to the relation "$\longrightarrow$" (happened before), in such a way that the following condition holds: Let $ts(m)$ be the logical timestamp of message m. Lamport's clock ensure that

$$send(m) \longrightarrow send(m') \Rightarrow ts(m) < ts(m')$$

So, in order to deliver the messages in a total order compatible with "$<$", it is sufficient to deliver the messages in a total order compatible with $ts(m)$ as follows: if $ts(m) < ts(m')$ then message m has to be delivered before message m'. Messages with the same logical timestamp are delivered according to an arbitrary order based on their sender, and denoted by $\prec$. The total order relation on events "$\Longrightarrow$" is then defined as follows: if m and m' are two messages, then $m \Longrightarrow m'$ if and only if either (1) $ts(m) < ts(m')$ or (2) $ts(m) = ts(m')$ and $sender(m) \prec sender(m')$.

Consider Figure 5. When a process wants to multicast a message m, it sends m to all processes with a Lamport timestamp. Upon reception of such a timestamped message m, processes store m as a received yet undelivered message. A process can deliver message m only after it knows that no other process can multicast a new message with a timestamp lower or equal to $ts(m)$. Messages are then delivered according to the total

[13]The happened before is defined as follows [78]. Let e_i and e_j be three events in a distributed system. The transitive relation $e_i \longrightarrow e_j$ holds if it satisfies the following three conditions: (1) e_i and e_j are two events in the same process, then e_i comes before e_j; (2) e_i is the sending of a message m by one process and e_j is the receipt of m by another process; or (3) There exists a third event e_k such that, $e_i \longrightarrow e_k$ and $e_k \longrightarrow e_j$ (transitivity).

[14]The algorithm in Figure 5 can also be seen as a simplification of the Newtop algorithm [52].

order relation "$\Longrightarrow$".

Observation 1
The algorithm of Figure 5 is not live. Indeed, considerer a scenario where a single process p broadcasts a single message m, and no other process ever broadcast any message. According to the algorithm in Figure 5, a process q can deliver m only after it has received, from every process, a message that was broadcast after *the reception of m. This is of course impossible if some of the processes never broadcast any message. To overcome this problem, communication history algorithms proposed in the literature usually force processes to send messages.*

Observation 2
In synchronous systems, communication history algorithms use physical timestamps and rely on synchronized clocks. The nature of such systems makes it unnecessary to force processes to send messages in order to guarantee the liveness of the algorithm. This can be seen as an example of the use of time to communicate [79].

5.2 Privilege-Based

Privilege-based algorithms rely on the idea that senders can multicast messages only when they are granted the privilege to do so. The arbitration between senders makes it possible to totally order the messages as they are sent. Building the total order requires to solve the problem of FIFO multicast (easily solved with sequence numbers at the sender), and to ensure that passing the privilege to the next sender does not violate this order.

Figure 6 illustrates this class of algorithms. The order is defined by the senders when they multicast their messages. The privilege to multicast (and order) messages is granted to only one process at a time, but this privilege circulates from process to process among the senders.

The algorithm of Figure 7 illustrates the principle of privilege-based algorithms. Messages receive a sequence number when they are multicast, and the privilege to multicast is granted by circulating a token message among the senders.

Senders circulate a token message that carries a sequence number for the next message to multicast. When a process wants to multicast a message m, it must first wait until it receives the token message. Then, it assigns a sequence number to each of its messages and sends them to all destinations. The sender then updates the token and sends it to the next sender. Destination processes deliver messages in increasing sequence numbers.

Observation 3
In privilege-based algorithms, senders usually need to know each other in order to circulate the privilege. This constraint makes privilege-based algorithms poorly suited to open groups, unless there is a fixed and previously known set of senders.

Observation 4
In synchronous systems, privilege-based algorithms are based on the idea that each sender process is allowed to send messages only during some predetermined time slots. These time slots are attributed to each process in such a way that no two process can send messages at the same time. By ensuring that the communication medium is accessed in mutual exclusion, the total order is easily guaranteed.

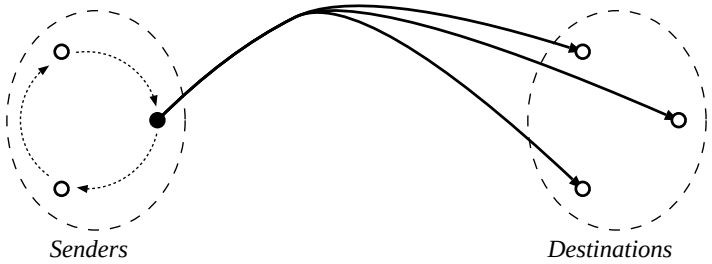


Figure 6: privilege-based algorithms.

```
1:  Senders (code of process s_i):
2:     Initialisation:
3:        tosend_{s_i} ← ∅
4:        if s_i = s_1 then
5:           send token(seqnum : 1) to s_1
6:        end if
7:     procedure TO-multicast(m)                    {To TO-multicast a message m}
8:        tosend_{s_i} ← tosend_{s_i} ∪ {m}
9:     end
10:    when receive token
11:       for each m' in tosend_{s_i} do
12:          send (m', token.seqnum) to destinations
13:          token.seqnum ← token.seqnum + 1
14:       tosend_{s_i} ← ∅
15:       send token to s_{i+1}
16:    end when

17: Destinations (code of process p_i):
18:    Initialisation:
19:       nextdeliver_{p_i} ← 1
20:       pending_{p_i} ← ∅
21:    when receive (m, seqnum)
22:       pending_{p_i} ← pending_{p_i} ∪ {(m, seqnum)}
23:       while ∃(m', seqnum') ∈ pending_{p_i} s.t. seqnum' = nextdeliver_{p_i} do
24:          deliver(m')
25:          nextdeliver_{p_i} ← nextdeliver_{p_i} + 1
26:       end while
27:    end when
```

Figure 7: Simple privilege-based algorithm.

5.3 Moving Sequencer

Moving sequencer algorithms are based on the idea that a group of processes successively act as sequencer. The responsibility of sequencing messages is passed among these processes. In Figure 8, the sequencer is chosen among several processes. It is however important to understand that, with moving sequencer algorithms, the roles of sequencer and destination processes are normally combined. Unlike privilege-based algorithms, the sequencer is related to the destinations rather than the senders.

The algorithm of Figure 9 illustrates the principle of moving sequencer algorithms. Messages receive a sequence number from a sequencer. The messages are then delivered by the destinations in increasing sequence numbers.

To multicast a message m, a sender sends m to the sequencers. Sequencers circulate a token message that carries a sequence number and a list of all messages for which a sequence number has been attributed (i.e., sequenced messages). Upon reception of the token, a sequencer assigns a sequence number to all received yet unsequenced messages. It sends the newly sequenced messages to the destinations, updates the token, and passes it to the next sequencer.

Observation 5
The major motivation for passing the token is to distribute the load of sequencing the messages among several processes.

Observation 6
It is tempting to consider that privilege-based and moving sequencer algorithms are equivalent. Indeed, both techniques rely on some token passing mechanism and hence use similar mechanisms. However, the two techniques differ in one important aspect: the total order is built by senders in privilege-based algorithms, and by sequencers in moving sequencer algorithms. This has at least two major consequences. First, moving sequencer algorithms are easily adapted to open groups. Second, in privilege-based algorithms the passing of token is necessary to ensure the liveness of the algorithm. In contrast, the passing of token in moving sequencer algorithms is to ensure load balancing (and sometimes collect acknowledgements).

5.4 Fixed Sequencer

In a fixed sequencer algorithm, one process is elected as the sequencer and is responsible for ordering messages. Unlike moving sequencer algorithms, there is only one such process, and the responsibility is not normally transfered to another processes (at least in the absence of failure). On Figure 10, the sequencer is depicted by the solid black circle.

The algorithm in Figure 11 illustrates the approach. One specific process takes the role of a sequencer and builds the total order. This algorithm does not tolerate a failure of the sequencer.

To multicast a message m, a sender sends m to the sequencer. Upon receiving m, the sequencer assigns it a sequence number and relays m with its sequence number to the destinations. The destinations then deliver messages in sequence thus according to some total order.

Observation 7
There exists a second variant to fixed sequencer algorithms. Unlike the algorithm of Figure 11, the sender sends its message m to the sequencer as well as the destinations. Upon receiving m, the

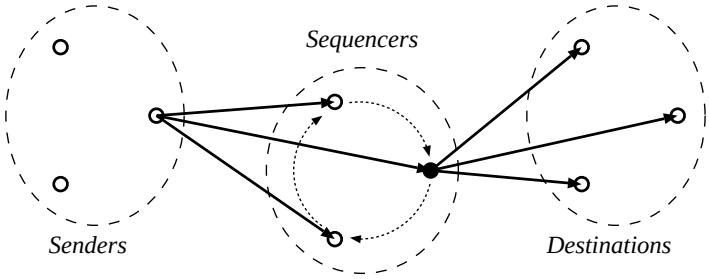


Figure 8: Moving sequencer algorithms.

```
1:  Sender:
2:     procedure TO-multicast(m)                          {To TO-multicast a message m}
3:        send (m) to all sequencers
4:     end

5:  Sequencers (code of process s_i):
6:     Initialisation:
7:        received_{s_i} ← ∅
8:        if s_i = s_1 then
9:           send token(sequenced : ∅, seqnum : 1) to s_1
10:       end if
11:    when receive m
12:       received_{s_i} ← received_{s_i} ∪ {m}
13:    end when
14:    when receive token from s_{i-1}
15:       for each m' in received_{s_i} \ token.sequenced do
16:          send (m', token.seqnum) to destinations
17:          token.seqnum ← token.seqnum + 1
18:          token.sequenced ← token.sequenced ∪ {m'}
19:       send token to s_{i+1}
20:    end when

21: Destinations (code of process p_i):
22:    Initialisation:
23:       nextdeliver_{p_i} ← 1
24:       pending_{p_i} ← ∅
25:    when receive (m, seqnum)
26:       pending_{p_i} ← pending_{p_i} ∪ {(m, seqnum)}
27:       while ∃(m', seqnum') ∈ pending_{p_i} s.t. seqnum' = nextdeliver_{p_i} do
28:          deliver (m')
29:          nextdeliver_{p_i} ← nextdeliver_{p_i} + 1
30:       end while
31:    end when
```

Figure 9: Simple moving sequencer algorithm.

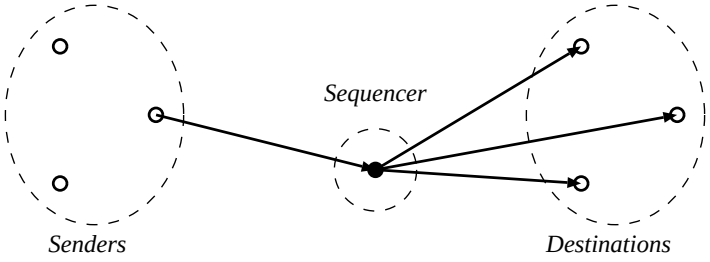


Figure 10: Fixed sequencer algorithms.

```
1:  Sender:
2:    procedure TO-multicast(m)                          {To TO-multicast a message m}
3:      send (m) to sequencer
4:    end

5:  Sequencer:
6:    Initialisation:
7:      seqnum ← 1
8:    when receive (m)
9:      sn(m) ← seqnum
10:     send (m, sn(m)) to all
11:     seqnum ← seqnum + 1
12:   end when

13: Destinations (code of process p_i):
14:   Initialisation:
15:     nextdeliver_{p_i} ← 1
16:     pending_{p_i} ← ∅
17:   when receive (m, seqnum)
18:     pending_{p_i} ← pending_{p_i} ∪ {(m, seqnum)}
19:     while ∃(m', seqnum') ∈ pending_{p_i} : seqnum' = nextdeliver_{p_i} do
20:       deliver (m')
21:       nextdeliver_{p_i} ← nextdeliver_{p_i} + 1
22:     end while
23:   end when
```

Figure 11: Simple fixed sequencer algorithm (code of process p).

sequencer sends a sequence number for m to all destinations. The destinations can then deliver messages in sequence after they have received both the message and its sequence number. This solution contributes to reducing the sequencer bottleneck, because the sequencer then only needs to send sequence numbers and not the whole content of messages.

5.5 Destinations Agreement

In destinations agreement algorithms, the order results from an agreement between destination processes. The destinations receive messages without any ordering information, and exchange information in order to define an order. Messages are then delivered according to this order. This approach is illustrated on Figure 12 and Figure 13.

The following algorithm is largely derived from Skeen's algorithm, as described by [17]. To multicast a message m, a sender sends m to all destinations. Upon receiving m, a destination assigns it a local timestamp and multicasts this timestamp to all destinations. Once a destination process has received a local timestamp for m from all other destinations, a unique global timestamp $sn(m)$ is assigned to m as the maximum of all local timestamps. A message m can only be delivered once it has received its global timestamp $sn(m)$ and there is no other undelivered message m' can receive a smaller global timestamp $sn(m')$.

5.6 Discussion: Time-Free *vs* Time-Based Ordering

We introduce a further distinction between algorithms, orthogonal to the ordering class. An important aspect of total order multicast algorithms is the use of physical time as a base for the ordering mechanisms. For instance, in Section 5.1 (see Fig. 5) we have presented a simple communication-history algorithm based on the use of *logical* time. It is indeed possible to design a similar algorithm based on the use of *physical* time and synchronized clocks instead.

In short, we discriminate between algorithms with time-free ordering and those with time-based ordering. Algorithms with *time-free ordering* are those which use an ordering mechanism that relies neither on physical time nor on the synchrony of the system for the ordering of messages. Conversely, algorithms with *time-based ordering* are those which do rely on physical time.

6 Survey of Existing Algorithms

We now consider the Total Order Multicast algorithms that are described in the literature. For each of the five classes of algorithms, we first enumerate the algorithms that belong to that class. Then, we explain the respective differences and similarities of the algorithms with respect to various issues. The description is structured as follows:

1. *Algorithms*. We give the list of algorithms that belong to the class, with the corresponding references.
2. *Ordering mechanism*. We discuss the way each algorithm builds the order, sometimes referring to the corresponding sample algorithm of Section 5.
3. *System model*. We present the assumptions made by the algorithms with respect to the system model, failure mode, etc (see Sect. 2).

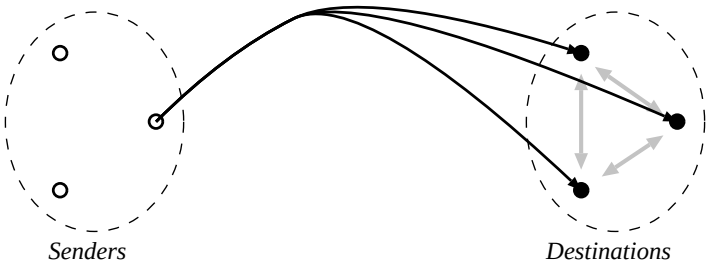


Figure 12: Destinations agreement algorithms.

1: *Sender:*
2: **procedure** *TO-multicast*(m) *{To TO-multicast a message m}*
3: send (m) to destinations
4: **end**

5: *Destinations (code of process p_i):*
6: Initialisation:
7: $stamped_{p_i} \leftarrow \emptyset$
8: $received_{p_i} \leftarrow \emptyset$
9: $LC_{p_i} \leftarrow 0$ *{LC_{p_i}: logical clock of process p_i}*
10: **when** receive m
11: $ts_i(m) \leftarrow LC_{p_i}$
12: $received_{p_i} \leftarrow received_{p_i} \cup \{(m, ts_i(m))\}$
13: send $(m, ts_i(m))$ to destinations
14: $LC_{p_i} \leftarrow LC_{p_i} + 1$
15: **end when**
16: **when** received $(m, ts_j(m))$ from p_j
17: $LC_{p_i} \leftarrow \max(ts_j, LC_{p_i} + 1)$
18: **if** received $(m, ts(m))$ from all destinations **then**
19: $sn(m) \leftarrow \max_{i=1 \cdots n} ts_i(m)$
20: $stamped_{p_i} \leftarrow stamped_{p_i} \cup \{(m, sn(m))\}$
21: $received_{p_i} \leftarrow received_{p_i} \setminus \{m\}$
22: $deliverable \leftarrow \emptyset$
23: **for each** m' in $stamped_{p_i}$ such that $\forall m'' \in received_{p_i} : sn(m') < ts_i(m'')$ **do**
24: $deliverable \leftarrow deliverable \cup \{(m', sn(m'))\}$
25: deliver all messages in $deliverable$ in increasing order of $sn(m)$
26: $stamped_{p_i} \leftarrow stamped_{p_i} \setminus deliverable$
27: **end if**
28: **end when**

Figure 13: Simple destinations agreement algorithm.

4. *Specification*. We discuss the exact properties satisfied by each algorithm. This includes a discussion on uniformity and contamination (see Sect. 3).

5. *Architectural properties*. This part discusses the adaptability of each algorithm to structures with open groups, or the possibility of multicast to multiple groups.

6. *Failure management*. We present the mechanisms that the algorithms use to tolerate failures. This includes retransmission mechanisms, as well as group membership services.

Note that we begin the discussion of each class of algorithm with the ordering mechanism, thus avoiding to remain too abstract.

6.1 Communication History Algorithms

Algorithms The following algorithms belong to the communication history class of algorithms:

- Berman and Bharali [14].
- COReL [74, 73].
- HAS atomic broadcast [37, 39, 35].
- Lamport [78].
- Newtop (symmetric protocol) [52].
- Ng [99].
- Psync [103].
- Redundant broadcast channels [33].
- ToTo [48].
- Total [96, 88, 95, 87, 93, 92].

Ordering mechanism [78] uses his definition of logical clocks to define a total order on events. The principle of total order multicast is to deliver messages according to the total order defined on their respective *send* events (see Sect. 5.1). Lamport describes a mutual exclusion algorithm that is based on a total order of events. Based on this mutual exclusion algorithm it is straightforward to derive a total order multicast algorithm. Newtop [52] is based on Lamport's principle but extends Lamport's algorithm in many ways (e.g., fault-tolerance, multiple groups; see discussions below).

Total [96, 88, 95, 87, 93, 92] is a total order multicast algorithm that is built on top of a reliable broadcast algorithm called Trans (Trans in defined together with Total). Trans uses an acknowledgement mechanism that defines a partial order on messages. Total builds a total order that is an extension of this partial order.

ToTo [48] is an "agreed multicast[15]" algorithm developed on top of the Transis partitionable group communication system [49]. ToTo extends the order of an underlying causal broadcast algorithm. It is based on

[15] Agreed Multicast [48, 4] is a problem based on Total Order Multicast, with a weaker definition to account for network partitions. Informally, an agreed multicast satisfies the properties of Total Order Multicast, as long as there is no partition. However, if partitions occur, then Agreement and Total Order are guaranteed within each partition, but are not necessarily satisfied across two different partitions.

dynamically building a causality graph of received messages. A particularity of ToTo is that, to deliver a message m, a process must have received acknowledgements for m from as few as a majority of the processes (instead of all processes).

COReL [74] is a total order multicast algorithm that sits on top of Transis. It aims at building consistent replicated services in partitionable systems like Transis. For this, it relies on the weaker definition of an underlying "agreed multicast[15]" algorithm (e.g., ToTo), and the possibility to store messages in stable storage. COReL gradually builds a global order by tagging messages according to three different color levels (red, yellow, green). A message starts at the red level; when a process has no knowledge about its position in the global order. As more information becomes available, the message is promoted to a higher level (yellow and then green). When the message becomes green, it can be delivered because its position in the global order is then known.

Psync [103] is a total order multicast algorithm used in the context of two group communication systems: Consul [91], and Coyote [15]. In Psync, processes dynamically build a causality graph of messages as they receive new messages. Psync then delivers messages according to a total order that is an extension of the causal order.

[99] presents an communication history algorithm that uses a minimum-cost spanning tree to propagate messages. The ordering of messages is based on Lamport's clocks, similar to Lamport's algorithm. However, messages and acknowledgement are propagated, respectively gathered, using a minimum-cost spanning tree. The use of a spanning tree improves the scalability of the algorithm and makes it adequate for wide-area networks.

Communication history algorithms with time-based ordering use physical clocks instead of logical ones. Thus, the order of messages is defined by assigning a physical timestamp to *send* events. The algorithms rely on the assumption that (1) physical clocks are synchronized, and (2) there is a known upper bound on communication delays.

[37] propose a collection of total order multicast algorithms (called HAS) that assume a synchronous system model with ϵ-synchronized clocks. The authors describe three algorithms—HAS-$\mathcal{O}$, HAS-$\mathcal{T}$, and HAS-$\mathcal{B}$—that are respectively tolerant to omission failures, timing failures, and authenticated Byzantine failures. These algorithms are based on the principle of *information diffusion*, which is itself based on the notion of flooding or gossiping. In short, when a process wants to broadcast a message m, it timestamps it with the time of emission T according to its local clock, and sends it to all neighbors. Whenever a process receives m for the first time, it relays it to its own neighbors. Processes deliver message m at time $T + \Delta$, according to their timestamp (where Δ is the termination time; a known constant that depends on the topology of the network, the number of failures tolerated, and the maximum clock drift ϵ).

[33] presents an adaption of the HAS-$\mathcal{O}$ algorithm (omission failures) to the context of broadcast channels. The system model assumes the availability of $f + 1$ independent broadcast channels (or networks) that connect all processes together, thus creating $f + 1$ independent communication paths between any two processes (where f is the maximum number of failures). As a result of this architecture, the algorithm for broadcast channels can achieve a significant message reduction over HAS-$\mathcal{O}$.

[14] present a technique that allows to transform an early-stopping Byzantine Agreement algorithm into a total order multicast algorithm with similar early-stopping properties.

System model Lamport's algorithm [78] assumes a failure-free asynchronous system model.

ToTo [48] relies on the Transis group membership service [49, 83, 4]. Transis is a partitionable group

membership and thus is not based on process controlled crash. Instead, the system allows partitioned groups to diverge.

COReL [74] relies on the weaker guarantees offered by algorithms such as ToTo, in addition to the group membership. Unlike ToTo, COReL does not allow disconnected groups to diverge, but relies for correctness (agreement, termination) on the assumption that disconnected processes eventually reconnect.

Total [96, 88, 95, 87] assumes an ad hoc asynchronous system model with process crash and message loss. If the system is fully asynchronous, the algorithm is only partially correct (i.e., safe but not live). Moser and Melliar-Smith [93, 92] also describe an extension of Total to tolerate Byzantine failures.

Psync [103] assumes an ad hoc asynchronous system model. As for failures, Psync assumes that process may crash (permanent failures) and that messages can be lost (transient failures).

[99] assumes an ad hoc asynchronous system model with crash failures and network partitions. Crash failures can be either transient or permanent, but network partitions must be transient. Also, the algorithm needs synchronized clocks for fairness, but no synchronization is needed for correctness.

As already mentioned, the HAS algorithms [37] and the adaptation to redundant broadcast channels [33], assume a synchronous model with ϵ-synchronized clocks. The HAS algorithms consist of three algorithms tolerant to omission, timing, and (authenticated) Byzantine failures. The adaptation to redundant broadcast channels is tolerant only to omission failures.

[14] give three methods to transform Byzantine Agreement into total order multicast. These methods are respectively based on the round synchronous model, the synchronous model with ϵ-synchronized clocks, and the synchronous model with timing uncertainty as described by [12]. The resulting algorithms are tolerant to Byzantine failures.

Specification Most communication history algorithms are uniform, or can easily adapted to become uniform. In fact, a uniform communication history algorithm does not generate more messages than a non-uniform one. The difference is that a non-uniform algorithm can normally deliver messages earlier than a uniform one. Satisfying uniform properties is actually related to the information passed by messages, to the delivery condition, to the actual characteristics of the group membership (or reconfiguration protocol), and to the failure model (for instance, Byzantine algorithms cannot be uniform; see Sect. 3.2).

Communication history algorithms can only make progress when processes communicate often with each other. So, to ensure liveness, all of these algorithms require (sometimes implicitly) that processes send empty messages if they have not sent messages for a long period. In algorithms with time-based ordering, physical time can efficiently replace empty messages [79].

According to the proofs, Newtop [52] ensures only Agreement and Total Order. Although it is not proven, it seems that the algorithm actually ensures Uniform Agreement and Uniform Total Order. According to the proofs, [99] ensures both Uniform Agreement and Uniform Total Order.

Psync [103] is left unspecified. From the description of the algorithm, it is not hard to believe that it actually builds a total order in the absence of failures. However, the exact properties enforced by the algorithm in the presence of failures are difficult to guess.

As mentioned earlier in this section, ToTo [48] solves a problem called Agreed multicast that allows partitioned groups to diverge (and it is proven correct for this). Thus, the algorithm does not solve the problem of total order multicast if processes can be partitioned. Nevertheless, if we consider the algorithm in an environment where network partitions never occur, it probably satisfies Agreement and Total Order.

The problem solved by COReL [74] does not allow disconnected groups to diverge, but it must weaken the termination property: "[COReL] guarantees that if a majority of the processes form a connected component then these processes eventually deliver all messages sent by any of them, in the same order." It is not difficult to see that, assuming that all correct processes are always connected, COReL satisfies the problem of Total Order Multicast defined in Section 3.1. In that case, it is probably easy to adapt the proofs in order to show that COReL satisfies both Uniform Agreement and Uniform Total Order.

All time-free communication history algorithms extend the partial order defined by causality. As a consequence, every communication history algorithm delivers messages according to a causal total order.

[37] prove that the three HAS algorithms satisfy Agreement. The authors do not prove Total Order but, by the properties of synchronized clocks and the timestamps, Uniform Total Order is not too difficult to enforce. However, if the synchronous assumptions do not hold, the algorithms may probably violate Total Order (safety property) rather than just Termination (liveness property).

[14] show that the resulting Atomic Broadcast algorithm inherits the properties of the underlying Byzantine Agreement algorithm. However, since the authors are not concerned with uniformity, they only mention that the resulting Atomic Broadcast algorithm must satisfy Agreement and Total Order.

Architectural properties In communication history algorithms, the destination processes usually require information from all potential senders in order to determine when a message can be delivered. For instance, in Algorithm 5 the destinations can deliver a message m only after they know that they will only receive new messages with a larger timestamp. This implies that destinations have some knowledge of the senders, and thus makes the approach poorly suited to open groups.

[99] supports open group. Besides, the protocol is well designed for wide-scale application, as the use of a spanning tree makes the algorithm more message-savvy.

Aside from Newtop [52] none of the communication history algorithm presented in this paper are designed to support multiple groups. To support multiple groups, Newtop uses a mechanism that is in essence similar to Skeen's algorithm, as described by [17] (Skeen's algorithm and its variants are discussed in Section 6.5).

HAS [37] and its extension to broadcast channels [33], and [14] are not conceptually restricted to closed groups. However, the assumed upper bound on communication delays, the assumed network topology, and the synchronized clocks[16] put some practical restrictions on the openness of the architecture.

Failure management The solution proposed by [78] does not consider failures. In fact, the crash of a process blocks the algorithm forever. Lamport's solution to total order can easily be made fault-tolerant by relying on a group membership. Newtop [52] is a good example of this.

Trans/Total does not rely on a group membership, but rather on its own failure detection scheme. Trans is designed to run on a broadcast network and uses a combination of positive and negative acknowledgements piggybacked on actual messages.

ToTo relies on the Transis group membership service to tolerate failures. This algorithm needs the group membership service to ensure its liveness in the case of failures and partitions, but its safety does not rely on it.

COReL [74] tolerates process crashes and communication failures resulting in network partitions. The algorithm relies entirely on the underlying group membership (Transis) to handle failures. COReL requires that some protocol messages are logged in stable storage in order to support the recovery of crashed processes.

[16] With the advent of GPS-based clock synchronization systems [121], clock synchronization is not a strong constraint anymore.

Psync [103] does not rely on a group membership to tolerate failures, but the actual mechanism is not described in details. The algorithm uses a negative acknowledgement to retransmit lost messages (thus implementing reliable channels). Crashed process are never excluded from a group. However, it seems that the crash of a process may lead some correct processes to *discard* messages while others don't, thus violating Agreement!

In [99], the group is managed through the spanning tree. Although the algorithm uses a timeout mechanism to tolerate failures, it does not force processes to crash. The algorithm considers transient and permanent failures for processes. A permanent failure means that the process will never recover, and must thus be excluded from the tree. According to the authors, permanent failures are rare and distinguishing them from transient failures is important in the long-term. Hence, the remaining hosts may be notified manually when a transient failure must be considered as permanent.

The three variant of HAS [37] tolerate omission, timing, or Byzantine failures respectively. There is no exclusion in the algorithm, and the failures of processes and communication links are masked by space redundancy. The algorithms must rely on the assumption that the subgraph consisting of all correct processes and all correct communication links is connected. In other words, the connectivity of the network is redundant enough that there is always a correct path between any two correct processes.

The adaptation of HAS-$\mathcal{O}$ to redundant broadcast channels [33] tolerates communication failures and crash failures. The algorithm tolerates crash failures in the same way. However, communication failures are tolerated as follows: since the algorithm is based on the availability of $f + 1$ communication channels (where f is the maximum number of failures assumed), then there is at least one correct communication channel.

6.2 Privilege-Based Algorithms

Algorithms Six algorithms fall into the class of privilege-based algorithms, among which four rely on time-free ordering:

- Gopal and Toueg [60].
- MARS [77].
- On-demand [42, 3].
- TPM [105].
- Totem [6, 5].
- Train protocol [34].

Ordering mechanism In the absence of failures, almost all algorithms behave in a similar fashion to Algorithm 7 (p. 20). With the exception of On-demand, all algorithms rely on the existence of a logical ring along which a token is passed. However, there are slight differences between those algorithms.

Although based on token-passing along a logical ring, the Train protocol [34] is based on the idea that the token (called a train) carries the messages in addition to the privilege to broadcast. More specifically, when a sender receives the token, it adds the messages that it wants to broadcast at the end.[17]

[17] The Train protocol (i.e., messages carried by the token) comes clearly in contrast with the other algorithms of the same class (i.e., messages are directly broadcast to the destinations). Unlike the former, the latter can take advantage of the properties of a broadcast network.

Unlike the other privilege-based algorithms, On-demand [42] does not use a logical ring. Processes which want to broadcast a message must request the token by sending a message to the current token holder. As a consequence, the protocol is more efficient if senders send long bursts of messages and such bursts rarely overlap. Also, in contrast to the other algorithms, *all processes* must know the identity of the token holder at any time. So, the token transfer is a broadcast to all.

All the algorithms (except Train) restrict the token holding time, by putting a bound on the number of messages that a process can broadcast before passing on the token (in the case of time-based ordering, the amount of time that it can keep the token). On the one hand, this prevents starvation by a process too eager to send. On the other hand, this helps to avoid buffer overflows at the destinations, which otherwise would lead to expensive message retransmissions. In particular, TPM sets a fixed bound on the number of messages, whereas Totem and On-demand implement more sophisticated flow control strategies.

Finally, Gopal and Toueg's algorithm [60] and MARS [77] are two algorithms that rely on physical time and synchronized clocks to order messages.

Gopal and Toueg's algorithm is based on the round synchronous model.[18] During each round, one of the processes is designated as the *transmitter* (the only process allowed to broadcast new messages; i.e., other than acknowledgements). Messages are delivered once they are acknowledged, three rounds after their initial transmission.

The MARS algorithm takes a different approach. It is based on the principle of time-division multiple-access (TDMA). TDMA consists in attributing predefined time slots to each process. Processes are then allowed to broadcast messages during their time slots only. Unlike the round synchronous model, this restriction also applies to acknowledgements. This can be seen as an example of using time to communicate [79] (here, to pass the token).

System model The four algorithms with time-free ordering (TPM, Totem, Train, On-demand) assume an ad-hoc asynchronous system model with process crashes and message omission failures. Channels are lossy, but their exact semantics is never properly specified. All algorithms except Totem assume that at least a majority of the processes are always correct and never suspected (Totem requires the existence of only one such process). These algorithm are based on a group membership used to define the logical ring, and hence rely on process controlled crash to solve Total Order Multicast.

The two algorithms with time based ordering (Gopal and Toueg, and MARS) assume a synchronous model with synchronized clocks. Gopal and Toueg is based on the round synchronous model. The algorithm assumes that crash failures as well as omission failures can occur. MARS uses TDMA, which can actually be seen as some implementation of a round synchronous system. The MARS algorithm, as it is described by [77], is only tolerant to crash failures.

Specification With the notable exception of [60], none of the privilege-based algorithms studied here are formally specified. Nevertheless, it is possible to reason about those algorithms and infer some of their properties. In a failure-free environment, all time-free privilege-based algorithms mentioned above satisfy both Total Order and Agreement. It is however less clear what exactly happens to those properties if a failure occurs (especially if the token must be regenerated).

[18] The *round synchronous model* is a computation model in which the processes evolution of processes is synchronized according to rounds. During each round, every process performs the same actions: (1) send a message to all processes, (2) receive a message from all processes, and then (3) perform some computations.

In these algorithms, the sender determines the total order by assigning a unique sequence number to each message. This means that the information from which the total order is inferred is carried by the messages themselves, and is directly interpretable by the destinations. Uniform Total Order is easily ensured if the algorithm can rely on a perfect failure detector. As a second consequence, a message is always received with the same sequence number, hence Uniform Integrity only requires to keep track of the sequence number of the last message delivered.

According to their respective authors, the time-free algorithms are quite different with respect to Agreement. It is claimed that TPM and Train satisfy Uniform Agreement, while On-demand satisfies only Agreement. According to its authors, Totem leaves the choice to the user and provides both "agreed order" (non-uniform agreement) and "safe order" (uniform agreement).

No time-free privilege-based algorithms that satisfy Uniform Agreement allow contamination. Indeed, all processes deliver messages with consecutive sequence numbers, making it impossible to generate "holes" in the sequence of delivered messages.

Gopal and Toueg's algorithm [60] satisfies Uniform Total Order. However, it guarantees Uniform Agreement only if the transmitter is correct. The papers about MARS [77, 76] do not provide enough information about the atomic multicast algorithm and its properties.

Architectural properties Due to their approach, all privilege-based algorithms are restricted to a single closed group. Most of the algorithms require that all sender processes are also destinations. Besides, as these algorithms introduce arbitration on the side of the senders, all senders must know each other. However, [105] include some ideas how to extend TPM to support multiple closed groups.

According to their designers, the ring and the token passing scheme make privilege-based algorithm highly efficient in broadcast LANs, but less suited to interconnected LANs. To overcome this problem, [1] extend Totem to an environment consisting of multiple interconnected LANs. The resulting algorithm performs better in such an environment, but otherwise has the same properties as the original version (single ring).

Failure management All algorithms tolerate message loss by relying on message retransmission. The time-free algorithms (except On-demand) use the token to carry retransmission requests (i.e., negative acknowledgements). [105] also propose a variant of TPM in which retransmission requests are sent separately from the token, in order to improve the behavior in networks with a high rate of message loss. On-demand takes a different approach with positive acknowledgement for groups of messages that are sent directly by the destinations to the token holder.

Except for Gopal and Toueg's algorithm, all algorithms rely on a group membership service to exclude crashed (and suspected) processes. On top of that, the exclusion of suspected processes is unilaterally decided by only one of the processes. Except for On-demand, the detection of crashes is combined with the passing of the token.

Gopal and Toueg's algorithm considers differently the failure of the transmitter and that of other processes. First, the failure of a process different from the transmitter is simply ignored. If a process is unable to communicate with enough processes because of omissions, then it commits suicide (process controlled crash). Finally, according to [9], the algorithm stops if the transmitter fails. This claim is however incomplete since Gopal and Toueg's algorithm leaves the system in such a state that the algorithm can safely be restarted.

6.3 Moving Sequencer Algorithms

Algorithms The following algorithms are based on a moving sequencer.

- Chang and Maxemchuck [24].
- Pinwheel [42, 41].
- RMP [122].

To the best of our knowledge, there is no moving sequencer algorithm that relies on time-based ordering. It is questionable whether a time-based ordering would make sense in this context.

The three algorithms behave in a similar fashion. In fact, both Pinwheel and RMP are based on Chang and Maxemchuck's algorithm. They improve some aspects of Chang and Maxemchuck's algorithm, but in a different manner. Pinwheel is optimized for a uniform message arrival pattern, while RMP provides various levels of guarantees.

Ordering mechanism The three algorithms are based on the existence of a logical ring along which a token is passed. The process that holds the token, also known as the token site, is responsible for sequencing the messages that it receives. However, unlike privilege-based algorithms, the token site is not necessarily the sender of the message that it orders.

For practical reasons, all three algorithms require that the logical ring spans all destination processes. This requirement is however not necessary for ordering messages, and hence these algorithms still qualify as sequencer-based algorithms according to our classification.

System model All three algorithms assume an ad-hoc asynchronous system model with process crashes and message omission failures.

Pinwheel assumes that a majority of the processes remains correct and connected at all time (majority group). The algorithm is based on the timed asynchronous model [40]. It relies on physical clocks for timeouts, but it does not need to assume that these clocks are synchronized. The authors explicitly make the assumption that "most messages are *likely* to reach their destination within a known delay δ."

In order to deliver messages with strong requirements, RMP also assumes that a majority of the processes remain correct and connected at any time in order to satisfy the properties of Total Order Multicast.

Specification None of the three moving sequencer algorithms are formally specified. While it is easy to infer that all three algorithms probably satisfy Uniform Integrity, this is more difficult with the other two properties.

Chang and Maxemchuck's algorithm and Pinwheel seem to satisfy Uniform Total Order. In both algorithms, the uniformity is only possible through an adequate support from the group membership mechanism (or reformation phase).

Assuming an adequate definition of the "likeliness" of a message to reach its destination within a known delay, Chang and Maxemchuck seem to satisfy Uniform Agreement. Depending on the user's choice, RMP satisfies Agreement, Uniform Agreement, or neither properties. Pinwheel only satisfies Agreement, but Cristian, Mishra, and Alvarez argue that the algorithm could easily be modified to satisfy Uniform Agreement [42]. This would only require that destination processes deliver a message after they have detected that it is stable.

Neither Chang and Maxemchuck nor Pinwheel are subject to contamination. RMP however does not preclude the contamination of the group.

Both Chang and Maxemchuck and Pinwheel deliver messages in a total order that is an extension of causal order. However, this requires the sender to be part of the destinations (i.e., closed group). It is also due to the constraint that a sender can broadcast a message only after the previous one has been acknowledged. As for RMP, this depends on the semantics that is required for each message.

Architectural properties Both Chang and Maxemchuck's algorithm and Pinwheel assume that the sender process is part of the logical ring, and hence one of the destinations (i.e., closed group). They use this particularity for the acknowledgement of messages. An adaptation to open groups would probably require only little changes, but then causal delivery would not be guaranteed.

RMP differs from the other two algorithms for it is designed to operate with open groups. [122] also claim that "RMP provides multiple multicast groups, as opposed to a single broadcast group." According to their description, supporting multiple multicast groups is a characteristic associated with the group membership service. It is then dubious that "multiple groups" is used with the same meaning as defined in Section 4.1, that is, with total order also guaranteed for processes that are at the intersection of two groups.

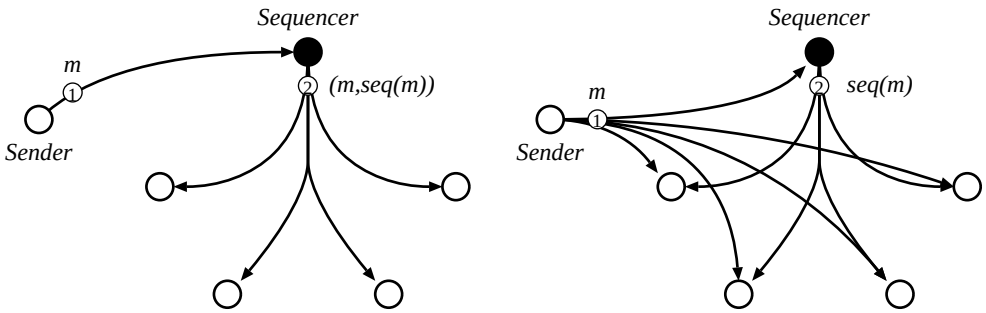
Failure management All three algorithms tolerate message loss by relying on a message retransmission protocol. The algorithms rely on a combination of negative and positive acknowledgements of messages. More precisely, when a process detects that it has missed a message, it issues a negative acknowledgement to the token site. On top of that, the token carries positive acknowledgements. The negative acknowledgement scheme is used for message retransmission, while the positive scheme is mostly used to detect the stability of messages.

6.4 Fixed Sequencer Algorithms

Algorithms The following algorithms are based on a fixed sequencer:

- ATOP [30].
- Amoeba (method 1, method 2) [72, 71].
- Garcia-Molina and Spauster [59, 58].
- Isis (sequencer) [18].
- Jia [70], later corrected by Chiu & Hsiao [29], and Shieh & Ho [116].
- MTP [54].
- Navaratnam, Chanson, and Neufeld [97].
- Newtop (asymmetric protocol) [52].
- Phoenix [124].
- Rampart [107, 106].
- Rodrigues, Fonseca, and Veríssimo [108].
- Tandem global update [20], as described by Cristian *et al* [38].

Ordering mechanism As briefly mentioned in Section 5.4, there are two basic variants to fixed sequencer algorithms. The first variant corresponds to Algorithm 11 (p. 23) and is illustrated in Figure 14(a). In this case, when a process p wants to broadcast a message m, it only sends m to the sequencer. In turn, the sequencer appends a sequence number to m and relays it to all destinations. In the second variant (Fig. 14(b)), the process p sends its message m to all destination processes as well as the sequencer. The sequencer then broadcasts a sequence number for m, which is then used by the destinations to deliver m in the proper order.



(a) Tandem-like variant (b) Isis-like variant

Figure 14: Common variants of fixed sequencer algorithms.

The first variant (Fig. 14(a)) is simple and generates very few messages. This approach is for instance taken by the Tandem global update protocol [20], the algorithm proposed by [97], and Amoeba (method 1) [72].

The second approach (Fig. 14(b)) generates more messages (in point-to-point networks), but it can reduce the load on the sequencer as well as make it easier to tolerate a crash of the sequencer. This second approach is taken by Isis (sequencer) [18], Amoeba (method 2) [72], Phoenix [124], and Rampart [107].

Newtop (asymmetric protocol) [52], [59], and Jia's algorithm [70] use more complicated techniques for ordering, as they cope with multiple overlapping groups. Here, the main difficulty is to ensure that two messages sent to different but overlapping groups are delivered in the same order on all the processes in the intersection of the two groups (see Sect. 4.1). When the system consists of a single destination group, the three algorithms are equivalent to the first variant illustrated in Figure 14(a).

In Newtop's asymmetric protocol, each group has an independent sequencer. All processes maintain logical clocks, and both the unicast from the sender to the sequencer and the multicast from the sequencer to the destination is timestamped. A process which belongs to multiple groups must delay the sending of a message (to the relevant sequencer) until it has received the sequence number for all previously sent messages (from the relevant sequencers).

A different approach is taken by Garcia-Molina and Spauster, as well as Jia. A propagation graph (a forest) is constructed first. Each group is assigned a starting node, and senders send their messages to these starting nodes. Messages travel along the edges of a *propagation graph*, and ordering decisions are resolved along the path. [70] creates simpler propagation graphs, using the notion of *meta-groups*; i.e., sets of processes having the same group membership (e.g., processes that belong simultaneously to a group G_1 and a group G_2, but to no other group, form a meta-group $G_{1\cap 2}$). If used in a single-group setting, these algorithms behave like

Algorithm 11 (i.e., the propagation graph constructed in this case is a tree of depth 1).

[108] present a special case, optimized for large networks. It uses a hybrid protocol for ordering messages: roughly speaking, on a local scale, ordering is decided by a fixed sequencer process, and on a global scale, the sequencer processes use a communication history based algorithm to decide on the order. More precisely, senders send their message to all destinations. Each sender has an associated sequencer process that issues a sequence number for the messages of the sender. Messages containing these sequence numbers are sent to all, and they are ordered using the communication history based algorithm. The authors also describe interesting heuristics to change a sequencer process to a non-sequencer process and vice versa. Reasons for such changes can be failures, membership changes or changes in the traffic pattern.

ATOP [30] is also an "agreed multicast[19]" algorithm developed for the Transis group communication system [49]. The originality of the algorithm is that it adapts to changes in the transmission rates, using some *adaptation policy* to influence the total order. The goal is to increase performance in systems where processes broadcast at very different rates.

System model All the algorithms assume an ad-hoc asynchronous system model, and rely on some form of process controlled crash: the killing of processes (e.g., Tandem), or the exclusion from a group membership (e.g., Isis, Rampart). Most of them only consider process crashes; Rampart is the only one of this class that tolerates Byzantine failures.

Tandem, Amoeba, and Garcia-Molina and Spauster's algorithm explicitly treat message omissions, whereas all the others rely on reliable channels. In addition, Isis (sequencer), Garcia-Molina and Spauster, Jia, Rodriguez et al. require FIFO channels. Rampart not only requires FIFO channels, but also that all messages are authenticated.

[97] consider a slightly more complicated failure model in which multiple processes may reside on the same machine, and hence fail in a dependent manner. However, this model is an easy extension of the usual model of independent process crashes.

Garcia-Molina and Spauster's [59] algorithm requires synchronized clocks. However, synchronized clocks are only necessary to yield bounds on the behavior of the algorithm when crash failures occur. Neither the ordering mechanism, nor the fault tolerance mechanism needs them.

Specification Garcia-Molina and Spauster, Phoenix and Rampart explicitly specify whether the Agreement and the Total Order properties guaranteed by the algorithm are uniform or not.

Garcia-Molina and Spauster, as well as the extension proposed by [70] guarantee Uniform Total Order and Uniform Agreement (this is called reliability type R3 by [59]).

Phoenix consists of three algorithms with different guarantees. The first algorithm (weak order) only guarantees Total Order and Agreement. The second algorithm (strong order) guarantees both Uniform Total Order and Uniform Agreement. Then, the third algorithm (hybrid order) combines both semantics on a per message basis.

Because it considers Byzantine processes in its system model, Rampart only satisfies Agreement and Total Order.

[19] See footnote (15) on page 26.

The other algorithms are not specified so precisely, but it is still possible to reason about their properties. Actually, all of them satisfy only Total Order (non-uniform) and Agreement (non-uniform), except for Navaratnam, Chanson, and Neufeld for which this is unclear.

Isis (sequencer) and the closed group version of Jia[20] [70, 29] ensure causal order delivery in addition to total order. Tandem, Garcia-Molina and Spauster, and Jia's original algorithm provide FIFO order. Note that the latter two provide FIFO order because they assume FIFO reliable channels and any multicast starts with a unicast to a fixed process.

Architectural properties As already mentioned when describing the ordering mechanisms, almost all of the algorithms are restricted to a single group, with the exception of Newtop (asymmetric protocol), Garcia-Molina and Spauster, and Jia.

Newtop (asymmetric protocol), Garcia-Molina and Spauster, and Jia's algorithm can manage multiple groups. In the two latter algorithm, the ordering of a message m may involve processes that are neither a destination nor the sender of m. This means that they only satisfy the Weak Minimality condition defined in Section 4.1.

We now discuss which algorithms have open groups or can be easily extended such that they work for open groups. Some of the fixed sequencer algorithms require that the sender unicasts to the sequencer (see Fig. 14(a), p. 35). This requires that senders know which process is the sequencer *at any time*, i.e., they have to follow changes in group membership. Only group members have this information, therefore the algorithms in question—Tandem; Navaratnam, Chanson, and Neufeld; Amoeba (method 1)—are closed group algorithms. Extending these algorithms to open groups requires that all possible clients are notified of every membership change. Furthermore, clients must be ready to retransmit their requests if the sequencer crashes. As a result, this considerably limits the extent of such an adaptation.

The three algorithms which can manage multiple groups—Newtop (asymmetric protocol), Garcia-Molina and Spauster, and Jia—are open in the sense that a process in any of the groups they manage can send to any of these groups, not only to their own group. There is however a limitation to this: processes that do not belong to some group cannot send messages to any of the groups. Also, Jia's algorithm satisfies causal order only if it is limited to operate with closed groups.

Most algorithms in which a sender starts by broadcasting to all members of the group (see Fig. 14(b), p. 35) can be adapted to open groups. Indeed, senders do not need to know the exact group membership if sending messages to excluded processes has no effect. Amoeba (method 2), Phoenix, and Rampart can be adapted in such a way. Also [108] can be adapted to open groups under certain circumstances.[21] As for Isis (sequencer), the algorithm can be adapted to open groups, thus allowing processes to broadcast messages to a group to which they do not belong. In this case, total order multicast no more satisfies causal order.

Failure management All algorithms assume reliable channels except for Tandem, Amoeba, and Garcia-Molina and Spauster, which explicitly consider message omissions. Tandem uses positive acknowledgements for each message that comes from the sender. Garcia-Molina and Spauster use negative acknowledgements

[20] The algorithm described by [70] violates causal order; [29] provide a correction that only works in the case of closed groups.

[21] The algorithm can be adapted provided that one of the sequencers assumes responsibility for ordering messages issued by senders that are outside the group.

combined with live messages, if actual multicast messages are rare.[22] This ensures that processes can detect message losses fast. Amoeba uses a combination of positive and negative acknowledgements.[23]

Almost all the algorithms use a group membership service to exclude faulty processes. Isis (sequencer), Phoenix, and [108] do not specify this service in detail. In the group membership used by both Amoeba, and [97], the exclusion of suspected processes is unilaterally decided by only one of the processes. In contrast, the membership service of Newtop requires that *all* processes that are not suspected suspect a process in order to exclude it from the group. Rampart requires that at least one third of all processes in the current view reach an agreement on the exclusion.[24] Considering that Rampart assumes that less than one third of the processes are Byzantine, this means that a single honest process (i.e., non-Byzantine) may unilaterally exclude another process from the group.

[59], and [70] behave in a peculiar manner. If a non-leaf process p fails, then its descendants do not receive any message until p recovers. Hence, these two algorithms tolerate failures only in a model where every crashed process eventually recovers. [70] also suggests a limited form of group membership where processes from a pool of spare processes replace failed nodes in the propagation graph (inspired from [99]).

6.5 Destinations Agreement Algorithms

Algorithms The following algorithms fall into the class of destinations agreement algorithms:

- ABP [90, 8].
- AMp [120] and xAMp [110].
- ATR [46].
- Chandra and Toueg [23].
- Fritzke, Ingels, Mostefaoui, and Raynal [56]
- Generic broadcast [102].
- Le Lann and Bres [81].
- Luan and Gligor [82].
- MTO [64], corrected by SCALATOM [109].
- Optimistic atomic broadcast [101].
- Prefix agreement [10].
- Skeen (Isis) [17].
- TOMP [43].

Ordering mechanism In destinations agreement algorithms, the order is built solely by the destination processes. The destinations receive messages without any ordering information, and they exchange information in order to agree on the order.[25]

[22] The sequencer sends null messages to communicate the highest sequence number to the recipients.

[23] The actual protocol is complicated because it is combined with flow control, and tries to minimize the communication cost.

[24] This condition is necessary to prevent Byzantine processes from purposely excluding correct processes from the group.

[25] The exchange of information may in fact involves the sender or some external process. However, these processes do not provide any ordering information, and merely arbitrate the exchange.

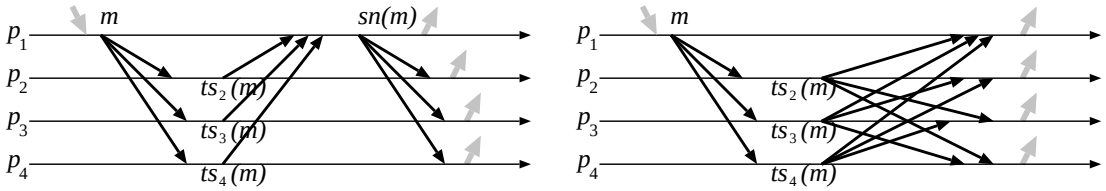
Destinations agreement algorithms are a little more difficult to relate to each other than algorithms of other classes. It is nevertheless possible to distinguish between two different families of algorithms. Algorithms of the first family use logical clocks similar to the algorithm depicted in Figure 13 (p. 25). In the second family, the algorithms are built around an agreement problem (e.g., Consensus, Atomic Commitment).

Algorithms of the first family are based on logical clocks, and generally operate according to the following steps:

1. To multicast a message m, a sender sends m to all destinations.[26] Upon receiving m, a destination uses its logical clock to timestamp the message.
2. Destination processes communicate the timestamp of the message to the system, and a global timestamp $sn(m)$ is computed that is then sent back to all destinations.
3. A message becomes deliverable once it has a global timestamp $sn(m)$. Deliverable messages are then delivered in the order defined by their global timestamp, provided that there is no other undelivered message m' that could possibly receive a smaller global timestamp $sn(m')$.

Skeen, TOMP, MTO/SCALATOM, and Fritzke et al. fall into the first family of destinations agreement algorithms.

In Skeen's algorithm, the sender coordinates the computation of the global timestamp. Destination processes send the local timestamp to the sender, which collects them. The sender then sends back the global timestamp once it is computed. Algorithm 13 is in fact largely inspired from Skeen's algorithm. The main difference is that Skeen's algorithm computes the global timestamp in a centralized manner, while Algorithm 13 does it in a decentralized way (see Fig. 15).



(a) Skeen (centralized)

(b) Algorithm 13 (decentralized)

Figure 15: Centralized and decentralized approaches to compute a global timestamp

MTO and SCALATOM are genuine multicast algorithms that are based on Skeen's algorithm. The global timestamp is computed using a variant of Chandra and Toueg's Consensus algorithm [23]. MTO requires that all messages are delivered in causal order. SCALATOM fixes some problems found in MTO and lifts the restriction on causal order by the introduction of an additional communication step.

Fritzke et al. also describe a genuine multicast algorithm. The algorithm uses a consensus algorithm to decide on the timestamp of a multicast message within each destination group (just like the algorithms following the second approach; see below). Then the destination groups exchange information to compute the final timestamp. Finally, a second consensus is executed in each group to update the logical clock.

[26] Some algorithms require a Reliable Broadcast to the destinations, depending on the fault tolerance properties of the algorithm.

TOMP is merely an optimization of Skeen's algorithm, based on piggybacking additional information on messages in order to deliver messages earlier. All comments made on Skeen's algorithm (as described by [17]) also apply to TOMP.

In the algorithms of the second family, processes agree on the message (or sequence of messages) to be delivered next, rather than on some timestamp. This agreement is usually obtained using either a Consensus or some Atomic Commitment protocol. Among the algorithms that belong to this family, [23], Anceaume's prefix agreement [10], optimistic atomic broadcast [101], generic broadcast [102] are based on Consensus or variations thereof.

In Chandra and Toueg's algorithm, a sequence of Consensus executions is used to repeatedly agree on a set of received yet unordered messages. The order is then inferred from the set of messages, by applying a predefined deterministic function.

Anceaume defines a variant of Consensus, called *Prefix Agreement*, in which processes agree on a stream of values rather than on a single value. Considering streams rather than single values makes the algorithm particularly well suited to the problem of Total Order Multicast. The total order multicast algorithm then repeatedly decides on the sequence of messages to be delivered next.

Optimistic atomic broadcast is based on the observation that messages that are broadcasted in a LAN are usually received in the same order by every process. Although the algorithm is still correct without this assumption, its performance are much better if it holds.

Generic Broadcast is not a total order multicast per se. Instead, the order to enforce is a parameter of the algorithm and depends on the semantics of the messages. Depending on this parameter, the ordering properties of generic broadcast can vary from a simple unordered broadcast to total order. The strong point of this algorithm is that performance varies according to the required "amount of ordering."

Still in the second family of algorithms, [82], ATR [46], Minet and Anceaume's ABP [89], AMp [120], xAMp [110], and [81] are based on some voting protocol, inspired from two and three phase commit.

In AMp and xAMp, the algorithm relies on the assumption that messages are most of the time received by all destinations in the same order (not unrealistic in LANs). So, the sender initiates some commit protocol when its sends its message. If the message is received by all destination processes in order, then the outcome is positive and all destinations commit and hence deliver the message. Otherwise, the message is rejected and the sender must try again (thus potentially leading to a livelock).

In a nutshell, ABP and Luan and Gligor's algorithm are similar to each other. In both algorithms, the order of message is pre-established in an optimistic manner (assuming that no failure occurs). Then, the order is validated with a commit protocol before message can actually be delivered.

[46] describe their total order algorithm on top of a layer which provides the abstraction of synchronized rounds. A broadcast takes two rounds: in even rounds, all processes send an ordered set of messages to each other, and thus construct the order. In the subsequent round (odd), the order is validated, and messages can be delivered.

System model Most of the destinations agreement algorithms assume an asynchronous system model with failure detectors, reliable channels, and process crashes. For instance, [23], Prefix Agreement [10], MTO/SCALATOM [65, 109], [56], optimistic atomic broadcast [101] require a $\Diamond\mathcal{S}$ failure detector, and assume that a majority of the processes are correct. Generic broadcast [102] also needs a $\Diamond\mathcal{S}$ failure detector, but it presupposes that two thirds of the processes are correct. Delporte and Fauconier's ATR algorithm [46] relies on a $\Diamond\mathcal{P}$ failure detector

and reliable FIFO channels. The authors argue that their algorithm enforces additional guarantees if the system is synchronous.

Other destinations agreement algorithms assume an ad hoc asynchronous system model, with process crashes and lossy channels.[27] These algorithms are Skeen's algorithm (Isis) [17], [82], and [81]. The articles do not specify the exact conditions under which the algorithms are live; the assumption of an ad hoc asynchronous system model is implicit. Luan and Gligor's algorithm requires FIFO channels and that a majority of processes are correct.[28] Le Lann and Bres require an upper bound on the total number of failures (crash and omission). Skeen's algorithm (Isis) is based on a group membership service (Isis). The group membership service is needed for the liveness of the algorithm; without it, the algorithm would block with the crash of a single process.

Minet and Anceaume's ABP [89], AMp and xAMp [120, 110] assume a synchronous system model. These algorithms rely on a group membership service, and assume that there is a known bound on the number of omission failures that a message can suffer.

Specification It is not difficult to see that all destinations agreement algorithms satisfy Uniform Integrity. Most of them also satisfy Uniform Agreement and Total Order. These properties are proved for Chandra and Toueg's algorithm [23], MTO/SCALATOM [65, 109], Anceaume's prefix agreement [10], and ATR [46].

The authors of Optimistic atomic broadcast [101] and Generic broadcast [102] only prove Agreement and Total Order. Similarly, [56] only prove Agreement and Total Order. Nevertheless, those algorithms probably also ensure Uniform Agreement and Uniform Total Order.

The following algorithms also probably satisfy the uniform properties: Luan and Gligor's algorithm [82] (unclear for Uniform Agreement), [81], ABP [89], AMp and xAMp [120, 110] (unclear for Uniform Total Order).

Skeen's algorithm [17] only guarantees Total Order. Depending on the group membership service, Skeen's algorithm can possibly guarantee Uniform Agreement.

Few articles are concerned with the problem of contamination. Nevertheless, Fritzke et al., and Minet and Anceaume (ABP) explicitly state that their algorithm prevents contamination. This is probably also true for the other algorithms.

Architectural properties The protocols which use logical clock values (Isis/Skeen, TOMP, MTO/SCALATOM, and Fritzke et al.) provide multiple groups ordering with the Strong Minimality property [65] (see Sect. 4.1): only the sender and the destinations are involved in a multicast. The other algorithms only provide single group ordering.

The very structure of destinations agreement algorithms makes them relatively easy to adapt to open groups. Indeed, even if the sender is sometimes involved for coordination, it does not contribute to the ordering as such.

Skeen (Isis), Luan and Gligor, and ABP are closed group algorithms for the following reasons: the sender coordinates ordering, and the mechanism that handles failures requires the sender to be a member of the group. By transferring the coordination to one of the destinations, the algorithms could be adapted to open groups.

ATR only supports closed groups. This is due to the properties of the underlying layer that provides the abstraction of synchronized rounds. Also, AMp and xAMp quite heavily rely on the assumption that the network transmits messages in a certain order, and liveness can be violated if this does not hold often enough. This

[27] The exact semantics of the lossy channels is rarely specified.

[28] In fact, Luan and Gligor's algorithm requires a little more than a majority of correct processes, but we do not go into such details.

assumption may not be valid in an environment of interconnected subnetworks. As this puts a restriction on the processes which can broadcast messages, it is difficult to adapt the algorithm to open groups.

Failure management Skeen (Isis) [17], [82], [81], ABP [89], and AMp/xAMp [120, 110] explicitly tolerate message losses by retransmissions. Skeen uses a windowed point-to-point acknowledgement protocol. Note that this is nothing else but an explicit implementation of reliable channels. In the voting scheme of ABP, AMp and xAMp, the sender waits for all replies, thus it is natural to use positive acknowledgements. In Luan and Gligor's algorithm [82], the sender only waits for a majority of replies, thus the use of negative acknowledgements is more natural.

All of the algorithms tolerate process crashes. Skeen (Isis), ABP, AMp and xAMp rely on a group membership service and thus process controlled crash for failure management. Without a group membership service, these algorithms would be blocked even by a single failure. Unlike the others, the group membership service used by ABP requires the agreement of a majority of the processes to exclude a process (in the other algorithms, exclusion is unilateral).

There is a consensus algorithm (or a variant) based on the rotating coordinator paradigm at the heart of most other algorithms. In short, the rotating coordinator paradigm goes as follows: if the current coordinator is suspected, another process takes its place and tries to finish the algorithm; the suspected coordinator continues to participate in the algorithm as a normal process. More specifically, destinations agreement algorithms that are based on this mechanism are [23], MTO/SCALATOM [120, 110], [56], Prefix agreement [10], Optimistic atomic broadcast [101], and Generic broadcast [102].

In Luan and Gligor's algorithm [82], the voting scheme does not block if a process crashes, and there are mechanisms to bring slow replicas[29] up to date. If a sender fails, a termination protocol decides on the outcome of the broadcast (commit or abort).

In ATR [46], the layer that provides the abstraction of synchronized rounds also takes care of handling process crashes. *Phases* roughly correspond to the views of a group membership service, but provide less guarantees and are supposedly easier to implement.

7 Other Work on Total Order and Related Issues

Hardware-Based Protocols Due to their specificity, we deliberately omit all algorithms that make explicit use of dedicated hardware. They however deserve to be cited as related work. Hardware-based protocols rely on specific hardware systems in various ways and for different reasons. Some protocols are based on a modification of the network controllers [21, 44, 68, 89]. The goal of these protocols is to slightly modify the network so that it can be used as a virtual sequencer. In our classification system, these protocols can be assimilated as fixed sequencer protocols. Some other protocols rely on the characteristics of specific networks such as a specific topology [32] or the ability to reserve buffers [27]. Finally, some protocols are based on a specific network architecture and require that some computers are used for the sole purpose of serving the protocol [54, 119]. These protocols are designed with a specific application field in mind, and hence sacrifice flexibility for the sake of performance or scalability issues.

[29] A slow replica is one that always fails to reply on time.

Consensus Due to the equivalence of the two problems, the large amount of literature that exists on the problem of consensus also relates to the problem of total order. Many references about the consensus problem have already been mentioned in this paper. However, for a more comprehensive presentation on consensus, the reader is invited to refer to [13], [53], [23].

Formal Methods Formal methods have been also applied to the problem of total order multicast, in order to prove of verify the ordering properties of multicast algorithms [125, 118].

Group communication *vs* transaction systems A few years ago, [28] began a polemic about group communication systems that provide causally and totally ordered communication. Their major argument against group communication systems was that systems based on transactions are more efficient while providing a stronger consistency model. This was later answered by [16] and [117]. Almost a decade later it seems that this antagonistic view was a little shortsighted considering that the work on transaction systems and on group communication systems tend to influence each other for a mutual benefit [123, 100, 2, 112, 62, 63].

8 Conclusion

The vast literature on Total Order Multicast and the large number of algorithms published so far are good hints of the practical importance of this problem. However, this abundance of information is a problem itself, because it makes it difficult to understand the exact tradeoffs associated with each solution proposed.

The main contribution of this paper is the definition of a classification for Total Order Multicast algorithms, thus making it easier to understand the relationship between them. This also provides a good base for comparing the algorithms and understand the related tradeoffs. Furthermore, this paper presents a vast survey of the existing algorithms and discusses their respective characteristics.

In spite of the sheer number of Total Order Multicast algorithms that have been published, most of them are merely improvements of existing solutions. As a result, there are actually few algorithms as such, but a large collection of various optimizations. Nevertheless, it is important to stress that clever optimizations of existing algorithms often have an important effect on performance. For instance, [55] show that piggybacking messages can significantly improve the performance of algorithms.

Even though the specification of Atomic Broadcast dates back to some of the earliest publications about the subject (e.g., [36]), few papers actually specify the problem that they solve. In fact, too few algorithms are properly specified, let alone proven correct. Gladly, this is changing as the authors of many recent publications actually specify and prove their algorithms (e.g., [48, 23, 10, 109, 101, 46]). Without pushing formalism to extremes, a clear specification and a sound proof of correctness are as important as the algorithm itself. Indeed, they clearly define the limits within which the algorithm can be used.

Acknowledgements

We are grateful to Bernadette Charron-Bost for her very helpful comments on the formal aspects of this paper, and her interesting views regarding system models. We would like to also thank Dahlia Malkhi, Fernando Pedone, and Matthias Wiesmann for their valuable comments and criticisms that helped define the classification system proposed here.

References

[1] D. A. Agarwal, L. E. Moser, P. M. Melliar-Smith, and R. K. Budhia. The Totem multiple-ring ordering and topology maintenance protocol. *ACM Transactions on Computer Systems*, 16(2):93–132, May 1998.

[2] D. Agrawal, G. Alonso, A. El Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'97)*, number 1300 in Lecture Notes in Computer Science, pages 496–503, Passau, Germany, August 1997. Extended abstract.

[3] G. Alvarez, F. Cristian, and S. Mishra. On-demand asynchronous atomic broadcast. In *5th IFIP Working Conference on Dependable Computing for Critical Applications*, Urbana-Champaign, IL, USA, September 1995.

[4] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 76–84, Boston, MA, USA, July 1992.

[5] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS-13)*, pages 551–560, Pittsburgh, PA, USA, May 1993.

[6] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, November 1995.

[7] Y. Amir and J. Stanton. The Spread wide area group communication system. TR CDNS-98-4, John Hopkins University, Baltimore, MD, USA, 1998.

[8] E. Anceaume. *Algorithmique de Fiabilisation de Systèmes Répartis*. PhD thesis, Université de Paris-sud (Paris XI), Paris, France, 1993.

[9] E. Anceaume. A comparison of fault-tolerant atomic broadcast protocols. In *Proceedings of the 4th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-4)*, pages 166–172, Lisbon, Portugal, September 1993.

[10] E. Anceaume. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proceedings of the 27th International Symposium on Fault-Tolerant Computing (FTCS-27)*, pages 292–301, Seattle, WA, USA, June 1997.

[11] E. Anceaume and P. Minet. Étude de protocoles de diffusion atomique. TR 1774, INRIA, Rocquencourt, France, October 1992.

[12] H. Attiya, C. Dwork, N. Lynch, and L. Stockmeier. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM*, 41(1):122–152, January 1994.

[13] M. Barborak, M. Malek, and A. Dahbura. The consensus problem in distributed computing. *ACM Computing Surveys*, 25(2):171–220, June 1993.

[14] P. Berman and A. A. Bharali. Quick atomic broadcast. In *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG-7)*, number 725 in Lecture Notes in Computer Science, pages 189–203, Lausanne, Switzerland, September 1993. Extended abstract.

[15] N. T. Bhatti, M. A. Hiltunen, R. D. Schlichting, and W. Chiu. Coyote: A system for constructing fine-grain configurable communication services. *ACM Transactions on Computer Systems*, 16(4):321–366, November 1998.

[16] K. Birman. A response to Cheriton and Skeen's criticism of causal and totally ordered communication. *ACM Operating Systems Review*, 28(1):11–21, January 1994.

[17] K. P. Birman and T. A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[18] 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.

[19] K. P. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, 1993.

[20] R. Carr. The Tandem global update protocol. *Tandem Systems Review*, June 1985.

[21] M. Cart, J. Ferrié, and S. Mardyanto. Atomic broadcast protocol, preserving concurrency for an unreliable broadcast network. In J. Cabanel, G. Pujole, and A. Danthine, editors, *Proc. IFIP Conf. on Local Communication Systems: LAN and PBX*. Elsevier Science Publishers, 1987.

[22] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[23] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[24] J.-M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[25] B. Charron-Bost, F. Pedone, and X. Défago. Private communications. Showed an example illustrating the fact that even the combination of strong agreement, strong total order, and strong integrity does not prevent a faulty process from reaching an inconsistent state., November 1999.

[26] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (ICDSN/FTCS-30)*, New York, NY, USA, June 2000.

[27] X. Chen, L. E. Moser, and P. M. Melliar-Smith. Reservation-based totally ordered multicasting. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, pages 511–519, Hong Kong, May 1996.

[28] D. R. Cheriton and D. Skeen. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the 14th ACM Symp. on Operating Systems Principles (SoSP-14)*, pages 44–57, Asheville, NC, USA, December 1993.

[29] G.-M. Chiu and C.-M. Hsiao. A note on total ordering multicast using propagation trees. *IEEE Trans. on Parallel and Distributed Systems*, 9(2):217–223, February 1998.

[30] G. V. Chockler, N. Huleihel, and D. Dolev. An adaptive total ordered multicast protocol that tolerates partitions. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC-17)*, pages 237–246, Puerto Vallarta, Mexico, June 1998.

[31] B. Chor and C. Dwork. Randomization in Byzantine Agreement. *Adv. Comput. Res.*, 5:443–497, 1989.

[32] J. Córdova and Y.-H. Lee. Multicast trees to provide message ordering in mesh networks. *Computer Systems Science & Engineering*, 11(1):3–13, January 1996.

[33] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *Real-Time Systems*, 2(3):195–212, September 1990.

[34] F. Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, 1991.

[35] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From simple message diffusion to Byzantine agreement. Technical Report 4540, IBM Research Lab., San Jose, CA, USA, December 1984.

[36] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the 15th International Symposium on Fault-Tolerant Computing (FTCS-15)*, Ann Arbor, Michigan, USA, June 1985.

[37] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118(1):158–179, 1995.

[38] F. Cristian, R. de Beijer, and S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering*, 1(4):177–201, June 1994.

[39] F. Cristian, D. Dolev, R. Strong, and H. Aghili. Atomic broadcast in a real-time environment. In B. Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing*, number 448 in Lecture Notes in Computer Science, pages 51–71. Springer-Verlag, 1990.

[40] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. on Parallel & Distributed Systems*, 10(6):642–657, June 1999.

[41] F. Cristian and S. Mishra. The pinwheel asynchronous atomic broadcast protocols. In *2nd Int'l Symposium on Autonomous Decentralized Systems*, pages 215–221, Phoenix, AZ, USA, April 1995.

[42] F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. *Distributed System Engineering Journal*, 4(2):109–128, June 1997.

[43] M. Dasser. TOMP: A total ordering multicast protocol. *ACM Operating Systems Review*, 26(1):32–40, January 1992.

[44] P. Decitre, A. Khider, and G. Vandôme. Protocole de diffusion fiable pour réseaux locaux. Technical Report RR-347, Laboratoire IMAG, Grenoble, France, January 1983.

[45] Xavier Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, August 2000. Number 2229.

[46] C. Delporte-Gallet and H. Fauconnier. Real-time fault-tolerant atomic broadcast. In *Proceedings of the 18th Symposium on Reliable Distributed Systems (SRDS-18)*, pages 48–55, Lausanne, Switzerland, October 1999.

[47] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchrony needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[48] D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered multicast in asynchronous environments. In *Proceedings of the 23rd International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 544–553, Toulouse, France, June 1993.

[49] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communnication. *Communications of the ACM*, 39(4):64–70, April 1996.

[50] 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.

[51] M. Endler. An atomic multicast protocol for mobile computing. In *Proceedings of the 3rd International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (DIAL M-3)*, pages 56–63, 1999.

[52] P. D. Ezhilchelvan, R. A. Macêdo, and S. K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS-15)*, pages 296–306, Vancouver, Canada, May 1995.

[53] 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, April 1985.

[54] A. Freier and K. Marzullo. MTP: An atomic multicast transport protocol. Technical Report TR90-1141, Cornell University, Computer Science Department, July 1990.

[55] R. Friedman and R. Van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *6th IEEE Symposium on High Performance Distributed Computing*, pages 233–242, Portland, OR, USA, August 1997.

[56] U. Fritzke, P. Ingels, A. Mostéfaoui, and M. Raynal. Fault-tolerant total order multicast to asynchronous groups. In *Proceedings of the 17th Symposium on Reliable Distributed Systems (SRDS-17)*, pages 228–234, West Lafayette, IN, USA, October 1998.

[57] C. Gailliard and D. Hué-Petillat. Un protocole de diffusion atomique. *TSI*, 2(6), December 1983.

[58] H. Garcia-Molina and A. Spauster. Message ordering in a multicast environment. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS-9)*, pages 354–361, Newport Beach, CA, USA, June 1989.

[59] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[60] A. Gopal and S. Toueg. Reliable broadcast in synchronous and asynchronous environment. In *Proceedings of the 3rd International Workshop on Distributed Algorithms (WDAG-3)*, number 392 in Lecture Notes in Computer Science, pages 111–123, Nice, France, September 1989.

[61] A. Gopal and S. Toueg. Inconsistency and contamination. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC-10)*, pages 257–272, Proceedings of the 10th ACM Symposium on Principles of Distributed Computing (PODC-10), August 1991.

[62] R. Guerraoui and A. Schiper. A generic multicast primitive to support transactions on replicated objects in distributed systems. In *Proceedings of the 5th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-5)*, pages 334–342, Cheju Island, Korea, August 1995.

[63] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: bridging the gap. In *Theory and Practice in Distributed Systems*, LNCS 938, pages 121–132. Springer-Verlag, September 1995.

[64] R. Guerraoui and A. Schiper. Genuine atomic multicast. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG-11)*, number 1320 in Lecture Notes in Computer Science, pages 141–154, Saarbrücken, Germany, September 1997.

[65] R. Guerraoui and A. Schiper. Total order multicast to multiple groups. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS-17)*, pages 578–585, Baltimore, Maryland, USA, May 1997.

[66] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 5, pages 97–146. Addison-Wesley, second edition, 1993.

[67] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. TR 94-1425, Dept. of Computer Science, Cornell University, Ithaca, NY, USA, May 1994.

[68] P. Jalote. Efficient ordered broadcasting in reliable CSMA/CD networks. In *Proceedings of the 18th International Conference on Distributed Computing Systems (ICDCS-18)*, pages 112–119, Amsterdam, The Netherlands, May 1998.

[69] P. R. James, M. Endler, and M.-C. Gaudel. Development of an atomic-broadcast protocol using LOTOS. *Software—Practice and Experience*, 29(8):699–719, 1999.

[70] X. Jia. A total ordering multicast protocol using propagation trees. *IEEE Trans. on Parallel and Distributed Systems*, 6(6):617–627, June 1995.

[71] M. F. Kaashoek and A. S. Tanenbaum. Fault tolerance using group communication. *ACM Operating Systems Review*, 25(2):71–74, April 1991.

[72] M. F. Kaashoek and A. S. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS-11)*, pages 222–230, Arlington, TX, USA, May 1991.

[73] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing (PODC-15)*, pages 68–76, Philadelphia, PA, USA, May 1996.

[74] I. Keidar and D. Dolev. Totally ordered broadcast in the face of network partitions. In D. Avresky, editor, *Dependable Network Computing*, chapter 3, pages 51–75. Kluwer Academic Publications, January 2000.

[75] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS-19)*, pages 424–431, Austin, TX, USA, June 1999.

[76] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The MARS approach. *IEEE Micro*, 9(1):25–40, February 1989.

[77] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avižienis and J.-C. Laprie, editors, *Dependable Computing for Critical Applications (DCCA)*, volume 4, pages 411–429, 1990.

[78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[79] L. Lamport. Using time instead of time-outs in fault-tolerant systems. *ACM Transactions on Programming Languages and Systems*, 6(2):256–280, 1984.

[80] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[81] G. Le Lann and G. Bres. Reliable atomic broadcast in distributed systems with omission faults. *ACM Operating Systems Review, SIGOPS*, 25(2):80–86, April 1991.

[82] S.-W. Luan and V. D. Gligor. A fault-tolerant protocol for atomic broadcast. *IEEE Trans. on Parallel and Distributed Systems*, 1(3):271–285, July 1990.

[83] D. Malkhi. *Multicast Communication for High Availability*. PhD thesis, Hebrew University of Jerusalem, Israel, May 1994.

[84] C. P. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, September 1996.

[85] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, TX, USA, October 1995.

[86] E. Mayer. An evaluation framework for multicast ordering protocols. In *Proceedings of the Conference on Applications, Technologies, Architecture, and Protocols for Computer Communication (SIGCOMM)*, pages 177–187, August 1992.

[87] P. M. Melliar-Smith and L. E. Moser. Fault-tolerant distributed systems based on broadcast communication. In *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS-9)*, pages 129–134, Newport Beach, CA, USA, June 1989.

[88] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):17–25, January 1990.

[89] P. Minet and E. Anceaume. ABP: An atomic broadcast protocol. Technical Report 1473, INRIA, Rocquencourt, France, June 1991.

[90] P. Minet and E. Anceaume. Atomic broadcast in one phase. *ACM Operating Systems Review*, 25(2):87–90, April 1991.

[91] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: a communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(2):87–103, 1993.

[92] L. E. Moser and P. M. Melliar-Smith. Total ordering algorithms for asynchronous Byzantine systems. In *Proceedings of the 9th International Workshop on Distributed Algorithms (WDAG-9)*, number 972 in Lecture Notes in Computer Science, pages 242–256, Le Mont-St-Michel, France, September 1995.

[93] L. E. Moser and P. M. Melliar-Smith. Byzantine-resistant total ordering algorithms. *Information and Computation*, 150(1):75–111, April 1999.

[94] L. E. Moser, P. M. Melliar-Smith, D. A. Agrawal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[95] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Total ordering algorithms. In *ACM Annual Computer Science Conference, Preparing for the 21st Century*, pages 375–380, 1991.

[96] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. *SIAM Journal on Computing*, 22(4):727–750, August 1993.

[97] S. Navaratnam, S. T. Chanson, and G. W. Neufeld. Reliable group communication in distributed systems. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS-8)*, pages 439–446, San Jose, CA, USA, June 1988.

[98] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[99] T. P. Ng. Ordered broadcasts for large applications. In *Proceedings of the 10th Symposium on Reliable Distributed Systems (SRDS-10)*, pages 188–197, Pisa, Italy, September 1991.

[100] F. Pedone, R. Guerraoui, and A. Schiper. Exploiting atomic broadcast in replicated databases. In *Proceedings of EuroPar (EuroPar'98)*, number 1470 in Lecture Notes in Computer Science, pages 513–520, Southampton, UK, September 1998.

[101] F. Pedone and A. Schiper. Optimistic atomic broadcast. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC-12)*, number 1499 in Lecture Notes in Computer Science, pages 318–332, Andros, Greece, September 1998.

[102] F. Pedone and A. Schiper. Generic broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC-13)*, number 1693 in Lecture Notes in Computer Science, pages 94–108, Bratislava, Slovak Republic, September 1999.

[103] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, 1989.

[104] S. Poledna. Replica determinism in distributed real-time systems: A brief survey. *Real-Time Systems*, 6(3):289–316, May 1994.

[105] B. Rajagopalan and P. McKinley. A token-based protocol for reliable, ordered multicast communication. In *Proceedings of the 8th Symposium on Reliable Distributed Systems (SRDS-8)*, pages 84–93, Seattle, WA, USA, October 1989.

[106] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems*, number 938 in Lecture Notes in Computer Science, pages 99–110, Dagstuhl Castle, Germany, September 1994. Springer-Verlag.

[107] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security (CCS-2)*, pages 68–80, 1994.

[108] L. Rodrigues, H. Fonseca, and P. Veríssimo. Totally ordered multicast in large-scale systems. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, pages 503–510, Hong Kong, May 1996.

[109] L. Rodrigues, R. Guerraoui, and A. Schiper. Scalable atomic multicast. In *Proc. of the 7th IEEE International Conference on Computer Communications and Networks*, pages 840–847, Lafayette (LA), USA, October 1998.

[110] L. Rodrigues and P. Veríssimo. *x*AMP: a multi-primitive group communications service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems (SRDS-11)*, pages 112–121, Houston, TX, USA, October 1992.

[111] L. Rodrigues, P. Veríssimo, and A. Casimiro. Using atomic broadcast to implement a posteriori agreement for clock synchronization. In *Proceedings of the 12th Symposium on Reliable Distributed Systems (SRDS-12)*, pages 115–124, Princeton, NJ, USA, October 1993.

[112] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[113] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, ACM Press Books, chapter 7, pages 169–198. Addison-Wesley, second edition, 1993.

[114] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[115] N. Sergent, X. Défago, and A. Schiper. Failure detectors: Implementation issues and impact on consensus performance. Technical Report SSC/1999/019, École Polytechnique Fédérale de Lausanne, Switzerland, May 1999.

[116] S.-P. Shieh and F.-S. Ho. A comment on "a total ordering multicast protocol using propagation trees". *IEEE Trans. on Parallel and Distributed Systems*, 8(10):1084, October 1997.

[117] S. K. Shrivastava. To CATOCS or not to CATOCS, that is the .. *ACM Operating Systems Review*, 28(4):11–14, October 1994.

[118] C. Toinard, G. Florin, and C. Carrez. A formal method to prove ordering properties of multicast algorithms. *ACM Operating Systems Review*, 33(4):75–89, October 1999.

[119] L. C. N. Tseung. Guaranteed, reliable, secure broadcast networks. *IEEE Network Magazine*, 3(6):33–37, November 1989.

[120] P. Veríssimo, L. Rodrigues, and M. Baptista. AMp: A highly parallel atomic multicast protocol. *Computer Communication Review*, 19(4):83–93, September 1989. Proc. of the SIGCOMM'89 Symposium.

[121] P. Veríssimo, L. Rodrigues, and A. Casimiro. CesiumSpray: A precise and accurate global clock service for large-scale systems. *Real-Time Systems*, 12(3):243–294, May 1997.

[122] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In Springer-Verlag, editor, *Theory and Practice in Distributed Systems*, number 938 in Lecture Notes in Computer Science, pages 33–57, Dagstuhl Castle, Germany, September 1994.

[123] M. Wiesmann, F. Pedone, and A. Schiper. A systematic classification of replicated database protocols based on atomic broadcast. In *Proceedings of the 3rd European Research Seminar on Advances in Distributed Systems (ERSADS'99)*, Madeira, Portugal, April 1999.

[124] U. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *Proceedings of the 14th Symposium on Reliable Distributed Systems (SRDS-14)*, pages 106–115, Bad Neuenahr, Germany, September 1995.

[125] P. Zhou and J. Hooman. Formal specification and compositional verification of an atomic broadcast protocol. *Real-Time Systems*, 9(2):119–145, September 1995.