Total Order Broadcast and Multicast Algorithms:[*]
Taxonomy and Survey

Xavier Défago

defago@jaist.ac.jp

Graduate School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan

"Information and Systems," PRESTO,
Japan Science and Technology Agency (JST)

André Schiper and **Péter Urbán**

{andre.schiper,peter.urban}@epfl.ch

School of Information and Communication Sciences
Swiss Federal Institute of Technology in Lausane (EPFL)
CH-1015 Lausanne, Switzerland

Abstract

Total order broadcast and multicast (also called atomic broadcast or atomic multicast) is an important problem in distributed systems, especially with respect to fault-tolerance. In short, the primitive ensures that messages sent to a set of processes are delivered by all these processes in the same total order.

The problem has inspired an abundant literature, with a plethora of proposed algorithms. This paper proposes a classification of total order broadcast and multicast algorithms based on their ordering mechanisms, and addresses a number of other important issues. The paper surveys about sixty algorithms, thus providing by far the most extensive study of the problem so far. The paper discusses algorithms for both the synchronous and the asynchronous system models, and studies the respective properties and behavior of the different algorithms.

Contents

1 Introduction

Distributed systems and applications are notoriously difficult to build. This is mostly due to the unavoidable concurrency in such systems, combined with the difficulty of providing a global control. This difficulty is greatly reduced by relying on group communication primitives that provide higher guarantees than standard point-to-point communication. One such primitive is called total order broadcast.[1,2] Informally, the primitive ensures that messages sent to a set of processes are delivered by all these processes in the same order. Total order broadcast is an important primitive that plays, for instance, a central role when implementing the state machine approach (also called active replication) [Lamport 1978a, Schneider 1990, Poledna 1994]). It has also other applications, such as clock synchronization [Rodrigues et al. 1993], computer supported cooperative writing, distributed shared memory, or distributed locking [Lamport 1978b]. More recently, it was also shown that an adequate use of total order broadcast can significantly improve the performance of replicated databases [Agrawal et al. 1997, Pedone et al. 1998, Kemme et al. 2003].

Literature on total order broadcast There exists a considerable amount of literature on total order broadcast, and many algorithms, following various approaches, have been proposed to solve that problem. 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 new requirements, the absence of a roadmap to the problem of total order broadcast 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.

---

[1]Total order broadcast is also known as atomic broadcast. Both terminologies are currently in use. There is a slight controversy with respect to using one over the other. We opt for the former, that is, "total order broadcast," because the latter is somewhat misleading. Indeed, atomicity suggests a property related to agreement rather than total order (defined in Sect. 3), and the ambiguity has already been a source of misunderstandings. In contrast, "total order broadcast" unambiguously refers to the property of total order.

[2]Total order *multicast* is sometimes used instead of total order *broadcast*. The distinction between the two primitives is explained later in the paper. When the distinction is not important, we use the term total order *broadcast* in the paper.

Related work Previous attempts have been made at classifying and comparing total order broadcast algorithms [Anceaume 1993b, Anceaume and Minet 1992, Cristian et al. 1994, Friedman and van Renesse 1997, Mayer 1992]. 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 [Anceaume and Minet 1992] (an extended version was later published in French by [Anceaume 1993b]) 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. Besides, the authors only study seven different algorithms, which are not truly representative: for instance, none is based on a communication history approach (one of the five classes of algorithms; details in Sect. 7.4).

[Cristian et al. 1994] 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.

[Friedman and van Renesse 1997] study the impact that packing messages has on the performance of algorithms. To this purpose, they study six algorithms, including those studied by [Cristian et al. 1994]. They measure the actual performance of those algorithms and confirm the observations made by [Cristian et al. 1994]. They show that packing several protocol messages into a single physical message indeed provides an effective way to improve the performance of algorithms. The comparison also lacks generality, but this is quite understandable as this is not the main concern of that paper.

[Mayer 1992] defines a framework in which total order broadcast 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 comparison of algorithms. However, the paper does not go so far as to actually compare the numerous existing algorithms.

Contributions In this paper, we propose a classification of total order broadcast algorithms based on the mechanism used to order messages. The reason for this choice is that the ordering mechanism is the characteristic with the strongest influence on the communication pattern of the algorithm: two algorithms of the same class are hence likely to exhibit similar behaviors. We define five classes of ordering mechanisms: *communication history*, *privilege-based*, *moving sequencer*, *fixed sequencer*, and *destinations agreement*.

In this paper, we also provide a vast survey of about sixty published total order broadcast algorithms. Wherever possible, we mention the properties and the assumptions of each algorithm. This is however not always possible because the information available in the papers is often not sufficient to accurately characterize the behavior of the algorithm (e.g., in the face of a failure).

Structure The paper is logically organized into three main parts: specifications, mechanisms, and survey. More precisely, the rest of the paper is structured as follows.

The first part of the paper is about definitions, specifications, and properties. Section 2 introduces important concepts, terminology, and notations. Section 3 presents the most common specification of the total order broadcast problem (also known as atomic broadcast). Based on this specification, Sections 4–6 discuss several important properties (and other issues) and their impact on the specification of the problem. More specifically, Section 4 discusses possible additional properties of algorithms (e.g., uniformity, ordering), Section 5 is concerned with the characteristics of destination groups (e.g., single versus multiple groups), and Section 6 illustrates the impact of the system model on the specification when considering partitionable systems and Byzantine failures.

The second part describes mechanisms. In Section 7, we define the following five classes of total order broadcast algorithms, according to the way messages are ordered: *communication history*, *privilege-based*, *moving sequencer*, *fixed sequencer*, and *destinations agreement*. Section 8 discusses the issue of fault-tolerance from a general perspective.

In the third part, we review existing algorithms. More specifically, Section 9 gives a broad survey of total order broadcast algorithms found in the literature. The algorithms are grouped along their respective classes, and we discuss the principal characteristics of each algorithm.

In Section 10, we talk about various other issues that are relevant to total order broadcast, and Section 11 concludes the paper.

$\mathcal{M}$	set of all valid messages.
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.
Π_{dest}	set of all destination processes in the system.

Table 1: Notation.

2 Basic Terminology and Notation

2.1 Notation

Table 1 summarizes some of the notation used throughout the paper. $\mathcal{M}$ is the set containing all possible valid messages. Π denotes the set of all processes in the system, which can be arbitrarily large. Given some arbitrary message m, $sender(m)$ designates the process in Π from which m originates, and $Dest(m)$ denotes the set of all destination processes for m.

In addition, Π_{sender} is the set of all processes in Π that can potentially send some message.

$$\Pi_{sender} \stackrel{\text{def}}{=} \bigcup_{m \in \mathcal{M}} sender(m) \tag{1}$$

Likewise, Π_{dest} is the set of all potential destinations.

$$\Pi_{dest} \stackrel{\text{def}}{=} \bigcup_{m \in \mathcal{M}} Dest(m) \tag{2}$$

2.2 Basic System Models

A distributed system consists of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that interact by exchanging uniquely identified messages through communication channels. There exist a quantity of models that restrict the behavior of the system. The most important characteristics to consider are its synchrony and failure modes.

2.2.1 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 *process speed interval*, which is given by the difference in the speed of the slowest and the fastest processes 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 considers the following levels 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 wherein both parameters are assumed to satisfy (1) is called a *synchronous system*. At the other extreme, a system in which 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 [Dolev et al. 1987, Dwork et al. 1988].

---

[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.*

2.2.2 Process Failures

The failure modes of a system specify the kinds of failures that are expected to occur, as well as the conditions under which these failures may or may not occur. A commonly used set of process failure classes is as follows:

- *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 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. 9), we implicitly refer to *authenticated* Byzantine failures.

  A *correct* process is defined as a process that never expresses any of the faulty behaviors mentioned above.

Note 1 ((On the peculiarities of timing failures)) *A system is characterized by its failure modes and the "amount of synchrony" it exhibits. While the failure modes are 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.*

2.2.3 Communication

There exist several definitions of communication channels, according to the guarantees they provide. We are concerned with the types of communication channels mentioned below. Unless stated otherwise, it is assumed in this paper that communication channels neither duplicate messages nor generate spurious ones.

Reliable channels   Reliable channels guarantee that if a correct process p sends a message m to a correct process q, then q will eventually receive m.[4] It is often assumed that reliable communication is provided by the network protocol stack (e.g., TCP/IP).

Lossy channels   Lossy channels are channels subject to the loss of messages. Some common reasons for losing messages are: network collisions, noisy channels, overloaded buffers, disconnected lines, corrupt routing tables, or intermittent connections. Although message losses are often taken care of by mechanisms implemented in the network stack (between physical and transport layers), situations may arise wherein the guarantees offered by the network are inadequate or insufficient. One can distinguish between two types of lossy channels.

In the simplest case, communication channels have an upper bound k on the number of message loss. Coping with such loses is easy, as it is sufficient to send a message $k + 1$ times in order to ensure that at least one copy is received. The model is poorly suited to represent systems in which message losses are not independent.

In contrast, fair-lossy channels allow for an unbounded number of message losses. In short, fair lossy communication channels are defined as follows [Basu et al. 1996]. The channels do not produce spurious messages, do not replicate messages, and do not transform the content of messages. In addition, a fair lossy channel guarantees that if an infinite number of messages are sent, an infinite subset of those messages is received.

---

[4][Aguilera et al. 1997] call such channels *quasi-reliable* to contrast them to reliable channels as defined by [Basu et al. 1996]. The latter definition assumes that m is eventually delivered to a correct process q even if p is faulty, which is not very realistic.

2.3 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.[5] 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.

2.3.1 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 being synchronized. 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 ϵ.

Depending on the assumptions, the information issued by the clocks can or cannot be related to real-time. Synchronized clocks are not necessarily 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 wide-area systems. Indeed, [Veríssimo et al. 1997] achieve clock synchronization with an accuracy of a few microseconds. In contrast, the accuracy usually obtained with software-based clock synchronization mechanisms is several orders of magnitude lower.

2.3.2 Failure Detectors

A failure detector is an oracle which provides information about the current status of processes, for instance, whether a given process has crashed or not.

The notion of failure detectors has been formalized by [Chandra and Toueg 1996]. Briefly, a failure detector is modeled 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 module FD_i about the status of other processes.

Failure detectors are considered *unreliable*, in the sense that they provide information that may not always correspond to the real state of the system. For instance, a failure detector module FD_i may provide the erroneous information that some process p_j has crashed whereas, in reality, p_j is correct and running. Conversely, FD_i may provide the information that a process p_k is correct, while p_k has actually crashed.

To reflect the unreliability of the information provided by failure detectors, we say that a process p_i *suspects* some process p_j whenever FD_i, the failure detector module attached to p_i, returns the (unreliable) information that p_j has crashed. In other words, a suspicion is a belief (e.g., "p_i believes that p_j has crashed") as opposed to a known fact (e.g., "p_j has crashed and p_i knows that").

There exist several classes of failure detectors, depending on how unreliable the information provided by the failure detector can be. This is defined by two properties, *completeness* and *accuracy*, which constrain the possible mistakes. These properties are better explained by an example. The class of failure detectors $\Diamond \mathcal{S}$ is defined by the following properties [Chandra and Toueg 1996]:

(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.

There exist other classes of failure detectors, but a complete description of all failure detectors that are presented by [Chandra and Toueg 1996] is well beyond the scope of this paper.

---

[5]Suggested by Bernadette Charron-Bost.

2.3.3 Random Oracle

Another approach to extend the power of a system model consists in introducing the ability to generate random values. For instance, processes could have access to a module that generates a random bit when queried (i.e., a Bernoulli random variable).

This is used by a class of algorithms called randomized algorithms. Those algorithms can solve problems such as consensus in a probabilistic manner. The probability that such algorithms terminate before some time t goes to one as t goes to infinity (e.g., [Ben-Or 1983, Chor and Dwork 1989]). Note that solving a problem deterministically and solving it with probability 1 are not the same.

2.4 Agreement Problems

Agreement problems constitute a fundamental class of problems in 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 the following four important agreement problems: *reliable broadcast*, *Byzantine agreement*, *consensus*, and *total order broadcast*.

2.4.1 Reliable Broadcast

As the name indicates, reliable broadcast is defined as a broadcast primitive. In short, reliable broadcast of message m guarantees that 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.

2.4.2 Byzantine Agreement

The problem of Byzantine agreement is also commonly known as the "Byzantine generals problem" [Lamport et al. 1982]. 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.

2.4.3 Consensus

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

2.4.4 Total Order Broadcast

The problem of total order broadcast, also known as atomic broadcast, is an agreement problem. In short, it is defined as a reliable broadcast problem which must also ensure that all delivered messages are delivered by all processes in the same order. The exact specification of the problem is given in Section 3.

2.4.5 Important Theoretical Results

There are at least four fundamental theoretical results that are directly relevant to the problem of total order broadcast and consensus. First, total order broadcast and consensus are equivalent problems, i.e., if there exists an algorithm that solves one problem, then it can be transformed to solve the other problem.[7] [Dolev et al. 1987] show that total order broadcast can be transformed into consensus, and [Chandra and Toueg 1996] show that consensus can be transformed into total order broadcast. Second, there is no deterministic solution to the problem of consensus in asynchronous systems if just a single process can crash [Fischer et al. 1985]. Nevertheless,

---

[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.

[7]The equivalence also holds in asynchronous systems with arbitrary failures, see [Chandra and Toueg 1996].

consensus can be solved in asynchronous systems extended with failure detectors [Chandra and Toueg 1996], with partial synchrony [Dolev et al. 1987, Dwork et al. 1988], or using randomization (see Sect. 2.3.3). Finally, [Chandra et al. 1996] have shown that the weakest failure detector to solve consensus in an asynchronous system is of class $\Diamond\mathcal{S}$.[8]

2.5 A Note on Asynchronous Total Order Broadcast Algorithms

In many papers about total order broadcast, the authors claim that their algorithm solves the problem in asynchronous systems with process failures. This claim is of course incorrect (see previous section), or incomplete at best.

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 agreement problems in real systems? This is because many papers do not formally address the liveness issue of the algorithm, or regard it as sufficient to consider some informal level of synchrony, captured by the assumption that "most messages are likely to reach their destination within a known delay δ" [Cristian et al. 1997, Cristian and Fetzer 1999]. This model is known as the timed asynchronous model [Cristian and Fetzer 1999] and is 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. However, formally this is not enough to establish correctness.

2.6 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 augments 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.

Automatic transformation of failures   [Neiger and Toueg 1990] 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.[9]

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   A perfect failure detector ($\mathcal{P}$) satisfies both strong completeness and strong accuracy (no process is suspected before it crashes [Chandra and Toueg 1996]). In practical systems, perfect failure detectors are extremely difficult to implement because of the difficulty to distinguish crashed processes from very slow ones. [Fetzer 2003] proposes a protocol to emulate a perfect failure detector in a timed asynchronous model with process controlled crash. His protocol uses watchdog (hardware or software) and ensures that no process is suspected before it crashes. Process controlled crash makes it also possible to emulate an *almost* perfect failure detector that satisfies a weaker accuracy property:

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

The idea of the emulation is simple. Let $\mathcal{X}$ be a failure detector that satisfies strong completeness and any form of accuracy: whenever $\mathcal{X}$ suspects a process p, then p is killed (forced to crash). As a result, false suspicions are corrected *a posteriori*, and the above "quasi-strong accuracy" property is satisfied. A primary partition group membership service with process controlled crash (see Sect. 8.2) typically emulates such a failure detector, which is used by several total order broadcast algorithm (see Sect. 9).

---

[8]The weakest failure detector to solve consensus is usually said to be $\Diamond\mathcal{W}$, which differs from $\Diamond\mathcal{S}$ by satisfying a weak completeness property instead of Strong Completeness. However, [Chandra and Toueg 1996] prove the equivalence of $\Diamond\mathcal{S}$ and $\Diamond\mathcal{W}$.

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

Cost of a Free Lunch Process controlled crash is often used in practice in the context of total order broadcast algorithms. However, the mechanisms has a price.

To understand this, 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 caused by some process, e.g., they are the result of process controlled crash.

A fault-tolerant algorithm can only tolerate the crash of a bounded number of processes.[10] In a system with process controlled crash, this limit includes not only 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.

3 Specification (Total Order Broadcast)

In this section, we give the formal specification of the total order broadcast problem. Although there exist many variants of Total Order Broadcast depending on factors such as the system model, this section describes the problem in its simplest form, i.e., crash failures and closed system. Then, in Sections 4 through 6, we consider several issues that have an influence on the algorithms, such as Byzantine failures, uniformity, or network partitions.

Formally, total order broadcast is defined in terms of two primitives, which are called *TO-broadcast*(m) and *TO-deliver*(m), where $m \in \mathcal{M}$ is some message. When a process p executes *TO-broadcast*(m) (respectively *TO-deliver*(m)), we may say that p TO-broadcasts m (respectively TO-delivers m). We assume that every message m can be uniquely identified, and carries the identity of its sender, denoted by $sender(m)$. In addition, we assume that, for any given message m and any run, *TO-broadcast*(m) is executed at most once. In this context, total order broadcast is defined by the following properties [Hadzilacos and Toueg 1994, Chandra and Toueg 1996]:

(VALIDITY) If a correct process TO-broadcasts a message m, then it eventually TO-delivers m.

(UNIFORM AGREEMENT) If a process TO-delivers a message m, then all correct processes eventually TO-deliver m.

(UNIFORM INTEGRITY) For any message m, every process TO-delivers m at most once, and only if m was previously TO-broadcast by $sender(m)$.

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

Validity and Uniform Agreement are liveness properties. Roughly speaking this means that, at any point in time, no matter what has happened up to that point, it is still possible for the property to eventually hold [Charron-Bost et al. 2000]. Uniform Integrity and Uniform Total Order are safety properties. This means that, if at some point in time the property does not hold, no matter what happens later, the property cannot eventually hold. Note that [Charron-Bost et al. 2000] have shown that, in the context of failures, some (non-uniform) properties that are commonly believed to be safety properties are actually liveness properties. They have proposed refinements of the concept of safety and liveness that avoid the counterintuitive classification.

Note 2 *The above definition is the most common definition of total order broadcast. However, in spite of its popularity, the definition is known to be prone to an important flaw called contamination. This issue is discussed in Sect. 4.2, where we give a better formulation for the order property.*

4 Properties of Algorithms

4.1 Uniformity

In the above definition of total order broadcast, the properties of agreement and total order are *uniform*. This means that these properties do not only apply to correct processes, but also to faulty ones. For instance, with Uniform Total Order, a process is not allowed to deliver any message out of order, even if it is faulty. Conversely,

---

[10]The recovery of processes and the dynamic join of new processes are discussed in Section 8.2.
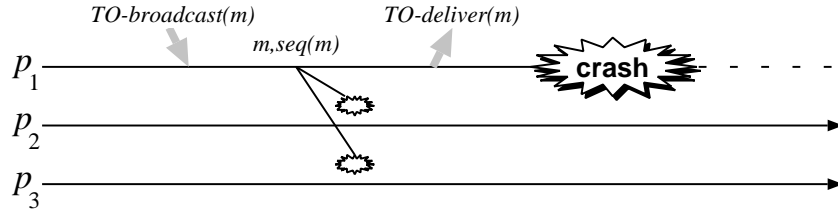
Figure 1: Violation of Uniform Agreement (example)

(non-uniform) Total Order applies only to correct processes, and hence does not put any restriction on the behavior of faulty processes.

Uniform properties are required by some classes of applications such as atomic commitment. However, since enforcing uniformity in an algorithm often has a considerable performance cost, it is also important to consider weaker problems specified using non-uniform properties. Non-uniform properties may lead to inconsistencies at the application level. However, this is not always a problem, particularly if the application knows how to correct such inconsistencies. Non-uniform Agreement and Total Order are specified as follows:

(AGREEMENT) If a **correct** process TO-delivers a message m, then all correct processes eventually TO-deliver m.

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

The combinations of uniform and non-uniform properties define four different specifications to the problem of fault-tolerant total order broadcast. Those definitions constitute a hierarchy of problems, as discussed extensively by [Wilhelm and Schiper 1995].

Figure 1 illustrates a violation of the Uniform Agreement property with a simple example. In this example, the sequencer p_1 sends a message m using total order broadcast. It first assigns a sequence number to m, then sends m to all processes, and finally delivers m. Process p_1 crashes shortly afterwards, and no other process receives m (this is possible, see Sect. 2.2.3). As a result no correct process (e.g., p_2) will ever be able to deliver m. Uniform Agreement is violated, but not (non-uniform) Agreement: no *correct* process ever delivers m (p_1 is not correct).

Note 3 *[Guerraoui 1995] shows that any algorithm that solves Consensus with $\Diamond \mathcal{P}$ (respectively $\mathcal{S}$, $\Diamond \mathcal{S}$), also solves uniform consensus with $\Diamond \mathcal{P}$ (respectively $\mathcal{S}$, $\Diamond \mathcal{S}$).*

It is easy to show that this result also holds for total order broadcast. Assume that there exists an algorithm that solves non-uniform total order broadcast (non-uniform Agreement, non-uniform Total Order) with $\Diamond \mathcal{P}$, $\mathcal{S}$ or $\Diamond \mathcal{S}$, but does not solves uniform total order broadcast. Using the transformation of total order broadcast to consensus (see Sect. 2.4.5) this algorithm could be used to obtain an algorithm that solves non-uniform consensus but not consensus. A contradiction.

Note however that the result does not hold for total order broadcast algorithms that rely on a perfect or almost perfect failure detector (see Sect. 2.6).

4.2 Contamination

The problem of contamination comes from the observation that, even with the strongest specification (i.e., with Uniform Agreement and Uniform Total Order), total order broadcast does not prevent a faulty process p from reaching an inconsistent state (i.e., before it crashes). This is a serious problem because p can "legally" TO-broadcast a message based on this inconsistent state, and thus *contaminate* correct processes [Gopal and Toueg 1991, Anceaume and Minet 1992, Anceaume 1993b, Hadzilacos and Toueg 1994].

4.2.1 Illustration

Figure 2 illustrates an example [Charron-Bost et al. 1999, Hadzilacos and Toueg 1994] where an incorrect process contaminates the correct processes. Process p_3 delivers messages m_1 and m_3, but not m_2. So, its state is
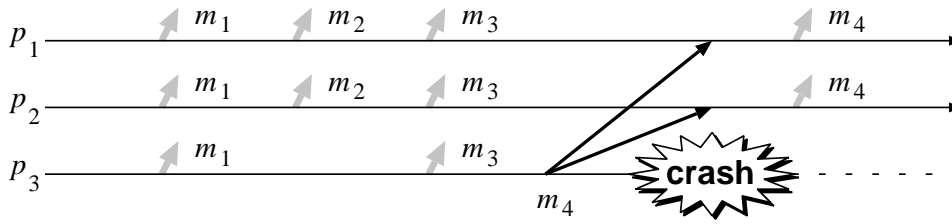
Figure 2: Contamination of correct processes (p_1, p_2) by a message (m_4) based on an inconsistent state (p_3 delivered m_3 but not m_2).

inconsistent when it multicasts m_4 to the other processes before crashing. The correct processes p_1 and p_2 deliver m_4, thus getting contaminated by the inconsistent state of p_3. It is important to stress again that the situation depicted in Figure 2 satisfies even the strongest specification.

4.2.2 Specification

It is possible to extend or reformulate the specification of total order broadcast in such a way that it disallows contamination. This can be achieved in two ways. The first option 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 solution is usually preferred, which consists in preventing any process from delivering a message that may lead to an inconsistent state.

Aguilera, Delporte-Gallet et al. [2000] propose a reformulation of Uniform Total Order which, unlike the traditional definition, is not prone to contamination as it does not allow gaps in the delivery sequence:

(GAP-FREE UNIFORM TOTAL ORDER) If some process delivers message m' after message m, then a process delivers m' only after it has delivered m.

As an alternative, an older formulation uses the history of delivery and requires that, for any two given processes, the history of one is a prefix of the history of the other. This is expressed by the following property [Anceaume and Minet 1992, Cristian et al. 1994, Keidar and Dolev 2000]:

(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 sequences of messages delivered by p and q, respectively.

Note 4 *The specification of total order broadcast using Prefix Order in fact precludes the dynamic join of processes (e.g., with a group membership). This can be circumvented, but the resulting property is much more complicated. For this reason, the simpler alternative proposed by Aguilera, Delporte-Gallet et al. [2000] is preferred.*

4.2.3 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, most of them avoid contamination. The algorithms either (1) prevent all processes from reaching an inconsistent state, or (2) prevent processes with an inconsistent state from sending messages to other processes.

4.3 Other Ordering Properties

The Total Order property (see Sect. 3, p.10) restricts the order of message delivery based solely on the destinations, that is, the property is independent of the sender processes. The definition can be further restricted by two properties related to the senders, namely, *FIFO Order* and *Causal Order*.

4.3.1 FIFO Order

Total Order alone does not guarantee that messages are delivered in the order in which they are sent (i.e., in first-in/first-out order). Yet, this property is sometimes required by applications in addition to Total Order. The property is called FIFO Order:

(FIFO ORDER) If a correct process TO-broadcasts a message m before it TO-broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.

4.3.2 Causal Order

The notion of causality in the context of distributed systems has been first formalized by [Lamport 1978b]. It is based on the relation "precedes"[11] (denoted by $\longrightarrow$) defined in his seminal paper and extended later in [Lamport 1986b]. The relation "precedes" is defined as follows.

Definition 1 *Let e_i and e_j be two events in a distributed system. The transitive relation $e_i \longrightarrow e_j$ holds if any one of the following three conditions is satisfied:*

1. *e_i and e_j are two events on the same process, and 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).*

This relation defines an irreflexive partial ordering on the set of events. The causality of messages can be defined by the "precede" relationship between their respective sending events. More precisely, a message m is said to precede a message m' (denoted $m \prec m'$) if the sending event of m precedes the sending event of m'.

The property of causal order for broadcast messages is defined as follows [Hadzilacos and Toueg 1994]:

(CAUSAL ORDER) If the broadcast of a message m causally precedes the broadcast of a message m', then no correct process delivers m' unless it has previously delivered m.

Hadzilacos and Toueg [Hadzilacos and Toueg 1994] also prove that the property of Causal Order is equivalent to combining the property of FIFO Order with the following property of Local Order.

(LOCAL ORDER) If a process broadcasts a message m and a process delivers m before broadcasting m', then no correct process delivers m' unless it has previously delivered m.

Note 5 (State-machine approach) *Causal total order broadcast is for instance required by the state machine approach [Lamport 1978a, Schneider 1990]. However, we think that some application may require causality, some others not.*

4.3.3 Source ordering

Some papers (e.g., [Garcia-Molina and Spauster 1991, Jia 1995]) make a distinction between single source and multiple source ordering. These papers define single source ordering algorithms as algorithms that ensure total order only if a *single* process broadcasts messages. This is a special case of FIFO broadcast, easily solved using sequence numbers. Source ordering is not particularly interesting in itself, and hence we do not discuss the issue further in this paper.

---

[11]Lamport initially called the relation "happened before" [Lamport 1978b], but he renamed it "precedes" in later work [Lamport 1986b, Lamport 1986a].

5 Properties of Destination Groups

So far, we have presented the problem of total order broadcast, wherein messages are sent to all processes in the system:

$$\forall m \in \mathcal{M} \; (Dest(m) = \Pi) \tag{3}$$

A *multicast* primitive is more general in the sense that it can send messages to any chosen subset of the processes in the system:

$$\exists m \in \mathcal{M} \, (sender(m) \notin Dest(m)) \wedge \exists m_i, m_j \in \mathcal{M} \, (Dest(m_i) \neq Dest(m_j)) \tag{4}$$

Although in wide use, the distinction between broadcast and multicast is not precise enough. This leads us to discuss a more relevant distinction, namely between closed versus open groups, and between single versus multiple groups.

5.1 Closed versus Open Groups

In the literature, many 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 parallel machines [Lamport 1978a] or highly available storage systems [Cristian et al. 1995]. 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, it is necessary for a process to be able to multicast messages to a group it does not belong to. Consequently, we consider it an important characteristic of algorithms to be easily adaptable to open interaction models.

5.1.1 Closed Group Algorithms

In closed groups algorithms, the sending process is always one of the destination processes:

$$\forall m \in \mathcal{M} \; (sender(m) \in Dest(m)) \tag{5}$$

So, these algorithms do not allow external processes (processes that are not member of the group) to multicast messages to the destination group.

5.1.2 Open Group Algorithms

Conversely, open group algorithms allow any arbitrary process in the system to multicast messages to a group, whether or not the sender process belongs to the destination group:

$$\exists m \in \mathcal{M} \; (sender(m) \notin Dest(m)) \tag{6}$$

Open group algorithms are more general than closed group algorithms: the former can be used with closed groups while the opposite is not true.

5.2 Single versus Multiple Groups

Most algorithms present in the literature assume that all messages are multicast to one single group of destination processes. Nevertheless, a few algorithms are designed to support multiple groups. In this context, we consider three situations: *single group*, *multiple disjoint groups*, *multiple overlapping groups*. We also discuss how useless trivial solutions can be ruled out with the notion of *minimality*. Since the ability to multicast messages to multiple destination sets is critical for certain classes of applications, we regard this ability as an important characteristic of an algorithm.

5.2.1 Single group ordering

With single group ordering, all messages are multicast to one single group of destination processes. As mentioned above, this is the model considered by a vast majority of the algorithms that are studied in this paper. Single group ordering can be defined by the following property:[12]

$$\forall m_i, m_j \in \mathcal{M} \ (Dest(m_i) = Dest(m_j)) \tag{7}$$

5.2.2 Multiple groups ordering (disjoint)

In some applications, the restriction to one single destination group is not acceptable. For this reason, algorithms have been proposed that support multicasting messages to multiple groups. The simplest case occurs when the multiple groups are *disjoint* groups, which can be expressed as follows:

$$\forall m_i, m_j \in \mathcal{M} \ (Dest(m_i) \neq Dest(m_j) \Rightarrow Dest(m_i) \cap Dest(m_j) = \emptyset) \tag{8}$$

In fact, adapting algorithms designed for one single group to work in a system with multiple disjoint groups is almost trivial.

5.2.3 Multiple groups ordering (overlapping)

In case of multiple groups ordering, it can happen that groups overlap. This can be expressed by the following predicate:

$$\exists m_i, m_j \in \mathcal{M} \ (Dest(m_i) \neq Dest(m_j) \wedge Dest(m_i) \cap Dest(m_j) \neq \emptyset) \tag{9}$$

The real difficulty of designing total order multicast algorithms for multiple groups arises when the groups can overlap. This is easily understood when one considers the problem of ensuring total order at the intersection of groups. In this context, [Hadzilacos and Toueg 1994] give three different properties for total order in the presence of multiple groups: *Local Total Order*, *Pairwise Total Order*, and *Global Total Order*.[13]

(LOCAL TOTAL ORDER) If correct processes p and q both TO-deliver messages m and m' and $Dest(m) = Dest(m')$, then p TO-delivers m before m' if and only if q TO-delivers m before m'.

Local Total Order is the weakest of the three properties. It requires that total order be enforced only for messages that are multicast within the same group.

Note also that multiple unrelated groups can be considered as disjoint groups even if they overlap. Indeed, destination processes belonging to the intersection of two groups can be seen as having two distinct identities; one for each group. It follows that an algorithm for distinct multiple groups can be trivially adapted to support overlapping groups with Local Total Order.

As pointed out by [Hadzilacos and Toueg 1994], the total order multicast primitive of the first version of Isis [Birman and Joseph 1987] guaranteed Local Total Order.[14]

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

Pairwise Total Order is strictly stronger than Local Total Order. Most notably, it requires that total order be enforced for all messages delivered at the intersection of two groups.

As far as we know, there is no straightforward algorithm to transform a total order multicast algorithm that enforces Local Total Order into one that also guarantees Pairwise Total Order (except for trivial solutions; see Sect. 5.2.4). [Hadzilacos and Toueg 1994] observe that, for instance, Pairwise Total Order is the order property guaranteed by the algorithm of [Garcia-Molina and Spauster 1989, Garcia-Molina and Spauster 1991].

---

[12]This definition and the following ones are static. They do not take into account the fact that processes can join groups and leave groups. Nevertheless, we prefer these simple static definitions, rather than more complex ones that would take dynamic destination groups into account.

[13]The ordering properties cited here are subject to contamination, see Section 4.2. Contamination can be avoided by formulating these properties similarly to the Gap-free Uniform Total Order property.

[14]It should be noted that, if the transformation is trivial from a conceptual point-of-view, the implementation was certainly a totally different matter, especially in the mid-80's.

Pairwise Total Order alone may lead to unexpected situations when there are three or more overlapping destination groups. For instance, [Fekete 1993] illustrates the problem with the following scenario. Consider three processes p_i, p_j, p_k, and three messages m_1, m_2, m_3 that are respectively sent to three different overlapping groups $G_1 = \{p_i, p_j\}$, $G_2 = \{p_j, p_k\}$, and $G_3 = \{p_k, p_i\}$. Pairwise Total Order allows the following histories on p_i, p_j, p_k:

$$p_i : \cdots TO\text{-}deliver(m_3) \longrightarrow \cdots \longrightarrow TO\text{-}deliver(m_1) \cdots$$
$$p_j : \cdots TO\text{-}deliver(m_1) \longrightarrow \cdots \longrightarrow TO\text{-}deliver(m_2) \cdots$$
$$p_k : \cdots TO\text{-}deliver(m_2) \longrightarrow \cdots \longrightarrow TO\text{-}deliver(m_3) \cdots$$

This situation is prevented by the specification of Global Total Order [Hadzilacos and Toueg 1994], which is defined as follows:

(GLOBAL TOTAL ORDER) The relation $<$ is acyclic, where $<$ is defined as follows: $m < m'$ if and only if any correct process delivers m and m', in that order.

Note 6 *[Fekete 1993] gives another specification for total order multicast which also prevents the scenario mentioned above. The specification, called AMC, is expressed as an I/O automaton [Lynch and Tuttle 1989, Lynch 1996] and uses the notion of pseudo-time to impose an order on the delivery of messages.*

5.2.4 Minimality and trivial solutions

Any algorithm that solves the problem of total order broadcast in a single group can easily be adapted to solve the problem for multiple groups with the following approach:

1. form a super-group with the union of all destination groups;

2. whenever a message m is multicast to a group, multicast it to the super-group, and

3. processes not in $Dest(m)$ discard m.

The problem of such a solution is its lack of scalability. Indeed, in very large distributed systems, even if destination groups are individually small, their union is likely to cover a very large number of processes.

To avoid this sort of solution, [Guerraoui and Schiper 2001] require the implementation of total order multicast for multiple groups to satisfy the following minimality property:

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

This property is often too strong: it disallows a lot of interesting algorithms that use a small number of external processes for message ordering (e.g., algorithms which disseminate messages along some propagation tree). A weaker property would allow an algorithm to involve a small set of external processes.

5.2.5 Transformation algorithm

[Delporte-Gallet and Fauconnier 2000] propose a generic algorithm that transforms a total order broadcast algorithm for a single closed group into one for multiple groups. The algorithm splits destination groups into smaller entities and supports multiple groups with Strong Minimality.

6 Other Specifications for Total Order Broadcast

The specification in Section 3 is the standard specification of total order broadcast in a *static* system, that is, a system in which all processes are created at system initialization. broadcast. In this section, we briefly discuss other specifications of total order broadcast, namely the case of dynamic groups, partitionable systems and Byzantine failures.

6.1 Dynamic Groups and Partitionable Systems

A *dynamic* group is a group of processes with a membership that can change during the computation: processes can be added to a group and removed from the group (e.g., due to failures). This requires to adapt the specification of total order broadcast.

In the case of a dynamic group, the successive memberships of the group are called the *views* of the group [Chockler, Keidar, and Vitenberg [2001]]. Views are defined by the *group membership* problem, which has two variants: (1) the *primary partition* membership problem, and (2) the *partitionable* membership problem. In the primary partition group membership, one of the partitions is recognized as primary, and processes are allowed to deliver messages only if they belong to the primary partition. In contrast, the partitionable group membership allows all processes to deliver messages, regardless of the partition they belong to.

With dynamic groups, the basic communication abstraction is called *view synchrony*, which can be seen as the counterpart of reliable broadcast in static systems. Reliable broadcast is defined by the Validity, Agreement and Integrity properties of Sect. 3. Roughly speaking, View Synchrony adopts a similar definition while relaxing the Agreement property.[15]

Total order broadcast in a system with dynamic groups can hence be specified as view synchrony plus an additional order property. Chockler, Keidar, and Vitenberg [2001] define three order properties in a partitionable system: Strong Total Order (messages are delivered in the same order by all processes that deliver them), Weak Total Order (the order requirement is restricted within a view), and Reliable Total Order (extends the Strong Total Order property to require processes to deliver a prefix of a common sequence of messages within each view). In other words, Strong Total Order corresponds somehow to the Uniform Total Order property of Sect. 3, and Reliable Total Order somehow to the Prefix Ordering property of Sect. 4.2. Other properties, such as Validity, are also defined differently in partitionable systems. This is explained is considerably more detail by Chockler, Keidar, and Vitenberg [2001] and [Fekete et al. 2001].

6.2 Byzantine Failures

Tolerating Byzantine failures has several important implications on the specification of the problem, in particular on uniformity and contamination.

Uniformity   Algorithms tolerant to Byzantine failures can guarantee none of the uniform properties given in Sect. 3. 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 in the wrong order (violates Total Order).

[Reiter 1994] proposes a more useful definition of uniformity for Byzantine systems. He distinguishes between crash and Byzantine failures. He says that a process is *honest* if it behaves according to its specification, and *corrupt* otherwise (i.e., Byzantine), where honest processes can also fail by crashing. In this context, uniform properties are those which are enforced by all honest processes, regardless whether they are correct or not. This definition is more sensible that the stricter definition of Sect. 3, as nothing is required from corrupt processes.

Contamination   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 broadcasting a bogus message that seems correct to every other process [Hadzilacos and Toueg 1994].

7 Mechanisms for Message Ordering

In this section, we propose a classification of total order broadcast 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 necessary for defining the order of messages (e.g., timestamp or sequence number).

We identify three different roles that a participating process can take with respect to the algorithm: sender, destination, or sequencer. A *sender* process is a process p_s from which a message originates (i.e., $p_s \in \Pi_{sender}$).

---

[15]Discussing this primitive in detail is beyond the scope of this survey (see paper by Chockler, Keidar, and Vitenberg [2001] for details).
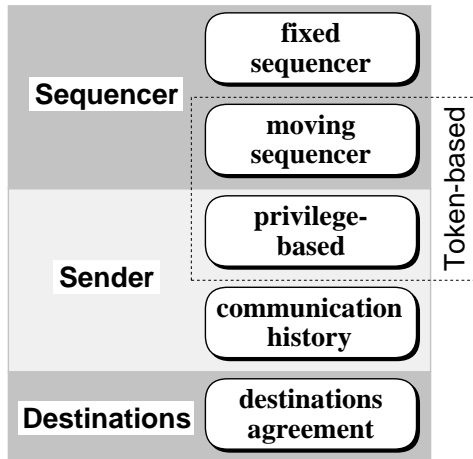
Figure 3: Classes of total order broadcast algorithms.

A *destination* process is a process p_d to which a message is destined (i.e., $p_d \in \Pi_{dest}$). Finally, a *sequencer* process is not necessarily a sender or a destination, but is somehow involved in the ordering of messages. A given process may simultaneously take several roles (e.g., sender *and* sequencer *and* destination). However, we represent these roles separately as they are conceptually different.

According to the three different roles mentioned above, we define three basic classes for total order broadcast algorithms, depending whether the order is respectively built by a sequencer, the sender, or destination processes. Among algorithms of the same class, significant differences remain. To account for this problem, we introduce a further division, leading to five subclasses in total. These classes are named as follows (see Fig. 3): *fixed sequencer*, *moving sequencer*, *privilege-based*, *communication history*, and *destinations agreement*. Privilege-based and moving sequencer algorithms are commonly referred to as token-based algorithms.

The terminology defined in this paper is partly borrowed from other authors. For instance, "communication history" and "fixed sequencer" was proposed by [Cristian and Mishra 1995]. The term "privilege-based" was suggested by Malkhi. Finally, [Le Lann and Bres 1991] group algorithms into three classes based on where the order is built. Unfortunately, their definition of classes is specific to a client-server architecture.

In the remainder of this section, we present each of the five classes, and illustrate each class with a simple algorithm. The algorithms are merely presented for the purpose of illustrating the corresponding category, and should not be regarded as full-fledged working examples. Although inspired from existing algorithms, they are largely simplified. Besides, none of these algorithms are fault-tolerant.

Note 7 (Atomic blocks) *The algorithms are written in pseudocode, with the assumption that blocks associated with a when-clause are executed atomically. This assumption simplifies the algorithms with respect to concurrency.*

7.1 Fixed Sequencer

In a fixed sequencer algorithm, one process is elected as the sequencer and is responsible for ordering messages. The sequencer is unique, and the responsibility is not normally transfered to another processes (at least in the absence of failure).

The approach is illustrated in Fig. 4 and Fig. 5. One specific process takes the role of a sequencer and builds the total order. To broadcast 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 latter then deliver messages according to the sequence numbers. This algorithm does not tolerate the failure of the sequencer.

In fact, three variants of fixed sequencer algorithms exist. We call these three variants "UB" (unicast-broadcast), "BB" (broadcast-broadcast), and "UUB" (unicast-unicast-broadcast), taking inspiration from [Kaashoek and Tanenbaum

In the first variant, called "UB" (see Fig. 6(a)), the protocol consists of a unicast to the sequencer, followed by a broadcast from the sequencer. This variant generates few messages, and it is the simplest of the three approaches. It is, for instance, adopted by [Navaratnam et al. 1988], and corresponds to the algorithm in Fig. 5.
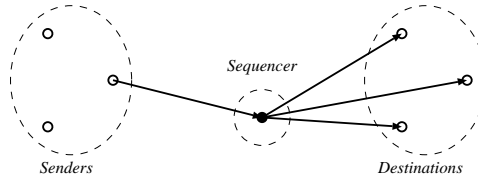
Figure 4: Fixed sequencer algorithms.

Sender:

    procedure *TO-broadcast*(m) { To *TO-broadcast* a message m }

        send (m) to sequencer

Sequencer:

    Initialization:

        $seqnum := 1$

    when receive (m)

        $sn(m) := seqnum$
        send $(m, sn(m))$ to all
        $seqnum := seqnum + 1$

Destinations (code of process p_i):

    Initialization:

        $nextdeliver_{p_i} := 1$
        $pending_{p_i} := \emptyset$

    when receive $(m, seqnum)$

        $pending_{p_i} := pending_{p_i} \cup \{(m, seqnum)\}$
        while $\exists (m', seqnum') \in pending_{p_i} : seqnum' = nextdeliver_{p_i}$ **do**
            deliver (m')
            $nextdeliver_{p_i} := nextdeliver_{p_i} + 1$

Figure 5: Simple fixed sequencer algorithm.
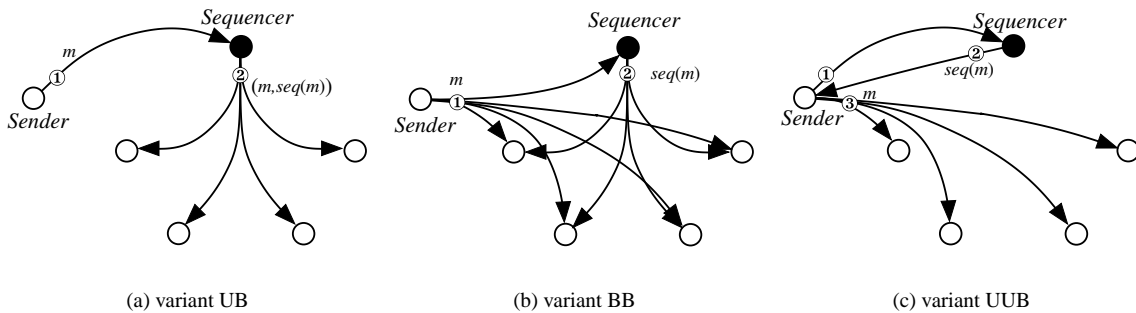
Figure 6: Common variants of fixed sequencer algorithms.

In the second variant, called "BB" (Fig. 6(b)), the protocol consists of a broadcast to all destinations plus the sequencer, followed by a second broadcast from the sequencer. This generates more messages than the previous approach, except in broadcast networks. However, it can reduce the load on the sequencer, and makes it easier to tolerate the crash of the sequencer. Isis (sequencer) [Birman et al. 1991] is an example of the second variant.

The third variant, called "UUB" (Fig. 6(c)), is less common than the others. In short, the protocol consists of the following steps. The sender requests a sequence number from the sequencer (unicast). The sequencer

replies with a sequence number (unicast). Then, the sender broadcasts the sequenced message to the destination processes.[16]

7.2 Moving Sequencer

Moving sequencer algorithms are based on the same principle as fixed sequencer algorithms, but allow the role of sequencer to be transferred between several processes. The motivation is to distribute the load among them. This is illustrated in Figure 7 where the sequencer is chosen among several processes. The code executed by each process is however more complex than with a fixed sequencer, which explains the popularity of the latter approach. Notice that, with moving sequencer algorithms, the roles of sequencer and destination processes are normally combined.

The algorithm of Figure 8 shows the principle of moving sequencer algorithms. To broadcast 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., all 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.

Note 8 *Similar to fixed sequencer algorithms, it is possible to develop a moving sequencer algorithm according to one of three variants. However, the difference between the variants is not as clearcut as it is for a fixed sequencer. It turns out that among the moving sequencer algorithms surveyed, all of them follow the equivalent of the variant BB of fixed sequencer. Hence we do not discuss this issue any further.*

Note 9 *As mentioned, the main motivation for using a moving sequencer is to distribute the load among several processes, thus avoiding the bottleneck caused by a single process. This is illustrated by several studies (e.g., [Cristian et al. 1994, Urbán et al. 2000]). One could then wonder when a fixed sequencer algorithm should be preferred to a moving sequencer algorithm. There are, in fact, at least three possible reasons. First, fixed sequencer algorithms are considerably simpler, leaving less room for implementation errors. Second, the latency of fixed sequencer algorithms is often better, as shown by [Urbán et al. 2000]. Third, it is often the case that some of the machines are more reliable, more trusted, better connected, or simply faster than others. When this is the case, it makes sense to use one of them as a fixed sequencer (see MTP in §9.1.2).*

7.3 Privilege-Based

Privilege-based algorithms rely on the idea that senders can broadcast messages only when they are granted the privilege to do so. Figure 9 illustrates this class of algorithms. The order is defined by the senders when they broadcast their messages. The privilege to broadcast (and order) messages is granted to only one process at a time, but this privilege circulates from process to process among the senders. In other words, due to the arbitration between senders, building the total order requires to solve the problem of FIFO broadcast (easily solved with sequence numbers at the sender), and to ensure that passing the privilege to the next sender does not violate this order.

The algorithm of Figure 10 illustrates the principle of privilege-based algorithms. Senders circulate a token message that carries a sequence number for the next message to broadcast. When a process wants to broadcast 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. Following this, the sender updates the token and sends it to the next sender. Destination processes deliver messages in increasing sequence numbers.

Note 10 *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, in which there is no fixed and previously known set of senders.*

Note 11 *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 processes can send messages at the same time. By ensuring that the communication medium is accessed in mutual exclusion, the total order is easily guaranteed. The technique is also known as* time division multiple access *(TDMA).*

---

[16]The protocol to tolerate failures is complex.
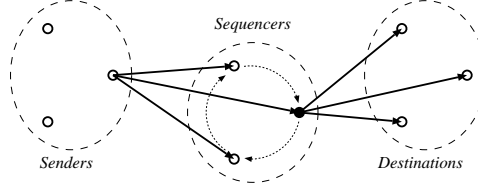
Figure 7: Moving sequencer algorithms.

Sender:

    procedure *TO-broadcast*(m) { *To TO-broadcast a message m* }
        send (m) to all sequencers

Sequencers (code of process s_i):

    Initialization:
        $received_{s_i} := \emptyset$
        if $s_i = s_1$ **then**
            $token.seqnum := 1$
            $token.sequenced := \emptyset$
            send *token* to s_1
    when receive m
        $received_{s_i} := received_{s_i} \cup \{m\}$
    when receive *token* from s_{i-1}
        for each m' in $received_{s_i} \setminus token.sequenced$ **do**
            send $(m', token.seqnum)$ to destinations
            $token.seqnum := token.seqnum + 1$
            $token.sequenced := token.sequenced \cup \{m'\}$
        send *token* to $s_{i+1 \ (\mathrm{mod}\ n)}$

Destinations (code of process p_i):

    Initialization:
        $nextdeliver_{p_i} := 1$
        $pending_{p_i} := \emptyset$
    when receive $(m, seqnum)$
        $pending_{p_i} := pending_{p_i} \cup \{(m, seqnum)\}$
        while $\exists (m', seqnum') \in pending_{p_i}$ s.t. $seqnum' = nextdeliver_{p_i}$ **do**
            deliver (m')
            $nextdeliver_{p_i} := nextdeliver_{p_i} + 1$

Figure 8: Simple moving sequencer algorithm.

Note 12 *It is tempting to consider that privilege-based and moving sequencer algorithms are equivalent, since both rely on a token passing mechanism. However, they differ in one significant aspect: the total order is built by senders in privilege-based algorithms, whereas it is built 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 whereas, with moving sequencer algorithms, it is mostly used for improving performance, e.g., by doing load balancing.*

Note 13 *With privilege-based algorithms, it is difficult to ensure fairness. Indeed, if a process has a very large number of messages to broadcast, it could keep the token for an arbitrary long time, thus prevented other processes from broadcasting their own messages. To overcome this problem, algorithms often enforce an upper limit on the number of messages and/or the time that some process can keep the token. Once the limit is passed, the process is compelled to release the token, regardless of the number of messages that remain to be broadcast.*
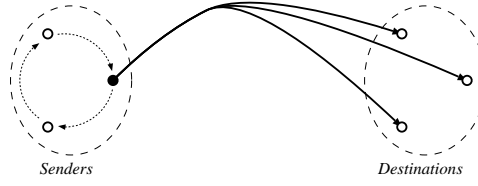
Figure 9: privilege-based algorithms.

Senders (code of process s_i):

    Initialization:
        $tosend_{s_i} := \emptyset$
        if $s_i = s_1$ **then**
            $token.seqnum := 1$
            send $token$ to s_1

    procedure *TO-broadcast*(m)　　　　　　　　　　　　　　　　{ *To TO-broadcast a message m* }
        $tosend_{s_i} := tosend_{s_i} \cup \{m\}$

    when receive $token$
        for each m' in $tosend_{s_i}$ **do**
            send $(m', token.seqnum)$ to destinations
            $token.seqnum := token.seqnum + 1$
        $tosend_{s_i} := \emptyset$
        send $token$ to $s_{i+1 \pmod n}$

Destinations (code of process p_i):

    Initialization:
        $nextdeliver_{p_i} := 1$
        $pending_{p_i} := \emptyset$

    when receive $(m, seqnum)$
        $pending_{p_i} := pending_{p_i} \cup \{(m, seqnum)\}$
        while $\exists (m', seqnum') \in pending_{p_i}$ s.t. $seqnum' = nextdeliver_{p_i}$ **do**
            deliver (m')
            $nextdeliver_{p_i} := nextdeliver_{p_i} + 1$

Figure 10: Simple privilege-based algorithm.

7.4 Communication History

Similarly to privilege-based algorithms, the delivery order is determined by the senders in communication history algorithms. However, in contrast to privilege-based algorithms, processes can broadcast messages at any time, and total order is ensured by delaying the delivery of messages. The messages usually carry a (physical or logical) timestamp. The destinations observe the messages generated by the other processes and their timestamps, i.e., the history of communication in the system, to learn when delivering a message will no longer violate the total order.

There are two fundamentally different variants of communication history algorithms. In the first variant, called *causal history*, communication history algorithms use a partial order defined by the causal history of messages and transform this partial order into a total order. Concurrent messages are ordered according to some predetermined function. In the second variant, known as *deterministic merge*, processes send messages timestamped independently (thus not reflecting causal order) and delivery takes place according to a deterministic policy of merging the streams of messages coming from each process. A simple example policy consists in delivering the next message from p_1, then the next one from p_2, etc., iterating over all processes in a round robin fashion.

Figure 11 illustrates a typical communication history algorithm of the first variant. The algorithm, inspired by [Lamport 1978b], works as follows. The algorithm uses logical clocks [Lamport 1978b] to "timestamp" each message m with the logical time of the *TO-broadcast*(m) event, denoted $ts(m)$. Messages are then delivered in the

Senders and destinations (code of process p; assumes FIFO channels):

Initialization:

$received_p := \emptyset$ { *Messages received by process p* }

$delivered_p := \emptyset$ { *Messages delivered by process p* }

$LC_p[p_1 \ldots p_n] := \{0, \ldots, 0\}$ { *$LC_p[q]$: logical clock of process q, as seen by process p* }

procedure *TO-multicast(m)*   { *To TO-multicast a message m* }

$LC_p[p] := LC_p[p] + 1$

$ts(m) := LC_p[p]$

send FIFO $(m, ts(m))$ to all

when receive $(m, ts(m))$

$LC_p[p] := \max(ts(m), LC_p[p]) + 1$

$LC_p[sender(m)] := ts(m)$

$received_p := received_p \cup \{m\}$

$deliverable := \emptyset$

for each message m' in $received_p \setminus delivered_p$ **do**

     if $ts(m') \leq \min_{q \in \Pi} LC_p[q]$ **then**

         $deliverable := deliverable \cup \{m'\}$

deliver all messages in $deliverable$, in increasing order of $(ts(m), sender(m))$

$delivered_p := delivered_p \cup deliverable$

Figure 11: Simple communication history algorithm.

order of their timestamps. However, we can have two messages m and m' with the same timestamp. To arbitrate between these messages, the algorithm uses the lexicographical order on the identifiers of sending processes. In Figure 11, we refer to this order as the $(ts(m), sender(m))$ order, where $sender(m)$ is the identifier of the sender process.

Note 14 *The algorithm of Figure 11 is not live. Indeed, consider a scenario where a single process p broadcasts a single message m, and no other process ever broadcasts any message. According to the algorithm in Figure 11, 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 at least one of the processes never broadcasts any message. To overcome this problem, communication history algorithms proposed in the literature usually send empty messages when no application messages are broadcast.*

Note 15 *In synchronous systems, communication history algorithms rely on synchronized clocks, and use physical timestamps instead of logical ones. The nature of such systems makes it unnecessary to send empty messages in order to ensure liveness. Indeed, this can be seen as an example of the use of time to communicate [Lamport 1984].*

7.5 Destinations Agreement

In destinations agreement algorithms, as the name indicates, the delivery order results from an agreement between destination processes (see Figure 12). We distinguish three different variants of agreement: (1) agreement on a message sequence number, (2) agreement on a message set, or (3) agreement on the acceptance of a proposed message order.

Figure 13 illustrates an algorithm of the first variant: for each message, the destination processes reach an agreement on a unique (yet not consecutive) sequence number. The algorithm is adapted from Skeen's algorithm (§9.5.1), albeit it operates in a decentralized manner. Briefly, the algorithm works as follows. To broadcast a message m, a sender sends m to all destinations. Upon receiving m, a destination assigns it a local timestamp and sends this timestamp to all destinations. Once a destination process has received a local timestamp for m from all destinations, a unique global timestamp $sn(m)$ is assigned to m, calculated as the maximum of all local timestamps. Messages are delivered in the order of their global timestamp, that is, a message m can only be delivered once it has been assigned its global timestamp $sn(m)$ and no other undelivered message m' can possibly receive a timestamp $sn(m')$ smaller or equal to $sn(m)$. As with the communication history algorithm (Figure 11), the identifier of the message sender is used to break ties between messages with the same global timestamp.
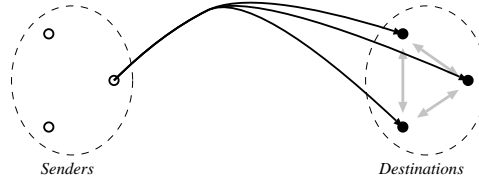
Figure 12: Destinations agreement algorithms.

Sender:

    procedure *TO-broadcast*(m)   { *To TO-broadcast a message* m }

        send (m) to destinations

Destinations (code of process p_i):

    Initialization:

        $stamped_{p_i} := \emptyset$

        $received_{p_i} := \emptyset$

        $LC_{p_i} := 0$ { LC_{p_i}: *logical clock of process* p_i }

    when receive m

        $ts_i(m) := LC_{p_i}$

        $received_{p_i} := received_{p_i} \cup \{(m, ts_i(m))\}$

        send ($m, ts_i(m)$) to destinations

        $LC_{p_i} := LC_{p_i} + 1$

    when received ($m, ts_j(m)$) from p_j

        $LC_{p_i} := \max(ts_j, LC_{p_i} + 1)$

        if received ($m, ts(m)$) from all destinations **then**

            $sn(m) := \max\limits_{k=1\cdots n} ts_k(m)$

            $stamped_{p_i} := stamped_{p_i} \cup \{(m, sn(m))\}$

            $received_{p_i} := received_{p_i} \setminus \{m\}$

            $deliverable := \emptyset$

            for each $(m', sn(m')) \in stamped_{p_i}$ s.t. $\forall m'' \in received_{p_i} : sn(m') < ts_i(m'')$ **do**

                $deliverable := deliverable \cup \{(m', sn(m'))\}$

            deliver all messages in $deliverable$ in increasing order of $(sn(m), sender(m))$

            $stamped_{p_i} := stamped_{p_i} \setminus deliverable$

Figure 13: Simple destinations agreement algorithm.

The most representative algorithm of the second variant of agreement is the algorithm proposed by [Chandra and Toueg 1996] (§9.5.4). The algorithm transforms total order broadcast into a sequence of consensus problems. Each consensus allows the processes to agree on a set of messages, i.e., consensus number k allows the processes to agree on a set Msg^k. For $k < k'$, the messages in Msg^k are delivered before the messages in $Msg^{k'}$. The messages in a set Msg^k are delivered according to some predetermined order (e.g., in the order of their identifiers).

With the third variant of agreement, a tentative message delivery order is first proposed (usually by one of the destinations). Then, the destination processes must agree either to accept or to reject the proposal. In other words, this variant of destinations agreement relies on an atomic commitment protocol.

Note 16 *The line is thin between the second and the third variant of agreement. For instance, Chandra and Toueg's total order broadcast algorithm relies on consensus, as described above. However, when combined with the rotating coordinator consensus algorithm using $\Diamond\mathcal{S}$, the resulting algorithm can be seen as an algorithm of the third form. Indeed, the coordinator proposes a tentative order (given as a set of message plus message identifiers) that it tries to validate. Thus it is important to note that two seemingly identical algorithms may use different forms of agreement, simply because they are described at different levels of abstraction.*

7.6 Time-free versus time-based ordering

We introduce a further distinction between algorithms, orthogonal to the above classification. The distinction is between algorithms that use the physical time for message ordering, and algorithms that do not use the physical time. For instance, in Sect. 7.4 (see Fig. 11) we have presented a simple communication-history algorithm based on *logical* time. It is indeed possible to design a similar algorithm that uses the *physical* time instead (and synchronized clocks).

In short, we distinguish algorithms with *time-based ordering* that rely on physical time, and algorithms with *time-free ordering* that do not use the physical time.

8 Mechanisms for Fault-Tolerance

The total order broadcast algorithms described in Section 7 are not tolerant to failures: if a single process crashes, the properties specified in Section 3 are not satisfied. To be fault-tolerant, total order algorithms rely on various techniques. In this section we present the most important of these techniques. Note that it is somehow difficult to discuss these techniques without getting into specific implementation details. Nevertheless, we try to keep the discussion as general as possible.

8.1 Failure detection

A recurrent pattern in all distributed algorithms is for a process p to wait for a message from some other process q. If q has crashed, process p is blocked. Failure detection is one basic mechanism to prevent p from being blocked.

Unreliable failure detection has been formalized by [Chandra and Toueg 1996] in terms of two properties: *accuracy* and *completeness* (see Sect. 2.3.2). Completeness is related to the blocking problem mentioned above. The role of accuracy is more difficult to summarize. Roughly speaking, accuracy prevents algorithms from running forever, without solving the problem (livelock).

Unreliable failure detectors might be too weak for some total order broadcast algorithms, which require *reliable* failure detection information, provided by a *perfect* failure detector, known as $\mathcal{P}$ (see Sect. 2.6).

8.2 Group Membership Service

The low-level failure detection mechanism is not the only way to address the blocking problem mentioned in the previous section. Blocking can also be prevented by relying on a higher level mechanism, namely a *group membership service*.

A group membership service is a distributed service that is responsible for managing the membership of groups of processes (see Sect. 6.1 and paper by Chockler, Keidar, and Vitenberg [2001]). The successive memberships of a group are called the *views* of the group. Whenever the membership changes, the service report changes to all group members, by providing them with the new view.

A group membership service usually provides strong completeness: if a process p member of some group crashes, the membership services provides to the surviving members a new view from which p is excluded. In the primary-partition model (see Sect. 6.1), the accuracy of failure notifications is ensured by forcing the crash of processes that have been incorrectly suspected and excluded from the membership, a mechanism called *process-controlled crash* (see Sect. 2.6). Moreover in the primary-partition model, the group membership service provides consistent notifications to the group members: the successive views of a group are notified in the *same order* to all its members.

To summarize, while failure detectors provide unreliable and inconsistent failure notifications, a group membership service provides consistent failure notifications. Moreover, total order algorithms that rely on a group membership service for fault tolerance, exploit another property that is usually provided together with the membership service, namely *view synchrony* (see Sect. 6.1). Roughly speaking, view synchrony ensures that between two successive views v and v', processes in the two views deliver the same set of messages. Group membership service and view synchrony have been used for implementing complex group communication systems (e.g., Isis [Birman and van Renesse 1993], Totem [Moser et al. 1996], Transis [[Dolev and Malkhi 1994];[1996] ; [Amir et al. 1992]], Phoenix [Malloth et al. 1995, Malloth 1996]).

8.3 Resilient communication patterns

As shown in the previous sections, an algorithm can rely on a failure detection mechanism or on a group membership service to avoid the blocking problem. To be fault-tolerant, another solution is to avoid any potential blocking pattern.

Consider for example a process p waiting for $n - f$ messages, where n is the number of processes in the system, and f the maximum number of processes that may crash. If all correct processes send a message to p, than the above pattern is non-blocking (and does not require any failure detector mechanism or group membership service). We call this pattern a *resilient* pattern.

Note that, to be fault-tolerant, a total order broadcast algorithm can use more than one of the mechanisms mentioned here, e.g., failure detection and resilient patterns.

8.4 Message stability

Avoiding blocking is not the only problem that fault-tolerant total order broadcasts algorithms have to address. Figure 1 (page 11) illustrates a violation of the Uniform Agreement property. The problem here is not related to blocking.

The mechanism that solves the problem is called *message stability*. A message m is said to be *k-stable* if m has been received by k processes. In a system in which at most f processes may crash, $f+1$-stability is the important property to detect: if some message m is $f+1$-stable, then m is received by at least one correct process. With such a guarantee an algorithm can easily ensure that m is eventually received by all correct processes. $f+1$-stability is often simply called *stability*. The detection of stability is generally based on some acknowledgment scheme or token passing.

8.5 Consensus

The mechanisms described so far are low-level mechanisms on which fault-tolerant total broadcast algorithms may rely.

Another option for a fault-tolerant total order broadcast algorithm is to rely on higher level mechanisms that solve all the problems related to fault tolerance (i.e., the problems mentioned above). The consensus problem (see Sect. 2.4.3) is such a mechanism. Some algorithms solve total order broadcast by a transformation into a consensus problems. This way, fault tolerance, including failure detection and message stability detection, is completely hidden within the consensus abstraction.

8.6 Mechanisms for lossy channels

Apart from the mechanisms used to tolerate process crashes, we need to say a few words about mechanisms to tolerate channel failures. First, it should be mentioned that several total order broadcast algorithms assume an underlying layer that takes care of lossy channels: these algorithms assume reliable channels, i.e., message loss is not discussed. Some other algorithms are build directly on top of lossy channels, and so address message loss explicitly.

To address message loss, the standard solution is to rely on a positive or a negative acknowledgment mechanism. With positive acknowledgment, the reception of messages is acknowledged; with negative acknowledgment, the detection of a missing message is signaled. The two schemes can be combined.

Token-based algorithms (i.e., moving sequencer or privilege-based algorithms) rely on the token passing to detect message losses: the token can be used to convey acknowledgments, or to detect missing messages. So token-based algorithms use the token for ordering purpose, but also for implementing reliable channels.

9 Survey of Existing Algorithms

This section provides an extensive survey of total order broadcast algorithms. We present about sixty algorithms published in scientific journals or conference proceedings over the past three decades. We have done every possible efforts to be exhaustive, and we are quite confident that this paper dresses a good picture of the field at the time of writing. However, because of the continuous flow of papers on the subject, we might have overlooked one algorithm or two.

○	*yes.*	
△	*somewhat.*	explained in the text.
×	*no.*	
spec.	*special.*	explained in the text.
inf.	*informal.*	explained in the text.
NS	*not specified.*	means also "not discussed."
n/a	*not applicable.*	
+a	*positive acknowledgment.*	
-a	*negative acknowledgment.*	
GM	*group membership.*	
FD	*failure detector/detection.*	
Cons.	*consensus.*	
RCP	*resilient communication patterns.*	
ByzA.	*Byzantine agreement.*	

Table 2: Abbreviations used in Tables 3–5.

In Tables 3–5, we present a synthetic overview of all surveyed algorithms, where we summarize the important characteristics of each algorithm. The tables present only factual information about the algorithms, as it appears in the relevant papers. In particular, the tables do not present information that is the result of extrapolation, or non-obvious deduction. The exception is when we had to interpret information to overcome differences in terminology. Also, properties that are discussed in the original paper, yet not proved correct, are reported as "informal" in the tables. For the sake of conciseness, several symbols and abbreviations have been used throughout the tables; they are explained in Table 2. For each algorithm, Tables 3–5, provide the following information:

- *General information*, i.e., the ordering mechanism (see Sect. 7), and whether the mechanism is time-based or not (Sect. 7.6).

- The *General information* rows are followed by rows describing the assumptions that the algorithm is based on, i.e., what is *provided* to it:

  - The *System model* rows specify the synchrony assumptions, the assumptions made about process failures and communication channels. The row *partitionable* shows if the algorithm works in a system with dynamic groups and partitionable membership semantics (see Sect. 6.1). In particular, algorithms in which only processes in a primary partition can work are not considered partitionable.

  - The rows called *Condition for liveness* discuss the assumptions necessary to ensure the liveness of the algorithm:

    1. The row *live...X* means that the liveness of the algorithm requires the liveness of the building block X (on which the algorithm relies). For example, *live... consensus* means that the algorithm is live if the consensus building block on which the algorithm relies is itself live.

    2. The row *other* adds the following information: *NS = not specified* means that liveness is not discussed in the paper; *n/a = not applicable* means that no additional assumption is needed to ensure liveness (this applies mostly to algorithms that assume a synchronous model); △ = *somewhat* and *spec. = special* refers to a discussion of liveness below in the paper; *recovery* means that the algorithm is blocking, i.e., liveness requires the recovery of crashed processes; $\Diamond\mathcal{P}$ refers to the failure detector needed to ensure liveness.

  - The next group of rows indicate the *building block(s)* used by the algorithm. The building blocks considered are: *view synchrony* (Sect. 8.2), which encompasses a *group membership* service; *reliable broadcast* (Sect. 2.4.1); *causal broadcast* (Sect. 4.3.2); *consensus* (Sect. 2.4.3); or *other*. *Other* can be either TDMA = *time division multiple access* (Note 11, Sect. 7.3), ByzA. = *Byzantine agreement* (Sect. 2.4.2), or spec. = *special*, which means that the explanation is in the text below.

- After discussing what is provided *"to"* the algorithms, we discuss what is provided *"by"* the algorithms.

– The first rows give the *Properties ensured* by the algorithms. As discussed in Section 3, total order broadcast is specified by the following properties: Validity, Uniform Agreement, Uniform Integrity, Uniform Total Order. Validity and Uniform Integrity do not appear in the table. The reason is that these properties are rarely discussed in the papers (authors usually assume they are trivially ensured).

We first discuss Agreement and Uniform Agreement, then Total Order and Uniform Total Order. Finally, we mention whether the algorithm additionally ensures FIFO order or causal order. In all these entries, one would might expect either a *yes* or a *no*. Unfortunately many papers do not provide proofs (often only informal arguments), which means that these properties can be questioned. In this case, *inf. = informal* appears in the table. If an algorithm does not discuss the properties of total order broadcast at all, the corresponding entry mentions *NS = not specified*. If the non-uniform property is only discussed informally, then the corresponding entry for the uniform property is left empty (in an informal discussion, the distinction between the uniform and the non-uniform property usually does not appear). $\bigcirc/\times$ (=*yes/no*) appears in some entries for the uniform property, meaning that these algorithms provide several levels of Quality of Service (QoS), which include a uniform and a non-uniform version of the algorithm, where the non-uniform version is more efficient. Moreover, for being able to compare non-partitionable algorithms with partitionable algorithms, we consider the properties enforced by the former when executed in a non-partitionable system model.

For the rows *FIFO order* and *causal order*, $\bigcirc$ = *yes* appears only if this characteristic is explicit in the paper. Otherwise the entry is simply left blank. Finally, if an algorithm is not fault-tolerant, then the distinction between the uniform and the non-uniform properties does not make sense. In this case the entry mentions *n/a = not applicable*.

– The rows called *destination groups* tell whether the algorithm supports the total order broadcast of a message to multiple groups (row *multiple*), and whether the algorithms support *open* groups (see Sect. 5). The entry is left blank if the issue is not discussed explicitly in the paper.

- The last group of rows, called *Fault-tolerant mechanisms*, discusses the mechanisms used to provide fault tolerance. The row *process* mentions the mechanisms used to tolerate process crashes (see Sect. 8). Note that some of these fault-tolerant mechanisms also appear as *building blocks*. However, not all building blocks have been reported as fault-tolerant mechanisms (e.g., reliable broadcast, causal broadcast).[17]

The row *comm.* mentions the mechanisms used to address message losses. Most of the algorithms assume underlying reliable channels, in which case the entry mentions *n/a = not applicable*. The acronyms +*a* and -*a* indicate a positive, respectively negative, acknowledgment mechanism. The other entries are *flood* (flooding), *special* (explanation in the text below), and GM = *group membership*. In the context of unreliable channels, the GM mechanism is used in the case were some process p waits for a message from some other process q: if no message is received (e.g., due to loss), then p requests the exclusion of q from the membership.

In Sections 9.1 through 9.6, we give a brief description of each individual algorithm, and complement the information provided in the tables. Unlike the tables, the textual descriptions also present information that we have deduced from the relevant papers. In some cases, the lack of technical details about the algorithms (in particular in case of failures) leads us to *extrapolate* their behavior. In this case, we avoid being too assertive (e.g., using conditional) and kindly recommend the reader to take this speculative information with appropriate circumspection.

We think that it is useful to stress again the respective roles of the tables and the accompanying text in Sections 9.1 to 9.6. The tables provide *factual* information about each algorithm, as it was published in the relevant papers. In contrast, the text provides complementary information, including information that we have *extrapolated*. In particular, the text explains the originality of each algorithm, and complements items that are left vague in the tables (i.e., those points are vague in the paper). In particular, for some of the algorithms, the properties reported in the tables are weaker than those the algorithm might ensure. In such a case, the text below mentions (and discusses) the stronger property that might hold. We insist on this point as misunderstanding the respective roles of text and tables might give the wrong impression that text and tables are in contradiction.

---

[17]The decision of what is a fault-tolerance mechanism and what is not is somehow arbitrary. We have decided to keep the number of mechanisms mentioned in Section 8 low, i.e., to mention only on key mechanisms.

Table 3: Overview of total order broadcast algorithms (part I).

algorithm	Amoeba §9.1.1	MTP §9.1.2	Tandem §9.1.3	Garcia-M. Spauster §9.1.4	Jia §9.1.5	Isis (seq.) §9.1.6	Navarat. et al. §9.1.7	Phoenix §9.1.8	Rampart §9.1.9	Chang Maxem. §9.2.1	RMP §9.2.2	DTP §9.2.3	Pin-wheel §9.2.4	On-demand §9.3.1	Train §9.3.2	Totem §9.3.3	TPM §9.3.4	Gopal Toueg §9.3.5	RTCAST §9.3.6	MARS §9.3.7
Ordering mechanism																				
class	fixed sequencer									moving sequencer				privilege-based						
time-based																				○
System model																				
synchrony	asynchronous													timed asynchronous	asynchronous			round synch.	synchronous synch. clocks	
crash	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
omission									○									○		
Byzantine									○											
partitionable																○				
reliable						○	○	○	○					○				○		○
FIFO				○	○	○			○									○	○	
authent.									○											
Condition needed for liveness																				
live ...						GM		GM					GM							
other	NS	△	NS	recovery			NS		NS	NS	NS			NS	NS	NS	NS	n/a	n/a	n/a
Building blocks																				
view sync.						○		○	○											
reliable b.																				
causal b.						○														
consensus																				
other																				TDMA
Properties ensured																				
Agreement	inf.	inf.	inf.	○	NS	inf.	inf.	○	○	inf.	inf.	inf.	○	inf.	inf.	○	○	○	○	NS
Unif. A				×	NS			○/×	○				×			○/×	○	○	○	NS
Total Order	inf.	○	inf.	○	○	inf.	inf.	○	○	inf.	inf.	○	inf.	inf.	inf.	○	○	○	○	NS
Unif. TO		×		×	×			○/×	○			○				○/×	○	○	○	NS
FIFO order					△	○					○					○				
causal ord.					△	○					○					○				
Destination groups																				
multiple				○	○												△			
open													○							
Fault tolerance mechanism																				
process	GM	spec.	GM	block.	GM	GM	GM	GM	GM	GM	GM	GM	GM	GM	GM	GM	GM	Cons.	GM	GM
comm.	+a,-a	-a	+a	-a	-a	n/a	n/a	n/a	n/a	+a,-a	+a,-a	+a,-a	-a	+a,-a	GM	-a	+a,-a	n/a	GM	n/a

Table 4: Overview of total order broadcast algorithms (part II).

algorithm	Lamport §9.4.1	Psync §9.4.2	Newtop (sym.) §9.4.3	Ng §9.4.4	ToTo §9.4.5	Total §9.4.6	ATOP §9.4.7	COReL §9.4.8	determ. merge §9.4.9	HAS §9.4.10	Redund. chan. §9.4.11	Quick-S §9.4.12	ABP §9.4.13	Atom §9.4.14	QoS preserv. §9.4.15	Newtop (asym.) §9.6.1	hybrid §9.6.2	indulg. unif. TO §9.6.3	opt. TO in WAN §9.6.4
Ordering mechanism																			
class	communication history															hybrid			
time-based									○	○	○	○							○
System model																			
synchrony	asynchronous									synchronous sync. clocks		spec.	sync.	synchronous sync. clocks		asynchronous			
crash		○	○	○	○	○	○	○		○	○	○	○	○	○	○	○	○	○
omission										○		○							
Byzantine							spec.			spec.		○							
partitionable			○		○	○	○	○								○			
reliable	○		○	○	○		○	○	○			n/a		○	○	○	○	○	○
FIFO	○		○				○	○	○			n/a		○	○	○	○		
Condition needed for liveness																			
live ...							GM											Cons.	
other		NS	NS	NS	NS	NS			spec.	n/a	n/a	n/a	n/a	n/a	n/a	NS	NS		NS
Building blocks																			
view sync.			○		○		○	○								○	○		
reliable b.																		○	○
causal b.					○														
consensus														△				○	
other									spec.			ByzA.							
Properties ensured																			
Agreement	inf.	△	○	○	○	△	×	○	○	○	○	○	○	○	×	○	inf.	○	inf.
Unif. A	n/a	△	×	○	○/×	n/a	n/a	○	n/a	×	×	○/×	○	○	n/a	×		○	
Total Order	inf.	inf.	○	inf.	○	○	inf.	○	○	○	○	○	○	○	○	○	inf.	○	inf.
Unif. TO	n/a		×		○/×	○		○	n/a	×	×	○/×	×	○	×	×		○	
FIFO order	○	○	○		○			○						○		○			
causal ord.	○	○	○		○			○								○			
Destination groups																			
multiple			○					○								○			
open																	????		
Fault tolerance mechanism																			
process	n/a	FD	GM	FD	GM	RCP	GM	GM	n/a	RCP	RCP	ByzA.	GM	GM	GM	GM	GM	Cons.	NS
comm.	n/a	+a	n/a	n/a	n/a	+a,-a	n/a	n/a	n/a	flood.	spec..	n/a	+a	n/a	n/a	n/a	n/a	n/a	n/a

Table 5: Overview of total order broadcast algorithms (part III).

31

algorithm	Skeen §9.5.1	Luan Gligor §9.5.2	Le Lann Bres §9.5.3	Chandra Toueg §9.5.4	Rodrigues Raynal §9.5.5	ATR §9.5.6	Scal-atom §9.5.7	Fritzke et al. §9.5.8	optim. ABcast §9.5.9	prefix agreem. §9.5.10	generic bcast §9.5.11	thrifty generic §9.5.12	weak order. §9.5.13	AMp xAMp §9.5.14	Quick-A §9.5.15
General															
class	destinations agreement														
time-based															
System model															
synchrony	asynchronous			asynchronous w/failure detectors								async. w/oracle		sync.	async. w/ByzA.
crash	○	○	○	○	spec.	○	○	○	○	○	○	○	○	○	○
omission			○												○
Byzantine															○
partitionable															
reliable	○			○			○	○	○	○	○	○	○	○	n/a
FIFO							○		○						n/a
Condition needed for liveness															
live ...				Cons.	Cons.		Cons.	Cons.	Cons.	Cons.	Cons.				
other	NS	NS	NS			◇𝒫						△	△	spec.	n/a
Building blocks															
view sync.															
reliable b.				○			○	○	○	○	○	○			
causal b.															
consensus				○	○		○	○	○			○			○
other												spec.			
Properties ensured															
Agreement	inf.	inf.	×	○	○	○	○	○	○	○	○	○	○	○	○
Unif. A			n/a	×	○	○	○	○	×	○	○	○	○	○	○/×
Total Order	inf.	inf.	○	○	○	○	○	○	○	○	○	○	○	×	○
Unif. TO			×	×	○	○	○	○	×	○	○	○	○		○/×
FIFO order												○		○	
causal ord.												○		○	
Destination groups															
multiple	○						○	○							
open	○						○								
Fault tolerance mechanism															
process	GM	FD	RCP	Cons.	Cons.	GM	Cons.	Cons.	Cons.	Cons.	Cons.	n/a	RCP	GM	Cons.
comm.	n/a	-a	RCP	n/a	gossip.	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a	n/a

9.1 Fixed Sequencer Algorithms

Regardless of the variant they adopt (see Sect. 7.1), all sequencer algorithms assume an asynchronous system model and use time-free ordering. They tolerate crash failures, except for Rampart which additionally tolerates Byzantine failures. Also, they all rely on process-controlled crash to cope with failures; either explicitly (e.g., Tandem), or through a group membership and exclusion (e.g., Isis, Rampart).

9.1.1 Amoeba

The Amoeba [Kaashoek and Tanenbaum 1996] group communication system supports algorithms of the first two variants of fixed sequencer algorithms. The first one corresponds to the variant UB (unicast-broadcast) illustrated in Figure 6(a) (Sect. 7.1). The second variant corresponds to BB (broadcast-broadcast), see Figure 6(b). The two variants share the same properties.

Amoeba assumes lossy channels and implements message retransmission as part of the total order broadcast algorithm. Amoeba uses a combination of positive and negative acknowledgments. The actual protocol is quite complex because it is combined with flow control, and also tries to minimize the communication cost. Amoeba tolerates failures using a group membership service. Suspected processes are excluded from the group as the result of the unilateral decision of a single process.

The properties of the Amoeba algorithms are only discussed informally in the paper. However, since messages are delivered before they are stable, the algorithm can only satisfy the non-uniform properties of Agreement and Total Order.

9.1.2 MTP

MTP [Armstrong et al. 1992] is an algorithm primarily designed for video streaming and other similar multimedia applications. The algorithm assumes that the system is not uniform with respect to the probability of process failures. In particular, it assumes that a process, called the master process, never fails. The master is then designated as the sequencer, and the protocols follows variant UUB (unicast-unicast-broadcast, see Fig. 6(c); p.19). When a process p has a message m to broadcast, p requests a sequence number for m from the sequencer. Once it has obtained the sequence number, it sends m together with the sequence number, to all destinations and the master. At the same time, destination processes learn about the status of previous messages and deliver those that have been accepted by the master.

The protocol tolerates crash failures of destination processes and senders, since all parts involving decisions are executed by the master. The failure of the master is briefly discussed at the end of the paper. The authors suggest that the master could be rendered more resilient by introducing redundancy and using replication techniques.

9.1.3 Tandem

The Tandem global update protocol [Carr 1985] is a fixed sequencer algorithm of variant UUB (see Fig. 6(c)). The algorithm allows at most one application message to be broadcast at a time, and thus does not need sequence numbers. Later, [Cristian et al. 1994] describe a variant UB of Tandem that allows concurrent broadcasts (and thus needs sequence numbers).

9.1.4 Garcia-Molina and Spauster

The algorithm proposed by [Garcia-Molina and Spauster 1991] is based on a propagation graph (a forest) to support multiple overlapping groups. The propagation graph is constructed is such a way that each group is assigned a starting node. Senders send their messages to the corresponding starting nodes and messages travel along the edges of the propagation graph. Ordering decisions are resolved along the path. When used in a single group setting, the algorithm behaves like other fixed sequencer algorithms (i.e., the propagation graph is a tree of depth 1).

The algorithm assumes an asynchronous model and requires synchronized clocks. However, synchronized clocks are only needed to yield bounds on the behavior of the algorithm when crash failures occur. Neither the ordering mechanism nor the fault tolerance mechanism actually need them.

In the event of failures, the algorithm behaves in an unconventional manner. Indeed, if a non-leaf process p crashes, then its descendants in the propagation graph do not receive any message until p has recovered. Hence, the algorithm tolerates process crashes only if those processes are guaranteed to eventually recover.

9.1.5 Jia

[Jia 1995] proposed another algorithm based on propagation graphs, which creates simpler graphs than the algorithm of Garcia-Molina and Spauster (see §9.1.4). Unfortunately, the algorithm originally proposed by [Jia 1995] is incorrect. [Chiu and Hsiao 1998] provide a correction to the algorithm which works only in a more restricted model (i.e., only closed groups). Also, [Shieh and Ho 1997] provide a correction to the message complexity calculated by [Jia 1995].

Jia's algorithm relies on the notion of meta-groups, which is defined in the paper as "the set of processes which have exactly the same group memberships" (i.e., the set of processes which belong to the exact same set of destination groups). The meta-groups are organized into propagation trees, according to the membership they represent. The flow of messages is streamlined down the trees, thus creating the delivery order.

[Jia 1995] describes a form of group membership mechanism which is used to redefine the parts of the propagation graph that must change when a process is deleted. Jia also suggests that, unlike Garcia-Molina and Spauster's algorithm (§9.1.4), the nodes in the tree consist of entire meta-groups rather than single processes. Thus, messages would not be stopped unless all members in an intermediary meta-group fail. The issue is however only addressed informally.

9.1.6 Isis (sequencer)

[Birman et al. 1991] describe several broadcast primitives of the Isis system, including a total order broadcast primitive called ABCAST. The ABCAST primitive is implemented using a fixed sequencer algorithm (different from the algorithm used in earlier versions of the system; see §9.5.1, p.40). The Isis (sequencer) algorithm is a fixed sequencer algorithm of variant BB (see Fig. 6(b), p.19), which uses a causal broadcast primitive. The algorithm assumes crash failures.

Being constructed over a causal broadcast primitive, the Isis ABCAST algorithm preserves causal order. Moreover, although the algorithm does not support total order for multiple overlapping groups, causal order is nevertheless preserved in this context. The total order broadcast algorithm ensures only the non-uniform properties of Agreement and Total Order.

For fault tolerance, the total order broadcast algorithm relies on a group membership service and on the property of view synchrony (Sect. 8.2).

Finally, the authors also briefly mention that moving the role of the sequencer in the absence of failures might be a way to avoid a bottleneck. However, the idea is not developed further.

9.1.7 Navaratnam et al.

[Navaratnam et al. 1988] propose a fixed sequencer protocol of variant UB (see Fig. 6(a)).

The fault tolerance of the algorithm relies on a group membership service and the ability to exclude wrongly suspected processes. Similar to Amoeba (§9.1.1), the decision to exclude a suspected process can be taken unilaterally by one single process.

The properties of this algorithm are discussed informally, and it is easy to see that it satisfies the non-uniform properties of Agreement and Total Order. The authors also make a brief remark suggesting that the algorithm does not guarantee uniform properties, but the wording is a little ambiguous and the information provided in the paper is not sufficient to verify this interpretation.

9.1.8 Phoenix

Phoenix [Wilhelm and Schiper 1995] consists of three algorithms which provide different levels of 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 guarantees on a per message basis.

The three algorithms are based on a group membership service and view synchrony (see Sect. 6.1).

9.1.9 Rampart

Unlike other sequencer algorithms, which only assume crash failures, the algorithm of Rampart [Reiter 1994, Reiter 1996] is designed to tolerate Byzantine failures. This sets this algorithm somewhat apart from the other

sequencer algorithms.

Rampart assumes an asynchronous system model with reliable FIFO channels, and a public key infrastructure where every process initially knows the public key of every other process. In addition, communication channels are assumed to be authenticated, so that the integrity of messages between two honest (i.e., non-Byzantine) processes is always guaranteed.

Unlike most early work on Byzantine failures, Rampart treats honest and Byzantine processes separately. In particular, the paper defines uniformity as a property that applies to honest processes only (see explanations in Sect. 6.2). With this definition, Rampart satisfies both Uniform Agreement and Uniform Total Order.

The algorithm is based on a group membership service, which requires that at least one third of all processes in the current view reach an agreement on the exclusion of some process from the group. This condition is necessary because Byzantine processes could otherwise purposely exclude correct processes from the group.

9.2 Moving Sequencer Algorithms

We describe here four moving sequencer algorithms, all of which are time-free. To the best of our knowledge, there is no time-based moving sequencer algorithm. It is actually questionable whether time-based ordering would even make sense for algorithms of this class.

The four algorithms behave in a very similar fashion. Actually, Pinwheel (§9.2.4), RMP (§9.2.2), and DTP (§9.2.3) are all three based on Chang and Maxemchuck's algorithm (§9.2.1), which they all improve in a different way. Pinwheel is optimized for a uniform message arrival pattern, RMP provides various levels of guarantees, and DTP provides a faster detection of message stability. The four algorithms also handle process failures very similarly, using a reformation algorithm (see §9.2.1).

The four algorithms tolerate message loss by relying on a message retransmission protocol that combines positive and negative acknowledgments. More precisely, the token carries positive acknowledgments, but when a process detects the miss of a message, it sends a negative acknowledgment to the token site. The negative acknowledgment scheme is used for message retransmission, whereas the positive scheme is used to detect message stability.

9.2.1 Chang and Maxemchuck

The algorithm proposed by [Chang and Maxemchuck 1984] is 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. The passing of the token simultaneously serves two different purposes: (1) the transmission of the sequencer role, and (2) the detection of message stability. Point 2 requires that the logical ring spans all destination processes. This requirement is however not necessary for ordering messages (point 1), and hence the algorithm qualifies as a sequencer-based algorithm according to our classification.

When a process failure is detected (perhaps wrongly) or when a process recovers, the algorithm goes through a reformation phase. The reformation phase redefines the logical ring and elects a new initial token holder. The reformation algorithm can be seen as an ad-hoc implementation of a group membership service.

The properties of the total order broadcast algorithm are discussed only informally. Nevertheless, it seems plausible that the algorithm ensures Uniform Total Order and Uniform Agreement.

9.2.2 RMP

RMP [Whetten et al. 1994] differs from the other three algorithms in that it is designed to operate with open groups. Beside, the authors claim that "RMP provides multiple multicast groups, as opposed to a single broadcast group." However, according to their description, supporting multiple multicast groups is merely a characteristic associated with the group membership service. It is hence dubious that "multiple groups" is used with the meaning that total order is guaranteed for processes that are at the intersection of two groups (see discussion in Sect. 5.2).

Depending on the user's choice, RMP satisfies Agreement, Uniform Agreement, or neither of these properties. However, in order to ensure the strong guarantees, RMP must assume that a majority of the processes remain correct and always connected. Also, RMP does not preclude the contamination of the group.

9.2.3 DTP

Unlike the other three algorithms, DTP [Kim and Kim 1997] does not rely on a logical ring for the passing of the token. Instead, the algorithm follows a heuristic whereby the token is always passed to the process seen as the least active. Doing this ensures that messages are acknowledged more quickly when the activity (i.e., broadcasting messages) is not uniform among processes.

9.2.4 Pinwheel

The originality of Pinwheel [Cristian et al. 1997] is that the token circulates among the processes at a speed proportional to the global activity of the sending processes (i.e., broadcasting rate).

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 of [Cristian and Fetzer 1999]. Although it relies on physical clocks for timeouts, Pinwheel does not need to assume that these clocks are synchronized. Furthermore, the algorithm is time-free since time is not used for ordering messages.

Pinwheel can ensure Uniform Total Order, given an adequate support from its group membership (not detailed in the paper). Beside, Pinwheel only satisfies (non-uniform) Agreement, but the authors argue that the algorithm could easily be modified to satisfy Uniform Agreement [Cristian et al. 1997]. Doing this would only require that destination processes wait until a message is known to be stable before delivering it. The authors claim that the algorithm preserves causal order, but this is valid only under certain restrictions that make the problem trivial to solve.[18]

9.3 Privilege-Based Algorithms

As for moving sequencer algorithms, most privilege-based algorithms are based on a logical ring, and for most of them rely on some kind of group membership or reconfiguration protocol to handle process failures.

9.3.1 On-demand

The On-demand protocol [Cristian et al. 1997], unlike other privilege-based algorithms, does not rely on a logical ring. Instead, processes with a message to broadcast must obtain the token by issuing a request to the current token holder. As a consequence of this approach, the protocol is more efficient if senders send long bursts of messages and such bursts rarely overlap. Also, in contrast with the other algorithms, *all processes* must be aware of the identity of the token holder. So, the passing of the token is done using a broadcast.

The on-demand protocol relies on the same model as the Pinwheel protocol (§9.2.4). In other words, it assumes a timed asynchronous system model, and physical clocks for timeouts.

A similar algorithm, called Reqtoken, is also described by [Friedman and van Renesse 1997].

9.3.2 Train

The Train protocol [Cristian 1991] is inspired by the image of a train that transports messages and circulates among processes. More concretely, a token (a.k.a., the train) moves along a logical ring and carries the messages. When a process gets the token, it receives the new messages carried by the token, acknowledges them, and appends its own messages to the token. Then, it passes the token to the next process. The Train protocol, where messages are carried by the token, comes in clear contrast with the other algorithms of the same class, where messages are broadcast directly to the destinations. The Train protocol is hence less attractive than the others in a broadcast network.

9.3.3 Totem

The specificity of Totem [Amir et al. 1995] compared to other privilege-based algorithms is that it is designed for partitionable systems. The ordering guarantee ensured is Strong Total Order. Totem provides both *(non-uniform)* agreement and total order (called *agreed order*) and *uniform* agreement and total order (called *safe order*) when operated in a non-partitionable system. Causal order is also ensured.

---

[18]In systems with a single closed group where processes are only allowed to communicate using total order broadcast, causal order is satisfied trivially by simply enforcing FIFO order.

The algorithm uses a membership protocol, which has the responsibility to detect processor failures, network partitioning and loss of the token. When such failures are detected, the membership protocol reconstructs a new ring, generates a new token, and recovers messages that had not been received by some of the processors when the failure occurred.

The authors observe that while moving sequencer algorithms (in which holding the token is not required to broadcast a message) have good latency at low loads, latency increases at high load and in the presence of processor crashes. Moreover, according to [Agarwal et al. 1998], 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, they 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 single-ring one.

9.3.4 TPM

TPM [Rajagopalan and McKinley 1989] is closely related to Totem. The main difference is that TPM is not partitionable (it only supports primary partition membership). Moreover, TPM only provides *uniform* agreement and total order. Finally, while TPM only supports a closed group, the authors discuss some ideas on how to extend the algorithm to support multiple closed groups.

[Rajagopalan and McKinley 1989] also propose a modification 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.

9.3.5 Gopal and Toueg

Gopal and Toueg's [1989] algorithm is based on the round synchronous model. The round synchronous model is a computation model in which the execution 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 non-crashed processes, and then (3) perform some computations.

The algorithm works as follows. For each round, one of the processes is designated as the *transmitter*. The transmitter of some round r is the only process which is allowed to broadcast new application messages in round r. In that round the other processes broadcast acknowledgments of previous messages. Messages are delivered once they are acknowledged, three rounds after their initial broadcast.

9.3.6 RTCAST

RTCAST [Abdelzaher et al. 1996] was designed for applications that need real-time guarantees. The algorithm assumes a synchronous system with synchronized clocks. These strong guarantees allow for simplification in the protocol. The paper also shows how the maximum token rotation time can be used for admission control and schedulability analysis of real-time messages (with the goal to guarantee the delivery deadline of these messages).

9.3.7 MARS

MARS [Kopetz et al. 1990] is based on the principle of *time division multiple-access* (TDMA; see Note 11, p.20). TDMA consists in having predetermined periodic time slots assigned to each process. Processes are then allowed to send or broadcast messages messages only during their own time slots. The system assumes that processes have synchronized clocks whereby they are able to accurately determine the beginning and the end of their own time slot. In addition, communication is assumed to be reliable and with bounded delays.

Based on the mutual exclusion provided by the TDMA model and the communication model, total order broadcast is easily implemented. The ordering mechanism can be seen as similar to Gopal and Toueg's algorithm (§9.3.5), but in a time-based model and where communication uses time rather than messages [Lamport 1984].

[Kopetz et al. 1990] do not discuss the behavior of their total order broadcast algorithm in the presence of failures. This makes it difficult to determine whether the algorithm is uniform or not. We believe that it is not uniform, simply because uniformity induces a cost in performance that the authors are unlikely to consider affordable.

9.4 Communication History Algorithms

9.4.1 Lamport

The principle of Lamport's algorithm [Lamport 1978b], which uses logical clocks, has been explained in Section 7.4 (see Fig. 11). Actually, the paper describes a mutual exclusion algorithm. However it is straightforward to derive a total order broadcast algorithm from the mutual exclusion algorithm. Since the delivery order of a message m is determined by the timestamp of the broadcast event of m, the total order is an extension of causal order. The algorithm is not tolerant to failures.

A similar algorithm is described by [Attiya and Welch 1994], when comparing consistency criteria.

9.4.2 Psync

The Psync algorithm [Peterson et al. 1989] is used in several group communication systems: Consul [Mishra et al. 1993], Coyote [Bhatti et al. 1998], and Cactus [Hiltunen et al. 1999]. In Psync, processes dynamically build a causality graph of messages they receive. Psync then delivers messages according to a total order that is an extension of the causal order.

Psync assumes an asynchronous system model with (permanent) crash failures and (transient) lossy communication. To tolerate process failures, the algorithm seems to assume a perfect failure detector, although this is not said explicitly in the paper. To implement reliable channels, the algorithm uses negative acknowledgments (to request the retransmission of lost messages).

Psync is specified only informally. Nevertheless, we believe that the protocol ensures Total Order in the absence of failures. The behavior in the face of failures is unfortunately not described with enough detail to make a confident claim about it. Agreement is a little more complex. In the absence of message loss, Psync ensures Agreement. However, with certain combinations of process crash and message loss, it is possible that some correct processes *discard* messages that are otherwise delivered by others. Hence, when message loss are considered, Agreement can be violated. This problem is discussed in details by the authors, who relate it to an instance of the "last acknowledgment problem."

[Malhis et al. 1996] provide an analysis of the performance of Psync in the presence of message loss. They conclude that Psync performs well if broadcast are frequent and message loss rare, but performs poorly when broadcast are infrequent and message loss common. They show that the performance can be improved by sending empty messages regularly, as is done by other communication history algorithms (see Note 14 in Sect. 7.4).

9.4.3 Newtop (symmetric)

[Ezhilchelvan et al. 1995] propose two algorithms: a symmetric one and an asymmetric one. The symmetric algorithm extends Lamport's algorithm (§9.4.1) in several ways: makes it fault-tolerant, allows a process to be member of multiple groups, and allows the broadcast of a message to multiple groups. As for Lamport's algorithm, Newtop preserves causal order.

Newtop is based on a partitionable group membership service (see Sect. 6.1, p.17). The Newtop platform leaves it to applications to decide whether or not they should maintain more than one subgroup in such a situation. Newtop satisfies the property of Weak Total Order mentioned in Sect. 6.1.

The asymmetric algorithm belongs to a different class, and is hence discussed there (see §9.6.1). The two algorithms (symmetric and asymmetric) can easily be combined to allow the use of the symmetric algorithm in some groups, and the asymmetric algorithm in other groups.

9.4.4 Ng

[Ng 1991] 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 acknowledgment 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.

9.4.5 ToTo

The ToTo algorithm [Dolev et al. 1993] ensures Weak Total Order (see Section 6.1; called "agreed multicast" in [Dolev et al. 1993]). It is build on top of the Transis partitionable group communication system [Dolev and Malkhi 1996]. ToTo extends the order of an underlying causal broadcast algorithm. It is based on dynamically building a causality graph of received messages. The Transis system offers both a uniform and a non-uniform variant of the algorithm. A particularity of ToTo (non-uniform variant) is that, to deliver a message m, a process must have received acknowledgments for m from as few as a majority of the processes in the current view (instead of all view members).

9.4.6 Total

The Total algorithm [Moser et al. 1993] is built on top of a reliable broadcast algorithm called Trans (Trans in defined together with Total). However, Trans is not used as a black box (which explains that we did not list reliable broadcast as a building block in Table 4). Trans uses an acknowledgment mechanism that defines a partial order on messages. Total extends the partial order of Trans into a total order. Two variants are defined: the more efficient one tolerates $f < n/3$ crashes and the other tolerates $f < n/2$ crashes.

The Total algorithm fulfills the Agreement property (in fact, Uniform Agreement) with high probability. Actually Total requires that the underlying Trans reliable broadcast protocol provides probabilistic guarantees about not reordering messages. This has some similarities with the notion of *weak ordering oracles* (see Section 9.5.13).

[Moser and Melliar-Smith 1999] propose an extension of Total to tolerate Byzantine failures.

9.4.7 ATOP

ATOP [Chockler et al. 1998] is an algorithm following the deterministic merge approach (Sect. 7.4). The focus of the paper is adapting the algorithm to different and possibly changing sending rates. A pseudo-random number generator is used in computing the delivery order.

The paper is mostly concerned with ensuring an ordering property. This property is Strong Total Order, defined in the context of partitionable systems (Sect. 6.1). The algorithm ensures FIFO order, and ensures causal order only trivially (see Footnote 18, p.35).

9.4.8 COReL

The COReL algorithm [Keidar and Dolev 2000] is built on top of a partitionable group membership service like Transis. The underlying service should also offer Strong Total Order (Sect. 6.1) as well as causal order. COReL gradually builds a global order (Reliable Total Order) by tagging messages according to three different color levels (red, yellow, green). A message starts as red (no knowledge about its position in the global order) then passes to yellow (received and acknowledged when the process is a member of a majority component) and green (all members of the majority component acknowledged the message, and its position in the global order is known). Green messages are delivered to the application. Messages are retransmitted and promoted to green whenever partitions merge. All acknowledgments sent by the algorithm are piggybacked. COReL provides the following liveness guarantee: if eventually there is a stable majority component, all messages sent by the members of this component are delivered.

COReL also supports process recovery if processes are equipped with stable storage. This requires that processes log each message that is sent (before sending the message) and each message that is received (before sending an acknowledgment).

[Fekete et al. 2001] formalize a variant of the COReL algorithm and the guarantees offered by the underlying group membership service, using I/O automata.

9.4.9 Deterministic merge

In the deterministic merge algorithm [Aguilera and Strom 2000], each message received deterministically defines the sender of the next message to be accepted. Senders put a physical timestamp on their messages, and upon receiving such a message, the destination process computes (using the timestamp) the next sender from which it will accept a message. The algorithm is most efficient if clocks are synchronized (but works even if clocks are not synchronized) and each sender sends messages at some fixed rate known a priori (the rate may be different for

each sender). To make the algorithm live, senders need to send empty messages if they have no message to send (these messages divide the execution into independent *epochs*). The algorithm, as described, is not fault-tolerant.

9.4.10 HAS

[Cristian et al. 1995] propose a collection of total order broadcast 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 neighbors. Processes deliver message m at time $T + \Delta$, according to their local clock (where Δ is constant that depends on the topology of the network, the number of failures tolerated, and the maximum clock drift ϵ).

The paper proves 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 could violate the safety of the protocol (i.e., Total Order) rather than just its liveness.

9.4.11 Redundant broadcast channels

[Cristian 1990] presents an adaption of the HAS-$\mathcal{O}$ algorithm (omission failures) to 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). Compared to HAS-$\mathcal{O}$, the algorithm for redundant broadcast channels issues significantly less messages.

9.4.12 Quick-S

[Berman and Bharali 1993] present several closely related total order broadcast algorithms in a variety of system models. In synchronous systems (3 variants in the paper) the algorithms are similar to the HAS algorithms: messages are timestamped (with physical or logical timestamps, depending on the system model), and a message timestamped with T can be delivered at $T + \Delta$ for some value of Δ. The difference is that they use a Byzantine agreement algorithm with bounded termination time to send messages. There are algorithms that work with Byzantine failures and ones that work with crash failures only; the latter ensure Uniform Prefix Order. For Byzantine failures, the algorithm ensures only non-uniform properties. This is because, unlike the *specification* of Rampart (§9.1.9), the *specification* used by Quick does not distinguish between Byzantine processes and those that only fail by crashing.

The paper also presents an algorithm for asynchronous systems. However, this algorithm belongs to the class of destinations agreement algorithms and is discussed there (Quick-A; §9.5.15).

9.4.13 ABP

The principle of ABP [Minet and Anceaume 1991b, Anceaume 1993a] is close to the principle of Lamport's algorithm (§9.4.1): messages are delivered according to timestamps attached to messages by their sender. Each process manages a local sequence number variable, used to timestamp messages. Let process p broadcast message m. In the first phase, m and its timestamp value ts_m are sent to all. Any process q receiving m replies with some message m' that it might have previously broadcast with the same timestamp value ($ts_{m'} = ts_m$), if any. Upon reception of all replies from correct processes, process p knows the set $Msg(ts_m)$ of all messages with the same timestamp value ts_m, and delivers these messages (in the order of the identifier of the sender of the messages). Process p also broadcasts the set $Msg(ts_m)$, allowing the other processes to deliver the same sequence of messages.

9.4.14 Atom

In Atom [Bar-Joseph et al. 2002], streams of messages from all senders are merged in a round-robin fashion. To make the algorithms live, senders need to send empty messages if they have no message to send. This approach can be seen as a special case of *deterministic merge* (see §9.4.9).

9.4.15 QoS preserving atomic broadcast

[Bar-Joseph et al. 2000] present another algorithm, based on the same ordering mechanism as Atom (§9.4.14). As its name indicates, the QoS preserving algorithm provides support for quality of service (QoS), unlike Atom. On the other hand, the QoS preserving algorithm does not guarantee Agreement (i.e., uniform or not), and only non-uniform Total Order.

9.5 Destinations Agreement Algorithms

9.5.1 Skeen

Skeen's algorithm, described by [Birman and Joseph 1987], was used in an early version of the Isis toolkit. The algorithm corresponds roughly to the algorithm in Fig. 13 (p.24). The main difference is that Skeen's algorithm computes the global timestamp in a centralized manner, whereas the algorithm in Fig. 13 does it in a decentralized way. Fault tolerance is achieved using a group membership service, which excludes suspected processes from the group.

[Dasser 1992] propose a simple optimization of Skeen's algorithm called TOMP, where additional information is appended to protocol messages in order to deliver application messages a little earlier.

9.5.2 Luan and Gligor

[Luan and Gligor 1990].

The algorithm is based on majority voting. The idea is the following. Upon execution of *TO-broadcast*(m), message m is sent to all processes. Upon reception of m by some process q, m is put into q's receiving buffer. The message delivery order is then decided by a voting protocol, which can be initiated by any of the processes. In case of concurrent initiation of the protocol, an arbitration rule is used.

Voting is initiated by broadcasting an "invitation" message. Consider this message broadcast by process p. Processes reply by sending the content of their receiving buffer to p. Process p waits for a majority of replies. Based on the messages received, process p then constructs a sequence of message identifiers, and broadcasts this sequence. A process receiving the sequence sends an acknowledgment to p. Once p has received acknowledgments from a majority of processes, the proposed sequence is *committed*.

To summarize, the protocol tries to reach consensus among the destination processes on a sequence of messages. However, the authors did not identify consensus as a subproblem to solve, which makes the protocol more complex. The consequence is also that the conditions under which liveness is ensured are not discussed (and difficult to infer).

9.5.3 Le Lann and Bres

[Le Lann and Bres 1991] wrote a position paper discussing total order broadcast in a system with omission faults. The paper sketches a total order broadcast algorithm based on quorums.

9.5.4 Chandra and Toueg

[Chandra and Toueg 1996] propose a transformation of atomic broadcast into a sequence of consensus problems, where each consensus decides on a set of messages, easily transformed into a sequence of messages. The transformation uses reliable broadcast. The idea of the transformation was described in Section 7.5, and is not repeated here.

The algorithm assumes an asynchronous system model, reliable broadcast, and a black box that solves consensus. The algorithm is extremely elegant, in the sense that all difficult issues related to fault tolerance are hidden into the consensus black box.

There has been several proposals to optimize this algorithm. For example, [Mostefaoui and Raynal 2000] propose an optimistic approach, where the consensus algorithm is split into two parts. The first phase is optimized, but does not always succeed. If this happens, the full consensus algorithm is executed.

9.5.5 Rodrigues and Raynal

[Rodrigues and Raynal 2000] present a total order broadcast algorithm in a model where processes have access to stable storage and may recover after a crash. The algorithm is very close to the Chandra-Toueg algorithm (§9.5.4): it uses the same transformation of total order broadcast to consensus. The only difference is that, because of the crash-recovery model, the algorithm relies on the crash-recovery consensus algorithm of Aguilera, Chen, and Toueg [2000]).

9.5.6 ATR

[Delporte-Gallet and Fauconnier 1999] describe the ATR algorithm, which is based on an abstraction called *Synchronized Phase system (SPS)*. The SPS abstraction is defined in an asynchronous system. An SPS decomposes the execution of an algorithm in rounds, almost like a synchronous round model. The ATR algorithm distinguishes between even and odd rounds. In even rounds, processes send ordered sets of messages to each other. Upon reception of these messages, each process constructs a sequence of messages. In the subsequent odd round, processes try to validate the order and deliver messages.

9.5.7 SCALATOM

SCALATOM [Rodrigues et al. 1998] is based on Skeen's algorithm (§9.5.1) and supports the broadcast of messages to multiple groups. The algorithm satisfies the Strong Minimality property (Sect. 5.2.4). The global timestamp is computed using a variant of Chandra and Toueg's [1996] consensus algorithm (§9.5.4). SCALATOM corrects an earlier algorithm called MTO [Guerraoui and Schiper 1997].

9.5.8 Fritzke et al.

[Fritzke et al. 2001] also propose an algorithm for the broadcast of messages to multiple groups. The algorithm satisfies the Strong Minimality property (Sect. 5.2.4). Consider a message m broadcast to multiple groups. First, the algorithm uses consensus to decide on the timestamp of m within each destination group. The destination groups then exchange information to compute the final timestamp, and a second consensus is executed in each group to update the logical clock.

9.5.9 Optimistic atomic broadcast

Optimism is a technique known since several years in the context of concurrency control [Bernstein et al. 1987] and file system replication [Guy et al. 1993]. However, it was only considered recently in the context of total order broadcast [Pedone 2001].

The optimistic atomic broadcast of Pedone and Schiper [Pedone and Schiper 1998, Pedone and Schiper 2003] is based on the experimental observation that messages broadcast in a LAN are usually received in the same order by every process. When this assumption is met, the algorithm delivers messages extremely fast. However, if the assumption does not hold, the algorithm is less efficient than other algorithms (but still delivers messages in total order).

Unlike most optimistic algorithms, the optimistic atomic broadcast of [Pedone and Schiper 2003] is optimistic *internally*. This means that the optimistic mechanism of the algorithm is not apparent to the application. In other words, there is no weakening of the delivery properties.

9.5.10 Prefix agreement

[Anceaume 1997] defines a variant of consensus, called *prefix agreement*, wherein processes agree on a stream of values rather than on a single value. Considering streams rather than single values makes the prefix agreement algorithm particularly well suited to solve total order broadcast. The total order broadcast algorithm uses prefix agreement to repeatedly decide on the sequence of messages to be delivered next.

9.5.11 Generic Broadcast

Generic broadcast [[Pedone and Schiper 1999]; [2002]] is not a total order broadcast per se. Instead, the algorithm assumes a *conflict* relation on the messages, and two messages m and m' are delivered in the same order at each destination process only if they conflict. Two messages m and m' that do not conflict are not ordered by the algorithm. If all messages conflict, then generic broadcast provides the same guarantee as total order broadcast. If no messages conflict, then generic broadcast provides the guarantees of (uniform) reliable broadcast. The strong point of this algorithm is that performance varies according to the required "amount of ordering": the generic broadcast algorithm uses a consensus algorithm only in case of conflicts.

9.5.12 Thrifty generic broadcast

Aguilera, Delporte-Gallet et al. [2000] propose also a generic broadcast algorithm. When conflicting messages are detected, [Pedone and Schiper 2002] solve generic broadcast by reduction to consensus, while [Aguilera et al. 2000] solve generic broadcast by reduction to total order broadcast. In addition, the algorithm is *thrifty* in the sense that, if there is a time after which broadcast messages do not conflict with each other, then eventually atomic broadcast is no longer used. The algorithm of [Pedone and Schiper 2002] also satisfies this property with respect to consensus, but the property was not identified in the paper.

9.5.13 Weak ordering oracles

[Pedone et al. 2002] define a *weak ordering oracle* as an oracle that orders messages that are broadcast, but is allowed to make mistakes (i.e., the messages broadcast may be delivered out of order). This oracle models the behavior observed in local-area networks, where broadcast messages are often spontaneously delivered in total order. The paper shows that total order broadcast can be solved using a weak ordering oracle. If the optimistic assumption is met, the proposed algorithm, which assumes $f < \frac{n}{3}$, solves total order broadcast in two communication steps.

Interestingly, the algorithm has the same structure as the randomized consensus algorithm proposed by [Rabin 1983]. The authors mention also that the weak ordering oracle could be used to design an total order broadcast algorithm that has the same structure as the randomized consensus algorithm proposed by [Ben-Or 1983].

9.5.14 AMp/xAMp

The AMp [Veríssimo et al. 1989] and xAMp [Rodrigues and Veríssimo 1992] algorithms rely on the assumption that messages broadcast are most of the time received by all destination processes in the same order (realistic assumption in LANs, as already mentioned). So, when a process broadcasts a message, it initiates a commitment protocol. If the message is received ordered by all destination processes, then the outcome is positive: all destination processes commit and deliver the message. Otherwise, the message is rejected and the sender must try again (thus potentially leading to a livelock).

9.5.15 Quick-A

[Berman and Bharali 1993] present a series of four algorithms, three of which belong to another class (see Quick-S; §9.4.12). Their algorithm for asynchronous systems is quite different from their algorithms for synchronous systems (§9.4.12). Processes maintain a round number, and messages broadcast are timestamped with this round number. The processes then execute a sequence of randomized binary consensus, to decide on the round in which messages are to be delivered.

9.6 Hybrid Algorithms

We give here algorithms that do not fit into one of our five classes of total order broadcast algorithms. These algorithms usually combine two different ordering mechanisms.

9.6.1 Newtop (asymmetric)

[Ezhilchelvan et al. 1995] propose two algorithms: a symmetric one and an asymmetric one. The symmetric algorithm is described earlier (§9.4.3).

The asymmetric algorithm uses a sequencer process, and allows a process to be member of multiple groups (each group has an independent sequencer). For ordering, the algorithm uses Lamport's logical clocks in addition to the sequencer. Hence the asymmetric algorithm is a hybrid between a communication history algorithm (due to the use of Lamport's clocks) and a fixed sequencer algorithm. The asymmetric algorithm, like the symmetric one, preserves causal order delivery. However, note that a process p, which is member of more than one group, cannot broadcast a message m to a group immediately after broadcasting some message m' to a different group. Process p can only do so once it has delivered m. Hence, the asymmetric algorithm does not technically allow a message to be broadcast to more than one group.

As mentioned earlier, Newtop supports the combination of groups even if one group uses the asymmetric algorithms and the other group uses the symmetric one. Also, Newtop is based on a partitionable group membership service.

9.6.2 Rodrigues et al.

[Rodrigues et al. 1996] present an algorithm optimized for large networks. The algorithm is hybrid: on a local scale, a sequence number is attached to each message by a fixed sequencer, and on a global scale, the ordering is of type communication history. More precisely, each sender p has an associated sequencer process that issues a sequence number for each message of p. The original message and its sequence number are sent to all, and messages are finally are ordered using a standard communication history technique (see §9.4.1). The authors also describe interesting heuristics to change the sequencer process. Reasons for such changes can be failures, membership changes or changes in the traffic pattern.

9.6.3 Indulgent uniform total order

[Vicente and Rodrigues 2002] propose an optimistic algorithm for wide-area networks. The algorithm is based on external optimism, as initially proposed by Kemme et al. [[1999] ; [2003]]. This means that the algorithm distinguishes between two delivery events following the broadcast of message m: the optimistic delivery, denoted *Opt-deliver(m)*, and the traditional total order delivery denoted *Adeliver(m)*. Upon *Opt-deliver(m)* the delivery order of m is not yet decided. However, the application can start processing m. If later *Adeliver(m)* invalidates the optimistic delivery order, then the application must rollback and undo the processing of m. The optimism of [Kemme et al. 2003] is related to the spontaneous total ordering in LANs.

The optimistic algorithm of [Vicente and Rodrigues 2002] extends the hybrid algorithm of [Rodrigues et al. 1996] (§9.6.2). The delivery order is determined by sequence numbers attached to messages. A sequence number attached to a message m needs to be validated by a majority of processes before the total order of m is decided. Nevertheless, the algorithm optimistically delivers m according to its sequence number before the sequence number is actually validated.

9.6.4 Optimistic total order in WANs

Optimistic total order broadcast algorithms rely heavily on the assumption that messages are very often received by all processes in some spontaneous total order. This assumption was motivated by observations made in local networks, often over a single hub. This assumption is however questionable for wide-area networks, where the spontaneous total order is significantly less likely to occur. [Sousa et al. 2002] propose a time-based solution to address this problem and increase the probability of spontaneous total order in wide-area networks. The technique, called *delay compensation*, consists in delaying received messages artificially, so that all destinations would process them at roughly the same time. A delay in kept for each incoming communication channel, and the duration of this delay is adapted dynamically.

10 Other Work on Total Order and Related Issues

Apart from papers proposing total order broadcast algorithms, there is some work, somehow closely related to the topic, that is worth mentioning. This is the purpose of this section.

Backoff protocol Chockler, Malkhi, and Reiter [2001] describe a replication protocol which emulates *state machine replication* [Lamport 1978a, Schneider 1990]. The protocol is based on quorum systems and relies on a mutual exclusion protocol. Basically, a client process wanting to perform some operation op on the replicated servers proceeds as follows: the client first waits to enter the critical section, and then (1) accesses a quorum of replicas to get an up-to-date state σ of the replicated servers, (2) performs the operation op on σ which leads to a new state σ', and (3) updates a quorum of replicas with the new state σ'. The protocol is safe even if the mutual exclusion protocol violates safety (more than one process in the critical section): safety of the mutual exclusion protocol is only needed to ensure progress of the replication protocol.

Optimistic active replication [Felber and Schiper 2001] describe another replication protocol that is integrated with a total order broadcast algorithm. The replication protocol is based on an optimistic fixed sequencer total order broadcast algorithm, which is executed among the servers. The optimistic algorithm may lead some servers to deliver messages out of order, in which case these servers have to rollback. Rollback is limited to servers: client processes never have to rollback.

Probabilistic protocols Recently, [Felber and Pedone 2002] have proposed a total ordered broadcast algorithm with probabilistic safety. This means that their algorithms enforce the properties of total order broadcast with a known probability. Doing so gives room for extremely scalable solutions, but is only acceptable for applications with very weak requirements. In particular, [Felber and Pedone 2002] propose a specification where agreement is guaranteed with probability γ_a, total order with probability γ_o, and validity with probability γ_v. The authors propose an algorithm based on gossiping and discuss sufficient conditions under which their algorithm can enforce the above properties with probability one.

Hardware-Based Protocols Due to their specificity, we have deliberately omitted algorithms that make explicit use of dedicated hardware. They however deserve to be cited here. Some protocols are based on a modification of the network controllers (e.g., [Jalote 1998, Minet and Anceaume 1991a]). The idea is to slightly modify the network so that it can be used as a virtual sequencer. In our classification system, these protocols can be classified as fixed sequencer protocols. Some other protocols rely on the characteristics of specific networks such as a specific topology [Córdova and Lee 1996] or the ability to reserve buffers [Chen et al. 1996].

Performance of Total Order Broadcast Algorithms Compared to the host of publications describing algorithms, relatively few papers are concerned with evaluating the performance of total order broadcast (e.g., [Cristian et al. 1994, Friedman and van Renesse 1997, Mayer 1992], described in Sect. 1). In recent work [Défago et al. 2003], we present a comparative performance analysis based on the classification developed in this survey: algorithms are taken from all five classes of ordering mechanisms, and both uniform and non-uniform algorithms are considered. [Urbán et al. 2003] go beyond evaluating some algorithm or comparing different algorithms: they propose benchmarks including well-defined performance metrics, workloads, and *faultloads* describing how failures and related events occur.

Formal Methods Formal methods have been applied to the problem of total order broadcast, in order to verify the properties of algorithms [Zhou and Hooman 1995, Toinard et al. 1999, Fekete et al. 2001]; and to the problem of consensus, in order to construct a truly formal proof for an algorithm [Nestmann et al. 2003]. The proofs of [Fekete et al. 2001] were subsequently checked by a theorem prover. [Liu et al. 2001] use the notion of meta-properties to describe and analyze a protocol which switches between two total order broadcast algorithms.

Group communication controversy A few years ago, [Cheriton and Skeen 1993] began a polemic about group communication systems that provide causally and totally ordered communication primitives. 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 [Birman 1994] and [Shrivastava 1994]. Almost a decade later it appears that work on transaction systems and on group communication systems tend to influence each other for a mutual benefit [Schiper and Raynal 1996, Agrawal et al. 1997, Pedone et al. 1998, Kemme and Alonso 2000, Wiesmann et al. 2000, Kemme et al. 2003].

11   Conclusion

The vast literature on total order broadcast and the large number of algorithms published show the complexity of the problem. However, this abundance of information is a problem by itself, because it makes it difficult to understand the exact tradeoffs associated with each proposed solution.

The main contribution of this paper is the definition of a classification for total order broadcast algorithms, which makes it easier to understand the relationship between them. This also provides a good basis for comparing the algorithms and understanding some tradeoffs. Furthermore, the paper has presented a vast survey of most of the existing algorithms and discussed their respective characteristics.

In spite of the number of total order broadcast algorithms published, most of them are merely improvements or variants of each other (even if this is not immediately obvious for untrained eyes). Actually, there are only few truly original algorithms, but a large collection of various optimizations. Nevertheless, it is important to stress that clever optimizations of existing algorithms often have a very significant impact on performance. For instance, [Friedman and van Renesse 1997] show that piggybacking messages, in spite of its simplicity, can significantly improve the performance of algorithms.

Even though the specification of the total order broadcast problem dates back to some of the earliest publications about the subject, 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 and we hope that this paper will contribute to more rigorous work in the future. 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.

References

[Abdelzaher et al. 1996] ABDELZAHER, T., SHAIKH, A., JAHANIAN, F., AND SHIN, K. 1996. RTCAST: Lightweight multicast for real-time process groups. In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*. Boston, MA, USA, 250–259.

[Agarwal et al. 1998] AGARWAL, D. A., MOSER, L. E., MELLIAR-SMITH, P. M., AND BUDHIA, R. K. 1998. The Totem multiple-ring ordering and topology maintenance protocol. *ACM Trans. Comput. Syst. 16,* 2 (May), 93–132.

[Agrawal et al. 1997] AGRAWAL, D., ALONSO, G., EL ABBADI, A., AND STANOI, I. 1997. Exploiting atomic broadcast in replicated databases. In *Proc. EuroPar'97)*. Number 1300 in Lecture Notes in Computer Science. Passau, Germany, 496–503. Extended abstract.

[Aguilera et al. 1997] AGUILERA, M. K., CHEN, W., AND TOUEG, S. 1997. Quiescent reliable communication and quiescent consensus in partitionable networks. TR 97-1632, Dept. of Computer Science, Cornell University, Ithaca, NY, USA. June.

[Aguilera et al. 2000] AGUILERA, M. K., CHEN, W., AND TOUEG, S. 2000. Failure detection and consensus in the crash-recovery model. *Distr. Comput. 13,* 2, 99–125.

[Aguilera et al. 2000] AGUILERA, M. K., DELPORTE-GALLET, C., FAUCONNIER, H., AND TOUEG, S. 2000. Thrifty generic broadcast. In *Proc. 14th Intl. Symp. on Distributed Computing (DISC'00)*, M. Herlihy, Ed. LNCS, vol. 1914. Toledo, Spain, 268–282.

[Aguilera and Strom 2000] AGUILERA, M. K. AND STROM, R. E. 2000. Efficient atomic broadcast using deterministic merge. In *Proc. 19th Annual ACM Intl. Symp. on Principles of Distributed Computing (PODC-19)*. Portland, OR, USA, 209–218.

[Amir et al. 1992] AMIR, Y., DOLEV, D., KRAMER, S., AND MALKI, D. 1992. Transis: A communication sub-system for high availability. In *Proc. 22nd Intl. Symp. on Fault-Tolerant Computing (FTCS-22)*. Boston, MA, USA, 76–84.

[Amir et al. 1995] AMIR, Y., MOSER, L. E., MELLIAR-SMITH, P. M., AGARWAL, D. A., AND CIARFELLA, P. 1995. The Totem single-ring ordering and membership protocol. *ACM Trans. Comput. Syst. 13,* 4 (Nov.), 311–342.

[Anceaume 1993a] ANCEAUME, E. 1993a. Algorithmique de fiabilisation de systèmes répartis. Ph.D. thesis, Université de Paris-sud (Paris XI), Paris, France.

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

[Anceaume 1997] ANCEAUME, E. 1997. A lightweight solution to uniform atomic broadcast for asynchronous systems. In *Proc. 27th Intl. Symp. on Fault-Tolerant Computing (FTCS-27)*. Seattle, WA, USA, 292–301.

[Anceaume and Minet 1992] ANCEAUME, E. AND MINET, P. 1992. Étude de protocoles de diffusion atomique. TR 1774, INRIA, Rocquencourt, France. Oct.

[Armstrong et al. 1992] ARMSTRONG, S., FREIER, A., AND MARZULLO, K. 1992. Multicast transport protocol. RFC 1301, IETF. Feb.

[Attiya and Welch 1994] ATTIYA, H. AND WELCH, J. L. 1994. Sequential consistency versus linearizability. *ACM Trans. Comput. Syst. 12,* 2 (May), 91–122.

[Bar-Joseph et al. 2000] BAR-JOSEPH, Z., KEIDAR, I., ANKER, T., AND LYNCH, N. 2000. QoS preserving totally ordered multicast. In *Proc. 4th Intl. Conf. on Principles of Distributed Systems (OPODIS)*. Studia Informatica, Paris, France, 143–162.

[Bar-Joseph et al. 2002] BAR-JOSEPH, Z., KEIDAR, I., AND LYNCH, N. 2002. Early-delivery dynamic atomic broadcast. In *Proc. 16th Intl. Symp. on Distributed Computing (DISC'02)*, D. Malkhi, Ed. LNCS, vol. 2508. Toulouse, France, 1–16.

[Basu et al. 1996] BASU, A., CHARRON-BOST, B., AND TOUEG, S. 1996. Simulating reliable links with unreliable links in the presence of process crashes. In *Proc. 10th Intl. Workshop on Distributed Algorithms (WDAG'96)*. Lecture Notes in Computer Science, vol. 1151. Springer-Verlag, Bologna, Italy, 105–122.

[Ben-Or 1983] BEN-OR, M. 1983. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. Second ACM Symp. on Principles of Distributed Computing (PODC-2)*. Montreal, Quebec, Canada, 27–30.

[Berman and Bharali 1993] BERMAN, P. AND BHARALI, A. A. 1993. Quick atomic broadcast. In *Proc. 7th Intl. Workshop on Distributed Algorithms (WDAG)*. Number 725 in Lecture Notes in Computer Science. Lausanne, Switzerland, 189–203. Extended abstract.

[Bernstein et al. 1987] BERNSTEIN, P., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley. http://research.microsoft.com/pubs/ccontrol/.

[Bhatti et al. 1998] BHATTI, N. T., HILTUNEN, M. A., SCHLICHTING, R. D., AND CHIU, W. 1998. Coyote: A system for constructing fine-grain configurable communication services. *ACM Trans. Comput. Syst. 16,* 4 (Nov.), 321–366.

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

[Birman and Joseph 1987] BIRMAN, K. P. AND JOSEPH, T. A. 1987. Reliable communication in the presence of failures. *ACM Trans. Comput. Syst. 5,* 1 (Feb.), 47–76.

[Birman et al. 1991] BIRMAN, K. P., SCHIPER, A., AND STEPHENSON, P. 1991. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst. 9,* 3 (Aug.), 272–314.

[Birman and van Renesse 1993] BIRMAN, K. P. AND VAN RENESSE, R. 1993. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press.

[Carr 1985] CARR, R. 1985. The Tandem global update protocol. *Tandem Systems Review 1,* 2 (June), 74–85.

[Chandra et al. 1996] CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. 1996. The weakest failure detector for solving consensus. *J. ACM 43,* 4 (July), 685–722.

[Chandra and Toueg 1996] CHANDRA, T. D. AND TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *J. ACM 43,* 2, 225–267.

[Chang and Maxemchuk 1984] CHANG, J.-M. AND MAXEMCHUK, N. F. 1984. Reliable broadcast protocols. *ACM Trans. Comput. Syst. 2,* 3 (Aug.), 251–273.

[Charron-Bost et al. 1999] CHARRON-BOST, B., PEDONE, F., AND DÉFAGO, X. 1999. 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.

[Charron-Bost et al. 2000] CHARRON-BOST, B., TOUEG, S., AND BASU, A. 2000. Revisiting safety and liveness in the context of failures. In *CONCUR.* Number 1877 in Lecture Notes in Computer Science. 552–565.

[Chen et al. 1996] CHEN, X., MOSER, L. E., AND MELLIAR-SMITH, P. M. 1996. Reservation-based totally ordered multicasting. In *Proc. 16th Intl. Conf. on Distributed Computing Systems (ICDCS-16).* Hong Kong, 511–519.

[Cheriton and Skeen 1993] CHERITON, D. R. AND SKEEN, D. 1993. Understanding the limitations of causally and totally ordered communication. In *Proc. 14th ACM Symp. on Operating Systems Principles (SoSP-14).* Ashville, NC, USA, 44–57.

[Chiu and Hsiao 1998] CHIU, G.-M. AND HSIAO, C.-M. 1998. A note on total ordering multicast using propagation trees. *IEEE Trans. Parall. Distrib. Syst. 9,* 2 (Feb.), 217–223.

[Chockler et al. 2001] CHOCKLER, G., KEIDAR, I., AND VITENBERG, R. 2001. Group communication specifications: A comprehensive study. *ACM Comput. Surv. 33,* 4 (Dec.), 1–43.

[Chockler et al. 1998] CHOCKLER, G. V., HULEIHEL, N., AND DOLEV, D. 1998. An adaptive total ordered multicast protocol that tolerates partitions. In *Proc. 17th ACM Symp. on Principles of Distributed Computing (PODC-17).* Puerto Vallarta, Mexico, 237–246.

[Chockler et al. 2001] CHOCKLER, G. V., MALKHI, D., AND REITER, M. K. 2001. Backoff protocols for distributed mutual exclusion and ordering. In *Proc. 21st IEEE Intl. Conf. on Distributed Computing Systems (ICDCS-21).* Mesa, AZ, USA, 11–20.

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

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

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

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

[Cristian et al. 1995] CRISTIAN, F., AGHILI, H., STRONG, R., AND DOLEV, D. 1995. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Inf. Comput. 118,* 1, 158–179.

[Cristian et al. 1994] CRISTIAN, F., DE BEIJER, R., AND MISHRA, S. 1994. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering 1,* 4 (June), 177–201.

[Cristian and Fetzer 1999] CRISTIAN, F. AND FETZER, C. 1999. The timed asynchronous distributed system model. *IEEE Trans. Parall. Distrib. Syst. 10,* 6 (June), 642–657.

[Cristian and Mishra 1995] CRISTIAN, F. AND MISHRA, S. 1995. The pinwheel asynchronous atomic broadcast protocols. In *Proc. 2nd Intl. Symp. on Autonomous Decentralized Systems.* Phoenix, AZ, USA, 215–221.

[Cristian et al. 1997] CRISTIAN, F., MISHRA, S., AND ALVAREZ, G. 1997. High-performance asynchronous atomic broadcast. *Distributed System Engineering Journal 4,* 2 (June), 109–128.

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

[Défago et al. 2003] DÉFAGO, X., SCHIPER, A., AND URBÁN, P. 2003. Comparative performance analysis of ordering strategies in atomic broadcast algorithms. *IEICE Trans. on Information and Systems.* Accepted for publication.

[Delporte-Gallet and Fauconnier 1999] DELPORTE-GALLET, C. AND FAUCONNIER, H. 1999. Real-time fault-tolerant atomic broadcast. In *Proc. 18th Symp. on Reliable Distributed Systems (SRDS).* Lausanne, Switzerland, 48–55.

[Delporte-Gallet and Fauconnier 2000] DELPORTE-GALLET, C. AND FAUCONNIER, H. 2000. Fault-tolerant genuine Atomic Broadcast to multiple groups. In *Proc. 4th Intl. Conf. on Principles of Distributed Systems (OPODIS).* Studia Informatica, Paris, France, 143–162.

[Dolev et al. 1987] DOLEV, D., DWORK, C., AND STOCKMEYER, L. 1987. On the minimal synchrony needed for distributed consensus. *J. ACM 34,* 1 (Jan.), 77–97.

[Dolev et al. 1993] DOLEV, D., KRAMER, S., AND MALKI, D. 1993. Early delivery totally ordered multicast in asynchronous environments. In *Proc. 23rd Intl. Symp. on Fault-Tolerant Computing (FTCS-23).* Toulouse, France, 544–553.

[Dolev and Malkhi 1994] DOLEV, D. AND MALKHI, D. 1994. The design of the Transis system. In *Theory and Practice in Distributed Systems*, K. Birman, F. Mattern, and A. Schiper, Eds. Lecture Notes in Computer Science, vol. 938. Springer-Verlag, Dagstuhl Castle, Germany, 83–98.

[Dolev and Malkhi 1996] DOLEV, D. AND MALKHI, D. 1996. The Transis approach to high availability cluster communnication. *Commun. ACM 39,* 4 (Apr.), 64–70.

[Dwork et al. 1988] DWORK, C., LYNCH, N. A., AND STOCKMEYER, L. 1988. Consensus in the presence of partial synchrony. *J. ACM 35,* 2 (Apr.), 288–323.

[Ezhilchelvan et al. 1995] EZHILCHELVAN, P. D., MACÊDO, R. A., AND SHRIVASTAVA, S. K. 1995. Newtop: A fault-tolerant group communication protocol. In *Proc. 15th Intl. Conf. on Distributed Computing Systems (ICDCS-15).* Vancouver, Canada, 296–306.

[Fekete 1993] FEKETE, A. 1993. Formal models of commuication services: A case study. *IEEE Computer 26,* 8 (Aug.), 37–47.

[Fekete et al. 2001] FEKETE, A., LYNCH, N., AND SHVARTSMAN, A. 2001. Specifying and using a partitionable group communication service. *ACM Trans. Comput. Syst. 19,* 2 (May), 171–216.

[Felber and Pedone 2002] FELBER, P. AND PEDONE, F. 2002. Probabilistic atomic broadcast. In *Proc. 21st IEEE Intl. Symp. on Reliable Distributed Systems (SRDS-21).* Suita, Japan, 170–179.

[Felber and Schiper 2001] FELBER, P. AND SCHIPER, A. 2001. Optimistic Active Replication. In *IEEE 21st Intl. Conf. Distributed Computing Systems.* 333–341.

[Fetzer 2003] FETZER, C. 2003. Perfect failure detection in timed asynchronous systems. *IEEE Trans. Comput. 52,* 2 (Feb.), 99–112.

[Fischer et al. 1985] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *J. ACM 32,* 2 (Apr.), 374–382.

[Friedman and van Renesse 1997] FRIEDMAN, R. AND VAN RENESSE, R. 1997. Packing messages as a tool for boosting the performance of total ordering protocols. In *6th IEEE Symp. on High Performance Distributed Computing.* Portland, OR, USA, 233–242.

[Fritzke et al. 2001] FRITZKE, U., INGELS, P., MOSTEFAOUI, A., AND RAYNAL, M. 2001. Consensus-based fault-tolerant total order multicast. *IEEE Trans. Parall. Distrib. Syst. 12,* 2 (Feb.), 147–156.

[Garcia-Molina and Spauster 1989] GARCIA-MOLINA, H. AND SPAUSTER, A. 1989. Message ordering in a multicast environment. In *Proc. 9th Intl. Conf. on Distributed Computing Systems (ICDCS-9).* Newport Beach, CA, USA, 354–361.

[Garcia-Molina and Spauster 1991] GARCIA-MOLINA, H. AND SPAUSTER, A. 1991. Ordered and reliable multicast communication. *ACM Trans. Comput. Syst. 9,* 3 (Aug.), 242–271.

[Gopal and Toueg 1989] GOPAL, A. AND TOUEG, S. 1989. Reliable broadcast in synchronous and asynchronous environment. In *Proc. 3rd Intl. Workshop on Distributed Algorithms (WDAG).* Number 392 in Lecture Notes in Computer Science. Nice, France, 111–123.

[Gopal and Toueg 1991] GOPAL, A. AND TOUEG, S. 1991. Inconsistency and contamination. In *Proc. 10th ACM Symp. on Principles of Distributed Computing (PODC-10).* Proc. 10th ACM Symp. on Principles of Distributed Computing (PODC-10), 257–272.

[Guerraoui 1995] GUERRAOUI, R. 1995. Revisiting the relationship between non-blocking atomic commitment and consensus. In *Proc. 9th Intl. Workshop on Distributed Algorithms (WDAG-9).* LNCS 972. Springer-Verlag, Le Mont-St-Michel, France, 87–100.

[Guerraoui and Schiper 1997] GUERRAOUI, R. AND SCHIPER, A. 1997. Genuine atomic multicast. In *Proc. 11th Intl. Workshop on Distributed Algorithms (WDAG).* Number 1320 in Lecture Notes in Computer Science. Saarbrücken, Germany, 141–154.

[Guerraoui and Schiper 2001] GUERRAOUI, R. AND SCHIPER, A. 2001. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci. 254,* 297–316.

[Guy et al. 1993] GUY, R. G., POPEK, G. J., AND PAGE JR., T. W. 1993. Consistency algorithms for optimistic replication. In *Proc. 1st IEEE Intl. Conf. on Network Protocols (ICNP'93).*

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

[Hiltunen et al. 1999] HILTUNEN, M. A., SCHLICHTING, R. D., HAN, X., CARDOZO, M. M., AND DAS, R. 1999. Real-time dependable channels: Customizing QoS attributes for distributed systems. *IEEE Trans. Parall. Distrib. Syst. 10,* 6 (June), 600–612.

[Jalote 1998] JALOTE, P. 1998. Efficient ordered broadcasting in reliable CSMA/CD networks. In *Proc. 18th Intl. Conf. on Distributed Computing Systems (ICDCS-18).* Amsterdam, The Netherlands, 112–119.

[Jia 1995] JIA, X. 1995. A total ordering multicast protocol using propagation trees. *IEEE Trans. Parall. Distrib. Syst. 6,* 6 (June), 617–627.

[Kaashoek and Tanenbaum 1996] KAASHOEK, M. F. AND TANENBAUM, A. S. 1996. An evaluation of the Amoeba group communication system. In *Proc. 16th Intl. Conf. on Distributed Computing Systems (ICDCS-16).* Hong Kong, 436–447.

[Keidar and Dolev 2000] KEIDAR, I. AND DOLEV, D. 2000. Totally ordered broadcast in the face of network partitions. In *Dependable Network Computing*, D. Avresky, Ed. Kluwer Academic Publications, Chapter 3, 51–75.

[Kemme and Alonso 2000] KEMME, B. AND ALONSO, G. 2000. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases*. Morgan Kaufmann, 134–143.

[Kemme et al. 1999] KEMME, B., PEDONE, F., ALONSO, G., AND SCHIPER, A. 1999. Processing transactions over optimistic atomic broadcast protocols. In *Proc. 19th Intl. Conf. on Distributed Computing Systems (ICDCS-19)*. Austin, TX, USA, 424–431.

[Kemme et al. 2003] KEMME, B., PEDONE, F., ALONSO, G., SCHIPER, A., AND WIESMANN, M. 2003. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng. 15,* 4 (July), 1018–1032.

[Kim and Kim 1997] KIM, J. AND KIM, C. 1997. A total ordering protocol using a dynamic token-passing scheme. *Distrib. Syst. Eng. 4,* 2 (June), 87–95.

[Kopetz et al. 1990] KOPETZ, H., GRÜNSTEIDL, G., AND REISINGER, J. 1990. Fault-tolerant membership service in a synchronous distributed real-time system. In *Dependable Computing for Critical Applications (DCCA)*, A. Avižienis and J.-C. Laprie, Eds. Vol. 4. 411–429.

[Lamport 1978a] LAMPORT, L. 1978a. The implementation of reliable distributed multiprocess systems. *Computer Networks 2*, 95–114.

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

[Lamport 1984] LAMPORT, L. 1984. Using time instead of time-outs in fault-tolerant systems. *ACM Trans. Program. Lang. Syst. 6,* 2, 256–280.

[Lamport 1986a] LAMPORT, L. 1986a. The mutual exclusion problem: Part I—a theory of interprocess communication. *J. ACM 33,* 2 (Apr.), 313–326.

[Lamport 1986b] LAMPORT, L. 1986b. On interprocess communication. part I: Basic formalism. *Distributed Computing 1,* 2, 77–85.

[Lamport et al. 1982] LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4,* 3, 382–401.

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

[Liu et al. 2001] LIU, X., VAN RENESSE, R., BICKFORD, M., KREITZ, C., AND CONSTABLE, R. 2001. Protocol switching: Exploiting meta-properties. In *Proc. 21st IEEE Intl. Conf. on Distributed Computing Systems Workshops (ICDCSW'01), Intl. Workshop on Applied Reliable Group Communication (WARGC 2001)*. Mesa, AZ, USA, 37–42.

[Luan and Gligor 1990] LUAN, S.-W. AND GLIGOR, V. D. 1990. A fault-tolerant protocol for atomic broadcast. *IEEE Trans. Parall. Distrib. Syst. 1,* 3 (July), 271–285.

[Lynch 1996] LYNCH, N. A. 1996. *Distributed Algorithms*. Morgan Kaufmann.

[Lynch and Tuttle 1989] LYNCH, N. A. AND TUTTLE, M. R. 1989. An introduction to input/output automata. *CWI Quarterly 2,* 3 (Sept.), 219–246.

[Malhis et al. 1996] MALHIS, L. M., SANDERS, W. H., AND SCHLICHTING, R. D. 1996. Numerical performability evaluation of a group multicast protocol. *Distrib. Syst. Eng. 3,* 1 (Mar.), 39–52.

[Malloth 1996] MALLOTH, C. P. 1996. Conception and implementation of a toolkit for building fault-tolerant distributed applications in large scale networks. Ph.D. thesis, École Polytechnique Fédérale de Lausanne, Switzerland. Number 1557.

[Malloth et al. 1995] MALLOTH, C. P., FELBER, P., SCHIPER, A., AND WILHELM, U. 1995. 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.

[Mayer 1992] MAYER, E. 1992. An evaluation framework for multicast ordering protocols. In *Proc. Conf. on Applications, Technologies, Architecture, and Protocols for Computer Communication (SIGCOMM)*. 177–187.

[Minet and Anceaume 1991a] MINET, P. AND ANCEAUME, E. 1991a. ABP: An atomic broadcast protocol. Tech. Rep. 1473, INRIA, Rocquencourt, France. June.

[Minet and Anceaume 1991b] MINET, P. AND ANCEAUME, E. 1991b. Atomic broadcast in one phase. *ACM Operating Systems Review 25,* 2 (Apr.), 87–90.

[Mishra et al. 1993] MISHRA, S., PETERSON, L. L., AND SCHLICHTING, R. D. 1993. Consul: a communication substrate for fault-tolerant distributed programs. *Distrib. Syst. Eng. 1,* 2, 87–103.

[Moser and Melliar-Smith 1999] MOSER, L. E. AND MELLIAR-SMITH, P. M. 1999. Byzantine-resistant total ordering algorithms. *Inf. Comput. 150,* 1 (Apr.), 75–111.

[Moser et al. 1996] MOSER, L. E., MELLIAR-SMITH, P. M., AGRAWAL, D. A., BUDHIA, R. K., AND LINGLEY-PAPADOPOULOS, C. A. 1996. Totem: A fault-tolerant multicast group communication system. *Commun. ACM 39,* 4 (Apr.), 54–63.

[Moser et al. 1993] MOSER, L. E., MELLIAR-SMITH, P. M., AND AGRAWALA, V. 1993. Asynchronous fault-tolerant total ordering algorithms. *SIAM J. Comput. 22,* 4 (Aug.), 727–750.

[Mostefaoui and Raynal 2000] MOSTEFAOUI, A. AND RAYNAL, M. 2000. Low cost consensus-based atomic broadcast. In *Proc. 7th IEEE Pacific Rim Symp. on Dependable Computing (PRDC'00)*. Los Angeles, CA, USA, 45–52.

[Navaratnam et al. 1988] NAVARATNAM, S., CHANSON, S. T., AND NEUFELD, G. W. 1988. Reliable group communication in distributed systems. In *Proc. 8th Intl. Conf. on Distributed Computing Systems (ICDCS-8)*. San Jose, CA, USA, 439–446.

[Neiger and Toueg 1990] NEIGER, G. AND TOUEG, S. 1990. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms 11,* 3, 374–419.

[Nestmann et al. 2003] NESTMANN, U., FUZZATI, R., AND MERRO, M. 2003. Modeling consensus in a process calculus. In *Proc. CONCUR 2003*. To appear.

[Ng 1991] NG, T. P. 1991. Ordered broadcasts for large applications. In *Proc. 10th Symp. on Reliable Distributed Systems (SRDS)*. Pisa, Italy, 188–197.

[Pedone 2001] PEDONE, F. 2001. Boosting system performance with optimistic distributed protocols. *IEEE Computer 34,* 12 (Dec.), 80–86.

[Pedone et al. 1998] PEDONE, F., GUERRAOUI, R., AND SCHIPER, A. 1998. Exploiting atomic broadcast in replicated databases. In *Proc. of EuroPar (EuroPar'98)*. Number 1470 in Lecture Notes in Computer Science. Southampton, UK, 513–520.

[Pedone and Schiper 1998] PEDONE, F. AND SCHIPER, A. 1998. Optimistic atomic broadcast. In *Proc. 12th Intl. Symp. on Distributed Computing (DISC)*. Number 1499 in Lecture Notes in Computer Science. Andros, Greece, 318–332.

[Pedone and Schiper 1999] PEDONE, F. AND SCHIPER, A. 1999. Generic broadcast. In *Proc. 13th Intl. Symp. on Distributed Computing (DISC)*. Number 1693 in Lecture Notes in Computer Science. Bratislava, Slovak Republic, 94–108.

[Pedone and Schiper 2002] PEDONE, F. AND SCHIPER, A. 2002. Handling message semantics with Generic Broadcast protocols. *Distributed Computing 15,* 2, 97–107.

[Pedone and Schiper 2003] PEDONE, F. AND SCHIPER, A. 2003. Optimistic atomic broadcast: A pragmatic viewpoint. *Theor. Comput. Sci. 291,* 79–101.

[Pedone et al. 2002] PEDONE, F., SCHIPER, A., URBÁN, P., AND CAVIN, D. 2002. Solving agreement problems with weak ordering oracles. In *Proc. 4th European Dependable Computing Conference (EDCC-4).* LNCS, vol. 2485. Springer-Verlag, Toulouse, France, 44–61.

[Peterson et al. 1989] PETERSON, L. L., BUCHHOLZ, N. C., AND SCHLICHTING, R. D. 1989. Preserving and using context information in interprocess communication. *ACM Trans. Comput. Syst. 7,* 3, 217–246.

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

[Rabin 1983] RABIN, M. 1983. Randomized Byzantine generals. In *Proc. 24th Annual ACM Symp. on Foundations of Computer Science (FOCS).* Tucson, AZ, USA, 403–409.

[Rajagopalan and McKinley 1989] RAJAGOPALAN, B. AND MCKINLEY, P. 1989. A token-based protocol for reliable, ordered multicast communication. In *Proc. 8th Symp. on Reliable Distributed Systems (SRDS).* Seattle, WA, USA, 84–93.

[Reiter 1994] REITER, M. K. 1994. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proc. 2nd ACM Conf. on Computer and Communications Security (CCS-2).* 68–80.

[Reiter 1996] REITER, M. K. 1996. Distributing trust with the Rampart toolkit. *Commun. ACM 39,* 4 (Apr.), 71–74.

[Rodrigues et al. 1996] RODRIGUES, L., FONSECA, H., AND VERÍSSIMO, P. 1996. Totally ordered multicast in large-scale systems. In *Proc. 16th Intl. Conf. on Distributed Computing Systems (ICDCS-16).* Hong Kong, 503–510.

[Rodrigues et al. 1998] RODRIGUES, L., GUERRAOUI, R., AND SCHIPER, A. 1998. Scalable atomic multicast. In *Proc. 7th IEEE Intl. Conf. on Computer Communications and Networks.* Lafayette (LA), USA, 840–847.

[Rodrigues and Raynal 2000] RODRIGUES, L. AND RAYNAL, M. 2000. Atomic broadcast in asynchronous crash-recovery distributed systems. In *Proc. 20th IEEE Intl. Conf. on Distributed Computing Systems (ICDCS-20).* Taipei, Taiwan, 288–295.

[Rodrigues and Veríssimo 1992] RODRIGUES, L. AND VERÍSSIMO, P. 1992. *x*AMP: a multi-primitive group communications service. In *Proc. 11th Symp. on Reliable Distributed Systems (SRDS).* Houston, TX, USA, 112–121.

[Rodrigues et al. 1993] RODRIGUES, L., VERÍSSIMO, P., AND CASIMIRO, A. 1993. Using atomic broadcast to implement a posteriori agreement for clock synchronization. In *Proc. 12th Symp. on Reliable Distributed Systems (SRDS).* Princeton, NJ, USA, 115–124.

[Schiper and Raynal 1996] SCHIPER, A. AND RAYNAL, M. 1996. From group communication to transactions in distributed systems. *Commun. ACM 39,* 4 (Apr.), 84–87.

[Schneider 1990] SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv. 22,* 4 (Dec.), 299–319.

[Shieh and Ho 1997] SHIEH, S.-P. AND HO, F.-S. 1997. A comment on "a total ordering multicast protocol using propagation trees". *IEEE Trans. Parall. Distrib. Syst. 8,* 10 (Oct.), 1084.

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

[Sousa et al. 2002] SOUSA, A., PEREIRA, J., MOURA, F., AND OLIVEIRA, R. 2002. Optimistic total order in wide area networks. In *Proc. 21st IEEE Intl. Symp. on Reliable Distributed Systems (SRDS-21)*. Suita, Japan, 190–199.

[Toinard et al. 1999] TOINARD, C., FLORIN, G., AND CARREZ, C. 1999. A formal method to prove ordering properties of multicast algorithms. *ACM Operating Systems Review 33,* 4 (Oct.), 75–89.

[Urbán et al. 2000] URBÁN, P., DÉFAGO, X., AND SCHIPER, A. 2000. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. In *Proc. 9th IEEE Intl. Conf. on Computer Communications and Networks (IC3N 2000)*. Las Vegas, NE, USA, 582–589.

[Urbán et al. 2003] URBÁN, P., SHNAYDERMAN, I., AND SCHIPER, A. 2003. Comparison of failure detectors and group membership: Performan ce study of two atomic broadcast algorithms. In *Proc. Int'l Conf. on Dependable Systems and Networks*. San Francisco, CA, USA, 645–654.

[Veríssimo et al. 1989] VERÍSSIMO, P., RODRIGUES, L., AND BAPTISTA, M. 1989. AMp: A highly parallel atomic multicast protocol. *Computer Communication Review 19,* 4 (Sept.), 83–93. Proc. SIGCOMM'89 Symp.

[Veríssimo et al. 1997] VERÍSSIMO, P., RODRIGUES, L., AND CASIMIRO, A. 1997. CesiumSpray: A precise and accurate global clock service for large-scale systems. *Real-Time Systems 12,* 3 (May), 243–294.

[Vicente and Rodrigues 2002] VICENTE, P. AND RODRIGUES, L. 2002. An indulgent uniform total order algorithm with optimistic delivery. In *Proc. 21st IEEE Intl. Symp. on Reliable Distributed Systems (SRDS-21)*. Suita, Japan, 92–101.

[Whetten et al. 1994] WHETTEN, B., MONTGOMERY, T., AND KAPLAN, S. 1994. A high performance totally ordered multicast protocol. In *Theory and Practice in Distributed Systems*, K. Birman, F. Mattern, and A. Schiper, Eds. Number 938 in Lecture Notes in Computer Science. Dagstuhl Castle, Germany, 33–57.

[Wiesmann et al. 2000] WIESMANN, M., PEDONE, F., SCHIPER, A., KEMME, B., AND ALONSO, G. 2000. Understanding replication in databases and distributed systems. In *Proc. 20th Intl. Conf. on Distributed Computing Systems (ICDCS-20)*. Taipei, Taiwan, 264–274.

[Wilhelm and Schiper 1995] WILHELM, U. AND SCHIPER, A. 1995. A hierarchy of totally ordered multicasts. In *Proc. 14th Symp. on Reliable Distributed Systems (SRDS)*. Bad Neuenahr, Germany, 106–115.

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