A Modular Approach to Fault-Tolerant Broadcasts and Related Problems

Vassos Hadzilacos [*]
Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A1

Sam Toueg [†]
Department of Computer Science
Cornell University
Ithaca, New York 14853, USA

1 Introduction

The design and verification of fault-tolerant distributed applications is widely viewed as a complex endeavor. To a large extent this is due to the fact that the communication primitives available in distributed systems are too weak. For example, many systems support primitives that allow a process to send a message to only one other process. If a process p wishes to send a message m to all processes, it must do so by sending m to each one separately. Should p fail in the middle of this activity, it is possible that some of the processes receive m while others do not. Similar inconsistencies may arise even in networks such as Ethernet that support broadcast as a low-level communication primitive: failures can cause a message to be received by some processes but not by others. Such inconsistencies complicate the task of building fault-tolerant distributed software.

Fault-tolerant broadcasts are communication primitives that facilitate the development of fault-tolerant applications. The weakest among these is *Reliable Broadcast*. Roughly speaking, this allows processes to broadcast messages such that all processes agree on the set of messages they deliver, despite failures. Stronger variants of Reliable Broadcast impose additional requirements on the order in which messages are delivered. For example, processes may have to deliver all messages in the same order. Systems and applications based on fault-tolerant broadcasts include SIFT [WLG+78], State Machines [Lam78a,Sch90], Atomic Commitment [BT93], Isis [BJ87,BCJ+90], Psync [PBS89], Amoeba [Kaa92], Delta-4 [VM90], Transis [ADKM92], Highly Available System [Cri87], and Advanced Automation System [CDD90].

Another paradigm that simplifies the task of designing fault-tolerant distributed applications is *Consensus*. Roughly speaking, Consensus allows processes to reach a common

[*]Supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.
[†]Supported in part by NSF grants CCR-8901780 and CCR-9102231.

decision that depends on their initial inputs, despite failures. Consensus algorithms can be used to solve many problems such as electing a leader or agreeing on the value of a replicated sensor. Theoretical research on fault-tolerant distributed computing has largely centered on Consensus, while applied research has focused on Reliable Broadcast and its variants. In fact, as we shall see in this paper, the two problems are closely related.

Given their wide applicability, fault-tolerant broadcasts and Consensus have been extensively studied for over a decade. This has resulted in a voluminous literature which, unfortunately, is not distinguished for its coherence. The differences in notation and the haphazard nature of the assumptions obfuscates the close relationship among these problems.

Our primary goal here is to develop this material in a coherent way so as to expose its intrinsic unity. We also strive to make it as simple as possible, without sacrificing precision or rigor. Our approach consists of the following elements. First, we draw a sharp line between the *specification* of problems and the *algorithms* that solve them. Second, we give *modular* specifications of the various types of fault-tolerant broadcasts: Starting with Reliable Broadcast we define progressively stronger types of broadcasts by adding one or more requirements on the order of message delivery. Third, we also derive algorithms for each of these broadcasts in a modular way: We first give an algorithm for Reliable Broadcast, and then describe a set of transformations that can be used to convert weaker broadcast algorithms into stronger ones. Each of the transformations enforces one of the message delivery order requirements. In this way, the modular derivation of algorithms parallels our modular specification of broadcasts. A crucial and novel feature of our methodology is that these transformations are *generic:* they work for a large class of failures, and regardless of the type or synchrony of the communication network.

The rest of this paper is organized as follows. In Section 2 we describe commonly used models of computation. In Section 3 we give modular specifications for Reliable Broadcast and its variants. Our development of broadcast algorithms follows a particular methodology, explained in Section 4. In Section 5 we exhibit our generic transformations that can be used to convert weaker broadcast algorithms into stronger ones. In Section 6 we focus on a particular type of system, namely point-to-point networks, and derive Reliable Broadcast algorithms for such a system. Applying the transformations of Section 5 to these algorithms results in a suite of broadcast algorithms of various strengths for point-to-point networks. In Section 7, we first define the Consensus problem, and then determine its relation to Atomic Broadcast, one of the broadcasts that we consider here. In Section 8 we define a variant of Reliable Broadcast and explain its relation to the Consensus problem. A variation of broadcast where a process can target its messages to a specified subset of the processes is called a *multicast*. In Section 9, we give modular specifications for various types of fault-tolerant multicasts.

To enhance the readability of this paper, many references, historical notes, and other tangential and potentially distracting material are collected at the end of each section. Given the vastness of the literature on this subject, the bibliographic references are extensive, but incomplete. A more comprehensive treatment of the subject matter of this paper, including many results that are not presented here, will be found in the forthcoming book [HJT].

2 Preliminaries

Several computational models of distributed computing have been studied. Such models fall into two broad categories, *message-passing* and *shared-memory*. In the former, processes communicate by exchanging messages; in the latter, they communicate by accessing shared objects, such as registers, queues, etc. In this paper we focus on message-passing models.

The chief characteristics of a message-passing model are: the type of communication network, the model of process and communication failures that may occur, and the synchrony of the system.

2.1 Types of Communication Networks

The *type* of network determines how processes communicate. In a *point-to-point* network communication takes place over links that connect pairs of processes. In this type of network a process can send a message to a single process through a link. In a *broadcast channel* network communication takes place over a single shared channel that connects all processes. In such a network a process can broadcast a message to all other processes. Examples are Ethernet, Token Bus, Token Ring, and FDDI networks. Other types of networks include redundant broadcast channel networks (e.g., Delta-4 [VM90] and [Cri90b]), packet radio networks (e.g., ALOHA [Abr85]), switch-based networks (e.g., AN2 [Owi93]), etc.

Many of the results in this paper are independent of the type of communication network. When we need to focus on a particular type of network we concentrate on *point-to-point* ones. This is because this is the most basic type, in the sense that it supports the most limited communication primitives and can be easily emulated by the other types of networks.

We now turn our attention to the remaining characteristics of a message-passing system, namely types of failure and synchrony. Since these depend on the network type, we focus on point-to-point networks (similar definitions exist for other types of networks as well). Our treatment in this section is informal; a more precise and detailed exposition of point-to-point networks is given in the Appendix.

2.2 Point-to-Point Networks

A point-to-point network can be modeled as a directed graph, with nodes representing processes, and edges representing communication links between processes. In such a network, any pair of processes that are connected by a link can communicate with each other by sending and receiving messages, as described below.

Consider the link from a process p to a process q. Associated with this link are two communication primitives, called `send` and `receive`. If p invokes `send` with a message m as a parameter, we say that *p sends m to q*; when it returns from that invocation we say that *p completes the sending of m to q*. When a process q returns from the execution of `receive` with message m as the returned value, we say that *q receives m (from p)*.

Also associated with the link from p to q are an *outgoing message buffer* at p and an

incoming message buffer at q. Informally, when p sends a message m to q, p inserts m into its outgoing message buffer, the link transports m to q's incoming message buffer, and m is then removed and received by q. Thus, we assume that:

1. If p sends a message m to q then q eventually receives m.[1]

Every process p executes a sequence of *steps*, where each step is the execution of an operation such as the writing of a local variable, or the sending or receipt of a message. Associated with p is an *automaton*, whose transition relation defines the set of legal sequences of steps that p can execute. We assume that:

2. Every process executes an infinite sequence of steps.[2]

The preceding description of a point-to-point network assumes that no failures occur. In the next section we consider some of the failures that can affect processes and links.

2.3 Some Failure Types

Failures can be defined as deviations from correct behavior. To model the violation of Property 2, we introduce a special step called *crash*. Every process p can execute a crash at any time, and after doing so it stops executing further steps. This is modeled by the addition of a new terminal state to the automaton associated with p, and a transition from every other state of p to that terminal state. The step associated with such a transition is defined as a crash.

Following is a list of some types of *process failures* that have been studied:

- A process commits a *crash* failure if it executes a crash step.
- A process commits a *send-omission* failure on a message m if it completes the sending of m but m is not inserted into its outgoing message buffer.
- A process commits a *receive-omission* failure on a message m if m is inserted into its incoming message buffer but it does not receive m.
- A process commits an *arbitrary* (sometimes called *Byzantine* or *malicious*) failure if the sequence of steps that it executes deviates arbitrarily from the sequence prescribed by its associated automaton. Thus, it can exhibit any behavior whatsoever.

We also consider the following type of *link failure*:

[1] Note that messages are not necessarily received in the order in which they are sent.

[2] A distributed system in which a process p can terminate (i.e., p's automaton has a terminal state), can be modeled by replacing each terminal state of p by a state with a single "no-op" transition to itself.

- A link l from a process p to a process q commits an *omission* failure on a message m if m is inserted into p's outgoing buffer buffer but l does not transport m into q's incoming buffer.

If a process or a link commits a failure, we say that it is *faulty*; otherwise it is *correct*. In a network with failures, Properties 1 and 2 hold only for correct processes and for correct links that connect correct processes. In other words, they hold only in the subnetwork that consists entirely of correct processes and links.

2.4 Synchronous and Asynchronous Networks

A point-to-point network is *synchronous* if it has the following properties (in addition to the ones stated in Section 2.2):

3. There is a known upper bound on the time required by any process to execute a step.
4. Every process has a local clock with known bounded rate of drift with respect to real time.
5. There is a known upper bound on message delay; this consists of the time it takes to send, transport, and receive a message over any link.

It is important to realize that all of the above properties are necessary for the use of *timeouts* to detect crash failures. If any of the three properties is violated, and a process p times-out on a message expected from a process q, p cannot conclude that q has crashed: The message delay could have been longer than expected, the clock used by p to measure the timeout could have been running too fast, or q could be executing steps slower than expected.

A point-to-point network is *asynchronous* if there are *no* timing assumptions whatsoever. In particular, there are no assumptions on the maximum message delay, clock drift, or the time needed to execute a step. This model is attractive and has recently gained much currency for several reasons: It has simple semantics; applications programmed on the basis of this model are easier to port than those incorporating specific timing assumptions; and in practice, variable or unexpected workloads are sources of asynchrony — thus synchrony assumptions are at best probabilistic.

Synchronous and asynchronous point-to-point networks are the two extremes of a spectrum of possible models. Many intermediate models of *partial synchrony* have also been studied. For example, there may be known bounds on clock drift and step execution time, but message delays could be unbounded. Or there may be bounds on clock drift, step execution time *and* message delay, but these bounds may be unknown.

2.5 Clock and Performance Failures in Synchronous Networks

The failure types described in Section 2.3 apply to both synchronous and asynchronous networks. Certain failures, however, are only pertinent to synchronous networks; these

occur when the assumed bounds on clock drift, message delay, or the time needed to execute a step are violated. Such failures are defined below.

Consider a synchronous network, i.e., one where processes and links are supposed to satisfy timing Properties 3–5 (of Section 2.4). A process commits a *clock failure* if it violates Property 4; i.e., there is some interval of time during which its clock drifts with respect to real time at a rate that exceeds the specified maximum. A process commits a *performance failure* if it violates Property 3; i.e., it completes a step in more time than the specified maximum. Similarly, a link commits a *performance failure* if it transports some message in more time than its specified bound. Recall that the message delay consists of the time needed for the sender to send the message, the link to transport it, and the receiver to receive it. Thus, a violation of Property 5 may be due to a performance failure of the sender, the receiver, or the link between them.

In a synchronous network with clock and performance failures, the bounds on the time to execute a step and on clock drift apply only to correct processes. Similarly, the bound on message delay applies only to messages sent between correct processes over correct links. In other words, only the subnetwork consisting entirely of correct processes and links is really synchronous.

2.6 Classification of Failures and Terminology

It is convenient to group failures into two classes that include both process *and* link failures:

- *omission failures* consist of crash, send-omission, and receive-omission failures of processes, as well as link omission failures.
- *timing failures* consist of omission, clock and performance failures.

A network with a certain class of failures, is one where processes and links may commit any of the failures included in that class, but no other failures. Thus, a network with omission failures is not subject to clock, performance, or arbitrary failures. Similarly, one with timing failures is not subject to arbitrary failures.

Benign failures is synonymous to omission failures in asynchronous networks and to timing failures in synchronous networks. In a system with benign failures, processes do *not* commit arbitrary failures. Thus, the sequence of steps executed by every process, whether correct or faulty, is always consistent with the automaton associated with that process. In particular, a faulty process does not change its state arbitrarily, or send a message that it was not supposed to send — two behaviors allowed by arbitrary failures. Benign failures are the most common in practice, and in this paper we focus almost exclusively on them.

2.7 Causal Precedence

We can view the computation of a distributed system as a partial order on a set of steps that processes execute, including communication steps. For example, consider systems

where processes communicate by broadcasting and delivering messages. In such systems, a step is any operation executed by a process, such as the writing of a local variable, or the broadcast or delivery of a message. A given subset of steps ("the steps of interest") *induces* a partial order as follows. Step e *causally precedes* step f, denoted $e \rightarrow f$, if and only if:

1. the same process executes both e and f, in that order, or
2. e is the broadcast of some message m and f is the delivery of m, or
3. there is a step h, such that $e \rightarrow h$ and $h \rightarrow f$.

The causal precedence relation $\rightarrow$ is acyclic because the broadcast of a message always precedes (in real time) the delivery of that message. By Clause 3, $\rightarrow$ is also transitive, and hence a partial order.

This causal precedence relation plays a central role in distributed computing: In asynchronous message-passing systems, step e can "influence" step f only if there is a sequence of steps starting with e and ending with f such that consecutive steps are related as in (1) or (2) above, i.e., only if $e \rightarrow f$.

2.8 Properties of clocks

Even in asynchronous systems, processes may have access to local clocks. Although such clocks do not measure real time, they can still be useful by capturing some temporal information. The minimum requirement for a local clock is that its values are non-decreasing in real time. Of course, to be useful, local clocks must satisfy stronger properties. One such property is:

- *Clock Monotonicity:* The local clock of a process p (whether correct or faulty) never decreases or skips values. Furthermore, if p is correct then its clock eventually reaches c, for any time c.

Another useful property for clocks is that they be consistent with the causal precedence relation $\rightarrow$ defined in Section 2.7. More precisely, let p be any process and e be a step that occurs at p; let $C_p(e)$ denote the value of p's local clock when e occurs. We say that the local clocks are *consistent with* $\rightarrow$ if they satisfy the following property:

- *Logical Clocks:* For any processes p and q, and any steps e and f that occur at p and q, respectively, if $e \rightarrow f$ then $C_p(e) < C_q(f)$.

In other words, if e can influence f then the time at which e occurs is before the time at which f occurs according to the local clocks.

Recall that in a synchronous system the clocks of correct processes have a bounded *rate* of drift with respect to real time. As time progresses, however, the actual values of

the clocks may drift arbitrarily far apart. It turns out that in synchronous systems it is possible to implement *approximately synchronized* clocks, which not only have a bounded rate of drift with respect to real time, but also satisfy the following property:

- ϵ-*Synchronized Clocks:* The clock values of *correct* processes at any real time t differ by at most a known constant ϵ.

Note that such clocks may violate the Clock Monotonicity and Logical Clock properties. It is possible, however, to implement approximately synchronized clocks that satisfy both of these properties.

Bibliographic Notes

Crash failures, in the context of broadcast and Consensus problems, were first considered in [LF82]. [SS83] defined a more restricted type of process failure, referred to as *fail-stop*. A fail-stop process fails by crashing but, in addition, all correct processes are informed of the crash, and they have access to any information written by the faulty process in its stable storage before it crashed. Send-omission failures were introduced in [Had84]. [PT86] introduced the class of *general-omission* failures consisting of crash, send- and receive-omission failures. Timing failures were first considered in [CASD85], and arbitrary failures in [PSL80,LSP82]. [PSL80,LSP82] also consider *message authentication*, a mechanism that restricts the faulty behavior of processes that are subject to arbitrary failures. A precise definition of the properties of message authentication, and a mechanism for providing them without digital signatures in point-to-point networks appears in [ST87b].

There are methods for *automatically* increasing the fault-tolerance of algorithms. This is achieved by translations that transform any given algorithm tolerant of a certain type of failure into an algorithm that tolerates a more severe type of failure. Such translations are given in [Bra87,Coa87,ST87b,NT90,BN91,BN92]. They can be used to transform any algorithm tolerant of crash failures into one tolerant of arbitrary failures, in both synchronous and asynchronous systems.

Much of the theoretical work on fault-tolerant algorithms focused on synchronous models, including the so-called "synchronous round model" where processes execute in lock-step fashion. The asynchronous model has been adopted by many systems, such as those described in [BJ87,PBS89,ADKM92]. Theoretical investigation of this model was spurred by the surprising result of Fischer, Lynch and Paterson, stating that Consensus cannot be solved deterministically in this model [FLP85]. A variety of models of partial synchrony are studied in [DDS87,DLS88].

The causal precedence relation, which is widely viewed as one of the most fundamental concepts in distributed computing, was introduced by Lamport in a seminal paper [Lam78b]. (The relation was called "happens-before" in [Lam78b], and was defined in terms of send and receive communication primitives, instead of the broadcast and deliver primitives we used in our definition.) In that paper Lamport also defined logical clocks and showed how to implement them. [NT87,Wel87] show that, for a large class of problems, access to logical clocks

is as good as access to real-time clocks. Many clock synchronization algorithms are known, including [Lam78b,HSSD84,LM85,CAS86,DHS86,BD87,KO87,ST87a,WL88,Cri89].

3 Broadcast Specifications

Roughly speaking, *Reliable Broadcast* — the weakest type of of fault-tolerant broadcast that we consider — guarantees three properties: (1) all correct processes agree on the set of messages they deliver, (2) all messages broadcast by correct processes are delivered, and (3) no spurious messages are ever delivered. While these properties may be sufficient for some applications, Reliable Broadcast imposes no restriction on the *order* in which the messages are delivered. In some applications this order is important. Thus, we define a collection of stronger broadcasts, differing in the guarantees they provide on message delivery order.

Informally, *FIFO Broadcast* is a Reliable Broadcast that guarantees that messages broadcast by the same sender are delivered in the order they were broadcast. *Causal Broadcast*, a strengthening of FIFO Broadcast, requires that messages be delivered according to the *causal precedence* relation discussed in Section 2.7: Roughly speaking, if the broadcast of m causally precedes the broadcast of m', then m must be delivered before m'. If two messages are *not* causally related, however, different processes can deliver them in different orders. *Atomic Broadcast* prevents this undesirable behavior by requiring processes to deliver all messages in the same order. Finally, *FIFO Atomic Broadcast* combines the requirements of FIFO Broadcast and Atomic Broadcast, and *Causal Atomic Broadcast* combines the requirements of Causal Broadcast and Atomic Broadcast.

In our definitions of the various types of broadcast, we assume that we are only dealing with benign failures. This not only simplifies the definitions, but also makes it possible to strengthen the properties of broadcasts in ways that are important in practice. In Section 3.10, we shall describe the few modifications necessary for arbitrary failures.

3.1 Reliable Broadcast

Informally, Reliable Broadcast requires that all correct processes deliver the same set of messages (Agreement), and that this set include all the messages broadcast by correct processes (Validity) but no spurious messages (Integrity). Formally, Reliable Broadcast is defined in terms of two primitives: `broadcast` and `deliver`. When a process p invokes `broadcast` with a message m as a parameter, we say that p *broadcasts* m. We assume that m is taken from a set a set $\mathcal{M}$ of possible messages. When a process q returns from the execution of `deliver` with message m as the returned value, we say that q *delivers* m.

Since every process can broadcast several messages, it is important to be able to determine the identity of a message's sender, and to distinguish the different messages broadcast by a particular sender. Thus, we assume that every message m includes the following fields: the identity of its sender, denoted $sender(m)$, and a sequence number, denoted $seq\#(m)$. If $sender(m) = p$ and $seq\#(m) = i$, then m is the ith message broadcast by p. These fields make every message unique.

Reliable Broadcast is a broadcast that satisfies the following three properties:

- *Validity:* If a correct process broadcasts a message m, then it eventually delivers m.
- *Agreement:* If a correct process delivers a message m, then all correct processes eventually deliver m.
- *Integrity:* For any message m, every correct process delivers m at most once, and only if m was previously broadcast by $sender(m)$.

Validity together with Agreement ensures that a message broadcast by a correct process is delivered by all correct processes. It is important to realize that if the sender of a message m is faulty, the specification of Reliable Broadcast allows two possible outcomes: either m is delivered by all correct processes or by none. For example, if a process p crashes immediately after invoking `broadcast`(m), correct processes will never be aware of p's intention to broadcast m, and thus cannot deliver anything. On the other hand, if p invokes `broadcast`(m) and fails during the execution of this primitive after having sent enough information about m, then correct processes may be able to deliver m.

3.2 FIFO Broadcast

In general, each message has a context without which it may be misinterpreted. Such a message should not be delivered by a process that does not know its context. In some applications, the context of a message m consists of the messages previously broadcast by the sender of m. For example, in an airline reservation system, the context of a message cancelling a reservation consists of the message that previously established that reservation: the cancellation message should not be delivered at a site that has not yet "seen" the reservation message. Such applications require the semantics of *FIFO Broadcast*, a Reliable Broadcast that satisfies the following requirement on message delivery:

- *FIFO Order:* If a process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.

Our definition of FIFO Order is subtler than meets the eye. Some alternative formulations of FIFO Broadcast which have appeared in the literature have a similar flavor, but are ambiguous or do not fully capture the desirable property described above. For example, consider the following definition: "all messages broadcast by the same process are delivered to all processes in the order they are sent." Suppose process p broadcasts messages m_1, m_2, and m_3 in that order, and correct process q delivers m_1 and then m_3 (but never delivers m_2). This scenario could happen if p suffers a transient failure while broadcasting m_2. Note that m_3 was delivered without its proper context, namely m_2. This undesirable behavior is allowed by the alternative definition (since m_1 and m_3 are indeed delivered in the order they are broadcast), but not by our definition of FIFO Order.[3]

[3]This alternative definition, taken from the literature, is also flawed in another way: it requires messages be delivered by *all* processes. Clearly, this is impossible, since a faulty process cannot be forced to deliver any message.

3.3 Causal Broadcast

FIFO Order is adequate when the context of a message m consists only of the messages that the sender of m broadcast before m. A message m, however, may also depend on messages that the sender of m *delivered* before broadcasting m. In this case, the message delivery order guaranteed by FIFO Broadcast is not sufficient. For example, in a network news application, if users distribute their articles with FIFO Broadcast, the following undesirable scenario could occur. User A broadcasts an article. User B, at a different site, delivers that article and broadcasts a response that can only be understood by a user who has already seen the original article. User C delivers B's response before delivering the original article from A and so misinterprets the response. Causal Broadcast is a strengthening of FIFO Broadcast that prevents the above problem by generalizing the notion of a message "depending" on another one, and ensuring that a message is not delivered until all the messages it depends on have been delivered. We capture this more general notion of dependence with the *causal precedence* relation on message broadcasts and deliveries defined in Section 2.7.

Given a causal precedence relation (induced by broadcasts and deliveries), we define *Causal Broadcast* to be a Reliable Broadcast that satisfies:

- *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.

The following alternative formulation of Causal Order has appeared in the literature: if the broadcast of m causally precedes the broadcast of m', then every correct process *that delivers both messages* must deliver m before m'. In a system with failures, this definition of Causal Order is flawed. In fact, it allows the same non-FIFO execution described in the previous section, where a faulty process broadcasts m_1, m_2, and m_3, and a correct process delivers m_1 and then m_3. This alternative definition also allows the following undesirable scenario from our network news example. Faulty user A broadcasts an article; faulty user B, who is the only one to deliver that message, broadcasts a response. Correct user C delivers B's response, although it never delivers A's original article. It is easy to see that this scenario satisfies all the properties of Reliable Broadcast, namely Validity, Agreement and Integrity, as well as the alternative definition of Causal Order (but not our definition of Causal Order). Note that defining Causal Order as "messages that are causally related are delivered in the causal order" is also flawed.

Causal Order is a generalization of FIFO Order; in fact, as we show below it is equivalent to the conjunction of FIFO Order and the following property:

- *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.

Theorem 1 *Causal Order is equivalent to FIFO Order and Local Order.*

`Proof:` It is obvious that Causal Order implies FIFO Order and Local Order. We now show that FIFO Order and Local Order imply Causal Order. Let m and m' be messages such

that the broadcast of m causally precedes the broadcast of m'. Consider any correct process p that delivers m'. We must show that p delivers m before m'. If m and m' are broadcast by the same process, this follows immediately by FIFO Order. Now assume that m and m' are broadcast by different processes. From the definition of the causal precedence relation it is easy to see that there exist processes $p_1, p_2, \ldots, p_k$ and messages $m_1, m_2, \ldots, m_k = m'$ ($k \geq 2$) such that:

- $p_1 = sender(m)$;
- p_i broadcasts m_i, for all $1 \leq i \leq k$;
- either $m = m_1$ or p_1 broadcasts m before it broadcasts m_1; and
- p_i delivers m_{i-1} before it broadcasts m_i, for all $2 \leq i \leq k$;

By Local Order (applied to p_k, m_{k-1}, m_k and p), p delivers m_{k-1} before $m_k = m'$. By applying Local Order again (to p_{k-1}, m_{k-2}, m_{k-1} and p), p delivers m_{k-2} before m_{k-1}. In general, an easy induction shows that p delivers m_{i-1} before m_i for all $1 < i \leq k$. Thus p delivers m_1 before $m_k = m'$. Recall that either $m = m_1$, or p_1 broadcasts m before m_1. In the former case, we immediately have that p delivers m before m'. In the latter case, FIFO Order ensures that p delivers m before m_1. Thus, in both cases, p delivers m before m', as we wanted to show. □

By Theorem 1, we can show that a broadcast algorithm satisfies Causal Order by proving that it satisfies FIFO Order and Local Order. This is easier than proving Causal Order directly: in general, the causal precedence between two broadcasts involves an arbitrarily long chain of intermediate broadcasts and deliveries. A direct proof of Causal Order requires a tedious induction on the length of this chain analogous to that in the proof of Theorem 1. In contrast, FIFO and Local Order involve chains of length one and two, respectively. Their proofs do not require induction, and so they are simpler. This is illustrated in the proofs of Theorems 3, 5, 6, 7, and 9.

3.4 Atomic Broadcast

If the broadcasts of two messages are not related by causal precedence, Causal Broadcast does not impose any requirement on the order they can be delivered. In particular, two correct processes may deliver them in different orders. This disagreement on message delivery order is undesirable in some applications. For example, consider a replicated database with two copies of a bank account x residing at different sites. Initially, x has a value of \$100. A user deposits \$20, triggering a broadcast of "add \$20 to x" to the two copies of x. At the same time, at a different site, the bank initiates a broadcast of "add 10% interest to x". Because these two broadcasts are not causally related, Causal Broadcast allows the two copies of x to deliver these update messages in different orders. This results in the two copies of x having different values, creating an inconsistency in the database.

To prevent such problems, Atomic Broadcast requires that all correct processes deliver *all* messages in the same order. This total order on message delivery ensures that all correct

processes have the same "view" of the system; hence they can act consistently without any additional communication. Formally, an *Atomic Broadcast* is a Reliable Broadcast that satisfies the following requirement:

- *Total Order:* If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

The Agreement and Total Order requirements of Atomic Broadcast imply that correct processes eventually deliver the same *sequence* of messages.

3.5 FIFO Atomic Broadcast

Atomic Broadcast does *not* require that messages be delivered in FIFO Order. For example, Atomic Broadcast allows the following scenario: a process suffers a transient failure during the broadcast of a message m, and then broadcasts m', and correct processes only deliver m'. Thus, Atomic Broadcast is not stronger than FIFO Broadcast.

We therefore define FIFO Atomic Broadcast which is a Reliable Broadcast that satisfies both FIFO Order and Total Order. FIFO Atomic Broadcast is stronger than both Atomic Broadcast and FIFO Broadcast.

3.6 Causal Atomic Broadcast

FIFO Atomic Broadcast does *not* require that messages be delivered in Causal Order. Reconsider the earlier network news example, and suppose FIFO Atomic Broadcast is used to disseminate articles. The following undesirable scenario is possible. Faulty user A broadcasts an article; faulty user B, who is the only one to deliver that message, broadcasts a response and then immediately crashes (before delivering its own response). Correct user C delivers the response, although it never delivers the original article. Thus, FIFO Atomic Broadcast does not necessarily satisfy Causal Order.

We therefore define Causal Atomic Broadcast which is a Reliable Broadcast that satisfies both Causal Order and Total Order. Causal Atomic Broadcast is stronger than both FIFO Atomic Broadcast and Causal Broadcast. This type of broadcast is the key mechanism of the State Machine approach to fault-tolerance [Lam78a,Sch90].

3.7 Timed Broadcasts

Many applications require that if a message is delivered at all, it is delivered within a bounded time after it was broadcast. This property is called Δ-*Timeliness*. As usual, in a distributed system elapsed time can be interpreted in two different ways: real time, as measured by an external observer, or local time, as measured by the local clocks of processes. This gives rise to two different ways of defining the Δ-Timeliness property. The one corresponding to real time is:

- *Real-Time Δ-Timeliness:* There is a known constant Δ such that if a message m is broadcast at real time t, then no correct process delivers m after real time $t + \Delta$.

On the other hand, the definition of Δ-Timeliness in terms of local clocks bounds the difference between the local broadcasting time and the local delivery time. To formally specify such a bound, we assume that each message m contains a *timestamp* $ts(m)$ denoting the local time at which m was broadcast according to the sender's clock. That is, if a process p wishes to broadcast a message m when its local clock shows c, then p tags m with $ts(m) = c$. The definition of Δ-Timeliness that corresponds to local time is:

- *Local-Time Δ-Timeliness:* There is a known constant Δ such that no correct process p delivers a message m after local time $ts(m) + \Delta$ on p's clock.

A broadcast that satisfies either version of the Δ-Timeliness property is called a *Timed Broadcast*. For example, *Timed Reliable Broadcast* is a Reliable Broadcast that satisfies Local- or Real-Time Δ-Timeliness. When referring to a Timed Broadcast, one must explicitly state which of the two Timeliness properties is assumed. The parameter Δ is called the *latency* of the Timed Broadcast.

3.8 Uniform Broadcasts

The Agreement, Integrity, Order, and Δ-Timeliness properties of the broadcasts defined so far place no restrictions on the messages delivered by faulty processes. Since we are dealing with benign failures, such restrictions are desirable and achievable. For example, the Agreement property states that if a *correct* process delivers a message m, then all correct processes eventually deliver m. This requirement allows a *faulty* process to deliver a message that is never delivered by the correct processes. This behavior is undesirable in many applications, such as Atomic Commitment in distributed databases [Gra78,BHG87,BT93], and can be avoided if the failures are benign. For such failures, we can strengthen the Agreement property to:

- *Uniform Agreement:* If a process (whether correct or faulty) delivers a message m, then all correct processes eventually deliver m.

Similarly, Integrity allows a faulty process to deliver a message more than once, and to deliver messages "out of thin air" (i.e., messages that were not previously broadcast). If failures are benign, this behavior can be avoided and we can strengthen the Integrity property as follows:

- *Uniform Integrity:* For any message m, every process (whether correct or faulty) delivers m at most once, and only if m was previously broadcast by $sender(m)$.

We can also strengthen each version of the Δ-Timeliness property by requiring that even faulty processes respect the bound on the broadcast latency:

- *Uniform Real-Time Δ-Timeliness:* There is a known constant Δ such that if a message m is broadcast at real time t, then no process (whether correct or faulty) delivers m after real time $t + \Delta$.
- *Uniform Local-Time Δ-Timeliness:* There is a known constant Δ such that no process p (whether correct or faulty) delivers a message m after local time $ts(m) + \Delta$ on p's clock.

Likewise, we can strengthen each of the Order properties, by requiring that even faulty processes do not violate them. Specifically, we define:

- *Uniform FIFO Order:* If a process broadcasts a message m before it broadcasts a message m', then no process (whether correct or faulty) delivers m' unless it has previously delivered m.
- *Uniform Local Order:* If a process broadcasts a message m and a process delivers m before broadcasting m', then no process (whether correct or faulty) delivers m' unless it has previously delivered m.
- *Uniform Causal Order:* If the broadcast of a message m causally precedes the broadcast of a message m', then no process (whether correct or faulty) delivers m' unless it has previously delivered m.
- *Uniform Total Order:* If any processes p and q (whether correct or faulty) both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

We can now state the uniform counterpart of Theorem 1 (the proof is omitted as it is almost identical to that of Theorem 1):

Theorem 2 *Uniform Causal Order is equivalent to Uniform FIFO Order and Uniform Local Order.*

Each broadcast type `T` has a *Uniform* counterpart obtained by replacing every one of the properties of `T` (except Validity) with the corresponding uniform version. For example, *Uniform Reliable Broadcast* satisfies Validity, Uniform Agreement, and Uniform Integrity.

3.9 Summary of Broadcast Specifications

All the broadcasts that we defined satisfy the properties of Reliable Broadcast, namely:

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

- *Integrity:* For any message m, every correct process delivers m at most once, and only if m was previously broadcast by $sender(m)$.

They only differ by the strength of their requirements for message delivery order. There are three such requirements:

- *FIFO Order:* If a process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.
- *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.
- *Total Order:* If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

Thus we have:

- Reliable Broadcast = Validity + Agreement + Integrity
- FIFO Broadcast = Reliable Broadcast + FIFO Order
- Causal Broadcast = Reliable Broadcast + Causal Order

Each of these broadcast types has an Atomic counterpart:

- Atomic Broadcast = Reliable Broadcast + Total Order
- FIFO Atomic Broadcast = FIFO Broadcast + Total Order
- Causal Atomic Broadcast = Causal Broadcast + Total Order

The relations among these six types of broadcasts, in terms of their order properties, is illustrated in Figure 1.

The above broadcasts do not place any bound on message delivery time. A *Timed* broadcast requires such a bound by having one of the following two properties:

- *Real-Time Δ-Timeliness:* There is a known constant Δ such that if a message m is broadcast at real time t, then no correct process delivers m after real time $t + \Delta$.
- *Local-Time Δ-Timeliness:* There is a known constant Δ such that no correct process p delivers a message m after local time $ts(m) + \Delta$ on p's clock.

Finally, we saw that every broadcast property (except Validity) has a uniform counterpart, which is defined by imposing the corresponding requirement even on messages delivered by faulty processes. A broadcast is *Uniform* if all its properties (except Validity) are uniform.

In the next section, we consider the above broadcast specifications in the context of arbitrary failures. This section can be skipped without loss of continuity.

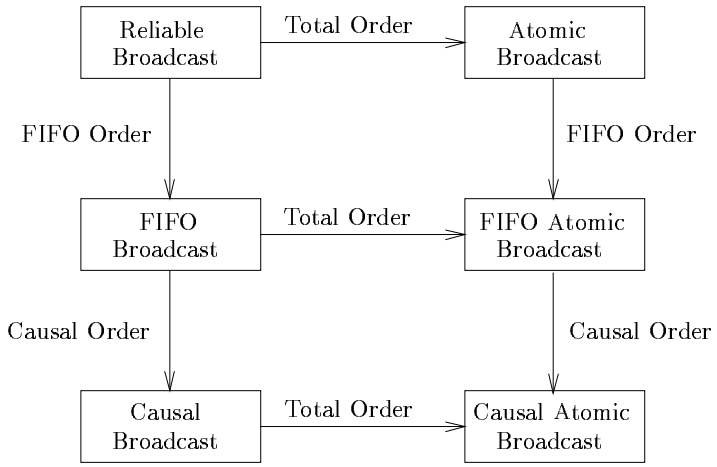

Figure 1: Relationship among Broadcast Primitives

3.10 Broadcast Specifications for Arbitrary Failures

The broadcast specifications given so far, were written with the assumption that only benign failures occur. When dealing with arbitrary failures, some minor modifications to these specifications are required.

Recall that any message $m \in \mathcal{M}$ that processes are allowed to broadcast and deliver must include some fields, such as a sender's id, $sender(m)$, a sequence number, $seq\#(m)$, and possibly a timestamp, $ts(m)$. In a system with arbitrary failures, we cannot assume that messages broadcast by processes that commit arbitrary failures are in $\mathcal{M}$; for example they may not have the appropriate fields. We assume that correct processes ignore (and thus never deliver) such messages. With this assumption, a correct process can always extract $sender(m)$, $seq\#(m)$, and when appropriate, $ts(m)$, from any message m that it delivers. It is important to realize that a process p that commits arbitrary failures may broadcast a message m with $sender(m) \neq p$, or with the wrong sequence number, or with a totally arbitrary timestamp.

Now consider Reliable Broadcast with arbitrary failures. The definitions of Validity and Agreement only refer to messages broadcast and delivered by *correct* processes. Since the meaning of such broadcasts and deliveries is unambiguous no matter what the failure model is, the definitions of these two properties do not change. The same is true about the first clause of Integrity. The second clause of Integrity, however, is problematic: Since $sender(m)$ may commit arbitrary failures, the meaning of "m was previously broadcast by $sender(m)$" is not clear. Even if the sender invokes `broadcast`(m), the external behavior of this invocation may look like an invocation of `broadcast`(m') to some or all other processes. The natural way to circumvent this problem is to redefine Integrity (for arbitrary failures) as follows:

- *Integrity:* For any message m, every correct process delivers m at most once, and if $sender(m)$ is correct then m was previously broadcast by $sender(m)$.

Integrity now refers only to broadcast and deliveries of correct processes.[4] This definition preserves the intended meaning of Integrity, restricted to the broadcasts of correct processes: No correct process can deliver a message m "out of thin air" if its field $sender(m)$ contains the identity of a correct process. In other words, faulty processes cannot "fool" correct ones into delivering a message m from a correct process that has not (yet) broadcast m. Thus, this formulation of Integrity guarantees the *authentication* of broadcasts. This is the broadcast/deliver counterpart of the send/receive message authentication discussed in Section 2.3.

Now consider FIFO Broadcast. The benign failure version of FIFO Order imposes an order on the delivery of messages broadcast by a process p that may be faulty. However, if p commits arbitrary failures, such an order is not meaningful. Thus, in the case of arbitrary failures, we weaken the order requirement by restricting its application only to messages broadcast by *correct* processes:

- *FIFO Order:* If a *correct* process broadcasts a message m before it broadcasts a message m', then no correct process delivers m' unless it has previously delivered m.[5]

A proper definition of Causal Broadcast in the presence of arbitrary failures is subtle and more complex. Moreover, the utility of such a broadcast is questionable: The context of a message broadcast by a correct process, i.e., its "causal past", may include the delivery of a message from a process that committed arbitrary failures. Thus, we do not pursue Causal Broadcast with such failures.

Next consider Atomic Broadcast, i.e., Reliable Broadcast with Total Order, for arbitrary failures. We have already given the definition of Reliable Broadcast in that case. The definition of Total Order refers only to deliveries by correct processes, and hence remains unchanged.

Finally, we consider each version of Δ-Timeliness. The definition of Local-Time Δ-Timeliness refers only to actions of correct processes, and remains unchanged. Real-Time Δ-Timeliness, however, refers to the real time at which a message is broadcast; this is now ambiguous as the sender of that message may be subject to arbitrary failures. We circumvent this problem by restricting the requirement to messages broadcast by correct processes only:

- *Real-Time Δ-Timeliness:* There is a known constant Δ such that if a message m is broadcast by a *correct* process at real time t, then no correct process delivers m after real time $t + \Delta$.

[4] Instead of reformulating Integrity, we could use the original definition with the convention that if a faulty process p is subject to arbitrary failures then the statement "p broadcasts m" is true for *all* $m \in \mathcal{M}$.

[5] This reformulation of FIFO Order allows correct processes to deliver messages broadcast by the *same* faulty sender in different orders. If desired, a stronger definition of FIFO Order can exclude this behavior.

With benign failures, every broadcast property (except Validity) has a uniform version that imposes some requirements on the messages that faulty processes deliver. With arbitrary failures, however, it is not possible to enforce *any* such requirement, so uniformity is a meaningless concept in this case.

3.11 Inconsistency and Contamination

Consider an application where processes communicate via fault-tolerant broadcasts (Figure 2). Assume that only benign failures may occur; thus, the current state of every process (whether correct or faulty) depends on the messages that it has delivered so far. This state, and the application protocol that the process executes, determines whether it should broadcast a message, and if so, the contents of that message.

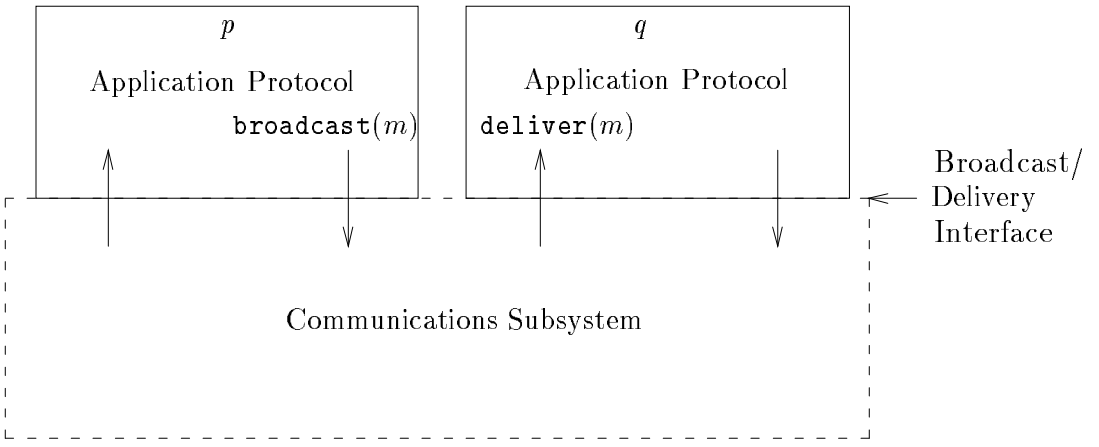

Figure 2: Application Protocol using Broadcasts

Suppose that a process p is faulty and omits to deliver a message that is delivered by all the correct processes. The state of p may now be "inconsistent" with respect to the state of correct processes. Suppose further that p continues to execute, and then, based on its inconsistent state, p broadcasts a message m that is delivered by all the correct processes. Note that m is "corrupted", i.e., its contents reflect p's erroneous state. Thus, by delivering m and changing state accordingly, the *correct* processes incorporate p's inconsistency into their own state — correct processes are now "contaminated." We come to the disconcerting conclusion that, even with benign failures, broadcasts can easily lead to the corruption of the *entire* system!

Unfortunately, the traditional specifications of most broadcasts, including Uniform broadcasts, allow the inconsistency of faulty processes, and the subsequent contamination of correct processes. For example, with Atomic Broadcast a faulty process may reach an inconsistent state in several ways: e.g., by omitting to deliver a message m that is delivered by all correct processes, or by delivering an extra message m that is not delivered by any

correct process, or by delivering messages out-of-order. With Uniform Atomic Broadcast, inconsistency may only result from skipping a message that is delivered by all correct processes. Once a faulty process becomes inconsistent, contamination can then follow. An example of inconsistency with respect to (Uniform) Atomic Broadcast is explained below.

A variable x with initial value 5 is replicated at three processes, p, q, and r. Process p atomically broadcasts an instruction to increment x, and q atomically broadcasts an instruction to double x. Processes p and q are correct, and they deliver the instructions to increment x and to double x, in that order. Their value of x is now 12. However, r is faulty: it first omits to deliver p's instruction to increment x, and then delivers q's instruction to double x. By skipping the increment x instruction, r becomes *inconsistent* — its new value of x (namely, 10) is now incorrect. Note that since r is faulty, this execution *does* indeed satisfy the usual specification of Atomic Broadcast. In fact, this particular execution can occur even with a Uniform Atomic Broadcast.

Once r is inconsistent, it can broadcast messages that are based on its erroneous state and thus contaminate all the correct processes. For example, suppose process r uses its new value of x to compute and broadcast the value of the replicated variable y, which is supposed to be $3x$ everywhere. Since r is inconsistent and has incorrectly computed x to be 10, r broadcasts $y := 30$, instead of the correct $y := 36$. When p and q deliver the message $y := 30$ and update their copies of y to be 30, they become contaminated.

Note that r becomes inconsistent by committing a simple "benign" failure — just skipping the delivery of a single message. However, as a result of this undetected failure, r subsequently broadcasts an incorrect message, and this broadcast "spreads" r's error to the rest of the system. At this point, it is almost as if r commits an "arbitrary-like" failure, even though it only fails by omission. Worse yet, r's failure corrupts the whole system.

It should be clear that preventing the inconsistency of faulty processes, or at least the contamination of correct ones, is desirable in many situations. Fortunately, this is possible with all the broadcasts that we considered in this paper, and for all benign failures.[6] Intuitively, a process can prevent its contamination by refusing to deliver messages from processes whose previous deliveries are not compatible with its own. The amount of information that each message should carry, so that every process can determine whether it is safe to deliver it, depends on the type of broadcast (e.g., FIFO Broadcast or Causal Atomic Broadcast), and on the failure assumptions. Preventing inconsistency is, however, more difficult and costly. Roughly speaking, it requires techniques that allow a faulty process to detect whether it is about to make a message delivery error, and, if so, to immediately stop.

A precise definition of inconsistency and contamination with respect to broadcasts is beyond the scope of this paper. We also omit the description of algorithms that prevent inconsistency and/or contamination. For a more complete treatment of this subject the reader is referred to [GT91,Gop92].

[6] With arbitrary failures, neither inconsistency nor contamination can be prevented. This is because the state of a faulty process may be inconsistent even if it delivers all messages correctly. This proces may then contaminate the rest of the system by broadcasting an erroneous message that seems correct to every process.

3.12 Amplification of Failures

A fault-tolerant broadcast is usually implemented by a broadcast algorithm that uses lower-level communication primitives, such as `send` and `receive` (Figure 3). With such a broadcast algorithm, the broadcasting or delivery of a message requires the execution of several instructions, and may include several sends and receives.

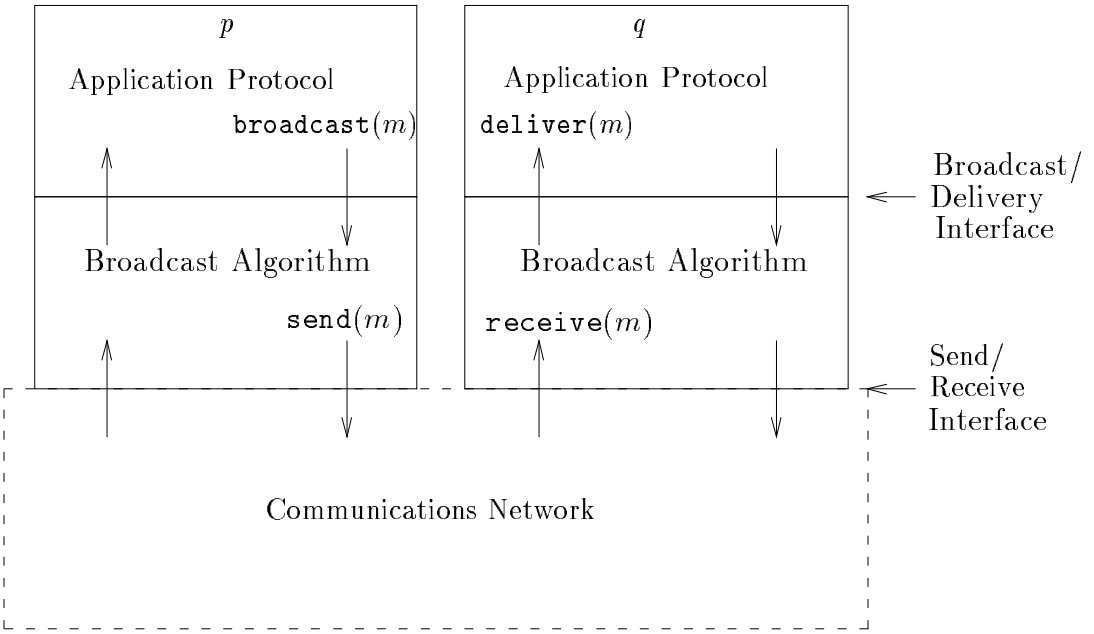

Figure 3: Application/Broadcast Layering

The models of failures commonly considered in the literature are defined in terms of failures that occur at the level of send and receive primitives, e.g., omissions to receive messages (Section 2.3). How do these failures affect the execution of higher-level primitives, such as broadcasts and deliveries? In particular, can we assume that if a process suffers a certain type of failure at the send/receive level, then it will always suffer the same type of failure at the broadcast/delivery level? For example, if a faulty process omits to receive messages, will it simply omit to deliver messages? Unfortunately, this is not always so. In general a broadcast algorithm is likely to *amplify* the severity of failures that occur at the low level. For example, there are Atomic Broadcast algorithms where the omission to receive messages causes a faulty process to deliver messages in the wrong order [Gop92].

But what if processes are only subject to *crash* failures? Can we assume that the message deliveries that a process makes before crashing are always "correct" (i.e., consistent with those of correct processes)? Intuitively, this seems very reasonable, since *by definition* a process that crashes executes perfectly until the moment it crashes. In other words, it seems impossible for such a process to make "mistakes" in its message deliveries before

crashing. However, this intuition *is* wrong. We illustrate this by a coordinator-based Atomic Broadcast algorithm that exhibits a surprising behavior: *even if a faulty process behaves correctly until it crashes, it may still deliver messages out-of-order before it crashes!* This algorithm, which satisfies the specification of Atomic Broadcast, is sketched below.

When a process intends to broadcast a message m, it first sends m to a *coordinator*. The coordinator delivers messages in the order in which it receives them, and periodically informs the other processes of this message delivery order. Other processes deliver messages according to this order. If the coordinator crashes, another process takes over as coordinator. Now, suppose a coordinator delivers m before m', and then crashes before informing any other process that m should be delivered before m'. The new coordinator cannot determine the order chosen by the faulty coordinator, and may decide that m' should be delivered before m. In this scenario, all correct processes follow the new coordinator and deliver m' before m. Thus, the faulty coordinator delivered messages out-of-order before crashing, even though it executed its protocol perfectly until it crashed.

The above example shows that even if a process is only subject to crash failures, it may become inconsistent *before* crashing. In other words, crash failures just by themselves do *not* guarantee reasonable behavior at the broadcast/delivery level. Furthermore, from the time that such a process becomes inconsistent to the time that it crashes, it may broadcast messages and thus contaminate all correct processes. Thus, even if processes can only fail by crashing, inconsistency and contamination can occur.[7]

These observations have subtle but important consequences. In particular, consider the State Machine approach to fault-tolerance (cf. [Lam84,Sch90]). This is a client/server system, where the server is replicated, and clients broadcast their requests to all servers using Causal Atomic Broadcast. Thus, all *correct* servers deliver the same set of requests, in the same causal order, and so they have identical state. When a server delivers a request from a client, it computes the appropriate reply to that request, and sends it to the client. Suppose that up to f servers are subject to general-omission failures. Clearly, the state of such a server can be erroneous, and so it may send incorrect replies. How many servers are needed to implement a fault-tolerant service? It is easy to see that $2f + 1$ servers are sufficient: a client is guaranteed to receive at least $f + 1$ identical replies (a majority) from correct servers. This scheme works even when servers are subject to arbitrary failures. However, requiring $2f + 1$ servers and computing the majority reply is expensive. Can we implement a fault-tolerant service with fewer servers if failures are less severe?

In particular, suppose that the f faulty servers are subject to *crash failures only*. In that case, it seems that $f + 1$ servers would now suffice: Since a faulty server executes correctly until it crashes, it is tempting to conclude that if *any* server sends a reply, that reply must be correct. And, since we have $f + 1$ servers, at least one of them will reply. Unfortunately, this reasoning is flawed. A reply may originate from a server s that will later crash. As we saw in our previous example, the particular Causal Atomic Broadcast algorithm used by clients to broadcast requests may be such that s delivers requests out-of-order before crashing. In other words, s could be in an inconsistent state and send the

[7] Of course, the prevention of inconsistency and contamination is much easier with crash failures, than with omission or timing failures.

wrong reply before crashing! A single reply is guaranteed to be correct *if and only if* the Causal Atomic Broadcast used is specifically designed to prevent inconsistency, as discussed in the previous section.

Bibliographic Notes

The specification of the various types of broadcasts given in this section was designed by the authors with the help of Ajei Gopal. The origins of FIFO Broadcast and Causal Broadcast are in the Isis system [BJ87], although many systems now provide such primitives, including Psync [PBS89] and Transis [ADKM92]. Atomic Broadcast goes back to the early work of Lamport on the State Machine approach to fault-tolerance [Lam78a], and is a central mechanism in the HAS project [CASD85].

The concept of Uniformity was introduced by Neiger and Toueg in [NT87] in connection to the Agreement property. The problems of inconsistency, contamination, and failure amplification, were first defined and studied in [GT91,Gop92].

4 Broadcast Algorithms I — Methodology

In the following two sections, we derive algorithms for the six types of broadcasts introduced in Section 3: Reliable, FIFO, Causal Broadcast, and their three Atomic counterparts.

Our derivation and exposition of broadcast algorithms follows a particular methodology which is made possible by the modularity of our broadcast specifications. In Section 3 we defined six types of broadcasts by adding FIFO, Causal, or Total Order to the specification of Reliable Broadcast, the weakest type of broadcast that we consider (see Figure 1). We derive broadcast algorithms by following the same modular paradigm: We start with any given Reliable Broadcast algorithm, and show how to achieve each one of these three order properties by a corresponding algorithmic transformation.

More precisely, we exhibit three transformations: one adds FIFO Order, i.e., it converts *any* Reliable Broadcast algorithm into a FIFO Broadcast; one adds Causal Order, i.e., it converts any FIFO Broadcast algorithm into a Causal Broadcast; and one adds Total Order, i.e., it converts any Reliable, FIFO, or Causal Broadcast algorithm (that satisfies Local-Time Δ-Timeliness) into its Atomic counterpart. These three transformations correspond to the arrows of Figure 1: The first one corresponds to the two top vertical arrows, the second one to the two bottom vertical arrows, and the last one to the three horizontal arrows.

Given any Reliable Broadcast algorithm, we can now obtain algorithms for every other type of broadcast by successively applying our transformations. For example, suppose we want to derive a Causal Atomic Broadcast algorithm. To do so, we can select any path from Reliable Broadcast to Causal Atomic Broadcast in Figure 1 (there are three such paths) and apply to the given Reliable Broadcast algorithm the transformations that correspond to the arrows along that path. Since for some of the arrows we actually give several alternative

transformations, there are many different Causal Atomic Broadcast algorithms that we can obtain from the given Reliable Broadcast in this manner.[8]

It is important to note that all our transformations are generic, i.e., they do not require any assumptions on the type or synchrony of the underlying communication network, and they work for any type and number of benign failures. Furthermore, all transformations preserve Uniform Agreement and, under certain assumptions, both versions of Δ-Timeliness: If the given broadcast algorithm satisfies any of these desirable properties, then so does the algorithm that results from the transformation.

This methodology of deriving broadcast algorithms by generic transformations has several advantages. The algorithms are developed modularly — thus, they are smaller, simpler, and easier to understand. The techniques required to achieve each one of the three order properties (FIFO, Causal, and Total Order) are shown separately, and independently from the characteristics of the underlying communication network. The proofs are also modular, easier, and "safer": a broadcast algorithm that invokes a weaker broadcast primitive as a "black box" can only rely on the *specification* of that "box", so its proof cannot erroneously rely on a property that is only true for a *particular implementation* of that "box".

A modular implementation of stronger broadcasts in terms of weaker ones also increases the portability of the broadcast software. Whenever we wish to develop a suite of fault-tolerant broadcast algorithms for a specific system S, all we have to do is provide an implementation for Reliable Broadcast that works in S (this implementation must provide Δ-Timeliness if we wish to obtain an Atomic Broadcast). Because all our transformations work for any type and number of benign failures, and do not rely on any assumptions about the underlying network, these will automatically yield broadcast algorithms that also work in S. Thus, if we consider our suite of algorithms to be a software package for fault-tolerant broadcasts, the layered construction allows us to port this package from one system to another by reimplementing and fine-tuning only Reliable Broadcast. On the other hand, layered implementations do have a disadvantage: They may lead to a decrease in efficiency, because they hide specific features of the underlying communications network that may be exploited by certain algorithms.

In Section 5 we present our system-independent transformations. In Section 6 we focus on a particular type of system, namely point-to-point networks, and describe a Reliable Broadcast algorithm for such a system. We also determine the conditions under which this algorithm achieves Uniform Agreement and Real-Time Δ-Timeliness, and describe a simple modification to achieve Local-Time Δ-Timeliness. By applying our system-independent transformations to this Reliable Broadcast algorithm, we immediately obtain broadcast algorithms of all types for point-to-point networks.

[8] In this example, the given Reliable Broadcast must satisfy Local-Time Δ-Timeliness. This is because the transformation that adds Total Order requires the given broadcast algorithm to satisfy this property.

5 Broadcast Algorithms II — Transformations

5.1 Introduction

In this section we describe our transformations. These are based on a small number of techniques, each of which enhances a given broadcast by adding FIFO, Causal, or Total Order. Adding Total Order, which corresponds to the three horizontal arrows in Figure 1, is effected by a single transformation (that works if the given broadcast satisfies Local-Time Δ-Timeliness). Adding FIFO Order, which corresponds to the two top vertical arrows, is also accomplished by one transformation. Adding Causal Order, which corresponds to the two bottom vertical arrows, can be done by using either one of two transformations. We also present a particularly efficient transformation for adding Causal Order to a FIFO Atomic Broadcast (this corresponds to the bottom right vertical arrow).

All our transformations preserve Uniform Agreement, and, under some conditions, both versions of Δ-Timeliness. This means that if the given broadcast algorithm satisfies any of these properties, then so does the resulting broadcast algorithm.

The transformations work for any type and number of benign failures, and regardless of the type or synchrony of the network. Thus, the resulting algorithm works in whatever system, and under whatever assumptions, the given algorithm works.

All the broadcasts that we consider here satisfy the *Uniform* version of Integrity. This is important to our modular approach because to build stronger broadcast primitives from weaker ones the latter are often required to satisfy Uniform Integrity. For the sake of brevity, in the rest of this section, when we mention a type of broadcast we always assume that it satisfies Uniform Integrity without explicitly saying so.

5.2 Definitions and Notation

Since we build our broadcast primitives in a layered fashion, it is typical for a higher-level broadcast primitive to invoke a lower-level one as a procedure. To disambiguate between the different broadcast primitives used in an algorithm, we introduce the following notation. We first define a short-hand notation for the *type* of a broadcast. In particular, `R` stands for Reliable Broadcast, `F` for FIFO Broadcast, and `C` for Causal Broadcast. Similarly, `A` stands for Atomic Broadcast, `FA` for FIFO Atomic Broadcast, and `CA` for Causal Atomic Broadcast. We denote by `broadcast`(`T`, m) and `deliver`(`T`, m), the two primitives corresponding to a broadcast of type `T`. When a process invokes `broadcast`(`T`, m), we say that it *T-broadcasts* m. When it returns from the execution of `deliver`(`T`, m), we say that it *T-delivers* m. For example, `broadcast`(`R`, m) is the broadcast primitive for Reliable Broadcast, and if a process invokes `broadcast`(`R`, m), we say that it R-broadcasts m. Similarly, `deliver`(`CA`, m) is the delivery primitive for Causal Atomic Broadcast, and if a process returns from the execution of `deliver`(`CA`, m), we say that it CA-delivers m.

Consider two problems denoted $\mathcal{A}$ and $\mathcal{B}$. A *transformation from $\mathcal{A}$ to $\mathcal{B}$* is an algorithm

$T_{\mathcal{A}\rightarrow\mathcal{B}}$ that converts *any* algorithm A that solves $\mathcal{A}$ into an algorithm B that solves $\mathcal{B}$.[9] We say that A is the *given* algorithm and B the *resulting* algorithm of the transformation. Transformation $T_{\mathcal{A}\rightarrow\mathcal{B}}$ *preserves* property P if it converts any algorithm for $\mathcal{A}$ that satisfies P into an algorithm for $\mathcal{B}$ that also satisfies P.

For example, in Section 5.4 we present an algorithm that transforms any algorithm for Reliable Broadcast into one for FIFO Broadcast. Roughly speaking it works as follows. To F-broadcast a message, a process simply R-broadcasts it. When a process R-delivers a message m, it delays the F-delivery of m, if necessary, until it has F-delivered all the messages that the sender of m F-broadcast before m. As we will see, this transformation also happens to preserve Total Order. This means that if the given Reliable Broadcast satisfies Total Order, (i.e., it is actually an Atomic Broadcast), then so does the resulting FIFO Broadcast (i.e., it is a FIFO Atomic Broadcast).

Informally, a transformation of one broadcast algorithm into another is *blocking* if the resulting broadcast algorithm has an execution in which a process delays the delivery of a message for a later time. For example, the transformation from Reliable Broadcast to FIFO Broadcast outlined above is blocking: It is possible that when a process p R-delivers a message m it has to delay the F-delivery of m while waiting for the F-delivery of some earlier message from the sender of m. A transformation is *non-blocking* if it is not blocking.

When we present an algorithm we give the pseudo-code for a typical process. In our algorithms every process executes the same code. Thus, different processes have local variables with the same name, and this can lead to ambiguity. In such cases, we avoid this problem by subscripting a variable local to a process with the identity of that process. Thus, var_p denotes the value of local variable var at process p.

5.3 Achieving Total Order

In this section we describe a single algorithm that can be used to transform a Reliable, FIFO or Causal Broadcast *that satisfies Local-Time* Δ*-Timeliness* into its Atomic counterpart, i.e., an Atomic, FIFO Atomic or Causal Atomic Broadcast. Thus, this transformation corresponds to the three horizontal arrows in Figure 1. It is based on a very simple idea which exploits the Timeliness property to guarantee Total Order, while preserving all the other properties of interest.

The algorithm in Figure 4 shows how to transform any broadcast algorithm that satisfies Local-Time Δ-Timeliness into one that satisfies both Total Order and Local-Time Δ-Timeliness. In this figure, `B` denotes the type of the given broadcast, and `BA` denotes the type of the broadcast that results from the transformation. This transformation preserves Validity, Agreement, Integrity, FIFO Order and Causal Order (and their uniform counterparts): If the given broadcast algorithm satisfies any of these properties, then so does the resulting broadcast algorithm.

The transformation works as follows. If p wishes to BA-broadcast m, it uses the given broadcast primitive to B-broadcast m. When a process B-delivers m, it schedules the BA-

[9] We also say that $T_{\mathcal{A}\rightarrow\mathcal{B}}$ is a reduction of problem $\mathcal{B}$ to problem $\mathcal{A}$.

delivery of m at local-time $ts(m)+\Delta$ (recall that $ts(m)$ is the sending time of m according to the sender's clock, and Δ is the bound on message latency guaranteed by the given broadcast algorithm). If two or more messages are scheduled to be BA-delivered at the same local time then they are BA-delivered in an *a priori* agreed order, e.g., in increasing order of the senders' ids. If a process B-delivers m after local time $ts(m)+\Delta$, then it never BA-delivers m. This transformation assumes that local clocks satisfy the Clock Monotonicity property (see Section 2.8).

Algorithm for process p:
To execute `broadcast(BA,`m`)`:
    `broadcast(B,`m`)`

`deliver(BA,`m`)` *occurs as follows:*
    upon `deliver(B,`m`)` **do**
        schedule `deliver(BA,`m`)` **at time** $ts(m)+\Delta$

Figure 4: Adding Total Order to Timed Broadcast

Theorem 3 *Suppose the Clock Monotonicity property holds. The algorithm in Figure 4 transforms any broadcast algorithm that satisfies Local-Time Δ-Timeliness into one that satisfies (the uniform versions of) both Local-Time Δ-Timeliness and Total Order. This transformation preserves Validity, Agreement, Integrity, FIFO Order and Causal Order, and their uniform counterparts.*

The significance of this theorem lies in the following:

Corollary 1 *Suppose the Clock Monotonicity property holds. The algorithm in Figure 4 transforms any Reliable, FIFO or Causal Broadcast algorithm that satisfies Local-Time Δ-Timeliness into its Atomic counterpart.*

`Proof of Theorem 3:` Assume that the given broadcast algorithm satisfies Local-Time Δ-Timeliness. We first show that the broadcast algorithm that results from the transformation satisfies the uniform versions of Local-Time Δ-Timeliness and Total Order.

Uniform Local-Time Δ-Timeliness: If any process (whether correct or faulty) BA-delivers a message m, it does so at local time $ts(m)+\Delta$.

Uniform Total Order: If any two processes p and q (whether correct or faulty) BA-deliver m and m', they do so at local times $ts(m)+\Delta$ and $ts(m')+\Delta$, respectively. If $ts(m) < ts(m')$, by Clock Monotonicity, both p and q BA-deliver m before m'. Similarly, if $ts(m') < ts(m)$, they both BA-deliver m' before m. Finally, if $ts(m) = ts(m')$, they both BA-deliver the messages in order of increasing sender ids. In all cases, p and q BA-deliver m and m' in the same order.

We now show that the transformation preserves each property listed in the theorem; i.e., if the given broadcast satisfies any of these properties then so does the resulting broadcast.

Validity: If a correct process p BA-broadcasts m, it B-broadcasts m. By Validity and Local-Time Δ-Timeliness of the given broadcast algorithm, p B-delivers m by local time $ts(m)+\Delta$, and schedules the BA-delivery of m for local time $ts(m) + \Delta$. By Clock Monotonicity, p eventually BA-delivers m.

Agreement: If any correct process BA-delivers a message m, then it must have B-delivered m. By Agreement and Local-Time Δ-Timeliness of the given broadcast algorithm, every correct process also B-delivers m, and does so by local time $ts(m) + \Delta$. By Clock Monotonicity, every correct process eventually BA-delivers m at local time $ts(m) + \Delta$.

Integrity: Immediate from Integrity of the given broadcast algorithm.

FIFO Order: Suppose a process q BA-broadcasts m before m'. Consider a correct process p that BA-delivers m'. We must show that p BA-delivers m before m'. By definition, q BA-broadcast m and m' at local times $ts(m)$ and $ts(m')$. By Clock Monotonicity, $ts(m) < ts(m')$, so $ts(m)+\Delta < ts(m')+\Delta$. From the algorithm and the hypothesis, it is clear that q B-broadcasts m before m', and that p B-delivers m'. By FIFO Order of the given broadcast algorithm, p B-delivers m before m'. By the algorithm, p schedules the BA-deliveries of m and m' at times $ts(m) + \Delta < ts(m') + \Delta$. Since p BA-delivers m', its clock reached $ts(m') + \Delta$. By Clock Monotonicity, p's clock reached $ts(m) + \Delta$ before $ts(m') + \Delta$. Thus, p BA-delivers m before m'.

By Theorem 1, Causal Order is equivalent to FIFO Order and Causal Order. Since we already showed that the transformation preserves FIFO Order, to show that it also preserves Causal Order it now suffices prove that it preserves Local Order.

Local Order: Suppose a process q BA-delivers m before it BA-broadcasts m'. Consider a correct process p that BA-delivers m'. We must show that p BA-delivers m before m'. By the algorithm, q BA-delivered m at local time $ts(m)+\Delta$, and later BA-broadcast m' at local time $ts(m')$. By Clock Monotonicity, $ts(m)+\Delta < ts(m')$, so $ts(m)+\Delta < ts(m')+\Delta$. From the algorithm and the hypothesis, it is clear that q B-delivers m before B-broadcasting m', and that p B-delivers m'. The proof now continues exactly as in the proof of FIFO Order.

It is easy to check that the transformation also preserves the *uniform* versions of Agreement, Integrity, FIFO Order and Causal Order. □

Note the absence of any induction argument in the proof of Causal Order, despite the fact that two broadcasts can be causally related by arbitrarily long chains of broadcasts and deliveries. This is the consequence of Theorem 1 whose proof factorized this induction.

5.4 Achieving FIFO Order

In this section we describe a simple algorithm that transforms any Reliable Broadcast algorithm into a FIFO Broadcast that satisfies *Uniform* FIFO Order. This transformation preserves Total Order; thus, if the given Reliable Broadcast is actually an Atomic Broadcast, the resulting algorithm is a FIFO Atomic Broadcast. In other words, this transformation

corresponds to the two top vertical arrows in Figure 1.

The transformation, shown in Figure 5, works as follows. To F-broadcast a message m, a process s simply R-broadcasts m. Recall that if m is the ith message F-broadcast by s, then m is tagged with $sender(m) = s$ and $seq\#(m) = i$. Every process p maintains a vector of counters $next$, such that $next[s]$ is the sequence number of the next F-broadcast from s that p is ready to F-deliver. When a process p R-delivers m with $sender(m) = s$, p checks whether m is F-deliverable immediately, i.e., whether $next[s] = seq\#(m)$. If m is not F-deliverable when it is R-delivered, p inserts m into a $msgSet_p$ for possible future F-delivery: m is now blocked. Otherwise, p F-delivers m right away and increments $next[s]$ to reflect that. This may cause previously blocked messages from s to become F-deliverable; p scans $msgSet_p$ and F-delivers any such messages.

For example, suppose p already F-delivered messages tagged 1 and 2 from s (i.e., $next[s] = 3$), and p's $msgSet$ already contains messages tagged 4 and 8 from s. If p R-delivers message m tagged 3 from s (i.e., $sender(m) = s$ and $seq\#(m) = 3$), p F-delivers m immediately, and increments $next[s]$ to 4. Now message tagged 4 from s in $msgSet$ also becomes F-deliverable, and p delivers it.

Theorem 4 *The algorithm in Figure 5 transforms any Reliable Broadcast algorithm into a FIFO Broadcast algorithm that satisfies Uniform FIFO Order. Furthermore, this transformation preserves [Uniform] Total Order.*

`Proof:` First we establish some basic facts about the transformation.

Claim 1: For any process p (whether correct or faulty), if $next_p[s] = k$ then the sequence of messages that p has F-delivered so far is the sequence of the first $k-1$ messages F-broadcast by s.

This claim can be easily shown by induction on k and using the Uniform Integrity of the given Reliable Broadcast; the details are omited.

Claim 2: Suppose a correct process p R-delivers a message m and F-delivers all the messages that $sender(m)$ F-broadcast before m. Then p also F-delivers m.

`Proof of Claim 2:` Suppose that p and m satisfy the hypothesis of the claim, and let $sender(m) = s$ and $seq\#(m) = k$. By hypothesis, p F-delivers all the $k-1$ messages that s F-broadcast before m. Since p increments $next_p[s]$ for each one of these F-deliveries, eventually $next_p[s] \geq k$. There are two possible cases. (i) Eventually $next_p[s]$ reaches the value $k+1$. By Claim 1, p F-delivers the first k messages F-broadcast by s, including m. (ii) Eventually $next_p[s]$ reaches the value k and then remains forever stuck at that value. Let m' be the message whose F-delivery by p makes $next_p[s] = k$. By hypothesis, p R-delivers m. If, when this occurs, m' has already been delivered, we have $next_p[s] = k$ and thus p will F-deliver m right away. Otherwise, p will insert m into $msgSet_p$ and when it later F-delivers m' and sets $next_p[s] = k$, it will also F-deliver m. Thus, in both cases p F-delivers m, as wanted. □ Claim 2

Using the above claims, we first show that the algorithm that results from the transformation satisfies the properties of FIFO Broadcast.

Variables of process p:
{ *$msgSet$: set of messages that p has R-delivered* }
{ *$next[s]$: sequence number of the next F-broadcast by s that p is ready to F-deliver* }

. . . . . .

Algorithm for process p:
Initialization:
$msgSet := \emptyset$
$next[s] := 1$, for each process s

To execute `broadcast(F, m)`:
`broadcast(R, m)`

`deliver(F, −)` *occurs as follows*:
upon `deliver(R, m′)` **do**
$s := sender(m')$
if $next[s] = seq\#(m')$
then
`deliver(F, m′)`
$next[s] := next[s] + 1$
while $(\exists m \in msgSet : sender(m) = s$ **and** $next[s] = seq\#(m))$ **do**
`deliver(F, m)`
$next[s] := next[s] + 1$
else
$msgSet := msgSet \cup \{m'\}$

Figure 5: Transforming Reliable Broadcast into FIFO Broadcast

Validity: Suppose that a correct process p F-broadcasts a message that it never F-delivers. Let m be the first such message that p F-broadcasts. Since p F-broadcasts m, it previously R-broadcast m. By Validity of Reliable Broadcast, p eventually R-delivers m. By the choice of m, p F-delivers all the messages that it F-broadcast before m. By Claim 2, p F-delivers m, contradicting the definition of m. Thus, a correct process F-delivers every message that it F-broadcasts.

Uniform FIFO Order: Suppose a process p F-delivers a message m. Let $sender(m) = s$ and $seq\#(m) = k$. By the algorithm, just before p F-delivers m, $next_p[s] = k$. By Claim 1, p has already F-delivered all the $k-1$ messages that s F-broadcast before m, as wanted.

Agreement: Suppose, for contradiction, that Agreement is violated. Thus, there are two correct processes p and q such that p F-delivers a message that q does not. Let m be such a message with the smallest possible sequence number. Since p F-delivers m, it previously R-delivered m; by Agreement of Reliable Broadcast, q also R-delivers m. Furthermore, by the Uniform FIFO Order property shown above, p previously F-delivered all the messages that $sender(m)$ F-broadcast before m. By the choice of m, q also F-delivers all these messages. By Claim 2, q F-delivers m, a contradiction.

Uniform Integrity: Suppose p F-delivers m with $sender(m) = s$ and $seq\#(m) = k$. By the algorithm, p previously R-delivered m. By Uniform Integrity of the given Reliable Broadcast, s R-broadcasts m. Therefore, s must have F-broadcast m. Furthermore, when p F-delivers m, it increments $next_p[s]$ from k to $k+1$. Since $next_p[s]$ never decreases, and must be k for m to be F-delivered, m is not F-delivered again.

Next we show that the transformation preserves [Uniform] Total Order.

Total Order: It is easy to see that the sequence of R-deliveries uniquely determines the sequence of F-deliveries. By Agreement and Total Order of the given Reliable Broadcast, all correct processes R-deliver the same sequence of messages. Hence, they all F-deliver the same sequence of messages. Therefore, the resulting FIFO Broadcast satisfies Total Order.

Uniform Total Order: For any message m, define $Past(m)$ to be the set of messages F-broadcast by $sender(m)$ up to and including the F-broadcast of m. By Uniform FIFO Order (shown above), if a process p F-delivers m then it must have previously F-delivered and thus R-delivered all the messages in $Past(m)$. Furthermore, it is clear that p F-delivers m as soon as it has R-delivered those messages.

Consider any two processes p and q that F-deliver messages m_1 and m_2. By the above argument, p and q must each R-deliver all the messages in $Past(m_1) \cup Past(m_2)$. Furthermore, the order in which each of p and q F-delivers messages m_1 and m_2 is determined by the order in which they R-deliver the messages in $Past(m_1) \cup Past(m_2)$. By Uniform Total Order of the given Reliable Broadcast, p and q R-deliver these messages in the same order. Since this order determines the order of F-delivering m_1 and m_2, p and q F-deliver m_1 and m_2 in the same order. This shows that the resulting FIFO Broadcast satisfies Uniform Total Order. □

Observation: This transformation also preserves Uniform Agreement and Δ-Timeliness

(both Real- and Local-Time, and their uniform versions).[10]

The transformation has some straightforward optimizations. First, once p F-delivers m, it can remove m from $msgSet$. This reduces the space needed for storing $msgSet$. Second, since a message from s can become unblocked only as a result of the F-delivery of some (earlier) message from s, it is more efficient to keep a separate $msgSet[s]$ for every process s, containing the blocked messages F-broadcast by s.

5.5 Achieving Causal Order

In the next two sections we describe two transformations from FIFO Broadcast to Causal Broadcast: one is blocking and the other not. Each of these preserves Total Order, i.e., if the given FIFO Broadcast algorithm is Atomic, the resulting algorithm is a Causal Atomic Broadcast. Thus, each transformation corresponds to the two bottom vertical arrows in Figure 1.

Both transformations require that the given FIFO Broadcast algorithm satisfy *Uniform* FIFO Order. Luckily, our transformation from Reliable Broadcast to FIFO Broadcast does result in such an algorithm (see Theorem 4 in the previous section). Thus, these transformations can be "chained" together.

5.5.1 Non-Blocking Transformation

In Figure 6, we give a non-blocking transformation of FIFO to Causal Broadcast. To C-broadcast a message m, a process p uses the given FIFO Broadcast algorithm to F-broadcast the sequence of messages $rcntDlvrs||m$, where $rcntDlvrs$ is the sequence of messages that p C-delivered since its previous C-broadcast ("||" is the concatenation operator). When a process q F-delivers such a sequence, q C-delivers all the messages in the sequence that it did not previously C-deliver.

Theorem 5 *The algorithm in Figure 6 transforms any FIFO Broadcast algorithm that satisfies Uniform FIFO Order into a Causal Broadcast algorithm that satisfies Uniform Causal Order. Furthermore, this transformation preserves Total Order.*

`Proof:` We first show that the algorithm that results from the transformation satisfies the properties of Causal Broadcast.

Validity: Suppose p is correct and C-broadcasts m. Thus, p F-broadcasts $\langle rcntDlvrs \,||\, m\rangle$, and by Validity of FIFO Broadcast, p eventually F-delivers $\langle rcntDlvrs \,||\, m\rangle$. From the algorithm, p C-delivers m.

Agreement: Suppose p is correct and C-delivers m. From the algorithm, it is clear that p F-delivered some sequence $\langle m_1, m_2, \ldots, m_l\rangle$ that contains m. From Agreement of FIFO Broadcast, all correct processes eventually F-deliver $\langle m_1, m_2, \ldots, m_l\rangle$, and thus C-deliver m.

[10] The preservation of *Local-Time* Δ-Timeliness requires that local clocks never decrease (Section 2.8).

Variable of process p:
{ *$rcntDlvrs$: sequence of messages that p C-delivered since its previous C-broadcast* }

. . . . . .

Algorithm for process p:
Initialization:
$rcntDlvrs := \perp$

To execute `broadcast`$(\mathtt{C}, m)$:
`broadcast`$(\mathtt{F}, \langle rcntDlvrs \,||\, m \rangle)$
$rcntDlvrs := \perp$

`deliver`$(\mathtt{C}, -)$ *occurs as follows*:
upon `deliver`$(\mathtt{F}, \langle m_1, m_2, \ldots, m_l \rangle)$ for some l **do**
for $i := 1..l$ **do**
if p has not previously executed `deliver`$(\mathtt{C}, m_i)$
then
`deliver`$(\mathtt{C}, m_i)$
$rcntDlvrs := rcntDlvrs \,||\, m_i$

Figure 6: Transforming FIFO Broadcast into Causal Broadcast: Non-Blocking Version

Uniform Integrity: From the algorithm, a process C-delivers a message m only if it has not previously executed `deliver(C,`m`)`. Thus, a process delivers m at most once. Consider the first process that C-delivers message m. It must have F-delivered $\langle rcntDlvrs || m \rangle$, for some $rcntDlvrs$. By Uniform Integrity of FIFO Broadcast, some process p F-broadcast $\langle rcntDlvrs || m \rangle$. This occurred when p C-broadcast m.

To prove Uniform Causal Order it suffices to prove Uniform FIFO Order and Uniform Local Order (see Theorem 2). To do so, we first show:

Claim: Suppose some process q F-broadcasts $\langle rcntDlvrs' || m' \rangle$, and either q previously F-broadcast $\langle rcntDlvrs || m \rangle$ or m is in $rcntDlvrs'$. Then no process (whether correct or faulty) C-delivers m' unless it has previously C-delivered m.

`Proof of Claim:` The proof of this claim is by contradiction. Assume that the hypothesis holds, and some process C-delivers m' but does not C-deliver m before m'. Let p be the first process to do so (in real time). There are two cases depending on what caused p to C-deliver m':

1. p F-delivered $\langle rcntDlvrs' || m' \rangle$. By hypothesis, there are two possible subcases. (a) Process q F-broadcast $\langle rcntDlvrs || m \rangle$ before $\langle rcntDlvrs' || m' \rangle$. By Uniform FIFO Order of FIFO Broadcast, p must have F-delivered $\langle rcntDlvrs || m \rangle$ before $\langle rcntDlvrs' || m' \rangle$. (b) m is in $rcntDlvrs'$. In both subcases, it is clear from the algorithm that p C-delivered m before m', a contradiction.

2. p F-delivered a message $\langle rcntDlvrs'' || m'' \rangle$, where m' is in $rcntDlvrs''$ and m is not before m' in $rcntDlvrs''$. Let $s = sender(\langle rcntDlvrs'' || m'' \rangle)$. Since m' is in $rcntDlvrs''$, s C-delivered m' before F-broadcasting $\langle rcntDlvrs'' || m'' \rangle$. By Uniform FIFO Order of FIFO Broadcast, p F-delivered all the previous F-broadcasts of s. Since p does not C-deliver m before m', m was not included in any of these F-broadcasts. Furthermore, m did not appear before m' in $rcntDlvrs''$. Thus, when s C-delivered m', it had not previously C-delivered m. Since s C-delivered m' before p, this contradicts the definition of p.

Since both cases lead to a contradiction, the claim follows. □ Claim

Uniform FIFO Order: Suppose a process q C-broadcasts m before m'. From the algorithm, it is clear that q F-broadcast $\langle rcntDlvrs || m \rangle$ before $\langle rcntDlvrs' || m' \rangle$, for some $rcntDlvrs$ and $rcntDlvrs'$, respectively. By the above claim, no process C-delivers m' unless it has previously C-delivered m.

Uniform Local Order: Suppose a process q C-delivers m before it C-broadcasts m', and a process p C-delivers m'. We must show that p C-delivers m before m'. Let m'' be the first message that q C-broadcast after it C-delivered m (m'' could be m'). When q C-broadcasts m'', it F-broadcasts $\langle rcntDlvrs'' || m'' \rangle$ for some $rcntDlvrs''$. By the definition of m'' and the algorithm, $rcntDlvrs''$ contains m. Thus, by the above claim, p C-delivers m before m''. If the C-broadcasts of m'' and m' are actually the same, then we are done. Otherwise, q C-broadcasts m'' before m'. By the Uniform FIFO Order property shown above, p C-delivers m'' before m'. Thus, p C-delivers m before m', as wanted.

Next we show that the transformation preserves Total Order (the argument is identical to the one used in the corresponding result of Theorem 4).

Total Order: It is easy to see that the sequence of F-deliveries uniquely determines the sequence of C-deliveries. By Agreement and Total Order of the given FIFO Broadcast, all correct processes F-deliver the same sequence of messages. Hence, they all C-deliver the same sequence of messages. Therefore, the resulting Causal Broadcast satisfies Total Order. □

Observation: This transformation preserves Uniform Agreement. In general, it does not preserve (Real- or Local-Time) Δ-Timeliness. If the given FIFO Broadcast satisfies *Uniform* Agreement, however, the transformation does preserve both versions of Δ-Timeliness. Finally, although it preserves Total Order, it does not preserve *Uniform* Total Order.

5.5.2 Blocking Transformation

In Figure 7, we give a blocking transformation of FIFO to Causal Broadcast. Its advantage over the non-blocking one just described is that it uses shorter messages. Like the non-blocking transformation, this also requires that the given FIFO Broadcast satisfy *Uniform* FIFO Order. It works as follows. Each process p maintains $numOfAllDlvrs$, a vector of counters such that $numOfAllDlvrs[s]$ is the number of C-broadcasts from s that p C-delivered since the beginning. To C-broadcast a message m, a process p uses the given FIFO Broadcast algorithm to F-broadcast m together with $numOfAllDlvrs$. When p F-delivers a message m' from s with its corresponding vector $numOfAllDlvrs'$, p checks whether m' is C-deliverable immediately. This is the case if p has already C-delivered every message that s had C-delivered at the time it C-broadcast m'. Process p checks this by testing whether $numOfAllDlvrs \geq numOfAllDlvrs'$. [11]

If m' is not C-deliverable immediately, then p inserts m' and the associated vector $numOfAllDlvrs'$ in a $msgList$ (where messages are kept in order of insertion) for possible future C-delivery: The message m' is now blocked. If, on the other hand, m' is C-deliverable, then p C-delivers it right away and updates $numOfAllDlvrs$ to reflect this fact. Since a C-delivery may cause some blocked messages to become C-deliverable, p scans $msgList$ to C-deliver the first C-deliverable message on that list. It then repeats the scanning of $msgList$ for as long as there is a C-deliverable message in the list.

Theorem 6 *The algorithm in Figure 7 transforms any FIFO Broadcast algorithm that satisfies Uniform FIFO Order into a Causal Broadcast algorithm that satisfies Uniform Causal Order. Furthermore, this transformation preserves [Uniform] Total Order.*

`Proof:` We first show that the algorithm that results from the transformation satisfies the properties of Causal Broadcast.

[11] Given two n-vectors V and V', we say that $V \geq V'$ if $V[i] \geq V'[i]$ for all $1 \geq i \geq n$. Similarly, $V + V'$ denotes the vector whose i-th element is $V[i] + V'[i]$.

Variables of process p:
{ *$numOfAllDlvrs[s]$: number of all the messages that p C-delivered from s* }
{ *$msgList$: list of messages that p F-delivered but not yet C-delivered* }

. . . . . .

Algorithm for process p:
Initialization:
$numOfAllDlvrs[s] := 0$, for each process s
$msgList := \perp$

To execute `broadcast`$(\mathtt{C}, m)$:
`broadcast`$(\mathtt{F}, \langle m, numOfAllDlvrs \rangle)$

`deliver`$(\mathtt{C}, -)$ *occurs as follows*:
upon `deliver`$(\mathtt{F}, \langle m', numOfAllDlvrs' \rangle)$ **do**
$s := sender(m')$
if $numOfAllDlvrs \geq numOfAllDlvrs'$
then
`deliver`$(\mathtt{C}, m')$
$numOfAllDlvrs[s] := numOfAllDlvrs[s] + 1$
while $(\exists \langle m, N \rangle \in msgList : numOfAllDlvrs \geq N)$ **do**
let $\langle m, N \rangle$ be the first message in $msgList$ s.t. $numOfAllDlvrs \geq N$
`deliver`$(\mathtt{C}, m)$
$numOfAllDlvrs[sender(m)] := numOfAllDlvrs[sender(m)] + 1$
$msgList := msgList - \langle m, N \rangle$
else
$msgList := msgList \,||\, \langle m', numOfAllDlvrs' \rangle$

Figure 7: Transforming FIFO Broadcast into Causal Broadcast: Blocking Version

Validity: Suppose p is correct and C-broadcasts m. Thus, p F-broadcasts $\langle m, numOfAllDlvrs \rangle$, and by Validity of FIFO Broadcast, p eventually F-delivers $\langle m, numOfAllDlvrs \rangle$. Since p's vector of counters $numOfAllDlvrs_p$ never decreases, p C-delivers m.

Agreement: Suppose, for contradiction, that Agreement is violated. Thus, there are two correct processes p and q such that p C-delivers a message that q does not C-deliver. Consider the sequence of messages that p C-delivers, in the order they are C-delivered. Let m be the first message on that sequence that q does not C-deliver. From the algorithm, since p C-delivered m, it must have previously F-delivered $\langle m, N \rangle$ for some vector N. By Agreement of FIFO Broadcast, q eventually F-delivers $\langle m, N \rangle$ as well. If this F-delivery occurs when $numOfAllDlvrs_q \geq N$, then q immediately C-delivers m — a contradiction to the definition of m. Thus, we may assume that when q F-delivers $\langle m, N \rangle$, $numOfAllDlvrs_q \not\geq N$ and q inserts $\langle m, N \rangle$ in $msgList_q$ for possible future C-delivery.

Let N_p be the value of p's vector $numOfAllDlvrs$ when p C-delivered m. From the algorithm, $N_p \geq N$. Clearly, when p C-delivered m, it had previously C-delivered exactly $N_p[r]$ messages that were C-broadcast by r, for every process r. By the definition of m, q eventually C-delivers all these messages. Thus, q eventually has $numOfAllDlvrs_q \geq N_p$. Consider the message m'' whose C-delivery by q causes for the first time $numOfAllDlvrs_q \geq N_p$. When q C-delivers m'', $\langle m, N \rangle$ is already in $msgList_q$. Immediately after the C-delivery of m'', $numOfAllDlvrs_q \geq N_p \geq N$, and thus q C-delivers m when it scans $msgList_q$ — a contradiction to the definition of m.

Uniform Integrity: From the Uniform Integrity of the given FIFO Broadcast, it is easy to see that a process C-delivers a message m only if $sender(m)$ previously C-broadcast m. It remains to show that no process p C-delivers m twice. To do so, we show that p does not insert the messages $\langle m, N \rangle$ and $\langle m, N' \rangle$, for $N' \neq N$, into $msgList$. This is true because, otherwise, by the Uniform Integrity of the given FIFO Broadcast, both of these messages would have been F-broadcast by the same process s, namely $sender(m)$, at two different times. Thus, s would have C-broadcast m twice, which is impossible.

To prove Uniform Causal Order it suffices to prove Uniform FIFO Order and Uniform Local Order (see Theorem 2).

Uniform FIFO Order: Suppose that a process s C-broadcasts m before it C-broadcasts m', and a process p C-delivers m'. We show that p C-delivers m before m'.

From the algorithm, s F-broadcasts $\langle m, N \rangle$ before it F-broadcasts $\langle m', N' \rangle$, and p must have F-delivered $\langle m', N' \rangle$, for some vectors N and N', with $N' \geq N$. By Uniform FIFO Order of the given FIFO Broadcast, p must have F-delivered $\langle m, N \rangle$ before $\langle m', N' \rangle$. If p C-delivered m without blocking it, we are done. So we may assume that p inserted $\langle m, N \rangle$ in $msgList_p$ before it F-delivered $\langle m', N' \rangle$. There are two cases:

- When p F-delivers $\langle m', N' \rangle$ it blocks m'. Thus, p inserts $\langle m', N' \rangle$ in $msgList_p$. Since (a) p C-delivers all the C-deliverable messages in $msgList_p$ in the order of they appear in that list, (b) $\langle m, N \rangle$ is before $\langle m', N' \rangle$ in $msgList_p$, (c) if m' is C-deliverable, so is m (because $N' \geq N$), and (d) p C-delivers m', we conclude that p C-delivers m before m', as wanted.

- When p F-delivers $\langle m', N' \rangle$ it C-delivers m' immediately. Thus, when this F-delivery occurs $numOfAllDlvrs_p \geq N'$. Consider the message m'' whose C-delivery by p first causes $numOfAllDlvrs_p \geq N'$. The C-delivery of m'' occurs after p F-delivered $\langle m, N \rangle$ (because p blocked m), and before p F-delivered $\langle m', N' \rangle$ (because p did not block m'). Thus, when p C-delivers m'', $\langle m, N \rangle$ is already in $msgList_p$. Since the C-delivery of m'' results in $numOfAllDlvrs_p \geq N'$, and $N' \geq N$, p C-delivers m when it scans $msgList_p$ immediately after it C-delivers of m''. Hence, p C-delivers m before m', as wanted.

Uniform Local Order: Suppose a process q C-delivers m before it C-broadcasts m'. By Uniform Integrity shown above, m was C-broadcast by some process s. Suppose m was the kth message C-broadcast by s. By Uniform FIFO shown above, when q C-delivers m, this is the kth message q has C-delivered from s. Thus, immediately after the C-delivery of m, q has $numOfAllDlvrs_q[s] = k$. Therefore, when q later C-broadcasts m', it F-broadcasts $\langle m', N' \rangle$ with $N'[s] \geq k$.

Suppose some process p C-delivers m'. This occurs after p F-delivers $\langle m', N' \rangle$ and $numOfAllDlvrs_p \geq N'$. Thus, when p C-delivers m' it has $numOfAllDlvrs_p[s] \geq N'[s] \geq k$, and therefore p has already C-delivered at least k messages C-broadcast by s. By Uniform FIFO shown above, these include the first k messages C-broadcast by s, and thus m. So p C-delivers m before m', as wanted.

Next we show that the transformation preserves [Uniform] Total Order.

Total Order: This proof is identical to the one given in the corresponding result of Theorem 5.

Uniform Total Order: This proof is the same as the one given in the corresponding result of Theorem 4, except that $Past(m)$ must be redefined. Let $\xrightarrow{C}$ be the causal precedence relation induced by C-broadcasts and C-deliveries (see Section 2.7). We now define $Past(m) = \{m' \mid m' = m \text{ or } \texttt{broadcast}(\texttt{C}, m') \xrightarrow{C} \texttt{broadcast}(\texttt{C}, m)\}$; intuitively, $Past(m)$ is the set of messages in the "causal past" of m. With this definition, the argument follows along the lines of the proof of Theorem 4. □

Observation: This transformation also preserves Uniform Agreement and Real-Time Δ-Timeliness. Moreover, under some assumptions explained below, the transformation also preserves Local-Time Δ-Timeliness. In Section 2.7, we defined the causal precedence relation $\rightarrow$ induced by the broadcast and delivery events. In Section 2.8 we explained what it means for local clocks to be consistent with $\rightarrow$. When executing the transformation in Figure 7, there are F-broadcasts and F-deliveries, which induce the causal precedence relation $\xrightarrow{F}$, and also C-broadcast and C-deliveries, which induce the relation $\xrightarrow{C}$. We can show that if the local clocks are consistent with $\xrightarrow{F}$ then they are also consistent with $\xrightarrow{C}$, and the transformation in Figure 7 preserves Local-Time Δ-Timeliness.

This transformation can be improved in the following way. In its present form, each process p keeps track of the number of *all* the messages it has C-delivered from every process, and sends this information, in the form of a vector of counters, along with every message it wishes to C-broadcast. Instead, p can send a vector containing only the number

of *new* messages that p C-delivered from each process since p's previous C-broadcast. These vectors contain smaller numbers, so messages are shorter. By maintaining the right kind of information, each process can use these vectors to determine if a message is immediately C-deliverable or should be blocked. We desist from giving the details of this optimization, as we shall describe in detail some transformations that use the same idea in Sections 5.6.2 and 5.6.3.

5.6 From FIFO Atomic to Causal Atomic Broadcast

In the previous section we described two different ways of transforming FIFO Broadcast into Causal Broadcast while preserving Total Order. Each of these transformations corresponds to *both* bottom vertical arrows in Figure 1. In this section, we describe three efficient transformations that correspond to the right bottom arrow *only*: each of these add Causal Order to a FIFO *Atomic* Broadcast. Surprisingly, these transformations are *non*-blocking even though they tag each message with just a vector of counters (like the blocking algorithm in Figure 7) rather than piggybacking a list of messages (like the non-blocking algorithm in Figure 6). It is now possible to use vectors of counters to achieve Causal Order without blocking because we start from a stronger broadcast algorithm — one that satisfies Total Order.

5.6.1 Basic Transformation

In the blocking transformation from FIFO to Causal Broadcast that we saw in Figure 7, if a process p F-delivers a message $\langle m', - \rangle$ and cannot C-deliver m' immediately, it inserts $\langle m', - \rangle$ in $msgList$ for possible future C-delivery: m' is now blocked. We can obtain a non-blocking transformation from FIFO Atomic to Causal Atomic Broadcast with a simple modification (see Figure 8). When p FA-delivers $\langle m', - \rangle$ such that it cannot CA-deliver m' immediately, p simply discards m' and inserts $sender(m')$ into $suspects_p$ — a set of processes that p suspects to be faulty. Process p routinely discards every subsequent message that originates from any process in that set. Thus, now p never saves any message for future delivery and messages are never blocked. This eliminates $msgList$ and the need to scan this list for possible deliveries.

Theorem 7 *The algorithm in Figure 8 transforms any FIFO Atomic Broadcast algorithm that satisfies Uniform FIFO Order into a Causal Atomic Broadcast algorithm that satisfies Uniform Causal Order.*

`Proof:` We show that the algorithm that results from the transformation satisfies the properties of Causal Atomic Broadcast.

Validity: We first claim that for all processes p (whether correct or faulty), p is never in $suspects_p$. The proof is by contradiction. Consider the first time t a process p inserts p into $suspects_p$. From the algorithm, it is clear that this occurs when p FA-delivers a message $\langle m', N' \rangle$ such that $p = sender(m')$ and $N_p \not\geq N'$, where N_p is the value of $numOfAllDlvrs_p$

Variables of process p:
{ *$numOfAllDlvrs[s]$: number of all the messages that p C-delivered from s* }
{ *suspects: set of processes that p suspects to be faulty* }

Algorithm for process p:
Initialization:
$numOfAllDlvrs[s] := 0$, for each process s
$suspects := \emptyset$

To execute `broadcast`$(\mathtt{CA}, m)$:
`broadcast`$(\mathtt{FA}, \langle m, numOfAllDlvrs \rangle)$

`deliver`$(\mathtt{CA}, -)$ *occurs as follows*:
upon `deliver`$(\mathtt{FA}, \langle m', numOfAllDlvrs' \rangle)$ **do**
$s := sender(m')$
if $s \notin suspects$ **and** $numOfAllDlvrs \geq numOfAllDlvrs'$
then
`deliver`$(\mathtt{CA}, m')$
$numOfAllDlvrs[s] := numOfAllDlvrs[s] + 1$
else
discard m'
$suspects := suspects \cup \{s\}$

Figure 8: Transforming FIFO Atomic Broadcast into Causal Atomic Broadcast: Basic Version

at time t. By Uniform Integrity of FIFO Atomic Broadcast, p must have previously FA-broadcast $\langle m', N' \rangle$. Thus, N' is the value of $numOfAllDlvrs_p$ at some time before t. Since the elements of vector $numOfAllDlvrs_p$ never decrease, $N_p \geq N'$, a contradiction.

We now show that if a correct process p CA-broadcasts m then it eventually CA-delivers m. Suppose p CA-broadcasts m. By the algorithm, p FA-broadcasts $\langle m, N \rangle$ for some N. By Validity of FIFO Atomic Broadcast, p eventually FA-delivers $\langle m, N \rangle$. If p does not CA-deliver m, it inserts p in $suspects_p$. Our previous claim shows that this cannot occur. Thus, p CA-delivers m, as wanted.

Agreement: Suppose, for contradiction, that Agreement is violated. Thus, there are two correct processes p and q such that p CA-delivers a message that q does not CA-deliver. Consider the sequence of messages that p CA-delivers, in the order in which p CA-delivers them. Let m be the first message on that sequence that q does not CA-deliver. From the algorithm, p CA-delivers m upon FA-delivering a message $\langle m, N \rangle$ for some vector of counters N. By Agreement of FIFO Atomic Broadcast, q eventually FA-delivers $\langle m, N \rangle$ as well.

Let $s = sender(m)$. Let S_p and N_p be the values of $suspects_p$ and $numOfAllDlvrs_p$, respectively, when p FA-delivered $\langle m, N \rangle$. Similarly, S_q and N_q are the values of $suspects_q$ and $numOfAllDlvrs_q$, respectively, when q FA-delivered $\langle m, N \rangle$. Since p CA-delivers m, $s \notin S_p$ and $N_p \geq N$. Since q does not CA-deliver m, $s \in S_q$ or $N_q \not\geq N$ (*).

Consider the (possibly empty) sequence σ of messages that p FA-delivers before $\langle m, N \rangle$. By Agreement and Total Order of FIFO Atomic Broadcast, σ is also the sequence of messages that q FA-delivers before $\langle m, N \rangle$. For all i, let $\sigma_i = \{m' \mid \langle m', - \rangle$ is in σ and $sender(m') = i\}$. Note that $N_p[i]$ is the number of messages in σ_i that p CA-delivers. Furthermore, $i \notin S_p$ if and only if p CA-delivers all the messages in σ_i. Symmetric remarks hold for $N_q[i]$ and S_q.

By the definition of m, for all i, every message in σ_i that p CA-delivers is also CA-delivered by q. Thus, for all i, $N_q[i] \geq N_p[i]$. Since $s \notin S_p$, p CA-delivered all the messages in σ_s. Thus, q also CA-delivered all the messages in σ_s. So $s \notin S_q$. In summary, $s \notin S_q$ and $N_q \geq N_p$ — a contradiction to (*).

Uniform Integrity: The proof is immediate from the Uniform Integrity of the given FIFO Atomic Broadcast and the observation that a process CA-delivers a message m only if it previously FA-delivers some message $\langle m, - \rangle$.

To prove Uniform Causal Order it suffices to prove Uniform FIFO Order and Uniform Local Order (see Theorem 2).

Uniform FIFO Order: Suppose that a process s CA-broadcasts m before it CA-broadcasts m', and a process p CA-delivers m'. We show that p CA-delivers m before m'. From the algorithm, s FA-broadcasts $\langle m, N \rangle$ before it FA-broadcasts $\langle m', N' \rangle$, and p FA-delivers $\langle m', N' \rangle$, for some vectors N and N'. By Uniform FIFO Order of the given FIFO Atomic Broadcast, p FA-delivers $\langle m, N \rangle$ before $\langle m', N' \rangle$. Assume, for contradiction, that when p FA-delivers $\langle m, N \rangle$ it does not CA-deliver m. In this case, p immediately inserts $s = sender(m)$ in $suspects_p$. Since s is also the sender of m' and it is now in $suspects_p$, when p subsequently FA-delivers $\langle m', N' \rangle$ p does not CA-deliver m' — a contradiction to our initial assumption.

Thus, p CA-delivers m immediately after it FA-delivers $\langle m, N \rangle$. This occurs before p FA-delivers $\langle m', N' \rangle$, and thus before it CA-delivers m'.

Uniform Local Order: The proof is the same as the one given in Theorem 6. □

Observation: The transformation in Figure 8 preserves Uniform Agreement and both versions of Δ-Timeliness.

5.6.2 First Optimization

In the basic transformation in Figure 8, when a process p wishes to CA-broadcast m, it FA-broadcasts $\langle m, numOfAllDlvrs \rangle$, where $numOfAllDlvrs$ is a vector of counters indicating the number of *all* the messages that p CA-delivered from each process *since the beginning*. In our first optimization (Figure 9), we seek to reduce the size of these counters: When a process p wishes to CA-broadcast m, it FA-broadcasts $\langle m, numOfRcntDlvrs \rangle$, where $numOfRcntDlvrs$ is a vector indicating the number of messages that p CA-delivered from each process *since its previous CA-broadcast only*. Clearly, $numOfRcntDlvrs$ is smaller than $numOfAllDlvrs$.

Unfortunately, in order to piggyback $numOfRcntDlvrs$ rather than $numOfAllDlvrs$ onto messages, we need to increase the *local space* required by each process. Each process p must now maintain a local matrix of counters, M_p, which it uses to reconstruct the information necessary to determine whether a message is CA-deliverable. The s-th row of M_p, denoted $M_p[s, *]$, is defined as follows. Let t be the time when s CA-broadcast the last message that p CA-delivered from s. $M_p[s, *]$ is a vector of counters indicating the number of messages that s CA-delivered from each process by time t.

The optimized transformation, shown in Figure 9, works as follows. When a process p wishes to CA-broadcast m, it FA-broadcasts $\langle m, numOfRcntDlvrs \rangle$, and resets $numOfRcntDlvrs$ to $\langle 0, 0, \ldots, 0 \rangle$. When p FA-delivers a message $\langle m', numOfRcntDlvrs' \rangle$ from some process s, p uses the matrix M to reconstruct $numOfAllDlvrs'$ — the vector that s would have piggybacked onto m' in the basic version of the transformation. Specifically, p computes $numOfAllDlvrs'$ by adding $numOfRcntDlvrs'$ to $M[s, *]$. Now p can test whether m' is CA-deliverable exactly as in the non-optimized version, namely by checking whether $s \notin suspects$ and $numOfAllDlvrs \geq numOfAllDlvrs'$. If p CA-delivers m', it updates its local vectors $numOfAllDlvrs$ and $numOfRcntDlvrs$ in the obvious way, and sets $M[s, *]$ to $numOfAllDlvrs'$.

The correctness of the optimized version (Figure 9), follows from the correctness of the basic version (Figure 8), and the fact that the value of the vector $numOfAllDlvrs'$, (piggybacked in the first version and reconstructed in the second one) is the same in both versions. So, the "CA-deliverability test" is actually the same in both versions.

5.6.3 Second Optimization

The first optimization requires each process p to maintain the vector $numOfAllDlvrs$, where $numOfAllDlvrs[s]$ indicates the total number of messages that p CA-delivered from

Variables of process p:

{ *$numOfAllDlvrs[s]$: number of all the messages that p CA-delivered from s* }
{ *$numOfRcntDlvrs[s]$: number of messages that p CA-delivered from s* }
{ *since p's previous CA-broadcast* }
{ *$M[s,r]$: number of messages that s CA-delivered from r by the time* }
{ *s CA-broadcast the last message that p CA-delivered* }
{ *suspects: set of processes that p suspects to be faulty* }

. . . . . .

Algorithm for process p:
Initialization:
$numOfAllDlvrs[s] := 0$, for each process s
$numOfRcntDlvrs[s] := 0$, for each process s
$M[s,r] := 0$, for all processes s, r
$suspects := \emptyset$

To execute `broadcast(CA,`m`)`:
`broadcast(FA,`$\langle m, numOfRcntDlvrs\rangle$`)`
$numOfRcntDlvrs := \langle 0, 0, \ldots, 0\rangle$

`deliver(CA,`$-$`)` *occurs as follows*:
upon `deliver(FA,`$\langle m', numOfRcntDlvrs'\rangle$`)` **do**
$s := sender(m')$
$numOfAllDlvrs' := M[s,*] + numOfRcntDlvrs'$
if $s \notin suspects$ **and** $numOfAllDlvrs \geq numOfAllDlvrs'$
then
`deliver(CA,`m'`)`
$numOfRcntDlvrs[s] := numOfRcntDlvrs[s] + 1$
$numOfAllDlvrs[s] := numOfAllDlvrs[s] + 1$
$M[s,*] := numOfAllDlvrs'$
else
discard m'
$suspects := suspects \cup \{s\}$

Figure 9: Transforming FIFO Atomic Broadcast into Causal Atomic Broadcast: First Optimization

s since the beginning. It also requires p to maintain a local matrix M, where, roughly speaking, $M[s, r]$ is a counter indicating the total number of messages that s CA-delivered from r in p's causal's past. The elements of $numOfAllDlvrs$ and M are monotonically increasing, and may grow to be too large in practice. The second optimization replaces these two data structures with a matrix that has smaller elements. More specifically, it maintains the matrix $numAhead$, such that $numAhead[s, r] = numOfAllDlvrs[r] - M[s, r]$.

We now describe how $numAhead$ can be used to replace both M and $numOfAllDlvrs$. In Figure 9, we use M and $numOfAllDlvrs$ only to test whether $numOfAllDlvrs \geq numOfAllDlvrs'$. Since $numOfAllDlvrs'$ is set to $M[s, *]+numOfRcntDlvrs'$ just before this test is executed, the test is equivalent to checking whether $numOfAllDlvrs \geq M[s, *]+numOfRcntDlvrs'$. Since $numAhead[s, *] = numOfAllDlvrs - M[s, *]$, the test is also equivalent to checking whether $numAhead[s, *] \geq numOfRcntDlvrs'$. This is the test used by our second optimization (Figure 10).

We now describe how p maintains $numAhead$. By definition, $numAhead[s, r] = numOfAllDlvrs[r] - M[s, r]$. Thus, the second optimization must update $numAhead$ every time $numOfAllDlvrs$ or M is updated in the first optimization. There are two such updates (after the CA-delivery of a message m' in Figure 9):

- $numOfAllDlvrs[s]$ is incremented by one. By definition, the corresponding update is $numAhead[*, s] := numAhead[*, s] + \langle 1, 1, \ldots, 1 \rangle$.
- $M[s, *]$ is set to $M[s, *]+numOfRcntDlvrs'$. By definition, the corresponding update is $numAhead[s, *] := numAhead[s, *] - numOfRcntDlvrs'$.

The correctness of the second optimization (Figure 10), follows directly from the correctness of the first optimization (Figure 9), and the fact that it correctly maintains the invariant $numAhead[s, *] = numOfAllDlvrs - M[s, *]$ that links the second optimization to the first one.

6 Broadcast Algorithms III — Point-to-Point Networks

In the preceding section, we described a set of *system-independent* transformations that can convert Reliable Broadcast algorithms into algorithms for every other type of broadcast. In this section, we describe a simple Reliable Broadcast algorithm for a particular type of system, namely, point-to-point networks with benign process and link failures that do not disconnect correct processes. This algorithm satisfies Uniform Integrity, a necessary requirement for our transformations. Under some additional assumptions on the type of failures, it also satisfies Uniform Agreement. Moreover, if the system is synchronous, this Reliable Broadcast algorithm satisfies Real-Time Δ-Timeliness, and, with a simple modification, Local-Time Δ-Timeliness. Finally, if the links of the point-to-point network are FIFO (an assumption that holds in many systems), it satisfies Causal Order, i.e., it is actually a Causal Broadcast!

Variables of process p:

{ *$numOfRcntDlvrs[s]$: number of messages that p CA-delivered from s* }
{ *since p's previous CA-broadcast* }
{ $numAhead[s, r] = numOfAllDlvrs[r] - M[s, r]$ }
{ *suspects: set of processes that p suspects to be faulty* }

. . . . . .

Algorithm for process p:
Initialization:

$numOfRcntDlvrs[s] := 0$, for each process s
$numAhead[s, r] := 0$, for all processes s, r
$suspects := \emptyset$

To execute `broadcast(CA,`m`)`:

`broadcast(FA,`$\langle m, numOfRcntDlvrs\rangle$`)`
$numOfRcntDlvrs := \langle 0, 0, \ldots, 0\rangle$

`deliver(CA,`$-$`)` *occurs as follows*:

upon `deliver(FA,`$\langle m', numOfRcntDlvrs'\rangle$`)` **do**
$s := sender(m')$
if $s \notin suspects$ **and** $numAhead[s, *] \geq numOfRcntDlvrs'$
then
`deliver(CA,`m'`)`
$numOfRcntDlvrs[s] := numOfRcntDlvrs[s] + 1$
$numAhead[*, s] := numAhead[*, s] + \langle 1, 1, \ldots, 1\rangle$
$numAhead[s, *] := numAhead[s, *] - numOfRcntDlvrs'$
else
discard m'
$suspects := suspects \cup \{s\}$

Figure 10: Transforming FIFO Atomic Broadcast into Causal Atomic Broadcast: Second Optimization

In asynchronous point-to-point networks, we can apply our transformations to this Reliable Broadcast algorithm, and automatically obtain FIFO and Causal Broadcast algorithms for such systems. In synchronous point-to-point networks, we can apply the transformations to the version of the Reliable Broadcast algorithm that satisfies Local-Time Δ-Timeliness, to obtain algorithms for every one of the six types of broadcast. Since our transformations preserve Uniform Agreement and, under some assumptions, both versions of Δ-Timeliness, if we start from the Reliable Broadcast algorithm that satisfy some of these properties, the resulting broadcast algorithms satisfy the same properties.

Note that the above approach does not yield an Atomic Broadcast algorithm (and *a fortiori* a FIFO Atomic or Causal Atomic Broadcast algorithm) in *asynchronous* point-to-point networks. This is not a limitation of this particular approach: Atomic Broadcast cannot be solved in such a system. In fact, this impossibility result holds even if the network is completely connected, has correct links, at most one process may fail, and it can only fail by crashing (see Theorem 13 in Section 7.2.1).

6.1 Model of Point-to-Point Networks

Recall from Section 2.2 that in a point-to-point network, a pair of processes connected by a link can communicate by means of `send` and `receive` primitives. We now establish some terminology regarding these primitives and state two properties that they satisfy. (A more detailed model of point-to-point networks is given in the Appendix.)

Consider the `send` and `receive` associated with the link from process p to process q. If p invokes `send` with a message m as a parameter we say that p *sends* m *to* q; in our algorithms we denote this invocation by "`send`(m) `to` q". When p returns from that invocation we say that p *completes the sending of* m *to* q. When a process q returns from the execution of `receive` with message m as the returned value, we say that q *receives* m; we denote this by "`receive`(m)". For convenience, we assume that every process can "send" messages to itself, and that it "receives" such messages instantaneously. This is only a fictional device, as a process does not really invoke the primitives `send` and `receive` for such messages. These primitives satisfy the following two properties:[12]

- *Validity:* If p sends m to q, and both p and q and the link from p to q are correct, then q eventually receives m. (If $p = q$ then p receives m instantaneously.)
- *Uniform Integrity:* For any message m, q receives m at most once from p, and only if p previously sent m to q.

6.2 Assumptions and Notation

All our Reliable Broadcast algorithms rely on the following two assumptions:

[12] To simplify the formulation of these properties, we assume that each message sent from p to q is unique. This can be easily enforced by using link sequence numbers.

a. *Benign Failures:* Process and link failures are benign.

b. *No Partitioning:* Every two correct processes are connected via a path consisting entirely of correct processes and links.

In these algorithms, a process p is required to send a message m to every one of its neighbors in the network (i.e., to every process q that is connect to p by a link). We use the notation "`send`(m) `to` all neighbors" as a short-hand for "**for all** q such that q is a neighbor of p **do** `send`(m) `to` q." If p fails while executing this **for** loop, it is possible that some neighbors of p receive m while others do not.

In this section, instead of R-broadcasts and R-deliveries we refer simply to broadcasts and deliveries: The algorithms do not use any other type of broadcast, and thus there is no ambiguity.

6.3 Reliable Broadcast

With the above assumptions, Reliable Broadcast can be easily implemented as follows. To broadcast a message, a process sends it to itself. When a process receives a message for the first time, it sends this message to all its neighbors, and then delivers it. This "message diffusion" algorithm is shown in Figure 11.

Some obvious optimizations are possible (e.g., if p receives m from q, it need not send m to q), but we do not consider such details here. It is important to note that this algorithm works even if the network is asynchronous.

Algorithm for process p:
To execute `broadcast`(R, m):
    `send`(m) `to` p

`deliver`(R, m) *occurs as follows:*
    upon `receive`(m) **do**
        if p has not previously executed `deliver`(R, m)
        then
            `send`(m) `to` all neighbors
            `deliver`(R, m)

Figure 11: Reliable Broadcast for Point-to-Point Networks

Theorem 8 *The algorithm in Figure 11 is a Reliable Broadcast.*[13]

[13] All the Reliable Broadcast algorithms in this section satisfy the Uniform version of Integrity. For brevity we omit to state this in our theorems.

`Proof:` We have to show that the algorithm satisfies the three properties of Reliable Broadcast, namely, Validity, Agreement, and Uniform Integrity.

Validity: If a correct process p broadcasts m, it sends m to itself. By Validity of `send` and `receive`, p receives m. So p delivers m.

Agreement: Let p and q be any correct processes. Suppose p delivers m. We must show that q also delivers m. By Assumption (b), there is a path $p_1 = p, p_2, \ldots, p_k = q$ consisting entirely of correct processes and links. By induction on i we prove that each p_i delivers m. The basis, $i = 1$, is true by assumption. For the induction step, suppose p_i delivers m; we show that p_{i+1} also delivers m. By the algorithm, since p_i delivers m, it must have previously sent m to every one of its neighbors, including p_{i+1}. By Validity of `send` and `receive`, p_{i+1} receives m and, being correct, delivers m.

Uniform Integrity: From the algorithm, p delivers m only if it has not previously executed `deliver`$(\texttt{R}, m)$, i.e., p delivers m at most once. Suppose some process delivers m; we must show that $sender(m)$ did in fact broadcast m. This follows from the Uniform Integrity of `send` and `receive`, and the fact that with benign failures if a process p broadcasts m then $sender(m) = p$. □

If we make further assumptions about the given point-to-point network, the algorithm in Figure 11 satisfies additional properties, namely, Uniform Causal Order, Uniform Agreement and Real-Time Δ-Timeliness. With a modification, it can also satisfy Local-Time Δ-Timeliness. This is shown in the next sections.

6.4 Achieving Uniform Causal Order

In many point-to-point networks, data-link layer transmission protocols provide reliable FIFO links. More precisely, they provide `send` and `receive` primitives that satisfy Validity and Uniform Integrity, as well as the following property:

- *Uniform FIFO Order:* If p sends m to q before it sends m' to q, then q does not receive m' unless it has previously received m.

With such links the algorithm given in the previous section is actually a Causal Broadcast!

Theorem 9 *Assume that the* `send` *and* `receive` *primitives satisfy Uniform FIFO Order and the* **upon** *statement is indivisible.*[14] *The Reliable Broadcast algorithm in Figure 11 satisfies Uniform Causal Order.*

`Proof:` We say that a process p *relays* m when it begins executing the statement "`send`(m) `to` all neighbors". Since p could fail during the execution of this statement, saying that p relayed m does not necessarily imply that all its correct neighbors will receive m.

[14] That is, each execution of the **upon** statement cannot be interrupted by another execution of this statement.

Claim: For any messages m and m', if $sender(m')$ relays m and m' in that order, then no process relays m' unless it has previously relayed m.

`Proof of Claim:` Suppose $sender(m')$ relays m and m', in that order. Assume, for contradiction, that some process relays m' without having previously relayed m. Let q be the first process to do so (in real time). Clearly, $q \neq sender(m')$. Thus q relayed m' because it previously received m' from a process $s \neq q$. By the Uniform Integrity of `send` and `receive`, s must have relayed m' to q. Note that the relay of m' by s precedes the one by q. By the choice of q, s relayed m before m'. So, s sent m to q before m'. By Uniform FIFO Order of `send` and `receive`, q received m before receiving m'. By the indivisibility of the **upon** statement, q relayed m before receiving m', and therefore before relaying m'. This contradicts the definition of q. □ Claim

We now show that the algorithm satisfies Uniform Causal Order by proving that it satisfies Uniform FIFO Order and Uniform Local Order (see Theorem 2). Suppose that $sender(m')$ broadcasts m or delivers m, before it broadcasts m'. We must show that no process delivers m' unless it has previously delivered m. We first prove that:

If $sender(m')$ relays m', then no process relays m' unless it has previously relayed m. (∗)

Assume that $sender(m')$ relays m'. By the above claim, it is sufficient to show that $sender(m')$ relayed m before m'. There are two cases to consider:

- $sender(m')$ delivers m before it broadcasts m'. From the algorithm, $sender(m')$ relayed m before delivering m, and it relayed m' after broadcasting m'. Thus, it relayed m before m'.
- $sender(m')$ broadcasts m before it broadcasts m'. From the algorithm, $sender(m')$ sends m to itself before m'. By the Uniform FIFO Order of `send` and `receive`, $sender(m')$ receives m before m'. By the indivisibility of the **upon** statement, $sender(m')$ relays m before m'.

This concludes the proof of (∗). Now consider a process q that delivers m'. We show that q delivers m before m'. Since q delivered m', it must have previously received and relayed m'. By Uniform Integrity of `send` and `receive`, it is easy to show that $sender(m')$ relayed m'. By (*), q relayed m before m'. By the indivisibility of the **upon** statement, q delivered m before m', as wanted. □

Note that the Uniform FIFO Order property of `send` and `receive`, applies even between two faulty processes (that is why we call it Uniform). This uniformity is crucial to Theorem 9: without it, the Reliable Broadcast algorithm in Figure 11 would not even satisfy FIFO Order, let alone Uniform Causal Order. The same is true about the required indivisibility of the **upon** statement.

6.5 Achieving Uniform Agreement

The Reliable Broadcast algorithm in Figure 11 works for any type of benign failures, provided that *correct* processes remain connected (see Assumption (b)). We now show that if

we restrict the type of failures and strengthen the connectivity assumption, this algorithm also satisfies *Uniform* Agreement.

Consider a point-to-point network where processes do *not* commit send-omission failures. With this restriction, the `send` and `receive` primitives satisfy a stronger Validity property than the one given in Section 6.1. If send-omission failures cannot occur then the following property holds (see Appendix):

- *Strong Validity:* If a process p (whether correct or faulty) completes the sending of a message m to a correct process q, and the link from p to q is correct, then q eventually receives m.

We can now show the following:

Theorem 10 *Consider a network such that: (1) processes do not commit send-omission failures, and (2) every process p (whether correct or faulty) is connected to every correct process via a path consisting entirely of correct processes and links (with the possible exception of p itself). The Reliable Broadcast algorithm in Figure 11 satisfies Uniform Agreement.*

`Proof:` We have already proved that the algorithm satisfies Validity and Uniform Integrity of Reliable Broadcast (cf. Theorem 8). It remains to prove that it satisfies:

Uniform Agreement: Let p be any process (correct of faulty) that delivers m, and let q be a correct process. We must show that q also delivers m. By assumption, there is a path $p_1 = p, p_2, \ldots, p_k = q$ consisting entirely of correct processes and links (with the possible exception of p). By induction on i we prove that each p_i delivers m. The basis, $i = 1$, is true by assumption. For the induction step, suppose p_i delivers m; we show that p_{i+1} also delivers m. By the algorithm, since p_i delivers m, it must have previously completed the sending of m to every one of its neighbors, including p_{i+1}. Since p_{i+1} and the link from p_i to p_{i+1} are correct, by Strong Validity of `send` and `receive` p_{i+1} eventually receives m. Since it is correct, p_{i+1} delivers m. □

Achieving Uniform Agreement in systems with send-omission and receive-omission failures can be done using techniques described in [NT90].

6.6 Impossibility of Δ-Timeliness in Asynchronous Systems

Consider an asynchronous point-to-point network. Clearly, no Reliable Broadcast algorithm can achieve Real-Time Δ-Timeliness in such a system. Can it achieve Local-Time Δ-Timeliness? If local clocks are required to satisfy the Clock Monotonicity property, the answer is negative. In fact, this impossibility result holds even if the network is completely connected, has correct links, at most one process may fail, and it can only fail by crashing. The proof is by contradiction. Suppose there is a Reliable Broadcast algorithm that satisfies Local-Time Δ-Timeliness in such a system. We could transform it into an Atomic Broadcast algorithm, as shown in Figure 4 (Section 5.3). This contradicts the impossibility of solving Atomic Broadcast in such a system (see Corollary 2 in Section 7.2.1).

6.7 Achieving Δ-Timeliness in Synchronous Systems

In contrast to the impossibility results above, both versions of Δ-Timeliness can be achieved in synchronous point-to-point networks: Real-Time Δ-Timeliness in networks with omission failures, and Local-Time Δ-Timeliness in networks with timing failures. To show this, we first state the properties of synchronous point-to-point networks.

6.7.1 Model of Synchronous Point-to-Point Networks

Roughly speaking, a point-to-point network is synchronous if there are known bounds on message delay, clock drift, and the time to execute a local step. More precisely, a point-to-point network is *synchronous* if, in addition to the properties in Section 6.1, it also satisfies the following three synchrony properties (in this definition the word "time" refers to real time):

1. τ-*Local Step:* There is a known constant $\tau \geq 0$ such that no process p completes a step later than τ time units of when it started that step.
2. ρ-*Clock Drift:* There is a known constant $\rho \geq 0$ such that every process p has a local clock whose drift with respect to time is bounded by ρ. That is, if $C_p(t)$ denotes the value of the local clock of p at time t, then for all $t > t'$:
$$\frac{1}{1+\rho} \leq \frac{C_p(t) - C_p(t')}{t - t'} \leq 1 + \rho$$
3. δ-*Timeliness:* There is a known constant $\delta \geq 0$ such that for any processes p and q connected by a link then q does not receive m after time $t + \delta$.

In a synchronous point-to-point network with *omission* failures the above properties are never violated. In one with *timing* failures, however, Properties 1 and 2 hold only if p is correct, and Property 3 holds only if p, q, and the link from p to q are all correct. In other words, in a synchronous point-to-point network with timing failures the synchrony properties hold only in the subnetwork consisting entirely of correct processes and links.

6.7.2 Assumptions

To achieve Δ-Timeliness we make the following assumptions:

c. f-*Failures:* There is a known upper bound f on the number of faulty processes.

d. d-*Diameter:* There is a known constant d such that every two correct processes are connected via a path of length at most d, consisting entirely of correct processes and links.

e. 0-*Local Step:* The time to execute a local step is 0. More precisely, we take $\tau = 0$ in Property 1 in the definition of synchronous point-to-point networks.

Regarding the first two assumptions, note that f and d can be taken to be the number of processes in the network, if no better bounds on these quantities are available. Regarding the third assumption, we note that it is not really necessary but it simplifies the algorithms and their analyses. Moreover, this assumption is reasonable for the type of algorithms that we are considering: The amount of local processing is negligible and can be absorbed in the upper bound on message delay δ.

6.7.3 Achieving Real-Time Δ-Timeliness

In a network with timing failures, no Reliable Broadcast can satisfy Real-Time Δ-Timeliness. Roughly speaking, the argument runs as follows. Suppose that the clocks of correct processes show real time, while faulty processes have clocks that are Δ' ahead of real time, for some $\Delta' > \Delta$. Assume that a faulty process p broadcasts a message m at real time $t - \Delta'$, i.e., at time t according to the clocks of faulty processes. Suppose the set of faulty processes "withhold" m for Δ' real-time units, and then "release" m into the rest of the network at real-time t. To any correct process q, it now seems that the broadcast of m was initiated at real time t, and that the sender of m, namely p, is actually correct. Thus, q has to assume that p delivers m (by Validity). To satisfy Agreement, q must also deliver m thereby violating Real-Time Δ-Timeliness.

In contrast, in a network where only omission failures occur (i.e., one where the synchrony assumptions are not violated), the Reliable Broadcast algorithm in Figure 11 does satisfy Real-Time Δ-Timeliness:

Theorem 11 *In a synchronous network with omission failures the Reliable Broadcast algorithm in Figure 11 satisfies Real-Time Δ-Timeliness, with $\Delta = (f + d)\delta$.*

`Proof:` We first show the following

Claim: For any two correct processes s and r, if s delivers a message m at real time t_s, then r delivers m at real time t_r such that $t_r \leq t_s + d\delta$.

`Proof of Claim:` Suppose s delivers m at real time t_s. By Assumption (d), there is a path $p_1 = s, p_2, \ldots, p_k = r$ (for some k, $1 \leq k \leq d + 1$) from s to r, consisting entirely of correct processes and links. We now show that:

$$\text{for all } i,\ 1 \leq i \leq k,\ p_i \text{ delivers } m \text{ by real time } t_s + (i-1)\delta. \qquad (*)$$

The proof of this is by induction on i. For $i = 1$, we have $p_1 = s$ and the induction hypothesis obviously holds. Suppose it holds for i; we show it also does for $i + 1$. By the induction hypothesis, p_i delivers m by real time $t_s + (i-1)\delta$. By the algorithm, just before delivering m, p_i sent m to all its neighbors, including p_{i+1}.

Since p_i, p_{i+1}, and the link between them are correct, by Validity and δ-Timeliness of `send` and `receive`, p_{i+1} receives m by real time $t_s + i\delta$. By Assumption (e), p_{i+1} delivers m by real time $t_s + i\delta$ — concluding the proof of (*). By (*), $p_k = r$ delivers m by real time $t_s + (k-1)\delta$. Noting that $k \leq d + 1$ concludes the proof of the claim. $\square_{\text{Claim}}$

We now prove that the algorithm satisfies Real-Time Δ-Timeliness.

Real-Time Δ-Timeliness: Suppose a process p broadcasts a message m at real time t, and some correct process q delivers m at real time t'. We must show that $t' \leq t + \Delta$, where $\Delta = (f + d)\delta$. If $p = q$, then $t' = t$ (by Validity of `send` and `receive`, the message delay of the "link" between p and itself is 0), and the result holds. Now assume $p \neq q$. From the algorithm, Uniform Integrity and δ-Timeliness of `send` and `receive`, and Assumption (e), there must be a sequence of $k \geq 2$ *distinct* processes $p_1 = p, p_2, \ldots, p_k = q$, such that for all i, $2 \leq i \leq k$, p_i receives m from p_{i-1} and delivers it by real time $t + (i-1)\delta$. Let p_j be the first *correct* process in this sequence. It delivers m by real time $t + (j-1)\delta$. Since $p_1, \ldots, p_{j-1}$ are faulty, and there are at most f faulty processes (Assumption (c)), $j - 1 \leq f$. Thus, p_j delivers m by real time $t + f\delta$. By the above claim, q delivers m at most $d\delta$ real-time units later, i.e., by real time $t + (f + d)\delta$. □

6.7.4 Achieving Local-Time Δ-Timeliness

Consider a synchronous point-to-point network with timing failures. In such networks, one can implement *approximately synchronized clocks*, i.e., clocks that are always close to each other and whose drift with respect to real time is bounded (for example, see [LM85,ST87a, Cri89]). More precisely these clocks satisfy:

f. (ϵ, ρ)-*Clock Synchronization:* There are known constants $\epsilon \geq 0$ and $\rho \geq 0$ such that for all correct processes p and q, and all real times $t > t'$:

- $|C_p(t) - C_q(t)| \leq \epsilon$, and
- $\frac{1}{1+\rho} \leq \frac{C_p(t) - C_p(t')}{t - t'} \leq 1 + \rho$

With such clocks, we can modify the Reliable Broadcast algorithm in Figure 11 so that it satisfies Local-Time Δ-Timeliness *even if timing failures occur*. The modified algorithm, shown in Figure 12, works as follows. With each message m we now associate a counter that indicates how many links m has traversed so far. Thus, processes send messages of the form $\langle m, k \rangle$, where k is the counter associated with message m. When a process wishes to broadcast a message m, it tags m with the local sending time $ts(m)$, and then sends $\langle m, 0 \rangle$ to itself. When a process p receives a message $\langle m, k \rangle$, p checks whether the local receipt time is less or equal to $ts(m) + k(\delta(1 + \rho) + \epsilon)$. If so, p relays $\langle m, k+1 \rangle$ to all its neighbors, and then delivers m. Otherwise, p simply discards m.

Theorem 12 *In a synchronous network with timing failures and approximately synchronized clocks (Assumption (f)), the algorithm in Figure 12 is a Reliable Broadcast that satisfies Local-Time Δ-Timeliness, with $\Delta = (f + d)\delta(1 + \rho) + (f + 1)\epsilon$.*

`Proof:` We first show the following

Claim: For any two correct processes s and r, if s delivers a message m at local time t_s, then r delivers m at local time t_r such that $t_r \leq t_s + d\delta(1 + \rho) + \epsilon$.

Algorithm for process p:
To execute `broadcast`$(\mathtt{R}, m)$:
    `send`$(\langle m, 0\rangle)$ `to` p

`deliver`$(\mathtt{R}, m)$ *occurs as follows:*
    upon `receive`$(\langle m, k\rangle)$ **do**
        if p has not previously executed `deliver`$(\mathtt{R}, m)$ **and**
            local time $\leq ts(m) + k(\delta(1+\rho)+\epsilon)$
        then
            `send`$(\langle m, k+1\rangle)$ `to` all neighbors
            `deliver`$(\mathtt{R}, m)$

Figure 12: Reliable Broadcast with Local-Time Δ-Timeliness

`Proof:` Suppose s delivers m at local time t_s on its clock. By Assumption (d), there is a path $p_1 = s, p_2, \ldots, p_k = r$ (for some k, $1 \leq k \leq d+1$) from s to r, consisting entirely of correct processes and links. We now show that: For all i, $1 \leq i \leq k$,

$$p_i \text{ delivers } m \text{ at time } t_s^i \leq t_s + (i-1)\delta(1+\rho) \textit{ according to the clock of } s. \qquad (*)$$

The proof of this is by induction on i. For $i = 1$, we have $p_1 = s$ and $t_s^1 = t_s$, and so the induction hypothesis holds. Suppose it holds for i; we now show it also does for $i+1$. By the induction hypothesis, p_i delivers m. By the algorithm, it does so upon receiving a message $\langle m, k_i\rangle$ at local time t^i (on p_i's clock) such that $t^i \leq ts(m) + k_i(\delta(1+\rho)+\epsilon)$. Furthermore, immediately after the receipt of $\langle m, k_i\rangle$, p_i sends $\langle m, k_i+1\rangle$ to all its neighbors, including p_{i+1}. By Assumption (e) p_i sends $\langle m, k_i+1\rangle$ to p_{i+1} at time t^i (on p_i's clock). Since p_i, p_{i+1}, and the link between them are correct, by Validity of `send` and `receive`, p_{i+1} eventually receives $\langle m, k_i+1\rangle$. By δ-Timeliness of `send` and `receive`, this receipt occurs within δ units of real time from the time p_i sent $\langle m, k_i+1\rangle$. Thus, by Assumption (f), p_{i+1} receives $\langle m, k_i+1\rangle$ at local time $t^{i+1} \leq t^i + \delta(1+\rho) + \epsilon$ on its clock. Note that $t^{i+1} \leq ts(m) + (k_i+1)(\delta(1+\rho)+\epsilon)$. Thus, by the algorithm and Assumption (e), p_{i+1} delivers m by local time t^{i+1}. By the induction hypothesis, p_i delivers m at time $t_s^i \leq t_s + (i-1)\delta(1+\rho)$, according to the clock of s. Since p_{i+1} delivers m within δ units of real time from this, p_{i+1} delivers m at time $t_s^{i+1} \leq t_s + i\delta(1+\rho)$, also according to the clock of s. Thus, the induction hypothesis holds for $i+1$ — concluding the proof of (*).

By (*), process $p_k = r$ delivers m at time $t_s^k \leq t_s + (k-1)\delta(1+\rho)$, according to the clock of s. By Assumption (f), this delivery occurs at time $t_r \leq t_s^k + \epsilon$ on r's clock, i.e., at time $t_r \leq t_s + (k-1)\delta(1+\rho) + \epsilon$. Noting that $k \leq d+1$ concludes the proof of the claim. □ Claim

We now prove that the algorithm satisfies the three properties of Reliable Broadcast as well as Local-Time Δ-Timeliness.

Validity: If a correct process p broadcasts m, it sends $\langle m, 0\rangle$ to itself at local time $ts(m)$.

By δ-Timeliness of `send` and `receive`, the message delay of the "link" between p and itself is 0. Thus, p receives $\langle m, 0\rangle$ at local time $ts(m)$. By the algorithm, p delivers m.

Agreement: Immediate from the above claim.

Uniform Integrity: The proof, similar to the one given for the Algorithm in Figure 11, is omitted.

Local-Time Δ-Timeliness: Suppose some correct process q delivers a message m at local time t_q. We must show that $t_q \leq ts(m) + \Delta$, where $\Delta = (f+d)\delta(1+\rho) + (f+1)\epsilon$. If $q = sender(m)$, then $t_q = ts(m)$ (by Validity of `send` and `receive`, the message delay of the "link" between q and itself is 0), and the result holds. Now assume $q \neq sender(m)$. In this case, from the algorithm and Uniform Integrity of `send` and `receive`, there is a sequence of $k \geq 2$ *distinct* processes $p_1 = sender(m), p_2, \ldots, p_k = q$ such that for all i, $1 \leq i \leq k-1$, p_i receives $\langle m, i-1\rangle$ by local time $t_i \leq ts(m) + (i-1)(\delta(1+\rho)+\epsilon)$, and sends $\langle m, i\rangle$ to p_{i+1} (before delivering m). Let p_j be the first *correct* process in this sequence. It receives $\langle m, j-1\rangle$ by local time $t_j \leq ts(m) + (j-1)(\delta(1+\rho)+\epsilon)$ on its clock. Since $p_1, \ldots, p_{j-1}$ are faulty, and there are at most f faulty processes (Assumption (c)), $j-1 \leq f$. Thus, $t_j \leq ts(m) + f(\delta(1+\rho)+\epsilon)$. By the above claim, since p_j delivers m at local time t_j, q delivers m at local time $t_q \leq t_j + d\delta(1+\rho)+\epsilon$. Thus, $t_q \leq ts(m) + f(\delta(1+\rho)+\epsilon) + d\delta(1+\rho)+\epsilon$, i.e., $t_q \leq ts(m) + \Delta$, where $\Delta = (f+d)\delta(1+\rho) + (f+1)\epsilon$. □

Observation: As in Theorem 9, if the links are FIFO then the algorithm in Figure 12 is actually a *Causal Broadcast* that satisfies Local-Time Δ-Timeliness! More precisely, this holds when the `send` and `receive` primitives satisfy Uniform FIFO Order and the **upon** statement is indivisible.

6.8 Obtaining Stronger Broadcasts

We now describe how to obtain algorithms for every type of broadcast in point-to-point networks. Starting from the Reliable Broadcast algorithm in Figure 11, we apply the generic transformation that adds FIFO Order (Figure 5) to obtain a FIFO Broadcast algorithm. We then add Causal Order by applying either the non-blocking transformation in Figure 6, or the blocking one in Figure 7. This results in two Causal Broadcast algorithms. Note that the algorithms that we get this way satisfy the *uniform* versions of FIFO and Causal Order.

To obtain Atomic, FIFO Atomic, or Causal Atomic Broadcast algorithms in a synchronous point-to-point network, we can proceed as follows:[15]

1. Run a clock synchronization algorithm that yields approximately synchronized clocks (Assumption (f) in Section 6.7.4) that also satisfy the Clock Monotonicity property.[16]

2. With such clocks, the Reliable Broadcast algorithm in Figure 12 satisfies Local-Time Δ-Timeliness.

[15] Recall that Atomic Broadcast cannot be solved in asynchronous systems.

[16] Typically, clock synchronization algorithms satisfy this property, or can be easily modified to do so.

3. Obtain FIFO and Causal Broadcast algorithms by successively applying the blocking transformations of Figure 5 and 7 to the Reliable Broadcast algorithm of Step 2. Under reasonable assumptions about the local clocks, these transformations preserve Local-Time Δ-Timeliness. Since the given Reliable Broadcast algorithm satisfies this property, so do the resulting FIFO and Causal Broadcast algorithms. [17]

4. Apply the transformation that adds Total Order (Figure 4) to the Reliable, FIFO, and Causal Broadcast algorithms that were obtained in the previous steps. This gives Atomic, FIFO Atomic and Causal Atomic Broadcast algorithms.

Another way to obtain FIFO Atomic and Causal Atomic Broadcast algorithms is: (1) Derive an Atomic Broadcast by adding Total Order to our Timed Reliable Broadcast as explained above, (2) add FIFO Order (recall that this transformation preserves Total Order), and (3) add Causal Order by applying one of the three non-blocking transformations of Section 5.6.

If a synchronous point-to-point network has FIFO links, one can also implement a Causal Atomic Broadcast algorithm as follows. Start with the algorithm in Figure 12. As we observed in Section 6.7.4, this is already a Causal Broadcast that satisfies Local-Time Δ-Timeliness. Now apply the transformation that adds Total Order (Figure 4).

Bibliographic Notes

The algorithm for Reliable Broadcast that satisfies Local-Time Δ-Timeliness and tolerates timing failures (Figure 12) is due to [CASD85]. That paper also presented an Atomic Broadcast algorithm for point-to-point networks using the idea of delaying the delivery of a message m until local time $ts(m) + \Delta$. This technique is the basis of our generic transformation that adds Total Order to any type of Timed Broadcast (Figure 4). Many other broadcast algorithms for specific network types, synchrony assumptions, and failure models have appeared in the literature, including those described in [CM84,BD85,BJ87, PBS89,GSTC90,VM90,BSS91,GMS91,ADKM92].

7 Consensus

7.1 Specification

In the Consensus problem, all correct processes propose a value and must agree on some value related to the proposed values. Formally, we define the Consensus problem in terms of two primitives, `propose` and `decide`. If p invokes `propose` with a value v as a parameter we say that p *proposes* v; in our algorithms we denote this invocation by "`propose`(v)".

[17] In general we cannot apply the non-blocking transformation from FIFO to Causal Broadcast described in Figure 6, because it does not preserve Δ-Timeliness unless the given FIFO Broadcast satisfies *Uniform* Agreement. We can apply this transformation if, in Step 1, we start with an Reliable Broadcast that satisfies Uniform Agreement.

The value proposed is taken from some set $\mathcal{V}$. When a process q returns from the execution of `decide` with value v, we say that q *decides* v; we denote this by "`decide`(v)". The *Consensus* problem requires that if each correct process proposes a value then the following hold:

- *Termination:* Every correct process eventually decides exactly one value.
- *Agreement:* If a correct process decides v, then all correct processes eventually decide v.
- *Integrity:* If a correct process decides v, then v was previously proposed by some process.

Integrity ensures that the decision of a correct process is not created "out of thin air". In particular, if all processes that propose a value, propose the same value v, then this decision can only be v. As usual, we can strengthen the Agreement and Integrity properties by requiring Uniformity:

- *Uniform Agreement:* If a process (whether correct or faulty) decides v, then all correct processes eventually decide v.
- *Uniform Integrity:* If a process (whether correct or faulty) decides v, then v was previously proposed by some process.

Now consider the specification of Consensus in the case of arbitrary failures. If a process p is subject to such failures, the meaning of "p proposes v" or "p decides v" is now ambiguous. To circumvent this problem, the properties of Consensus should now refer only to proposals and decisions of *correct* processes (see Section 3.10). This already holds for Termination and Agreement, but for arbitrary failures Integrity must be redefined as follows:

- *Integrity:* If all processes are correct and a process decides v, then v was previously proposed by some process.

7.2 Relating Consensus and Atomic Broadcast

In this section we examine the relation between Consensus and Atomic Broadcast. We shall see that the two problems are, under certain conditions, equivalent to each other. That is we can transform any algorithm for one into an algorithm for the other.

The transformation from Atomic Broadcast to Consensus tolerates any number of benign failures. The one from Consensus to Atomic Broadcast assumes that Reliable Broadcast is available and that only crash failures occur.[18] Both transformations make no assumptions on the type or synchrony of the communication network.

[18] A more complex transformation can actually work for any type of failures [CT92], but it is beyond the scope of this paper.

These two transformations have important consequences regarding the solvability of Atomic Broadcast in *asynchronous* point-to-point networks with crash failures:

1. Atomic Broadcast can *not* be solved, even if we assume that links are reliable, at most one process may fail, and it can only fail by crashing (Corollary 2).
2. Atomic Broadcast can be solved using *randomization* or *failure detectors* (Corollary 5).

7.2.1 Transforming Atomic Broadcast to Consensus

In Figure 13, we show how to transform any Atomic Broadcast algorithm into a Consensus algorithm. To propose a value v, a process uses the given Atomic Broadcast algorithm to A-broadcast v. To decide a value, a process selects the value of the first message that it A-delivers. By Agreement and Total Order of Atomic Broadcast, all correct processes choose the same value; hence Agreement of Consensus is satisfied. It is easy to verify that Termination and Integrity of Consensus also hold. This transformation makes no assumptions on the type or synchrony of the communication network, and it tolerates any number of benign failures.

Algorithm for process p:
To execute `propose`(v):
    `broadcast`$(\texttt{A}, v)$

`decide`$(-)$ *occurs as follows:*
    upon `deliver`$(\texttt{A}, u)$ **do**
        if p has not previously executed `deliver`$(\texttt{A}, -)$
        then `decide`(u)

Figure 13: Transforming Atomic Broadcast into Consensus

Theorem 13 *The algorithm in Figure 13 transforms any Atomic Broadcast algorithm into a Consensus algorithm.*

Corollary 2 *Atomic Broadcast cannot be solved in an asynchronous point-to-point network, even if the network is completely connected, all the links are correct, at most one process may fail, and it can only fail by crashing.*

`Proof:` It is well-known that Consensus cannot be solved in such a system [FLP85]. The result now follows from the previous theorem. □

The impossibility of Atomic Broadcast in asynchronous systems seems paradoxical since this primitive is a basic service provided by many practical systems which, on the

face of it, appear to be asynchronous. (Such systems include Isis [BJ87,BCJ+90], Amoeba [Kaa92], Delta-4 [VM90], and Transis [ADKM92].) There is no contradiction here. What this indicates is that such systems, at some level, explicitly or implicitly use of one of the mechanisms previously mentioned for circumventing the impossibility result. For example, the Isis Atomic Broadcast algorithm uses a failure detector mechanism based on timeouts. This means that it relies, at some level, on synchrony assumptions.

We close this section with a word of caution against confusing the impossibility of Atomic Broadcast and Consensus in asynchronous systems with a seemingly similar but, in fact, quite different impossibility result in fault-tolerant distributed computing, sometimes known as "the generals' paradox" [Gra78]. This concerns a problem, technically known as *Non-Blocking Atomic Commitment* [BHG87], that cannot be solved if the communication network may partition into two or more components, so that no messages can be exchanged between processes in different components. It is the possibility of network partioning that makes Non-Blocking Atomic Commitment unsolvable. In fact, this problem cannot be solved even if the network is *synchronous* (i.e., each message is either delivered within a known bound or not at all). In contrast, the impossibility of Consensus is due to the combination of asynchrony and process failures, and it holds even if communication is reliable, i.e., even if partitioning cannot occur. The difference in the reasons underlying these two impossibility results is reflected in their proofs, which are based on entirely different ideas (compare [FLP85] and [Gra78]).

7.2.2 Transforming Reliable Broadcast and Consensus to Atomic Broadcast

In Figure 14, we show how to transform any Reliable Broadcast and Consensus algorithms into an Atomic Broadcast algorithm [CT91]. This transformation uses repeated (possibly concurrent but completely *independent*) executions of Consensus. Informally, the kth execution of Consensus is used to decide on the kth batch of messages to be A-delivered. Processes disambiguate between these executions by tagging all the messages pertaining to the kth execution of Consensus with a counter k. Tagging with such counters constitutes a minor modification to any given Consensus algorithm. The `propose` and `decide` primitives corresponding to the kth execution of Consensus are denoted by `propose`$(k, -)$ and `decide`$(k, -)$.

When a process wishes to A-broadcast a message m, it uses the given Reliable Broadcast algorithm to R-broadcast m (Task 1). When a process p R-delivers m, it adds m to the set $R\_delivered_p$ (Task 2). Thus, $R\_delivered_p$ contains all the messages submitted for Atomic Broadcast (since the beginning) that p is aware of. When p A-delivers a message m, it adds m to the set $A\_delivered_p$ (in Task 3). Thus, $R\_delivered_p - A\_delivered_p$ is the set of messages that were submitted for Atomic Broadcast but not yet A-delivered, according to p. This set is denoted by $A\_undelivered_p$. Process p periodically checks whether $A\_undelivered_p$ is not empty; if so, p participates in the next execution of Consensus, say the kth one, by proposing $A\_undelivered_p$ as the kth batch of messages to be A-delivered. It then waits for the decision $msgSet$ of this Consensus execution. Finally, it A-delivers all the messages in $msgSet$ except those it previously A-delivered. More precisely, p A-delivers all the messages in $batch_p(k) = msgSet - A\_delivered_p$, and it does so in some deterministic

Algorithm for process p:

Initialization:
$R\_delivered := \emptyset$
$A\_delivered := \emptyset$
$k := 0$

To execute `broadcast`$(\mathtt{A}, m)$: { *Task 1* }

`broadcast`$(\mathtt{R}, m)$

`deliver`$(\mathtt{A}, -)$ *occurs as follows*:

upon `deliver`$(\mathtt{R}, m)$ **do** { *Task 2* }
$R\_delivered := R\_delivered \cup \{m\}$

do forever { *Task 3* }
$A\_undelivered := R\_delivered - A\_delivered$
if $A\_undelivered \neq \emptyset$ **then**
$k := k + 1$
`propose`$(k, A\_undelivered)$
wait for `decide`$(k, msgSet)$
$batch(k) := msgSet - A\_delivered$
A-deliver all messages in $batch(k)$ in some deterministic order
$A\_delivered := A\_delivered \cup batch(k)$

Figure 14: Transforming Consensus and Reliable Broadcast into Atomic Broadcast

order that was agreed *a priori* by all processes, e.g., in lexicographical order.

This transformation assumes that the given Reliable Broadcast algorithm satisfies Uniform Integrity, and that only crash failures occur. On the other hand, it makes no assumptions on the type or synchrony of the communication network, and it tolerates any number of crash failures.

The following sequence of lemmata show that the transformation in Figure 14 results in an algorithm that satisfies all the properties of Atomic Broadcast.

Lemma 1 *For any two correct processes p and q, and any message m, if $m \in R\_delivered_p$ then eventually $m \in R\_delivered_q$.*

`Proof:` If $m \in R\_delivered_p$ then p R-delivered m (in Task 2). Since p is correct, by Agreement of Reliable Broadcast q eventually R-delivers m, and inserts m into $R\_delivered_q$. □

Lemma 2 *For all correct processes p and all $k \geq 1$:*

1. *$batch_p(k)$ is a set of messages.*
2. *p does not A-deliver the messages in $batch_p(k+1)$ unless it has previously A-delivered those in $batch_p(k)$.*

`Proof:` Let v be p's decision value of the kth execution of Consensus. By Integrity of Consensus, some process must have proposed v. By the algorithm, v is a set of messages. Part (1) of the lemma follows from the observation that $batch_p(k) = v - A\_delivered_p$, and $A\_delivered_p$ is also a set of messages. Part (2) of the lemma is obvious from Task 3. □

Lemma 3 *For any two correct processes p and q, and all $k \geq 1$:*

1. *If p executes* `propose`$(k, -)$*, then q eventually executes* `propose`$(k, -)$.
2. *If p A-delivers messages in $batch_p(k)$, then q eventually A-delivers messages in $batch_q(k)$, and $batch_p(k) = batch_q(k)$.*

`Proof:` The proof is by simultaneous induction on (1) and (2). For $k = 1$, we first show that if p executes `propose`$(1, -)$, then q eventually executes `propose`$(1, -)$. When p executes `propose`$(1, -)$, $R\_delivered_p$ must contain some message m. By Lemma 1, m is eventually in $R\_delivered_q$. Since $A\_delivered_q$ is initially empty, eventually $R\_delivered_q - A\_delivered_q \neq \emptyset$. Thus, q eventually executes Task 3 and `propose`$(1, -)$.

We now show that if p A-delivers the messages in $batch_p(1)$, then q eventually A-delivers the messages in $batch_q(1)$, and $batch_p(1) = batch_q(1)$. From the algorithm, if p A-delivers messages in $batch_p(1)$, it previously executed `propose`$(1, -)$. From part (1) of the lemma, all correct processes eventually execute `propose`$(1, -)$. By Termination of Consensus, every

correct process eventually executes `decide`$(1, -)$ exactly once. By Agreement of Consensus, all correct processes execute `decide`$(1, msgSet)$ with the same $msgSet$. Since $A\_delivered_p$ and $A\_delivered_q$ are initially empty, $batch_p(1) = batch_q(1) = msgSet$.

Now assume that the lemma holds for all k, $1 \leq k < l$. We first show that if p executes `propose`$(l, -)$, then q eventually executes `propose`$(l, -)$. When p executes `propose`$(l, -)$, $R\_delivered_p$ must contain some message m that is not in $A\_deliver_p$. Thus, m is not in $\bigcup_{k=1}^{l-1} batch_p(k)$. By the induction hypothesis, $batch_p(k) = batch_q(k)$ for all $1 \leq k \leq l-1$. So m is not in $\bigcup_{k=1}^{l-1} batch_q(k)$. Since m is in $R\_delivered_p$, by Lemma 1, m is eventually in $R\_delivered_q$. Thus, there is a time after q A-delivers $batch_q(l-1)$ such that m is in $R\_delivered_q - A\_delivered_q$. So q eventually executes Task 3 and `propose`$(l, -)$.

We now show that if p A-delivers messages in $batch_p(l)$, then q A-delivers messages in $batch_q(l)$, and $batch_p(l) = batch_q(l)$. Since p A-delivers messages in $batch_p(l)$, it must have executed `propose`$(l, -)$. By part (1) of this lemma, all correct processes eventually execute `propose`$(l, -)$. By Termination of Consensus, every correct process eventually executes `decide`$(l, -)$ exactly once. By Agreement of Consensus, all correct processes execute `decide`$(l, msgSet)$ with the same $msgSet$. Note that $batch_p(l) = msgSet - \bigcup_{k=1}^{l-1} batch_p(k)$, and $batch_q(l) = msgSet - \bigcup_{k=1}^{l-1} batch_q(k)$. By the induction hypothesis, $batch_p(k) = batch_q(k)$ for all $1 \leq k \leq l-1$. Thus $batch_p(l) = batch_q(l)$. □

Corollary 3 *Agreement and Total Order of Atomic Broadcast are satisfied.*

`Proof:` Immediate from Lemmata 2, 3, and the fact that correct processes A-deliver messages in each batch in the same deterministic order. □

Lemma 4 *Validity of Atomic Broadcast is satisfied.*

`Proof:` The proof is by contradiction. Suppose some correct process p A-broadcasts m but never A-delivers m. By Corollary 3, no correct process A-delivers m. By Task 1 of the algorithm, p R-broadcasts m. By Validity and Agreement of Reliable Broadcast, every correct process q eventually R-delivers m, and inserts m in $R\_delivered_q$ (Task 2). Since correct processes never A-deliver m, they never insert m in their $A\_delivered$ set. Thus, for every correct process q, there is a time after which m is *permanently* in $R\_delivered_q - A\_delivered_q$. By the algorithm and Lemma 3, it is easy to show there is a k_1, such that for all $l > k_1$, all correct processes execute `propose`$(l, -)$, and they do so with sets that always include m.

Since all faulty processes eventually crash, there is a k_2 such that no faulty process executes `propose`$(l, -)$ with $l > k_2$. Let $k = max\{k_1, k_2\}$. Since all correct processes execute `propose`$(k, -)$, by Termination and Agreement of Consensus, all correct processes execute `decide`$(k, msgSet)$ with the same $msgSet$. By Integrity of Consensus, some process q executed `propose`$(k, msgSet)$. From our definition of k, q is correct and $msgSet$ contains m. Thus all correct processes, including p, A-deliver m — a contradiction that concludes this proof. □

Lemma 5 *Integrity of Atomic Broadcast is satisfied.*

`Proof:` We must show that for any message m, every correct process A-delivers m at most once, and only if m was A-broadcast by $sender(m)$. Suppose a correct process p A-delivers m. After p A-delivers m, it inserts m in $A\_delivered_p$. From the algorithm, it is clear that p cannot A-deliver m again. Furthermore, since p A-delivers m, it must have previously executed `decide`$(k, msgSet)$ for some k and some $msgSet$ that contains m. By Integrity of Consensus, some process q must have executed `propose`$(k, msgSet)$. So q previously R-delivered all the messages in $msgSet$, including m. By Uniform Integrity of Reliable Broadcast, $sender(m)$ R-broadcast m, and therefore it A-broadcast m. □

Theorem 14 *In a system with crash failures, the algorithm in Figure 14 transforms any algorithms for Reliable Broadcast and Consensus into an Atomic Broadcast algorithm.*

`Proof:` Immediate from Lemmata 4, 5, and Corollary 3. □

From the proof of Lemma 5, it is easy to see that if the given Consensus algorithm satisfies *Uniform* Integrity, the resulting Atomic Broadcast also does so. From Theorems 13 and 14:

Corollary 4 *Consensus and Atomic Broadcast are equivalent in any system with crash failures where Reliable Broadcast can be implemented.*

Theorem 14 does not make any assumption on the type or synchrony of the communication network. Now consider asynchronous point-to-point networks with crash failures. In such systems, Reliable Broadcast can be implemented (see Section 6), and Consensus can be solved using *randomization* [CD89] or *failure detectors* [CT91]. These results and Theorem 14 imply:

Corollary 5 *In asynchronous point-to-point networks with crash failures, Atomic Broadcast can be implemented using randomization or failure detectors.*

Bibliographic Notes

Fischer, Lynch and Paterson [FLP85] proved the basic result that Consensus is not solvable deterministically in asynchronous systems. Soon after this result was first published, it was shown that Consensus can be solved with randomized algorithms in asynchronous systems. Such algorithms include [Ben83,Rab83,Bra87], and are surveyed in [CD89]. Unreliable failure detectors were introduced by Chandra and Toueg in [CT91], who gave Consensus algorithms based on failure detectors of varying strength. [CT91] also shows that Reliable Broadcast together with Consensus can be transformed into Atomic Broadcast (Theorem 14). [CHT92] determines the weakest failure detector that can be used to solve Consensus. Ricciardi and Birman consider failure detectors in the context of the group membership problem in [RB91].

Agreement-like problems that are solvable in asynchronous systems in the presence of failures are described in [DLP+86,ABD+87,BW87,Fek90,Fek93]. [BMZ88] gives a graph-theoretic characterization of the problems that can be solved (deterministically) in asyn-

chronous systems with one crash failure. Similar results in the context of shared-memory distributed systems are given by [BG93,HS93,SZ93].

8 Terminating Reliable Broadcast

8.1 Specification

Recall that with Reliable Broadcast any process is allowed to broadcast any message from a set $\mathcal{M}$ of possible messages, at any time. In particular, processes have no *a priori* knowledge of the impending broadcasts. Thus, as we noted before, if a process p fails immediately after invoking the broadcast primitive, the correct processes cannot be required to deliver any message, as they were not even aware of p's intention to broadcast.

In contrast, in some applications there is *a priori* knowledge that a particular process, say sender s, is supposed to broadcast a single message in $\mathcal{M}$. For example, a distributed control system may have a temperature sensor process s that is supposed to reliably broadcast the temperature at a particular time to three monitoring processes. This broadcast is an instance of *Terminating Reliable Broadcast for sender* s, a type of broadcast that requires correct processes to *always* deliver a message ("from" s) — even if the sender s is faulty and, say, crashes before the broadcast! For this requirement to be satisfiable, processes must be allowed to deliver a message that was not actually broadcast by s. Thus, we now allow the delivery of a special message $\mathtt{F}_s \notin \mathcal{M}$ stating that the sender s is faulty. By convention, we assume $sender(\mathtt{F}_s) = s$.

With *Terminating Reliable Broadcast* (*TRB*) *for sender* s, s can broadcast any message $m \in \mathcal{M}$, processes can deliver any message $m \in \mathcal{M} \cup \{\mathtt{F}_s\}$, and the following hold:

- *Termination:* Every correct process eventually delivers exactly one message.
- *Validity:* If s is correct and broadcasts a message m, then it eventually delivers m.
- *Agreement:* If a correct process delivers a message m, then all correct processes eventually deliver m.
- *Integrity:* If a correct process delivers a message m then $sender(m) = s$. Furthermore, if $m \neq \mathtt{F}_s$ then m was previously broadcast by s.

The reader should verify that the specification of TRB for sender s implies that a correct process delivers $\mathtt{F}_s$ only if s is faulty. Just as with Reliable Broadcast, we can strengthen TRB by requiring it to satisfy Uniform Agreement, or one of the two versions of Δ-Timeliness.

This problem has been studied extensively in the case of arbitrary failures under the name of "Byzantine Agreement" or "Byzantine Generals' Problem". If the sender s is subject to arbitrary failures, the second clause of Integrity, namely the sentence "Furthermore, if $m \neq \mathtt{F}_s$ then m was previously broadcast by s", is now ambiguous. To circumvent this problem in the usual manner (see Section 3.10), we must reformulate this statement so that

it applies only to the case that s is correct. In that case, however, the other three properties of TRB already determine that correct processes must deliver the message broadcast by s, and only that. Hence, for arbitrary failures, we can simply drop this second clause, and Integrity reduces to: "If a correct process delivers a message m then $sender(m) = s$."

8.2 Relating Consensus and Terminating Reliable Broadcast

In this section we relate Consensus and Terminating Reliable Broadcast:

- In some *synchronous* point-to-point networks, Consensus is *equivalent* to TRB. This allows us to translate both positive and negative results proven for one problem, to the other. For example, the transformation from Consensus to TRB is message- and time-efficient. In particular, it can convert any constant-time randomized Consensus algorithm (such as the one in [FM90]), into a constant-time randomized TRB algorithm.
- In *asynchronous* systems, these two problems are not equivalent: TRB can be transformed to Consensus, but the converse does not hold (Consensus is solvable with randomization, but TRB is not).

8.2.1 Transforming Terminating Reliable Broadcast to Consensus

The transformation from TRB to Consensus requires the concurrent execution of several independent copies of the given TRB algorithm, one for each process as the sender. Let TRB(p) denote the copy of TRB for sender p, and $\mathcal{F}$ be the set $\{\mathtt{F}_q | \text{ for all processes } q\}$.

The transformation is shown in Figure 15. To propose a value v (we assume that $v \notin \mathcal{F}$), a process p uses TRB(p) to broadcast v. When p delivers a value from a process q, it inserts that value into entry $V[q]$ of a vector V that has one entry per process. Once all the entries of V have been filled, p decides the first non-$\mathcal{F}$ value in V (i.e., the first component of V whose value is not in $\mathcal{F}$).

This transformation makes no assumptions on the type or synchrony of the communication network, and it works with any type and number of benign failures.

Theorem 15 *The algorithm in Figure 15 transforms any Terminating Reliable Broadcast algorithm into a Consensus algorithm.*

`Proof:` We show that the algorithm that results from the transformation satisfies the properties of Consensus.

Termination and Integrity: Consider any correct process p. From the structure of the algorithm, it is clear that p decides at most once. We now show that p does decide, and that its decision value was previously proposed by some process. Suppose p proposes, and hence broadcasts, v. By assumption, $v \notin \mathcal{F}$. By Validity of TRB(p), p delivers v. Since $sender(v) = p$, it sets $V[p] := v$. By Termination and Integrity of TRB, p never delivers

Algorithm for process p:
To execute `propose`(v):
    `broadcast`(`TRB`, v)

`decide`$(-)$ *occurs as follows:*
    $V := \langle \perp, \perp, \ldots, \perp \rangle$
    cobegin
        upon `deliver`(`TRB`, u) **do** $V[sender(u)] := u$
    //
        wait for $(\forall q, V[q] \neq \perp)$
            `decide`(first non-$\mathcal{F}$ value of V)
    coend

Figure 15: Transforming Terminating Reliable Broadcast into Consensus

any other message u such that $sender(u) = p$. Therefore p never sets $V[p]$ to a value u different than v.

By Termination of TRB(q) for each process q, p eventually delivers exactly one message from q. Thus, p's execution of the **wait for** statement eventually terminates, and at that time V contains at least one non-$\mathcal{F}$ value, namely $V[p] = v$. Therefore, p eventually decides a non-$\mathcal{F}$ value u, where $u = V[s]$ for some process s. From the algorithm, p previously delivered u and $sender(u) = s$. By Integrity of TRB, the delivery of u occurred during the execution of TRB(s). By Integrity of TRB(s), u was previously broadcast by s. Thus, u was previously proposed by s.

Agreement: By Agreement of TRB(q) for each process q, all correct processes have the same vector V when they use it to decide. Thus, they all decide the same value. □

8.2.2 Transforming Consensus into Terminating Reliable Broadcast

In Figure 16 we show how to transform any Consensus algorithm into a TRB algorithm for any given sender s. In contrast to the converse transformation, this one requires several assumptions: The system is a synchronous point-to-point network, it is completely connected with no link failures, and faulty processes are subject to send-omission failures only. Furthermore, we assume that processes have *a priori* knowledge of the time t_0 at which the sender s is supposed to broadcast, and have perfect real time clocks (i.e., $C_p(t) = t$ for all processes p). Recall that in a synchronous point-to-point network there is a known upper bound δ on message transmission time over a link (Condition 3 in Section 6.7).

To broadcast a message m, s sends m to every process at the designated time t_0. At time $t_0+\delta$, if a process previously received a message from s then it uses the given Consensus algorithm to propose that message, otherwise it proposes $\mathtt{F}_s$. To deliver a message, a process

waits for the decision value of the Consensus algorithm and delivers it.

`broadcast(TRB,`m`)` *occurs as follows:*
  The sender s *at time* t_0*:*
    `send(`m`) to` all processes { $m \neq \mathtt{F}_s$ }

  Every process at time $t_0 + \delta$*:*
    if `receive(`m`) from` s by time $t_0 + \delta$
    then `propose(`m`)`
    else `propose(F`$_s$`)`

`deliver(TRB,`$-$`)` *occurs as follows:*
  Every process:
    upon `decide(`v`)` **do** { v is a message m or $\mathtt{F}_s$ }
      `deliver(TRB,`v`)`

Figure 16: Transforming Consensus into Terminating Reliable Broadcast for Sender s

---

Theorem 16 *Consider a synchronous point-to-point network that satisfies the assumptions listed above. The algorithm in Figure 16 transforms any Consensus algorithm into a Terminating Reliable Broadcast algorithm for any given sender s.*

The proof is omitted.

9 Multicast Specifications

So far we have assumed that each broadcast is targeted to all the processes in the system. In some applications, the system is configured as a collection of (possibly overlapping) groups, each consisting of a subset of processes. A *multicast* is a broadcast that is targeted exclusively to the members of some particular group. We shall assume that groups are static, and that each process knows to which groups it belongs and the members of each of these groups. We shall not address the question of how these groups are formed, or how processes join or leave them. This is the *group membership* problem discussed in [Cri90a,RB91], and is outside the scope of this paper.

Formally, a *group* G is a name for a subset of the processes in the system. We say that process p *is in* G (or p *is a member of* G), and write $p \in G$, when p is in the subset of processes named G. Since each message m is multicast to a particular group, it is tagged with the name of that group, i.e., we assume that m has a field denoted $group(m)$.

Multicasts are defined in terms of two primitives, `multicast` and `deliver`. When a process p invokes `multicast` with a message m as a parameter, we say that p *multicasts* m

(to group(m)). We assume that m is taken from a set a set $\mathcal{M}$ of possible messages. When a process q returns from the execution of `deliver`(m) with message m as the returned value, we say that *q delivers m (in group(m))*. As with broadcasts, our definitions of multicasts assume benign failures. The modifications for arbitrary failures are similar to those given in Section 3.10 for broadcasts, and are omitted.

9.1 Reliable Multicast

Reliable Multicast is essentially the Reliable Broadcast problem adapted to groups. It is the basis for all the types of multicasts that we consider. Informally, Reliable Multicast requires that all correct processes in each group G deliver the same set of messages, that this set include all the messages multicast to G by correct processes, and that no spurious messages are ever delivered. More precisely, *Reliable Multicast* is defined in terms of `multicast` and `deliver` primitives that satisfy the following:

- *Validity:* If a correct process multicasts a message m, then some correct process in $group(m)$ eventually delivers m or no process in that group is correct.
- *Agreement:* If a correct process delivers a message m, then all correct processes in $group(m)$ eventually deliver m.
- *Integrity:* For any message m, every correct process p delivers m at most once, and only if p is in $group(m)$ and m was previously multicast by $sender(m)$.

The reader should verify that Reliable Multicast is a generalization of Reliable Broadcast: if $group(m)$ is the set of all processes then the above specification is equivalent to the specification of Reliable Broadcast given in Section 3. The key difference from Reliable Broadcast is that only processes in $group(m)$ can deliver m.

In some applications processes must be able to multicast messages to groups to which they do not belong. For example, consider a client-server application where the server is replicated for fault-tolerance. In this case, a client broadcasting a request to the distributed server is not a member of the group of processes that implement the service. Thus, the specification of Reliable Multicast does *not* require that the sender of a message m be a member of $group(m)$.[19]

9.2 FIFO Multicast

Informally, FIFO Multicast is a Reliable Multicast in which a process p delivers a message m only if it has already delivered all messages previously multicast by the sender of m, with the exception of those targeted to groups of which p is not a member. More precisely, *FIFO Multicast* is a Reliable Multicast that satisfies the following requirement:

[19] This is why the formulation of Validity for multicasts is different from the one for broadcasts: We can no longer require that the sender of a message deliver its own message.

- *Global FIFO Order:* If a process multicasts a message m before it multicasts a message m', then no correct process in $group(m)$ delivers m' unless it has previously delivered m.

In some applications, a weaker order requirement is adequate. It stipulates that FIFO Order applies only to messages that are multicast to the same group. More precisely,

- *Local FIFO Order:* If a process multicasts a message m before it multicasts a message m' such that $group(m') = group(m)$, then no correct process delivers m' unless it has previously delivered m.

To see the difference between the two FIFO Order properties, consider the following example. Suppose a process p multicasts a message m to group $G = \{p, q, r\}$ and then it multicasts a message m' to group $G' = \{p, q, r'\}$. Suppose, further, that there are no failures. Since q is in both groups, it must deliver both messages. In the case of Global FIFO Order, q must deliver m before m'. In the case of Local FIFO Order, since m and m' were multicast to different groups, q is not constrained as to the order in which it delivers them.

If the application is such that a process p delivering a message m can properly interpret m if it has already delivered every message that $sender(m)$ previously multicast to $group(m)$, then Local FIFO Order can be used. In general, however, it may be that to properly understand m, p must have already delivered every message that $sender(m)$ previously multicast to *any* group of which p is a member. In this case Global FIFO Order should be used.

9.3 Causal Multicast

As we strengthened FIFO Broadcast to obtain Causal Broadcast, we can strengthen FIFO Multicast to obtain Causal Multicast. To do so, we must first define the *causal precedence* induced by the multicasts and deliveries of messages. This is just the $\rightarrow$ relation defined in Section 2.7 except that the word "broadcast" is substituted with "multicast" in Clause 2. We can now define *Causal Multicast* to be a Reliable Multicast that satisfies:

- *Global Causal Order:* If the multicast of a message m causally precedes the multicast of a message m', then no correct process in $group(m)$ delivers m' unless it has previously delivered m.

In the above definition, the causal precedence relation crosses group boundaries: it may relate multicasts and deliveries that "occur" in different groups (hence the name *Global* Causal Order). For example, suppose there are two groups $G = \{p, q\}$ and $G' = \{p, q, r\}$, and consider the following sequence of events: (a) p multicasts m to G; (b) p multicasts m'' to G'; (c) r delivers m'' in G'; (d) r multicasts m' to G'. By definition, the multicast of m causally precedes the multicast of m'. Since q is in both G and G', Global Causal

Order requires that q deliver m before m', even though these two messages were multicast to *different* groups. Similarly, it requires that q deliver m'' before m'.

Local Causal Order, a weaker type of Causal Order that does not cross group boundaries, is defined as follows. Given any group G, the causal precedence relation induced by the multicasts and deliveries of messages m such that $group(m) = G$ is called *causal precedence in group* G. Note that this relation ignores the multicasts and deliveries of messages that are not in group G. We now define Local Causal Order as follows:

- *Local Causal Order:* If the multicast of a message m causally precedes *in* $group(m)$ the multicast of a message m', then no correct process delivers m' unless it has previously delivered m.

In the previous example, $group(m) = G \neq group(m') = G'$. So, the multicast of m does *not* causally precede in $group(m)$ the multicast of m', and, in contrast to Global Causal Order, Local Causal Order allows q to deliver m' and m in any order. However, the multicast of m'' causally precedes in G' the multicast of m', so Local Causal Order requires that q deliver m'' before m'.

The following example shows that even if the multicast of m causally precedes the multicast of m' *and* $group(m) = group(m')$, it is still possible that the multicast of m does *not* causally precede in $group(m)$ the multicast of a m'. Consider groups $G = \{p, q\}$ and $G' = \{p, r\}$, and the following sequence of events: (a) p multicasts m to G; (b) p multicasts m'' to G'; (c) r delivers m'' in G'; (d) r multicasts m' to G (even though r does not belong to G). By definition, the multicast of m causally precedes the multicast of m'. Since both messages are multicast to a group of which q is a member, Global Causal Order requires that q deliver m before m'. However, even though m and m' were multicast to the *same* group G, the multicast of m does *not* causally precede the multicast of m' in G, since the causality is established via the multicast and delivery of m'', a message that is not in G. So, in contrast to Global Causal Order, Local Causal Order allows q to deliver m' before m.

9.4 Atomic Multicast

We consider three types of Atomic Multicast that differ by the strength of their message delivery order requirement.

9.4.1 Local Atomic Multicast

A *Local Atomic Multicast* is a Reliable Multicast that satisfies the following property:

- *Local Total Order:* If correct processes p and q both deliver messages m and m' and $group(m) = group(m')$, then p delivers m before m' if and only if q delivers m before m'.

This is the total order property guaranteed by the Atomic Multicast primitive that the Isis system provides [BJ87,BCJ+90].

9.4.2 Pairwise Atomic Multicast

Local Total Order allows two correct processes to disagree on the order in which they deliver messages. For example, consider two groups $G = \{p, q, r\}$ and $G' = \{p, q, r'\}$. Suppose r multicasts m to G, and r' multicasts m' to G'. Local Total Order allows p and q to deliver the two messages in different order. This disagreement, which is undesirable in some applications, is prevented by *Pairwise Atomic Multicast*, a Reliable Multicast that satisfies the following property:

- *Pairwise Total Order:* If correct processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

This is the total order requirement of several Atomic Multicasts that have appeared in the literature (e.g., [GMS91]).

9.4.3 Global Atomic Multicast

Pairwise Total Order is not the strongest possible message ordering requirement for Atomic Multicasts. In particular, it allows cycles in message delivery order. For example, consider three groups, $G_1 = \{p, q\}$, $G_2 = \{q, r\}$ and $G_3 = \{r, p\}$. Note that the intersection of any two of these groups consists of exactly one process. The messages m_1, m_2 and m_3 are multicast to groups G_1, G_2 and G_3, respectively. Pairwise Total Order allows process p to deliver m_3 before m_1, q to deliver m_1 before m_2, and r to deliver m_2 before m_3. This cycle of deliveries is counter to the intuition that Atomic Multicast must provide the abstraction of indivisible, i.e., "simultaneous", deliveries. Moreover, Pairwise Total Order is not strong enough for some applications, as we illustrate with a simple example below.

Consider the *Dining Philosophers' Problem* with philosophers G_1, G_2 and G_3, and forks p, q and r. G_1 needs forks p and q, G_2 needs forks q and r, and G_3 needs forks r and p. Each philosopher G_i competes for her forks by broadcasting m_i to her forks. G_i wins a fork if her message is the first to be delivered at that fork. If the philosophers use Pairwise Atomic Multicast to broadcast their messages, and a cycle of deliveries occurs as in the scenario described above, each philosopher wins exactly one fork, and they all starve.

Global Atomic Multicast is a type of Atomic Multicast that precludes such cycles. Consider the set of messages delivered by correct processes. We define the relation $<$ on this set as follows: $m < m'$ if and only if any correct process delivers m and m', in that order. A *Global Atomic Multicast* is a Reliable Multicast that satisfies the following property:

- *Global Total Order:* The relation $<$ is acyclic.

Since $<$ is acyclic, the set of messages delivered by correct processes can be totally ordered in a way that is consistent with $<$, i.e., with the order of message deliveries at each correct process.

Global Total Order is strictly stronger than Pairwise Total Order which is strictly stronger than Local Total Order. However, none of these three properties guarantees FIFO Order. In principle, we can define six types of FIFO Atomic Multicasts by selecting one of the two FIFO Orders (Local or Global) and any one of the three Total Orders above. Of these, the FIFO Atomic Multicast that requires the Local versions of FIFO Order and Total Order, and the one requiring the Global versions of these two properties, seem the most sensible. The others are of questionable value.

Similarly, we can define six types of Causal Atomic Multicasts, by combining any one of the two types of Causal Order with one of the three Total Orders above. As before, of these six combinations two seem particularly useful: the one combining the Local versions of the order properties, and the one combining the Global versions.

9.5 Timeliness and Uniformity

As with broadcasts, we can impose a bound on the latency of multicasts in terms of local or real time. More precisely, we can require one of these two properties:

- *Local-Time Δ_G-Timeliness:* For each group G, there is a known constant Δ_G such that no correct process p delivers a message m after local time $ts(m) + \Delta_G$ on p's clock.
- *Real-Time Δ-Timeliness:* For each group G, there is a known constant Δ_G such that if a message m is broadcast at real time t, then no correct process delivers m after real time $t + \Delta_G$.

Note that the bound Δ_G on the latency now depends on the group G. A multicast that satisfies any of the above properties is called a *Timed Multicast.*

As with broadcast, we can also define the *Uniform* counterparts for the Agreement, Integrity, Order, and Δ-Timeliness properties of multicasts. The formal definitions are straightforward and are omitted.

A Appendix — Model of Point-to-Point Networks

In this appendix we describe our model of point-to-point networks in more detail than in Sections 2 and 6. The overall plan is as follows: First we describe the correct behavior of processes and links. We then define different types of (process and link) failures as violations of some of the properties that describe correct behavior. In the case of synchronous systems, where correct behavior includes the *timely* occurrence of actions, correctness can be violated in two ways: actions that are supposed to take place never occur (omission failures), or they occur late (performance and clock failures).

A.1 Networks with No Failures

A point-to-point network can be modeled as a directed graph, with nodes representing processes, and edges representing communication links between processes. In such a network, any pair of processes that are connected by a link can communicate with each other by sending and receiving messages, as described below. In this section we assume that processes and links do not fail.

Properties of Processes:

Each process is capable of executing certain operations, such as the writing of a local variable, or the sending or receipt of a message. The execution of an operation by a process p is a *step* of p. We do not assume that the steps are atomic; a step consists of a sequence of atomic *events*, delimited by a `start` and an `end` event. (The fact that steps are not atomic will permit us, in the next section, to model failures that interrupt the execution of an operation in the middle.) Hence, the execution of a process p is modeled as a sequence of events grouped into steps such that the `start` event of each step (except the first one) immediately follows the `end` event of the previous step. If this sequence includes the `start` event of a step, we say that p has *started* that step; if it includes the `end` event of a step, we say that p *completed* that step. Associated with each process p is an automaton whose transition relation describes the legal sequences of events (and thus of steps) for p. We assume that:

a. Every process completes an infinite number of steps.

This implies that every process eventually completes every step that it starts.

Properties of `send` and `receive`:

Let p and q be any two processes connected by a link from p to q. Associated with this link are the communication primitives `send` and `receive`, which are among the operations that can be executed by p and q, respectively. The operation `send` takes a message as a parameter; `receive` returns a message. The execution of the `send` primitive with parameter m is a step denoted `send`(m); the execution of the `receive` primitive with return value m is a step denoted `receive`(m). We say that p *sends* m *to* q if p starts the step `send`(m); we say that q *receives* m if q completes the step `receive`(m).

Associated with the link from p to q, p has an *outgoing message buffer*, denoted $omb(p, q)$, and q has an *incoming message buffer*, denoted $imb(p, q)$. Informally, when p sends a message m to q, p inserts m in $omb(p, q)$, the link transports m from $omb(p, q)$ to $imb(p, q)$, and q receives m from $imb(p, q)$. More precisely, the `send` and `receive` primitives associated with the link from p to q satisfy:[20]

b. If p completes the sending of m to q, then m is eventually inserted into $omb(p, q)$.

c. If m is inserted into $omb(p, q)$, then m is eventually inserted into $imb(p, q)$.

[20] To simplify the formulation of these properties, we assume that each message sent from p to q is unique. This can be easily enforced by using link sequence numbers.

d. If m is inserted into $imb(p,q)$, then q eventually receives m.

These three properties imply:

- If p sends m to q then q eventually receives m.[21]

We also assume that:

b′. m is inserted into $omb(p,q)$ at most once, and only if p sends m to q.

c′. m is inserted into $imb(p,q)$ at most once, and only if m is in $omb(p,q)$.

d′. q receives m at most once, and only if m is in $imb(p,q)$.

Properties (b′)–(d′) imply:

- *Uniform Integrity:* For any message m, q receives m at most once from p, and only if p previously sent m to q.

To simplify the exposition and the correctness proofs of our algorithms for point-to-point networks, we found it convenient to allow each process to send a message *to itself.* This is only a fictitious device, and a message "sent" this way does not really go through any buffer or link. We postulate the following property regarding such messages:

- If p sends m to itself, then p receives m instantaneously.

The preceding definition of a point-to-point network assumes that no failures occur. In the next section we consider some of the failures that can affect processes and links. These failures will be defined as violations of Properties (a)–(d). We will *not* allow the violation of Properties (b′)–(d′); thus, Uniform Integrity holds even in networks with failures. We will also not allow the violation of the postulated property regarding messages sent by a process to itself.

A.2 Networks with Omission Failures

Failures can be defined as deviations from correct behavior. In networks with omission failures, processes and links may violate Properties (a)–(d).

Violating Property (a) of Processes:

To model the violation of Property (a), we introduce a special event called `crash`. Every process p can execute a `crash` at any time[22], and after doing so it stops executing

[21] Note that messages are not necessarily received in the order in which they are sent.

[22] In particular, a `crash` event may occur in the middle of a step, i.e., between the `start` and the `end` events of a step. This models a crash that interrupts the execution of a non-atomic operation in the middle.

further events. This is modeled by the addition of a new terminal state to the automaton associated with p, and a transition from every other state of p to that terminal state. The event associated with such a transition is defined as a `crash`. We say that *p commits a crash failure* if it executes a `crash` event.

Since no event can follow `crash`, a process that crashes can execute only a finite number of events, and therefore completes only a finite number of steps. Thus, a process that crashes violates Property (a). We assume, however, that *only* processes that crash violate that property. That is, a process that does not crash completes an infinite number of steps.

Violating Properties (b), (c), and (d) of `send` and `receive`:

- ▷ Process *p commits a send-omission failure on m* if p completes the sending of m to q but m is never inserted into $omb(p, q)$ (violation of Property (b)).
- ▷ The link from p to q *commits an omission failure on m* if m is inserted into $omb(p, q)$ but m is never inserted into $imb(p, q)$ (violation of Property (c)).
- ▷ Process *q commits a receive-omission failure on m* if m is inserted into $imb(p, q)$ but q never receives m and does not crash (violation of Property (d)).

If a process or a link commits a failure, we say that it is *faulty*; otherwise it is *correct*. Recall that in networks with no failures, if p sends m to q then q eventually receives m. The properties of point-to-point networks with omission failures imply:

- *Validity:* If p sends m to q and q does not receive m, then one of the following holds:
  1. p does not complete the sending of m, or
  2. p commits a send-omission failure on m, or
  3. the link from p to q commits an omission failure on m, or
  4. q commits a receive-omission failure on m, or
  5. q crashes.

This formulation of Validity implies the simpler one stated in Section 6.1. It also implies the Strong Validity property given in Section 6.5 for systems where send-omission failures do not occur.

A.3 Synchronous Networks with No Failures

Consider the sequence of events executed by a process. Since events are atomic, we can associate with each the real time at which it occurred. We say that *p sends m at time t* (respectively, *q receives m at time t*), if the `start` event of $send(m)$ (respectively, the `end` event of $receive(m)$) occurs at time t.

A point-to-point network with no failures is *synchronous* if, in addition to the properties of Section A.1, all processes and links satisfy Properties (e)–(i) below (in all our definitions for synchronous networks, the word "time" always refers to real time):

Synchrony Properties of Processes:

e. *τ-Local Step:* There is a known constant $\tau \geq 0$ such that no process completes a step later than τ time units of when it started that step.

f. *ρ-Clock Drift:* There is a known constant $\rho \geq 0$ such that every process p has a local clock whose drift with respect to time is bounded by ρ. That is, if $C_p(t)$ denotes the value of the local clock of p at time t, then for all $t > t'$:

$$\frac{1}{1+\rho} \leq \frac{C_p(t) - C_p(t')}{t - t'} \leq 1 + \rho$$

Synchrony Properties of `send` and `receive`:

There are known constants τ_1, τ_2, and $\tau_3 \geq 0$, such that for any processes p and q connected by a link from p to q:

g. If p sends m to q at time t, then m is not inserted into $omb(p, q)$ after time $t + \tau_1$.

h. If m is inserted into $omb(p, q)$ at time t, then m is not inserted into $imb(p, q)$ after time $t + \tau_2$.

i. If m is inserted into $imb(p, q)$ at time t, then q does not receive m after time $t + \tau_3$.

Let $\delta = \tau_1 + \tau_2 + \tau_3$. The properties of synchronous point-to-point networks imply:

- *δ-Timeliness:* There is a known constant $\delta \geq 0$ such that for any processes p and q connected by a link, if p sends m to q at time t then q does not receive m after time $t + \delta$.

A.4 Synchronous Networks with Omission Failures

In synchronous networks with omission failures, all processes and links satisfy all the synchrony properties (i.e., (e)–(i)), but some may violate Properties (a)–(d). The synchrony properties allow us to redefine crash and omission failures in a way that includes information about *the time* when each failure occurred.

Violating Property (a) of Processes:

As in Section A.2, we introduce a `crash` event which can be executed by any process p at any time, but after which p may not execute any event. We say that p *crashes at time* t if it executes a `crash` event at time t.

Violating Properties (b), (c), and (d) of `send` and `receive`:

- ▷ Process p *commits a send-omission failure on* m *during* $[t, t+\tau_1]$ if p sends m to q at time t, completes the sending of m, but m is never inserted into $omb(p,q)$ (violation of Property (b) for a message m sent at time t).
- ▷ The link from p to q *commits an omission failure on* m *during* $[t, t+\tau_2]$ if m is inserted into $omb(p,q)$ at time t but m is never inserted into $imb(p,q)$ (violation of Property (c) for a message m inserted into $omb(p,q)$ at time t).
- ▷ Process q *commits a receive-omission failure on* m *during* $[t, t+\tau_3]$ if m is inserted into $imb(p,q)$ at time t but q never receives m and does not crash by time $t+\tau_3$ (violation of Property (d) for a message m inserted into $imb(p,q)$ at time t).

A.5 Synchronous Networks with Clock and Performance Failures

In synchronous networks with clock and performance failures, processes and links may violate the synchrony properties (e), (f), (g), (h), and (i).

Violating Synchrony Properties (e) and (f) of Processes:

- ▷ Process p *commits a performance failure* if p completes a step later than τ time units of when it started that step (violation of Property (e)).
- ▷ Process p *commits a clock failure* if during some time interval the local clock C_p of p drifts more than ρ with respect to real time (violation of Property (f)).

Violating Synchrony Properties (g), (h), and (i) of `send` and `receive`:

- ▷ Process p *commits a performance failure on the sending of* m if p sends m to q at time t and m is inserted into $omb(p,q)$ after time $t+\tau_1$ (violation of Property (g)).
- ▷ The link from p to q *commits a performance failure on* m if m is inserted into $omb(p,q)$ at time t and m is inserted into $imb(p,q)$ after time $t+\tau_2$ (violation of Property (h)).
- ▷ Process q *commits a performance failure on the receipt of* m if m is inserted into $imb(p,q)$ at time t and q receives m after time $t+\tau_3$ (violation of Property (i)).

A.6 Classification of Failures and Terminology

In the preceding sections, we have defined crash, send-omission, receive-omission failures of processes, and omission failures of links. For synchronous systems, we also defined performance and clock failures of processes, and performance failures of links.

It is convenient to group failures into two classes that include both process *and* link failures:

- *omission failures* consist of crash, send-omission, and receive-omission failures of processes, as well as link omission failures.

- *timing failures* consist of omission, clock and performance failures.

A network *with* a certain class of failures, is one where processes and links may commit any of the failures included in that class, but no other failures. Thus, a network with omission failures is not subject to clock, performance, or arbitrary failures. Similarly, one with timing failures is not subject to arbitrary failures. *Benign failures* is synonymous to omission failures in asynchronous networks and to timing failures in synchronous networks.

We have not included arbitrary failures among the types of process and link failures defined in this appendix, since these failures are only considered in passing in this paper. It is not difficult to incorporate such failures in the framework of our model. Recall that we defined a network with no process or link failures by stating various properties that the processes and links must satisfy. The types of failures considered so far were defined as specific violations of some of these properties. We say that a process or link commits *arbitrary failures* if it can violate arbitrarily the properties that specify its correct behavior.

Acknowledgements

We are indebted to a large number of colleagues for extremely useful discussions and comments on a draft of this paper: Navin Budhiraja, Tushar Chandra, David Cooper, Prasad Jayanti, Mike Reiter, and King Tan. Ajei Gopal helped us formulate the properties of broadcast problems. Over the past few years we have received invaluable feedback on this material from students of CS 444 and CS 618 at Cornell, and CSC 2221 at the University of Toronto.

References

[ABD+87] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rüdiger Reischuk. Achievable cases in an asynchronous environment. In *Proceedings of the Twenty-Eighth Symposium on Foundations of Computer Science*, pages 337–346. IEEE Computer Society Press, October 1987.

[Abr85] N. Abramson. Development of the ALOHANET. *IEEE Transactions on Information Theory*, 31:119–123, March 1985.

[ADKM92] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication sub-system for high availability. In *Proceedings of the Twenty Second International Symposium on Fault-Tolerant Computing*, pages 76–84. IEEE Computer Society Press, July 1992.

[BCJ+90] Kenneth P. Birman, Robert Cooper, Thomas A. Joseph, Kenneth P. Kane, and Frank Bernhard Schmuck. *Isis - A Distributed Programming Environment*, June 1990.

[BD85] Özalp Babaoğlu and Rogério Drummond. Streets of Byzantium: Network architectures for fast reliable broadcasts. *IEEE Transactions on Software Engineering*, 11(6):546–554, June 1985.

[BD87] Özalp Babaoğlu and Rogério Drummond. (Almost) no cost clock synchronization. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, pages 42–47. IEEE Computer Society Press, July 1987.

[Ben83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second ACM Symposium on Principles of Distributed Computing*, pages 27–30, August 1983.

[BG93] Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty fifth ACM Symposium on Theory of Computing*, pages 91–100. ACM Press, May 1993.

[BHG87] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[BJ87] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BMZ88] Ofer Biran, Shlomo Moran, and Shmuel Zaks. A combinatorial characterization of the distributed tasks that are solvable in the presence of one faulty processor. In *Proceedings of the Seventh ACM Symposium on Principles of Distributed Computing*, pages 263–275, August 1988.

[BN91] Rida Bazzi and Gil Neiger. Optimally simulating crash failures in a Byzantine environment. In S. Toueg, P.G. Spirakis, and L. Kirousis, editors, *Proceedings of the Fifth International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science, pages 108–128. Springer-Verlag, October 1991.

[BN92] Rida Bazzi and Gil Neiger. Simulating crash failures with many faulty processors. In A. Segal and S. Zaks, editors, *Proceedings of the Sixth International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science, pages 166–184. Springer-Verlag, November 1992.

[Bra87] Gabriel Bracha. Asynchronous Byzantine agreement protocols. *Information and Computation*, 75(2):130–143, November 1987.

[BSS91] Kenneth P. Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[BT93] Özalp Babaoğlu and Sam Toueg. Non-blocking atomic commitment. In Sape J. Mullender, editor, *Distributed Systems*, chapter 6. Addison-Wesley, 1993.

[BW87] Michael F. Bridgland and Ronald J. Watro. Fault-tolerant decision making in totally asynchronous distributed systems. In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 52–63, August 1987.

[CAS86] Flaviu Cristian, Houtan Aghili, and H. Raymond Strong. Approximate clock synchronization despite omission and performance faults and processor joins. In *Proceedings of the Sixteenth International Symposium on Fault-Tolerant Computing*, July 1986.

[CASD85] Flaviu Cristian, Houtan Aghili, H. Raymond Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, June 1985. A revised version appears as IBM Research Laboratory Technical Report RJ5244 (April 1989).

[CD89] Benny Chor and Cynthia Dwork. Randomization in byzantine agreement. *Advances in Computer Research*, 5:443–497, 1989.

[CDD90] Flaviu Cristian, Robert D. Dancey, and Jon Dehn. Fault-tolerance in the advanced automation system. Technical Report RJ 7424, IBM Research Laboratory, April 1990.

[CHT92] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 147–158, August 1992.

[CM84] Jo-Mei Chang and N.F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[Coa87] Brian A. Coan. *Achieving Consensus in Fault-Tolerant Distributed Computer Systems: Protocols, Lower Bounds, and Simulations.* PhD thesis, Massachusetts Institute of Technology, June 1987.

[Cri87] Flaviu Cristian. Issues in the design of highly available computing services. In *Annual Symposium of the Canadian Information Processing Society*, pages 9–16, July 1987. Also IBM Research Report RJ5856, July 1987.

[Cri89] Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[Cri90a] Flaviu Cristian. Reaching agreement on processor group membership in synchronous distributed systems. Technical Report RJ5964, IBM Research Laboratory, October 1990.

[Cri90b] Flaviu Cristian. Synchronous atomic broadcast for redundant broadcast channels. *Real-Time Systems*, 2(3):195–212, September 1990. Also IBM Research Report RJ7203, December 1989 (revised April 1990).

[CT91] Tushar Chandra and Sam Toueg. Unreliable failure detectors for asynchronous systems (preliminary version). In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM Press, August 1991.

[CT92] Tushar Deepak Chandra and Sam Toueg. Personal communication. 1992.

[DDS87] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DHS86] Danny Dolev, Joseph Y. Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 22(2):230–250, April 1986.

[DLP+86] Danny Dolev, Nancy A. Lynch, Shlomit S. Pinter, Eugene W. Stark, and William E. Weihl. Reaching approximate agreement in the presence of faults. *Journal of the ACM*, 33(3):499–516, July 1986.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[Fek90] Alan D. Fekete. Asymptotically optimal algorithms for approximate agreement. *Distributed Computing*, 4(1):9–30, 1990.

[Fek93] Alan D. Fekete. Asynchronous approximate agreement. *Information and Computation*, 1993. To appear.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FM90] Paul Feldman and Silvio Micali. An optimal algorithm for synchronous Byzantine agreement. Technical Report MIT/LCS/TM-425, Laboratory for Computer Science, Massachusetts Institueof Technology, June 1990.

[GMS91] Hector Garcia-Molina and Annemarie Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, August 1991.

[Gop92] Ajei Gopal. *Fault-Tolerant Broadcasts and Multicasts: The Problem of Inconsistency and Contamination.* PhD thesis, Cornell University, January 1992.

[Gra78] James N. Gray. Notes on database operating systems. In R. Bayer, R. M. Graham, and G. Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 66 of *Lecture Notes on Computer Science*. Springer-Verlag, 1978. Also appears as IBM Research Laboratory Technical report RJ2188.

[GSTC90] Ajei Gopal, Ray Strong, Sam Toueg, and Flaviu Cristian. Early-delivery atomic broadcast. In *Proceedings of the Ninth ACM Symposium on Principles of Distributed Computing*, pages 297–310, August 1990.

[GT91] Ajei Gopal and Sam Toueg. Inconsistency and contamination. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 257–272, August 1991.

[Had84] Vassos Hadzilacos. *Issues of Fault Tolerance in Concurrent Computations*. PhD thesis, Harvard University, June 1984. Department of Computer Science Technical Report 11-84.

[HJT] Vassos Hadzilacos, Prasad Jayanti, and Sam Toueg. *Fundamentals of Fault-Tolerant Distributed Computing*. Forthcoming.

[HS93] Maurice Herlihy and Nir Shavit. The asynchronous computability theorem for t-resilient tasks. In *Proceedings of the Twenty fifth ACM Symposium on Theory of Computing*, pages 111–120. ACM Press, May 1993.

[HSSD84] Joseph Y. Halpern, Barbara Simons, Ray Strong, and Danny Dolev. Fault-tolerant clock synchronization. In *Proceedings of the Third ACM Symposium on Principles of Distributed Computing*, pages 89–102, August 1984.

[Kaa92] M. Frans Kaashoek. *Group Communication in Distributed Computer Systems*. PhD thesis, Vrije Universiteit, December 1992.

[KO87] Hermann Kopetz and Wilhelm Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, C-36(8):933–940, August 1987.

[Lam78a] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[Lam78b] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[Lam84] Leslie Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, April 1984.

[LF82] Leslie Lamport and Michael Fischer. Byzantine generals and transaction commit protocols. Technical Report 62, SRI International, April 1982.

[LM85] Leslie Lamport and P. M. Melliar-Smith. Synchronizing clocks in the presence of faults. *Journal of the ACM*, 32(1):52–78, January 1985.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[NT87] Gil Neiger and Sam Toueg. Substituting for real time and common knowledge in asynchronous distributed systems (preliminary version). In *Proceedings of the Sixth ACM Symposium on Principles of Distributed Computing*, pages 281–293, August 1987. A revised and expanded version appears as: Simulating

Synchronized Clocks and Common Knowledge in Distributed Systems. *Journal of the ACM*, 40(2):334–367, April 1993.

[NT90] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, September 1990.

[Owi93] Susan S. Owicki. A perspective on AN2: Local area network as distributed system. In *Proceedings of the Twelfth ACM Symposium on Principles of Distributed Computing*, pages 1–12, August 1993.

[PBS89] Larry L. Peterson, Nick C. Bucholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[PSL80] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, April 1980.

[PT86] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, March 1986.

[Rab83] Michael Rabin. Randomized Byzantine generals. In *Proceedings of the Twenty-Fourth Symposium on Foundations of Computer Science*, pages 403–409. IEEE Computer Society Press, November 1983.

[RB91] Aleta Ricciardi and Ken Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 341–351. ACM Press, August 1991.

[Sch90] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[SS83] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems*, 1(3):222–238, August 1983.

[ST87a] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[ST87b] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[SZ93] Michael Saks and Fotios Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. In *Proceedings of the Twenty fifth ACM Symposium on Theory of Computing*, pages 101–110. ACM Press, May 1993.

[VM90] Paulo Veríssimo and José A. Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, Huntsville, Alabama-USA, October 1990. IEEE.

[Wel87] Jennifer Lundelius Welch. Simulating synchronous processors. *Information and Computation*, 74(2):159–171, August 1987.

[WL88] Jennifer Lundelius Welch and Nancy A. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, April 1988.

[WLG+78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P.M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, October 1978.