Broadcast Specifications

William Hipschman
Department of Computer Science
The University of North Carolina at Chapel Hill

Abstract

Many distributed programs assume an underlying method of group communication. Each of Consensus, Dining Philosophers, and the Byzantine Generals Problem need some type of broadcast. Well-defined broadcasts provide properties such as reliability, order, atomicity, or uniformity. These properties provide a foundation upon which distributed processes can cooperate. Proper use of broadcast contributes to tolerance of failures of differing severity. We use a modular approach to survey reliable, ordered, and timed broadcast variations.

1. Introduction

The field of distributed computing is broad; it includes hardware design, proofs of safety and progress, synchronization, shared memory, message passing and much more. Without a method for process communication it would be difficult for distributed processes to work together, severely impeding the benefits a distributed system is designed to give. Though shared memory and message passing are equivalent models of distributed communication, message passing provides a useful abstraction for the programmer, often at the cost of the efficiency that shared memory models provide. Additionally, the message passing paradigm is applicable in more settings than shared memory.

A specific subset of message passing, group communication is important in distributed systems. Many distributed programs assume there is some underlying method of group communication. Each of Consensus, Dining Philosophers, the Byzantine Generals Problem, and replication, require some method for processes to communicate. In practice, this means that distributed file systems, databases, and transactions often depend on some type of group communication.

Well-defined group communication primitives, such as broadcast, may provide properties such as reliability, order, atomicity, or uniformity. These properties provide a foundation upon which distributed processes can cooperate. Proper use of broadcast helps make a system fault-tolerant.

This resilience to failure, or fault-tolerance, is a vital component for a system involving the cooperation of multiple processes. The designer must guarantee that their system continues to work correctly, or fails gracefully, when one or more processes fail.

Proper failure management is important on many levels. When a process crashes, the remaining correct processes must have a consistent view of the system. When a process provides inaccurate output, the system as a whole must be able to remain correct. When a process hangs indefinitely, the system must be able to continue its work. These failures may be merely inconvenient to a user or may represent significant monetary loss to the company that owns the system.

Despite the widespread use of group communication protocols, operating systems often do not support group communication primitives; they merely provide access to kernel-level send and receive operations. It is then necessary for an application programmer to build their own communication protocols.

In this paper we survey common group communication protocols and analyze them for their tolerance of process and communication failure. Before presenting these protocols, we cover background information that is required for a proper understanding of the protocols. This includes definitions of failure, an analysis of various synchrony conditions required to implement broadcasts, and an overview of basic communication types. We then proceed to define different measures of efficiency for distributed communication, so that we have a metric for differentiating between different fault-tolerant

communication algorithms. The background material concludes in Section 4 with a brief survey of important results that will dictate how many failures each of protocols can tolerate.

This paper by presents the formal model used and then defines several properties that may be useful for a group communication protocol. We then survey common group communication protocols by presenting modular constructions where each protocol uses the previous as a building block. We define each protocol and analyze them for their fault tolerance and efficiency.

2. Background and Definitions

In order to understand fault-tolerant broadcasts, we must first understand what it means to be fault-tolerant and what it means to broadcast. Here we will define fault tolerance and discuss different levels of failure severity. We then discuss differing levels of system synchrony, recognizing that synchrony dictates whether a program can detect and handle failures. We conclude the background information by comparing different types of communication primitives.

2.1 Fault-Tolerance

Fault-tolerance is a common buzzword in distributed computing; however, the phrase is highly system-dependent. It is obvious that a system where all processes crash must stop working but that does not help us define fault tolerance. We need a more fine-grained definition. We will model all communication links as their own processes housing a queue and defining non-deterministic failure operations. This allows us to discuss only process failures but apply the results to both process and communication failures.

The number of failures a system can tolerate, the *resilience* of the system, is sensitive to the application. The resilience is also sensitive to the type of failure. A system might continue functioning if a process stops, but consider a process outputting incorrect data an un-recoverable event. It is helpful to define different failure types so a system designer can decide what fault-tolerance means for their application [3, 4, 5, 8, 9]. A *contract* for an application is an agreement between the program and the user dictating what types of failures, how many failures, and when failures are allowed.

The type of failure a system allows is not the only criteria for classifying its fault tolerance. We must also consider whether the system can recover from failures, the expected frequency of failures, and issues of consistency and corruption.

If some central process fails and no replication or recovery protocol is built-in then a distributed application may have no alternative but to fail. On the other hand, an application with some replication of recovery method could continue running. Though we will not discuss them at length, replication and recovery go hand-in-hand with failure tolerance and should not be overlooked in designing a fault-tolerant application [4, 5, 24].

2.1 Failure Types

The most harmless failure is a *crash failure*. A crash failure exhibits two criteria; it forces whatever operation the process is currently performing, often a send or receive, to undergo an *omission failure*, and it causes the process to perform no further actions. Crash failures are sometimes referred to as fail-stops or as process deaths [3, 4, 5, 8, 9].

As an example, we will consider a group of processes that share a common variable, x. Each process can send arithmetic operations to any other process. A process crashes at time t if it fails to send or receive any arithmetic operations after time t. When a process recovers from a crash (or other type of failure) it requests the value of x from other processes.

A process that undergoes the first criteria of a crash failure, omission, but continues to operate experiences an omission failure. A process in send omission failure can receive messages and perform all actions except for sending a message. A process that is failing due to receive omission is the converse. A general omission failure exhibits send omission failures, receive omission failures, or both [3, 4, 5, 9].

Timing failures are failures where a message breaks some time sensitive contract between processes. Often this is because a message arrived late; a message arriving early is infrequently a cause for a failure [3, 9].

Each of the previous failures is a benign failure. These failures break some contract imposed by the system, but their failure is not adversarial, the failing processes are not intentionally sabotaging the system [3, 9].

A process that is intentionally sabotaging the system, is exhibiting *arbitrary failure*. In our example, if one process is recovering and needs to learn the correct value of x from another processes, an arbitrary failing process may send an incorrect value of x. A process that is experiencing arbitrary failure may fail in any of the benign ways, and may additionally send messages that are counterfeit or break contracts between processes The least-severe arbitrary failures are *authenticated arbitrary failures*. These failures are authenticated in that each process that receives a message from a process that is experiencing arbitrary authenticated failure knows the identity of the sender; the message cannot be counterfeit. Both types of arbitrary failures are also called *Byzantine failures*, named for the Byzantine Generals Problem [3, 6, 9, 10, 25, 27, 28].

2.2 Synchrony

The Byzantine Generals problem is in part famous for showing how the level of synchrony a system has impacts its ability to tolerate failures.

The synchrony of a program is defined on a spectrum that runs from completely synchronous to completely asynchronous. Most programs, though, are somewhere in the middle. A high level of asynchrony presents difficult programming challenges but allows for more concurrency [13].

Informally, processes collaborating synchronously wait for one another. Each may wait for replies from all other processes before continuing with its computation [3, 9, 11, 13, 17, 18]. More formally, a *synchronous* system is one in which there are known bounds on message delay, clock differences and drift, and step execution time [3, 13, 18]. An asynchronous system lacks these characteristics; there is no bound on message delay, clock drift, or time required to execute a step [3, 18]. Asynchronicity is often defined imprecisely. Systems that wait for few messages or no messages, have no known upper bound on clock differences *(incomparable clocks)*, have no notion of timeout, or lack explicit synchronization protocols are often referred to as asynchronous [3, 9].

It is more accurate, and often more helpful, to discuss differing levels of synchrony than to assume complete synchrony/asynchrony. While many areas of research focus primarily on asynchronous systems for the sake of efficiency, not all systems are as asynchronous as the literature expects. It is useful to determine where asynchrony exists and when it is a problem. The level of synchrony can change based on the order of message delivery, which is partially dependent on the type of underlying communication protocol. Is it some type of broadcast or a point-to-point transmission? Are the send and receive operations of a process treated together as an indivisible operation or are they separate calls? These questions are important to consider when designing a distributed system, and provide benefits and drawbacks for group communication that we will discuss later.

2.3 Communication Primitives

Communication between processes can take many forms. In this paper we will, for the most part, ignore the underlying communication links and assume that all communication between processes takes the form of message transmissions and receptions. We call a method for group communication a *primitive* if it is provided as a utility to higher functions. In this paper we refer to all forms of broadcast as primitives, despite the fact that some are more basic than others.

The simplest of these forms, *point-to-point* message transmission, is when one process sends a single message to another process [9, 11]. This is the building block for more complex forms of communication. Point-to-point message transmission is sometimes called unicast [9].

We can organize a group communication taxonomy by asking where a message originates and where it is delivered. The most common form of group communication, a process sending a message (or replicas of a message) to all other processes, is called *broadcast* or one-to-N communication [3, 9, 11]. This type of communication can be built from point-to-point (one-to-one) communication and is a foundation for other types of broadcasts. If all processes broadcast, so that each process sends a message to all other processes, it is called N-to-N communication. This type of communication is especially common in agreement, or consensus, protocols where a group of processes need to come to a mutually agreed-upon conclusion [25, 27, 28].

Often processes are divided into groups and it is necessary for a process to broadcast to processes in a certain group. This is called *multicast*. Broadcast is a special case of multicast where there is only one group. Some driving research areas in multicast communication revolve around group management, specifically focusing on naming and dynamic group membership [3, 5, 9, 19, 23, 30]. We will not investigate these topics except as they relate to protocol correctness and fault tolerance.

An infrequent type of communication, N-to-one, is seldom discussed in the literature, primarily due to few applicable uses.

In addition to deciding the source and destination of messages, a designer should specify a contract to which communication protocols must adhere. What happens if half of the processes receive a broadcast and half do not? What happens if a process receives a message that breaks the contract for expected message form? What happens if a process receives messages out of order? We will answer these questions in Sections 5 and 6.

Because many group communication implementations are built on top of point-to-point message systems provided by the operating system, there is a disconnect between when a broadcast is sent by the application and when the corresponding messages are actually sent by the operating system. There is an equivalent disconnect on the receiving end.

In this paper we will treat message sends and receives from the perspective of the group communication protocol; we assume that a failure of the underlying infrastructure causes the initiating process to be considered as failing. We say that a process broadcasts or multicasts a message when it initiates the sending of that message. We say that a process *delivers* a message when a message is successfully received by the application [3, 9].

It is important to note that the definitions of these communication primitives and the definitions of many varieties and variations on broadcast are independent of any implementation. We are concerned with communication properties and their relation to fault-tolerant distributed systems, not any implementation or optimization of an algorithm. We will show simple implementations that illustrate relevant properties, as opposed to industry-grade implementations.

3. Group Communication Complexity

Before we address these implementations, we must decide on metrics to differentiate good solutions from bad ones. In non-distributed systems this can be done by analyzing time and space complexity. We need efficiency measures that make sense for distributed systems and are relevant for group communication. We present them here and later use them to analyze various broadcast protocols.

Measuring complexity in a sequential system is in many ways simple: we just have to count the number of lines of executed code. Measuring the complexity of distributed systems poses many additional challenges.

The first challenge to confront is how to handle interaction with other processes. We must take care that a process avoids traditional pitfalls such as deadlock, livelock, and starvation. We must additionally be careful to define exactly what time complexity means across systems that may have different or even incomparable clocks. Further, it is important to recognize that part of proving complexity in a sequential system involves proving termination. Many distributed systems never terminate. It is then necessary to address complexity for systems that run indefinitely.

The efficiency of a group communication protocol can be divided into three separate measurements: fault-tolerance, message complexity, and time complexity.

3.1 Fault-Tolerance

The *fault-tolerance* of a program is a measurement of how many processes can fail [3, 9]. It is measured in relation to the number of processes so, for example, a system with N processes that can continue operation as long as half of the processes are functioning would have fault-tolerance of $\left\lceil \frac{N}{2} \right\rceil$. Fault-tolerance is equivalent to the resilience of the system.

3.2 Message Complexity

The *message complexity* of a program is the predominant complexity measure in the literature; it measures the number of messages required to fulfill some purpose [3, 4, 5, 9]. Because these programs run indefinitely, the message complexity typically refers to the number of messages required for some task such as a broadcast or recovery protocol, not the total number of messages across the life of the program.

3.3 Time Complexity

The time complexity of a program is perhaps the most difficult measure to define. At a strictly practical level the time complexity of a program is a measurement of the number of operations per second. This can be helpful for benchmarking algorithms but does not give us an idea of the complexity of a distributed algorithm that has not yet been implemented. Some have proposed that a program that runs indefinitely can be viewed as a sequence of rounds containing actions. Time complexity could then be measured by counting the number of actions per round.

More formally, *synchronous time complexity* in a distributed system is a measurement of the number of remote memory calls required to perform some task [3, 9]. While this is often a good heuristic for the amount of time an algorithm will take, we must remember that completely asynchronous systems have incomparable clocks; two algorithms with identical synchronous time complexities may take wildly different amounts of time on completely asynchronous processors. Time complexity should mirror reality in that the time complexity of a program should give the implementer some inkling of run time, especially asymptotic run time in comparison to the complexity of some other algorithm. This is not realized in completely asynchronous systems. *Asynchronous time complexity* therefore remains undefined [3, 9].

3.4 Complexity Measures

Both message and time complexity are asymptotic. If a process requires two messages per program execution and executes no other remote memory calls, then both message and time complexity are $O(1)$ for that process. These measures are often taken across the entire system, so that N processes executing the aforementioned algorithm would have aggregated message and time complexity of $O(N)$.

A group communication algorithm must attempt to minimize message and time complexity while maintaining fault-tolerance and safety requirements. Informally, supporting increased fault-tolerance comes at the cost of message or time complexity. For this reason it is important that the designer of a distributed system support the fault-tolerance necessary for their application, but not more or else risk poor performance.

It is important to note that the complexity measure used to evaluate a group communication scheme should depend heavily on the application in which the scheme is applied. The designer of any system must investigate the impact of network congestion, timing failures, and memory references to gauge which components have the most impact on their application.

4. System Requirements

In addition to designing efficient broadcast algorithms, it is important to design algorithms

that are applicable in a broad range of system settings. For distributed systems, this means that much of the research attempts to solve problems in completely asynchronous settings so that the results also apply to settings with varying levels of synchrony. We find that many types of broadcast cannot be implemented in complete asynchrony if we allow failures. For some broadcast variants, even with timeouts, there is a bound on the number of processes that can fail. Here we cover the relevant impossibility results, which we will apply to broadcast types in Section 6.

Lamport, Shostak, and Peace showed in [27] that, under the assumption that a process can tell when a message is omitted, a group of $3m + 1$ processes can tolerate at most m Byzantine process failures while agreeing upon a value. Under the same assumption, agreement can be solved with authenticated Byzantine failures for any number of failures. Both these results use a synchrony assumption that indicates that the processes involved must have some common notion of timeout, that is, their clocks are synchronized. This problem is referred to as the *Byzantine Generals Problem.*

When this synchrony assumption is abandoned, Fischer, Lynch, and Patterson have shown that it is impossible to solve the consensus problem even with only one crash failure in a deterministic setting. They also found that if all the failures occur before the beginning of the consensus algorithm (some processes start crashed) and no processes fail during the algorithm, then it is possible to solve the completely asynchronous consensus problem, as long as a majority of the processes still live [18].

Dolev, Dwork, and Stockmeyer investigated these findings and determined that the fundamental problem that asynchrony imposes is that processes cannot determine if other processes have failed or if they are merely taking a long time to respond. Contingent upon this, they found that N-resilience could be attained for crash failures with synchronized communication or synchronized message order when all other system components are asynchronous [13].

Many people have tried to circumvent this fundamental impossibility results by giving some arbitrary method for distinguishing between failed processes and processes taking a long time to return. We cover these methods in Section 7.

5. Group Communication Requirements

Now that we have addressed system requirements, failure types, and complexity, we must decide what we expect a broadcast protocol to do. Should it guarantee message delivery? Should it provide an order for message delivery? Should the requirements be imposed only on correct process? In this section we will define formal representations, correctness properties, order properties, and timing properties. We will then present communication protocols that fulfill the requirements imposed by these properties.

5.1 Finite State Machine Representation

This paper will not prove that group communication protocols fulfill properties formally. We will use formal representation to define these properties in an unambiguous way. We will present the formal representation briefly here. This model was originally developed by Lamport [29] but the notation presented by Gopal seems more intuitive and so will be used here [20].

A system can be described by a set of distinct processes, P, each of which has a virtual clock, C, and an application state, Q. The application state contains a member, the *premature halting state*:

$$\perp \in Q$$

We refer to the clock and state of a process, p, by $\boldsymbol{C}(p)$and $\boldsymbol{Q}(p)$, respectively.

A message, m, is a tuple $(p, c, data)$ where data is the information to be sent and

$$p \in P \text{ and } c \in C$$

The set of all messages is denoted by M and the set of all sequences of messages is denoted by M^+. An application protocol, Π, is a relation from a state to a message, and also from a state to a state. That is:

$$\Pi: P \times C \times Q \times M^{+} \to M \cup \emptyset$$
$$\Pi: P \times C \times Q \times M^{+} \to Q \text{ [20].}$$

In order to differentiate between pseudocode names and predicates over variables, we define the predicates **D** and **B** over all messages and processes to represent, respectively, whether or not a processes has delivered or broadcast a message.

$\mathbf{D}(\mathrm{T}, p, m)$:
$\forall p \in P: \forall m \in M$:
process p delivered message m that came from broadcast protocol T

$\mathbf{B}(\mathrm{T}, p, m)$:
$\forall p \in P: \forall m \in M$:
process p broadcast message m using broadcast protocol T

Each of these constructs corresponds to a type of broadcast, that is:

T ∈
{Broadcast (B), Reliable Broadcast (R),
Causal Broadcast (C), Atomic Broadcast (A)}

When the broadcast type is unnecessary or implied by context, we will omit it and merely write $\mathbf{D}(p, m)$ or $\mathbf{B}(p, m)$. Note that, for simplicity, we assume that when a process broadcasts a message, it also sends the message to itself. Therefore a process always broadcasts a message for it delivers that message to itself.

5.2 Delivery Properties

We say a protocol is *valid* [3, 9] if, when any correct process broadcasts a message, m, then all correct processes eventually deliver m:

$\forall p \in P: \mathbf{Q}(p) \neq \perp: \mathbf{B}(p, m) \to$
$\forall q \in P: \mathbf{Q}(q) \neq \perp: \Diamond\, \mathbf{D}(q, m)$

We say a protocol *agrees* [3, 9] if, when any correct process delivers m, then all correct processes eventually deliver m:

$\forall p \in P: \mathbf{Q}(p) \neq \perp: \mathbf{D}(p, m) \to$
$\forall q \in P: \mathbf{Q}(p) \neq \perp: \Diamond\, \mathbf{D}(q, m)$

In our example, assume a process sends $x = x + 1$ to all processes, but crashes after only half the messages have been sent. Then half the correct processes deliver $x = x + 1$ and half do not; the behaviors do not agree.

We say a protocol has *integrity* [3, 9] if, for any m, every correct process delivers m at most once, and only if some process previously broadcast m:

$\forall m, m' \in M: \forall p \in P: \mathbf{Q}(p) \neq \perp:$
$\mathbf{D}(p, m) \wedge \mathbf{D}(p, m') \to$
$(m \neq m' \wedge \exists q \in P: \mathbf{B}(q, m))$

While properties like these allow us to define when a message is broadcast or delivered, they don't help to enforce requirements on the order of delivery between messages. In some settings it is helpful to impose order on broadcast messages.

5.3 Broadcast Order

A protocol is *first-in-first-out (FIFO)* if all messages delivered at some process q from some process p are delivered in the same order that p sent them. This is a partial order; messages originating at separate processes are incomparable [3, 9]. We show an example in Figure 1.

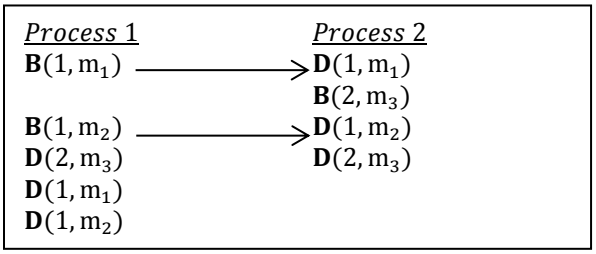

Figure 1. FIFO Order but not Causal: Messages from a process are delivered in the order they are broadcast. Message $\mathbf{m_1}$ is delivered at Process 2 before $\mathbf{m_3}$ is broadcast, but Process 1 delivers $\mathbf{m_3}$ before $\mathbf{m_1}$, so the order is not causal.

A causal order, Figure 2, expands this partial order to allow messages that originated at

separate processes to be compared. The idea of a causal order was originally proposed by Lamport [29] and was adapted to group communication settings. In the literature, the causal order relationship is often informally defined as a "happened before" relation. We say that a message m' causally precedes a message m, written $m' \underset{c}{\rightarrow} m$, iff:

1. If m' and m are messages from the same process and m' was broadcast before m was.
2. If m' is delivered by one process and m is later broadcast by that process.
3. If $m' \underset{c}{\rightarrow} x \wedge x \underset{c}{\rightarrow} m$.

A group communication protocol is a *causal order* if, any time a process broadcasts a message m' that causally precedes a message m, then no correct process delivers m unless it has previously delivered m' [3, 4, 5, 9, 29]. Causal ordering is strictly stronger than FIFO ordering because it orders all local messages of a process and provides a partial order over messages between processes [3, 9].

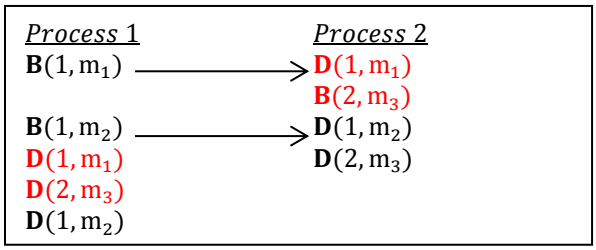

Figure 2. Causal Order but not Total: Messages $\mathbf{m_1}$ and $\mathbf{m_2}$ are delivered in order. Message $\mathbf{m_1}$ causally precedes message $\mathbf{m_3}$ and so must be delivered first. Messages $\mathbf{m_2}$ and $\mathbf{m_3}$ are not related and so can be delivered in any order. Because they are not delivered in the same order, the order is not total.

The next logical step is a group communication protocol that allows a total order over its messages, shown in Figure 3. We say that a group communication protocols satisfies a *total order* if whenever two processes, p and q, deliver messages m and m', p delivers m' before m iff q delivers m' before m [3, 9].

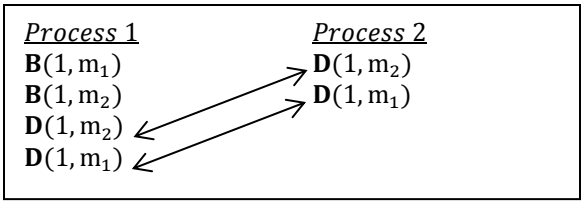

Figure 3. Total Order but not FIFO or Causal: All messages are delivered in the same order by all processes, so the order is total. The messages are delivered in a LIFO order, which is neither FIFO nor causal.

5.4 Multicast Order

While the aforementioned ordering properties make sense for broadcast, or even for multicasts to non-overlapping groups, inconsistencies arise when they are applied to overlapping groups. We will not address multicast here. Several solutions to these problems can be found in [9].

5.5 Uniformity

None of the delivery or order properties given above address incorrect processes. We say that a protocol is *uniform* for some requirement if the requirement is imposed on both correct and failing processes [3,20, 22] . This yields a corresponding uniform requirement for each of the correctness properties in Sections 5.2. For example, uniform validity is:

$$\forall p \in P: \mathbf{B}(p, m) \rightarrow \forall q \in P: :\Diamond\, \mathbf{D}(q, m)$$

Note that the only change to the property is that the predicate $\mathbf{Q}(p) \neq \perp$ is dropped, making the requirement hold for processes that are not in the premature halting state.

5.6 Timing

Until now, the requirements we have mentioned are asynchronous. In synchronous settings, timing requirements can be imposed on message transmission. We say that broadcasts under this specification are *timed broadcasts*. There are two variations. The strictest is *real-time δ-timed broadcast* where a message that is

broadcast at real time t is delivered at all necessary processes no later than real time $t + \delta$ for some δ. *Logical-time δ-timed broadcast* is a similar concept, but is based on the local time of a processor instead of a real time [3, 9, 20, 22]. These requirements only make sense in a system with comparable clocks, which have a known upper bound on their difference. We can model non-timed broadcasts or broadcasts in systems without comparable clocks as ∞-timed broadcasts [20, 22].

Other applications expect messages to be received periodically. In an artificial intelligence setting, some agent may periodically sense *precepts*, series of measurements from its environment, and act upon them. If the sensors of the agent fail, the program may still expect a precept. This could be achieved by a group communication protocol that uses terminating broadcast. A *terminating broadcast* is a broadcast that delivers a message during every round [3]. It may be the case that no message was received, in which case we define some flag value to represent this case and send it.

Finally, we must address atomicity. A fundamental concept in distributed systems, an atomic instruction can be thought of as an uninterruptible operation or one that appears to take zero time under some linearization of the programs execution history. We will not formally use the word atomic to describe group communication protocols except to name a protocol introduced in Section 6. Instead we will refer to the constraints that each group communication protocol satisfies.

6. Group Communication Protocols

We will now discuss several types of broadcast. Each of the following subsections will describe the properties that the broadcast must fulfill, give an overview of a potential implementation, argue informally for correctness, and discuss optimizations and variants. Each implementation is semantic; it meets the necessary requirements but ignores many real-world needs.

We provide pseudocode for many broadcast types, modified slightly from [3]. Following the conventions of Bernstein, Hadzilacos, and Goodman, the pseudocode format $\mathbf{broadcast}(\mathrm{T}, m)$ means that we send message m using a broadcast of type T.

We start by examining *basic broadcast*, $\mathbf{broadcast}(\mathrm{B}, m)$.

We then show that *reliable broadcast*, $\mathbf{broadcast}(\mathrm{R}, m)$, can be built from basic broadcast.

Reliable broadcast is the foundation for *FIFO broadcast*, *causal broadcast*, *and timed broadcast*, referred to in pseudocode as, $\mathbf{broadcast}(\mathrm{F}, m)$, $\mathbf{broadcast}(\mathrm{C}, m)$, and $\mathbf{broadcast}(R^{\Delta}, m)$.

Timed broadcast can be used to build timed *FIFO* and *timed Causal broadcasts*, $\mathbf{broadcast}(F^{\Delta}, m)$, and $\mathbf{broadcast}(C^{\Delta}, m)$.

We then proceed to build *timed causal atomic broadcast*, and *causal atomic broadcast*, $\mathbf{broadcast}(CA^{\Delta}, m)$, and $\mathbf{broadcast}(CA, m)$ from timed causal broadcast and FIFO atomic broadcast.

It is worth noting that each broadcast presented builds on a previous broadcast [3, 9, 23]. They grow along the dimensions of message order and timing requirements. The hierarchies shown in Figure 4 shows untimed versions but each broadcast has corresponding timed variations.

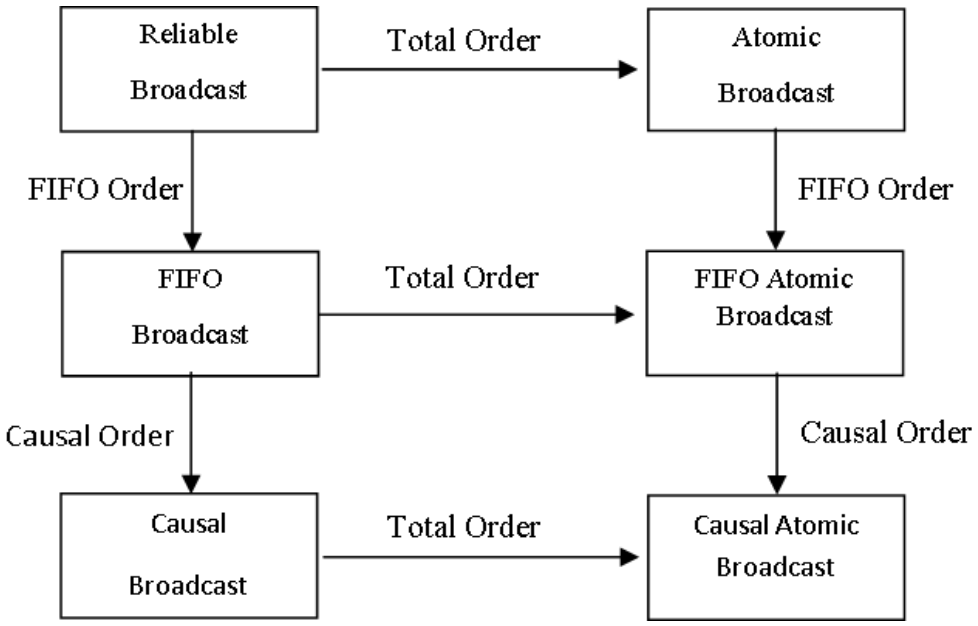

Figure 4. Broadcast Hierarchy [3]: Reliable broadcast is the foundation for all broadcast variations. No order, FIFO order, causal order, or total order order dictates the type of broadcast. Each of the six varieties above has an equivalent real-time and logical-time version, for a total of 18 broadcast variations.

6.1 Broadcast

The simplest of all the variants, *broadcast* makes no guarantee about correctness or order; it merely sends a message to all processes. This is implemented by having a process send point-to-point messages to every other process [9, 11].

```
Every Process p executes the following:

to execute broadcast(B, m):
    for each process i:
        send m to i

deliver(B, m) occurs as follows:
    upon receive m
        deliver(B, m)
```

Figure 5. Broadcast

This requires $O(N)$ messages per process. Broadcast has validity but does not maintain agreement or integrity [9]. That is, a successful broadcast of a message implies that all correct processes eventually deliver that message. If a correct process delivers a message, however, not all correct processes must deliver it, as the broadcast may have failed halfway through. The above implementation makes no requirements on timing and so can by implemented in a completely asynchronous system.

Broadcast makes no correctness claims, and therefore functions properly given any number of failures; it is N-resilient. Further, because broadcast has no synchronization requirements, it can be implemented in a completely asynchronous system.

We can impose a FIFO order by tagging messages with sequence numbers. Receiving processes will then wait to deliver a message until they are looking for its sequence number.

We can impose a causal order by sending all the messages that are causally related to the message we are trying to broadcast along with it. The receiving process will then deliver all the un-delivered previous messages, and then the current one.

We defer discussion of totally ordered broadcasts until Section 6.6.

Because broadcast does not guarantee agreement, there is no guarantee that a correct process will receive the sequence number (or all the causally related messages if they are split across sends) it is looking for. Without agreement, it is likely that processes that fulfill ordered broadcast will not deliver any messages. For this reason, ordered broadcasts are seldom used. In order to solve these problems, we introduce a new primitive, reliable broadcast.

6.2 Reliable Broadcast

Reliable broadcast guarantees validity, integrity, and agreement. Integrity and validity are maintained in the same way as broadcast. Agreement is ensured because whenever a correct process delivers a message, it then broadcasts it. Reliable Broadcast can be implemented trivially by having each process broadcast every message it receives to each other process, as shown in Figure 6. Each process delivers messages that it has not seen before [3, 9, 23]. This requires $O(N)$ messages per process or $O(N^2)$ across all processes. If we assume that the only failures are process failures and that the underlying network is fully connected, then this method requires $O(N)$ remote memory calls per process and $O(N^2)$ for the whole system.

```
Every Process p executes the following:

to execute broadcast(R, m):
    broadcast(B, m)

deliver(R, m) occurs as follows:
    upon deliver(B, m) do:
        if p has not previously
        executed deliver(R, m)
        then
            if sender(m) ≠ p
            then broadcast(B, m)
            deliver(R, m)
```

Figure 6. Reliable Broadcast using Broadcast [3]

If we use reliable broadcast to send messages in our example, then we guarantee that correct processes receive all correctly broadcast arithmetic operations. However, they may not receive them in the same order, yielding inconsistent views of $x's$ value.

6.3 FIFO Reliable Broadcast

We call a reliable broadcast a *FIFO reliable broadcast* if when a correct process, p, sends a message, m', before a different message, m, every correct process delivers m' before m. The reliable broadcast implementation shown in Figure 6 can be modified to be FIFO by attaching a sequence number to each message. A process then delivers a message from another process, p, only if it has not previously delivered it and the sequence number is the next number it expects from p [3, 9, 23]. This requires each process to have a local priority queue but does not change the message or time complexity. This construction is shown in Figure 7.

Every Process p executes the following:
Initialization:
    $msgBag = \emptyset$
    $next[q] \coloneqq 1$ for all q

to execute **broadcast**(F, m):
    broadcast(R, m)

deliver(F, m) *occurs as follows*:
    upon deliver(R, m) **do**:
        $q \coloneqq sender(m)$
        $msgBag \coloneqq msgBag \cup \{m\}$
     while ($\exists m' \in msgBag: sender(m')$
            and *seq#(m')=next[q])* **do**
            deliver(F, m')
            $next[q] \coloneqq next[q] + 1$
            $msgBag \coloneqq msgBag - \{m'\}$

Figure 7. FIFO Reliable Broadcast using Reliable Broadcast [3]

6.4 Causal Broadcast

Causal reliable broadcasts are called *causal broadcasts*. They guarantee validity, integrity, and agreement and additionally impose a causal order on the messages sent. Causal broadcast is shown in Figure 8.

Every Process p executes the following:
Initialization:
    $prevDlvrs = \perp$

to execute **broadcast**(C, m):
    broadcast(F, $< prevDlvrs || m >$)
    $prevDlvrs \coloneqq \perp$

deliver(C, m) *occurs as follows*:
    upon deliver(F, $\langle m_1 \dots m_l \rangle$) **do**:
        for i $\coloneqq$ 1 … l **do**
            if p has not previously
                executed **deliver**(C, m_i)
            then
                deliver(C, m_i)
                $prevDlvrs \coloneqq prevDlvrs || m_i$

Figure 8. Causal Broadcast using FIFO Broadcast [3]

The time complexity of this implementation is still $O(N)$ per process and $O(N^2)$ across the system. The message complexity is also the same as previous broadcast implementations, but each message can be much larger.

Reliable broadcast guarantees nothing about failed processes. Because of this fact, the reliable, FIFO, and causal broadcasts above can tolerate any number of failures; they are N-resilient to arbitrary failures and by extension all weaker failures. Further, none of the broadcast variations we presented here have any timing requirements and so they can be implemented in a completely asynchronous system.

While the aforementioned reliable broadcast variants are not as efficient as they could be, they are modular; the FIFO reliable broadcast is built using reliable broadcast and causal broadcast is built using FIFO reliable broadcast. Simple optimizations such as eliminating the message that processes send to themselves and piggybacking ACKs at the link layer can be implemented. They are covered in depth in [4, 5, 9, 23].

6.5 Timed Broadcast

In a completely asynchronous setting, implementing real-time or local-time broadcasts is impossible [3]. However, if we make some assumptions about the underlying system, we can achieve both results. Assumptions from [3] are given below:

1. At most f processes can fail.
2. Every two correct processes are connected via a path of length at most d, constisting entirely of correct processes and links.
3. There is a known upper bound δ on message delay.
4. The time to execute a local step is taken to be zero.
5. Clocks of correct processes are approximately synchronized to within ϵ of each other, and their drift with respect to real time is bounded by ρ.

The reliable broadcast scheme given in Section 6.2 and shown in Figure 6 satisfies the requirements for a real-timed broadcast if assumptions one through four, from above, are met and there are no timing failures. If a broadcast was initiated at time t then all correct processes would deliver the broadcast by time $t + (f + d)\,\delta$, so it is a real-time $(f + d)\delta$-reliable broadcast. It is important to note that if we add uniformity to assumption 2, then the timed broadcast also becomes a uniform real-time $(f + d)\delta$-timed reliable broadcast [3].

The previous construction worked under the assumption that there are no timing failures. If we allow timing failures then it is impossible to implement a real-timed broadcast. We can build a local-timed broadcast under assumptions 1 through 5. We assume that the bound, δ, only applies to messages sent between correct processes on correct links. The construction is a modification of the reliable broadcast protocol from Section 6.2. When a message is sent it is marked with a timestamp and a counter, k, that is incremented on each link traversal. When a process receives a message, it calculates the timestamp of that message minus the current timestamp. If this value is greater than expected, the receiving process discards the message. Otherwise it sends the message to all of its neighbors before delivering it [3].

Both of these constructions can be extended to real-timed and local-timed FIFO and Causal broadcasts merely by using the corresponding underlying timed reliable broadcast. We show constructions of timed atomic broadcasts in Sections 6.7, 6.8, and 6.9.

6.6 Atomic Broadcast

Totally ordered reliable broadcasts are called *atomic broadcasts*. To construct an atomic broadcast we must have all processes agree on the order of all messages. At a more basic level, we must have all processes agree on a sequence number for each individual message. If each message has its own globally unique sequence number then atomic broadcast can be implemented in the same way we built FIFO reliable broadcast [3, 9].

We can assign sequence numbers by declaring one process to act as a sequencer. Each time a process wants to broadcast it first sends a point-to-point message to the sequencer which replies with a sequence number. The original broadcast then reliably broadcasts its message along with the assigned sequence number. This takes one more message than reliable broadcast implementation and so has $O(N^2)$ message complexity [9]. Though the asymptotic message complexity is identical, the latency doubles.

A centralized sequencer has a maximum of 1-resilience without some replication or failure scheme, because the sequencer may fail. While replicated schemes have been built and are often used in practice, it may be necessary to provide atomic broadcast in a non-centralized way.

To build a non-centralized solution, we can have a message broadcast a request for a sequence number for its next message, along with a proposed sequence number. Each process that delivers this request will respond with a proposed sequence number one greater than the highest sequence number it has yet broadcast. The original process will then choose the maximum sequence number over all proposed sequence numbers as the value for its next message. It will then reliably broadcast that

message with the agreed upon sequence number [9].

If our example uses atomic broadcast to send its arithmetic operations, then all correct processes receive all operations in the same order, guaranteeing that, eventually, all processes arrive at the same value for x.

Ultimately, in order to build atomic broadcast we must have a way for a group of N processes to agree upon a value. This is a famous problem in the field of distributed systems and is called the consensus problem [25, 27, 28]. It can be used to solve the atomic broadcast problem as shown in Figure 9, where R is a reliable broadcast protocol and A is the atomic broadcast protocol.

We saw in Section 4 that distributed consensus cannot be solved in a completely asynchronous system that is deterministic. This means that atomic broadcast cannot be implemented in that system. It follows from our impossibility results in Section 4 that atomic broadcast can be implemented in any system with a synchrony mechanism that allows a process to differentiate between a failed process and a slow process. Systems with comparable clocks, synchronized message order, synchronized communication, or any of many other synchronization primitives allow us to implement atomic broadcast. We can implement atomic broadcast in any system where we can solve the distributed consensus problem [13].

In the same way that atomic broadcast can be solved using consensus, we can also solve consensus using atomic broadcast, as can be seen in Figure 10, where A stands for the atomic broadcast protocol. In this sense atomic broadcast and consensus are equivalent problems; if we can solve one then we can solve the other [3].

In Section 7 we cover several non-deterministic methods for solving distributed consensus. These methods can be used to build a non-deterministic atomic broadcast protocol by building a sequencer out of the consensus solution and using it to give a total order over all messages.

Shostak, Lamport, and Peace [27] showed that the consensus problem can be solved given at most $\frac{N}{3}$ arbitrary failing processes. In the same way, atomic broadcast is $\frac{N}{3}$-resilient for arbitrary failures and by extension for any weaker failures.

Every process p executes the following:
$broadcast := \emptyset$
$delivered := \emptyset$
$k := 0$

to execute **broadcast**(A, m):
    broadcast(R, m)

deliver(A, m)*occurs as follows*:
    upon deliver(R, m) **do**
        $broadcast := broadcast \cup m$
        if $broadcast - delivered = \emptyset$
            end
        $k := k + 1$
        $undelivered :=$
            $broadcast - delivered$
        $msgs_k =$
            Consensus$(k, undelivered)$
        deliver all $m' \in msgs_k$
            in some deterministic order
        $delivered := delivered \cup msgs_k$

Figure 9. Atomic Broadcast Reduces to Consensus [7]

Every process p executes the following:
    $acceptedValue := \perp$

to execute **propose** ($value$):
    broadcast($A, value$)

decide *occurs as follows*:
    upon deliver(A, $value$) **do**
        if $acceptedValue = \perp$
        then
            $acceptedValue := value$
        else
            discard $value$

Figure 10. Consensus Reduces to Atomic Broadcast

6.7 Timed Atomic Broadcast

A different means of implementing atomic broadcast is by building it out of a timed reliable broadcast. Assume we have a local-timed reliable broadcast, R. We can build a timed atomic broadcast from R. To atomically broadcast a message, we merely broadcast the message using R. When a message is delivered by R, the atomic broadcast protocol schedules its delivery of the message at $t + \delta$ where t is the original timestamp of the message and δ is the upper bound on message delay. The construction is shown in Figure 11. If two messages are scheduled to be delivered at the same time then process IDs or message IDs can be used to break the tie [3]. It is worth noting that if the underlying timed reliable broadcast is uniform then the timed atomic broadcast will be uniform

Every Process p executes the following:

to execute **broadcast**(A^{Δ}, m):
    broadcast(R^{Δ}, m)

deliver(A^{Δ}, m) *occurs as follows*:
    upon deliver(R^{Δ}, m) **do**:
        schedule deliver(A^{Δ}, m) **at time** $timestamp(m) + \Delta$

.

Figure 11. Timed Atomic Broadcast using Timed Reliable Broadcast [3]

6.8 FIFO Atomic Broadcast

First-in-first-out atomic broadcast is simple to implement given a system where atomic broadcast is possible. Atomic broadcast guarantees total order, a FIFO implementation merely restricts that order. To build a FIFO implementation we must ensure that all the sequence numbers of a process are ordered such that they will be delivered in the same order they are sent. This construction is identical to the one provided for reliable broadcast in Section 6.2 and shown in Figure 4 [3].

6.9 Causal Atomic Broadcast

We will build a causal atomic broadcast in two ways. First from a timed causal broadcast and then from a FIFO atomic broadcast.

Constructing a timed causal atomic broadcast is simple. We will use the same strategy we used to build the timed atomic broadcast. Assume we have a local-timed causal broadcast, C. To atomically broadcast a message, we merely broadcast the message using C. When a message is delivered by C, the causal atomic broadcast protocol schedules its delivery of the message at $t + \delta$ where t is the original timestamp of the message and δ is the upper bound on message delay. We can break ties by ID. Again, if the underlying broadcast is uniform then the timed causal atomic broadcast will be uniform [3]. This is shown in Figure 12.

Every Process p executes the following:

to execute **broadcast**$(\mathrm{CA}^{\Delta}, m)$:
    broadcast(C^{Δ}, m)

deliver$(\mathrm{CA}^{\Delta}, m)$ *occurs as follows*:
    upon deliver(C^{Δ}, m) **do**:
        schedule deliver$(\mathrm{CA}^{\Delta}, m)$ **at time** $timestamp(m) + \Delta$

Figure 12. Timed Causal Atomic Broadcast using Time Causal Broadcast [3]

The second construction is from FIFO atomic broadcast and is shown in Figure 10. In the figure *CA* stands for causal atomic and *FA* stands for FIFO atomic. The set $prevDlvrs$ is the set of messages p received since its last broadcast. The set $suspects$ is a set of processes that p suspects are faulty [3].

```
Every process p executes the following:
Initialization:
        prvDlvrs := ∅
        suspects := ∅

To execute broadcast(CA, m):
    broadcast(FA, ⟨m, prvDlvrs⟩)
    prvDlvrs := ∅

deliver(CA, m) occurs as follows
    upon deliver(FA, ⟨m, prevDlvrs⟩) do
        if sender(m) ∉ suspects and
          P has previously executed
         deliver(CA, m′) for all
         m′ ∈ prevDlvrs

        then
            deliver(CA, m)

            prvDlvrs := prvDlvrs ∪ {m}
        else
            discard m
            suspects := suspects
                ∪ {sender(m)}
```

Figure 13. Causal Atomic Broadcast using FIFO Atomic Broadcast from [3]

To broadcast a message, p broadcasts using the underlying FIFO protocol. To deliver a message, p first delivers it using the FIFO protocol. The process p then ensures that all previous messages have been delivered and that the sending process is not a member of the suspected processes. If both predicates are true, then p delivers the message in its causal atomic protocol. Otherwise it discards the message, assuming that the sending process is faulty [3].

There are several optimizations to this implementation that are covered in [3, 9, 23]. Additionally, this method of suspecting that processes are faulty introduces the concept of failure detectors, a powerful non-deterministic method for solving the consensus problem that we cover in more depth in Section 7.3.

```
Input: Boolean value input
Ouput: Boolean value stored in output
Data: Boolean preference, integer round

begin
    preference := input
    round := 1
    while true do
        send(1, round, preference) to all processes
        wait to receive n − t (1, round, ∗) messages
        if received more than n/2 (1, round, v) messages
        then
            send (2, round, v, ratify) to all processes
        else
            send (2, round, ?) to all processes
        end
        wait to receive n − t (2, round, ∗) messages
        if received a(2, round, v, ratify) message
        then
            preference := v
            if received more than t (2, round, v, ratify)
                                    messages
                output := v
            end
        else
            preference := CoinFlip()
        end
        round := round + 1
    end
end
```

Figure 12. Randomized Consensus from Ben-Or [1]

7. Different Approaches

Each of the preceding atomic broadcast algorithms requires a solution to the consensus problem. We saw earlier that deterministic consensus is impossible. In this section we will discuss three methods of solving the non-deterministic consensus problem in completely asynchronous systems: randomization, approximation, and failure detectors.

7.1 Randomization

Ben-Or followed the impossibility results given by Fischer, Lynch, and Peterson [18] by solving the consensus problem in a nondeterministic manner. His method, shown in Figure 11 below, solves the consensus problem

where N processes are agreeing on a binary value [1].

The system proceeds in rounds. During each round a process, p, proposes a potential value which it sends to all other processes. It also delivers the proposed values of other processes. If p decides on a value then all other processes will decide the same value in the next round. Otherwise the value will be decided in the next round with some bounded probability [1, 2]. The authors show that the consensus problem can be solved with probability one given fewer than $\frac{N}{5}$ Byzantine failures [1, 2]. Their algorithm may run for unbounded time; however, the probability that it does not terminate is zero. On average the algorithm will terminate in a constant number of steps. Further, they show that on average this solution will take a constant number of rounds, circumventing the impossibility results found by Dolev, Fischer, and Lynch [13] that require $N + 1$ rounds to decided consensus.

Bracha later improved these results by changing the communication primitives in the algorithm proposed by Ben-Or from point-to-point message passing to reliable broadcasting. These modifications allowed the consensus problem to be solved in the presence of up to $\frac{N}{3}$ Byzantine failures [6]. This matches the optimal bounds presented by Lamport, Shostak, and Peace [27]. The result shown by Bracha is particularly relevant for this paper as it displays the power of using reliable broadcast as a primitive instead of conventional point-to-point communication.

The preceding results for consensus are important for group communication primarily because they show that atomic broadcast can be implemented using randomization in completely asynchronous systems.

Sequence numbers can be determined using this algorithm. The sequence numbers will be decided, on average, in a constant number of steps, but the time required to agree, and by extension the time required to broadcast, is unbounded. However the probability that the broadcast will never terminate is zero.

7.2 Approximation

The consensus problem deals with all processes deciding on a binary value. Dolev, *et. al*. have shown that if we relax the problem so that the processes only have to decide on the same value to within some margin of error then the consensus problem can be solved in a completely asynchronous system [14]. The error bounds depend on the ratio of the number of Byzantine failures to the total number of processors. This is especially important for a fault-tolerant system because it allows the approximation to degrade gracefully as the system experiences an increasing number of failures. Their algorithm is guaranteed to converge when there are no more than $\frac{N}{3}$ faulty processors, a result that is consistent with the results shown by Lamport, Shostak, and Peace [27].

The method employed by Dolev, *et. al.* is to perform a series of successive approximations. Whenever a process receives proposed values from another process it runs them through an averaging function to obtain a new proposed value. This method approximates with arbitrary precision [14, 16].

The reader may note that the number of failures and the results obtained by approximation are quite similar to those obtained by the randomization method above. Both methods achieve optimal results, but they achieve them in quite different ways; Dolev, *et. al*. use a successive series of approximations to solve the binary consensus problem to within some error bounds while Ben-Or and Bracha use a probabilistic algorithm to exactly solve the consensus problem with some probability.

Approximate atomic broadcast can be built in a completely asynchronous system using this method. The system must be able to tolerate some bounded error in sequence number agreement. This means that some bounded error must be accepted in the total order.

7.3 Failure Detectors

We have mentioned that the primary difficulty in solving consensus, and by reduction atomic broadcast, in an asynchronous system is

that we cannot differentiate between a failed process and a slow process. We show here that allowing a process to suspect another process gives us enough synchrony to solve the consensus problem in many scenarios. We will first formally define what it means to suspect a process.

A *failure detector* is an external mechanism for deciding whether or not a process has failed. A failure detector maintains a set of processes that it believes have failed. The failure detector can make mistakes; it can add a correct process to this set. The failure detector may continuously add and remove processes [3, 7].

Each process has access to a *local failure detector* that monitors a subset of the processes in the system. A process can query its failure detector as an oracle to ascertain whether its local failure detector believes any given process has failed. We say that a process suspects another process if it queried its failure detector and its failure detector replied that the requested process had failed [3, 7].

No process, even the local one, may access a failure detector that belongs to a failed process. If a process, p, is suspected of failure by another correct process but is actually correct then p must continue to act in a correct manner. For example, if p reliably broadcasts a message, m, then all other processes, including those that suspect p has failed, must eventually deliver m [3, 7].

A set of failure detectors satisfies *strong completeness* if eventually every process that crashes is permanently suspected by every correct process. A set of failure detectors satisfies *weak completeness* if eventually every process that crashes is suspected by some correct process [7].

Completeness can be trivially satisfied if a failure detector suspects all process. A failure detector must also be accurate. A set of failure detectors fulfills *strong accuracy* if correct processes are never suspected and *weak accuracy* if some correct process is never suspected [7].

Each completeness and accuracy requirement has a corresponding eventual requirement. We say that a set of failure detectors fulfills an *eventual* requirement if there is a time after which that requirement is satisfied [7].

The set of failure detectors that satisfies both strong completeness and strong accuracy is called the set of *perfect failure detectors*. The set that satisfies strong completeness and weak accuracy is called *strong failure detectors*. *Weak failure detectors* satisfy weak completeness and weak fairness. Each of these sets corresponds to a set that holds the same eventual properties [7].

Chandra and Toueg show that any weak failure detector can be transformed into a strong failure detector. Further, they show that any eventually weak failure detector can be transformed into an eventually strong failure detector [7].

They then show that consensus can be solved by strong and eventually strong failure detectors, indicating that the weaker counterparts can also solve the consensus problem. By solving consensus, they allow a solution to the atomic broadcast problem. They conclude with the following corollaries relating the atomic broadcast solution to the number of failures, f, and the number of processes, n [7]:

1. Given any weak failure detector, atomic broadcast is solvable in asynchronous systems with $f < n$.
2. Given any eventually weak failure detector, atomic broadcast is solvable in asynchronous systems with $f < \frac{n}{2}$.

Like the probabilistic solution presented earlier, this solution has potentially unbounded time complexity. Failure detectors are defined such that their properties hold eventually this is necessary; it is what makes the solution non-deterministic. In practice, systems are not completely asynchronous, so solutions are not unbounded. It is important to notice that timeouts, the conventional method for differentiating failed processes from slow processes, are a real-life example of the use of failure detectors.

8. Inconsistency and Contamination

The careful reader may have noticed that many of our correctness conditions allow for

arbitrarily failing processes to continue to affect the system. For example the reliable broadcast specifications do not address what a correct process should do when it receives a broadcast from a failed process; the correct process can deliver the message and still be considered correct.

When a process fails in a way that makes the internal state of that process different from what it ought to be, then that process is called *inconsistent.* The spread of inconsistency from failed processes to correct processes is called *contamination* [3, 20, 22].

Given the numerous correctness conditions already covered in this paper it may seem like this is not a common occurrence. Indeed, most systems do not display arbitrary failures and much of the literature is concerned with crash failures. However, because the broadcast and deliver components may both experience distinct failures, it is possible for omission failures to turn into arbitrary failures [3, 20, 22].

Informally, atomic broadcast delivery consistency can be attained by imposing the additional requirement that "the sequence of messages delivered by *every* process (correct or faulty) is a *prefix* of the sequence of messages delivered by the correct process during the entire execution." Atomic broadcast consistency can be realized if, "a process may only broadcast a message after all of its previous broadcasts are successfully completed" [20].

Contamination for an atomic broadcast can be avoided if, when a process p, delivers any message, m, broadcast by a process, q, at or before time c, then the sequence of messages that q had delivered when it broadcast m must be a prefix of the entire sequence of messages p has ever delivered [20].

Solving the common broadcast problems in a manner that prevents inconsistency and contamination significantly complicates the constructions. Book-length theses have been written on the topic. We refer the interested reader to Gopal [20] who covers the extensive territory more formally in addition to presenting multicast protocols for ordered reliable and atomic broadcasts that avoid problems of inconsistency and contamination.

9. Conclusion

Group communication is vital to the construction of any distributed system. Because it is not often supported at the operating system level, it is important to be able to implement broadcast protocols at a higher level.

We have shown modular constructions for broadcast, reliable broadcast, atomic broadcast, ordered broadcasts, and timed broadcasts along with informal analyses of their complexities. Each of these protocols works in some subset of systems. The implementer should carefully investigate the level of synchrony and the fault model necessary for their system.

While many broadcasts are not possible in completely asynchronous systems, in practice there are plenty of practical implementations. These range from supporting some synchronization, to more nuanced approaches such as approximation, randomization, or failure detectors.

10. References

[1] J. Aspnes, "Randomized Protocols for Asynchronous Consensus," Distributed Computing, Vol. 16, 2002, pp. 2-3.

[2] M. Ben-Or, "Another advantage of free choice: Completely asynchronous agreement protocols," Proceedings of the Second ACM Annual Symposium on Principles of Distributed Computing, 1983, pp. 27-30.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman, "Distributed Systems," Addison-Wesley, 1987.

[4] K. Birman, T. Joseph, "Reliable Communication in the Presence of Failures," ACM Transactions on Computer Systems, Vol. 5, Issue 1, Feb. 1987, pp.47-76.

[5] K. Birman, A. Schiper, and P. Stephenson, "Lightweight Causal and Atomic Group Multicast," ACM Transactions on Computer Systems, Vol. 9, No. 3, 1991, pp. 272-314.

[6] G. Bracha, "Asynchronous Byzantine Agreement Protocols," Information and Computation, Vol. 75, No. 2, 1987, pp.130-143.

[7] T. Chandra, S. Toueg, "Unreliable Failure Detectors for Reliable Distributed

Systems," Journal of the ACM, Vol. 43, No. 2, March 1996, pp. 225-267.

[8] J. Chang, N.F. Maxemchuk, "Reliable Broadcast Protocols," ACM Transactions on Computer Systems, Vol. 2, Issue 3, Aug. 1984, pp.251-273.

[9] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, "Distributed Systems: Concepts and Design," Edition Five, Addison-Wesley, 2012.

[10] F. Cristian, H. Strong, and D. Dolev, "Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement," Proceedings of the 15th International Symposium on Fault-Tolerant Computing, 1985, pp. 200-206.

[11] P. Dewan, Distributed Systems Course, The University of North Carolina at Chapel Hill, Fall 2011.

[12] E.W. Dijkstra, "Self-stabilizing Systems in Spite of Distributed Control," EWD391, in Selected Writings on Computing: A Personal Perspective, Springer-Verlag, Berlin, 1982, pp. 41-46.

[13] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consenus," Journal of the ACM, Vol. 34, No. 1, 1987, pp. 77-97.

[14] D. Dolev, N. Lynch, S. Pinter, E. Startk, and W. Weihl, "Reaching Approximate Agreement in the Presence of Faults," Journal of the ACM, Vol. 33, No. 3, 1986, pp. 499-516.

[15] P.D. Ezhilchelvan, "Newtop: A Fault-Tolerant Group Communication Protocol," Proceedings of the 15th International Conference on Distributed Computing Systems, 1995, pp.296-306.

[16] A. Fekete, "Asymptotically Optimal Algorithms for Approximate Agreement," Distributed Computing, Vol. 4, No. 1, 1990, pp.9-30.

[17] A. Fekete, "Asynchronous Approximate Agreement," Proceedings of the Sixth Annual Symposium on Principles of Distributed Computing, 1987, pp.64-76.

[18] M. Fischer, N. Lynch, and M. Patterson, "Impossibility of Distributed Consensus with One Faulty Process," Journal of the ACM, Vol. 32, No. 2, April 1985, pp. 374-382.

[19] H. Garcia-Molina, A. Spauster, "Ordered and Reliable Multicast Communication," ACM Transactions on Computer Systems, Vol. 9, No. 3, 1991, pp. 242-271.

[20] A. Gopal, "Fault-Tolerant Broadcasts and Multicasts: The Problem of Inconsistency and Contamination," Cornell University, Ph.D. Dissertation, 1991.

[21] A. Gopal, R. Strong, S. Toueg, and F. Cristian, "Early-Delivery Atomic Broadcast," Proceedings of the Ninth ACM Annual Symposium on Principles of Distributed Computing, 1990, pp. 297-310.

[22] A. Gopal, and S. Toueg, "Inconsistency and Contamination," Proceedings of the Tenth Annual Symposium on Principles of Distributed Computing, 1991, pp.257-272.

[23] V. Hadzilacos, and S. Toueg, "A Modular Apprach to Fault-Tolerant Broadcasts and Related Problems," Technical Report, Department of Computer Science, University of Toronto.

[24] M. Kaashoek, A. Tanenbaum, "Fault Tolerance Using Group Communication," ACM SIGOPS Operating Systems Review, Vol. 25, Issue 2, April 1991, pp. 71-74.

[25] L. Lamport, "The Part-time Parliament," ACM Transactions on Computer Systems, Vol. 16, No. 2, May 1998, pp. 133-169.

[26] L. Lamport, "Paxos Made Simple," ACM SIGACT News (Distributed Computing Column), Vol. 32, No. 4, pp. 18-25, December 2001.

[27] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," ACM Transactions on Programming Languages and Systems, Vol. 4, No. 3, July 1982, pp. 382-401.

[28] L. Lamport, R. Shostak, and M. Pease, "Reaching Agreement in the Presence of Faults," Journal of the ACM, Vol. 27, No. 2, April 1980, pp. 228-234.

[29] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Communications of the ACM, Vol. 21, No. 7, July 1978, pp. 558-565.

[30] L.E. Moser, P.M. Meliar-Smith, D.A. Agarwal, R.K. Budhia, C.A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication System,"

Communications of the ACM, Vol. 39, Issue 4, April 1996, pp. 54-63.

[31] M. Tanenbaum, K. Verstoep, "Using Group Communication to Implement a Fault-Tolerant Directory Service," International Conference on Distributed Computing Systems, 1993, pp.130-139.