Massachusetts Institute of Technology
6.895: Advanced Distributed Algorithms
Professor Nancy Lynch

Lecture 10
October 8, 2002

Fault-Tolerant Broadcasts

Today's papers:
V. Hadzilacos and S. Toueg. A modular approach to the specification and implementation of fault-tolerant broadcasts. May 1994
Kenneth Birman, et al. Bimodal multicast. May 1999

1 Hadzilacos, Toueg

1.1 Overview

The topic of this lecture is fault-tolerant broadcasts. The purpose of this lecture is to study the implementation of some message-passing communication systems that can be used in distributed algorithms. Not only will this lecture cover how to build reliable broadcast systems, but it also covers what different types of broadcasts systems there are and how to determine which type is suitable for a particular distributed system. Roughly half of the lecture will just cover the types and their uses and understanding the subtleties between them. The other half will cover transformations from a weaker broadcast to a stronger one, including Atomic Broadcast. The transformations to stronger broadcasts are independent of the underlying network topology and implementation and thus can be used with any network. It follows then that if it is possible to produce the weakest kind of broadcast, then it is possible to produce the stronger types of broadcasts. This provides stronger co! mmunication primitives needed for fault-tolerant, distributed applications. An interesting result is that Atomic Broadcasts produces enough synchrony to solve consensus, and that reduction will be given. Then reliable broadcast will be generalized to reliable multicast.

1.2 Background Information on Networks

1.2.1 Types of Communication Networks

Most of the transformations, described in section 5, are independent of the type of network used, which is desirable because it is possible to design an algorithm for a computer system without having to change any network that may be in place. This lecture will not really cover the underlying topology of the networks, since they are not important to understand with regard to the transformations. However, the paper covers point-to-point networks pretty intensely, especially in section 6 when they go over how to use this specific topology to implement all the various kinds of broadcasts. Another very common type of network is the broadcast channel, which includes Ethernet and Token Ring. Others include redundant broadcast channel, packet radio, and switched-bases. These may all be used with the transformations.

1.2.2 Properties of Networks

Ideally, all correct networks that are modeled have two properties:

1. If a process sends a message to another process, the second process eventually receives the message.
2. Every process executes an infinite sequence of steps. (Processes that stop are models as performing an infinite number of no-op's when it stops.)

In addition, some networks are synchronous if they exhibit the following three additional properties to make it possible to detect when process crashes by timeouts:

1. There is a known upper bound on the time needed to process a step.
2. Every process has a local clock with known bounded rate of drift with respect to real time.
3. There is a known upper bound on message delay, including time to send, transport, and receive a message through a link.

1.2.3 Ways Networks Fail

First, benign failures are defined, since these are the failures assumed possible for the broadcast specifications covered here.

1. crash failure - A process stops executing after a certain point in time.
2. send-omission failure - m is not inserted into the outgoing message buffer, but completes the sending of m anyway.
3. receive-omission failure - If m is inserted into the incoming message buffer, but does not receive m.
4. link-omission failure - In a link from p to q, m is inserted into p's outgoing message buffer, but m never gets transported into q's incoming message buffer.

In addition, there are the following additional benign failures for synchronous networks.

1. clock failure - There is some point at time in which a process's local clock drifts with respect to real time at a rate that exceeds the limit specified.
2. performance failure - A process takes more time to commit a step that the specified maximum.

There are also Byzantine failures. In a network prone to such failures, it is necessary to strengthen the broadcast specifications we are about to cover. Anyone interested in seeing these specifications can see section 3.10 in the paper.

1.3 Broadcast Specifications for Benign Failures

This section covers different broadcast specifications. Reliable broadcast, the weakest kind of fault-tolerant broadcast, only requires that the broadcast is reliable, but imposes no ordering on the order of delivery of the broadcasts. The other types of broadcasts are all forms of reliable broadcasts, but impose differing standards on the delivery order of the messages. All the specifications are given, along with intuition on what type of system these broadcast networks are suitable for. Broadcast A is as strong as Broadcast B if A satisfies the requirements of B's specification. Oftentimes, I will say that A is stronger than B if A satisfies B's specification, but B does not satisfy A's specification.

1.3.1 Reliable Broadcast

Reliable Broadcast gives the minimal requirements needed to ensure that correct processes will eventually deliver messages correctly. These requirements are:

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

These requirements imply that in reliable broadcast, due to validity and agreement, all messages broadcast by a correct process gets broadcast correctly. If a faulty process broadcasts a message, correct processes can either choose to all deliver this message or all will not deliver the message. Also, these requirements allow a faulty process to deliver incorrect messages or not deliver correct messages, but the integrity requirement does not allow correct processes to further deliver this message.

1.3.2 FIFO Broadcast

FIFO broadcast is a reliable broadcast (thus ensures all the properties above) and in addition, requires that in order for a broadcast to be correct, all previous broadcasts from the same process must be already delivered by all correct processes and in the order they were broadcasted. Officially, it is defined below.

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.*

FIFO broadcast is suitable for networks where the contents of any particular broadcast depends only on the previous broadcasts originating from the same process. A good system to consider is a grocery store checkout registers. What they were trying to emphasize about that system is that a particular computer may broadcast a message to cancel the purchase of an item, which only makes sense if the previous broadcast to make the reservation was already received. Thus, this system certainly needs FIFO ordering to work correctly. The purchases made from one register are presumed independent from the purchases made from another register.

1.3.3 Causal Broadcast

Causal Broadcast is a stronger type of broadcast than FIFO broadcasts. The paper defines a term called causal precedence to define causal order and then proves this ordering is equal to FIFO order + Local order. Both definitions are given.

In order to understand causal order, it is first necessary to define causal precedence.
Step e causally precedes step f, (denoted e . f), iff:

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 . h and h . f.

Now, the official definition of causal order is given.

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.*

Not only are all broadcasts from one particular process delivered in the order broadcasted, but any message a process delivers prior to making a broadcast is delivered before the broadcast. It is easy to see how this definition of causal ordering also exhibits FIFO ordering. A message delivers its previous broadcast before making another broadcast, thus a previous broadcast from a single processor must be broadcast before a later one. Note that this particular definition not only restricts the delivery order of messages, but all that all messages a process delivers before a broadcast are delivered everywhere by all correct processes before delivering the broadcast.

Now, define local order:
Local Order: *If a process broadcasts a message m and a process delivers m before broadcasting m', then no correct processes delivers m' unless it has previously delivered m.*
Causal Order: FIFO Order + Local Order

Causal Order is necessary for networks where processes broadcast responses to other process's broadcasts. This may include a community talk channel. For example, user A broadcasts a message and user B receives this message and thereafter, broadcasts a response to A's message. All processes should receive A's message before B's response.

1.3.4 Atomic Broadcast

Sometimes, the delivery order of two messages is pertinent to maintaining consistency within a system. Atomic Broadcasts ensures that simultaneous broadcasts from two different processors are delivered in the same consistent order by all correct processes. Atomic Broadcast ensures what is called Total Order:

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'.*

Banks Systems require Atomic Broadcasts to act consistently. One process can initiate an increase of x by 10 percent, whereas another process can simultaneously broadcast a message to add 20 to x. The resulting value of x will differ depending on what messages was seen first. Thus, it is necessary for each process to see the messages in the same order.

An important thing to note about Atomic Broadcast is that it does not ensure FIFO ordering. Atomic Broadcast only ensures consistency in the ordering of all messages that are delivered by correct processes, but allows a process to miss a broadcast from another process. In FIFO ordering if a correct process never delivers the first message from a faulty process, then it will never deliver any messages from that process. In atomic broadcast, the correct process could still deliver other messages from the faulty process even after having skipped one.

1.3.5 Hybrid of these Broadcast and Relations Between Them

Atomic Broadcasts can be combined with FIFO or Causal ordering in order to produce even stronger delivery properties. Consulting p. 17 of the paper will show the relations between these orderings. An arrow indicates an extra delivery property which produces a new kind of stronger broadcast. If you can trace a path backwards from a broadcast A to another broadcast B, than A also satisfies the requirements of broadcast B. This diagram also describes what kinds of transformations will be needed to make the different kind of broadcasts. For example, to have a FIFO Atomic broadcast, either apply the transformation for total order and then for FIFO order or vice versa.

1.3.6 Additional Properties for all Broadcasts

Δ-Timeliness - A broadcast exhibits Δ-Timeliness when if a message is delivered at all, it is delivered within Δ after it was broadcast, where Δ is a known bounded number. There are two flavors of Δ-Timeliness:

- 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 + Δ.
- Local-Time Δ- Timeliness: There is a known constant Δ such that no correct process delivers m after local time ts(m) + Δ according to its own clock.

Uniform Broadcasts - Uniform broadcasts restrict the delivery behavior of faulty processes. Some uniform properties are:

- Uniform Agreement: If a process(whether correct or faulty) delivers a message m, then all correct processes eventually deliver m.
- Uniform Integrity: For any message m, every process (whether correct of faulty) delivers m at most once, and only if m was previously broadcast by sender(m).
- Uniformed Timeliness: Faulty processes also don't deliver the message after the deadline.
- Uniform Ordering: Faulty processes can't deliver messages out of order either.

1.3.7 Potential Problems with Broadcasts

Hadzilacos and Toueg point out that their specifications for reliable broadcasts still can't ensure that a system communicates the way intended, and they give specific examples of how errors can still occur. In particular, they point out that even broadcasts that meet the specifications they have described can still exhibit inconsistency and contamination, which tends to amplify failures. More work must be done so that the network deals with these potential problems.

- Inconsistency - A faulty process p which does not deliver a message that is delivered by all correct processes may have a state that is inconsistent with the other correct processes.
- Contamination - An inconsistent process p can still broadcasts messages, which all correct processes may deliver. If these messages reflect p's state, then all of these correct processes may get contaminated with p's bad state.

By contamination of correct processes is one mechanism by which one failure can be amplified. Unfortunately, generally, broadcasts amplify any errors that occur at a lower level (in one of the send/recv primitives of the network) too. In an Atomic Broadcast network, a receive omission can cause a process to deliver messages out of order. Even faulty processes that behave completely correctly before crashing can cause failures which will be amplified. By not passing on information about what order messages should be delivered in, a faulty process can in-effect, deliver a message out of order before it crashes. (see page 22).

Luckily, it is possible to design broadcasts that are not prone to inconsistency, or at least contamination of the errors. Prevent inconsistency requires a process to be able to detect if its about to make a message delivery error and stop itself first. This is difficult to do. Contamination is a little easier to deal with; correct processes do not deliver messages whose state is inconsistent with its own. These topics are not dealt with heavily in this paper, and instead, the authors refer to other papers about this topic(GT91,Gop92).

1.4 Broadcast Transformations

It is assumed in the transformations following that the broadcasts all satisfy uniform integrity, (all messages faulty processors deliver have been broadcast by someone and only delivers each message at most once.) All the transformations preserve uniform agreement in the original broadcast. It will be explicity stated when the algorithms preserve timeliness. Another property of the transformation to consider is if it is non-blocking, if the transformation does not block the immediate delivery of the messages received. These transformations are modular, making them easy to understand. These broadcasts assume that all processes know about each other a priori.

Processes can broadcast messages and deliver messages in the broadcast problem. When broadcasting a message, a process tags the message with its own unique ID.

1.4.1 Achieving Total Order

H and T describe a transformation from a Reliable, FIFO, or Causal Broadcast that satisfies Local-Time Δ-Timeliness its atomic counterpart. This transformation is blocking, meaning it can delay the delivery of messages it receives.

To atomically broadcast a message, the process just broadcasts the messages reliably and tags the message when the time according to its own local clock.

When the process is supposed to reliably delivers a message, it now schedules the message to be delivered Δ past the time the message was tagged delivered at. Δ is the guaranteed latency for the system. Since we assume that in the original reliable broadcast, correct processes delivered this message by time of the broadcast + Δ, then the process should be able to atomically deliver this message at the right time. Intuitively, the process just delays delivering messages as long as possible so that it can receive all other messages that may need to be delivered before this one. In the event that two messages are supposed to be delivered at the same time, ties are resolved according to some a priori agreement, like highest sender ID first.

Later we'll see that atomic broadcast reduces to consensus, but consensus is supposed to be unsolvable in asynchronous networks. For one, these processes each have their own clock and secondly, there are restrictions on how long it can take to deliver a message, so we get around this impossibility result.

1.4.2 Achieving FIFO Order

This algorithm transforms any reliable (not necessarily uniform) broadcast into a Uniform FIFO broadcast and preserves atomicity in the process. This algorithm is blocking also.

Each process p keeps two variables:

- msgSet, which is initially the empty set and is a set of all the messages the process cannot send yet because it is waiting to get previous messages before delivering them.
- next[s], which contains an integer for each other process, s, denoting one more than the number of messages p has FIFO-delivered that s broadcasted. Each element of the array is initially set to 1. Note that next has an element for itself, which is used to maintain an internal count of how many messages the process itself has broadcasted.

A process p FIFO broadcasts a message after the reliable broadcast, except now the process tags the message next[p], so that other processors know what how many messages were broadcasted from p before this one.

When the process is ready to reliably deliver the message, it first compares the number of the broadcast with the corresponding integer in next[] for the process that broadcasted the message. If these numbers are equal, the process delivers the message and increments the corresponding element in next[]. Otherwise, the process adds this message to msgSet where it can be delivered later. Whenever a process delivers a message, it must then check msgSet to see if there are any more messages in that set that can be delivered and delivers these messages.

1.4.3 Achieving Causal Order

There are two transformations for going from a Uniform FIFO broadcast to a Casual one, both which preserve atomicity. One transformation is blocking while the other is not. Here the blocking algorithm is covered first because it is more similar to the transformation just covered and thus easy to understand. The transformation to causal order requires a Uniform FIFO, but the previous transformation given also produces this.

Blocking Transformation
Each process p7 again keeps two variables:

- msgList, which is initially the empty set and is a set of all the messages the process cannot yet send (along with the messages corresponding numOfAllDlvrs' array) because it is waiting to deliver other messages before it can deliver these messages.
- numOfAllDlvrs[s], which contains an integer for each other process, s, which is the number of all messages p has causally delivered that s broadcasted. Each element is originally set to 0. Not that numOfAllDlvrs is the entire array.

To causally broadcast a message, a process FIFO broadcasts it, except now the process must also broadcast its entire numOfAllDlvrs along with the message, so that other processors know what messages it must deliver before it can deliver this one.

When the process is ready to FIFO deliver a message m, it instead compares its own numOfAllDlvrs array with the numOfAllDlvrs' of m. If every element of its own array is greater than or equal to m's array, then is has already delivered every message the broadcaster of m delivered before the broadcast of m, so it can now causally deliver m. If not, then m is added to msgList so it can be delivered later. Whenever the process delivers a message, it then checks msgList to see if there are any more messages it is ready to deliver and delivers any of these messages.

Non-blocking transformation
Each process p keeps an order list, rcntDlvrs, which is a list of all the messages this process has causally delivered since it last broadcasted a message. It also keeps track of all messages it has causally delivered.

To causally broadcast a message, a process FIFO broadcasts the message and rcntDlvrs. After broadcasting this message, it sets rcntDlvrs back to an empty list.

When the process is ready to FIFO deliver a message, it causally delivers any messages in rcntDlvrs that it has not previously delivers and then delivers the actual message contained in the broadcast.

Since we know that the process FIFO delivers the messages, we know that the process must have causally broadcasted the message before the current one. Therefore, the only messages p may not have delivered is those that the broadcaster has delivered between the broadcasts. RcntDlvrs makes sure that all the messages are properly delivered first, but this transformation comes with the negative side effect of potentially large messages due to the size of the rcntDlvrs.

1.5 Relation of Consensus and Atomic Broadcast

In the last lecture, the concept that Atomic Broadcast and consensus are related was introduced. This paper again goes over that, showing that the two are reducible to one another. The loophole again used is that there is some amount of synchrony in our system to use to solve consensus in an asynchronous system. Last lectures, failure detectors introduced some amount of synchrony to the system.

1.5.1 Transforming Atomic Broadcast to Consensus

This transformation takes an atomic broadcast to achieve consensus. To propose a value, a process atomically broadcasts the value it wishes to propose. Each process then decides the value of the first messages it delivers. Basically, Atomic Broadcast guarantees that all correct processes will deliver the messages in the same order, so they must all agree on what the first message to be delivered is and this is the value agreed on.

The paper does not try to achieve uniform consensus. I don't think this is possible using this transformation, but I'm not sure about this. Even if we started with uniform atomic broadcasts, which means that faulty processes must deliver messages in the same atomic order agree on by the correct processes, uniform consensus is still not achieve. The faulty process may not deliver any messages that are inconsistent with the atomic ordering, but it can still fail to send the first message agreed on. This faulty process would not halt, but instead decide the second message incorrectly.

1.5.2 Transforming Reliable Broadcast with Uniform Integrity and Consensus to Atomic Broadcast

Basically, this transformation consists of rounds of consensus decision and the delivery of batches of messages and requires only crash failures. Whenever a process wants to atomically broadcast a message, it reliably broadcasts that message. When a process reliably delivers a message, to atomically deliver it, it first adds the message to a list called R-delivered. The process also keeps A-delivered, which is a list of messages the process has A-delivered. If R-delivered - A-delivered is not null, then the process proposes all these atomically undelivered messages for a round of consensus, marked with k for the round. Once this value is decided on, then the process atomically delivers all the messages in the value that is decided in the order decided, adding all these messages to A-delivered.

In this case, I think Uniform Consensus is enough to get Uniform Atomic Broadcast.

1.6 Multicast Specifications

Multicast is a generalization of the reliable broadcast problem. In Multicast, there are groups that are static (possibly overlapping), where each process knows the membership of each group. (For more on who processes continue to know this information even when processes are joining and leaving, see Cri90a and RB91.) Multicast is when a process wants a message to be delivered to all members of specific groups, and only the processes in the group to which the message is to be delivered can deliver the message. This is not to say however, that other processes cannot send the message. A broadcast then is a multicast targeted exclusively to members of one's own group.

1.6.1 Reliable Multicast

- 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).

1.6.2 FIFO Multicast

Inherently, this requires that all messages multicasted from a particular process must all be delivered and in the order that they were multicasted. However, there are two flavors of FIFO Multicast, global and local. Global requires that all messages multicasted from the same process must be delivered in FIFO order, which local requires only messages multicasted from the same process to be delivered to the same group must be delivered in FIFO order. This means if a process gets two multicasts, one to group 1 which it is part of, and then to group 2 which it is a part of, then in global FIFO, the process must deliver the first message first. In local FIFO, these messages can be delivered in whatever order.

1.6.3 Causal Multicast

Causal Multicast also has Global and Local flavors as the FIFO multicast, but now the delivery properties are strengthened so that all messages delivered before the multicast must be delivered.

1.6.4 Atomic Multicast

Atomic Multicast is multicast with a total ordering. This has three types instead of two. Local requires that messages multicast to the same group be delivered in the same order, and this is the weakest guarantee. Pairwise atomic requires that if two correct processes both deliver m and m', then one will deliver m before m' iff the other delivers m before m'. Pairwise atomic also requires that all overlapping members of two messages to different groups will be delivered in the same order. However, there can still be cyclic relations in pairwise atomicity, which is why global atomic multicast is needed.

2 Bimodal Multicast Paper

This paper claims that actual implementation of Reliable Multicast are not scalable and have poor performance, while best-effort reliability overcomes these hurdles at the expense of having unpredictable message guarantees when the system is highly faulty. The algorithm they propose is claimed to be scalable and have predictable reliability. They call this algorithm either bimodal multicast or probabilistic broadcast.

The algorithm first uses an unreliable, hierarchal broadcast to make a best-effort attempt at delivering messages. Then the algorithm uses a second protocol that attempts to fix any mistakes made in the first part. To fix potential mistakes, gossip messages are exchanged to try to determine if processes have missed messages. If so, then these processes will send responses to the gossip messages to ask for the messages lost.