3. Reliable Broadcast

He said: "I could have been someone";
She replied: "So could anyone."
(The Pogues)

This chapter covers *broadcast communication* abstractions. These are used to disseminate information among a set of processes and differ according to the reliability of the dissemination. For instance, *best-effort broadcast* guarantees that all correct processes deliver the same set of messages if the senders are correct. Stronger forms of reliable broadcast guarantee this property even if the senders crash while broadcasting their messages. Even stronger broadcast abstractions are appropriate for the arbitrary-fault model and ensure consistency with Byzantine process abstractions.

We will consider several related abstractions for processes subject to crash faults: *best-effort broadcast, (regular) reliable broadcast, uniform reliable broadcast, stubborn broadcast, probabilistic broadcast*, and *causal broadcast*. For processes in the crash-recovery model, we describe *stubborn broadcast*, *logged best-effort broadcast*, and *logged uniform reliable broadcast*. Finally, for Byzantine processes, we introduce *Byzantine consistent broadcast* and *Byzantine reliable broadcast*. For each of these abstractions, we will provide one or more algorithms implementing it, and these will cover the different models addressed in this book.

3.1 Motivation

3.1.1 Client–Server Computing

In traditional distributed applications, interactions are often established between two processes. Probably the most representative of this sort of interaction is the now classic *client–server* scheme. According to this model, a *server* process exports an interface to several *clients*. Clients use the interface by sending a request to the server and by later collecting a reply. Such interaction is supported by *point-to-point* communication protocols. It is extremely useful for the application if such a protocol

C. Cachin et al., *Introduction to Reliable and Secure Distributed Programming*,
DOI: 10.1007/978-3-642-15260-3_3,

is *reliable*. Reliability in this context usually means that, under some assumptions (which are, by the way, often not completely understood by most system designers), messages exchanged between the two processes are not lost or duplicated, and are delivered in the order in which they were sent. Typical implementations of this abstraction are reliable transport protocols such as TCP on the Internet. By using a reliable point-to-point communication protocol, the application is free from dealing explicitly with issues such as acknowledgments, timeouts, message retransmissions, flow control, and a number of other issues that are encapsulated by the protocol interface.

3.1.2 Multiparticipant Systems

As distributed applications become bigger and more complex, interactions are no longer limited to bilateral relationships. There are many cases where more than two processes need to operate in a coordinated manner. Consider, for instance, a multiuser virtual environment where several users interact in a virtual space. These users may be located at different physical places, and they can either directly interact by exchanging multimedia information, or indirectly by modifying the environment.

It is convenient to rely here on *broadcast* abstractions. These allow a process to send a message within a *group* of processes, and make sure that the processes agree on the messages they deliver. A naive transposition of the reliability requirement from point-to-point protocols would require that no message sent to the group be lost or duplicated, i.e., the processes agree to deliver every message broadcast to them. However, the definition of agreement for a broadcast primitive is not a simple task. The existence of multiple senders and multiple recipients in a group introduces degrees of freedom that do not exist in point-to-point communication. Consider, for instance, the case where the sender of a message fails by crashing. It may happen that some recipients deliver the last message sent while others do not. This may lead to an inconsistent view of the system state by different group members. When the sender of a message exhibits arbitrary-faulty behavior, assuring that the recipients deliver one and the same message is an even bigger challenge.

The broadcast abstractions in this book provide a multitude of reliability guarantees. For crash-stop processes they range, roughly speaking, from *best-effort*, which only ensures delivery among all correct processes if the sender does not fail, through *reliable*, which, in addition, ensures *all-or-nothing* delivery semantics, even if the sender fails, to *totally ordered*, which furthermore ensures that the delivery of messages follow the same global order, and *terminating*, which ensures that the processes either deliver a message or are eventually aware that they should never deliver the message.

For arbitrary-faulty processes, a similar range of broadcast abstractions exists. The simplest one among them guarantees a form of *consistency*, which is not even an issue for crash-stop processes, namely, to ensure that two correct processes, if they deliver a messages at all, deliver the same message. The reliable broadcast abstractions and total-order broadcast abstractions among arbitrary-faulty processes

additionally provide all-or-nothing delivery semantics and totally ordered delivery, respectively.

In this chapter, we will focus on best-effort and reliable broadcast abstractions. Stronger forms of broadcast will be considered in later chapters. The next three sections present broadcast abstractions with crash-stop process abstractions. More general process failures are considered afterward.

3.2 Best-Effort Broadcast

A broadcast abstraction enables a process to send a message, in a one-shot operation, to all processes in a system, including itself. We give here the specification and an algorithm for a broadcast communication primitive with a weak form of reliability, called *best-effort broadcast*.

3.2.1 Specification

With best-effort broadcast, the burden of ensuring reliability is only on the sender. Therefore, the remaining processes do not have to be concerned with enforcing the reliability of received messages. On the other hand, no delivery guarantees are offered in case the sender fails. Best-effort broadcast is characterized by the following three properties depicted in Module 3.1: *validity* is a liveness property, whereas the *no duplication* property and the *no creation* property are safety properties. They descend directly from the corresponding properties of perfect point-to-point links. Note that broadcast messages are implicitly addressed to all processes. Remember also that messages are unique, that is, no process ever broadcasts the same message twice and furthermore, no two processes ever broadcast the same message.

Module 3.1: Interface and properties of best-effort broadcast

Module:

Name: BestEffortBroadcast, **instance** *beb*.

Events:

Request: $\langle$ *beb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *beb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

BEB1: *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.

BEB2: *No duplication:* No message is delivered more than once.

BEB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

Algorithm 3.1: Basic Broadcast

Implements:
BestEffortBroadcast, **instance** *beb*.

Uses:
PerfectPointToPointLinks, **instance** *pl*.

upon event $\langle$ *beb*, *Broadcast* | m $\rangle$ **do**
forall $q \in \Pi$ **do**
trigger $\langle$ *pl*, *Send* | q, m $\rangle$;

upon event $\langle$ *pl*, *Deliver* | p, m $\rangle$ **do**
trigger $\langle$ *beb*, *Deliver* | p, m $\rangle$;

3.2.2 Fail-Silent Algorithm: Basic Broadcast

We provide here algorithm "Basic Broadcast" (Algorithm 3.1) that implements best-effort broadcast using perfect links. This algorithm does not make any assumption on failure detection: it is a fail-silent algorithm. The algorithm is straightforward. Broadcasting a message simply consists of sending the message to every process in the system using perfect point-to-point links, as illustrated by Fig. 3.1 (in the figure, white arrowheads represent request/indication events at the module interface and black arrowheads represent message exchanges). The algorithm works because the properties of perfect links ensure that all correct processes eventually deliver the message, as long as the sender of a message does not crash.

Correctness. The properties of best-effort broadcast are trivially derived from the properties of the underlying perfect point-to-point links. The *no creation* property follows directly from the corresponding property of perfect links. The same applies to *no duplication*, which relies in addition on the assumption that messages broadcast by different processes are unique. *Validity* is derived from the *reliable delivery* property and the fact that the sender sends the message to every other process in the system.

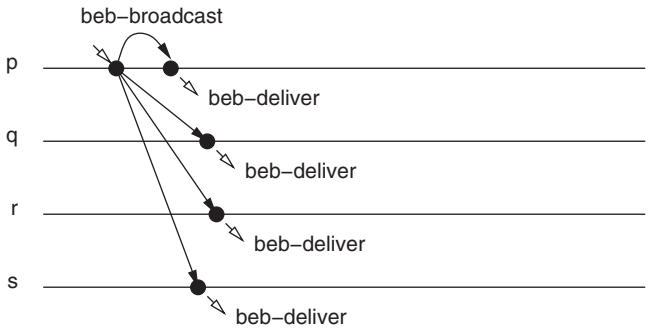


Figure 3.1: Sample execution of basic broadcast

Performance. For every message that is broadcast, the algorithm requires a single communication step and exchanges $O(N)$ messages.

3.3 Regular Reliable Broadcast

Best-effort broadcast ensures the delivery of messages as long as the sender does not fail. If the sender fails, some processes might deliver the message and others might not deliver it. In other words, they do not *agree* on the delivery of the message. Actually, even if the process sends a message to all processes before crashing, the delivery is not ensured because perfect links do not enforce the delivery when the sender fails. Ensuring agreement even when the sender fails is an important property for many practical applications that rely on broadcast. The abstraction of *(regular) reliable broadcast* provides exactly this stronger notion of reliability.

3.3.1 Specification

Intuitively, the semantics of a reliable broadcast algorithm ensure that the correct processes agree on the set of messages they deliver, even when the senders of these messages crash during the transmission. It should be noted that a sender may crash before being able to transmit the message, in which case no process will deliver it. The specification of reliable broadcast in Module 3.2 extends the properties of the best-effort broadcast abstraction (Module 3.1) with a new liveness property called *agreement*. The other properties remain unchanged (but are repeated here for completeness). The very fact that *agreement* is a liveness property might seem

Module 3.2: Interface and properties of (regular) reliable broadcast

Module:

Name: ReliableBroadcast, **instance** *rb*.

Events:

Request: $\langle$ *rb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *rb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

RB1: *Validity:* If a correct process p broadcasts a message m, then p eventually delivers m.

RB2: *No duplication:* No message is delivered more than once.

RB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

RB4: *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

counterintuitive, as the property can be achieved by not having any process ever deliver any message. Strictly speaking, it is, however, a liveness property as it can always be ensured in extensions of finite executions. We will see other forms of *agreement* that are safety properties later in the book.

3.3.2 Fail-Stop Algorithm: Lazy Reliable Broadcast

We now show how to implement regular reliable broadcast in a fail-stop model. In our algorithm, depicted in Algorithm 3.2, which we have called "Lazy Reliable Broadcast," we make use of the best-effort broadcast abstraction described in the previous section, as well as the perfect failure detector abstraction $\mathcal{P}$ introduced earlier.

Algorithm 3.2: Lazy Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEffortBroadcast, **instance** *beb*;
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *rb*, *Init* $\rangle$ **do**
correct := Π;
from[p] := $[\emptyset]^N$;

upon event $\langle$ *rb*, *Broadcast* | m $\rangle$ **do**
trigger $\langle$ *beb*, *Broadcast* | [DATA, *self*, m] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [DATA, s, m] $\rangle$ **do**
if $m \notin$ *from*[s] **then**
trigger $\langle$ *rb*, *Deliver* | s, m $\rangle$;
from[s] := *from*[s] $\cup \{m\}$;
if $s \notin$ *correct* **then**
trigger $\langle$ *beb*, *Broadcast* | [DATA, s, m] $\rangle$;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
correct := *correct* $\setminus \{p\}$;
forall $m \in$ *from*[p] **do**
trigger $\langle$ *beb*, *Broadcast* | [DATA, p, m] $\rangle$;

To *rb*-broadcast a message, a process uses the best-effort broadcast primitive to disseminate the message to all. The algorithm adds some implementation-specific parameters to the exchanged messages. In particular, its adds a message descriptor (DATA) and the original source of the message (process s) in the *message header*. The result is denoted by [DATA, s, m] in the algorithm. A process that receives the message (when it *beb*-delivers the message) strips off the message header and *rb*-delivers it immediately. If the sender does not crash, then the message will be

rb-delivered by all correct processes. The problem is that the sender might crash. In this case, the process that delivers the message from some other process detects that crash and relays the message to all others. We note that this is a language abuse: in fact, the process relays a copy of the message (and not the message itself).

At the same time, the process also maintains a variable *correct*, denoting the set of processes that have not been detected to crash by $\mathcal{P}$. Our algorithm is said to be *lazy* in the sense that it retransmits a message only if the original sender has been detected to have crashed. The variable *from* is an array of sets, indexed by the processes in Π, in which every entry s contains the messages from sender s that have been *rb*-delivered.

It is important to notice that, strictly speaking, two kinds of events can force a process to retransmit a message. First, when the process detects the crash of the source, and, second, when the process *beb*-delivers a message and realizes that the source has already been detected to have crashed (i.e., the source is not anymore in *correct*). This might lead to duplicate retransmissions when a process *beb*-delivers a message from a source that fails, as we explain later. It is easy to see that a process that detects the crash of a source needs to retransmit the messages that have already been *beb*-delivered from that source. On the other hand, a process might *beb*-deliver a message from a source after it detected the crash of that source: it is, thus, necessary to check for the retransmission even when no new crash is detected.

Correctness. The *no creation* (respectively *validity*) property of our reliable broadcast algorithm follows from the *no creation* (respectively *validity*) property of the underlying best-effort broadcast primitive. The *no duplication* property of reliable broadcast follows from our use of a variable *from* that keeps track of the messages that have been *rb*-delivered at every process and from the assumption of unique messages across all senders. *Agreement* follows here from the *validity* property of the underlying best-effort broadcast primitive, from the fact that every process relays every message that it *rb*-delivers when it detects the sender, and from the use of a perfect failure detector.

Performance. If the initial sender does not crash then the algorithm requires a single communication step and $O(N)$ messages to *rb*-deliver a message to all processes. Otherwise, it may take $O(N)$ steps and $O(N^2)$ messages in the worst case (if the processes crash in sequence).

3.3.3 Fail-Silent Algorithm: Eager Reliable Broadcast

In the "Lazy Reliable Broadcast" algorithm (Algorithm 3.2), when the *accuracy* property of the failure detector is not satisfied, the processes might relay messages unnecessarily. This wastes resources but does not impact correctness. On the other hand, we rely on the *completeness* property of the failure detector to ensure the broadcast *agreement*. If the failure detector does not ensure *completeness* then the processes might omit to relay messages that they should be relaying (e.g., messages broadcast by processes that crashed), and hence might violate *agreement*.

Algorithm 3.3: Eager Reliable Broadcast

Implements:
ReliableBroadcast, **instance** *rb*.

Uses:
BestEffortBroadcast, **instance** *beb*.

upon event ⟨ *rb*, *Init* ⟩ **do**
delivered := ∅;

upon event ⟨ *rb*, *Broadcast* | m ⟩ **do**
trigger ⟨ *beb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *beb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
if $m \notin$ *delivered* **then**
delivered := *delivered* ∪ $\{m\}$;
trigger ⟨ *rb*, *Deliver* | s, m ⟩;
trigger ⟨ *beb*, *Broadcast* | [DATA, s, m] ⟩;

In fact, we can circumvent the need for a failure detector (i.e., the need for its *completeness* property) by adopting an *eager* scheme: every process that gets a message relays it immediately. That is, we consider the worst case, where the sender process might have crashed, and we relay every message. This relaying phase is exactly what guarantees the *agreement* property of reliable broadcast. The resulting algorithm (Algorithm 3.3) is called "Eager Reliable Broadcast."

The algorithm assumes a fail-silent model and does not use any failure detector: it relies only on the best-effort broadcast primitive described in Sect. 3.2. In Fig. 3.2, we illustrate how the algorithm ensures *agreement* even if the sender crashes: process p crashes and its message is not *beb*-delivered by processes r and by s. However, as process q retransmits the message, i.e., *beb*-broadcasts it, the remaining processes also *beb*-deliver it and subsequently *rb*-deliver it. In our "Lazy

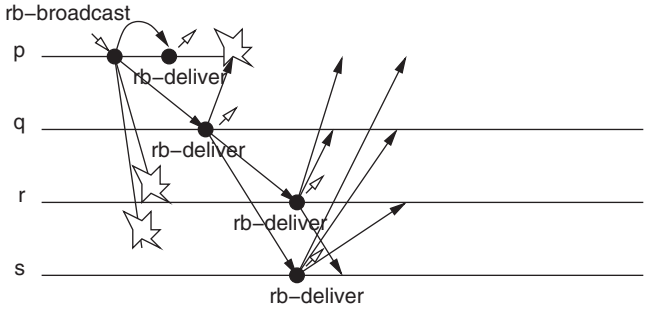


Figure 3.2: Sample execution of reliable broadcast with faulty sender

Reliable Broadcast" algorithm, process q will be relaying the message only after it has detected the crash of p.

Correctness. All properties, except *agreement*, are ensured as in the "Lazy Reliable Broadcast." The *agreement* property follows from the *validity* property of the underlying best-effort broadcast primitive and from the fact that every correct process immediately relays every message it *rb*-delivers.

Performance. In the best case, the algorithm requires a single communication step and $O(N^2)$ messages to *rb*-deliver a message to all processes. In the worst case, should the processes crash in sequence, the algorithm may incur $O(N)$ steps and $O(N^2)$ messages.

3.4 Uniform Reliable Broadcast

With regular reliable broadcast, the semantics just require the *correct* processes to deliver the same set of messages, regardless of what messages have been delivered by faulty processes. In particular, a process that *rb*-broadcasts a message might *rb*-deliver it and then crash, before the best-effort broadcast abstraction can even *beb*-deliver the message to any other process. Indeed, this scenario may occur in both reliable broadcast algorithms that we presented (eager and lazy). It is thus possible that no other process, including correct ones, ever *rb*-delivers that message. There are cases where such behavior causes problems because even a process that *rb*-delivers a message and later crashes may bring the application into a inconsistent state.

We now introduce a stronger definition of reliable broadcast, called *uniform reliable broadcast*. This definition is stronger in the sense that it guarantees that the set of messages delivered by *faulty* processes is always a *subset* of the messages delivered by correct processes. Many other abstractions also have such *uniform* variants.

3.4.1 Specification

Uniform reliable broadcast differs from reliable broadcast by the formulation of its *agreement* property. The specification is given in Module 3.3.

Uniformity is typically important if the processes interact with the external world, e.g., print something on a screen, authorize the delivery of money through a bank machine, or trigger the launch of a rocket. In this case, the fact that a process has delivered a message is important, even if the process has crashed afterward. This is because the process, before crashing, could have communicated with the external world after having delivered the message. The processes that did not crash should also be aware of that message having been delivered, and of the possible external action having been performed.

Figure 3.3 depicts an execution of a reliable broadcast algorithm that is not uniform. Both processes p and q *rb*-deliver the message as soon as they *beb*-deliver

Module 3.3: Interface and properties of uniform reliable broadcast

Module:

Name: UniformReliableBroadcast, **instance** *urb*.

Events:

Request: $\langle$ *urb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *urb*, *Deliver* | p, m $\rangle$: Delivers a message m broadcast by process p.

Properties:

URB1–URB3: Same as properties RB1–RB3 in (regular) reliable broadcast (Module 3.2).

URB4: *Uniform agreement:* If a message m is delivered by some process (whether correct or faulty), then m is eventually delivered by every correct process.

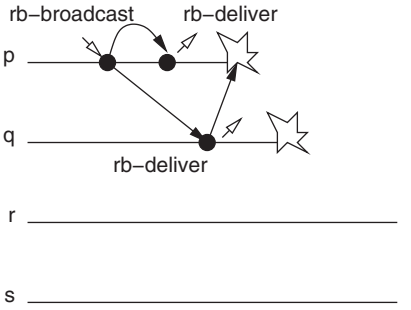


Figure 3.3: Nonuniform reliable broadcast

it, but crash before they are able to relay the message to the remaining processes. Still, processes r and s are consistent among themselves (neither has *rb*-delivered the message).

3.4.2 Fail-Stop Algorithm: All-Ack Uniform Reliable Broadcast

Basically, our "Lazy Reliable Broadcast" and "Eager Reliable Broadcast" algorithms do not ensure *uniform agreement* because a process may *rb*-deliver a message and then crash. Even if this process has relayed its message to all processes (through a best-effort broadcast primitive), the message might not reach any of the remaining processes. Note that even if we considered the same algorithms and replaced the best-effort broadcast abstraction with a reliable broadcast one, we would still not implement a uniform broadcast abstraction. This is because a process may deliver a message before relaying it to all processes.

Algorithm 3.4, named "All-Ack Uniform Reliable Broadcast," implements uniform reliable broadcast in the fail-stop model. Basically, in this algorithm, a process

Algorithm 3.4: All-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEffortBroadcast, **instance** *beb*.
PerfectFailureDetector, **instance** $\mathcal{P}$.

upon event $\langle$ *urb*, *Init* $\rangle$ **do**
delivered := $\emptyset$;
pending := $\emptyset$;
correct := Π;
forall m **do** *ack*$[m]$:= $\emptyset$;

upon event $\langle$ *urb*, *Broadcast* | m $\rangle$ **do**
pending := *pending* $\cup$ $\{(self, m)\}$;
trigger $\langle$ *beb*, *Broadcast* | [DATA, *self*, m] $\rangle$;

upon event $\langle$ *beb*, *Deliver* | p, [DATA, s, m] $\rangle$ **do**
ack$[m]$:= *ack*$[m]$ $\cup$ $\{p\}$;
if $(s, m) \notin$ *pending* **then**
pending := *pending* $\cup$ $\{(s, m)\}$;
trigger $\langle$ *beb*, *Broadcast* | [DATA, s, m] $\rangle$;

upon event $\langle$ $\mathcal{P}$, *Crash* | p $\rangle$ **do**
correct := *correct* $\setminus$ $\{p\}$;

function *candeliver*(m) **returns** Boolean **is**
return $(correct \subseteq ack[m])$;

upon exists $(s, m) \in$ *pending* such that *candeliver*$(m) \wedge m \notin$ *delivered* **do**
delivered := *delivered* $\cup$ $\{m\}$;
trigger $\langle$ *urb*, *Deliver* | s, m $\rangle$;

delivers a message only when it knows that the message has been *beb*-delivered and thereby *seen* by all correct processes. All processes relay the message once, after they have seen it. Each process keeps a record of processes from which it has already received a message (either because the process originally sent the message or because the process relayed it). When all correct processes have retransmitted the message, all correct processes are guaranteed to deliver the message, as illustrated in Fig. 3.4.

The algorithm uses a variable *delivered* for filtering out duplicate messages and a variable *pending*, used to collect the messages that have been *beb*-delivered and seen, but that still need to be *urb*-delivered.

The algorithm also uses an array *ack* with sets of processes, indexed by all possible messages. The entry *ack*$[m]$ gathers the set of processes that the process knows have seen m. Of course, the array can be implemented with a finite amount of

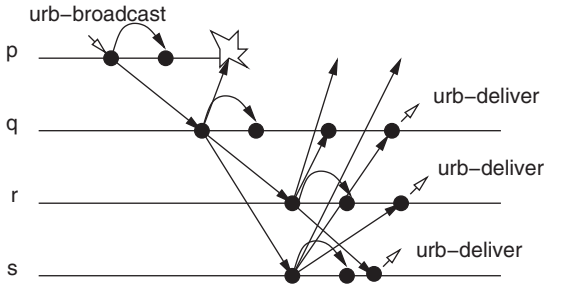


Figure 3.4: Sample execution of all-ack uniform reliable broadcast

memory by using a sparse representation. Note that the last *upon* statement of the algorithm is triggered by an internal event defined on the state of the algorithm.

Correctness. The *validity* property follows from the *completeness* property of the failure detector and from the *validity* property of the underlying best-effort broadcast. The *no duplication* property relies on the *delivered* variable to filter out duplicates. *No creation* is derived from the *no creation* property of the underlying best-effort broadcast. *Uniform agreement* is ensured by having each process wait to *urb*-deliver a message until all correct processes have seen and relayed the message. This mechanism relies on the *accuracy* property of the perfect failure detector.

Performance. When considering the number of communication steps, in the best case, the algorithm requires two communication steps to *urb*-deliver a message to all processes. In such scenario, in the first step it sends N messages and in the second step $N(N-1)$ messages, for a total of N^2 messages. In the worst case, if the processes crash in sequence, $N+1$ steps are required. Therefore, uniform reliable broadcast requires one step more to deliver a message than its regular counterpart.

3.4.3 Fail-Silent Algorithm: Majority-Ack Uniform Reliable Broadcast

The "All-Ack Uniform Reliable Broadcast" algorithm of Sect. 3.4.2 (Algorithm 3.4) is not correct if the failure detector is not perfect. *Uniform agreement* would be violated if *accuracy* is not satisfied and *validity* would be violated if *completeness* is not satisfied.

We now give a uniform reliable broadcast algorithm that does not rely on a perfect failure detector but assumes a majority of correct processes, i.e., $N > 2f$ if we assume that up to f processes may crash. We leave it as an exercise to show why the majority assumption is needed in the fail-silent model, without any failure detector. Algorithm 3.5, called "Majority-Ack Uniform Reliable Broadcast," is similar to Algorithm 3.4 ("All-Ack Uniform Reliable Broadcast") in the fail-silent model, except that processes do not wait until all correct processes have seen a message, but only until a majority quorum has seen and retransmitted the message. Hence, the algorithm can be obtained by a small modification from the previous one, affecting only the condition under which a message is delivered.

Algorithm 3.5: Majority-Ack Uniform Reliable Broadcast

Implements:
UniformReliableBroadcast, **instance** *urb*.

Uses:
BestEffortBroadcast, **instance** *beb*.

// Except for the function *candeliver*$(\cdot)$ below and for the absence of $\langle$ *Crash* $\rangle$ events
// triggered by the perfect failure detector, it is the same as Algorithm 3.4.

function *candeliver*(m) **returns** Boolean **is**
return $\#(ack[m]) > N/2$;

Correctness. The algorithm provides uniform reliable broadcast if $N > 2f$. The *no duplication* property follows directly from the use of the variable *delivered*. The *no creation* property follows from the *no creation* property of best-effort broadcast.

To argue for the *uniform agreement* and *validity* properties, we first observe that if a correct process p *beb*-delivers some message m then p eventually *urb*-delivers m. Indeed, if p is correct, and given that p *beb*-broadcasts m according to the algorithm, then every correct process *beb*-delivers and hence *beb*-broadcasts m. As we assume a majority of the processes to be correct, p eventually *beb*-delivers m from more than $N/2$ processes and *urb*-delivers it.

Consider now the *validity* property. If a correct process p *urb*-broadcasts a message m then p *beb*-broadcasts m, and hence p *beb*-delivers m eventually; according to the above observation, p eventually also *urb*-delivers m. Consider now *uniform agreement*, and let q be any process that *urb*-delivers m. To do so, q must have *beb*-delivered m from a majority of the processes. Because of the assumption of a correct majority, at least one correct process must have *beb*-broadcast m. Hence, all correct processes eventually *beb*-deliver m by the *validity* property of best-effort broadcast, which implies that all correct processes also *urb*-deliver m eventually according to the observation made earlier.

Performance. The performance of the algorithm is similar to the performance of the "All-Ack Uniform Reliable Broadcast" algorithm.

3.5 Stubborn Broadcast

This section presents a *stubborn broadcast* abstraction that works with crash-stop process abstractions in the fail-silent system model, as well as with crash-recovery process abstractions in the fail-recovery model.

3.5.1 Specification

The stubborn broadcast abstraction hides a retransmission mechanism and delivers every message that is broadcast by a correct process an infinite number of times,

Module 3.4: Interface and properties of stubborn best-effort broadcast

Module:

Name: StubbornBestEffortBroadcast, **instance** *sbeb*.

Events:

Request: ⟨ *sbeb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *sbeb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

SBEB1: *Best-effort validity:* If a process that never crashes broadcasts a message m, then every correct process delivers m an infinite number of times.

SBEB2: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

similar to its point-to-point communication counterpart. The specification of *best-effort stubborn broadcast* is given in Module 3.4. The key difference to the best-effort broadcast abstraction (Module 3.1) defined for fail-no-recovery settings lies in the stubborn and perpetual delivery of every message broadcast by a process that does not crash. As a direct consequence, the *no duplication* property of best-effort broadcast is not ensured.

Stubborn broadcast is the first broadcast abstraction in the fail-recovery model considered in this chapter (more will be introduced in the next two sections). As the discussion of logged perfect links in Chap. 2 has shown, communication abstractions in the fail-recovery model usually rely on logging their output to variables in stable storage. For stubborn broadcast, however, logging is not necessary because every delivered message is delivered infinitely often; no process that crashes and recovers finitely many times can, therefore, miss such a message.

The very fact that processes now have to deal with multiple deliveries is the price to pay for saving expensive logging operations. We discuss a *logged best-effort broadcast* in the next section, which eliminates multiple deliveries, but adds at the cost of logging the messages.

The stubborn best-effort broadcast abstraction also serves as an example for stronger stubborn broadcast abstractions, implementing reliable and uniform reliable stubborn broadcast variants, for instance. These could be defined and implemented accordingly.

3.5.2 Fail-Recovery Algorithm: Basic Stubborn Broadcast

Algorithm 3.6 implements stubborn best-effort broadcast using underlying stubborn communication links.

Correctness. The properties of stubborn broadcast are derived directly from the properties of the stubborn links abstraction used by the algorithm. In particular,

Algorithm 3.6: Basic Stubborn Broadcast

Implements:
StubbornBestEffortBroadcast, **instance** *sbeb*.

Uses:
StubbornPointToPointLinks, **instance** *sl*.

upon event ⟨ *sbeb*, *Recovery* ⟩ **do**
// do nothing

upon event ⟨ *sbeb*, *Broadcast* | m ⟩ **do**
forall $q \in \Pi$ **do**
trigger ⟨ *sl*, *Send* | q, m ⟩;

upon event ⟨ *sl*, *Deliver* | p, m ⟩ **do**
trigger ⟨ *sbeb*, *Deliver* | p, m ⟩;

validity follows from the fact that the sender sends the message to every process in the system.

Performance. The algorithm requires a single communication step for a process to deliver a message, and exchanges at least N messages. Of course, the stubborn links may retransmit the same message several times and, in practice, an optimization mechanism is needed to acknowledge the messages and stop the retransmission.

3.6 Logged Best-Effort Broadcast

This section and the next one consider broadcast abstractions in the *fail-recovery model* that rely on logging. We first discuss how fail-recovery broadcast algorithms use stable storage for logging and then present a best-effort broadcast abstraction and its implementation.

3.6.1 Overview

Most broadcast specifications we have considered for the fail-stop and fail-silent models are not adequate for the fail-recovery model. As explained next, even the strongest one of our specifications, uniform reliable broadcast, does not provide useful semantics in a setting where processes that crash can later recover and participate in the computation.

For instance, suppose a message m is broadcast by some process p. Consider another process q, which should eventually deliver m. But q crashes at some instant, recovers, and never crashes again; in the fail-recovery model, q is a *correct* process. For a broadcast abstraction, however, it might happen that process q delivers m and crashes immediately afterward, without having processed m, that is, before the application had time to react to the delivery of m. When the process recovers later, it

Module 3.5: Interface and properties of logged best-effort broadcast

Module:

Name: LoggedBestEffortBroadcast, **instance** *lbeb*.

Events:

Request: ⟨ *lbeb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *lbeb*, *Deliver* | *delivered* ⟩: Notifies the upper layer of potential updates to variable *delivered* in stable storage (which log-delivers messages according to the text).

Properties:

LBEB1: *Validity:* If a process that never crashes broadcasts a message m, then every correct process eventually log-delivers m.

LBEB2: *No duplication:* No message is log-delivered more than once.

LBEB3: *No creation:* If a process log-delivers a message m with sender s, then m was previously broadcast by process s.

has no memory of m, because the delivery of m occurred asynchronously and could not be anticipated. There should be some way for process q to find out about m upon recovery, and for the application to react to the delivery of m. We have already encountered this problem with the definition of logged perfect links in Sect. 2.4.5.

We adopt the same solution as for logged perfect links: the module maintains a variable *delivered* in stable storage, stores every delivered messages in the variable, and the higher-level modules retrieve the variable from stable storage to determine the delivered messages. To notify the layer above about the delivery, the broadcast abstraction triggers an event ⟨ *Deliver* | *delivered* ⟩. We say that a message m is *log-delivered* from sender s whenever an event ⟨ *Deliver* | *delivered* ⟩ occurs such that *delivered* contains a pair (s, m) for the first time. With this implementation, a process that log-delivers a message m, subsequently crashes, and recovers again will still be able to retrieve m from stable storage and to react to m.

3.6.2 Specification

The abstraction we consider here is called *logged best-effort broadcast* to emphasize that it log-delivers messages by "logging" them to local stable storage. Its specification is given in Module 3.5. The logged best-effort broadcast abstraction has the same interface and properties as best-effort broadcast with crash-stop faults (Module 3.1), except that messages are log-delivered instead of delivered. As we discuss later, stronger logged broadcast abstractions (regular and uniform) can be designed and implemented on top of logged best-effort broadcast.

Algorithm 3.7: Logged Basic Broadcast

Implements:
LoggedBestEffortBroadcast, **instance** *lbeb*.

Uses:
StubbornPointToPointLinks, **instance** *sl*.

upon event ⟨ *lbeb*, *Init* ⟩ **do**
delivered := ∅;
store(*delivered*);

upon event ⟨ *lbeb*, *Recovery* ⟩ **do**
retrieve(*delivered*);
trigger ⟨ *lbeb*, *Deliver* | *delivered* ⟩;

upon event ⟨ *lbeb*, *Broadcast* | m ⟩ **do**
forall $q \in \Pi$ **do**
trigger ⟨ *sl*, *Send* | q, m ⟩;

upon event ⟨ *sl*, *Deliver* | p, m ⟩ **do**
if $(p, m) \notin$ *delivered* **then**
delivered := *delivered* ∪ $\{(p, m)\}$;
store(*delivered*);
trigger ⟨ *lbeb*, *Deliver* | *delivered* ⟩;

3.6.3 Fail-Recovery Algorithm: Logged Basic Broadcast

Algorithm 3.7, called "Logged Basic Broadcast," implements logged best-effort broadcast. Its structure is similar to Algorithm 3.1 ("Basic Broadcast"). The main differences are the following:

1. The "Logged Basic Broadcast" algorithm uses stubborn best-effort links between every pair of processes for communication. They ensure that every message that is sent by a process that does not crash to a correct recipient will be delivered by its recipient an infinite number of times.
2. The "Logged Basic Broadcast" algorithm maintains a log of all delivered messages. When a new message is received for the first time, it is added to the log, and the upper layer is notified that the log has changed. If the process crashes and later recovers, the upper layer is also notified (as it may have missed a notification triggered just before the crash).

Correctness. The *no creation* property is derived from that of the underlying stubborn links, whereas *no duplication* is derived from the fact that the delivery log is checked before delivering new messages. The *validity* property follows from the fact that the sender sends the message to every other process in the system.

Performance. The algorithm requires a single communication step for a process to deliver a message, and exchanges at least N messages. Of course, stubborn links

may retransmit the same message several times and, in practice, an optimization mechanism is needed to acknowledge the messages and stop the retransmission. Additionally, the algorithm requires a log operation for each delivered message.

3.7 Logged Uniform Reliable Broadcast

In a manner similar to the crash-no-recovery case, it is possible to define both reliable and uniform variants of best-effort broadcast for the fail-recovery setting.

3.7.1 Specification

Module 3.6 defines a *logged uniform reliable broadcast* abstraction, which is appropriate for the fail-recovery model. In this variant, if a process (either correct or not) log-delivers a message (that is, stores the variable *delivered* containing the message in stable storage), all correct processes should eventually log-deliver that message. The interface is similar to that of logged best-effort broadcast and its properties directly correspond to those of uniform reliable broadcast with crash-stop processes (Module 3.3).

Module 3.6: Interface and properties of logged uniform reliable broadcast

---

Module:

Name: LoggedUniformReliableBroadcast, **instance** *lurb*.

Events:

Request: $\langle$ *lurb*, *Broadcast* | m $\rangle$: Broadcasts a message m to all processes.

Indication: $\langle$ *lurb*, *Deliver* | *delivered* $\rangle$: Notifies the upper layer of potential updates to variable *delivered* in stable storage (which log-delivers messages according to the text).

Properties:

LURB1–LURB3: Same as properties LBEB1–LBEB3 in logged best-effort broadcast (Module 3.5).

LURB4: *Uniform agreement:* If a message m is log-delivered by some process (whether correct or faulty), then m is eventually log-delivered by every correct process.

3.7.2 Fail-Recovery Algorithm: Logged Majority-Ack Uniform Reliable Broadcast

Algorithm 3.8, called "Logged Majority-Ack Uniform Reliable Broadcast," implements logged uniform broadcast, assuming that a majority of the processes

Algorithm 3.8: Logged Majority-Ack Uniform Reliable Broadcast

Implements:
LoggedUniformReliableBroadcast, **instance** *lurb*.

Uses:
StubbornBestEffortBroadcast, **instance** *sbeb*.

upon event ⟨ *lurb*, *Init* ⟩ **do**
delivered := ∅;
pending := ∅;
forall m **do** *ack*$[m]$:= ∅;
store(*pending*, *delivered*);

upon event ⟨ *lurb*, *Recovery* ⟩ **do**
retrieve(*pending*, *delivered*);
trigger ⟨ *lurb*, *Deliver* | *delivered* ⟩;
forall $(s, m) \in$ *pending* **do**
trigger ⟨ *sbeb*, *Broadcast* | [DATA, s, m] ⟩;

upon event ⟨ *lurb*, *Broadcast* | m ⟩ **do**
pending := *pending* ∪ {(*self*, m)};
store(*pending*);
trigger ⟨ *sbeb*, *Broadcast* | [DATA, *self*, m] ⟩;

upon event ⟨ *sbeb*, *Deliver* | p, [DATA, s, m] ⟩ **do**
if $(s, m) \notin$ *pending* **then**
pending := *pending* ∪ {(s, m)};
store(*pending*);
trigger ⟨ *sbeb*, *Broadcast* | [DATA, s, m] ⟩;
if $p \notin$ *ack*$[m]$ **then**
ack$[m]$:= *ack*$[m]$ ∪ {p};
if $\#$(*ack*$[m]$) $> N/2 \wedge (s, m) \notin$ *delivered* **then**
delivered := *delivered* ∪ {(s, m)};
store(*delivered*);
trigger ⟨ *lurb*, *Deliver* | *delivered* ⟩;

is correct. It log-delivers a message m from sender s by adding (s, m) to the *delivered* variable in stable storage. Apart from *delivered*, the algorithm uses two other variables, a set *pending* and an array *ack*, with the same functions as in "All-Ack Uniform Reliable Broadcast" (Algorithm 3.4). Variable *pending* denotes the messages that the process has seen but not yet *lurb*-delivered, and is logged. Variable *ack* is not logged because it will be reconstructed upon recovery. When a message has been retransmitted by a majority of the processes, it is log-delivered. Together with the assumption of a correct majority, this ensures that at least one correct process has logged the message, and this will ensure the retransmission to all correct processes.

Correctness. Consider the *agreement* property and assume that some correct process p log-delivers a message m. To do so, a majority of the processes must have

retransmitted the message. As we assume a majority of the processes is correct, at least one correct process must have logged the message (in its variable *pending*). This process will ensure that the message is eventually *sbeb*-broadcast to all correct processes; all correct processes will hence *sbeb*-deliver the message and acknowledge it. Hence, every correct process will log-deliver m. To establish the *validity* property, assume some process p *lurb*-broadcasts a message m and does not crash. Eventually, the message will be seen by all correct processes. As a majority of processes is correct, these processes will retransmit the message and p will eventually *lurb*-deliver m. The *no duplication* property is trivially ensured by the definition of log-delivery (the check that $(s, m) \notin$ *delivered* before adding (s, m) to *delivered* only serves to avoid unnecessary work). The *no creation* property is ensured by the underlying links.

Performance. Suppose that some process *lurb*-broadcasts a message m. All correct processes log-deliver m after two communication steps and two causally related logging operations (the variable *pending* can be logged in parallel to broadcasting the DATA message).

3.8 Probabilistic Broadcast

This section considers randomized broadcast algorithms, whose behavior is partially determined by a controlled random experiment. These algorithms do not provide deterministic broadcast guarantees but, instead, only make *probabilistic* claims about such guarantees.

Of course, this approach can only be used for applications that do not require full reliability. On the other hand, full reliability often induces a cost that is too high, especially for large-scale systems or systems exposed to attacks. As we will see, it is often possible to build scalable probabilistic algorithms that exploit randomization and provide good reliability guarantees.

Moreover, the abstractions considered in this book can almost never be mapped to physical systems in real deployments that match the model completely; some uncertainty often remains. A system designer must also take into account a small probability that the deployment fails due to such a mismatch. Even if the probabilistic guarantees of an abstraction leave room for error, the designer might accept this error because other sources of failure are more significant.

3.8.1 The Scalability of Reliable Broadcast

As we have seen throughout this chapter, in order to ensure the reliability of broadcast in the presence of faulty processes (and/or links with omission failures), a process needs to *send messages* to all other processes and needs to collect some form of *acknowledgment*. However, given limited bandwidth, memory, and processor resources, there will always be a limit to the number of messages that each process can send and to the acknowledgments it is able to collect in due time. If the

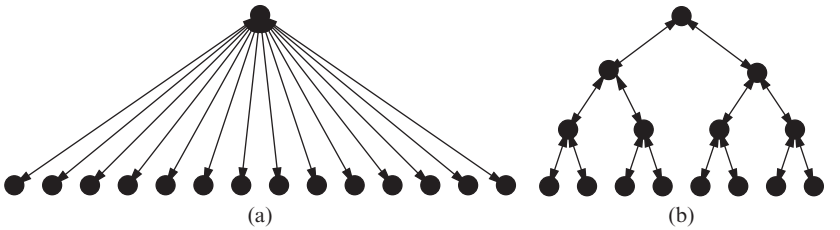


Figure 3.5: Direct vs. hierarchical communication for sending messages and receiving acknowledgments

group of processes becomes very large (say, thousands or even millions of members in the group), a process sending out messages and collecting acknowledgments becomes overwhelmed by that task (see Fig. 3.5a). Such algorithms inherently do not *scale*. Sometimes an efficient hardware-supported broadcast mechanism is available, and then the problem of collecting acknowledgments, also known as the *ack implosion* problem, is the worse problem of the two.

There are several ways to make algorithms more scalable. One way is to use some form of hierarchical scheme to send messages and to collect acknowledgments, for instance, by arranging the processes in a binary tree, as illustrated in Fig. 3.5b. Hierarchies can reduce the load of each process but increase the latency of the communication protocol. Additionally, hierarchies need to be reconfigured when faults occur (which may not be a trivial task), and even with this sort of hierarchies, the obligation to send and receive information, directly or indirectly, to and from every other process remains a fundamental scalability problem of reliable broadcast. In the next section we discuss how randomized approaches can circumvent this limitation.

3.8.2 Epidemic Dissemination

Nature gives us several examples of how a randomized approach can implement a fast and efficient broadcast primitive. Consider how an epidemic spreads through a population. Initially, a single individual is infected; every infected individual will in turn infect some other individuals; after some period, the whole population is infected. Rumor spreading or gossiping uses exactly the same mechanism and has proved to be a very effective way to disseminate information.

A number of broadcast algorithms have been designed based on this principle and, not surprisingly, these are often called *epidemic*, *rumor mongering*, *gossip*, or *probabilistic broadcast* algorithms. Before giving more details on these algorithms, we first define the abstraction that they implement, which we call *probabilistic broadcast*. To illustrate how algorithms can implement the abstraction, we assume a model where processes can only fail by crashing.

Module 3.7: Interface and properties of probabilistic broadcast

Module:

Name: ProbabilisticBroadcast, **instance** *pb*.

Events:

Request: ⟨ *pb*, *Broadcast* | m ⟩: Broadcasts a message m to all processes.

Indication: ⟨ *pb*, *Deliver* | p, m ⟩: Delivers a message m broadcast by process p.

Properties:

PB1: *Probabilistic validity:* There is a positive value ε such that when a correct process broadcasts a message m, the probability that every correct process eventually delivers m is at least $1 - \varepsilon$.

PB2: *No duplication:* No message is delivered more than once.

PB3: *No creation:* If a process delivers a message m with sender s, then m was previously broadcast by process s.

3.8.3 Specification

The probabilistic broadcast abstraction is depicted in Module 3.7. Its interface is the same as for best-effort broadcast (Module 3.1), and also two of its three properties, *no duplication* and *no creation*, are the same. Only the *probabilistic validity* property is weaker than the ordinary *validity* property and accounts for a failure probability ε, which is typically small.

As for previous communication abstractions introduced in this chapter, we assume that messages are implicitly addressed to all processes in the system, i.e., the goal of the sender is to have its message delivered to all processes of a given group, constituting what we call the system.

3.8.4 Randomized Algorithm: Eager Probabilistic Broadcast

Algorithm 3.9, called "Eager Probabilistic Broadcast," implements probabilistic broadcast. The sender selects k processes at random and sends them the message. In turn, each of these processes selects another k processes at random and forwards the message to those processes, and so on. The parameter k is called the *fanout* of a gossip algorithm. The algorithm may cause a process to send the message back to the same process from which it originally received the message, or to send it to another process that has already received the message.

Each step consisting of receiving a message and resending it is called a *round* of *gossiping*. The algorithm performs up to R rounds of gossiping for each message.

The description of the algorithm uses a function *picktargets*(k), which takes the fanout k as input and outputs a set of processes. It returns k random samples chosen from $\Pi \setminus \{self\}$ according to the uniform distribution without replacement. The

Algorithm 3.9: Eager Probabilistic Broadcast

Implements:
ProbabilisticBroadcast, **instance** *pb*.

Uses:
FairLossPointToPointLinks, **instance** *fll*.

upon event ⟨ *pb*, *Init* ⟩ **do**
delivered := ∅;

procedure *gossip*(*msg*) **is**
forall $t \in$ *picktargets*(k) **do trigger** ⟨ *fll*, *Send* | t, *msg* ⟩;

upon event ⟨ *pb*, *Broadcast* | m ⟩ **do**
delivered := *delivered* ∪ {m};
trigger ⟨ *pb*, *Deliver* | *self*, m ⟩;
gossip([GOSSIP, *self*, m, R]);

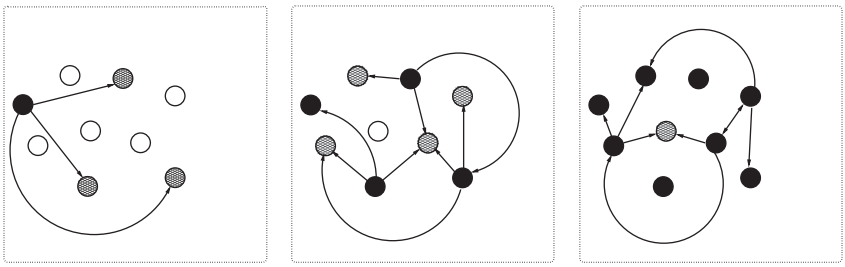
upon event ⟨ *fll*, *Deliver* | p, [GOSSIP, s, m, r] ⟩ **do**
if $m \notin$ *delivered* **then**
delivered := *delivered* ∪ {m};
trigger ⟨ *pb*, *Deliver* | s, m ⟩;
if $r > 1$ **then** *gossip*([GOSSIP, s, m, $r - 1$]);

function *random*($\mathcal{S}$) implements the random choice of an element from a set $\mathcal{S}$ for this purpose. The pseudo code looks like this:

function *picktargets*(k) **returns** set of processes **is**
targets := ∅;
while #(*targets*) $< k$ **do**
candidate := random($\Pi \setminus$ {*self*});
if *candidate* $\notin$ *targets* **then**
targets := *targets* ∪ {*candidate*};
return *targets*;

The fanout is a fundamental parameter of the algorithm. Its choice directly impacts the performance of the algorithm and the probability of successful reliable delivery (in the *probabilistic validity* property of probabilistic broadcast). A higher fanout not only increases the probability of having the entire population infected but also decreases the number of rounds required to achieve this. Note also that the algorithm induces a significant amount of redundancy in the message exchanges: any given process may receive the same message many times. A three-round execution of the algorithm with fanout three is illustrated in Fig. 3.6 for a system consisting of nine processes.

However, increasing the fanout is costly. The higher the fanout, the higher the load imposed on each process and the amount of redundant information exchanged



(a) round 1 (b) round 2 (c) round 3

Figure 3.6: Epidemic dissemination or gossip (with fanout 3)

over the network. Therefore, to select the appropriate fanout value is of particular importance. Note that there are runs of the algorithm where a transmitted message may not be delivered to all correct processes. For instance, all processes that receive the message directly from the sender may select exactly the same set of k target processes and forward the message only to them, and the algorithm may stop there. In such a case, if k is much smaller than N, not all processes will deliver the message. As another example, there might be one process that is simply never selected by any process and never receives the message. This translates into the fact that reliable delivery is not guaranteed, that is, the probability that some process never delivers the message is nonzero. But by choosing large enough values of k and R in relation to N, this probability can be made arbitrarily small.

Correctness. The *no creation* and *no duplication* properties are immediate from the underlying point-to-point links and from the use of the variable *delivered*.

For the *probabilistic validity* property, the probability that for a particular broadcast message, all correct processes become infected and deliver the message depends on the fanout k and on the maximum number of rounds R.

We now derive a simple estimate of the probability that a particular correct process delivers a message. Suppose that the underlying fair-loss links deliver every message sent by the first infected correct process (i.e., the original sender) but no further message; in other words, only the sender disseminates the broadcast message. In every round, a fraction of $\gamma = k/N$ processes become infected like this (some may have been infected before). The probability that a given correct process remains uninfected is at most $1 - \gamma$. Hence, the probability that this process is infected after R rounds is at least about $E_1 = 1 - (1 - \gamma)^R$.

Toward a second, more accurate estimate, we eliminate the simplification that only one process infects others in a round. Suppose a fraction of $d = (N - f)/N$ processes are correct; assume further that in every round, the number of *actually* infected processes is equal to their *expected* number. Denote the expected number of infected and correct processes after round r by I_r. Initially, only the sender is infected and $I_0 = 1$. After round r for $r > 0$, we observe that I_{r-1} correct processes stay infected. Among the remaining $N - I_{r-1}$ processes, we expect that a fraction of d is correct and a fraction of γ of them becomes infected:

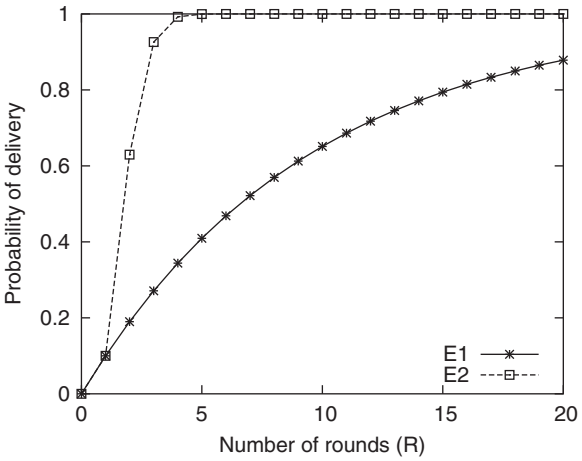


Figure 3.7: Illustration of gossip delivery probability to one correct process using the "Eager Probabilistic Broadcast" algorithm with $R = 1, \ldots, 20$ rounds, in terms of estimates E_1 and E_2 from the text

$$I_r \;=\; I_{r-1} + d\gamma(N - I_{r-1}).$$

As all I_r processes infect others in round $r+1$, the infections in round $r+1$ spread about as fast as if one process would have infected the others during additional I_r rounds. Summing this up over all R rounds, we obtain our second estimate: the probability of some correct process being infected after R rounds is about

$$E_2 \;=\; 1 - (1-\gamma)^{\sum_{r=0}^{R-1} I_r}.$$

The two estimates E_1 and E_2 of the delivery probability for one process are plotted in Fig. 3.7 for a system of $N = 100$ processes, assuming that $f = 25$ faulty processes crash initially, and fanout $k = 10$.

Performance. The number of rounds needed for a message to be delivered by all correct processes also depends on the fanout. Every round involves one communication step. The algorithm may send $O(N)$ messages in every round and $O(N^R)$ messages in total, after running for R rounds; generally, the number of messages sent by the algorithm is dominated by the messages of the last round.

3.8.5 Randomized Algorithm: Lazy Probabilistic Broadcast

The "Eager Probabilistic Broadcast" algorithm described earlier uses only gossiping to disseminate messages, where infected processes *push* messages to other processes. A major disadvantage of this approach is that it consumes considerable resources and causes many redundant transmissions, in order to achieve reliable delivery with high probability. A way to overcome this limitation is to rely on epidemic push-style broadcast only in a first phase, until many processes are infected,

Algorithm 3.10: Lazy Probabilistic Broadcast (part 1, data dissemination)

Implements:
ProbabilisticBroadcast, **instance** *pb*.

Uses:
FairLossPointToPointLinks, **instance** *fll*;
ProbabilisticBroadcast, **instance** *upb*. // an *unreliable* implementation

upon event ⟨ *pb*, *Init* ⟩ **do**
next := $[1]^N$;
lsn := 0;
pending := ∅; *stored* := ∅;

procedure *gossip*(*msg*) **is**
forall $t \in$ *picktargets*(k) **do trigger** ⟨ *fll*, *Send* | t, *msg* ⟩;

upon event ⟨ *pb*, *Broadcast* | m ⟩ **do**
lsn := *lsn* + 1;
trigger ⟨ *upb*, *Broadcast* | [DATA, *self*, m, *lsn*] ⟩;

upon event ⟨ *upb*, *Deliver* | p, [DATA, s, m, sn] ⟩ **do**
if *random*([0, 1]) > α **then**
stored := *stored* ∪ {[DATA, s, m, sn]};
if sn = *next*[s] **then**
next[s] := *next*[s] + 1;
trigger ⟨ *pb*, *Deliver* | s, m ⟩;
else if sn > *next*[s] **then**
pending := *pending* ∪ {[DATA, s, m, sn]};
forall *missing* ∈ [*next*[s], ..., $sn - 1$] **do**
if no m' exists such that [DATA, s, m', *missing*] ∈ *pending* **then**
gossip([REQUEST, *self*, s, *missing*, $R - 1$]);
starttimer(Δ, s, sn);

and to switch to a *pulling* mechanism in a second phase afterward. Gossiping until, say, half of the processes are infected is efficient. The pulling phase serves a backup to inform the processes that missed the message in the first phase. The second phase uses again gossip, but only to disseminate messages about which processes have missed a message in the first phase. This idea works for scenarios where every sender broadcasts multiple messages in sequence.

For describing an implementation of this idea in a compact way, we assume here that the first phase is realized by an *unreliable* probabilistic broadcast abstraction, as defined by Module 3.7, with a large probability ε that reliable delivery fails, in its *probabilistic validity* property. Concretely, we expect that a constant fraction of the processes, say, half of them, obtains the message after the first phase. The primitive could typically be implemented on top of fair-loss links (as the "Eager Probabilistic Broadcast" algorithm) and should work efficiently, that is, not cause an excessive amount of redundant message transmissions.

Algorithm 3.10–3.11, called "Lazy Probabilistic Broadcast," realizes probabilistic broadcast in two phases, with push-style gossiping followed by pulling. The

Algorithm 3.11: Lazy Probabilistic Broadcast (part 2, recovery)

upon event $\langle$ *fll*, *Deliver* | p, [REQUEST, q, s, sn, r] $\rangle$ **do**
  if exists m such that [DATA, s, m, sn] $\in$ *stored* **then**
    trigger $\langle$ *fll*, *Send* | q, [DATA, s, m, sn] $\rangle$;
  else if $r > 0$ **then**
    gossip([REQUEST, $q, s, sn, r-1$]);

upon event $\langle$ *fll*, *Deliver* | p, [DATA, s, m, sn] $\rangle$ **do**
  pending := *pending* $\cup$ {[DATA, s, m, sn]};

upon exists [DATA, s, x, sn] $\in$ *pending* such that $sn = next[s]$ **do**
  $next[s] := next[s] + 1$;
  pending := *pending* $\setminus$ {[DATA, s, x, sn]};
  trigger $\langle$ *pb*, *Deliver* | s, x $\rangle$;

upon event $\langle$ *Timeout* | s, sn $\rangle$ **do**
  if $sn > next[s]$ **then**
    $next[s] := sn + 1$;

algorithm assumes that each sender is transmitting a stream of numbered messages. Message omissions are detected based on gaps in the sequence numbers of received messages. Each message is disseminated using an instance *upb* of unreliable probabilistic broadcast. Each message that is retained by a randomly selected set of receivers for future retransmission. More precisely, every process that *upb*-delivers a message stores a copy of the message with probability α during some maximum amount of time. The purpose of this approach is to distribute the load of storing messages for future retransmission among all processes.

Omissions can be detected using sequence numbers associated with messages. The array variable *next* contains an entry for every process p with the sequence number of the next message to be *pb*-delivered from sender p. The process detects that it has missed one or more messages from p when the process receives a message from p with a larger sequence number than what it expects according to $next[p]$. When a process detects an omission, it uses the gossip algorithm to disseminate a retransmission request. If the request is received by one of the processes that has stored a copy of the message then this process retransmits the message. Note that, in this case, the gossip algorithm does not have to ensure that the retransmission request reaches *all* processes with high probability: it is enough that the request reaches, with high probability, one of the processes that has stored a copy of the missing message. With small probability, recovery will fail. In this case, after a timeout with delay Δ has expired, a process simply jumps ahead and skips the missed messages, such that subsequent messages from the same sender can be delivered.

The pseudo code of Algorithm 3.10–3.11 uses again the function *picktargets*(k) from the previous section. The function *random*$([0,1])$ used by the algorithm returns a random real number from the interval $[0,1]$. The algorithm may invoke multiple timers, where operation *starttimer*$(\Delta, parameters)$ starts a timer instance identified by *parameters* with delay Δ.

Garbage collection of the stored message copies is omitted in the pseudo code for simplicity. Note also that when a timeout for some sender s and sequence number sn occurs, the pseudo code may skip some messages with sender s in *pending* that have arrived meanwhile (be it through retransmissions or delayed messages from s) and that should be processed; a more complete implementation would deliver these messages and remove them from *pending*.

Correctness. The *no creation* and *no duplication* properties follow from the underlying point-to-point links and the use of sequence numbers.

The probability of delivering a message to all correct processes depends here on the fanout (as in the "Eager Probabilistic Broadcast" algorithm) and on the reliability of the underlying dissemination primitive. For instance, if half of the processes *upb*-deliver a particular message and all of them were to store it (by setting $\alpha = 0$) then the first retransmission request to reach one of these processes will be successful, and the message will be retransmitted. This means that the probability of successful retransmission behaves like the probability of successful delivery in the "Eager Probabilistic Broadcast" algorithm.

Performance. Assuming an efficient underlying dissemination primitive, the broadcasting of a message is clearly much more efficient than in the "Eager Probabilistic Broadcast" algorithm.

It is expected that, in most cases, the retransmission request message is much smaller that the original data message. Therefore, this algorithm is also much more resource-effective than the "Eager Probabilistic Broadcast" algorithm.

Practical algorithms based on this principle make a significant effort to optimize the number of processes that store copies of each broadcast message. Not surprisingly, the best results can be obtained if the physical network topology is taken into account: for instance, in a wide-area system with processes in multiple LANs, an omission in a link connecting a LAN with the rest of the system affects all processes in that LAN. Thus, it is desirable to have a copy of the message in each LAN (to recover from local omissions) and a copy outside the LAN (to recover from the omission in the link to the LAN). Similarly, the retransmission procedure, instead of being completely random, may search first for a copy in the local LAN and only afterward at more distant processes.

3.9 FIFO and Causal Broadcast

So far, we have not considered any ordering guarantee among messages delivered by different processes. In particular, when we consider a reliable broadcast abstraction, messages can be delivered in any order by distinct processes.

In this section, we introduce reliable broadcast abstractions that deliver messages according to *first-in first-out (FIFO) order* and according to *causal order*. FIFO order ensures that messages broadcast by the same sender process are delivered in the order in which they were sent. Causal order is a generalization of FIFO order that additionally preserves the potential causality among messages from multiple senders. These orderings are orthogonal to the reliability guarantees.

http://www.springer.com/978-3-642-15259-7

Introduction to Reliable and Secure Distributed Programming

Cachin, C.; Guerraoui, R.; Rodrigues, L.

2011, XIX, 367 p., Hardcover
ISBN: 978-3-642-15259-7