Lecture notes: Studying distributed systems – Reliable Broadcast

M2 MOSIG: Distributed Systems

Thomas Ropars

2025

This lecture studies the implementation of reliable broadcast primitives[1]. Considering a crash-stop failure model, it studies broadcast algorithms that differ according to the reliability and ordering guarantees they offer.

1 Motivations

In a distributed system, it is often useful to be able to send a message to a group of processes. This is the service a broadcast primitives implements.

Since faults might occur, we are interested in *reliable* broadcast primitives. Defining the properties of a reliable broadcast primitive is not as simple as it seems. A simple property would be: all processes deliver the same set of messages. But what should we do if a sender crash after it has sent its message to some processes but not to all?

2 Safety and liveness

When defining the expected properties for a distributed algorithms, we need to define properties that cover two categories: *safety* properties and *liveness* properties. Roughly speaking, a safety property stipulates that "*nothing bad*" will ever happen; a liveness property stipulates that "*something good*" will eventually happen.

More precisely, a safety property is a property whose violation can be observed by looking only at the prefix of an execution (i.e., by looking only at an execution up to a certain time). This is not the case for a liveness property. A liveness property is a property whose violation cannot be observed on a prefix of an execution: if the property does not hold on a prefix of an execution, it might still hold later.

3 Best-effort broadcast

For this algorithm and each of the following ones, we adopt the same approach. We start be defining the required safety and liveness properties of the algorithm. Then we propose an implementation that satisfies these properties.

[1]Acknowledgments: This lecture is strongly inspired from Chapter 3 of the book of Cachin, Guerraoui, and Rodrigues [1]

For all algorithms, we consider a *crash-stop* failure model for processes and we assume *quasi-reliable* channels.

3.1 Specification

The best-effort broadcast abstraction has the following properties:

- *Integrity:* Each process delivers message m at most once, and only if was broadcasted by some process.
- *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.

Integrity is a safety property while *validity* is a liveness property.

3.2 Implementation

Figure 1 presents an implementation of the best-effort broadcast primitive. The group of processes into which message m is broadcasted is noted Π.

```
1   Implements:
2   BestEffortBroadcast, instance beb.

4   Uses:
5   QuasiReliablePointToPointLinks, instance qrl.

7   Upon beb.broadcast(m):
8       for all q ∈ Π:
9           qrl.send(q, m)

11  Upon qrl.deliver(p, m)
12      Trigger beb.deliver(m)
```

Figure 1: Implementation of Best-Effort Broadcast.

The implementation is trivial as it simply relies on the properties of quasi-reliable links.

Performance: To analyze the performance of the algorithm, we consider 2 metrics: 1) The number of communication steps required to terminate one operation; 2) The total number of messages exchanged during one operation. We want to compute these metrics as of function of N, the number of processes in Π.

The algorithm presented in Figure 1 requires 1 communication step and exchanges $\mathcal{O}(N)$ messages.

4 Regular Reliable Broadcast

With the best-effort broadcast primitive, if a sender crashes, some processes might deliver a message m while the others don't. The processes do not *agree* on the set of messages to deliver. In practice,

it can be problematic if not all correct processes deliver the same set of messages. The goal of the (regular) Reliable Broadcast primitive[2] is to ensure this property.

4.1 Specification

The reliable broadcast abstraction has the following properties:

- *Integrity:* Each process delivers message m at most once, and only if was broadcasted by some process.
- *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.
- *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.

Compared to best-effort broadcast, we introduce an additional *Agreement* property, which is a liveness property[3].

4.2 Implementation

Figure 2 presents an implementation of the reliable broadcast primitive. For this implementation, we rely on the best-effort broadcast primitive. We also rely on a perfect failure detector.

The algorithm handles two main cases to ensure *agreement*:

- If a process delivers a message m for which the source has already crashed, it broadcasts (using best-effort broadcast) m again (line 31).
- When a process becomes aware that a process q has crashed, it best-effort broadcasts all the messages from q it has already delivered (line 36).

Performance : If no process crash (best case), it takes 1 communication step and $\mathcal{O}(N)$ messages to *rb-deliver* a message to all processes. The worst case is if all processes crash one by one and if the message is delivered only by one process before the sender crashes, and the process who delivered the message is the next one to crash. In this case, N communication step and and $\mathcal{O}(N^2)$ messages are required.

5 Uniform reliable broadcast

With the reliable broadcast primitive described above, we can have a scenario where a faulty process delivers a message m before it crashes, and where the correct processes never deliver this message: The agreement property only applies to correct processes.

[2]From this point on, when we simply say "*reliable broadcast*", we are referring to the regular reliable broadcast abstraction.

[3]The fact that this *agreement* property is a liveness property can be counter-intuitive since if message m is not delivered by any process, the property is ensured. However, considering the definition of *liveness* given above, it fits in this category: We can't conclude that the property is not ensured by considering a prefix of the execution.

```
13      Implements:
14      ReliableBroadcast, instance rb.

16      Uses:
17      BestEffortBroadcast, instance beb
18      PerfectFailureDetector, instance 𝒫

20      Variables:
21      correct = Π # The set of processes considered correct
22      from[N] = {∅, ∅, ..., ∅} # N is the total number of processes

24      Upon rb.broadcast(m):
25          beb.broadcast([self, m]) # self is the id of the process

27      Upon beb.deliver(p, [s, m]) # s is the id of the original source
28          if m ∉ from[s]:
29              Trigger rb.deliver(m)
30              from[s] = from[s] ∪ m
31              if s ∉ correct:
32                  beb.broadcast([s, m])

34      Upon event crash(q) raised by 𝒫:
35          correct = correct \ {q} # remove q from the set
36          for all m ∈ from[q]:
37              beb.broadcast([q, m])
```

Figure 2: Implementation of Reliable Broadcast.

In some scenarii, such behavior is not acceptable. In any scenario where processes interact with the *outside world*, we do not want this kind of behavior. Imagine that the messages are displayed on a screen[4], or that the messages are orders for transferring money. In such a case, we cannot assume anymore that it is not a problem that a process delivered m because that process crashed afterwards.

We want to implement a broadcast primitive with a stronger agreement property, which is called *Uniform Agreement*. We find the need for such a uniform property in many other abstractions.

5.1 Specification

The uniform reliable broadcast abstraction has the following properties:

- *Integrity:* Each process delivers message m at most once, and only if was broadcasted by some process.
- *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.
- *Uniform Agreement:* If a message m is delivered by some process (whether correct or not), then m is eventually delivered by every correct process.

[4]The user might have read the message before the process crashes and it might not be acceptable to consider that the message did not exist.

5.2 Implementation

Figure 3 presents an implementation of the uniform reliable broadcast primitive. For this implementation, we still rely on the best-effort broadcast abstraction. We also rely on a perfect failure detector.

```
38  Implements:
39  UniformReliableBroadcast, instance urb.

41  Uses:
42  BestEffortBroadcast, instance beb
43  PerfectFailureDetector, instance 𝒫

45  Variables:
46  correct = Π # The set of processes considered correct
47  delivered = ∅
48  pending = ∅
49  for all m:
50      ack[m] = ∅

52  Upon urb.broadcast(m):
53      pending = pending ∪ [self,m]
54      beb.broadcast([self, m])

56  Upon beb.deliver(p, [s, m]) # s is the id of the source
57      ack[m] = ack[m] ∪ p
58      if [s, m] ∉ pending:
59          pending = pending ∪ [, m]
60          beb.broadcast([s, m])
61       tryDeliver([s,m])

63  Upon event crash(q) raised by 𝒫:
64      correct = correct \ {q}
65      for all [s, m] ∈ pending:
66          tryDeliver([s,m])

68  Function canDeliver(m):
69      return (correct ⊆ ack[m]) # we receive m from all correct processes

71  Function tryDeliver([s, m]):
72      if canDeliver(m) and m ∉ delivered:
73          delivered = delivered ∪ m
74          Trigger urb.deliver([s, m])
```

Figure 3: Implementation of Uniform Reliable Broadcast.

The proposed solution is called *All-ack* Uniform Reliable Broadcast. The basic idea is that a process can deliver a message only when it has received a copy of that message from all correct processes (condition tested by the function `canDeliver()`), which implies that they have all received the message. The `ack` array includes one entry per message, that stores the list of processes from which a given message has already been received (line 57).

Performance : If no process crash (best case), it takes 2 communication steps to *rb-deliver* a message to all processes. In the first step, the source *rb-broadcast* to all (N messages sent). In the second step, all processes except the source *rb-broadcast* to all ($(N-1) \times N$ messages sent). The total number of messages sent is N^2. In the worst case, when processes crash in sequence, $N+1$ steps are required to terminate.

6 FIFO reliable broadcast

In addition to *agreement* guarantees, some applications have also requirements on the order in which messages are delivered. For instance, if we implement a distributed game, and we broadcast the successive positions of each player in the game.

We consider now the implementation of a FIFO (reliable) broadcast, that extends the reliable broadcast abstraction to guarantee if a process broadcasts two messages, they are delivered in the order they were sent.

6.1 Specification

The FIFO broadcast abstraction has the following properties:

- *Integrity:* Each process delivers message m at most once, and only if was broadcasted by some process.
- *Validity:* If a correct process broadcasts a message m, then every correct process eventually delivers m.
- *Agreement:* If a message m is delivered by some correct process, then m is eventually delivered by every correct process.
- *FIFO delivery:* If some process broadcasts message m_1 before it broadcasts message m_2, then no correct process delivers m_2 unless it has already delivered m_1.

6.2 Implementation

Figure 3 presents an implementation of the FIFO reliable broadcast primitive. A per-process sequence number piggybacked on each message allows deciding when a message can be delivered.

References

[1] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

```
76      Implements:
77      FIFOReliableBroadcast, instance frb.

79      Uses:
80      ReliableBroadcast, instance rb.

82      Variables:
83      lsn = 0 # local sequence number
84      pending = ∅
85      next[N] = {1, 1, ..., 1} # seq. number of the next message to deliver

87      Upon frb.broadcast(m):
88          lsn = lsn + 1
89          rb.broadcast([m, lsn])

91      Upon rb.deliver(p, [m, lsn]):
92      pending = pending ∪ {[p, m ,lsn]}
93      while exists (p, m′, lsn′) ∈ pending such that lsn′ = next[p]:
94          next[p] = next[p] + 1
95          pending = pending \ {m′}
96          Trigger frb.deliver(m′)
```

Figure 4: Implementation of FIFO Broadcast.