Lecture notes: Studying distributed systems – Atomic Broadcast

M2 MOSIG: Large-Scale Data Management and Distributed Systems

Thomas Ropars

2023

This lecture studies the *atomic broadcast* communication primitive (also sometimes called, *total-order broadcast*). [1].

1 Motivation

The *atomic broadcast* communication primitive is a group communication primitive, as the other broadcast primitives studied in a previous lecture. Compared to reliable broadcast primitives studied earlier, it orders all messages, even those from different senders.

Atomic broadcast is an important abstraction as it is necessary to implement active replication: it is used to ensure consistency between multiple active replicas that are used to implement a logical service, and whose behavior can be captured by a deterministic state machine.

2 Definition of atomic broadcast

Before defining atomic broadcast, we recall the definition of reliable broadcast.

2.1 Reliable broadcast

We introduced (Regular) Reliable Broadcast and Uniform Reliable Broadcast abstractions in a previous lecture. Both abstractions share the same *integrity* and *validity* properties, and only differ in their *agreement* properties (Uniform reliable broadcast implements uniform agreement):

- *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.

[1] Acknowledgments: Parts of these notes are strongly inspired by the lectures notes of Andre Schiper on *Distributed Algorithms*.

- *Uniform Agreement:* If a message m is delivered by some process (whether correct or not), then m is eventually delivered by every correct process.

In the following, we assume that reliable broadcast implements two primitives:

- *rbcast(m)* is called by a process that wants to broadcast a message
- *rdeliver(m)* is called by the broadcast abstraction to deliver a message to the application layer.

2.2 Atomic broadcast

Atomic broadcast is formally defined by the primitives *abcast* and *adeliver*, which satisfy the validity, uniform agreement, integrity properties of reliable broadcast, and the following *uniform order* property:[2]

- *Uniform total order:* If some process (correct of faulty) adelivers m before m', then every process adelivers m' only after it has adelivered m.

The name *atomic* broadcast relates to the fact that the *Uniform total order* property makes the delivery of messages appear as one *atomic* action. A message is delivered to all or to none of the processes and, if the message is delivered, every other message is ordered either before or after this message.

3 Impossibility result

Before discussing the implementation of atomic broadcast, we show that atomic broadcast is also subject to the FLP impossibility result. The proof is by contradiction: we show that if atomic broadcast is solvable, then consensus is also solvable, which contradicts the FLP impossibility result.

To show that atomic broadcast solvable implies consensus solvable, consider consensus to be solved among a set Π of processes:

- $\forall p_i \in \Pi$, let v_i be the initial value of p_i;
- $\forall p_i \in \Pi$, process p_i executes abcast(v_i);
- $\forall p_i \in \Pi$, let v be the first value adelivered by p_i: process p_i decides v.

By the ordering property of atomic broadcast, all processes decide the same value, i.e., consensus is solved. This shows the contradiction.

Therefore, *there exists no deterministic algorithm that solves atomic broadcast in an asynchronous system with reliable channels if one single process may crash.*

[2] Actually the primitive should be called *uniform* atomic broadcast. Atomic broadcast is defined by the corresponding non-uniform properties. To simplify, we use the name "atomic broadcast" to actually mean "uniform atomic broadcast".

4 Modular implementation of atomic broadcast

In this section, we present a modular implementation of atomic broadcast. We call it *modular* because it uses consensus as a black box.

Said differently, the modular implementation is based on the notion of *reduction*. Let P_1, P_2 be two problems. P_1 is *reducible* to P_2 if the solution for P_2 yields a solution for P_1.

A reduction applies in some system model. The reduction of atomic broadcast (problem P_1) to consensus (problem P_2) applies in an asynchronous system with quasi-reliable channels. The solution has been proposed by Chandra and Toueg [1]. The reduction uses the reliable broadcast primitive.

4.1 Reduction of atomic broadcast to consensus

We now describe the reduction of atomic broadcast to consensus in an asynchronous system with quasi-reliable channels [1].[3]

The idea is the following. Every process executes a sequence of consensus numbered 1, 2, ... The initial value and the decision of each consensus instance is a *set of messages* (the messages that have to be ordered). Let msg^k be the set of messages decided by consensus $\#k$:

- Each process adelivers the messages in msg^k before the messages in msg^{k+1}.
- Each process adelivers the messages in msg^k in some *deterministic* order (e.g., according to their IDs).

The algorithm that uses Consensus is given by Figure 1. The execution of abcast(m) by p_i leads first to rbcast(m) (line 14). Process p_i starts a new instance of consensus whenever there are messages that have been rdelivered but not adelivered (line 19). The initial value of p_i for each instance of consensus is the set of messages that p_i has rdelivered but not adelivered (line 22).

Note that Algorithm 1 does not launch one instance of consensus for every execution of abcast(m): more than one message may be adelivered by one instance of consensus.

Equivalence of atomic broadcast and consensus Let P_1, P_2 be two problems. If P_1 is *reducible* to P_2 and P_2 is reducible to P_1, the two problems are said to be *equivalent* (from a solvability point of view).

The solution presented in this section shows that atomic broadcast is reducible to consensus in an asynchronous system with quasi-reliable channels. The discussion in Section 3 was presenting a solution showing that consensus is reducible to atomic broadcast. It follows that, from a solvability point of view, consensus and atomic broadcast are equivalent in an asynchronous system with quasi-reliable channels.

5 Leader-based implementation of atomic broadcast

We present now a non-modular implementation of atomic broadcast. The algorithm is non-modular in the sense that it does not use consensus as a black box. The algorithm assumes quasi-reliable channels, and is not expressed in the round model.

[3] Asynchronous system and quasi-reliable channels are for the reduction. Consensus of course needs a stronger system model.

```
1   Implements:
2   AtomicBroadcast, instance ab.

4   Uses:
5   ReliableBroadcast, instance rb
6   Consensus, instance c

8   Variables:
9   k_i = 0 # Consensus number
10  adelivered_i = ∅ # set of messages adelivered by p_i
11  rdelivered_i = ∅ # set of messages rdelivered by p_i

13  Upon ab.broadcast(m): # abcast(m)
14      rb.broadcast(m)

16  Upon rb.deliver(m):
17      rdelivered_i = rdelivered_i ∪ m

19  Upon rdelivered_i − adelivered_i ≠ ∅:
20      k_i = k_i + 1
21      a_undelivered := rdelivered_i − adelivered_i
22      c.propose(k_i, a_undelivered) # Start of consensus instance k_i
23      Wail until c.decide(k_i, msg^{k_i}) # msg^{k_i} is the decision (a set of messages)
24      for all m ∈ msg^{k_i}: # iterate through the msgs in a deterministic order
25          ab.adeliver(m)
26      adelivered_i := adelivered_i ∪ msg^{k_i}
```

Figure 1: Implementation of Atomic Broadcast.

The algorithm we describe is basically Lamport's *MultiPaxos* algorithm [2]. The algorithm is based on a sequencer process s responsible for ordering messages. To abcast message m, a process sends m to s. Upon receiving m, the sequencer s assigns a sequence number i to m, and sends (m, i) to the destination processes[4]. The latter then deliver messages according to the sequence numbers[5].

This simple idea needs to be completed to handle the crash of the sequencer. We only sketch the solution and do not discuss the details. To understand the issues to be addressed, consider the following scenario:

- The sequencer has received message m, assigned sequence number i to m, and has sent (m, i) to the destinations. One destination process p receives (m, i) and delivers m at rank i.
- The sequencer crashes, and no other destination process receives (m, i).

The crash of s requires to select a new sequencer s' among the remaining processes. However, if s' has not received (m, i), it might assign the sequence number i to some other message m', leading to the violation of the properties of atomic broadcast.

To address this problem, a process p that (i) has delivered message m' with sequence number i–1, and (ii) receives (m, i), does not adeliver m immediately. Let n be the total number of destinations, and f the maximum number of faulty destinations. Process p adelivers m only once it knows that

[4] "Destination processes" refer to all processes involved in the group communication.

[5] In *MultiPaxos* the sequencer is called *leader*. This explains the reference to *leader-based* in the title of the section.

$f+1$ processes – i.e., *one correct process* – have received (m, i). This ensures that the new sequencer s' can learn that i has been assigned to m.

The following ensures that p adelivers m only after it knows that $f + 1$ processes have received (m, i):

1. Each destination process, once it has received (m, i), sends an acknowledgement message (ack, m, i) to all destinations.

2. Message m is adelivered by p only once p has received $f + 1$ messages (ack, m, i).[6]

This mechanism allows to handle the sequencer change as follows. When process p learns that the new sequencer is s', process p sends to s' the set:

$$rcv_{p,s} = \{(m, i) \mid (m, i) \text{ received by } p \text{ from } s\}.^{7}$$

The sequencer s' waits to receive $rcv_{p,s}$ from $n - f$ processes p. If one single set $rcv_{p,s}$ received by s' contains some pair (m, i), then s' knows that, if some message has been adelivered by one process at rank i, it is message m. Moreover, if s' receives no $rcv_{p,s}$ with $(-, i') \in rcv_{p,s}$, then no process can have adelivered any message at rank i'.[8]

Some other issues need to be addressed. One issue is the selection of the new sequencer. A simple technique is to rely on the *rotating* sequencer paradigm: the potential sequencers are ordered in a circular list known to all; if the sequencer needs to be changed, the next process in the list is chosen.

The complete algorithm combines the ideas presented here with the LastVoting consensus algorithm; the coordinator of LastVoting and the sequencer for atomic broadcast become the same process.

References

[1] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[2] L. Lamport. The part-time parliament. In *Concurrency: the Works of Leslie Lamport*, pages 277–317. 2019.

[6] Waiting for $f + 1$ messages will not block p if $f + 1 \leq n - f$, i.e., $n > 2f$.

[7] The information sent to s' can be optimized.

[8] Assume that some message m' is adelivered by some process at rank i'. Therefore, $f + 1$ processes p have sent $rcv_{p,s}$ with $(m', i') \in rcv_{p,s}$ to s'. The sequencer s' receives sets $rcv_{p,s}$ from $n-f$ processes. Since $(n-f)+(f+1) > n$, at least one set $rcv_{p,s}$ received by s' contains (m', i').