

5 Consensus and Reliable Broadcasts

5.1 Introduction

Consider a complete *asynchronous* network of n servers $\mathcal{P} = \{P_1, \dots, P_n\}$. Up to t servers may *fail* by silently *crashing* (and they do not recover). A server that never crashes is called *correct*. Every *pair* of servers is linked by a reliable point-to-point communication channel; this means that when a correct server sends a message to another correct server, the latter will *eventually* receive the message.

Protocols in this model need to cope with the uncertain timing of events: a message may be delayed in transit arbitrarily and a computation step of a server may take an arbitrary amount of time. The key difficulty is that a server cannot tell whether a remote server is simply slow or whether it has crashed. Designing a protocol relying on pessimistic timeouts is problematic because longer timeouts mean longer average running times, as soon as only one server has crashed.

As a compromise, one assumes that a system oscillates between

- *stable periods*, where the message delay is at most Δ and every processing step takes at most time Φ ; and
- *unstable periods*, which are asynchronous, without bounds on message delay and processing time.

This assumption is backed by experience with real systems, where one assumes values for Δ and Φ that hold with high probability.

Formally, this is captured in the model of *eventual synchrony* [DLS88]: The system starts out with an unstable period, but there exists an (unknown) time t_S , called the *global stabilization time*, after which the system is stable and remains stable forever. Of course, real systems do not remain in stable periods forever, but typically long enough for our protocol to terminate.

Protocols for coordination among the servers in this model are important for practice and have received considerable attention in research. Such protocols can ensure *liveness* (that something good eventually happens) only during stable periods, but they must guarantee *safety* (that nothing bad happens) during all periods.

Many relevant practical problems, such as atomic broadcast for state-machine replication or decentralized atomic commit for database transactions, can be reduced to the problem of reaching consensus.

5.2 Failure Detectors

A failure detector (FD) represents an abstraction of a timing assumption; it simplifies protocol design and implementation because the protocol specification does not have to contain timeout values. As an abstraction, a FD is described by its properties rather than through an implementation.

Definition 1 (Failure Detector [CT96]). Every server P_i has a local *failure detector* module \mathcal{D}_i that (periodically) outputs a list of servers that it suspects to have crashed. We say P_i *suspects* P_j whenever $P_j \in \mathcal{D}_i$.

We usually speak of “the” failure detector \mathcal{D} when every server has access to a local FD module \mathcal{D}_i with the properties of \mathcal{D} ; note that the outputs of the modules at different servers may differ from each other.

Definition 2 (Completeness).

- A failure detector satisfies *strong completeness* if eventually *every* server that crashes is permanently suspected by *every* correct server.
- A failure detector satisfies *weak completeness* if eventually *every* server that crashes is permanently suspected by *some* correct server.

Completeness alone is trivial to satisfy and hence not useful.

Definition 3 (Accuracy).

- A failure detector satisfies *strong accuracy* if *no* server is suspected before it crashes.
- A failure detector satisfies *weak accuracy* if *some* correct server is never suspected.

Such failure detectors must never output false suspicions about any server (strong accuracy) or about one particular server (weak accuracy). Therefore they are rather difficult to implement, and one considers also the following relaxation.

Definition 4 (Eventual Accuracy).

- A failure detector satisfies *eventual strong accuracy* if there is a time after which *no* correct server is suspected by any correct server.
- A failure detector satisfies *eventual weak accuracy* if there is a time after which *some* correct server is never suspected.

A failure detector is characterized by a completeness and by an accuracy condition. Two notions of completeness and four forms of accuracy define eight classes of FD:

<i>completeness</i>	<i>accuracy</i>			
			<i>eventually</i>	
	<i>strong</i>	<i>weak</i>	<i>strong</i>	<i>weak</i>
<i>strong</i>	\mathcal{P}	\mathcal{S}	$\diamond\mathcal{P}$	$\diamond\mathcal{S}$
<i>weak</i>	\mathcal{Q}	\mathcal{W}	$\diamond\mathcal{Q}$	$\diamond\mathcal{W}$

\mathcal{P} is also called the *class* of *perfect* failure detectors, or simply the *perfect* FD. Likewise, \mathcal{S} is called the *strong* FD and \mathcal{W} is called the *weak* FD. $\diamond\mathcal{P}$ is the *eventually perfect* FD, etc.

Definition 5 (Reducibility). If there exists an algorithm that emulates all properties of a FD \mathcal{D}' using only the output from a FD \mathcal{D} , we say that \mathcal{D}' is *reducible* to \mathcal{D} and that \mathcal{D}' is *weaker* than \mathcal{D} , written $\mathcal{D}' \leq \mathcal{D}$.

Similarly for classes of FD: if every FD in a class \mathcal{C}' is reducible to a FD in a class \mathcal{C} , we say that \mathcal{C}' is *reducible* to \mathcal{C} and write $\mathcal{C}' \leq \mathcal{C}$.

If $\mathcal{D} \leq \mathcal{E}$ and $\mathcal{E} \leq \mathcal{D}$, then \mathcal{D} and \mathcal{E} are *equivalent*, written $\mathcal{D} \equiv \mathcal{E}$.

Trivially, we have $\mathcal{Q} \leq \mathcal{P}$, $\mathcal{W} \leq \mathcal{S}$, etc.

Theorem 6. *Weak and strong completeness are equivalent, i.e., $\mathcal{P} \equiv \mathcal{Q}$, $\mathcal{S} \equiv \mathcal{W}$, etc.*

Proof. Reduce FD \mathcal{S} with strong completeness to FD \mathcal{D} with weak completeness as follows: every P_i periodically sends the output of \mathcal{D}_i to all servers; when P_i receives such a message with the output of \mathcal{D}_j , it updates \mathcal{S}_i to $\mathcal{S}_i \cup \mathcal{D}_j \setminus \{P_j\}$. \square

5.3 Consensus

Consensus is defined in terms of two events, *propose* and *decide*; every server P_i executes *propose*(v), where v is the value that P_i “proposes,” and every server P_i executes *decide*(v), where v is the value for which P_i “decides.”

Definition 7 (Consensus). A *consensus* protocol satisfies:

Validity: If a server *decides* v , then v was *proposed* by some server.

Agreement: No two servers decide differently.

Termination: Every correct server eventually *decides*.

This actually defines *uniform* consensus, which means that the properties hold also for faulty servers until they fail; in *non-uniform* consensus, *agreement* is restricted to the correct servers, which is sometimes easier to achieve.

It is not possible to implement consensus strictly according Definition 7 in asynchronous systems, even if $t = 1$ [FLP85]. Possible solutions are to use randomization or to make *timing assumptions*. We defer the discussion of randomized solutions and discuss it in the context of Byzantine agreement, and explore the use of timing assumptions encapsulated in a failure detector next.

Algorithm 8 (Consensus using Failure Detector \mathcal{S} or $\diamond\mathcal{S}$ [MR99]). Every P_i has access to a failure detector \mathcal{D}_i ; \mathcal{D}_i is either in \mathcal{S} or in $\diamond\mathcal{S}$ for all servers. P_i executes the following algorithm.

```

propose( $v$ ):
   $r \leftarrow 0$  // current round
  while not decided do
     $c \leftarrow (r \bmod n) + 1$  // current coordinator
     $u \leftarrow \perp$  // value received from coordinator  $P_c$  or  $\perp$  if none
    if  $i = c$  then
      send message (propose,  $r, v$ ) to all
    wait for message (propose,  $r, v'$ ) from  $P_c$  or  $P_c \in \mathcal{D}_i$ 
    if a message (propose,  $r, v'$ ) was received then
       $u \leftarrow v'$ 
    send message (vote,  $r, u$ ) to all
    wait for messages (vote,  $r, u'$ ) from all  $P_j \in Q$  for some  $Q$  s.t.
       $Q = \mathcal{P} \setminus \mathcal{D}_i$  with FD  $\mathcal{S}$ 
       $|Q| = \lceil \frac{n+1}{2} \rceil$  with FD  $\diamond\mathcal{S}$ 

```

$U \leftarrow$ set of values u' received in `vote` messages
if $U = \{u'\}$ for some $u' \neq \perp$ **then**
 send message `(decide, u')` to all
else if $U = \{u', \perp\}$ **then**
 $v \leftarrow u'$
 $r \leftarrow r + 1$

upon receiving a message `(decide, v')`:

if not *decided* **then**
 send the message `(decide, v')` to all
 decide(v')

The algorithm uses the “rotating coordinator” paradigm and provides “early termination.” The way in which the `decide` message is disseminated is a “reliable broadcast” that tolerates crash failures (see Section 5.5).

Theorem 9. *Algorithm 8 implements consensus with a strong failure detector (\mathcal{S}) for $n > t$.*

Proof sketch. Let P_c be the correct server that is never suspected and let v_c be its vote at begin of round c . All correct servers will decide in round c . Note that $c \leq n$. \square

Theorem 10. *Algorithm 8 implements consensus with an eventually strong failure detector ($\diamond\mathcal{S}$) for $n > 2t$.*

Proof sketch. Note that all sets of $\lceil \frac{n+1}{2} \rceil$ servers form a quorum system, i.e., every two quorums intersect.

- If two servers decide in the same round, then they decide the same value.
- If in some round a server receives only \perp in `vote` messages and sets $U = \{\perp\}$, then no server decides in that round, because every other server also receives at least one `vote` message with \perp (by quorum intersection).
- Suppose some server decides v' in round r . Then the value v' is contained in the `propose` message of round r and has been “locked” in the sense that all servers receive at least one `vote` message containing v' by quorum intersection. Thus, all servers proceed to the next round with $v = v'$ or decide right away. Hence, it is not possible for any server in round $r' > r$ to propose $u' \neq v'$, to decide $u' \neq v'$, or to assign $u' \neq v'$ to its v .
- If some server decides, then every other server eventually decides (because it eventually receives a `decide` message).
- There is some round in which the coordinator P_c is not suspected by any server (by weak accuracy of $\diamond\mathcal{S}$); all correct servers decide in this round.
- No server waits forever for a `propose` message by the strong completeness property of $\diamond\mathcal{S}$.
- No server waits forever for enough `vote` messages because $n - t \geq \lceil \frac{n+1}{2} \rceil$ servers are correct for $n > 2t$.

□

Combining Theorems 9 and 10 with Theorem 6 shows that consensus can also be implemented using the weak failure detectors \mathcal{W} and $\diamond\mathcal{W}$. Moreover, it has been shown that $\diamond\mathcal{W}$ is the weakest failure detector that solves consensus in the sense of Definition 5 [CHT96].

Corollary 11. *Consensus can be implemented in asynchronous systems with a weak failure detector for $n > t$ and with an eventually weak failure detector for $n > 2t$.*

5.4 Non-Blocking Atomic Commit

At the end of a distributed computation, a group of servers runs a protocol to decide if the computed state changes should be applied to their local state or the changes should be ignored. For consistency of the application the protocol should guarantee that either all servers apply the changes or none. This occurs, for example, in a *distributed database system* at the end of a transaction; the property is called *atomicity* in the context of databases.

Every server may propose to *commit* or to *abort* the computation; if one server *aborts*, then all others must also *abort*, otherwise they must *commit*. Some servers may fail (we assume here that they *never* recover).

Hence, the *non-blocking atomic commit* problem is a variation of consensus with domain $\{\text{commit}, \text{abort}\}$ and the following notion of *validity*:

- a) If some server proposes *abort*, then all servers must decide *abort*.
- b) If all servers are correct and propose *commit*, then all servers must decide *commit*.

We use a perfect failure detector and a consensus primitive to implement non-blocking atomic commit with the following algorithm:

1. Every server sends its proposed action, *commit* or *abort*, to all others.
2. When a server receives n messages indicating *commit*, it starts consensus and proposes *commit*; otherwise, when the server receives at least one message that indicates *abort* or when the server suspects some other server, it starts consensus and proposes *abort*.
3. Every server returns whatever the consensus protocol decides.

However, the database literature usually allows recoveries, which makes this a different problem. Practical database systems use a single transaction coordinator that implements the decision [BHG87]; when the coordinator fails, the 2PC protocol blocks until the coordinator recovers, and the 3PC protocol needs a complex and synchronous recovery procedure when the backup coordinators fail as well. See also [BT93, GL06].

5.5 Reliable Broadcasts

Our system model includes only point-to-point links for communication. If a server wants to broadcast a message to all others, the server may crash during the operation and it is possible that some servers receive a message but others don't. The purpose of *reliable broadcast* and its extensions is to prevent that. When additional ordering requirements are imposed (partial orders such as FIFO and causal or total order), the problem becomes harder to solve. This section is based on [HT93].

5.5.1 Reliable Broadcast

Reliable broadcast (RBC) requires that all correct servers deliver the same set of messages, and that this set includes all messages broadcast by correct servers but no spurious messages. The sender associated to a particular message is a distinguished server and its identity is assumed to be known. Formally, RBC is characterized by two events $r\text{-broadcast}(m)$, executed by the sender to “r-broadcast” the message m , and $r\text{-deliver}(m)$, executed by all servers when they “r-deliver” m .

When multiple messages are broadcast, one may imagine that the servers run multiple *instances* of a broadcast primitive. Every instance is associated with a unique identifier that is also added to all messages generated by the protocol; since the sender is known, this identifier may also include the identity of the sender.

Definition 12 (Reliable Broadcast). A protocol for *reliable broadcast* satisfies:

Validity: If a correct server $r\text{-broadcasts}$ a message m , then it eventually $r\text{-delivers}$ m .

Agreement: If a server $r\text{-delivers}$ a message m , then all correct servers eventually $r\text{-deliver}$ m .

Integrity: Every server delivers at most one message m , and only if m was previously broadcast by the associated sender.

Thus if the sender is faulty, either all servers deliver a message or none. This actually defines *uniform* reliable broadcast; all other broadcasts in this section are also uniform.

Algorithm 13 (Reliable Broadcast). We consider the implementation of a single instance (a protocol for broadcasting multiple messages is obtained in a straightforward way by aggregating as many instances as there are messages). Assume that the sender of every broadcast instance is known implicitly; server P_i executes the following steps:

```
 $r\text{-broadcast}(m)$ :    // only when  $P_i$  is the sender
    send the message  $(\text{send}, m)$  to itself
upon receiving a message  $(\text{send}, m)$ :
    if message  $m$  has not been  $r\text{-delivered}$  yet then
        send the message  $(\text{send}, m)$  to all
         $r\text{-deliver}(m)$ 
```

Although our network model assumes reliable point-to-point links between all servers, the algorithm works even if every pair of correct servers is connected only via a path consisting entirely of correct servers (in which case the statement “send to all” means “send to all directly connected servers”). The following theorem is immediate.

Theorem 14. *Algorithm 13 implements reliable broadcast for $n > t$.*

5.5.2 FIFO Broadcast

When multiple messages are reliably broadcast concurrently, RBC does not guarantee anything about the order in which the messages are delivered. One of the simplest orderings is provided by FIFO broadcast, which guarantees that messages from the same sender are delivered in the same sequence as they were broadcast by the sender; this does not affect messages from different senders.

A protocol for *FIFO broadcast* is a protocol for reliable broadcast defined in terms of two events *f-broadcast* and *f-deliver* that also satisfies:

FIFO Order: If a server *f-broadcasts* a message *m* before it *f-broadcasts* a message *m'*, then no server *f-delivers* *m'* unless it has previously *f-delivered* *m*.

Algorithm 15 (FIFO Broadcast from Reliable Broadcast). Given an implementation of reliable broadcast, server P_i executes the following steps:

initialization:

$\mathcal{M} \leftarrow \emptyset$ // set of received but not *f-delivered* messages
 $s \leftarrow 0$ // P_i 's sequence number
 $s_j \leftarrow 0$ ($\forall j \in [1, n]$) // next sequence number to be *f-delivered* from P_j

f-broadcast(*m*): // only when P_i is the sender

r-broadcast the message (*s*, *m*)
 $s \leftarrow s + 1$

upon *r-deliver* (*s'*, *m'*) with sender P_j :

$\mathcal{M} \leftarrow \mathcal{M} \cup \{(j, s', m')\}$
while $\exists (j, t, m) \in \mathcal{M}$ such that $t = s_j$ **do**
 f-deliver(*m*)
 $s_j \leftarrow s_j + 1$

Theorem 16. Given a protocol for reliable broadcast, Algorithm 15 implements FIFO broadcast.

5.5.3 Causal Broadcast

The causal precedence relation is an important concept in distributed computing. An event *e* *causally precedes* *f*, written $e \rightarrow f$, whenever the same server executes *e* before *f*, or when *e* is the event of sending a message and *f* the event of receiving it, or if there is an event *g* such that $e \rightarrow g$ and $g \rightarrow f$. Causal order is a specialization of FIFO order.

A protocol for *causal broadcast* is a protocol for reliable broadcast defined in terms of two events *c-broadcast* and *c-deliver* that also satisfies:

Causal Order: If the *c-broadcast* of a message *m* causally precedes the *c-broadcast* of a message *m'*, then no server *c-delivers* *m'* unless it has previously *c-delivered* *m*.

Algorithm 17 (Causal Broadcast from FIFO Broadcast). Given an implementation of FIFO broadcast, server P_i executes the following steps:

initialization:

$M \leftarrow []$ // list of recently *c-delivered* messages

c-broadcast(m): // only when P_i is the sender

f-broadcast the message ($M||m$), where $||$ means to append an element m to a list M

$M \leftarrow []$

upon *f-deliver* ($[m_1, m_2, \dots, m_l]$):

for $k = 1, \dots, l$ **do**

if m_k has not been *c-delivered* yet **then**

c-deliver(m_k)

$M \leftarrow M||m_k$

Theorem 18. *Given an implementation of FIFO broadcast, Algorithm 17 implements causal broadcast.*

5.5.4 Atomic Broadcast

FIFO and causal orders are partial orders. In particular, causal order does not impose anything for two causally *unrelated* messages and it is possible that the servers deliver such messages in different orders. Many applications do not allow such behavior because they must maintain a consistent state at all servers; these applications require that the same state updates are executed by all servers and that every server executes them in the same order. Such a total order is provided by atomic broadcast.

A protocol for *atomic broadcast* is a protocol for reliable broadcast defined in terms of two events *a-broadcast* and *a-deliver* that also satisfies:

Total Order: If two servers P_i and P_j both *a-deliver* messages m and m' , then P_i *a-delivers* m before m' if and only if P_j *a-delivers* m before m' .

Note that *total order* does *not* imply *FIFO* or *causal order*; hence, FIFO and causal broadcasts are orthogonal to atomic broadcast, and it is possible to consider also FIFO atomic and causal atomic broadcasts.

Implementing the total order property is considerably more difficult than the other orderings considered before. In fact, atomic broadcast is as powerful as consensus and hence impossible in asynchronous networks using deterministic protocols.

Theorem 19. *Given a protocol for atomic broadcast, there is a protocol for consensus that does not involve any additional messages.*

Proof sketch. To *propose* a value v , a server uses the atomic broadcast protocol and *a-broadcasts* v ; then every server waits for the *a-delivery* of the *first* message v' and *decides* for v' . The *agreement* and *total order* properties of atomic broadcast imply *agreement* of consensus. \square

A convenient way to implement atomic broadcast is to use a consensus primitive. The atomic broadcast algorithm below proceeds in global rounds; it uses one instance of consensus in every round to agree on a set of messages, which are then delivered in a fixed order at the end of the round.

Algorithm 20 (Atomic Broadcast from Consensus and Reliable Broadcast [CT96]). Given an implementation of consensus and reliable broadcast, server P_i executes the following steps:

initialization:

$\mathcal{R} \leftarrow \emptyset$ // set of r -delivered messages
 $\mathcal{A} \leftarrow \emptyset$ // set of a -delivered messages
 $r \leftarrow 0$ // round number

a -broadcast(m):

r -broadcast(m)

upon r -deliver(m):

$\mathcal{R} \leftarrow \mathcal{R} \cup \{m\}$

repeat forever: // concurrently with the above statements

if $\mathcal{R} \setminus \mathcal{A} \neq \emptyset$ **then**

$propose(\mathcal{R} \setminus \mathcal{A})$ in consensus r

wait for $decide(\mathcal{S})$ of consensus r

a -deliver all messages in $\mathcal{S} \setminus \mathcal{A}$ in some deterministic order

$\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{S}$

$r \leftarrow r + 1$

Theorem 21. Given protocols for consensus and for reliable broadcast, Algorithm 20 implements atomic broadcast.

Proof sketch. Validity follows from the validity of reliable broadcast and from the validity and agreement of consensus (if a correct server a -broadcasts a message m , it is eventually contained in the set \mathcal{R} of every correct server) combined with the integrity of consensus (eventually, every set proposed in consensus contains m).

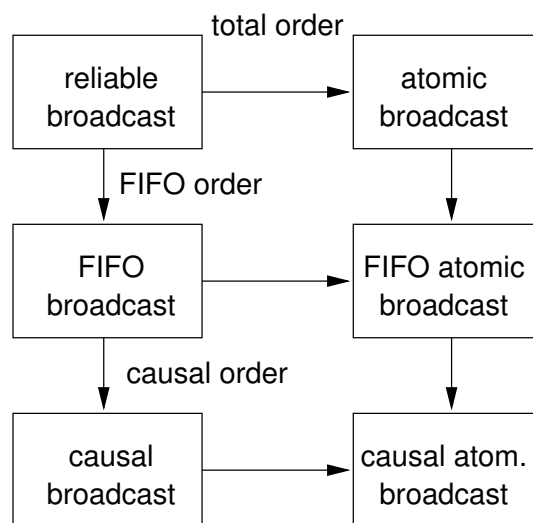
Agreement and total order are based on the following two facts. Suppose servers P_i and P_j are correct. Then:

- If P_i executes $propose$ for consensus r , then P_j eventually executes $propose$ for consensus r .
- Let $\mathcal{B}_r(i)$ denote the set $\mathcal{S} \setminus \mathcal{A}$ of P_i in round r of the algorithm. If P_i a -delivers all messages in $\mathcal{B}_r(i)$, then P_i eventually a -delivers all messages in $\mathcal{B}_r(i)$; moreover, $\mathcal{B}_r(i) = \mathcal{B}_r(j)$ for all $r \geq 0$.

□

Corollary 22. Atomic broadcast and consensus are equivalent in asynchronous distributed systems with reliable point-to-point links and crash failures.

5.5.5 Summary



Relations among the broadcast primitives [HT93].

References

- [BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*, Addison-Wesley, 1987.
- [BT93] Ö. Babaoglu and S. Toueg, *Non-blocking atomic commitment*, Distributed Systems (S. J. Mullender, ed.), ACM Press & Addison-Wesley, New York, 1993.
- [CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg, *The weakest failure detector for solving consensus*, Journal of the ACM **43** (1996), no. 4, 685–722.
- [CT96] T. D. Chandra and S. Toueg, *Unreliable failure detectors for reliable distributed systems*, Journal of the ACM **43** (1996), no. 2, 225–267.
- [DLS88] C. Dwork, N. Lynch, and L. Stockmeyer, *Consensus in the presence of partial synchrony*, Journal of the ACM **35** (1988), no. 2, 288–323.
- [FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson, *Impossibility of distributed consensus with one faulty process*, Journal of the ACM **32** (1985), no. 2, 374–382.
- [GL06] J. Gray and L. Lamport, *Consensus on transaction commit*, ACM Transactions on Database Systems **31** (2006), no. 1, 133–160.
- [HT93] V. Hadzilacos and S. Toueg, *Fault-tolerant broadcasts and related problems*, Distributed Systems (S. J. Mullender, ed.), ACM Press & Addison-Wesley, New York, 1993, Expanded version appears as Technical Report TR94-1425, Department of Computer Science, Cornell University, Ithaca NY, 1994.
- [MR99] A. Mostfaoui and M. Raynal, *Solving consensus using Chandra-Toueg’s unreliable failure detectors: A general quorum-based approach*, Proc. 13th International Symposium on Distributed Computing (DISC) (P. Jayanti, ed.), Lecture Notes in Computer Science, vol. 1693, Springer, 1999.