8 Fault-Tolerant Distributed Systems

8.1 Introduction

Consider a complete *asynchronous* network of n servers $P_1, \ldots, P_n$. 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 (i.e., if a correct server sends a message to another correct server, it will *eventually* receive the message).

Coordination of all (correct) servers in this model has received considerable attention. Many relevant practical problems, such as atomic broadcast or decentralized atomic commitment of transactions, can be reduced to the problem of reaching consensus.

8.2 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 8.1 (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 Definition 8.1 in asynchronous systems [FLP85], even if $t = 1$. Possible solutions are (1) to use randomization (cf. Section on Asynchronous Byzantine Agreement) or (2) to make *timing assumptions*. We explore (2) here and discuss (1) in the context of Byzantine agreement.

Example 8.2 (Non-Blocking Atomic Commitment using Consensus). At the end of a distributed computation, a group of processes (servers) enters a protocol to commit the changes of their local state. Every process may propose to *commit* or to *abort* the computation; if one process *aborts*, then all others must also *abort*, otherwise they must *commit*. On top, some processes may fail (we assume here that it *never* recovers) and if a process believes that another process has failed (e.g., by using a "failure detector," see below), it is also possible to *abort*.

This is a variation of consensus with domain $\{commit, abort\}$ and the following notion of *validity*:

- If a process decides *commit*, then processes all have proposed *commit*.
- If all processes propose *commit* and none of them is believed to have failed, then they must decide *commit*.

The following algorithm implements distributed non-blocking atomic commitment using consensus:

- First, every process sends its proposed action, *commit* or *abort*, to all others.
- When a process receives n messages indicating *commit*, it starts consensus and proposes to *commit*; otherwise, when at least one process sent a message that indicates *abort* or when a process believes that another process has failed, it starts consensus and proposes to *abort*.
- The output is set to the decision of the consensus protocol.

However, the database literature usually allows recoveries and hence most practical database systems use a centralized controller (when the coordinator fails, the operation stalls until it recovers).

8.3 Failure Detectors

Definition 8.3 (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 $j \in \mathcal{D}_i$

A failure detector (FD) represents an abstraction of a timing assumption; a FD is described by its abstract properties rather than through an implementation. 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 8.4 (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 8.5 (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 and are therefore rather difficult to implement. Therefore one considers also:

Definition 8.6 (Eventual Accuracy).

- A failure detector satisfies *eventual strong accuracy* if there is a time after which *no* server is suspected before it crashes.
- 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:

completeness	*accuracy*			
			eventually	
	strong	weak	strong	weak
strong	$\mathcal{P}$	$\mathcal{S}$	$\Diamond\mathcal{P}$	$\Diamond\mathcal{S}$
weak	$\mathcal{Q}$	$\mathcal{W}$	$\Diamond\mathcal{Q}$	$\Diamond\mathcal{W}$

$\mathcal{P}$ is also called the [class of] "perfect," $\mathcal{S}$ the [class of] "strong," and $\mathcal{W}$ the [class of] "weak" failure detectors; read $\Diamond$ as "eventually."

Definition 8.7 (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 $\mathcal{Q} \leq \mathcal{P}$, $\mathcal{S} \leq \mathcal{W}$, etc.

Theorem 8.8. *Weak and strong completeness are equivalent, i.e., $\mathcal{P} \equiv \mathcal{Q}$, $\mathcal{S} \equiv \mathcal{W}$, etc.*

Proof. There is a transformation that reduces a FD $\mathcal{S}$ with strong completeness to FD $\mathcal{D}$ with weak completeness: 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 $\mathcal{S}_i \cup \mathcal{D}_j \setminus \{P_j\}$. □

8.4 Consensus using Failure Detectors

Algorithm 8.9 ($\mathcal{S}$-based Consensus). Every P_i has access to a failure detector $\mathcal{D}_i \in \mathcal{S}$.

upon *propose*(v):
  for $r = 1, \ldots, n$ **do**
    if $r = i$ **then**
      send the message $(\texttt{vote}, r, v)$ to all
    wait for a message $(\texttt{vote}, r, v')$ or $r \in \mathcal{D}_i$

if a message $(\texttt{vote}, r, v')$ has been received **then**
$v \leftarrow v'$
decide(v)

Theorem 8.10. *Algorithm 8.9 implements consensus with a strong failure detector for $n > t$.*

Proof idea. Let P_c be the correct server that is never suspected and let v_c be its vote at begin of round c. At the end of round c and ever after, all servers have $v = v_c$. □

Algorithm 8.11 ($\Diamond\mathcal{S}$-based Consensus). Every P_i has access to a failure detector $\mathcal{D}_i \in \Diamond\mathcal{S}$ and executes the following algorithm.

upon *propose*(v):
$r \leftarrow 0$ // current round
$\tau \leftarrow 0$ // last round in which v was updated
while not *decided* **do**
$c \leftarrow (r \mod n) + 1$
send message $(\texttt{vote}, r, v, \tau)$ to all
if $i = c$ **then**
wait for messages $(\texttt{vote}, r, v', \tau')$ from $\lceil \frac{n+1}{2} \rceil$ servers
$t \leftarrow$ largest τ' received in `vote` messages
$v \leftarrow$ some v' received in a `vote` message with $\tau' = t$
send message $(\texttt{propose}, r, v)$ to all
wait for a message $(\texttt{propose}, r, v')$ from P_c or $c \in D_i$
if a $(\texttt{propose}, \dots)$ message was received **then**
$v \leftarrow v'$; $\tau \leftarrow r$
send message `ack` to P_c
else
send message `nack` to P_c
if $i = c$ **then**
wait for `ack` or `nack` messages from $\lceil \frac{n+1}{2} \rceil$ servers
if all are `ack` messages **then**
send message $(\texttt{decide}, v)$ to all
$r \leftarrow r + 1$

upon *receiving a message* $(\texttt{decide}, v')$:
if not *decided* **then**
send the message $(\texttt{decide}, v')$ to all
decide(v')

The algorithm uses the "rotating coordinator" paradigm; the way in which the `decide` message is disseminated is a "reliable broadcast" that tolerates crash failures.

Theorem 8.12. *Algorithm 8.11 implements consensus with an eventually strong failure detector for $n > 2t$.*

Proof idea. *Agreement* and *termination* are based on these two facts:

- Suppose there is a round r in which the coordinator P_c is not suspected by any server; then the value v_c contained in the `propose` message of round r will also be contained in any `propose` message of rounds $r' > r$ because $\lceil \frac{n+1}{2} \rceil$ servers form a quorum.
- If some server decides, then every other server eventually decides.

□

Combining Theorems 8.10 and 8.12 with Theorem 8.8 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 8.7 [CHT96].

Corollary 8.13. *Consensus can be implemented in asynchronous systems with a weak failure detector for $n > t$ and with an eventually weak failure detector for $n > 2t$.*

8.5 Broadcast Problems

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].

8.5.1 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-broadcast*(m), executed by the sender to "r-broadcast" the message m, and *r-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 8.14 (Reliable Broadcast). A protocol for *reliable broadcast*[1] satisfies:

Validity: If a correct server *r-broadcasts* a message m, then it eventually *r-delivers* m.

Agreement: If a server *r-delivers* a message m, then all correct servers eventually *r-deliver* m.

[1] This actually defines *uniform* reliable broadcast; all other broadcasts in this section are also uniform.

Integrity: Every server delivers any particular message m at most once, 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.

Algorithm 8.15 (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). Let P_s denote the sender of the broadcast instance; server P_i executes the following steps:

upon *r-broadcast*(m): // sender P_s only
  send the message (send, m) to itself

upon *receiving message* (send, m):
  if message m has not been *r-delivered* yet **then**
    send the message (send, m) to all
    r-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 8.16. *Algorithm 8.15 implements reliable broadcast for $n > t$.*

8.5.2 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 *r-broadcasts* a message m', then no server *f-delivers* m' unless it has previously *f-delivered* m.

Algorithm 8.17 (FIFO Broadcast from Reliable Broadcast). Given an implementation of reliable broadcast, server P_i executes the following steps:

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

upon *f-broadcast*(m): // sender P_s only
  r-broadcast the message (s, m)
  $s \leftarrow s + 1$

upon *r-delivering* (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 $t = n_j$ **do**
    f-deliver(m)
    $n_j \leftarrow n_j + 1$

Theorem 8.18. *Given a protocol for reliable broadcast, Algorithm 8.17 implements FIFO broadcast.*

Proof omitted.

8.5.3 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: 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 8.19 (Causal Broadcast from FIFO Broadcast). Given an implementation of FIFO broadcast, server P_i executes the following steps:

initialization:
  $M \leftarrow \emptyset$ // list of recently *c-delivered* messages

upon *c-broadcast*(m): // sender P_s only
  f-broadcast the message $(M \| m)$, where $\|$ means to append an element m to a list M
  $M \leftarrow \perp$

upon *f-delivering* $([m_1, m_2, \ldots, m_l])$:
  for $k = 1, \ldots, l$ **do**
    if m_k has not been *c-delivered* yet **then**
      c-deliver(m_k)
      $M \leftarrow M \| m_k$

Theorem 8.20. *Given an implementation of FIFO broadcast, Algorithm 8.19 implements causal broadcast.*

Proof omitted.

8.5.4 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 the 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 8.21. *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. □

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 8.22 (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

upon *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 8.23. *Given protocols for consensus and for reliable broadcast, Algorithm 8.22 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).

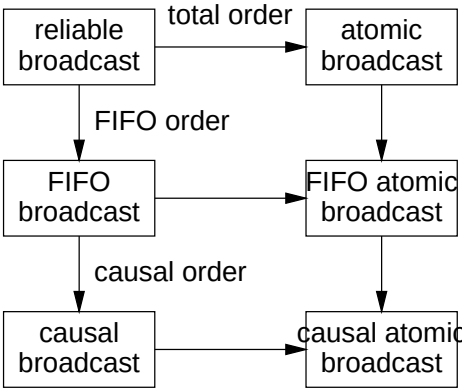
Agreement and total order are based on the following two facts. Let $\mathcal{B}_r(i)$ denote the set $\mathcal{S} \setminus \mathcal{A}$ of server P_i in round r of the algorithm and suppose P_i and P_j are correct. Then:

- If P_i executes *propose* for consensus r, then P_j eventually executes *propose* for consensus r.
- 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 8.24. *Atomic broadcast and consensus are equivalent in asynchronous distributed systems with reliable point-to-point links and crash failures.*

8.5.5 Summary



Relations among the broadcast primitives [HT93].

References

[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.

[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.

[HT93] V. Hadzilacos and S. Toueg, *Fault-tolerant broadcasts and related problems*, Distributed Systems (New York) (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.