

Distributed algorithms

Pierre Colson

Wednesday 05 January 2022

Contents

General	2
Fair-loss links	2
Stubborn links	2
Reliable (Perfect) links	2
Reliable Broadcast	3
Best-effort Broadcast (beb)	3
Reliable Broadcast (rb)	3
Uniform Reliable Broadcast (urb)	4
Causal Broadcast	4
Causality	4
Causal broadcast	4
Reliable Causal Broadcast (rcb)	4
Uniform Causal Broadcast (ucb)	4
Reliable Causal Order Broadcast (rco)	5
Total Order Broadcast (tob)	5
(Uniform) Consensus	6
Total Order (to)	6
Shared Memory	6
Regular register	6
Atomic Register	8
Terminating Reliable Broadcast (TRB)	9

Markdown version on [github](#)

Compiled using [pandoc](#) and [gpdf script](#)

More fiches [here](#)

General

- The distributed system is made of a finite set of **processes** : each process models a **sequential** program
- Every pair of processes is connected by a **link** through which the processes exchange **messages**
- **Safety** is a property which states that nothing bad should happen
- **Liveness** is a property which states that something good should happen
- Two kinds of failures are mainly considered
 - **Omissions** : The process omits to send messages it is supposed to send
 - **Arbitrary** : The process sends messages it is not supposed to send
- A **correct** process is a process that does not fail (that does not crash)
- A **Failure detector** is a distributed oracle that provides processes with suspicions about crashed processes
 - It is implemented using *timing assumptions*
 - **Perfect** :
 - * *Strong Completeness* : Eventually, every process that crashes is permanently suspected by every other correct process
 - * *Strong Accuracy* : No process is suspected before it crashes
 - **Eventually Perfect** :
 - * *Strong Completeness*
 - * *Eventually Strong Accuracy* : Eventually, no correct process is ever suspected

Fair-loss links

- **FL1. Fair-loss** : If a message is sent infinitely often by p_i to p_j and neither p_i or p_j crashes then m is delivered infinitely often by p_j
- **FL2. Finite duplication** : If a message m is sent a finite number of times by p_i to p_j , m is delivered a finite number of times by p_j
- **FL3. No creation** : No message is delivered unless it was sent

Stubborn links

- **SL1. Stubborn delivery** : If a process p_i sends a message m to a correct process p_j , and p_i does not crash, then p_j delivers m an infinite number of times
- **SL2. No creation** : No message is delivered unless it was sent

```
Implements: StubbornLinks (sp2p)
Uses : FairLossLinks (flp2p)
upon event <sp2pSend, dest, m> do
  while (true) do
    trigger <flp2pSend, dest, m>
upon event <flp2pDeliver, src, m> do
  trigger <sp2pDeliver, src, m>
```

Reliable (Perfect) links

- **PL1. Validity** : If p_i and p_j are correct
- **PL2. No duplication** : No message is delivered (to a process) more than once
- **PL3. No creation** : No message is delivered unless it was sent
- Roughly speaking, reliable links ensure that messages exchanged between correct processes are *not lost*

```
Implements: PerfectLinks (pp2p)
Uses: StubbornLinks (sp2p)
upon event <Init> do delivered := emptySet
upon event <pp2pSend, dest, m> do
  trigger <sp2Send, dest, m>
```

```

upon event <sp2pDeliver, src, m> do
  if m not in delivered then
    trigger <pp2pDeliver, src, m>
    add m to delivered

```

Reliable Broadcast

Best-effort Broadcast (beb)

- **BEB1. Validity** : If p_i and p_j are correct then every message broadcast by p_i is eventually delivered by p_j
- **BEB2. No duplication** : No message is delivered more than once
- **BEB3. No creation** : No messages is delivered unless it was broadcast

Implements: BestEffortBroadcast (beb)

Uses: PerfectLinks (pp2p)

```

upon event <bebBroadcast, m> do
  forall pi in S do
    trigger <pp2pSend, pi, m>
upon event <pp2pDeliver, pi, m> do
  trigger <dedDeliver, pi, m>

```

Reliable Broadcast (rb)

- **RB1** = BEB1
- **RB2** = BEB2
- **RB3** = RB3
- **RB4. Agreement** : For any message m , if any correct process delivers m , then every correct process delivers m

Implements: ReliableBroadcast (rb)

Uses:

BestEffortBroadcast (beb)

PerfectFailureDetector (P)

```

upon event <Init> do
  delivered := emptySet
  correct := S
  forall pi in S do from[pi] := emptySet
upon event <rbBroadcast, m> do
  delivered := delivered U {m}
  trigger <rbDeliver, self, m>
  trigger <bebBroadcast, [data, self, m]>
upon event <crash, pi> do
  correct := correct \ {pi}
  forall [pj, m] in from[pi] do
    trigger <bebBroadcast, [data, pj, m]>
upon event <bebDeliver, pi, [data, pj, m]> do
  if m not in delivered then
    delivered := delivered U {m}
    trigger <rbDeliver, pj, m>
    if pi not in correct then
      trigger <bebBroadcast, [data, pj, m]>
  else
    from[pi] := from[pi] U {[pj, m]}

```

Uniform Reliable Broadcast (urb)

- URB1 = BEB1
- URB2 = BEB2
- URB3 = BEB3
- URB4. **Uniform Agreement** : For any message m , if any process delivers m , then every process delivers m

Implements: UniformBroadcast (urb)

Uses:

BestEffortBroadcast (beb)
PerfectFailureDetector (P)

```
upon event <Init> do
  correct := S
  delivered := forward := emptySet
  ack[Message] := emptySet
upon event <urbBroadcast, m> do
  forward := forward U {[self, m]}
  trigger <bebBroadcast, [data, self, m]>
upon event <bebDeliver, pi, [data, pj, m]> do
  ack[m] := ack[m] U {pi}
  if [pi, m] not in forward then
    forward := forward U {[pj, m]}
    trigger <bebBroadcast, [data, pj, m]>
upon event (for any [pj, m] in forward) <correct in ack[m]> and <m not in delivered> do
  delivered := delivered U {m}
  trigger <urbDeliver, pj, m>
```

Causal Broadcast

- A **non-blocking** algorithm using the past
- A **blocking** algorithm using **vector clocks**

Causality

- Let m_1 and m_2 be any two messages : $m_1 \rightarrow m_2$ (m_1 causally precedes m_2) iff
 - **C1. Fifo order** : Some process p_i broadcast m_1 before broadcasting m_2
 - **C2. Local order** : Some process p_i delivers m_1 and then broadcast m_2
 - **C3. Transitivity** : There is a message m_3 such that $m_1 \rightarrow m_3$ and $m_3 \rightarrow m_2$

Causal broadcast

- **CO** : If any process p_i delivers a message m_2 , then p_i must have delivered every message m_1 such that $m_1 \rightarrow m_2$

Reliable Causal Broadcast (rcb)

- RB1, RB2, RB3, RB4
- CO

Uniform Causal Broadcast (ucb)

- URB1, URB2, URB3, URB4
- CO

Reliable Causal Order Broadcast (rco)

```
Implements: ReliableCausalOrderBroadcast (rco)
Uses : ReliableBroadcast (rb)
upon event <Init> do
  delivered := past := emptySet
upon event <rcoBroadcast, m> do
  trigger <rbBroadcast, [data, past, m]>
  past := past U {[self, m]}
upon event <rbDeliver, pi [data, pastm, m]> do
  if m not in delivered then
    forall [sn, n] in pastm do
      if n not in delivered then
        trigger <rcoDeliver, sn, n>
        delivered := delivered U {n}
        past := past U {[self, n]}
    trigger <rcoDeliver, pi, m>
    delivered := delivered U {m}
    past := past U {[pi, m]}
```

```
Implements ReliableCausalOrderBroadcast (rco)
Uses: ReliableBroadcast (rb)
upon event <Init> do
  forall pi in S: VC[pi] := 0
  pending := emptySet
upon event <rcoBroadcast, m> do
  trigger <rcoDeliver, self, m>
  trigger <rbBroadcast, [data, VC, m]>
  VC[self] := VC[self] + 1
upon event <rbDeliver, pj, [data, VCm, m]> do
  if pj not self then
    pending := pending U (pj, [data, VCm, m])
    deliver-pending
procedure deliver-pending is
  while (s, [data, VCm, m]) in pending do
    if forall pk: (VC[pk] >= VCm[pk]) do
      pending := pending - (s, [data, VCm, m])
      trigger <rcoDeliver, self, m>
      VC[s] := VC[s] + 1
```

- These algo ensure causal reliable broadcast
- If we replace reliable broadcast with uniform reliable broadcast, these algo would ensure uniform causal broadcast

Total Order Broadcast (tob)

- In **reliable** broadcast, the processes are free to deliver messages in any order they wish
- In **causal** broadcast, the processes need to deliver messages according to some order (causal order)
 - The order imposed by causal broadcast is however partial : some messages might be delivered in different order by the processes
- In **total order** broadcast, the processes must deliver all messages according to the same order(i.e. the order is now total)
 - This order does not need to respect causality (or event FIFO ordering)
- **RB1. Validity** : If p_i and p_j are correct, then every message broadcast by p_i is eventually delivered by

- p_j
- **RB2. No duplication** : No message is delivered more than once
- **RB3. No creation** : No message is delivered unless it was broadcast
- **RB4. (Uniform) Agreement** : For any message m . If a correct (any) process delivers m , then every correct process delivers m
- **(Uniform) Total order** : Let m and m' be any two messages. Let p_i be any (correct) process that delivers m without having delivered m' . Then no (correct) process delivers m' before m

(Uniform) Consensus

- In the (uniform) consensus problem the processes propose values and need to agree on one among these values
- **C1. Validity** : Any value decided is a value proposed
- **C2. (Uniform) Agreement** : No two correct (any) processes decide differently
- **C3. Termination** : Every correct process eventually decides
- **C4. Integrity** : Every process decides at most once

Total Order (to)

```

Implements: TotalOrder (to)
Uses:
    ReliableBroadcast (rb)
    Consensus (cons)
upon event <Init> do
    unordered := delivered := emptySet
    wait := false;
    sn := 1
upon event <toBroadcast, m> do
    trigger <rbBroadcast, m>
upon event <rbDeliver, sm, m> and (m not in delivered) do
    unordered := unordered U {(sm, m)}
upon event (unordered not emptySet) and not wait do
    wait := true
    trigger <Propose, unordered>sn
upon event <Decide, decided>sn do
    unordered := unordered \ decided
    ordered := deterministicSort(decided)
    forall (sm, m) in ordered do
        trigger <toDeliver, sm, m>
        delivered := delivered U {m}
    sn := sn + 1
    wait = false

```

Shared Memory

Regular register

- Assumes only one writer
- Provides *strong* guarantees when there is no concurrent operations
- When some operations are concurrent, the register provides *minimal* guarantees
- **Read()** returns :
 - The last value written if there is no concurrent or failed operations

- Otherwise the last value written on *any* value concurrently written i.e. the input parameter of some `Write()`
- We assume **fail-stop** model
 - Process can fail by crashing (no recovery)
 - Channels are reliable
 - Failure detection is perfect
- We implement a **regular** register
 - Every process p_i has a local copy of the register value v_i
 - Every process reads **locally**
 - The writer writes **globally**

```
Write(v) at pi
  send [W, w] to all
  forall pj, wait until either
    receive [ack] or
    detect [pj]
  return ok
```

```
Read() at pi
  return vi
```

```
At pi
  when receive [W, w] from pj
    vi := w
    send [ack] to pj
```

- We assume while failure detection is not perfect
 - P_1 is the writer and any process can be reader
 - A majority of the process is correct
 - Channels are reliable
- We implement a **regular** register
 - Every process p_i maintains a local copy of the register v_i , as well as a sequence number sn_i and a read timestamp rs_i
 - Process p_1 maintains in addition a timestamp ts_1

```
Write(v) at p1
  ts1 ++
  send [W, ts1, v] to all
  when receive [W, ts1, ack] from majority
    return ok
```

```
Read() at pi
  rsi ++
  send [R, rsi] to all
  when receive [R, rsi, snj, vj] from majority
    v := vj with the largest snj
  return v
```

```
At pi
  when receive [W, ts1, v] from p1
    if ts1 > sni then
      vi = v
      sni := ts1
      send[W, ts1, ack] to p1
  when receive [R, rsj] from pj
    send [R, rsj, sni, vi] to pj
```

Atomic Register

- An **Atomic Register** provides strong guarantees even when there is concurrency and failures : the execution is equivalent to a sequential and failure-free execution
- Every failed (write) operation appears to be either complete or not to have been invoked at all
- Every complete operation appears to be executed at some instant between its invocation and reply time events
- We implement a **fail-stop 1-N atomic register**
 - Every process maintains a local value of the register as well as a sequence number
 - The writer, p_1 , maintains, in addition a timestamp ts_1
 - Any process can read in the register

```
Write(v) at p1
  ts1++
  send [W, ts1, v] to all
  forall pi wait until either
    receive [ack] or
    detect [pi]
  return ok
```

```
Read() at pi
  send [W, sni, vi] to all
  forall pi wait until either
    receive [ack] or
    suspect [pj]
  return vi
```

```
At pi
  When pi receive [W, ts, v] from pj
    if ts > sni then
      vi := v
      sni := ts
    send [ack] to pj
```

- We implement a **fail-stop N-N atomic register**

```
Write(v) at pi
  send [W] to all
  forall pj wait until either
    receive [W, snj] or
    suspect [pj]
  (sn, id) := (highest snj + 1, i)
  send [W, (sni, id), v] to all
  forall pj wait until either
    receive [W, (sn, id), ack] or
    detect [pj]
  return ok
```

```
Read() at pi
  send [R] to all
  forall pj wait until either
    receive [R, (snj, idj), vj] or
    suspect pj
  v = vj with the highest (snj, idj)
  (sn, id) = highest (snj, idj)
```



```

send [W, (sn, id), v] to all
forall pj wait until either
  receive [W, (sn, id), ack] or
  detect [pj]
return v

```

At p_i

T1 :

```

when receive [W] from pj
  send [W, sn] to pj
when receive [R] from pj
  send [R, (sn, id), vi] to pj

```

T2 :

```

when receive [W, (snj, idj), v] from pj
  if (snj, idj) > (sn, id) then
    vi := v
    (sn, id) = (snj, idj)
  send [W, (sn, id), ack] to pj
when receive [W, (snj, idj), v] from pj
  if (snj, idj) > (sn, id) then
    vi := v
    (sn, id) := (snj, idj)
  send [W, (sn, id), ack] to pj

```

- From fail-stop to **fail-silent**
 - We assume a majority of correct processes
 - In the 1-N algorithm, the writer writes in a majority using a timestamp determined locally and the reader selects a value from a majority and then imposes this value on a majority
 - In the N-N algorithm, the writers determines first the timestamp using a majority

Terminating Reliable Broadcast (TRB)

- Like reliable broadcast, terminating reliable broadcast (TRB) is a communication primitive used to disseminate a message among a set of processes in a reliable way
- TRB is however strictly stronger than (uniform) reliable broadcast
- Like with reliable broadcast, correct processes in TRB agree on the set of messages they deliver
- Like with (uniform) reliable broadcast, every correct process in TRB delivers every message delivered by any correct process
- Unlike with reliable broadcast, every correct process delivers a message, even if the broadcaster crashes
- The problem is defined for a specific broadcaster process $p_i = src$ (known by all processes)
 - Process src is supposed to broadcast a message m (distinct from φ)
 - The other processes need to deliver m if src is correct but may deliver φ if src crashes
- **TRB1. Integrity** : If a process delivers a message m , then either m is φ or m was broadcasted by src
- **TRB2. Validity** : If the sender src is correct and broadcasts a message m , then src eventually delivers m
- **TRB3. (Uniform) Agreement** : For any message m , if a correct (any) process delivers m , then every correct process delivers m
- **TRB4. Termination** : Every correct process eventually delivers exactly one message

Implements: trbBroadcast (trb)

Uses:

```

BestEffortBroadcast (beb)
PerfectFailureDetector (P)

```

```

Consensus (cons)
upon event <Init> do
  prop := 0
  correct := S
upon event <trbBroadcast, m> do
  trigger <bebBroadcast, m>
upon event <crash, src> and (prop = 0) do
  prop := phi
upon event <bebDeliver, src, m> and (prop = 0) do
  prop := m
upon event (prop != 0) do
  trigger <Propose, prop>
upon event <Decide, decision> do
  trigger <trbDeliver, src, decision>

```

- We give an algorithm that implements P unsigned TRB. More precisely, we assume that every process p_i can use an infinite number of instances of TRB where p_i is the sender src
 1. Every process p_i keeps on trbBroadcasting messages m_{i1}, m_{i2} etc
 2. If a process p_k delivers φ_i , p_k suspects p_i