Classes of process failures

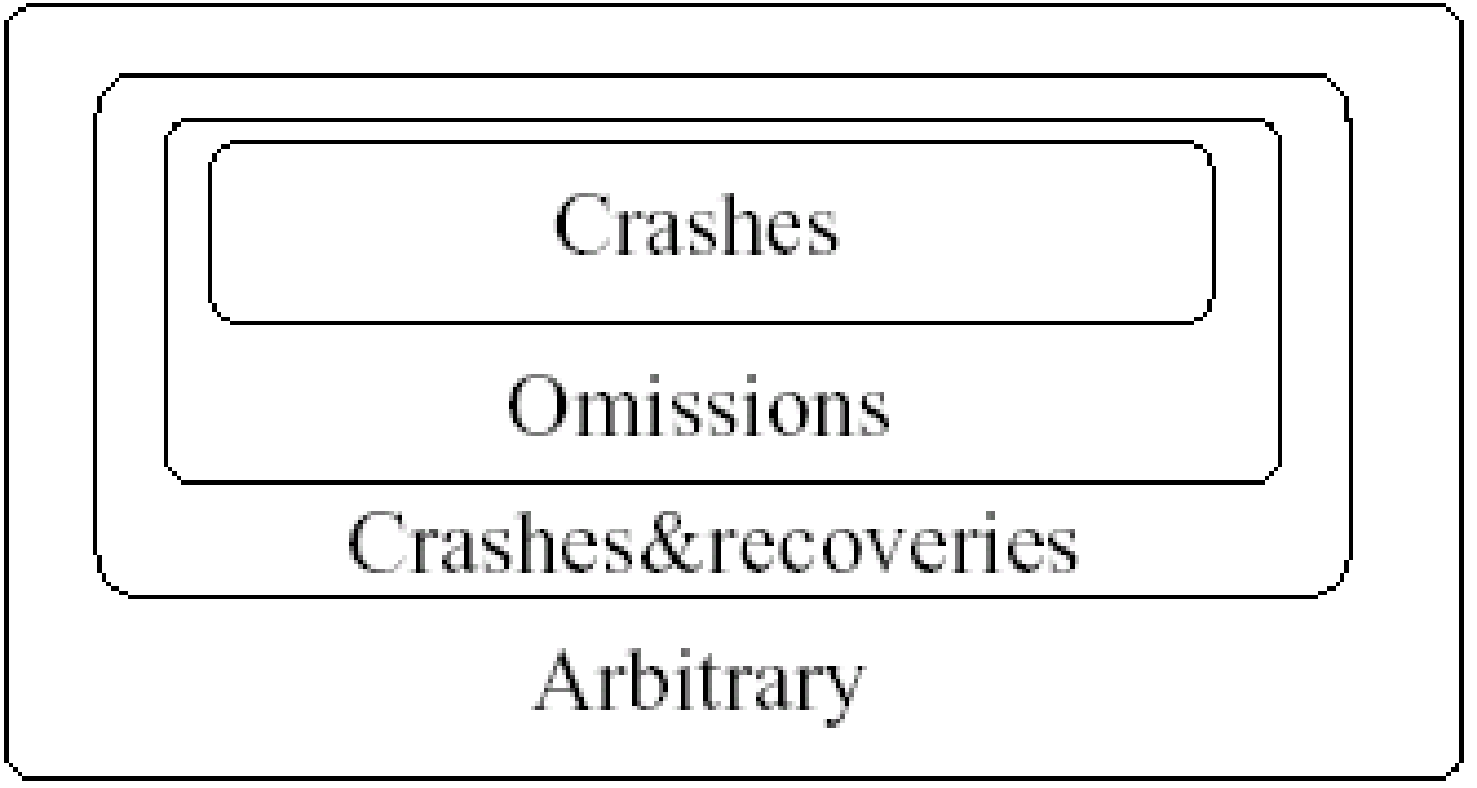


Figure 2.2. Failure modes of a process

Module:

Name: FairLossPointToPointLinks (flp2p).

Events:

Request: ⟨ *flp2pSend*, dest, m ⟩: Used to request the transmission of message m to process *dest*.

Indication: ⟨ *flp2pDeliver*, src, m ⟩: Used to deliver message m sent by process *src*.

Properties:

FLL1: *Fair loss:* If a message m is sent infinitely often by process p_i to process p_j, and neither p_i nor p_j crash, then m is delivered an infinite number of times by p_j.

FLL2: *Finite duplication:* If a message m is sent a finite number of times by process p_i to process p_j, then m cannot be delivered an infinite number of times by p_j.

FLL3: *No creation:* If a message m is delivered by some process p_j, then m has been previously sent to p_j by some process p_i.

Module 2.1 Interface and properties of fair-lossy point-to-point links.

Module:

Name: StubbornPointToPointLink (sp2p).

Events:

Request: ⟨ *sp2pSend*, *dest*, m ⟩: Used to request the transmission of message m to process *dest*.

Indication:⟨ *sp2pDeliver*, *src*, m ⟩: Used to deliver message m sent by process *src*.

Properties:

SL1: *Stubborn delivery:* Let p_i be any process that sends a message m to a correct process p_j. If p_i does not crash, then p_j delivers m an infinite number of times.

SL2: *No creation:* If a message m is delivered by some process p_j, then m was previously sent to p_j by some process p_i.

Module 2.2 Interface and properties of stubborn point-to-point links.

Algorithm 2.1 Stubborn links using fair-loss links.

Implements:
StubbornPointToPointLink (sp2p).

Uses:
FairLossPointToPointLinks (flp2p).

upon event ⟨ *sp2pSend*, dest, m ⟩ **do**
    while (true) **do**
        trigger ⟨ *flp2pSend*, dest, m ⟩;

upon event ⟨ *flp2pDeliver*, src, m ⟩ **do**
    trigger ⟨ *sp2pDeliver*, src, m ⟩;

Module:

Name: PerfectPointToPointLink (pp2p).

Events:

Request: ⟨ *pp2pSend*, *dest*, m ⟩: Used to request the transmission of message m to process *dest*.

Indication:⟨ *pp2pDeliver*, *src*, m ⟩: Used to deliver message m sent by process *src*.

Properties:

PL1: *Reliable delivery:* Let p_i be any process that sends a message m to a process p_j. If neither p_i nor p_j crashes, then p_j eventually delivers m.

PL2: *No duplication:* No message is delivered by a process more than once.

PL3: *No creation:* If a message m is delivered by some process p_j, then m was previously sent to p_j by some process p_i.

Module 2.3 Interface and properties of perfect point-to-point links.

Algorithm 2.2 Perfect links using stubborn links.

Implements:
PerfectPointToPointLinks (pp2p).

Uses:
StubbornPointToPointLinks (sp2p).

upon event ⟨ *Init* ⟩ **do**
delivered := ∅;

upon event ⟨ *pp2pSend*, dest, m ⟩ **do**
trigger ⟨ *sp2pSend*, dest, m ⟩;

upon event ⟨ *sp2pDeliver*, src, m ⟩ **do**
if $m \notin$ delivered **then**
delivered := delivered ∪{m};
trigger ⟨ *pp2pDeliver*, src, m ⟩;

Partially synchronous systems

- System that eventually is synchronous (without stating when exactly).
- System that might not always be synchronous and there is no bound on the period during which it is asynchronous.
- System characterized by periods during which it is synchronous, and some of these periods are long enough for an algorithm to terminate its execution.

Module:

Name: PerfectFailureDetector ($\mathcal{P}$).

Events:

Indication: $\langle$ *crash*, p_i $\rangle$: Used to notify that process p_i has crashed.

Properties:

PFD1: *Strong completeness:* Eventually every process that crashes is permanently detected by every correct process.

PFD2: *Strong accuracy:* No process is detected by any process before it crashes.

Module 2.4 Interface and properties of the perfect failure detector.

Algorithm 2.3 Perfect failure detector with perfect links and timeouts.

Implements:
PerfectFailureDetector ($\mathcal{P}$).

Uses:
PerfectPointToPointLinks (pp2p).

upon event $\langle$ *Init* $\rangle$ **do**
alive := Π;
suspected := $\emptyset$;

upon event $\langle$ *TimeDelay* $\rangle$ **do**
$\forall_{p_i \in \Pi}$:
if $p_i \notin$ *alive* **and** $p_i \notin$ *suspected* **then**
suspected := suspected $\cup \{p_i\}$;
trigger $\langle$ *crash*, p_i $\rangle$;
alive := $\emptyset$;
$\forall_{p_i \in \Pi}$: **trigger** $\langle$ *pp2pSend*, p_i, [DATA, heartbeat] $\rangle$;

upon event $\langle$ *pp2pDeliver*, src, [DATA, heartbeat] $\rangle$ **do**
alive := alive $\cup$ {*src*};

Module:

Name: EventuallyPerfectFailureDetector ($\Diamond\mathcal{P}$).

Events:

Indication: $\langle$ *suspect*, p_i $\rangle$: Used to notify that process p_i is suspected to have crashed.

Indication: $\langle$ *restore*, p_i $\rangle$: Used to notify that process p_i is not suspected anymore.

Properties:

EPFD1: *Eventual strong completeness:* Eventually, every process that crashes is permanently suspected by every correct process.

EPFD2: *Eventual strong accuracy:* Eventually, no correct process is suspected by any correct process.

Module 2.5 Interface and properties of the eventually perfect failure detector.

Algorithm 2.4 Eventually perfect failure detector with perfect links and timeouts.

Implements:
EventuallyPerfectFailureDetector ($\diamond\mathcal{P}$).

Uses:
PerfectPointToPointLinks (pp2p).

upon event $\langle$ *Init* $\rangle$ **do**
alive := Π;
suspected := $\emptyset$;

upon event $\langle$ *TimeDelay* $\rangle$ **do**
$\forall_{p_i \in \Pi}$:
if $p_i \notin$ *alive* **then**
suspected := suspected $\cup \{p_i\}$;
trigger $\langle$ *crash*, p_i $\rangle$;
else
if $p_i \in$ *suspected* **then**
suspected := suspected $\setminus \{p_i\}$;
TimeDelay := TimeDelay + Δ;
trigger $\langle$ *restore*, p_i $\rangle$;
alive := $\emptyset$;
$\forall_{p_i \in \Pi}$: **trigger** $\langle$ *pp2pSend*, p_i, [DATA, heartbeat] $\rangle$;

upon event $\langle$ *pp2pDeliver*, src, [DATA, heartbeat] $\rangle$ **do**
alive := alive $\cup \{src\}$;

Module:

Name: EventualLeaderDetector (Ω).

Events:

Indication: $\langle$ *trust*, p_i $\rangle$: Used to notify that process p_i is trusted to be leader.

Properties:

CD1: *Eventual accuracy:* There is a time after which every correct process trusts some correct process.

CD2: *Eventual agreement:* There is a time after which no two correct processes trust different processes.

Module 2.6 Interface and properties of the eventual leader detector.

Algorithm 2.5 Eventually leader election with crash-recovery processes, stubborn links and timeouts .

Implements:

EventualLeaderDetector (Ω).

Uses:

StubbornPointToPointLinks (flp2p).

upon event $\langle$ *Init* $\rangle$ **do**
leader := p_1;
possible := Π;
epoch := 0;

upon event $\langle$ *Recovery* $\rangle$ **do**
retrieve(epoch);
epoch := epoch + 1;
store(epoch);

upon event $\langle$ *TimeDelay* $\rangle$ **do**
if *leader* $\neq$ select(possible) **then**
TimeDelay := TimeDelay + Δ;
leader := select(possible);
if {*leader*} $\neq \emptyset$ **then**
trigger $\langle$ *trust*, leader $\rangle$;
possible := $\emptyset$;
$\forall_{p_i \in \Pi}$: **trigger** $\langle$ *sp2pSend*, p_i, [DATA, heartbeat,epoch] $\rangle$;

upon event $\langle$ *flp2pDeliver*, src, [DATA, heartbeat,epc] $\rangle$ **do**
possible := possible $\cup$ {(*src*, *epc*)};

Specific system models

- **Fail-stop model.** Processes execute deterministic algorithms, unless they possibly crash and stop executing any computation, links are perfect, and there available the perfect failure detector.
- **Fail-silent model**. Processes execute deterministic algorithms, unless they possibly crash and stop executing any computation, links are perfect.

Specific system models

- **Fail-noisy model**. Processes execute deterministic algorithms, unless they possibly crash and stop executing any computation, links are perfect, and moreover there available the eventually perfect failure detector.
- **Randomized model**. Processes may use a random oracle to choose among several steps to execute.

Module:

Name: BestEffortBroadcast (beb).

Events:

Request: ⟨ *bebBroadcast*, m ⟩: Used to broadcast message m to all processes.

Indication: ⟨ *bebDeliver*, src, m ⟩: Used to deliver message m broadcast by process *src*.

Properties:

BEB1: *Best-effort validity:* For any two processes p_i and p_j. 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:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

Module 3.1 Interface and properties of best-effort broadcast.

Algorithm 3.1 Basic Broadcast.

Implements:

BestEffortBroadcast (beb).

Uses:

PerfectPointToPointLinks (pp2p).

upon event $\langle$ *bebBroadcast*, m $\rangle$ **do**

$\forall_{p_i \in \Pi}$:

trigger $\langle$ *pp2pSend*, p_i, m $\rangle$;

upon event $\langle$ *pp2pDeliver*, p_i, m $\rangle$ **do**

trigger $\langle$ *bebDeliver*, p_i, m $\rangle$;

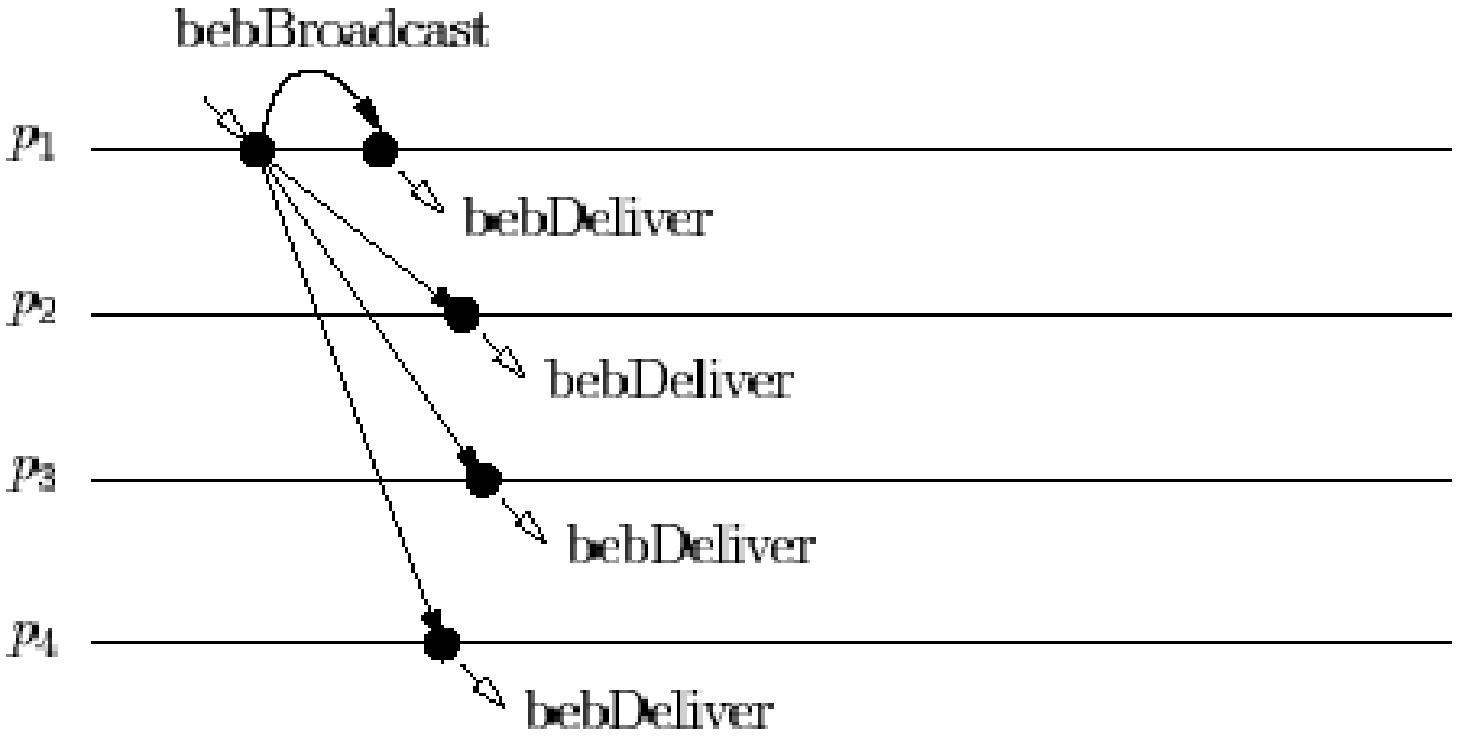


Figure 3.1. Sample execution of Basic Broadcast algorithm.

Module:

Name: (regular)ReliableBroadcast (rb).

Events:

Request: $\langle$ *rbBroadcast*, m $\rangle$: Used to broadcast message m.

Indication: $\langle$ *rbDeliver*, src, m $\rangle$: Used to deliver message m broadcast by process src.

Properties:

RB1: *Validity:* If a correct process p_i broadcasts a message m, then p_i eventually delivers m.

RB2: *No duplication:* No message is delivered more than once.

RB3: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

RB4: *Agreement:* If a message m is delivered by some correct process p_i, then m is eventually delivered by every correct process p_j.

Module 3.2 Interface and properties of reliable broadcast.

Algorithm 3.2 Lazy reliable broadcast.

Implements:
ReliableBroadcast (rb).

Uses:
BestEffortBroadcast (beb).
PerfectFailureDetector ($\mathcal{P}$).

upon event $\langle$ *Init* $\rangle$ **do**
delivered := $\emptyset$;
correct := Π;
$\forall_{p_i \in \Pi}$: from[p_i] := $\emptyset$;

upon event $\langle$ *rbBroadcast*, m $\rangle$ **do**
trigger $\langle$ *bebBroadcast*, [DATA, self, m] $\rangle$;

upon event $\langle$ *bebDeliver*, p_i, [DATA, s_m, m] $\rangle$ **do**
if $m \notin$ delivered **then**
delivered := delivered $\cup$ $\{m\}$
trigger $\langle$ *rbDeliver*, s_m, m $\rangle$;
from[p_i] := from[p_i] $\cup$ $\{[s_m, m]\}$
if $p_i \notin$ *correct* **then**
trigger $\langle$ *bebBroadcast*, [DATA, s_m, m] $\rangle$;

upon event $\langle$ *crash*, p_i $\rangle$ **do**
correct := correct $\setminus$ $\{p_i\}$
forall $[s_m, m] \in$ *from*[p_i]: **do**
trigger $\langle$ *bebBroadcast*, [DATA, s_m, m] $\rangle$;

Algorithm 3.3 Eager reliable broadcast.

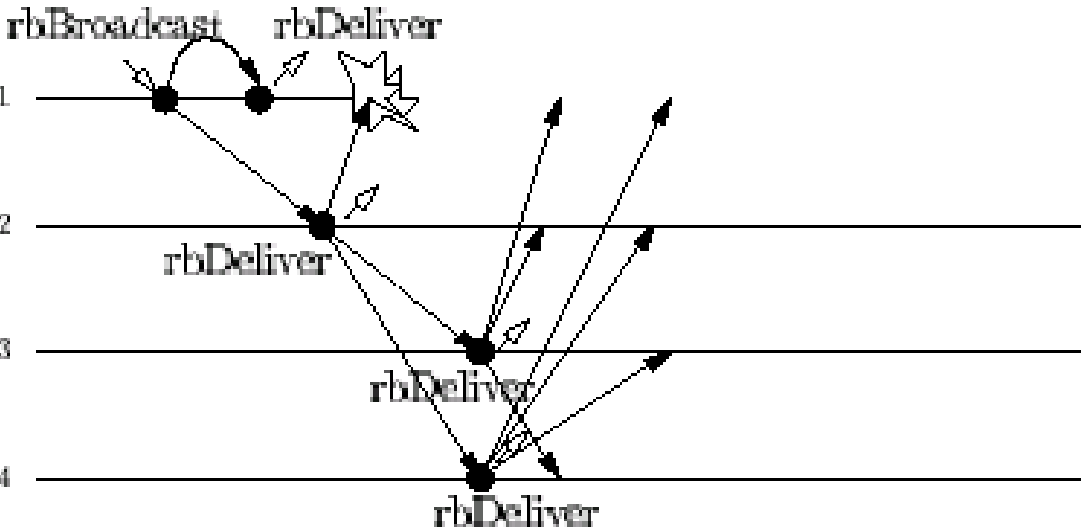
Implements:
ReliableBroadcast (rb).

Uses:
BestEffortBroadcast (beb).

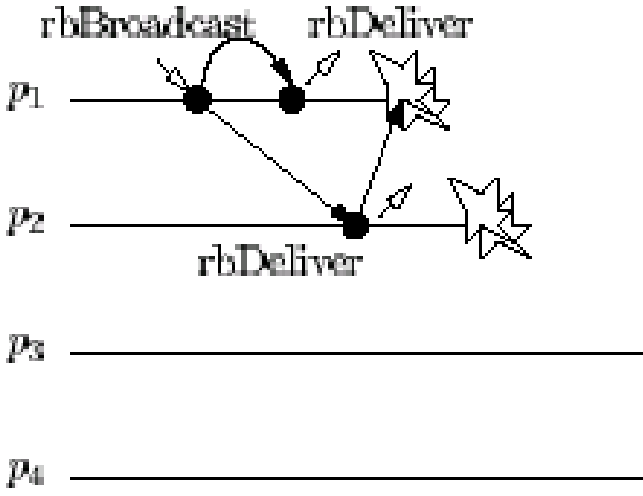
upon event $\langle$ *Init* $\rangle$ **do**
delivered := $\emptyset$;

upon event $\langle$ *rbBroadcast*, m $\rangle$ **do**
delivered := delivered $\cup$ $\{m\}$
trigger $\langle$ *rbDeliver*, self, m $\rangle$;
trigger $\langle$ *bebBroadcast*, [DATA, self, m] $\rangle$;

upon event $\langle$ *bebDeliver*, p_i, [DATA, s_m, m] $\rangle$ **do**
if $m \notin$ delivered **do**
delivered := delivered $\cup$ $\{m\}$
trigger $\langle$ *rbDeliver*, s_m, m $\rangle$;
trigger $\langle$ *bebBroadcast*, [DATA, s_m, m] $\rangle$;



(a)



(b)

igure 3.2. Sample executions of eager reliable broadcast.

Module:

Name: UniformReliableBroadcast (urb).

Events:

$\langle$ *urbBroadcast*, m $\rangle$, $\langle$ *urbDeliver*, src, m $\rangle$, with the same meaning and interface as in regular reliable broadcast.

Properties:

RB1-RB3: Same as in regular reliable broadcast.

URB4: *Uniform Agreement:* If a message m is delivered by some process p_i (whether correct or faulty), then m is also eventually delivered by every other correct process p_j.

Module 3.3 Interface and properties of uniform reliable broadcast.

Algorithm 3.4 All-ack uniform reliable broadcast.

Implements:
UniformReliableBroadcast (urb).

Uses:
BestEffortBroadcast (beb).
PerfectFailureDetector ($\mathcal{P}$).

function canDeliver(m) **returns** boolean **is**
return (correct $\subset$ ack$_m$) $\wedge$ ($m \notin$ delivered);

upon event $\langle$ *Init* $\rangle$ **do**
delivered := forward := $\emptyset$;
correct := Π;
ack$_m$:= $\emptyset, \forall_m$;

upon event $\langle$ *urbBroadcast*, m $\rangle$ **do**
forward := forward $\cup$ $\{m\}$
trigger $\langle$ *bebBroadcast*, [DATA, self, m] $\rangle$;

upon event $\langle$ *bebDeliver*, p_i, [DATA, s_m, m] $\rangle$ **do**
ack$_m$:= ack$_m$ $\cup$ $\{p_i\}$
if $m \notin$ forward **do**
forward := forward $\cup$ $\{m\}$;
trigger $\langle$ *bebBroadcast*, [DATA, s_m, m] $\rangle$;

upon event $\langle$ *crash*, p_i $\rangle$ **do**
correct := correct $\backslash\{p_i\}$;

upon (canDeliver(m)) **do**
delivered := delivered $\cup$ $\{m\}$;
trigger $\langle$ *urbDeliver*, s_m, m $\rangle$;

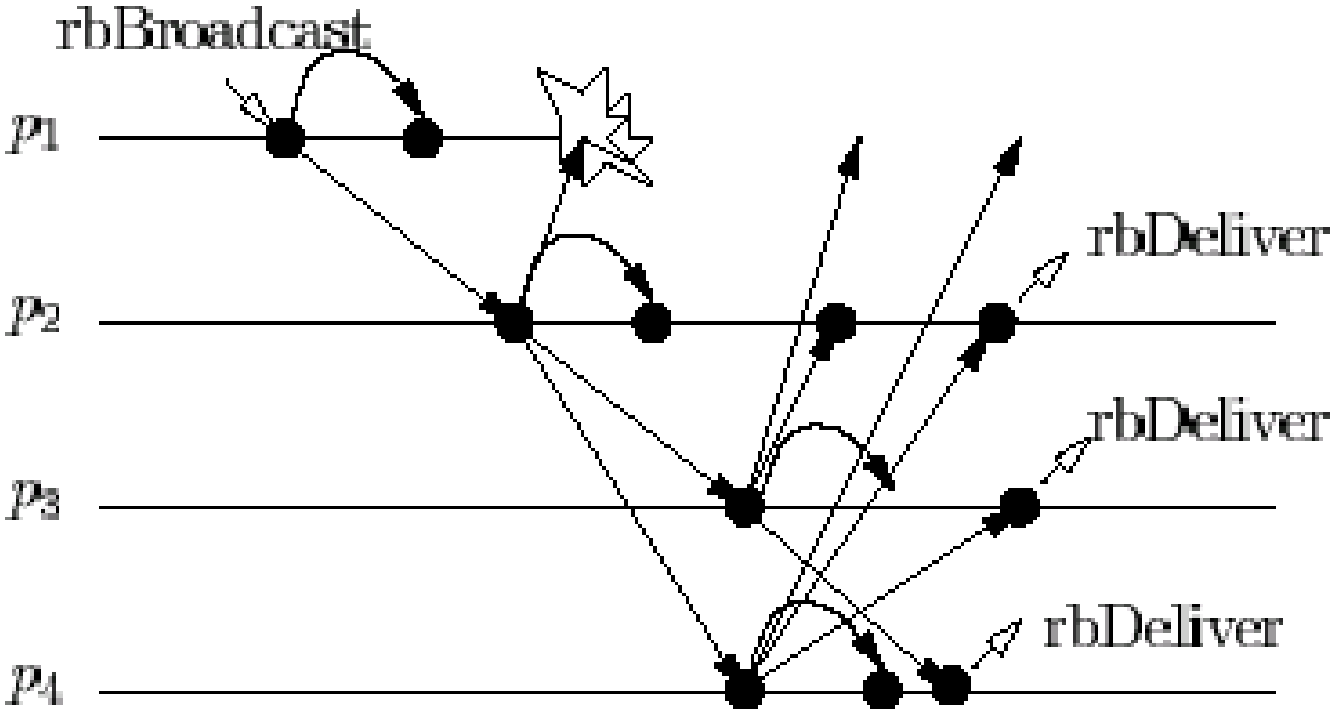


Figure 3.3. Sample execution of uniform reliable broadcast.

lgorithm 3.5 Majority-ack uniform reliable broadcast.

nplements:
UniformReliableBroadcast (urb).

ses:
BestEffortBroadcast (beb).

nction canDeliver(m) **returns** boolean **is**
return ($|\text{ack}_m| > \text{N}/2$) $\wedge$ ($m \notin$ delivered);
Except for the function above, and the non-use of the // perfect failure detector, same as Algorithm 3.

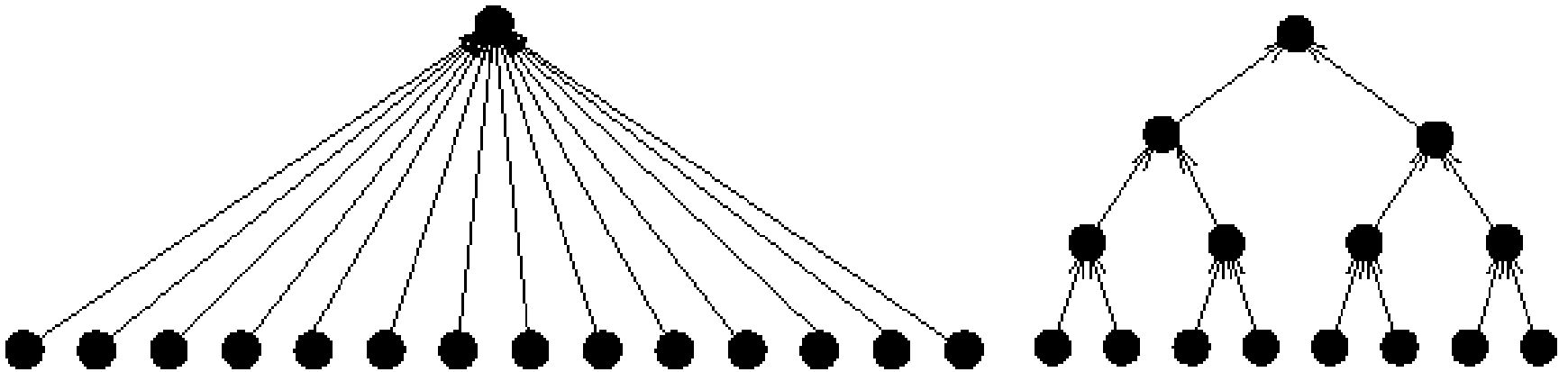


Figure 3.4. Ack implosion and ack tree.

Module:

Name: Probabilistic Broadcast (pb).

Events:

Request: ⟨ *pbBroadcast*, m ⟩: Used to broadcast message m to all processes.

Indication: ⟨ *pbDeliver*, src, m ⟩: Used to deliver message m broadcast by process *src*.

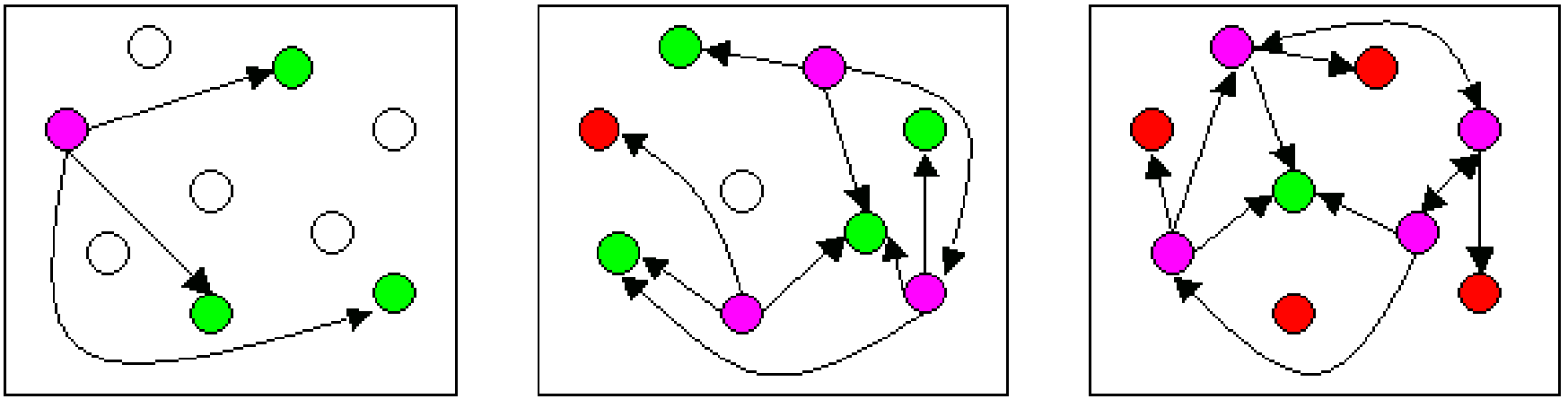
Properties:

PB1: *Probabilistic validity:* There is a given probability such that for any p_i and p_j that are correct, every message broadcast by p_i is eventually delivered by p_j with this probability.

PB2: *No duplication:* No message is delivered more than once.

PB3: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

Module 3.7 Interface and properties of probabilistic broadcast.



(a) (b) (c)

Figure 3.5. Gossip Dissemination.

Algorithm 3.9 Eager Probabilistic Broadcast.

Implements:
ProbabilisticBroadcast (pb).

Uses:
unreliablePointToPointLinks (up2p).

upon event ⟨ *Init* ⟩ **do**
delivered := ∅;

function pick-targets (fanout) **returns** set of processes **do**
targets := ∅;
while | targets | < fanout **do**
candidate := random (Π);
if candidate ∉ targets ∧ candidate ≠ self **then**
targets := targets ∪ { candidate };
return targets;

procedure gossip (msg) **do**
forall t ∈ pick-targets (fanout) **do**
trigger ⟨ *up2pSend*, t, msg ⟩;

upon event ⟨ *pbBroadcast*, m ⟩ **do**
gossip ([GOSSIP, s_m, m, maxrounds−1]);

upon event ⟨ *up2pDeliver*, p_i, [GOSSIP, s_m, m, r] ⟩ **do**
if $m \notin$ delivered **then**
delivered := delivered ∪ {m}
trigger ⟨ *pbDeliver*, s_m, m ⟩;
if $r > 0$ **then** gossip ([GOSSIP, s_m, m, $r-1$]);

Implements:
ProbabilisticBroadcast (pb).

Uses:
UnreliablePointToPointLinks (up2p), UnreliableBroadcast (unb).

upon event ⟨ *Init* ⟩ **do**
$\forall_{p_i \in \Pi}$ delivered[p_i] := 0; lsn := 0; pending := ∅; stored := ∅;

procedure deliver-pending (s) **do**
$\forall_x$: [DATA, s, x, sn_x] *in* pending such that sn_x = delivered[s]+1 **do**
delivered[s] := delivered[s]+1; pending := pending \ { [DATA, s, x,sn_x]};
trigger ⟨ *pbDeliver*, s, x ⟩;

procedure gossip (msg) **do**
forall t ∈ pick-targets (fanout) **do**
trigger ⟨ *up2pSend*, t, msg ⟩;

upon event ⟨ *pbBroadcast*, m ⟩ **do**
lsn := lsn+1;
trigger ⟨ *unBroadcast*, [DATA, self, m, *lsn*] ⟩;

upon event ⟨ *unDeliver*, p_i, [DATA, s_m, m, sn_m] ⟩ **do**
if random > store-threshold **then** stored := stored ∪ { [DATA, s_m, m,sn_m] };
if sn_m = delivered[s_m]+1 **then**
delivered[s_m] := delivered[s_m]+1;
trigger ⟨ *pbDeliver*, s_m, m ⟩;
else
pending := pending ∪ { [DATA, s_m, m, sn_m] };
∀ seqnb ∈ [$s_m - 1$, delivered[s_m]+1] **do**
gossip ([REQUEST, self, s_m, seqnb, maxrounds−1]);

upon event ⟨ *up2pDeliver*, p_j, [REQUEST, p_i, s_m, sn_m, r] ⟩ **do**
if [DATA, s_m, m, sn_m] ∈ stored **then**
trigger ⟨ *upp2pSend*, p_i, [DATA, s_m, m, sn_m] ⟩;
else if $r > 0$ **then** gossip ([REQUEST, p_i, s_m, sn_m, $r - 1$]);

upon event ⟨ *up2pDeliver*, p_j, [DATA, s_m, m, sn_m] ⟩ **do**
if sn_m = delivered[s_m]+1 **then**
delivered[s_m] := delivered[s_m]+1;
trigger ⟨ *pbDeliver*, s_m, m ⟩;
deliver-pending (s_m);
else
pending := pending ∪ { [DATA, s_m, m, sn_m] };

Algorithm 3.11: A randomized partial membership algorithm.

Implements:
Probabilistic Partial Membership (ppm).

Uses:
unreliablePointToPointLinks (up2p).

upon event ⟨ *Init* ⟩ **do**
view := set of known group members;
subs := ∅; unsubs := ∅;

every T units of time **do**
for 1 to fanout **do**
target := random (view);
trigger ⟨ *upp2pSend*, target, [GOSSIP, subs, unsubs] ⟩;

upon ⟨ *ppmJoin* ⟩ **do**
subs := subs ∪ { self };

upon ⟨ *ppmLeave* ⟩ **do**
unsubs := unsubs ∪ { self };

upon event ⟨ *up2pDeliver*, p_i, [GOSSIP, s, u] ⟩ **do**
view := view \ u;
view := view ∪ s \ { self };
unsubs := unsubs ∪ u;
subs := subs ∪ s \ { self };
//trim variables
while | view | > viewsz **do**
target := random (view);
view := view \ { target };
subs := subs ∪ { target };
while | unsubs | > unsubssz **do** unsubs := unsubs \ { random(unsubs) };
while | subs | > subssz **do** subs := subs \ { random(subs) };

Module:

Name: (regular) Consensus (c).

Events:

Request: $\langle$ *cPropose*, v $\rangle$: Used to propose a value for consensus.

Indication: $\langle$ *cDecide*, v $\rangle$: Used to indicate the decided value for consensus.

Properties:

C1: *Termination:* Every correct process eventually decides some value.

C2: *Validity:* If a process decides v, then v was proposed by some process.

C3: *Integrity:* No process decides twice.

C4: *Agreement:* No two correct processes decide differently.

Module 5.1 Interface and properties of consensus.

Algorithm 5.1: Flooding consensus algorithm.

Implements:
Consensus (c);

Uses:
BestEffortBroadcast (beb);
PerfectFailureDetector ($\mathcal{P}$);

upon event $\langle$ *Init* $\rangle$ **do**
correct := correct-last-round := Π;
proposal-set := correct-this-round := $\emptyset$;
decided := $\perp$;
round := 1;

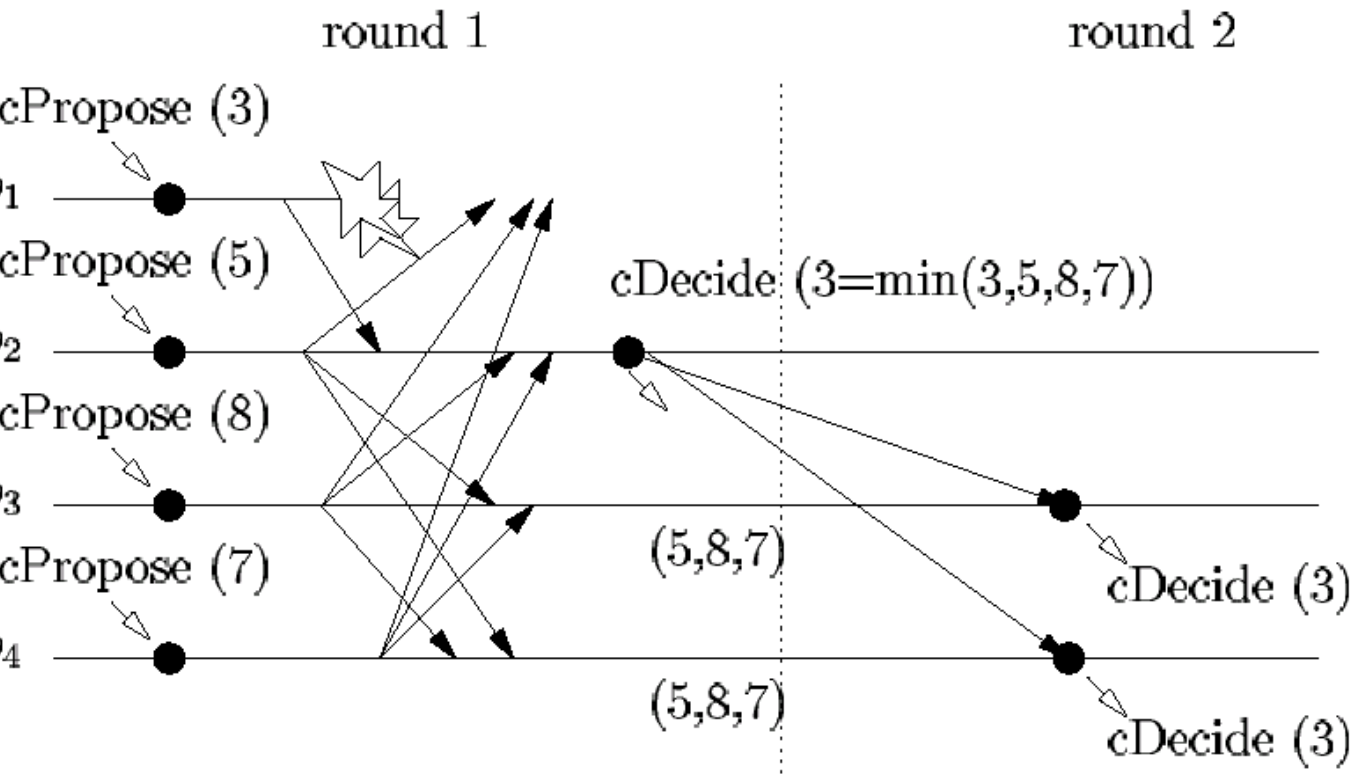
upon event $\langle$ *crash*, p_i $\rangle$ **do**
correct := correct $\setminus \{p_i\}$;

upon event $\langle$ *cPropose*, v $\rangle$ **do**
proposal-set := $\{v\}$;
trigger $\langle$ *bebBroadcast*, [MySet, round, proposal-set] $\rangle$;

upon event $\langle$ *bebDeliver*, p_i, [MySet, round, set] $\rangle$ **do**
correct-this-round := correct-this-round $\cup$ $\{p_i\}$;
proposal-set := proposal-set $\cup$ set;

upon correct $\subset$ correct-this-round **do**
round := round +1;
if (correct-this-round = correct-last-round) $\wedge$ (decided = $\perp$) **then**
trigger $\langle$ *cDecide*, min (proposal-set) $\rangle$;
trigger $\langle$ *bebBroadcast*, [Decided, round, min (proposal-set)] $\rangle$;
else
correct-last-round := correct-this-round;
correct-this-round := $\emptyset$;
trigger $\langle$ *bebBroadcast*, [MySet, round, proposal-set] $\rangle$;

upon event $\langle$ *bebDeliver*, p_i, [Decided, round, v] $\rangle$ $\wedge$ (decided = $\perp$) **do**
decided := v;
trigger $\langle$ *cDecide*, v $\rangle$;
trigger $\langle$ *bebBroadcast*, [Decided, round + 1, min (proposal-set)] $\rangle$;



igure 5.1. Sample execution of the flooding consensus algorithr

Algorithm 5.2 A hierarchical consensus algorithm.

Implements:
Consensus (c);

Uses:
BestEffortBroadcast (beb);
PerfectFailureDetector ($\mathcal{P}$);

upon event $\langle$ *Init* $\rangle$ **do**
suspected := $\emptyset$;
round := 1;
proposal := *nil*;
for i = 1 **to** N **do** pset[i] := p_i;
for i = 1 **to** N **do** delivered[*round*] := false;
for i = 1 **to** N **do** broadcast[*round*] := false;

upon event $\langle$ *crash*, p_i $\rangle$ **do**
suspected := suspected $\cup\{p_i\}$;

upon event $\langle$ *cPropose*, v $\rangle$ **do**
proposal := v;

upon (pset[round] = self) $\wedge$ (proposal $\neq$ *nil*) $\wedge$ (broadcast[*round*] = false) **do**
trigger $\langle$ *cDecide*, proposal $\rangle$;
trigger broadcast[*round*] := true;
trigger $\langle$ *bebBroadcast*, proposal $\rangle$;

upon (pset[round] $\in$ suspected) $\vee$ (delivered[*round*] = true) **do**
round := round + 1;

upon event $\langle$ *bebDeliver*, pset[round],value $\rangle$ **do**
if self.id > round **then**
proposal := value;
delivered[*round*] := true;

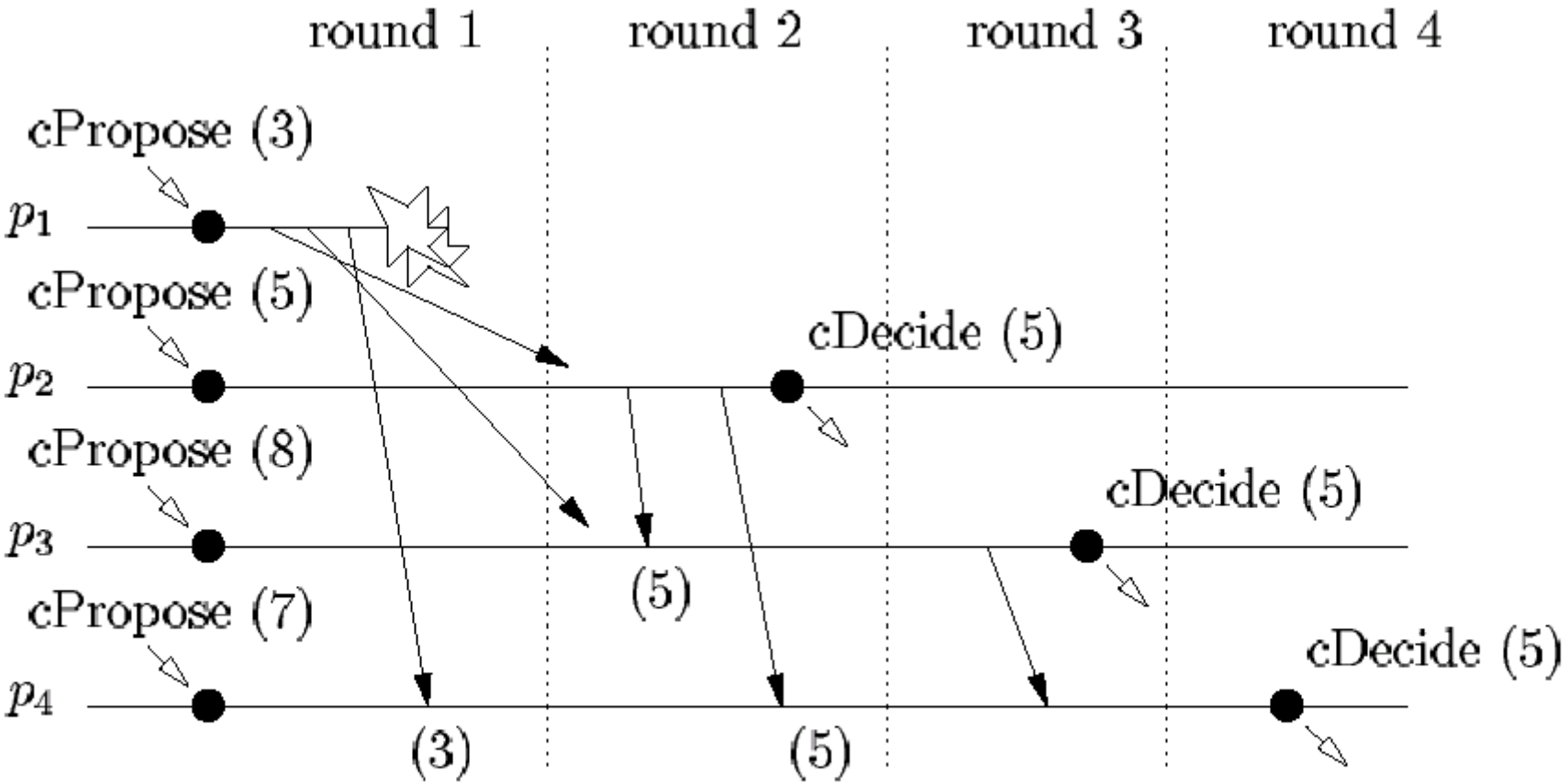


Figure 5.2. Sample execution of hierarchical consensus.

Module:

Name: UniformConsensus (uc).

Events:

⟨ *ucPropose*, v ⟩, ⟨ *ucDecide*, v ⟩: with the same meaning and interface of the consensus interface.

Properties:

C1-C3: from consensus.

C4': *Uniform Agreement:* no two processes decide differently..

Module 5.2 Interface and properties of uniform consensus.

Algorithm 5.3 A flooding uniform consensus algorithm.

Implements:
UniformConsensus (c);

Uses:
BestEffortBroadcast (beb).
PerfectFailureDetector ($\mathcal{P}$);

upon event $\langle$ *Init* $\rangle$ **do**
correct := Π;
round := 1;
for i = 1 **to** N **do** set[i] := delivered[i] := $\emptyset$;
proposal-set := $\emptyset$;
decided := false;

upon event $\langle$ *crash*, p_i $\rangle$ **do**
correct := correct $\setminus \{p_i\}$;

upon event $\langle$ *ucPropose*, v $\rangle$ **do**
proposal-set := $\{v\}$;
trigger $\langle$ *bebBroadcast*, [MySet, round, proposal-set] $\rangle$;

upon event $\langle$ *bebDeliver*, p_i, [MySet, round, newSet] $\rangle \wedge (p_i \in$ correct) **do**
set[round] := set[round] $\cup$ newSet;
delivered[round] := delivered[round] $\cup \{p_i\}$;

upon (correct $\subseteq$ delivered[round]) $\wedge$ (decided = false) **do**
if round = N **then**
decided := true;
trigger $\langle$ *ucDecide*, min(proposal-set $\cup$ set[round]) $\rangle$;
else
proposal-set := proposal-set $\cup$ set[round];
round := round + 1;
trigger $\langle$ *bebBroadcast*, [MySet, round, proposal-set] $\rangle$;

Implements:
UniformConsensus (uc);

Uses:
PerfectPointToPointLinks (pp2p);
ReliableBroadcast (rb).
BestEffortBroadcast (beb).
PerfectFailureDetector ($\mathcal{P}$);

upon event $\langle$ *Init* $\rangle$ **do**
proposal := decided := $\perp$;
round := 1;
suspected := ack-set := $\emptyset$;
for i = 1 **to** N **do** pset[i] := p_i;

upon event $\langle$ *crash*, p_i $\rangle$ **do**
suspected := suspected $\cup$ $\{p_i\}$;

upon event $\langle$ *ucPropose*, v $\rangle$ **do**
proposal := v;

upon (pset[round] = self) $\wedge$ (proposal $\neq \perp$) $\wedge$ (decided = $\perp$) **do**
trigger $\langle$ *bebBroadcast*, [PROPOSE, round, proposal] $\rangle$;

upon event $\langle$ *bebDeliver*, p_i, [PROPOSE, round, v] $\rangle$ **do**
proposal := v;
trigger $\langle$ *pp2pSend*, p_i, [ACK, round] $\rangle$;
round := round + 1;

upon event (pset[round] $\in$ suspected) **do**
round := round + 1;

upon event $\langle$ *pp2pDeliver*, p_i, [ACK, round] $\rangle$ **do**
ack-set := ack-set $\cup$ $\{p_i\}$;

upon event (ack-set $\cup$ suspected = Π) **do**
trigger $\langle$ *rbBroadcast*, [DECIDED, proposal] $\rangle$;

upon event $\langle$ *rbDeliver*, p_i, [DECIDED, v] $\rangle$ $\wedge$ (decided = $\perp$) **do**
decided := v;
trigger $\langle$ *ucDecide*, v $\rangle$;

Module:

Name: Randomized Consensus (rc).

Events:

Request: $\langle$ *rcPropose*, v $\rangle$: Used to propose a value for consensus.

Indication: $\langle$ *rcDecide*, v $\rangle$: Used to indicate the decided value for consensus.

Properties:

RC1: *Termination:* With probability 1, every correct process decides some value.

RC2: *Validity:* If a process decides v, then v was proposed by some process.

RC3: *Integrity:* No process decides twice.

RC4: *Agreement:* No two correct processes decide differently.

Module 5.6 Interface and properties of randomized consensus.

Implements:
Randomized Consensus (rc);

Uses:
ReliableBroadcast (rb).
BestEffortBroadcast (beb).

upon event ⟨ *Init* ⟩ **do**
decided := ⊥; estimate := ⊥; round := 0;
for i = 1 **to** N **do** val[i] := ⊥;
val := ∅;

upon event ⟨ *rcPropose*, v ⟩ **do**
trigger ⟨ *bebBroadcast*, [INIVALUE, v] ⟩;
estimate := v; round := round +1;
val:= val ∪ {v};
trigger ⟨ *bebBroadcast*, [PHASE1, round, v] ⟩;

upon event ⟨ *bebDeliver*, p_i, [INIVAL, v] ⟩ **do**
val:= val ∪ {v};

upon event ⟨ *bebDeliver*, p_i, [PHASE1, r, v] ⟩ **do**
phase1[r] := phase1[r] ⊕ v;

upon (decided=⊥ ∧ |phase1[round]| > $N/2$) **do**
if ∃$_v$: ∀x ∈ phase1[round]: $x = v$ **then** estimate := v;
else estimate := ⊥;
trigger ⟨ *bebBroadcast*, [PHASE2, round, estimate] ⟩;

upon event ⟨ *bebDeliver*, p_i, [PHASE2, r, v] ⟩ **do**
phase2[r] := phase2[r] ⊕ v;

upon (decided=⊥ ∧ |phase2[round]| > $N/2$) **do**
if ∃$v \neq \perp$: ∀x ∈ phase1[round]: $x = v$ **then**
decided := v;
trigger ⟨ *rbBroadcast*, [DECIDED, round, decided] ⟩;
else
if ∃v ∈ phase2[round]: $v \neq \perp$ **then** estimate := v;
else estimate := random(val);
round := round +1; // start one more round
trigger ⟨ *rbBroadcast*, [PHASE1, round, estimate] ⟩;

upon event ⟨ *rbDeliver*, p_i, [PHASE2, r, v] ⟩ **do**
decided := v;
trigger ⟨ *rcDecided*, decided ⟩;

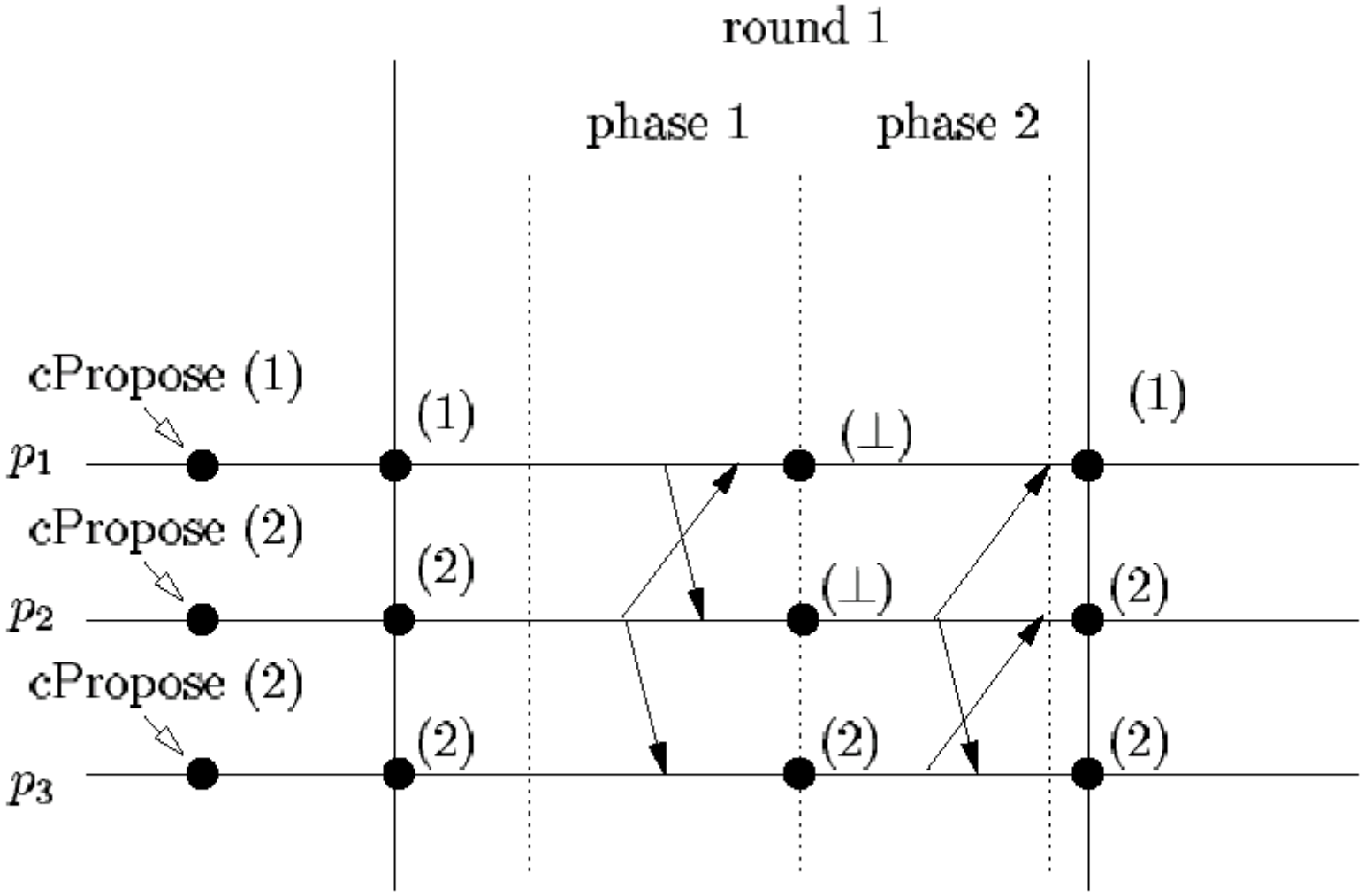


Figure 5.3. Role of randomization.