Composition Model and its code

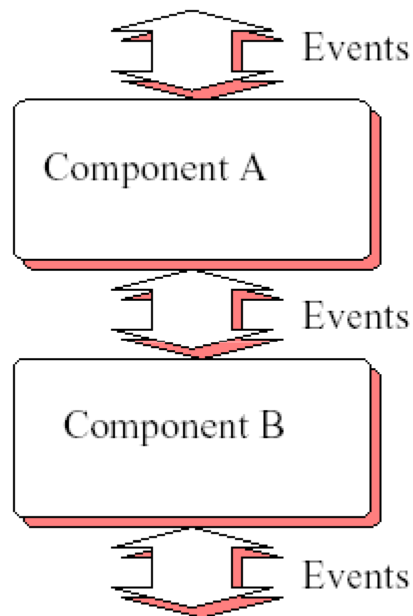


Figure 1.1. Composition model

The code of each component looks like this:

---

upon event ⟨ *Event1*, att_1^1, att_1^2, ... ⟩ **do**
  something
  // send some event
  trigger ⟨ *Event2*, att_2^1,att_2^2, ... ⟩;

upon event ⟨ *Event3*, att_3^1, att_3^2, ... ⟩ **do**
  something else
  // send some other event
  trigger ⟨ *Event4*, att_4^1, att_4^2, ... ⟩;

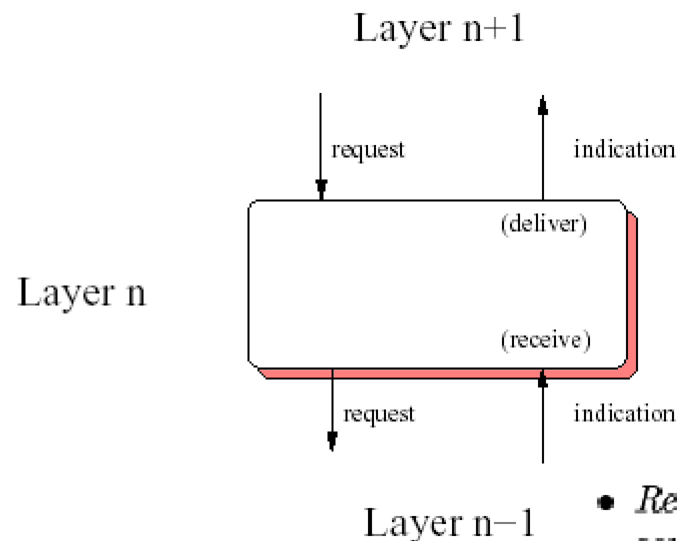


Figure 1.2. Layering

- *Request* events are used by a component to request a service from another component: for instance, the application layer might trigger a *request* event at a component in charge of broadcasting a message to a set of processes in a group with some reliability guarantee, or proposing a value to be decided on by the group.
- *Confirmation* events are used by a component to confirm the completion of a request. Typically, the component in charge of implementing a broadcast will confirm to the application layer that the message was indeed broadcast or that the value suggested has indeed been proposed to the group: the component uses here a *confirmation* event.

- *Indication* events are used by a given component to *deliver* information to another component. Considering the broadcast example above, at every process that is a destination of the message, the component in charge of implementing the actual broadcast primitive will typically perform some processing to ensure the corresponding reliability guarantee, and then use an *indication* event to deliver the message to the application layer. Similarly, the decision on a value will be indicated with such an event.

Module:

Name: Print (lpr).

Events:

Request: ⟨ *lprPrint*, rqid, string ⟩**:** Requests a string to be printed. The token rqid is an identifier of the request.

Confirmation:⟨ *lprOk*, rqid ⟩**:** Used to confirm that the printing request with identifier rqid succeeded.

Module 1.1 Interface of a printing module.

Algorithm 1.1 Printing service.

Implements:

Print (lpr).

upon event ⟨ *lprPrint*, rqid, string ⟩ **do**

print string;

trigger ⟨ *lprOk*, rqid ⟩;

Module:

Name: BoundedPrint (blpr).

Events:

Request: ⟨ *blprPrint*, rqid, string ⟩: Request a string to be printed. The token rqid is an identifier of the request.

Confirmation:⟨ *blprStatus*, rqid, status ⟩: Used to return the outcome of the printing request: Ok or Nok.

Indication:⟨ *blprAlarm* ⟩: Used to indicate that the threshold was reached.

Module 1.2 Interface of a bounded printing module.

Algorithm 1.2 Bounded printer based on (unbounded) print service.

Implements:
BoundedPrint (blpr).

Uses:
Print (lpr).

upon event ⟨ *Init* ⟩ **do**
bound := PredefinedThreshold;

upon event ⟨ *blprPrint*, rqid, string ⟩ **do**
if bound > 0 **then**
bound := bound-1;
trigger ⟨ *lprPrint*, rqid, string ⟩;
if bound $= 0$ **then trigger** ⟨ *blprAlarm* ⟩;
else
trigger ⟨ *blprStatus*, rqid, Nok ⟩;

upon event ⟨ *lprOk*, rqid ⟩ **do**
trigger ⟨ *blprStatus*, rqid, Ok ⟩;

bound:=bound+1

Model of a Computation

- Model of Synchrony
  - Processes
  - Communication Channels
    - Reliable communication
    - Unreliable communication
- Model of Failures

Processes

Processes and messages

Automata

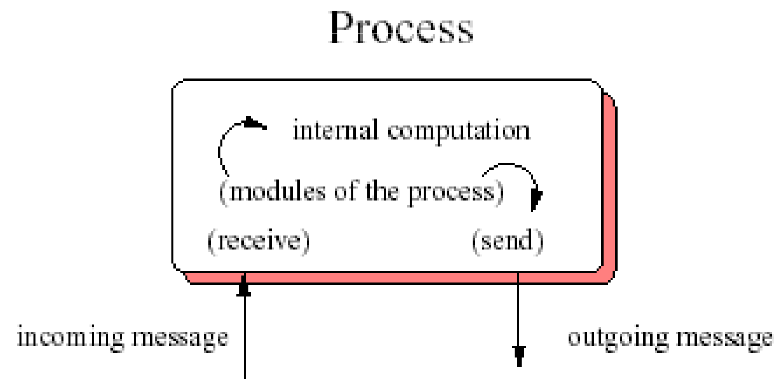


Figure 2.1. Step of a process

Synchronous Processing

Characterized by two properties

1. Synchronous processing
   - Known Upper Bound on the time taken by a process to execute a basic step
2. Synchronous physical clocks
   - Known Upper Bound on drift of a local clock wrt real time

Abstractions

- Specify problems
- E.g. Mutual Exclusion
- Specification of the correctness
  - Safety:
    - No bad things happen
  - Liveness:
    - Eventually something good happens

Distributed Algorithms/protocols

- Implement Abstractions
- A set of automata one per process
- Correctness formally proved

Model of Failures

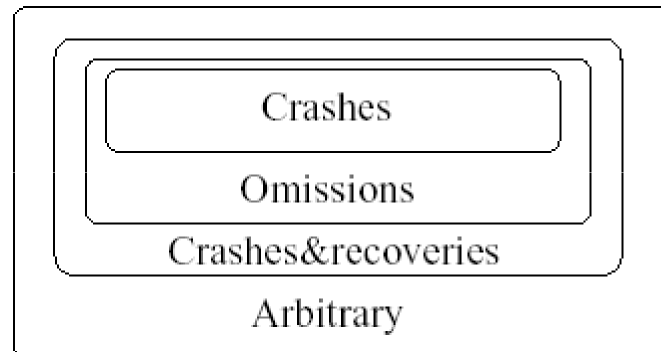


Figure 2.2. Failure modes of a process

Models of Synchrony (Communication)

- Synchronous
- Asynchronous
- Partially synchronous

Synchronous Communication

Known Upper Bound on the time taken by a message to reach a destination

Synchronous System

Characterized by three properties

1. Synchronous processing
   - Known Upper Bound on the time taken by a process to execute a basic step
2. Synchronous Communication
   - Known Upper Bound on the time taken by a message to reach a destination
3. Synchronous physical clocks
   - Known Upper Bound on drift of a local clock wrt real time

Services provided in Synchronous systems

- Timed failure detection
- Measure of transit delay
- Coordination based on time
- Worst case performance (e.g. response time of a service in case of failures)
- Synchronized clocks

Major problem the coverage of the synchrony assumption!!!!
This turns out in difficulty of building a system where timing
Assumptions hold with high probability

Partial (eventual) synchrony

- Generally distributed systems are synchronous most of the time and then they experience bounded asynchrony periods
- One way to capture partial synchrony is "eventual synchrony" I.e., there is an unknown time t after which the system becomes synchronous
- This assumption captures the fact that the system does not behave always as synchronous
- It does not mean that
  - After t all the system (including hardware, software and network components) becomes synchronous forever
  - The system start asynchronous and then after some (may be long) time it becomes synchronous

What do we expect from partial synchrony

- There is a period of synchrony long enough to terminate the distributed algorithm

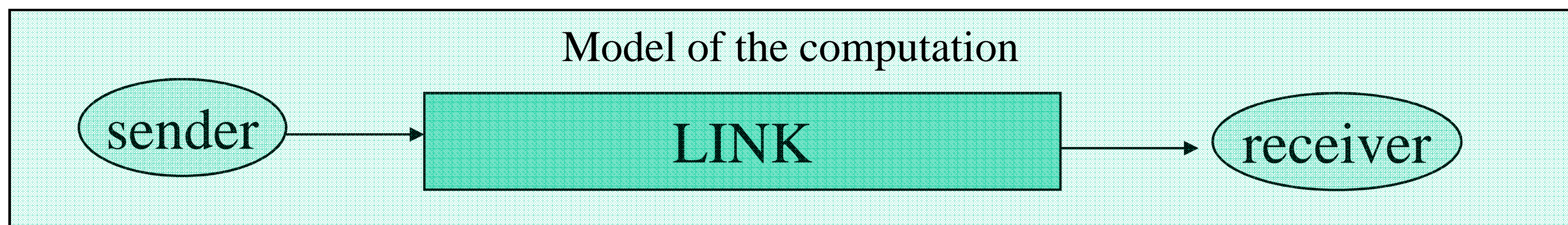
Reliable vs unreliable communication

- Each message sent arrive to a destination (i.e., no message is lost)
- Messages can be lost

LINKS

Links

- Model of the computation
  - Two processes (sender and receiver)
  - Messages
    - can be lost
    - experience an unpredictable time to reach the destination
  - Processes
    - can crash
    - The time taken by each process to execute an operation is bounded (such a bound can be unknown)



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*.

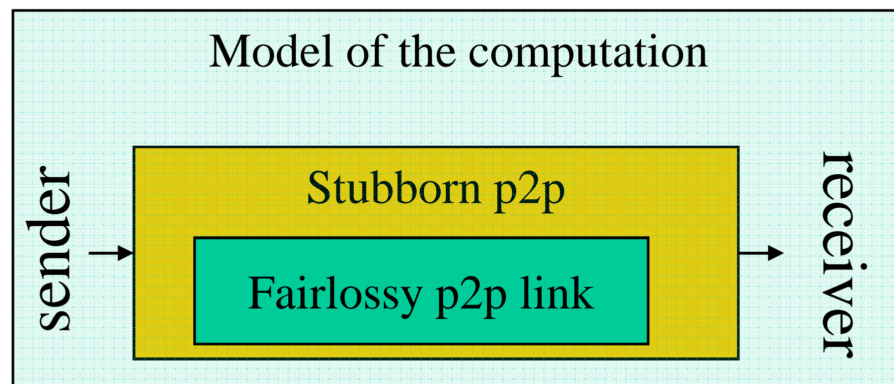
Indication:⟨ *sp2pDeliver*, *src*, *m* ⟩: Used to deliver message m sent by process *src*.

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*.

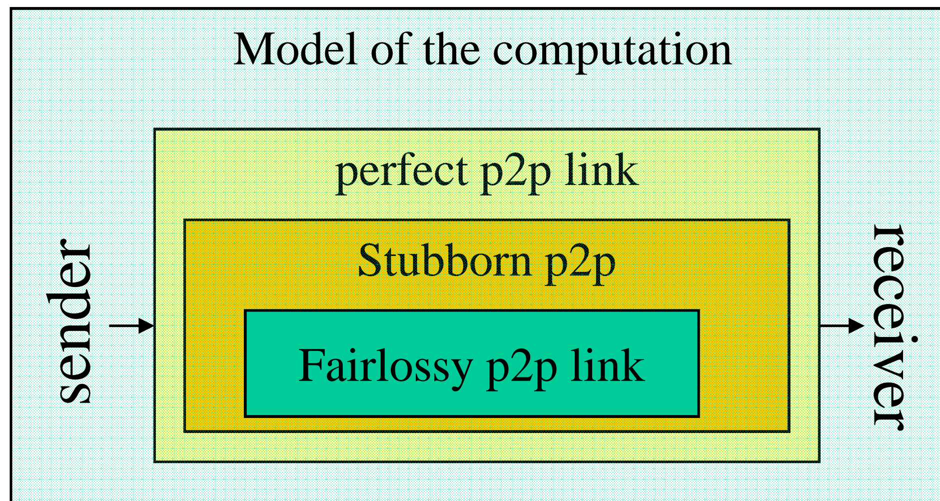
Properties:

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 ∉ delivered then
        delivered := delivered ∪{m};
        trigger ⟨ pp2pDeliver, src, m ⟩;
```

Links: conclusion

- Relation with existing protocols stacks
- Performance issues

Failure Detectors

Models of Synchrony wrt timing assumptions

- Synchronous
  - timing assumptions are explicit either on
    - Bounds on process executions and communication channels, or
    - Existence of a common global clock, or
    - both
- Asynchronous
  - (there are no timing assumptions)

Models of Synchrony wrt timing assumptions

- Partial synchrony requires abstract timing assumptions (after an unknown time t the system becomes synchronous)
- Two choices:
  - Put assumption on the system model (including links and processes)
  - Create a separate abstractions that encapsulates those timing assumptions
- Note: manipulating time inside a protocol/algorithm is complex and the correctness proof become very involved

Failure Detector Abstraction

- Software module to be used together with process and link abstractions
- It encapsulates timing assumptions of a either partially synchronous or fully synchronous system
- The stronger are the timing assumption, the more accurate the information provided by a failure detector will be.
- Described by two properties:
  - Accuracy
  - Completeness

Perfect Failure detectors (P)

- System model
  - synchronous system
  - crash failures
- Using a own clock and the bounds of the synchrony model, a process can infer if another process has crashed

Perfect failure detectors (P)

- Characterized by two 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.

- Indication crash(p_i) used to notify that p_i has crashed

- Equivalence between Perfect failure detector and synchronous system model

Algorithm 2.4 Exclude on Timeout.

```
Implements:
    PerfectFailureDetector (P).

Uses:
    PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    alive := Π;
    detected := ∅;
    startTimer (TimeDelay);

upon event ⟨ Timeout ⟩ do
    forall p_i ∈ Π do
        if (p_i ∉ alive) and (p_i ∉ detected) then
            detected := detected ∪ {p_i};
            trigger ⟨ crash | p_i ⟩;
        trigger ⟨ pp2pSend | p_i, [HEARTBEAT] ⟩;
    alive := ∅;
    startTimer (TimeDelay);

upon event ⟨ pp2pDeliver | src, [HEARTBEAT] ⟩ do
    alive := alive ∪ {src};
```

Discuss message pattern and correctness (note that links are perfect)

Module:

Name: LeaderElection (le).

Events:

Indication: ⟨ *leLeader* | p_i ⟩: Used to indicate that process p_i is now the leader.

Properties:

LE1: Either there is no correct process, or some correct process is eventually the leader.

LE2: If a process is leader, then all previously elected leaders have crashed.

Module 2.6 Interface and properties of leader election.

This function O associates, to every process, those that precede it in the ranking. A process can only become leader if those that precede it have crashed. Think of the function as representing the royal ordering in a monarchical system. The prince becomes leader if and only if the queen dies. If the prince dies, maybe his little sister is the next on the list, etc. Typically, we would assume that $O(p_1) = \emptyset$, $O(p_2) = \{p_1\}$, $O(p_3) = \{p_1, p_2\}$, and so forth. The order in this case is $p_1; p_2; p_3; ...; p_k : p_{k+1}$.

Algorithm 2.5 Monarchical Leader Election.

Implements:
LeaderElection (le);

Uses:
PerfectFailureDetector (P);

upon event ⟨ *Init* ⟩ **do**
suspected := $\emptyset$;

upon event ⟨ *crash* | p_i ⟩ **do**
suspected := suspected $\cup\{p_i\}$;

upon event O(self) $\subseteq$ suspected **do**
trigger ⟨ *leLeader* | self ⟩;

Eventually perfect failure detectors (◊P)

- System model
  - partial synchrony
  - Crash failures
  - Perfect point-to-point links
- There is a (unknown) time t after that crashes can be accurately detected
- Before t the systems behaves as an asynchronous one
- The failure detector makes mistake in that periods assuming correct processes as crashed.
- The notion of detection becomes suspicious

Module:

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

Events:

Indication: ⟨ *suspect*, p_i ⟩: Used to notify that process p_i is suspected to have crashed.

Indication: ⟨ *restore*, p_i ⟩: 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.

Basic constructions rules of an eventually perfect FD

- Use timeouts to suspect processes that did not sent expected messages
- A suspect may be wrong. A process p may suspect another one q because the chosen timeout was too short
- P is ready to reverse its judgment as soon as it receives a message from q (updating also the timeout value)
- If q has actually crashed, p does not change its judgment anymore.

Algorithm 2.6 Increasing Timeout.

```
Implements:
    EventuallyPerfectFailureDetector (◇P).

Uses:
    PerfectPointToPointLinks (pp2p).

upon event ⟨ Init ⟩ do
    alive := Π;
    suspected := ∅;
    period := TimeDelay;
    startTimer (period);

upon event ⟨ Timeout ⟩ do
    if (alive ∩ suspected) ≠ ∅ then
        period := period + Δ;
    forall p_i ∈ Π do
        if (p_i ∉ alive) ∧ (p_i ∉ suspected) then
            suspected := suspected ∪ {p_i};
            trigger ⟨ suspect | p_i ⟩;
        else if (p_i ∈ alive) ∧ (p_i ∈ suspected) then
            suspected := suspected \ {p_i};
            trigger ⟨ restore | p_i ⟩;
        trigger ⟨ pp2pSend | p_i, [HEARTBEAT] ⟩;
    alive := ∅;
    startTimer (period);

upon event ⟨ pp2pDeliver | src, [HEARTBEAT] ⟩ do
    alive := alive ∪ {src};
```

Correctness

- Strong completeness. If a process crashes, it will stop to send messages. Therefore the process will be suspected by any correct process and no process will revise the judgement.
- Eventual strong accuracy. After time T the system becomes synchronous. i.e., after that time a message sent by a correct process p to another one q will be delivered within a bounded time. If p was wrongly suspected by q, then q will revise its suspicious.

Eventual leader election (Ω)

- Agree on a process that has not failed
- This process could act as a *coordinator*

Module:

Name: EventualLeaderDetector (Ω).

Events:

Indication: ⟨ *trust*, p_i ⟩: Used to notify that process p_i is trusted to be leader.

Properties:

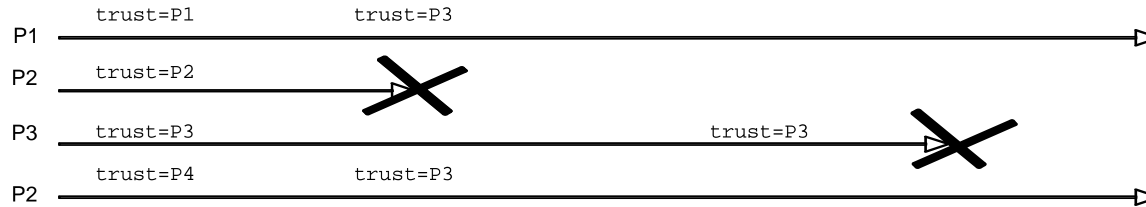
CD1: *Eventual accuracy:* There is a time after which every correct process trusts some correct process.

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

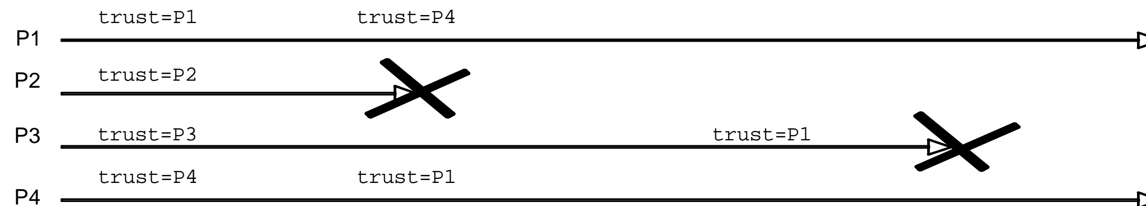
Module 2.6 Interface and properties of the eventual leader detector.

Study of Properties

Run 1



Run 2



	Run 1	Run 2
Eventual Accuracy	Not verified	Verified
Eventual Agreement	Verified	Not verified

Eventual leader election (Ω)

- Using Crash-stop process abstraction
  - Obtained directly by <>P by using a deterministic rule on processes that are not suspected by <>P
  - trust the process with the highest identifier among all processes that are not suspected by <>P

- Assume the existence of a correct process (otherwise Ω cannot be built)

Eventual leader election (Ω)

- System model
  - Crash-Recovery
  - Partial synchrony
- Under this assumption, a correct process means:
  1. A process that does not crash or
  2. A process that crashes, eventually recovers and never crashes again

Ω With crash recovery, fair lossy links and timeouts

Algorithm 2.7 Elect Lower Epoch (initialization and recovery)

Implements:
EventualLeaderDetector (Ω).

Uses:
FairLossPointToPointLinks (flp2p).

```
upon event ⟨ Init ⟩ do
    leader := p1;
    trigger ⟨ trust | leader ⟩;
    period := TimeDelay;
    epoch := 0;
    store(epoch);
    forall pi ∈ Π do
        trigger ⟨ flp2pSend | pi, [HEARTBEAT, ep
    candidateset := ∅;
    startTimer (period);

upon event ⟨ Recovery ⟩ do
    leader := p1;
    trigger ⟨ trust | leader ⟩;
    period := TimeDelay;
    retrieve(epoch);
    epoch := epoch + 1;
    store(epoch);
    forall pi ∈ Π do
        trigger ⟨ flp2pSend | pi, [HEARTBEAT, ep
    candidateset := ∅;
    startTimer (period);

upon event ⟨ Timeout ⟩ do
    newleader = select(candidateset);
    if (leader ≠ newleader) then
        period := period + Δ;
        leader := newleader;
        trigger ⟨ trust | leader ⟩;
    forall pi ∈ Π do
        trigger ⟨ flp2pSend | pi, [HEARTBEAT, epoch] ⟩;
    candidateset := ∅;
    startTimer (period);

upon event ⟨ flp2pDeliver | src, [HEARTBEAT, epc] ⟩ do
    if exists (s, e) ∈ candidateset such that (s=src) ∧ (e<epc) then
        candidateset := candidateset \ {(s, e)};
    candidateset := candidateset ∪ {(src, epc)};
```

Ω With crash recovery, fair lossy links and timeouts

- Epoch number
- *candidate* list of "non-crashed" processes of process P potentially candidates to be a leader
- *Select* (*candidate*), deterministic function returning one process among all processes in possible (the same at each process)
- Deterministic rule: return the process with the lowest epoch number and among the ones with the same epoch number the one with the lowest identifier.

- avoid that the leader will be a process that goes up and down infinitely many times

Correctness

Eventual accuracy

- Assume by contradiction a correct process Pi trusts permanently a faulty one Pj. Two cases
  1. Pj eventually crashes and never recovers again…stop sending heartbeats!
  2. Pj keeps crashing and recoveries forever….
     1. Pi stops receiving messages from Pj……… a process stays up only for finite periods of time (fair lossy link)
     2. Pi continues receiving messages from Pj….. epoch number increases forever

Broadcast

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:

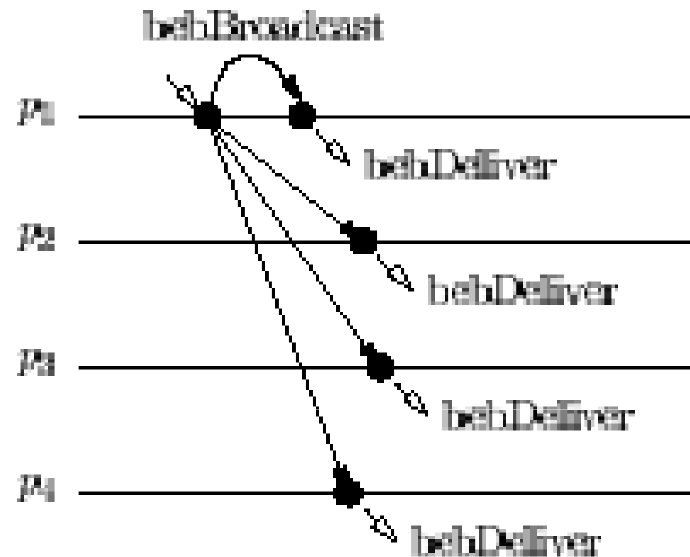
BEB1: *Best-effort validity:* For any two processes p_i and p_j. If p_i and p_j are correct, then every message broadcast by p_i is eventually delivered by p_j.

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.

Best-Effort Broadcast



Beb ensures the delivery of messages as long as the sender does not fail

If the sender fails processes may disagree on whether or not deliver the Message (perfect links does not ensure delivery if the sender fail!)

Best-Effort Broadcast

System model
• Asynchronous communication system
• crash failures

Algorithm 3.1 Basic Broadcast.

Implements:
BestEffortBroadcast (beb).

Uses:
PerfectPointToPointLinks (pp2p).

upon event ⟨ *bebBroadcast*, m ⟩ **do**
$\forall_{p_i \in \Pi}$:
trigger ⟨ *pp2pSend*, p_i, m ⟩;

upon event ⟨ *pp2pDeliver*, p_i, m ⟩ **do**
trigger ⟨ *bebDeliver*, p_i, m ⟩;

Regular Reliable Broadcast

- Stronger form of reliability than BeB
- Liveness: agreement
- Implementation: relaying messages

Module:

Name: (regular)ReliableBroadcast (rb).

Events:

Request: ⟨ *rbBroadcast*, m ⟩: Used to broadcast message m.

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

Properties:

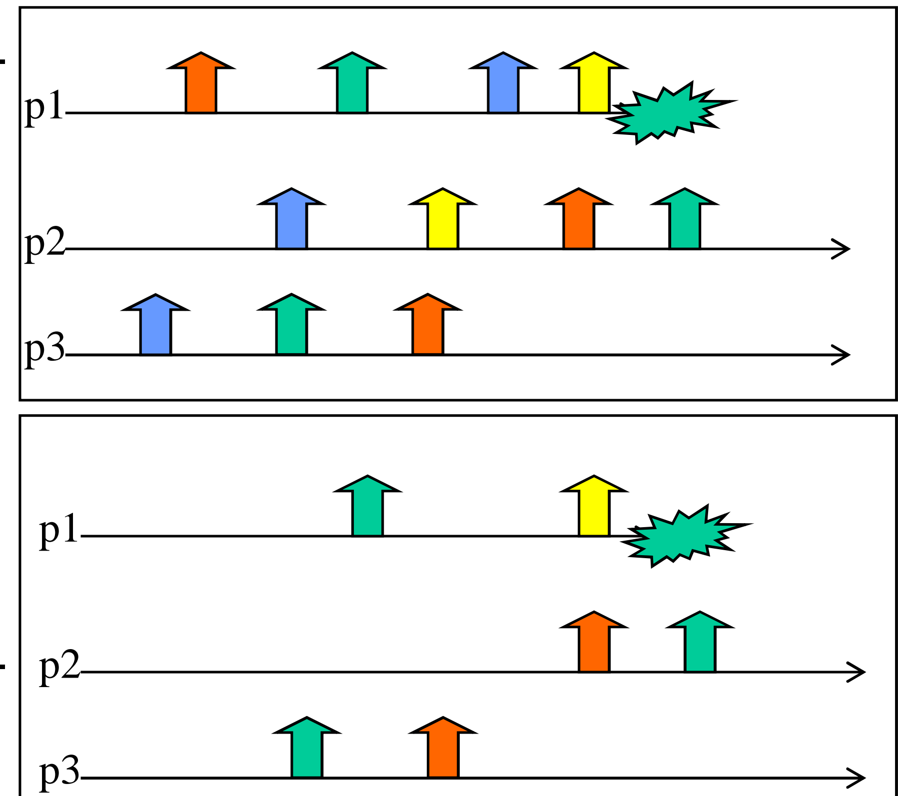
RB1: *Validity:* If a correct process p_i broadcasts a message m, then p_i eventually delivers m.

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.



Regular Reliable Broadcast

Algorithm 3.2 Lazy reliable broadcast.

```
Implements:
    ReliableBroadcast (rb).

Uses:
    BestEffortBroadcast (beb).
    PerfectFailureDetector (P).

upon event ⟨ Init ⟩ do
    delivered := ∅;
    correct := Π;
    ∀_{p_i ∈ Π} : from[p_i] := ∅;

upon event ⟨ rbBroadcast, m ⟩ do
    trigger ⟨ bebBroadcast, [DATA, self, m] ⟩;

upon event ⟨ bebDeliver, p_i, [DATA, s_m, m] ⟩ do
    if m ∉ delivered then
        delivered := delivered ∪ {m}
        trigger ⟨ rbDeliver, s_m, m ⟩;
        from[p_i] := from[p_i] ∪ {[s_m, m]}
        if p_i ∉ correct then
            trigger ⟨ bebBroadcast, [DATA, s_m, m] ⟩;

upon event ⟨ crash, p_i ⟩ do
    correct := correct \ {p_i}
    forall [s_m, m] ∈ from[p_i]: do
        trigger ⟨ bebBroadcast, [DATA, s_m, m] ⟩;
```

System model
- Synchronous system
- crash failures

Performance:
Best case: 1 beb broadcast message per one rb broadcast message
Worst case: n-1 beb broadcast messages per one rb broadcast (this is the case with n-1 failures)

The algorithm is lazy.
It retransmits a message only if the sender has been detected as crashed

Algorithm 3.3 Eager reliable broadcast.

Implements:
ReliableBroadcast (rb).

Uses:
BestEffortBroadcast (beb).

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

upon event ⟨ *rbBroadcast*, m ⟩ **do**
delivered := delivered ∪ $\{m\}$
trigger ⟨ *rbDeliver*, self, m ⟩;
trigger ⟨ *bebBroadcast*, [DATA, self, m] ⟩;

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

System model
- Asynchronous system
- crash failures

Does not use any FD

Performance:
N beb broadcast messages per one rb broadcast message

It is eager in the sense that it forwards any delivered message.

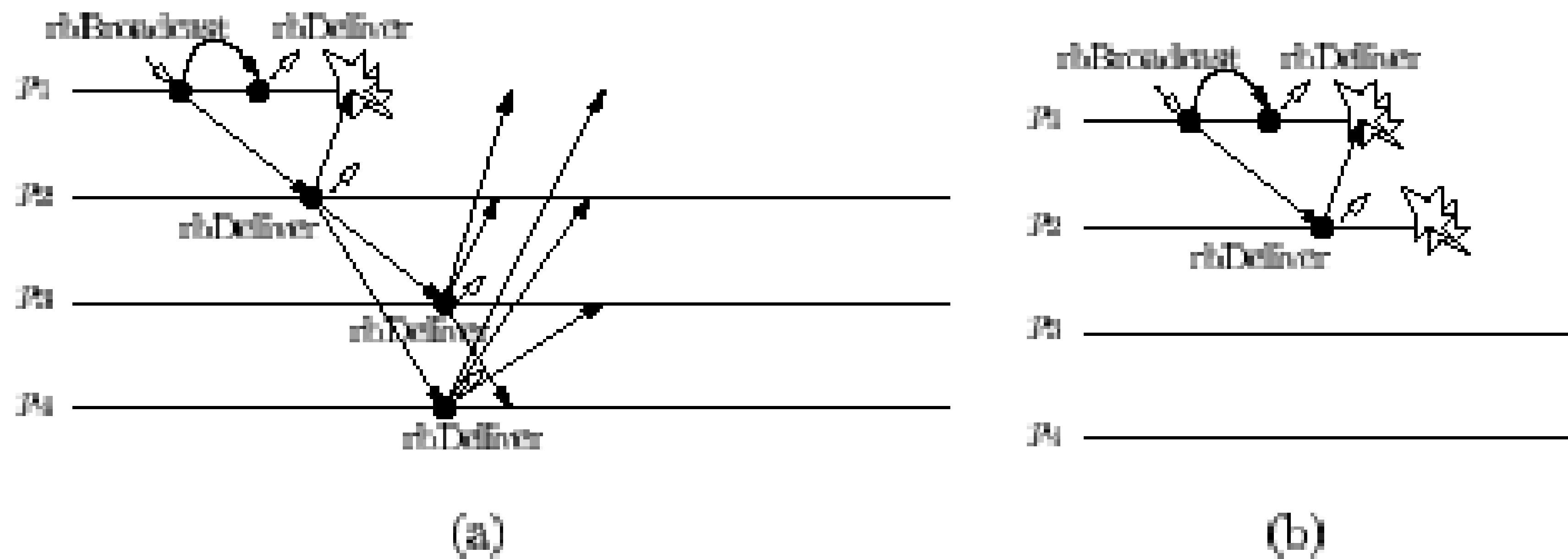


Figure 3.2. Sample executions of eager reliable broadcast.

Uniform Reliable Broadcast

- Stronger form of reliability than RRB
- Agreement on a message delivered by any process (crashed or not)!

Module:

Name: UniformReliableBroadcast (urb).

Events:

⟨ *urbBroadcast*, *m* ⟩, ⟨ *urbDeliver*, *src*, *m* ⟩, 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.

Uniformity states that the set of messages delivered by a correct process is a superset of the ones delivered by a faulty one

---

Module:

Name: UniformReliableBroadcast (urb).

Events:

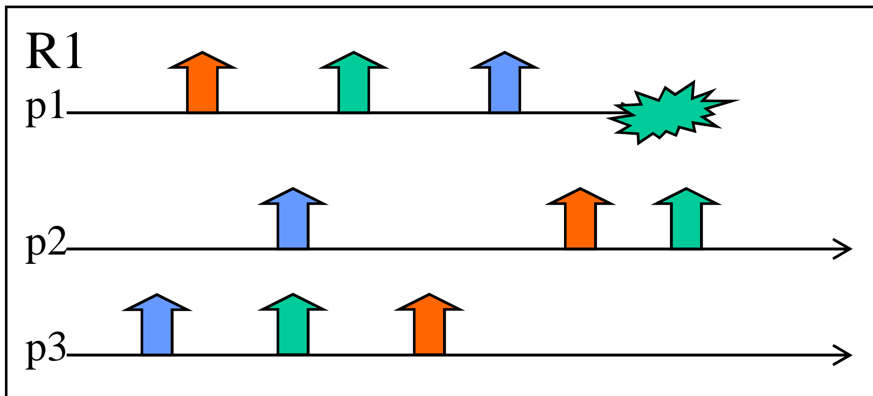
⟨ *urbBroadcast*, m ⟩, ⟨ *urbDeliver*, src, m ⟩, with the same meaning and interface as in regular reliable broadcast.

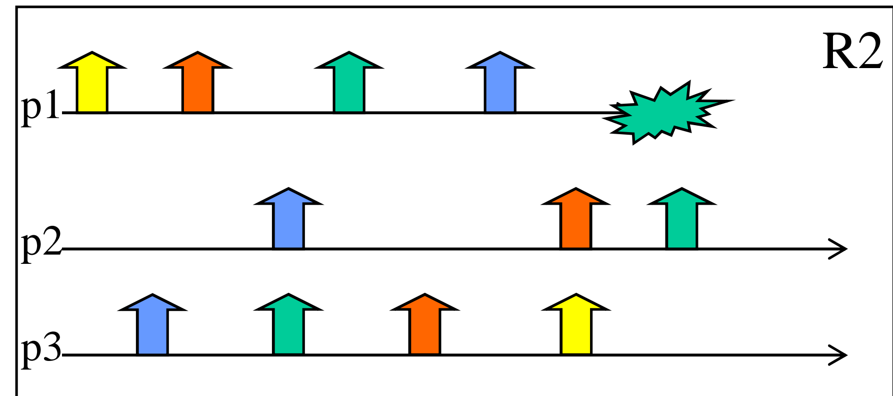
Properties:

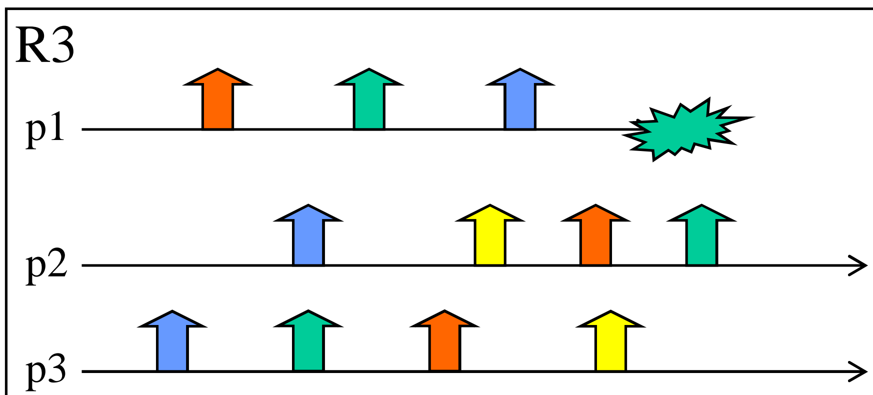
RB1-RB3: Same as in regular reliable broadcast.

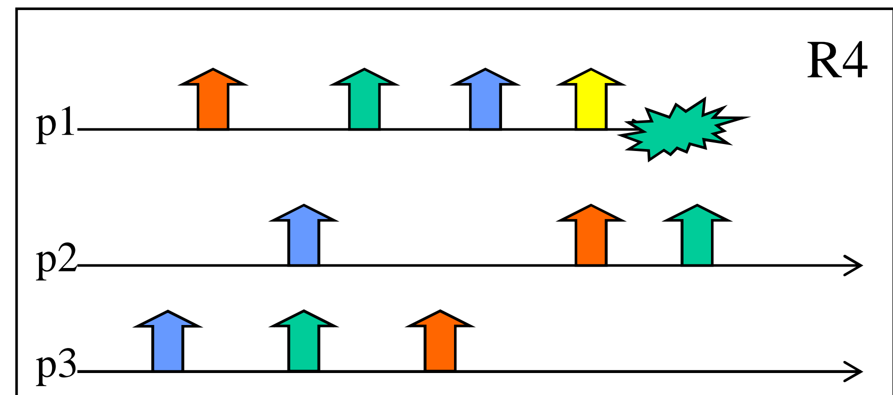
URB4: *Uniform Agreement:* If a message m is delivered by some process p_i (whether correct or faulty), then m is also eventually delivered by every other correct process p_j.

Module 3.3 Interface and properties of uniform reliable broadcast.









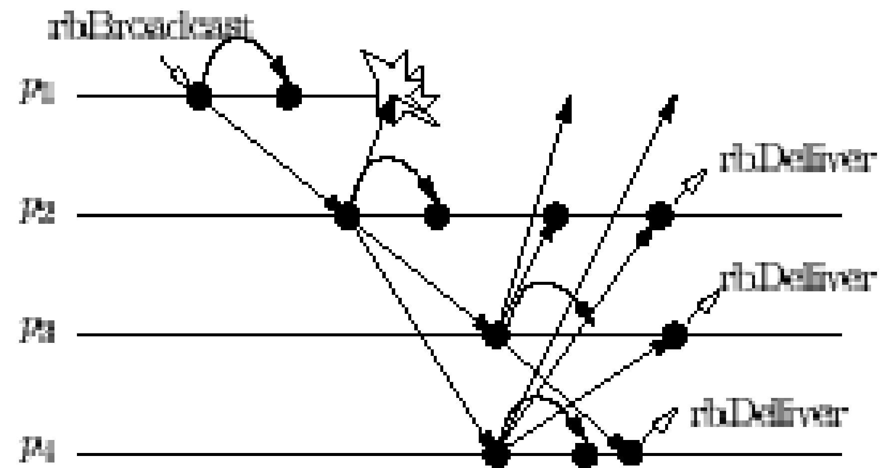


Figure 3.3. Sample execution of uniform reliable broadcast.

Algorithm 3.4 All-Ack Uniform Reliable Broadcast

```
Implements:
    UniformReliableBroadcast (urb).

Uses:
    BestEffortBroadcast (beb).
    PerfectFailureDetector (P).

function canDeliver(m) returns boolean is
    return (correct ⊆ ack_m);

upon event ⟨ Init ⟩ do
    delivered := pending := ∅;
    correct := Π;
    forall m do ack_m := ∅;

upon event ⟨ urbBroadcast | m ⟩ do
    pending := pending ∪ {(self, m)};
    trigger ⟨ bebBroadcast | [DATA, self, m] ⟩;

upon event ⟨ bebDeliver | p_i, [DATA, s_m, m] ⟩ do
    ack_m := ack_m ∪ {p_i};
    if ((s_m, m) ∉ pending) then
        pending := pending ∪ {(s_m, m)};
        trigger ⟨ bebBroadcast | [DATA, s_m, m] ⟩;

upon event ⟨ crash | p_i ⟩ do
    correct := correct \{p_i};

upon exists (s_m, m) ∈ pending such that canDeliver(m) ∧ m ∉ delivered do
    delivered := delivered ∪ {m};
    trigger ⟨ urbDeliver | s_m, m ⟩;
```

System model
- Synchronous system
- crash failure

Algorithm 3.5 Majority-Ack Uniform Reliable Broadcast

Implements:

UniformReliableBroadcast (urb).

Extends:

All-Ack Uniform Reliable Broadcast (Algorithm 3.4).

Uses:

BestEffortBroadcast (beb).

```
function canDeliver(m) returns boolean is
    return (|ack_m| > N/2);

// Except for the function above, and the non-use of the
// perfect failure detector, same as Algorithm 3.4.
```

System model

- Asynchronous system
- crash failure
- majority of correct processes

Uniform Reliable Broadcast

- Exists an algorithm for synchronous system using Perfect failure detector
- Exists an algorithm for asynchronous system when assuming a "majority of correct processes"

- Can we devise a uniform reliable broadcast algorithm for a partially synchronous system (using an eventually perfect failure detector) but without the assumption of a majority of correct processes?

Probabilistic broadcast

- Message delivered 99% of the times
- Not fully reliable
- Large & dynamic groups
- Acks make reliable broadcast not scalable

Ack Implosion and ack tree

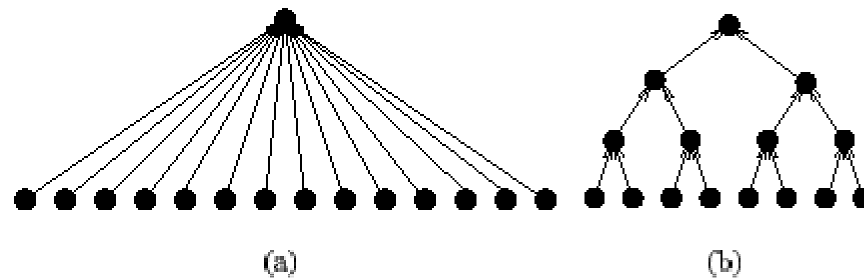


Figure 3.4. Ack implosion and ack tree.

Problems:

Process spends all its time by doing the ack task

Maintaining the tree structure

Probabilistic Broadcast

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:

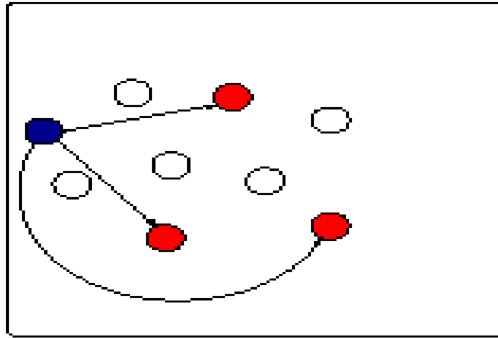
PB1: *Probabilistic validity:* There is a given probability such that for any p_i and p_j that are correct, every message broadcast by p_i is eventually delivered by p_j with this probability.

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

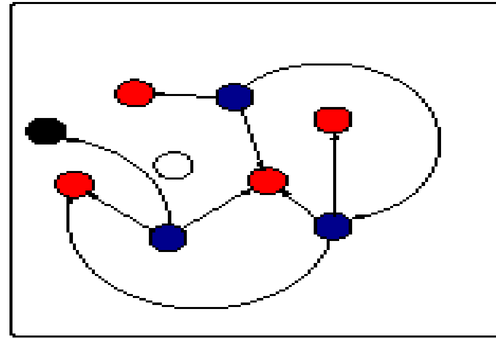
PB3: *No creation:* If a message m is delivered by some process p_j, then m was previously broadcast by some process p_i.

Module 3.7 Interface and properties of probabilistic broadcast.

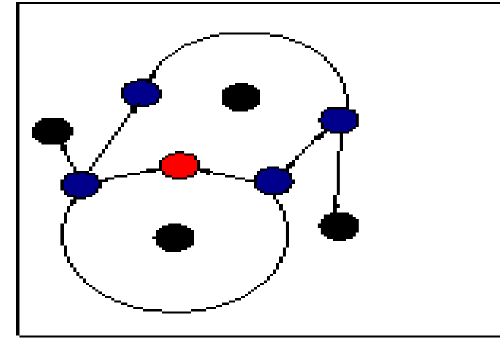
Gossip Dissemination



(a)



(b)



(c)

- A process sends a message to a set of randomly choosen k processes
- A process receiving a message for the first time forwards it to a set of k randomly choosen processes (this operation is also called a round)
- The algorithm performs a maximum number of r rounds

Algorithm 3.9 Eager Probabilistic Broadcast

```
Implements:
    ProbabilisticBroadcast (pb).

Uses:
    FairLossPointToPointLinks (flp2p).

upon event ⟨ Init ⟩ do
    delivered := ∅;

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

procedure gossip (msg) is
    forall t ∈ pick-targets (fanout) do trigger ⟨ flp2pSend | t, msg ⟩;

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

upon event ⟨ flp2pDeliver | p_i, [GOSSIP, s_m, m, r] ⟩ do
    if (m ∉ delivered) then
        delivered := delivered ∪ {m}
        trigger ⟨ pbDeliver | s_m, m ⟩;
    if r > 0 then gossip ([GOSSIP, s_m, m, r − 1]);
```