Abstractions for distributed programming

Rachid Guerraoui

and

Luis Rodrigues

Roadmap

- ***(1) Overview and basic abstractions***
  - Overview
  - Broadcast
  - Consensus
- ***(2) Advanced abstractions***
  - Motivation: database replication
  - Atomic commit
  - Total order broadcast

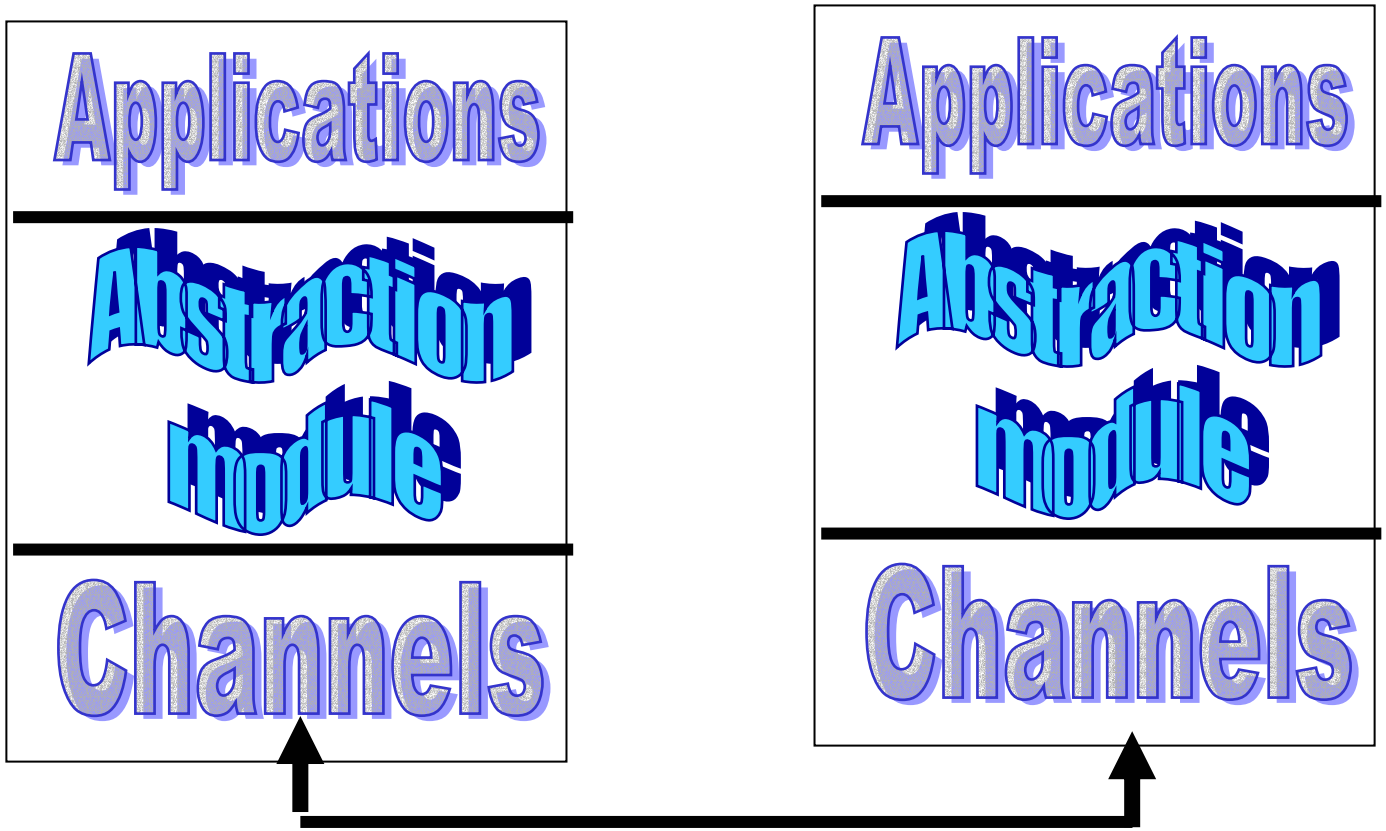
Programming abstractions

- ***Sequential programming***
  - Array, set, record ,..
- ***Concurrent programming***
  - Thread, semaphore, monitor
- ***Distributed programming***
  - ???

Programming abstractions

- ***Sequential programming***
  - Array, set, record ,..
- ***Concurrent programming***
  - Thread, semaphore, monitor
- ***Distributed programming***
  - Reliable broadcast, shared memory, consensus, etc

Distributed programming abstractions



Reliable broadcast
Causal order broadcast
Shared memory
Consensus
Total order broadcast
Atomic commit
Leader election
Terminating reliable broadcast

Underlying abstractions

- (1): processes (abstracting computers)
- (2): channels (abstracting networks)
- (3): failure detectors (abstracting time)

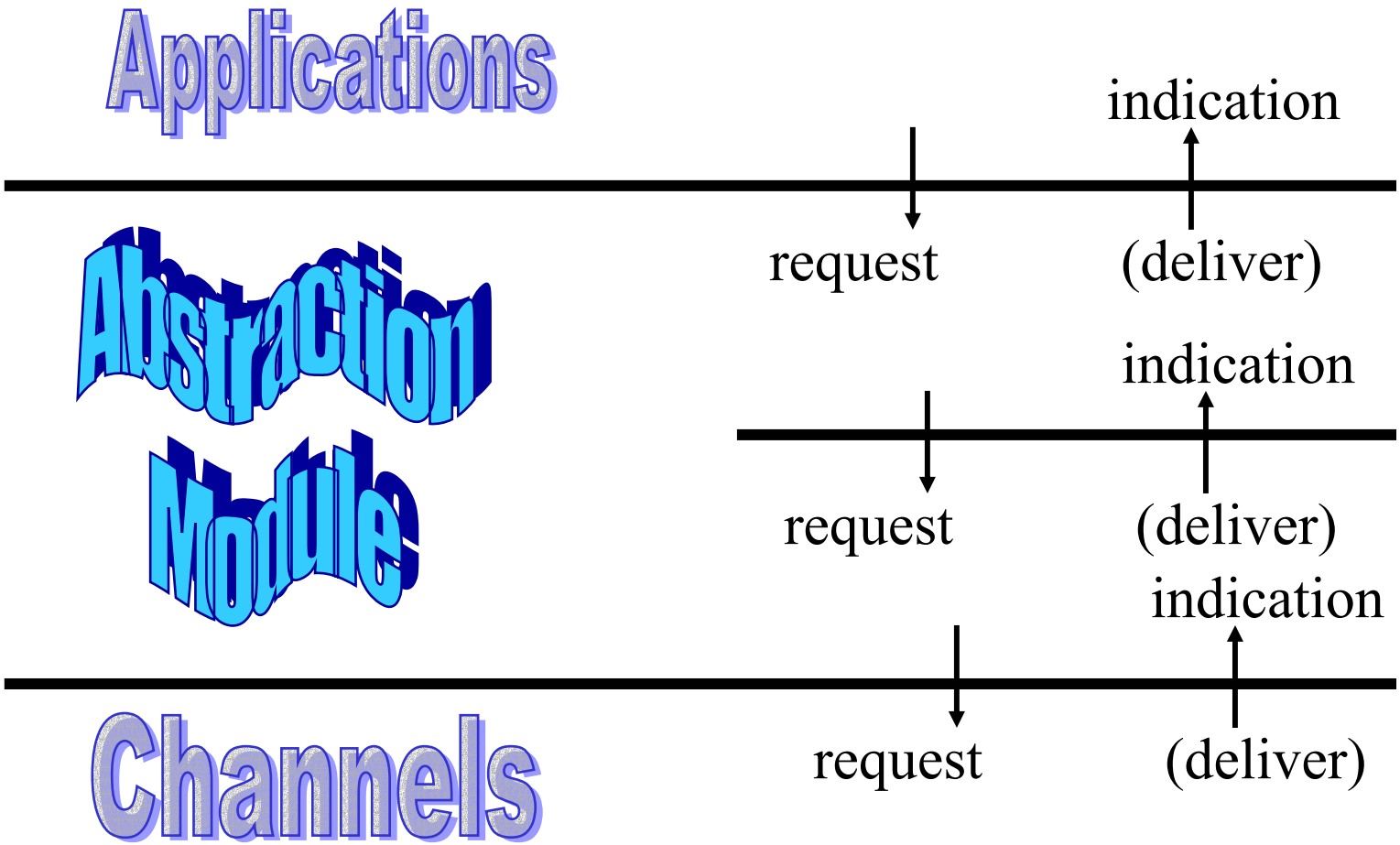
Processes

- The distributed system is made of a finite set S of processes p1,..pN
- Each process models a sequential program
- Processes have unique identities and know each other
- In this tutorial, processes can only fail by crashing and do not recover after a crash: a ***correct*** process does not crash

Processes

- The program of a process is made of a finite set of modules (or components) organized as a software stack
- Modules within the same process interact by exchanging events
- **upon event** < Event1, att1, att2,..> do
  - // something
  - **trigger** < Event2, att1, att2,..>

Modules of a process



Channels

- We assume ***reliable*** links (also called ***perfect***) throughout this tutorial
- Roughly speaking, reliable links ensure that messages exchanged between correct processes are not lost
- Messages are uniquely identified and the message identifier includes the sender's identifier

Channels

- Two primitives: pp2pSend and pp2pdeliver
- Properties:
  - ***PL1. Validity***: If pi and pj are correct, then every message sent by pi to pj is eventually delivered by pj
  - ***PL2. No duplication:*** No message is delivered (to a process) more than once
  - ***PL3. No creation:*** No message is delivered unless it was sent

Failure Detection

- A ***failure detector*** is a distributed oracle that provides processes with suspicions about crashed processes
- It is implemented using (i.e., it encapsulates) timing assumptions
- According to the timing assumptions, the suspicions can be accurate or not

Failure Detection

- We assume in this tutorial ***perfect*** failure detection:
  - *Strong Completeness:* Eventually, every process that crashes is permanently suspected by every correct process
  - *Strong Accuracy:* No process is suspected before it crashes

Roadmap

- ***(1) Overview and basic abstractions***
  - Overview
  - Broadcast
  - Consensus
- ***(2) Advanced abstractions***
  - Motivation: database replication
  - Atomic commit
  - Total order broadcast

Broadcast

- Broadcast is useful for instance in applications where some processes subscribe to events published by other processes (e.g., stocks), and require some **reliability** *guarantees* from the broadcast service (we say sometimes *quality of service* – *QoS*) that the underlying network does not provide
- Broadcast is also useful for (database) replication (as we will see later)

Broadcast

- We shall consider three forms of reliability for a broadcast primitive
  - **(a) *Best-effort broadcast***
  - **(b) *(Regular) reliable broadcast***
  - **(c) *Uniform (reliable) broadcast***
- We shall give first ***specifications*** and then *algorithms*

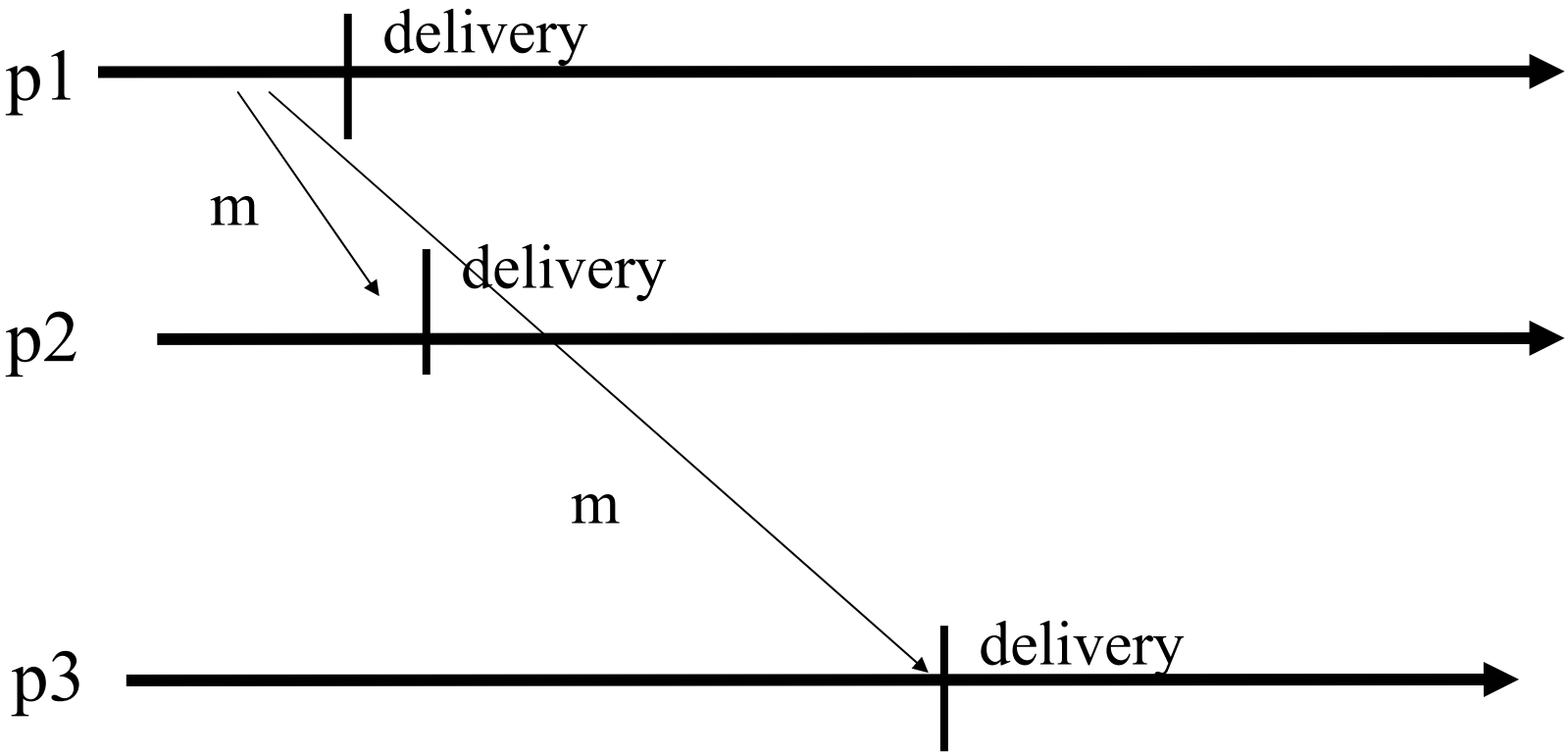
Best-effort broadcast (beb)

- ***Events***
  - Request: <bebBroadcast, m>
  - Indication: <bebDeliver, src, m>
- ***Properties: BEB1, BEB2, BEB3***

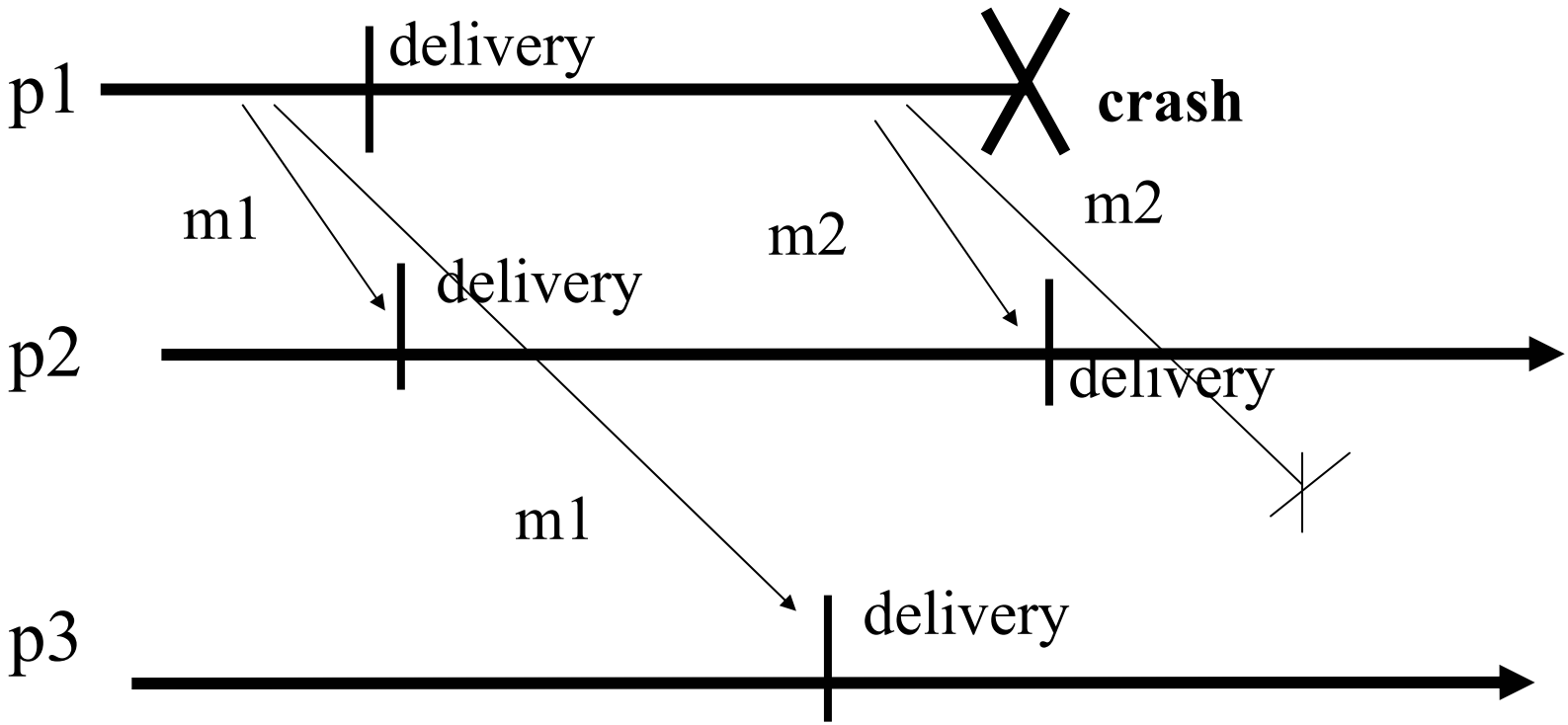
Best-effort broadcast (beb)

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

Best-effort broadcast



Best-effort broadcast



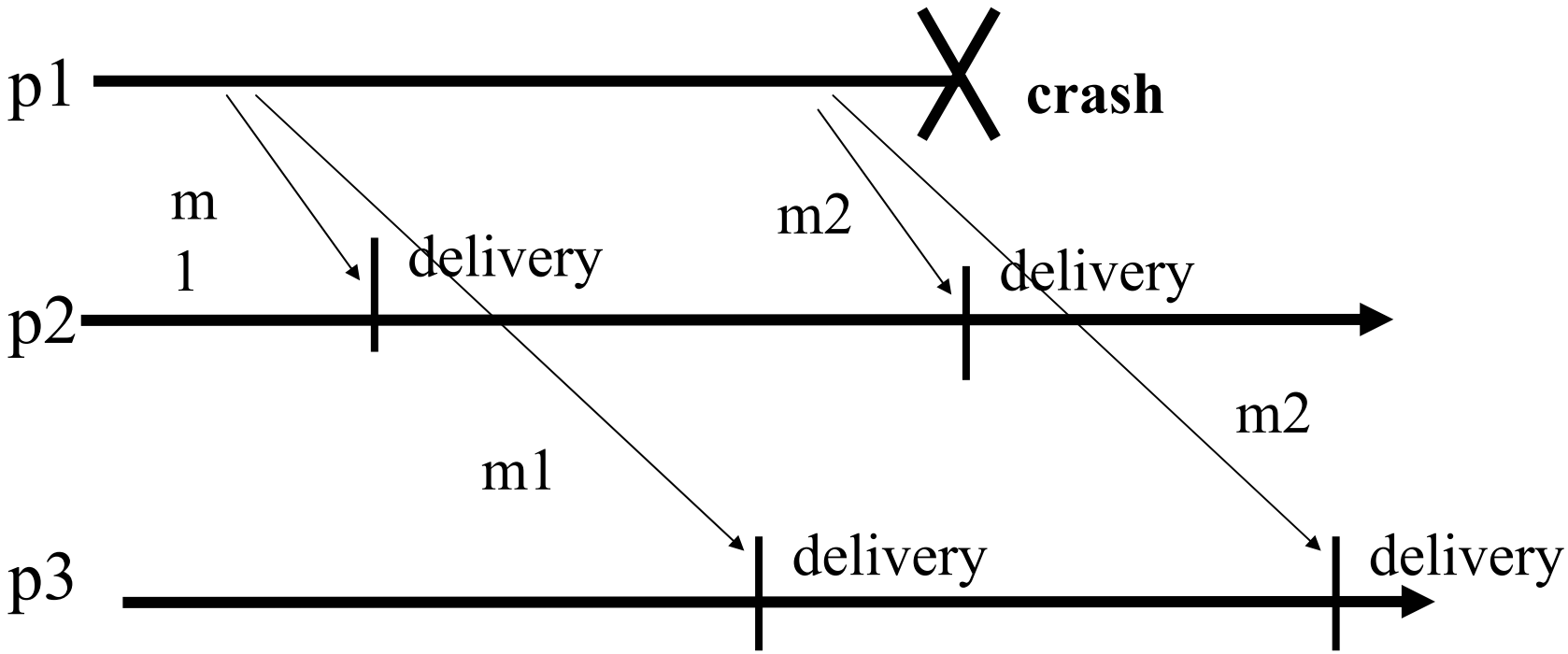
Reliable broadcast (rb)

- ***Events***
  - Request: <rbBroadcast, m>
  - Indication: <rbDeliver, src, m>
- ***Properties: RB1, RB2, RB3, RB4***

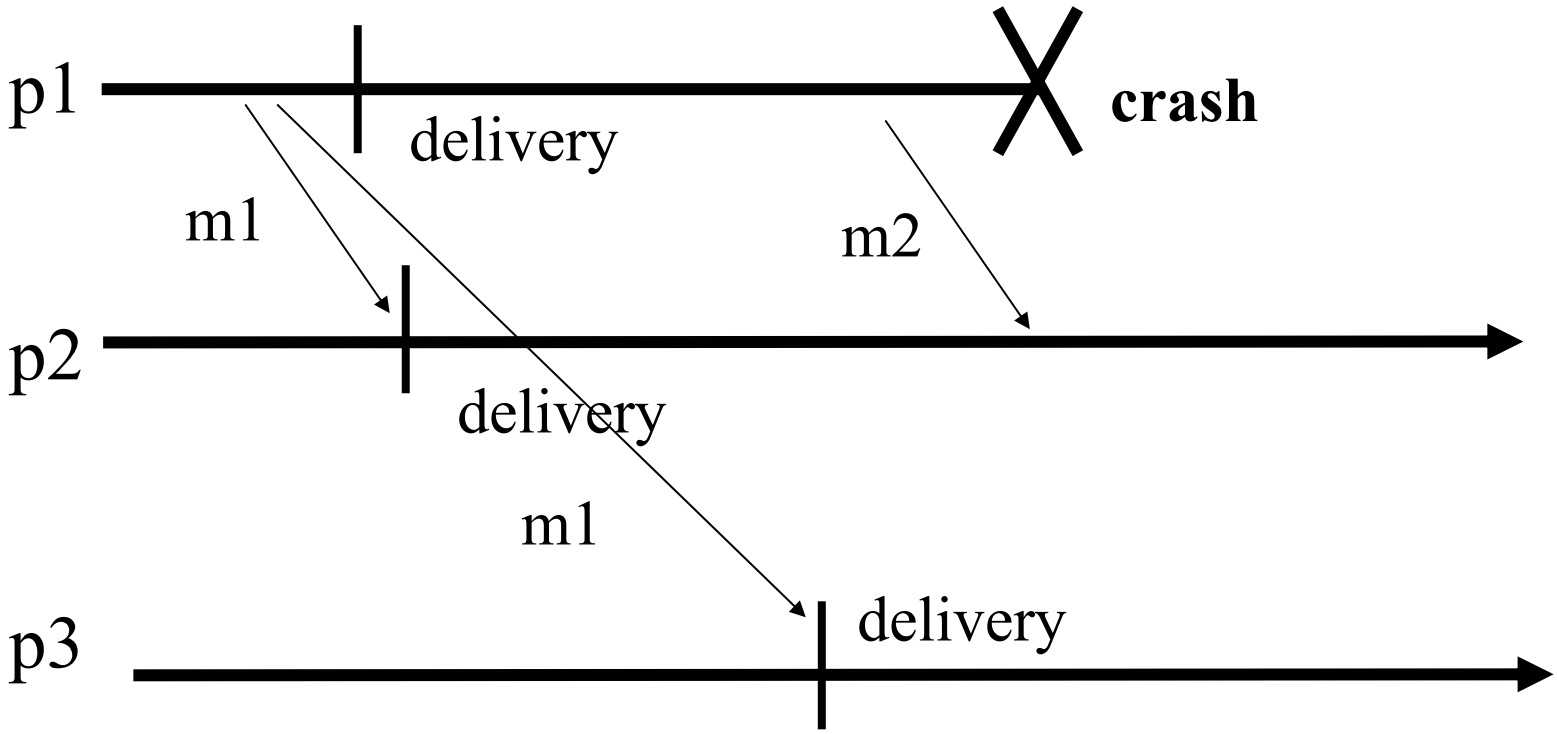
Reliable broadcast (rb)

- ***Properties***
  - ***RB1 = BEB1.***
  - ***RB2 = BEB2.***
  - ***RB3 = BEB3.***
  - ***RB4. Agreement:*** For any message m, if a correct process delivers m, then every correct process delivers m

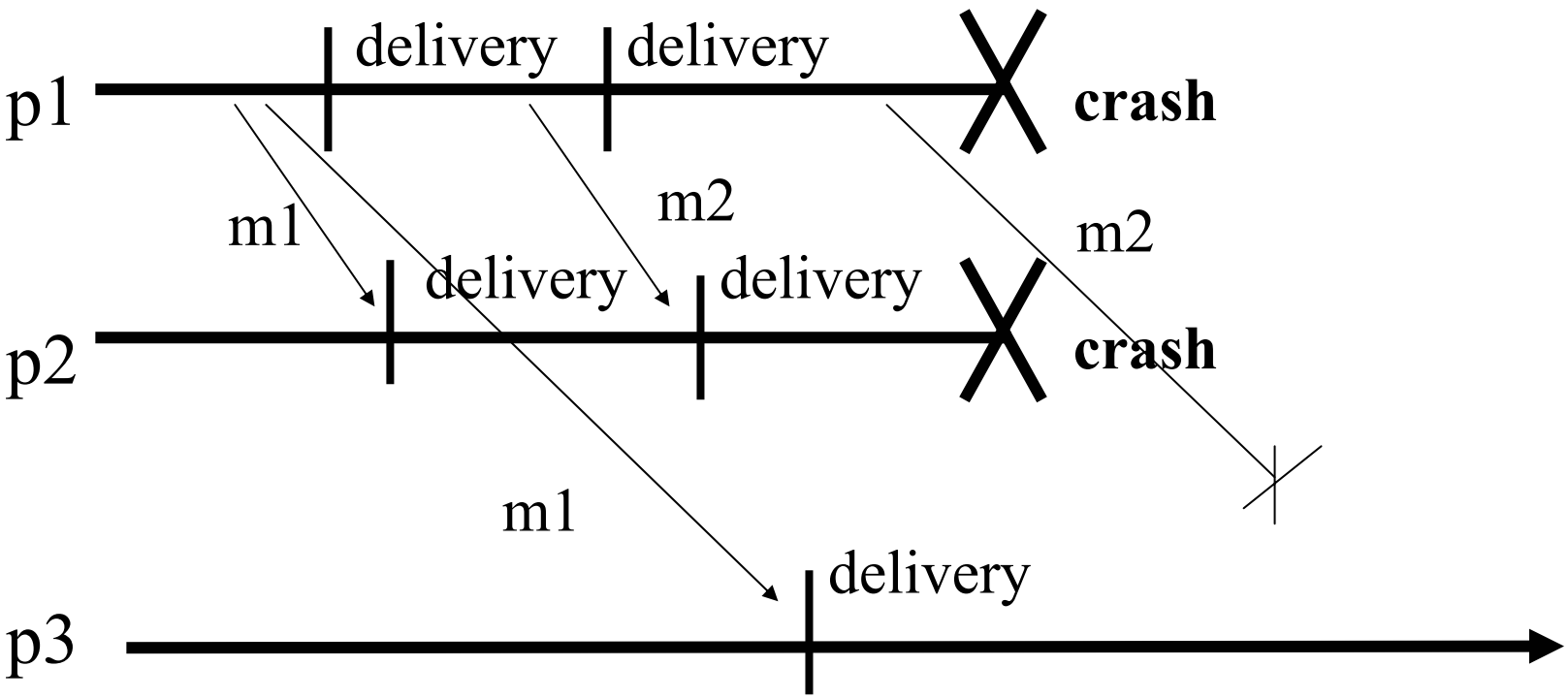
Reliable broadcast



Reliable broadcast



Reliable broadcast



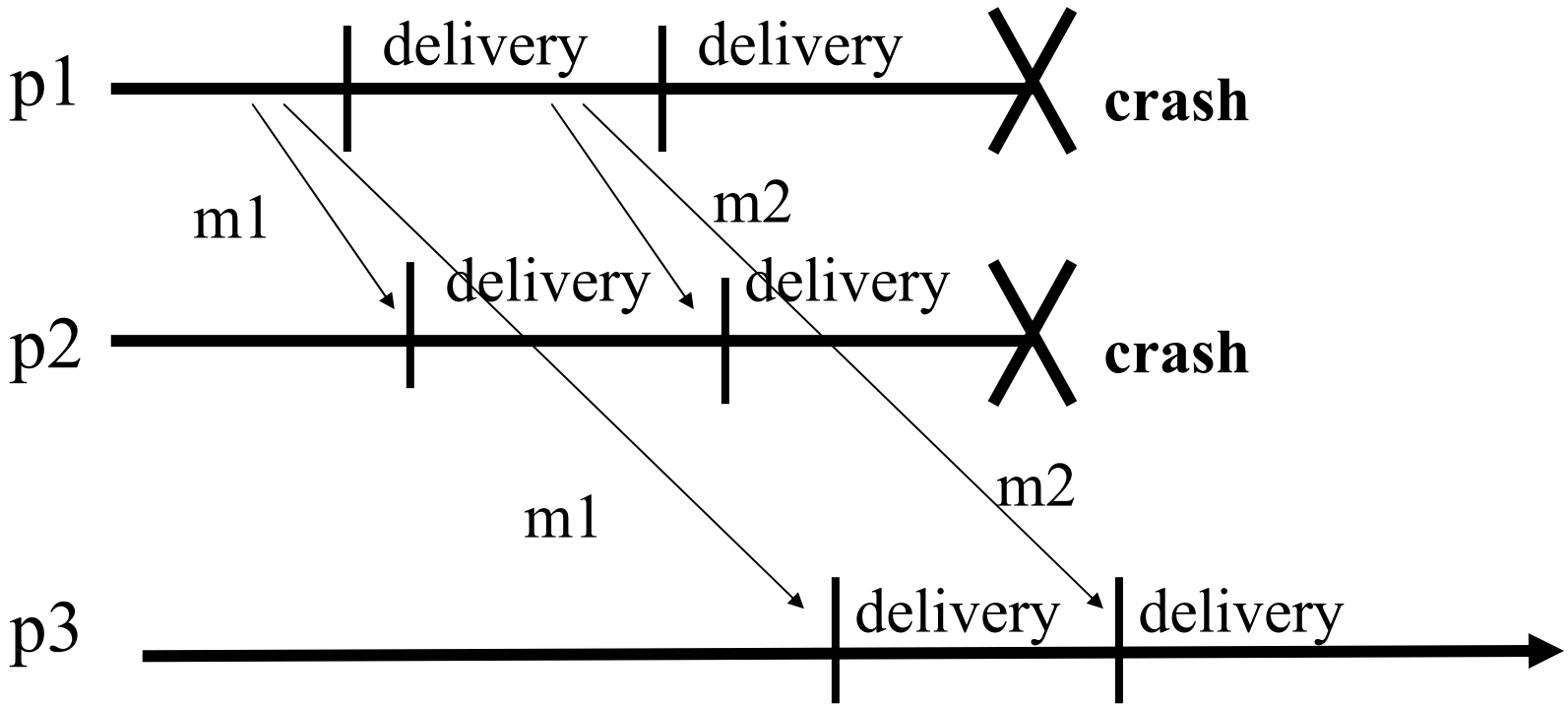
Uniform broadcast (urb)

- ***Events***
  - Request: <urbBroadcast, m>
  - Indication: <urbDeliver, src, m>
- ***Properties: URB1, URB2, URB3, URB4***

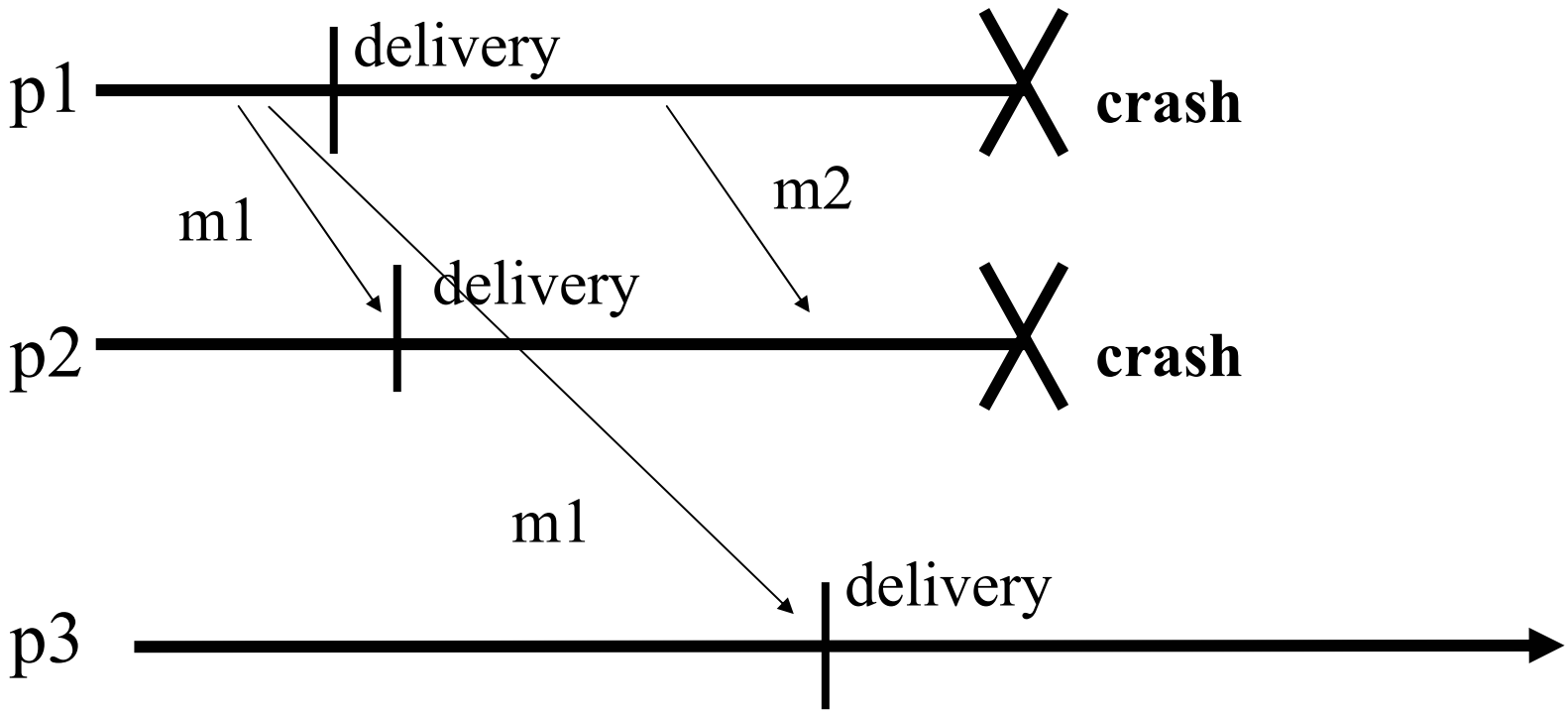
Uniform broadcast (urb)

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

Uniform reliable broadcast



Uniform reliable broadcast



Algorithm (beb)

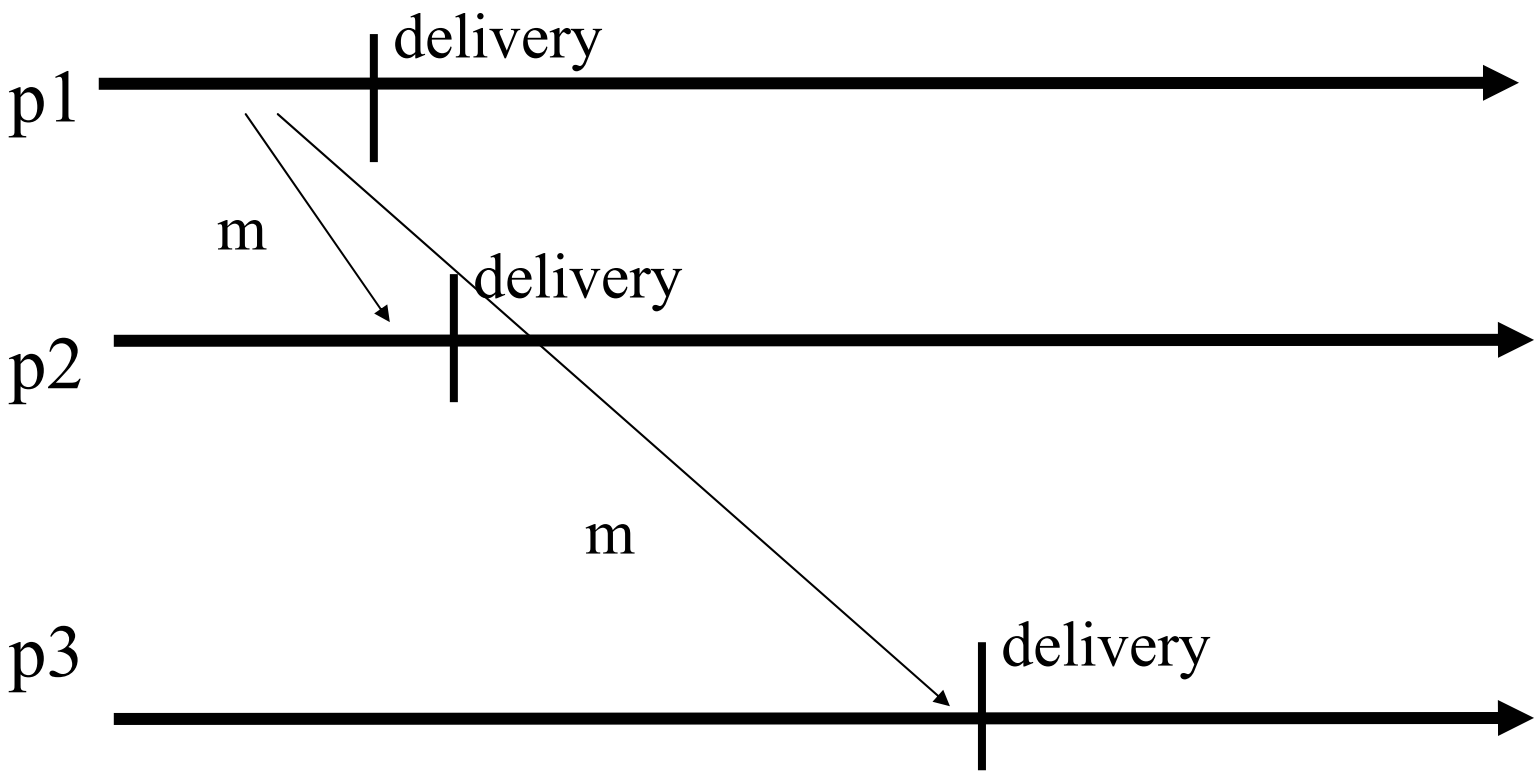
Implements: BestEffortBroadcast (beb).

Uses: PerfectLinks (pp2p).

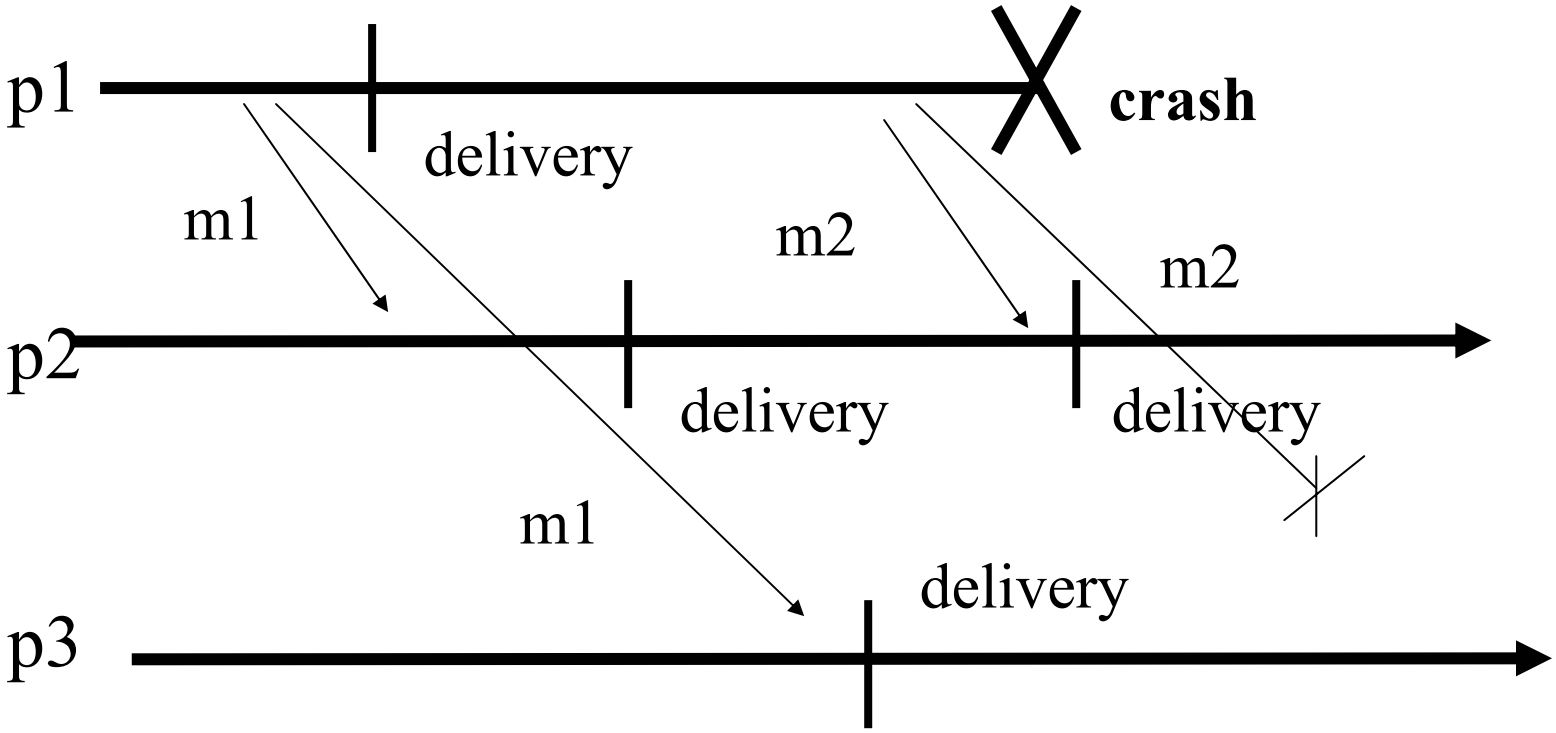
```
upon event < bebBroadcast, m> do
    forall pi in S do
        trigger < pp2pSend, pi, m>;

upon event < pp2pDeliver, pi, m> do
    trigger < bebDeliver, pi, m>;
```

Algorithm (beb)



Algorithm (beb)



Algorithm (rb)

Implements: ReliableBroadcast (rb).

Uses:

BestEffortBroadcast (beb).
PerfectFailureDetector (P).

upon event < Init > **do**

delivered := ∅;
correct := S;
forall pi ∈ S **do** from[pi] := empty;

Algorithm (rb – cont'd)

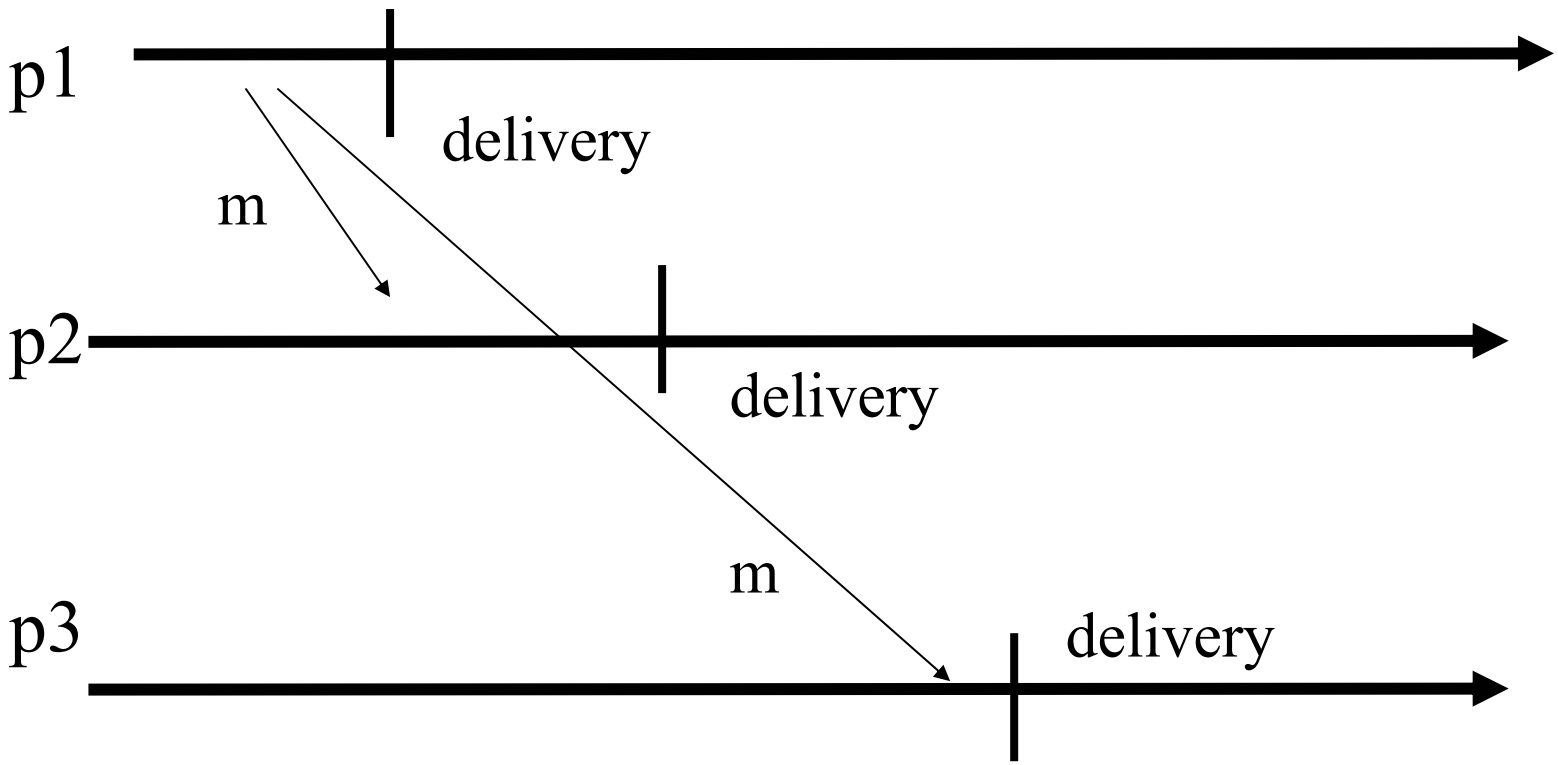
```
upon event < rbBroadcast, m> do
    delivered := delivered U {m};
    trigger < rbDeliver, self, m>;
    trigger < bebBroadcast, [Data,self,m]>;

upon event < crash, pi > do
    correct := correct \ {pi};
    forall [pj,m] ∈ from[pi] do
        trigger < bebBroadcast,[Data,pj,m]>;
```

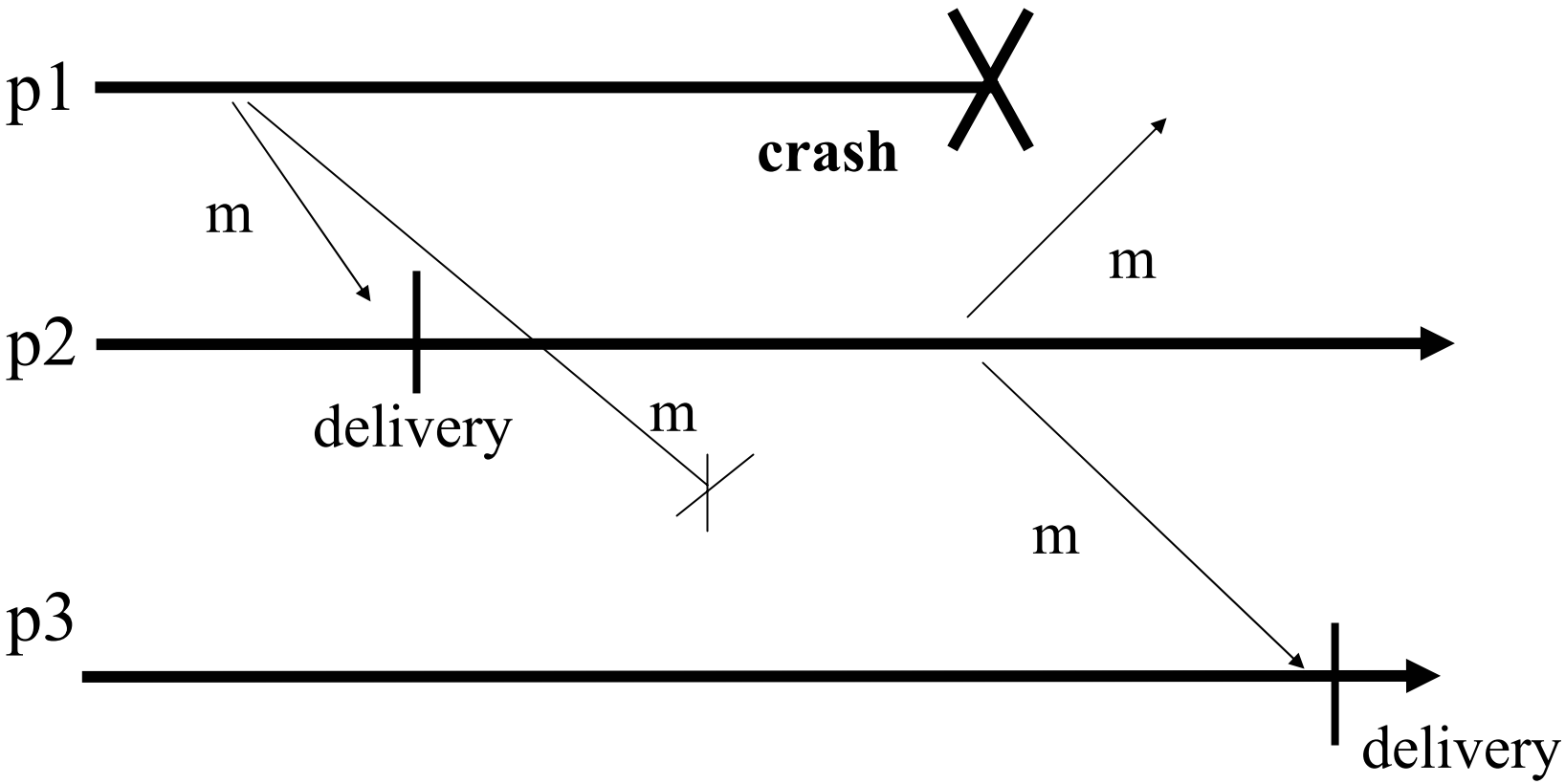
Algorithm (rb – cont'd)

```
upon event < bebDeliver, pi, [Data,pj,m]> do
   if m ∉ delivered then
      delivered := delivered U {m};
      trigger < rbDeliver, pj, m>;
     if pi ∉ correct then
        trigger < bebBroadcast,[Data,pj,m]>;
     else
          from[pi] := from[pi] U {[pj,m]};
```

Algorithm (rb)



Algorithm (rb)



Algorithm (urb)

Implements: uniformBroadcast (urb).

Uses:

BestEffortBroadcast (beb).

PerfectFailureDetector (P).

Algorithm (urb)

upon event < Init > **do**

correct := S;

delivered := forward := empty;

ack[Message] := ∅;

upon event < crash, pi > **do**

correct := correct \ {pi};

Algorithm (urb – cont'd)

upon event < urbBroadcast, m> **do**

forward := forward U {[self,m]};

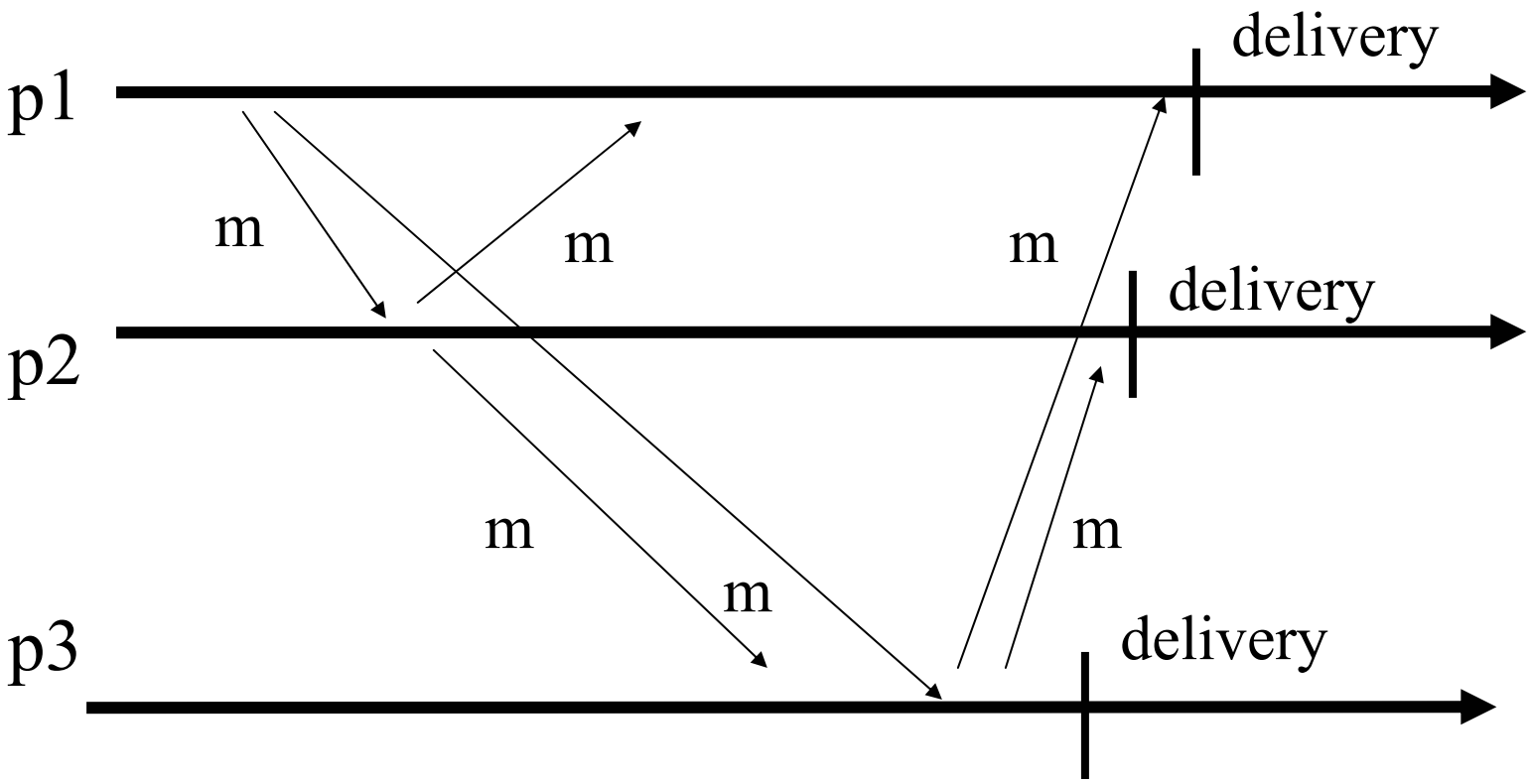
trigger < bebBroadcast, [Data,self,m]>;

upon event (for any [pj,m] in forward) <correct $\subseteq$ ack[m]> **and** <m $\notin$ delivered> **do**

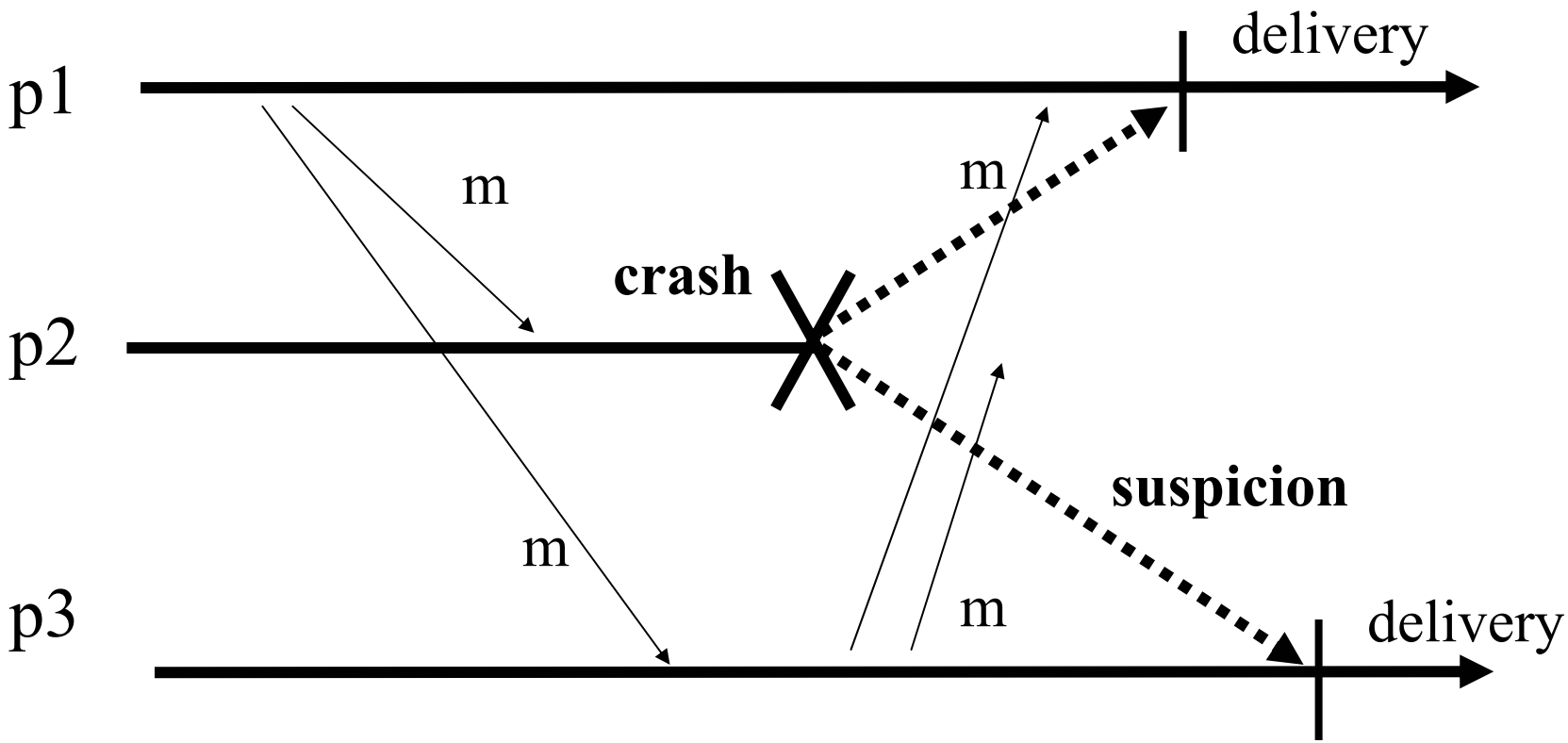
delivered := delivered U {m};

trigger < urbDeliver, pj, m>;

Algorithm (urb)



Algorithm (urb)



Roadmap

- ***(1) Overview and basic abstractions***
  - Overview
  - Broadcast
  - Consensus
- ***(2) Advanced abstractions***
  - Motivation: database replication
  - Atomic commit
  - Total order broadcast

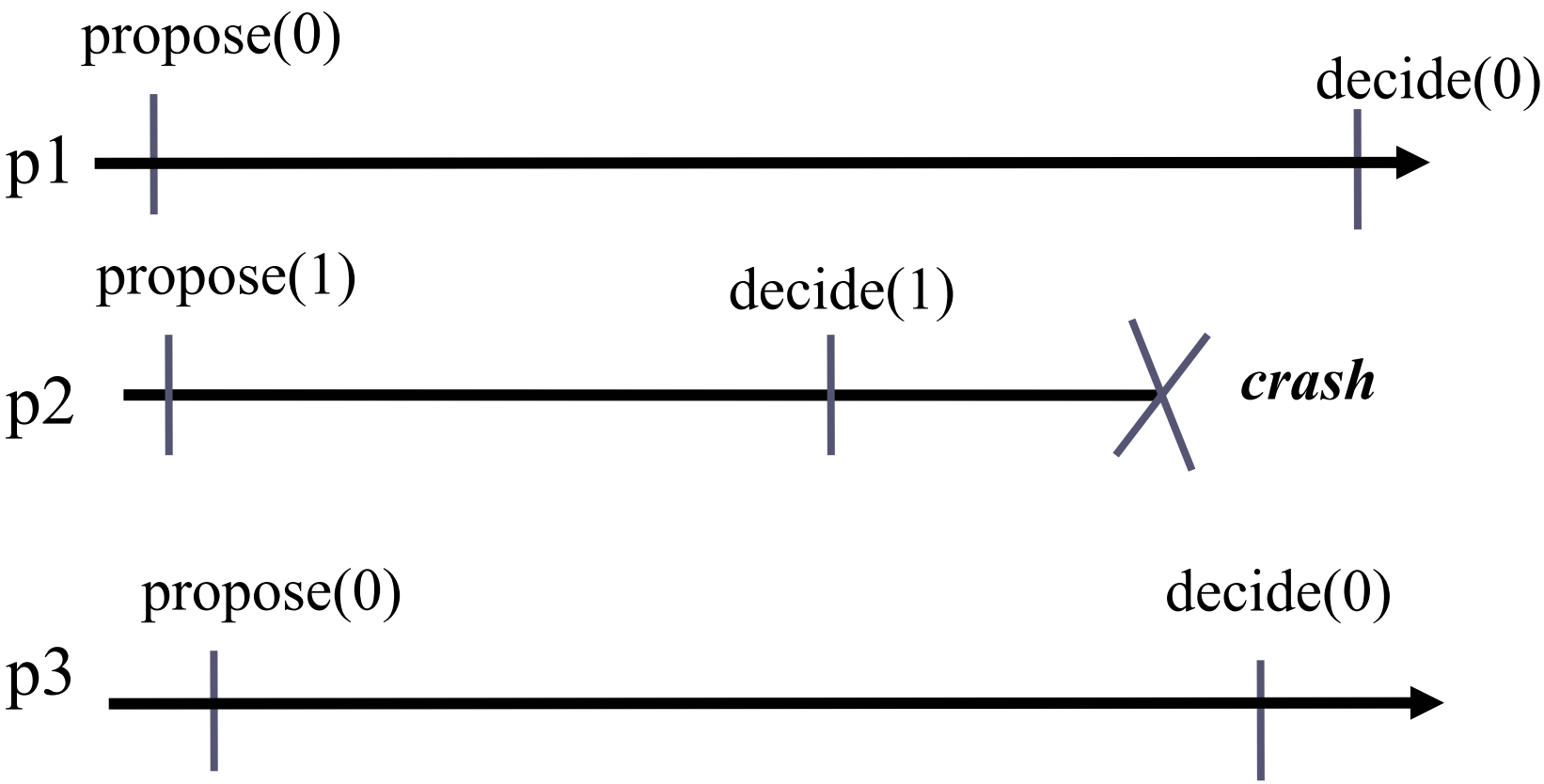
Consensus

- In the consensus problem, the processes propose values and have to agree on one among these values
- Solving consensus is key to solving many problems in distributed computing (e.g., total order broadcast, atomic commit, terminating reliable broadcast)

Consensus

- ***C1. Validity:*** Any value decided is a value proposed
- ***C2. Agreement:*** No two correct processes decide differently
- ***C3. Termination:*** Every correct process eventually decides
- ***C4. Integrity:*** No process decides twice

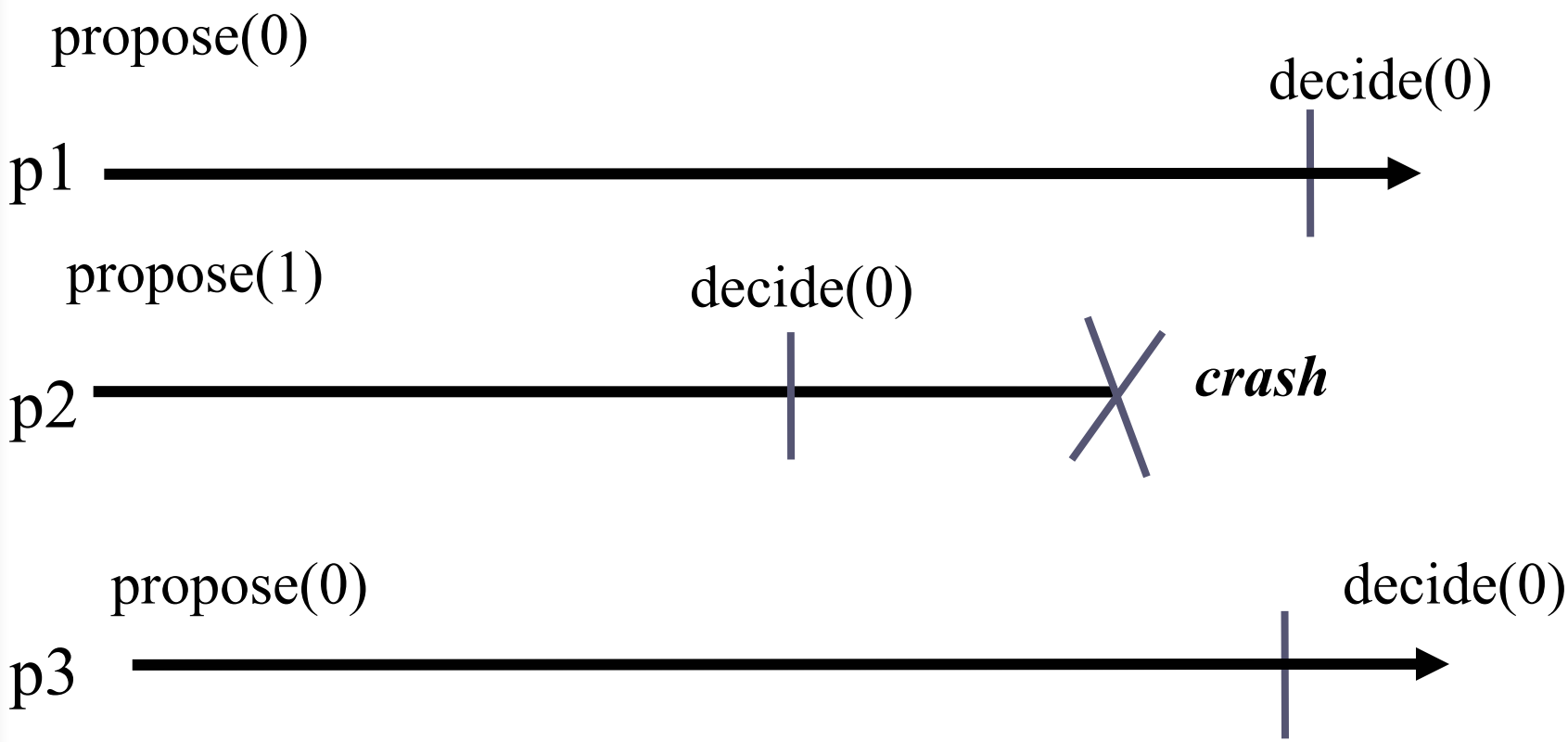
Consensus


propose(0)
decide(0)
p1
propose(1)
decide(1)
p2
crash
propose(0)
decide(0)
p3

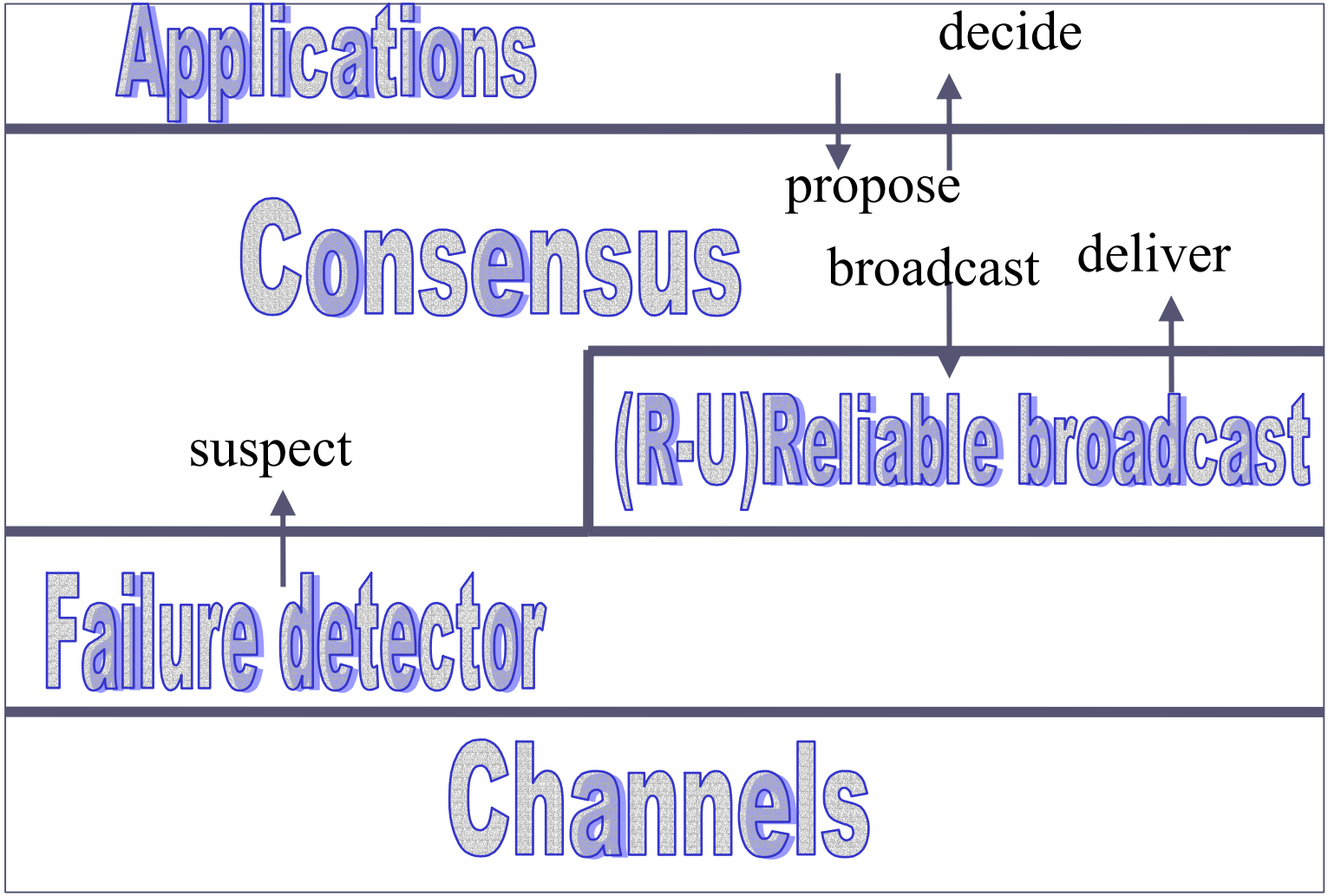
Uniform consensus

- ***C1. Validity:*** Any value decided is a value proposed
- ***C2'. Uniform Agreement:*** No two processes decide differently
- ***C3. Termination:*** Every correct process eventually decides
- ***C4. Integrity:*** No process decides twice

Uniform consensus



Modules of a process



Algorithm (consensus)

- The processes go through rounds incrementally (1 to n): in each round, the process with the id corresponding to that round is the leader of the round
- The leader of a round decides its current proposal and broadcasts it to all
- A process that is not leader in a round waits (a) to deliver the proposal of the leader in that round to adopt it, or (b) to suspect the leader

Algorithm (consensus)

Implements: Consensus (cons).

Uses:

BestEffortBroadcast (beb).
PerfectFailureDetector (P).

upon event < Init > **do**

detected := ∅;
round := 1;
currentProposal := nil;

Algorithm (consensus cont´d)

upon event < crash, pi > **do**

detected := detected U {pi};

upon event < Propose, v> and currentProposal= nil **do**

currentProposal := v;

upon event pround=self **and** currentProposal□nil **do**

trigger <Decide, currentProposal>;

trigger <bebBroadcast, currentProposal>;

Algorithm (consensus cont´d)

upon event < bebDeliver, pround, value > **do**

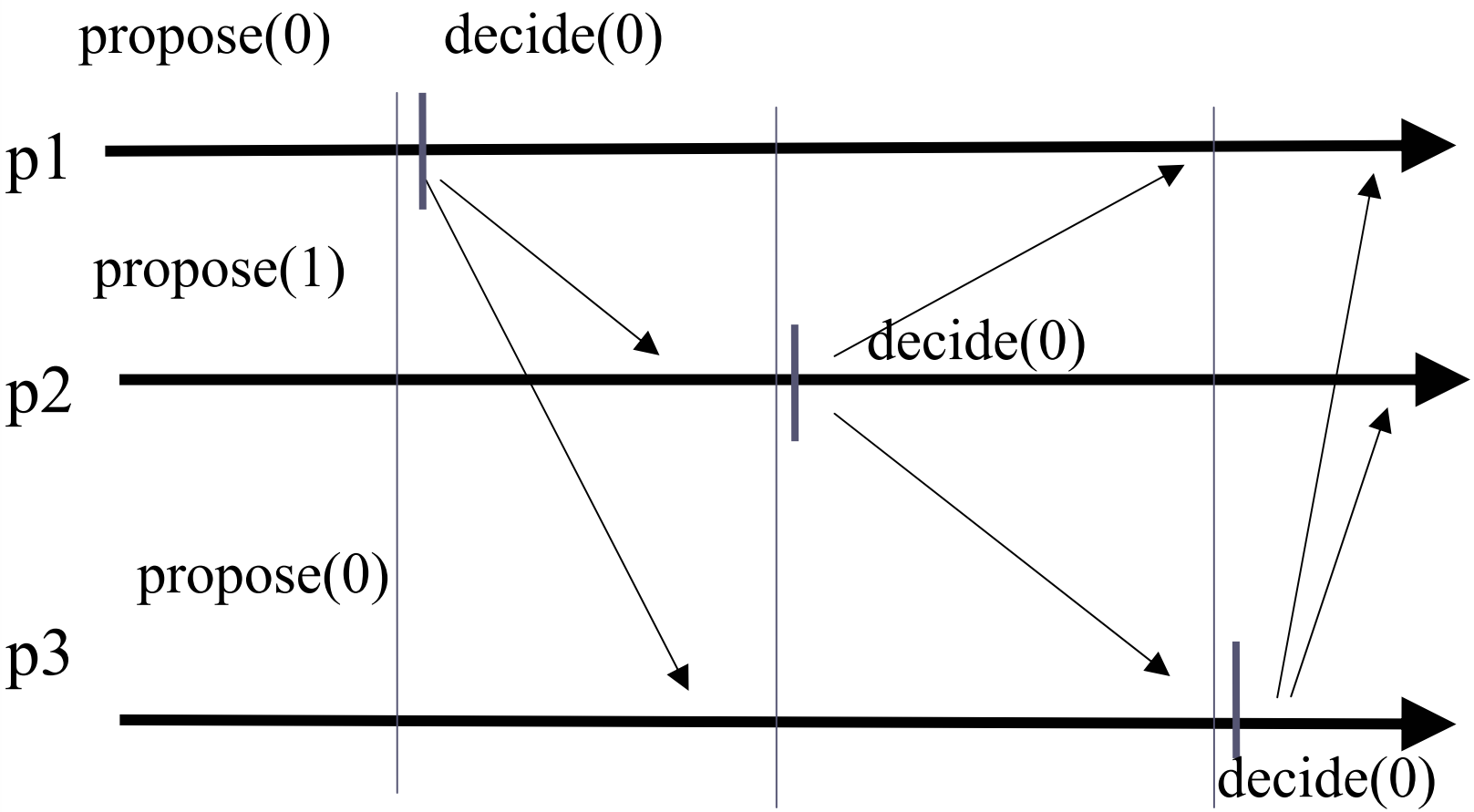
currentProposal := value;

round := round + 1;

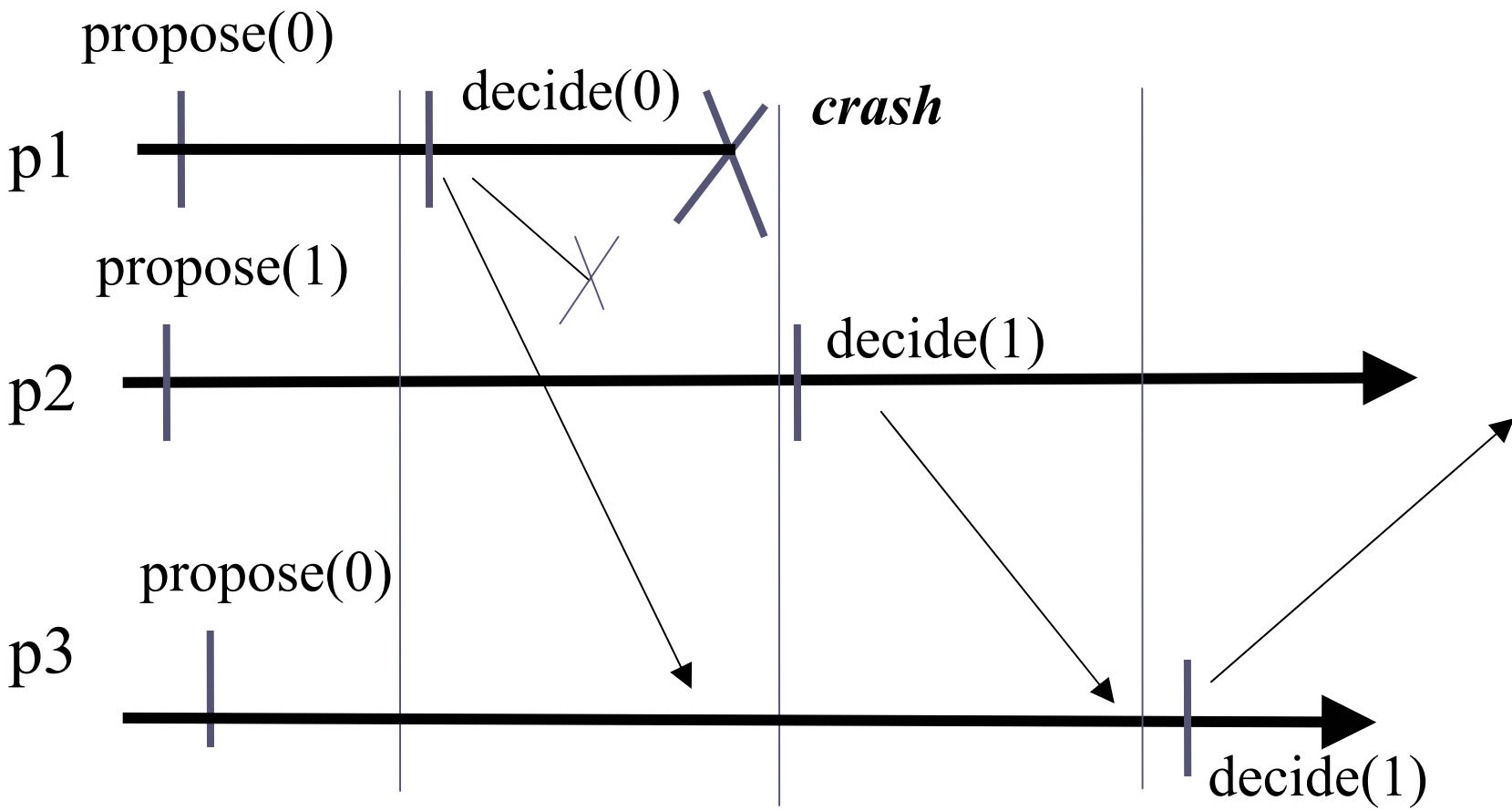
upon event pround in detected **do**

round := round + 1;

Algorithm (consensus)



Algorithm (consensus)



Algorithm (uCons)

Implements: UniformConsensus (uCons).

Uses:

BestEffortBroadcast (beb).

ReliableBroadcast(rb).

PerfectFailureDetector (P).

PerfectPointToPointLinks(pp2p);

Algorithm (uCons)

upon event < Init > **do**

detected := ackSet:= ∅;

round := 1;

currentProposal := nil;

upon event < Propose, v> and currentProposal = nil **do**

currentProposal := v;

upon event < crash, pi > **do**

detected := detected U {pi};

Algorithm (uCons cont´d)

upon event pround=self **and** currentProposal□nil **do**

trigger <bebBroadcast, currentProposal>;

upon event < bebDeliver, pround, value > **do**

currentProposal := value;

trigger <pp2pSend, pround>;

round := round + 1;

upon event pround in detected **do**

round := round + 1;

Algorithm (uCons cont´d)

upon event < pp2pDeliver, pi, round > **do**
    ackSet := ackSet U {pi};

upon event ackSet ∪ detected = S do
    trigger <rbBroadcast, Decided, currentProposal>;

upon event < rbDeliver, pi,Decided,v > **do**
    trigger <Decide, v>;

Roadmap

- ***(1) Overview and basic abstractions***
  - Overview
  - Broadcast
  - Consensus
- ***(2) Advanced abstractions***
  - Motivation: database replication
  - Atomic commit
  - Total order broadcast

Example: database replication

- To show how to apply the abstractions presented before to build concrete applications
- To introduce a number of relevant variants of the consensus abstraction
  - Atomic commit
  - Total order broadcast

Databases

- Fundamental building block of today´s information systems
- Repository of information
- Data accessed in the context of a transaction.

Transactions

- Sequence of operations on data that is executed as an atomic unit.
- Operations:
  - Read item
  - Write item
- Sequenced of operations bounded by begin_transaction and end_transaction operators.

Transactions

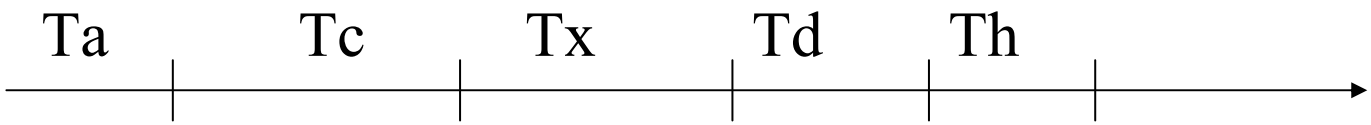
- Example: T1
  - begin_transaction
    - local_variable = Read (A);
    - local_variable = local_variable *2
    - Write (A)
    - Write (B)
  - end_transaction

Transactions: ACID properties

- Atomicity
  - All operations execute or none executes
- Consistency
  - Transactions leave the database in a consistent state
- Isolation
  - Transactions execute as if there is no other transaction accessing the database
- Durability
  - Transaction results persist across failures

Transactions: serializability

- The result of a concurrent execution of transactions must be equivalent to the execution of those transactions in some serial order



Trasactions: serializability

- Transactions that do not conflict may be executed concurrently
- Order must be enforced on transactions that conflict
  - i.e., that access the same data items
- Serializability ensured by concurrency control mechanism
  - Two-phase locking, timestamps, etc

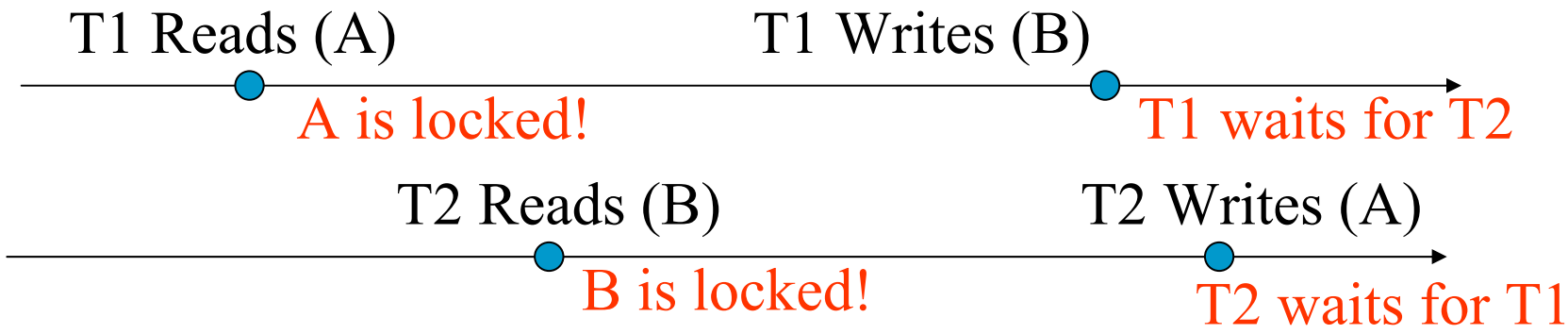
Transactions: two-phase locking

- When a transaction accesses a data item, it acquires a lock
  - Read lock
  - Write lock
- If the item is already locked, the transaction waits
- When the transaction terminates, all locks acquired are released

Transactions: termination

- If the transaction concludes all its operations with success, the transaction commits
- If the transaction cannot conclude, the transaction aborts

Transactions: deadlock



- Both transactions are waiting for each other!
- Usually resolved by timeout and by aborting one or both transactions
  - (it is also possible to execute a deadlock detection algorithm)

Database replication

- If there is a single copy of the database, updates may be lost if there is a failure in that copy:
  - Hardware failure
  - Natural disaster
  - Enemy attack

Database replication

- Solution: database replication!
  - To keep multiple copies of the database
  - Ensure that all updates are performed at every copy
    - Zero loss!

Database replication: pros and cons

- Disadvantage
  - Update operations are more expensive
    - Need to update all copies
- Advantages
  - Read operations are more efficient
    - Data is local
  - Non-conflicting operations may potentially be executed by different copies
- For certain loads, replication can improve the performance of the database
  - In addition to provide fault-tolerance

Database replication: how to?

- A protocol to ensure that all replicas are updated
  - Updates are not instantaneous
- Protocol to ensure that replication is transparent to clients
  - Consistency protocol
  - One-copy equivalence

Naïve approach

- Transform any write operation in a sequence of write operations (one for each copy of the item)
  - For instance:
    - Write (A) -> Write (A1); Write (A2); Write (A3)
- Read any of the copies
- Use local concurrency control at each replica to detect conflicts

Example

```
begin_transaction
local_variable = Read (A);
local_variable = local_variable *2
Write (A)
end_transaction
```

local_variable = 0

Replica 1

A1 = 1 (V=1)

B1 = 2 (V=2)

Replica 2

A2 = 1 (V=1)

B2 = 3 (V=2)

Replica 3

A3 = 1 (V=1)

B3 = 3 (V=2)

Distributed transaction

- Transaction updates items (more precisely replicas of items) at different copies.
- Transaction must be atomic
  - All replicas are updated or no replica is updated
- Distributed atomic commitment!

Roadmap

- ***(1) Overview and basic abstractions***
  - Overview
  - Broadcast
  - Consensus
- ***(2) Advanced abstractions***
  - Motivation: database replication
  - Atomic commit
  - Total order broadcast

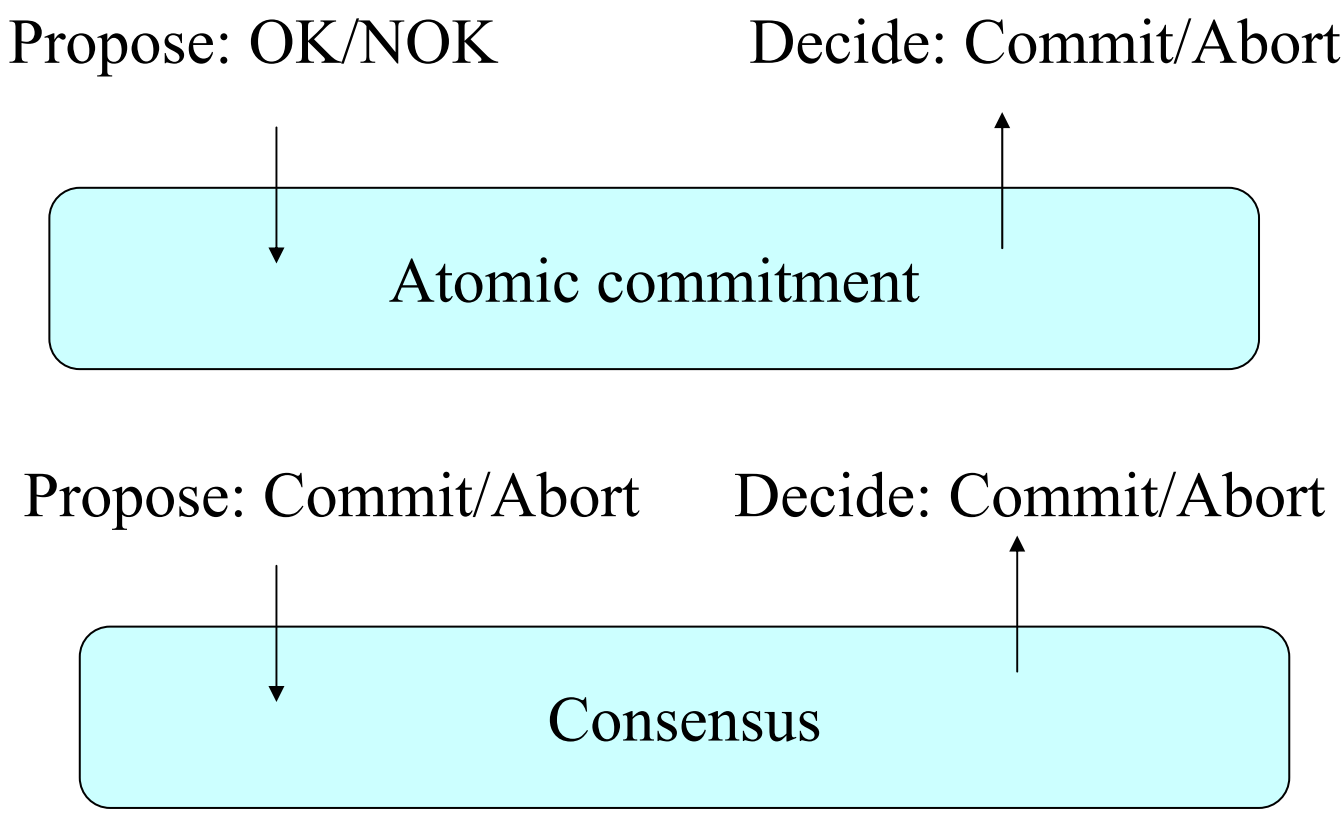
Distributed atomic commitment

- Variant of the consensus abstraction presented before.
- Processes propose one of the following values:
  - OK
  - NOK
- Result:
  - Commit if all nodes propose OK
  - Abort it at least one node proposes NOK or if at least one node fails

Non-Blocking Atomic Commit

- **NBAC1. Agreement:** No two processes decide differently
- ***NBAC2. Termination:*** Every correct process eventually decides
- ***NBAC3. Commit-Validity:*** 1 can only be decided if all processes propose 1
- ***NBAC4. Abort-Validity:*** 0 can only be decided if some process crashes or votes 0

Non-blocking atomic commitment



Algorithm (NBAC)

upon event < nbacPropose | v >
    trigger < bebBroadcast | v >

upon event < bebDeliver | p, NOK >
    trigger < ucPropose | Abort >

upon event < crash | p >
    trigger < ucPropose | Abort>

Algorithm (NBAC cont´d)

upon event < bebDeliver | p, OK >

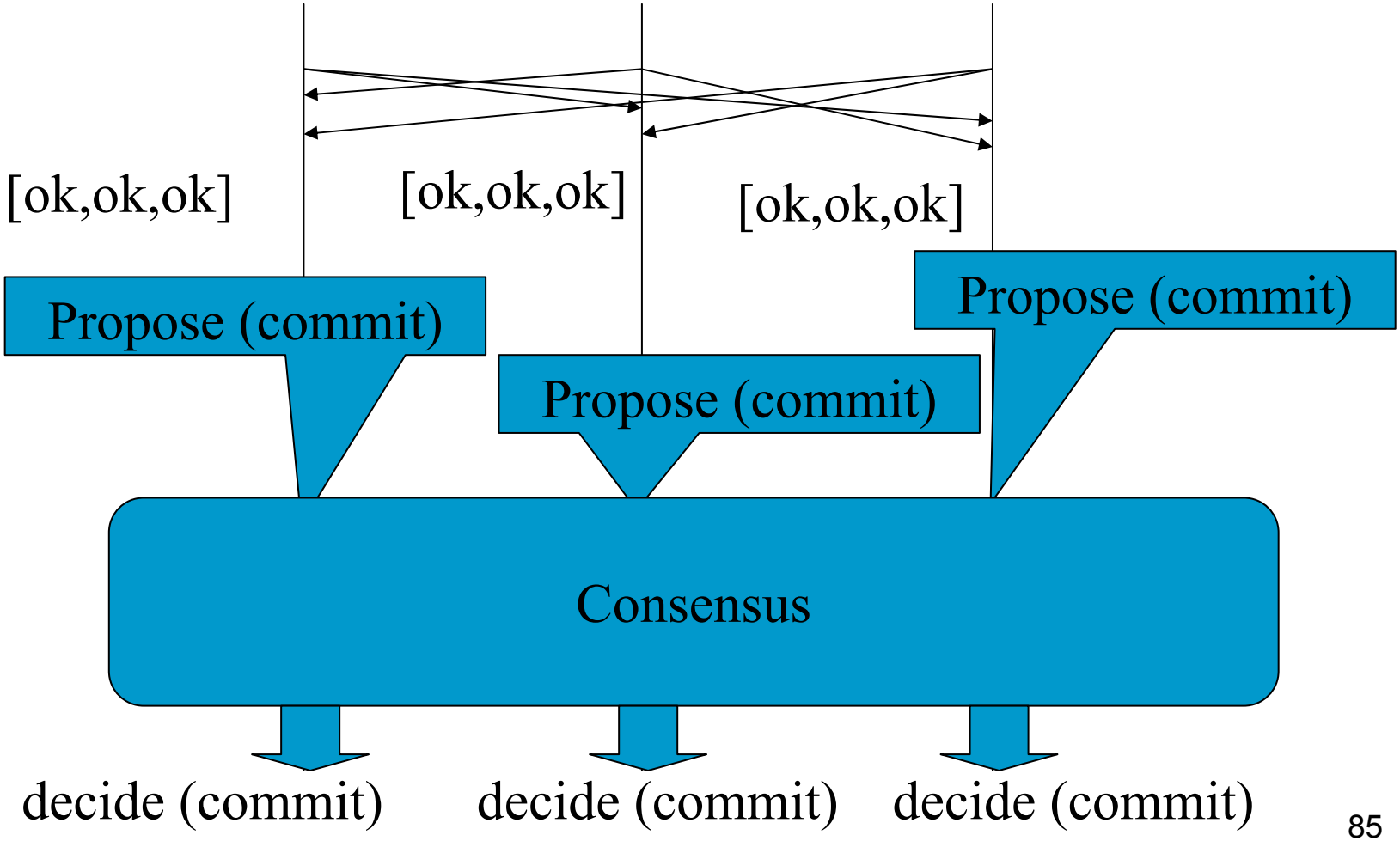
  if all processes have sent OK **then**

    trigger < ucPropose | Commit >

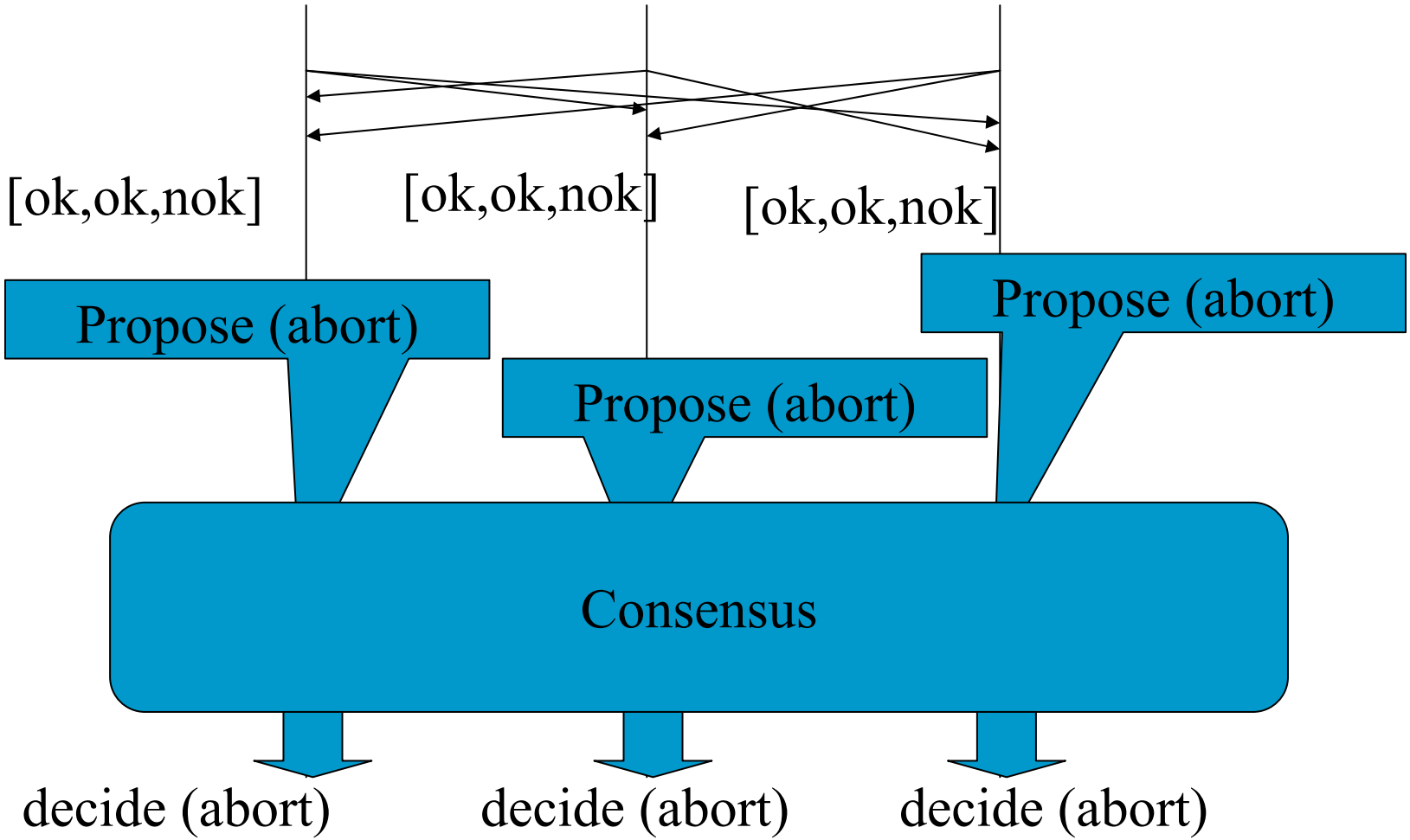
upon event < ucDecide | result >

  trigger < nbacDecide | result >

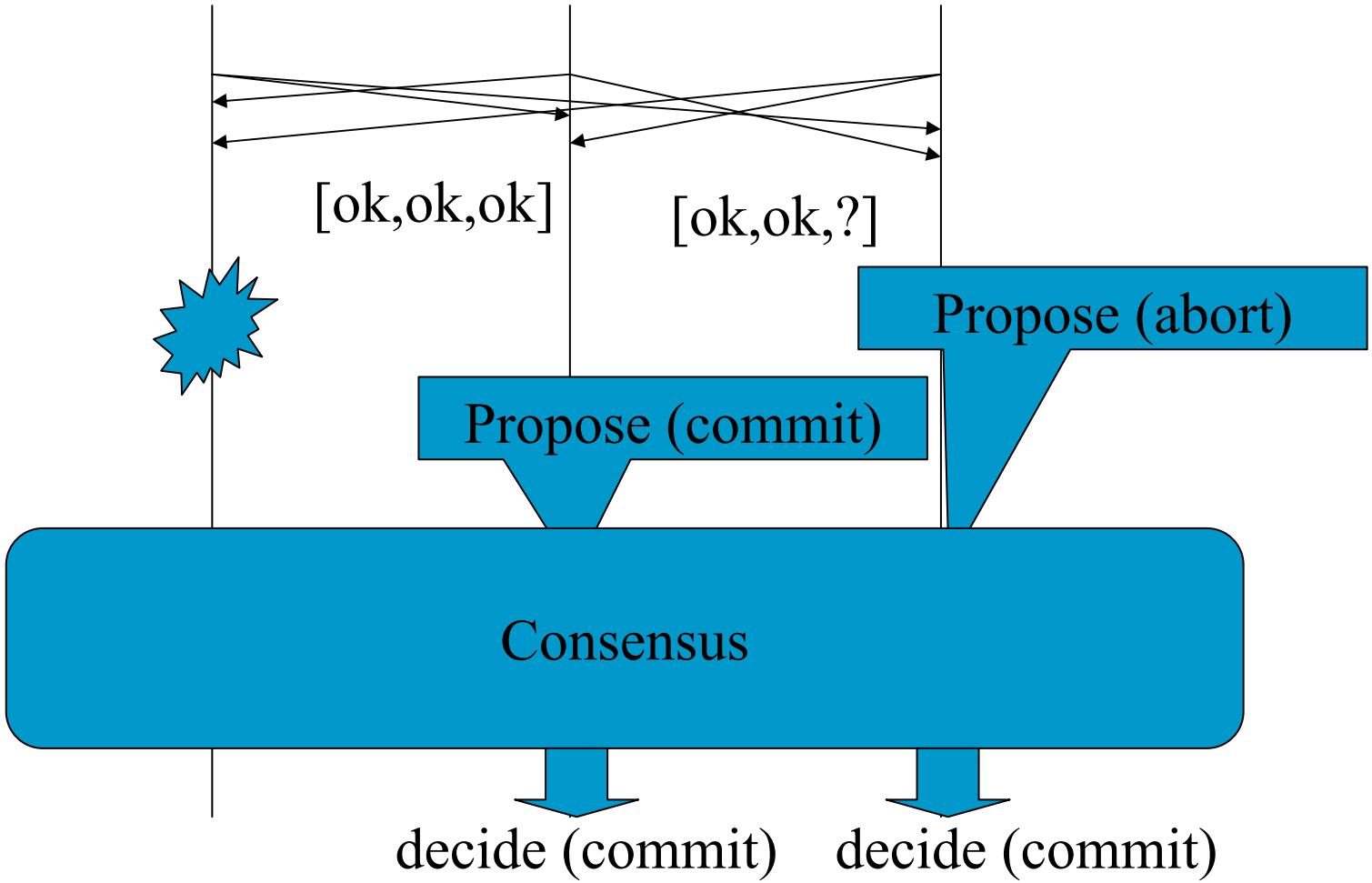
Non-blocking atomic commitment



Non-blocking atomic commitment



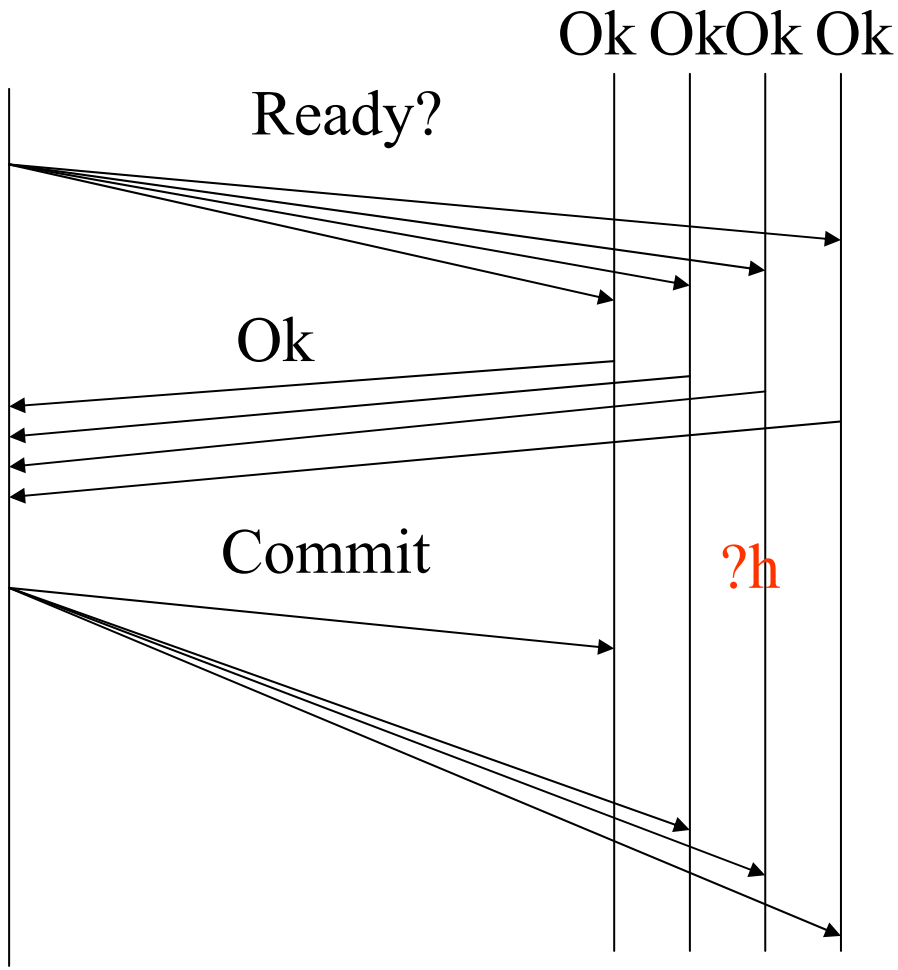
Non-blocking atomic commitment



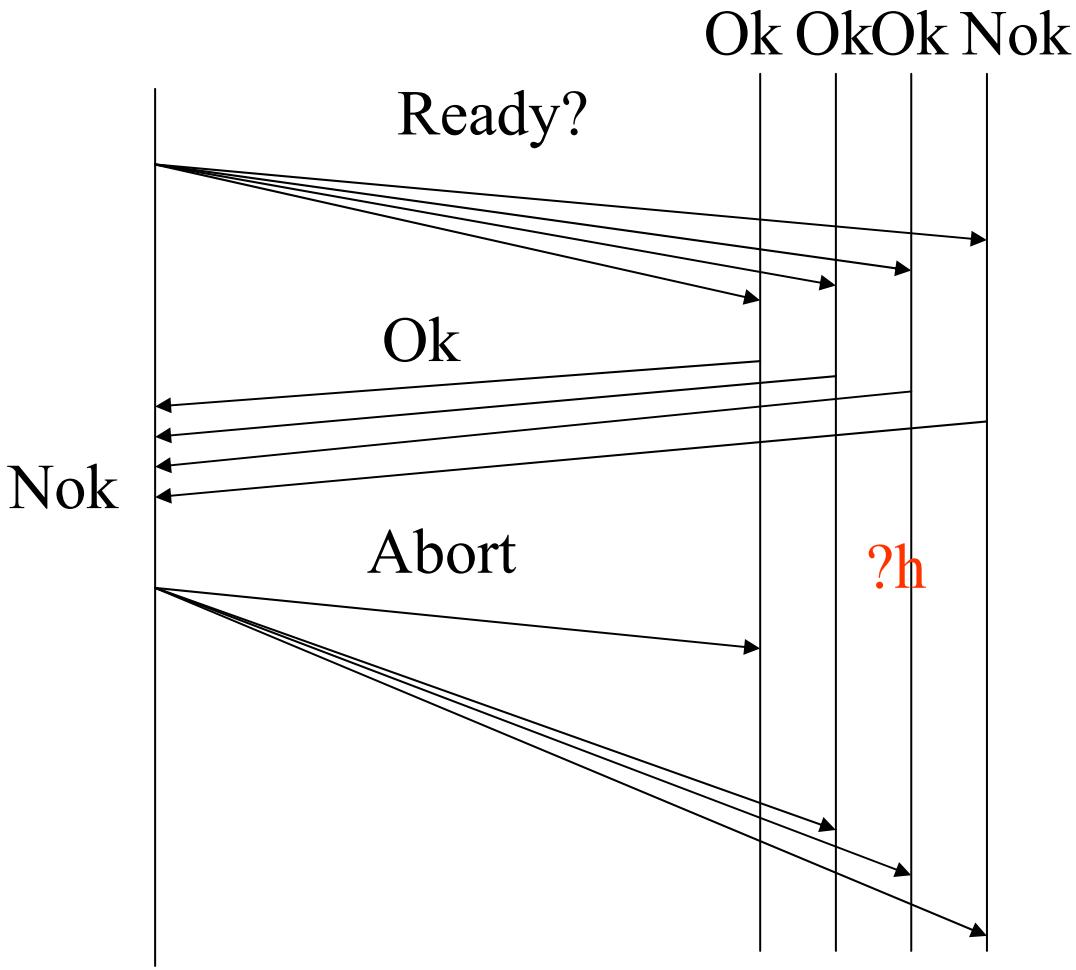
Consensus based atomic commitment vs two-phase commit

- Two-phase commit is a widely used protocol that implements atomic commitment
- It is coordinator based
- If the coordinator fails, unlike the solution above, two-phase commits blocks!

Two-phase commit



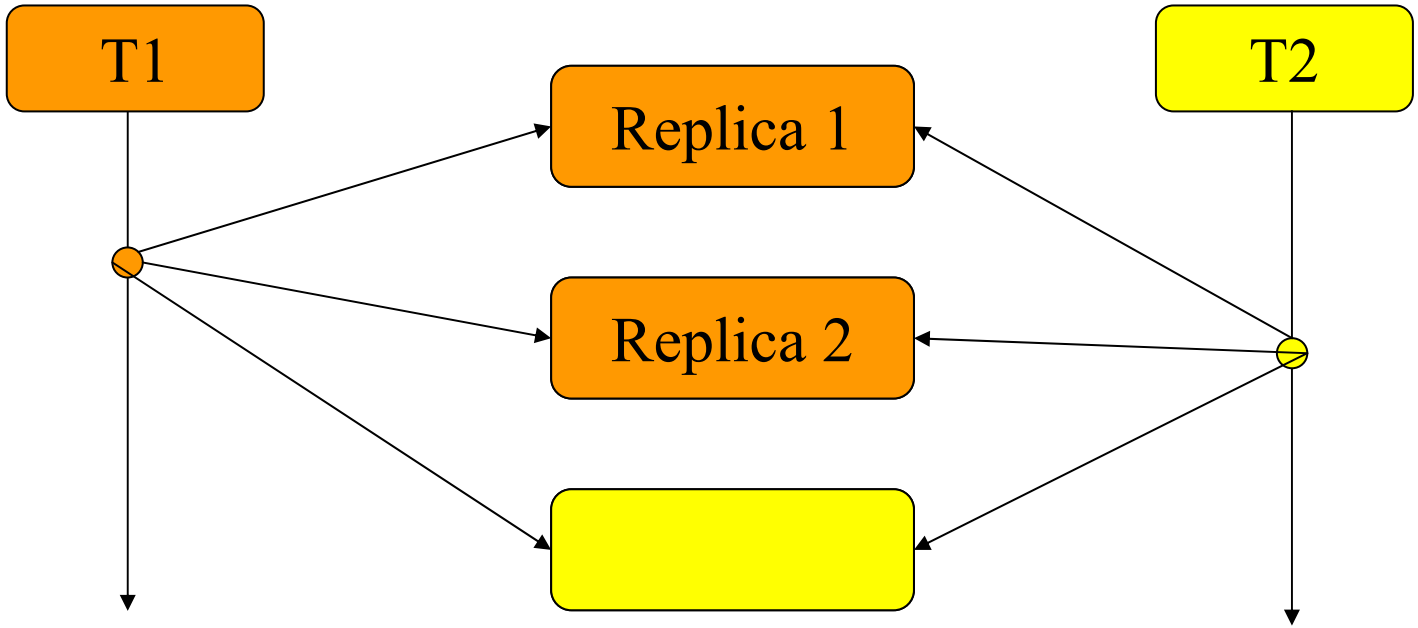
Two-phase commit



NBAC enough?

- Read-one/ write-all + distributed atomic commitment can ensure correctness of replicated database
- However, this solution may suffer from performance problems
  - When multiple transactions try to access the same data

Distributed deadlock



Replication and deadlock

- Even with a single data item, replication may cause a distributed deadlock
- This may occur if different transactions lock different replicas in different orders
- Solution:
  - Ensure that all replicas are accessed in the same order!

Roadmap

- ***(1) Overview and basic abstractions***
  - Overview
  - Broadcast
  - Consensus
- ***(2) Advanced abstractions***
  - Motivation: database replication
  - Atomic commit
  - Total order broadcast

Total order broadcast

- Broadcast primitive that ensures that all processes receive the same set of messages in exactly the same order
- Also called atomic broadcast
- Can also be expressed as a variant of consensus
  - Processes need to agree on the order they will deliver messages

Total order

Validity: If pi and pj are correct, then every message broadcast by pi is eventually delivered by pj

No duplication: No message is delivered more than once

No creation: No message is delivered unless it was broadcast

(Uniform) Agreement: For any message m. If a correct (any) process delivers m, then every correct process delivers m

Total order (cont'd)

(Uniform) Total order:

Let m and m' be any two messages.

Let pi be any (correct) process that delivers m without having delivered m'

Then no (correct) process delivers m' before m

Total order broadcast: algorithm

- Each process keeps two sets of messages:
  - Unordered
  - Ordered
- Messages are first disseminated using an (unordered) uniform reliable broadcast primitive:
  - This ensures that all processes receive the same messages.
  - Messages received this way are placed in the unordered set.

Total order broadcast: algorithm

- How to move from the unordered to the ordered set?
  - Use a sequence of consensus
- First instance of consensus decides which messages will take sequence number 1
- Second instace of consensus decides which messages will take sequence number 2
- Etc

Total order broadcast: algorithm

while (true) **do**

wait until *unordered* set is not empty

sn := *sn* + 1

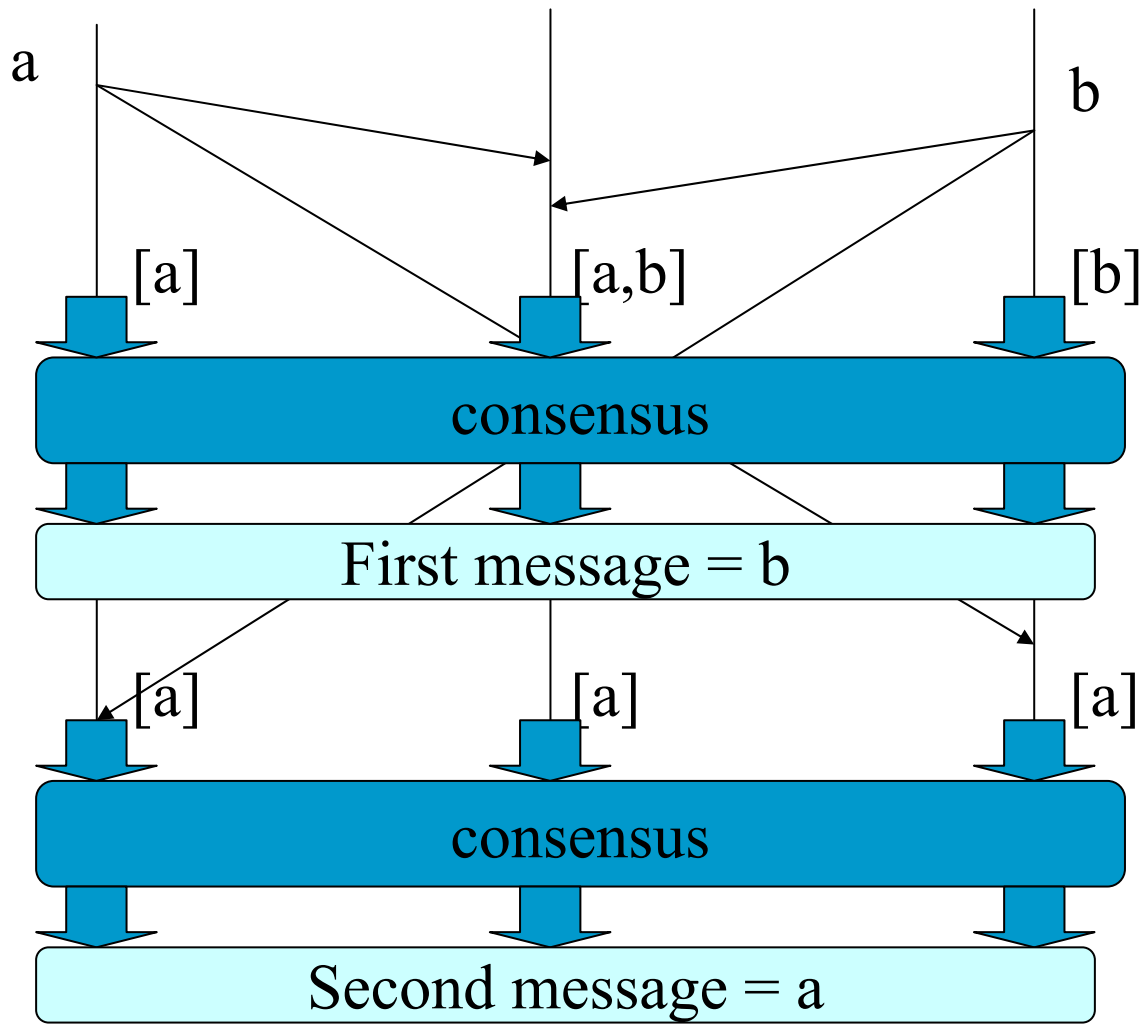
decision = propose (*sn, unordered*);

deliver (decision) // in deterministic order

unordered := unordered \ decision

ordered := ordered + decision

Algorithm


a
b
[a]
[a,b]
[b]
consensus
First message = b
[a]
[a]
[a]
consensus
Second message = a

Algorithm (toBroadcast)

Implements: TotalOrder (to).

Uses:

ReliableBroadcast (rb).

Consensus (cons);

upon event < Init > **do**

unordered = delivered = empty;

wait := false;

sn := 1;

Algorithm (toBroadcast)

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 (unordered not empty) and not(wait) **do**

  wait := true:

  trigger < Propose, unordered>sn;

Algorithm (toBroadcast)

```
upon event <Decide,decided>sn do
    unordered := unordered \ decided;
    ordered := deterministicSort(decided);
    for all (sm,m) in ordered:
        trigger < toDeliver,sm,m>;
´       delivered := delivered U {m};
    sn : = sn + 1;
    wait := false;
```

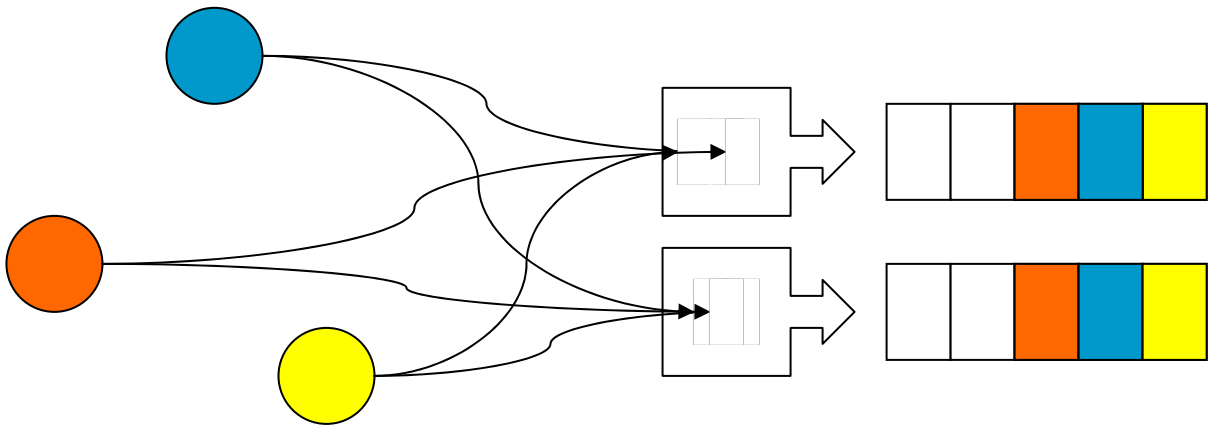
Replicated state-machine

- The replicated state-machine approach is a fundamental technique to replicated deterministic servers
- A server is characterized by a state-machine if, at any point, it state depends only of:
  - Its initial state
  - The sequence of (deterministic) commands it has executed

Replicated state-machine

- A state-machine can be replicated by:
  - Running a replica of the state machine in a different node
  - Make all replicas have the same initial state
  - Distribute all commands using a total order broadcast

Replicated state-machine



Database state-machine (DBSM)

- Change the state machine scheduler to ensure deterministic processing of SQL commands
- Broadcast SQL commands in total order to all replicas

Database state-machine
(DBSM, middleware solution)

- To avoid changing the database…
- Introduce replicated proxy between clients and the database.
- Clients send SQL commands to proxies using atomic broadcast
- Each proxy submits write operations, one by one, to the attached database.

Disadvantage of DBSM

- Requires the execution of a total order broadcast every time the client executes a SQL command.
- Excessive use of total order broadcast may limit the throughput of the system.

Optimistic approaches

- Transactions are executed in a single replica without explicit synchronization with the remaining replicas until commit time.
- At commit time, control information is sent in a single atomic broadcast to all replicas.
- Based on the control information, all replicas apply a deterministic certification step.
- The certification step checks if the transaction is serializable.
  - Depending on the control information, the certification step may require further communication.
  - The certification step decides the fate of the transaction (commit or abort).

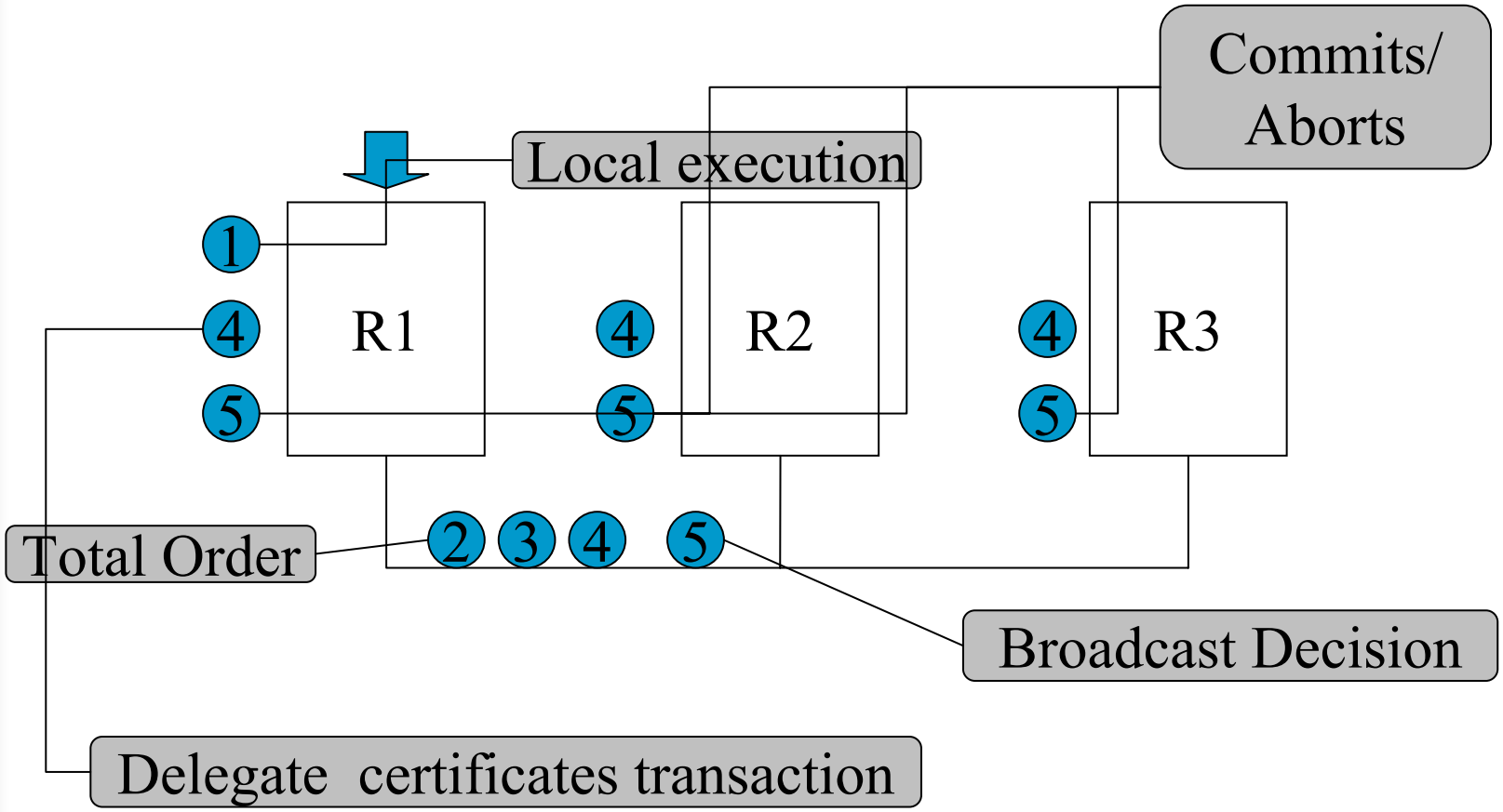
Optimistic approach (non-voting)

- Transactions execute locally in a node called the delegate node.
  - Different transactions may execute in different delegate nodes.
  - Locks are acquired locally at the delegate node.
- When the transaction is ready to commit, read and write sets are sent to all replicas using atomic broadcast.
- Total order defines precedence order for concurrent transaction.
  - If read set is no longer valid, all replicas abort.
  - If read set is still valid, all replicas commit and apply the write set.

Optimistic approach (voting)

- (same as before)
- When the transaction is ready to commit, write set is sent to all replicas using atomic broadcast.
- Total order defines precedence order for concurrent transaction.
- Delegate node checks the read set.
  - If read set is no longer valid, decides abort.
  - If read set is still valid, decides commit.
- Decision is broadcast to all replicas.

Optimistic approach



Database replication in the real-world

- Most commercial solutions only offer lazy-replication
  - Updates are applied in a master copy and propagated later to other copies
  - Does not ensure strong consistency
  - Updates may be lost or become unavailable

Database replication in the real-world

- Due to the distributed deadlock problem, synchronous replication typically works as follows:
  - All updates are serialized in a single replica and immediately copied to the backups
  - Two-phase commit is used
- Disadvantages:
  - Suffer from the blocking problem
  - To not take advantage of multiple nodes for processing updates

Database replication in the real-world

- Recent approaches take advantages of the abstractions presented in this tutorial:
  - Emic offers a commercial solution for MySQL that implements the state machine approach
  - INRIA has developed a middleware solution for clusters of databases using a similar approach

Concluding remark (1)

- Some people tend to dismiss the use of distributed programming abstractions due to performance costs.
- In fact, some of the abstraction described in this tutorial are expensive.
- There are known lower bounds for the number of steps required to solve some problems:
  - The cost is not an artifact of the implementation

Concluding remark (1)

- Therefore:
  - If you do not need an abstraction, don't use it!
- Always use the weaker abstract needed by your application:
  - Check if it possible to weaken the requirements.
- On the other hand…

Concluding remark (1)

- If you do need an abstraction, use it!
  - The problem is not going to vanish just because you decide to implement the solution entangled with the application code.
  - Lower bounds will still be there.
  - Avoid the “denial trap”.

Concluding remark (1)

- The implementation of these abstractions is difficult
  - There are many subtle issues, some of them have been addressed in this tutorial.
- The use of a modular approach simplifies the task of optimizing the performance of the system
  - By selecting the most appropriate implementation of the abstraction.

Concluding remark (2)

- Historically, abstractions have been first experimented as ***libraries*** of programming languages and then became, in some form, ***constructs*** of the language
  - ***Example:*** from concurrency libraries of C++ to synchronized methods in Java

Concluding remark (2)

- We provide a ***library*** or distributed programming abstractions: ***Appia***
- Appia is the descendent of ***Isis***, ***Horus***, ***Amoeba***, ***Transis***, ***Totem***, ***Arjuna***, ***Bast***, ***Psync***
- What about a (reliable) distributed programming language?

Concluding remark (3)

- We considered in this tutorial a simple (if not simplistic) model: fail-stop
- The same abstractions have also been studied (in our manuscript and others) in other models:
  - fail-silent, fail-noisy, fail-recovery, randomized, Byzantine