Research Group
Distributed Systems

Christian-Albrechts-Universität zu Kiel
Technische Fakultät

Distributed Systems

Fault-Tolerance III

Olaf Landsiedel

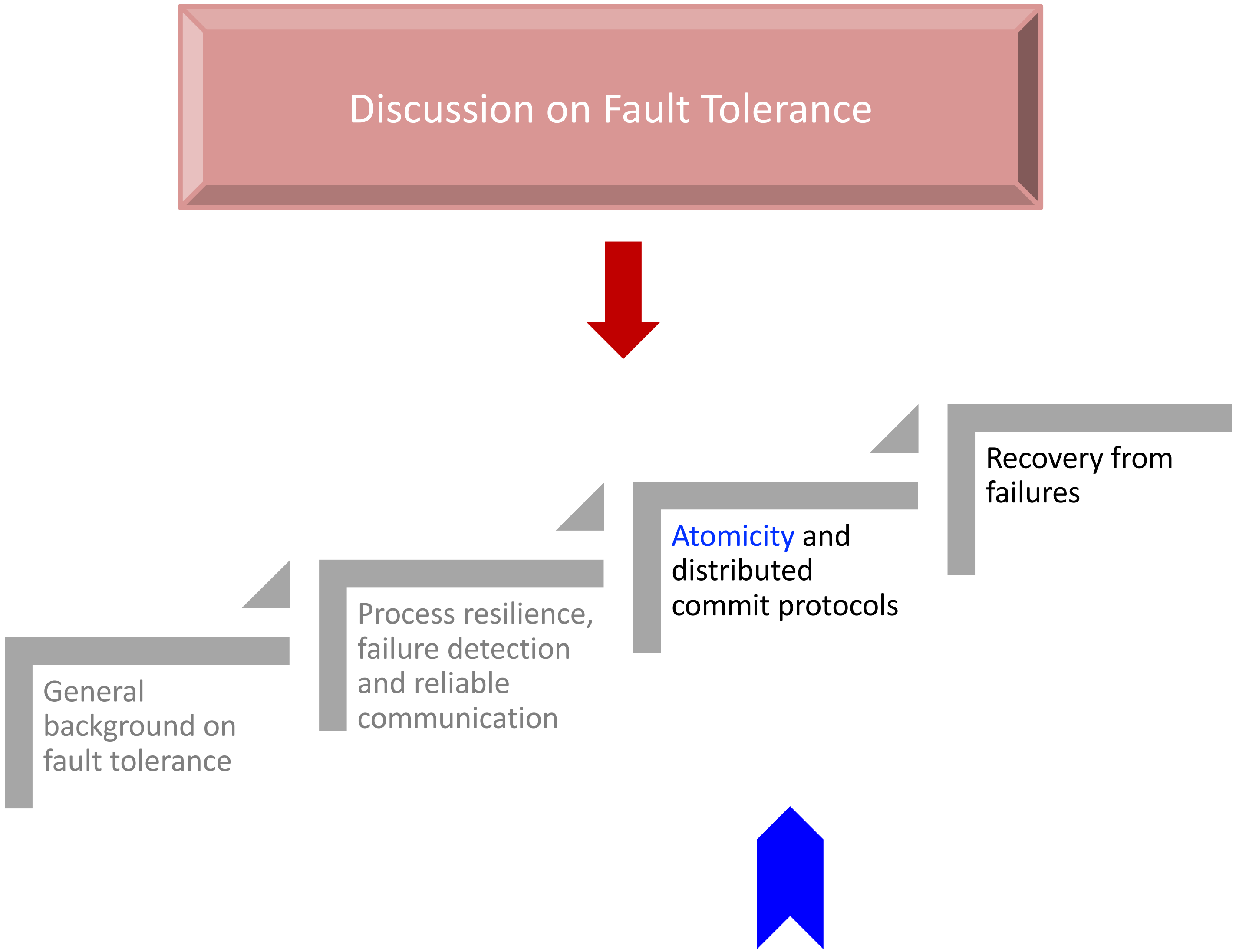
Last Time

- Fault Tolerance II
  - Reliable Client Server Communication
  - 2 Generals Problem
  - Reliable Group Communication
    - A Basic Reliable-Multicasting Scheme
    - Scalability in Reliable Multicasting

Reliable Group Communication

- A Basic Reliable-Multicasting Scheme
- Scalability in Reliable Multicasting
- Atomic Multicast

Objectives


Discussion on Fault Tolerance
General background on fault tolerance
Process resilience, failure detection and reliable communication
Atomicity and distributed commit protocols
Recovery from failures

Atomic Multicast

- What is often needed in a distributed system: Guarantee, that
  1. message is delivered to all processes or to none
  2. all messages are delivered in the same order to all processes

- Satisfying 1 and 2: known as atomic multicast

- Atomic multicast:
  – Ensures that non-faulty processes maintain a consistent view
  – Forces reconciliation when a process recovers and rejoins the group

Virtual Synchrony (1)

- Virtual Synchrony provides atomic multicast
- In Virtual Synchrony
  - A multicast message *m* is uniquely associated with a list of processes to which it should be delivered
  - This delivery list corresponds to a *group view (G)*

- There is only one case in which delivery of *m* is allowed to fail:
  - When a group-membership-change is the result of the sender of *m* crashing
  - In this case, *m* may either be delivered to all remaining processes, or ignored by each of them
    - Note: still consistent outcome

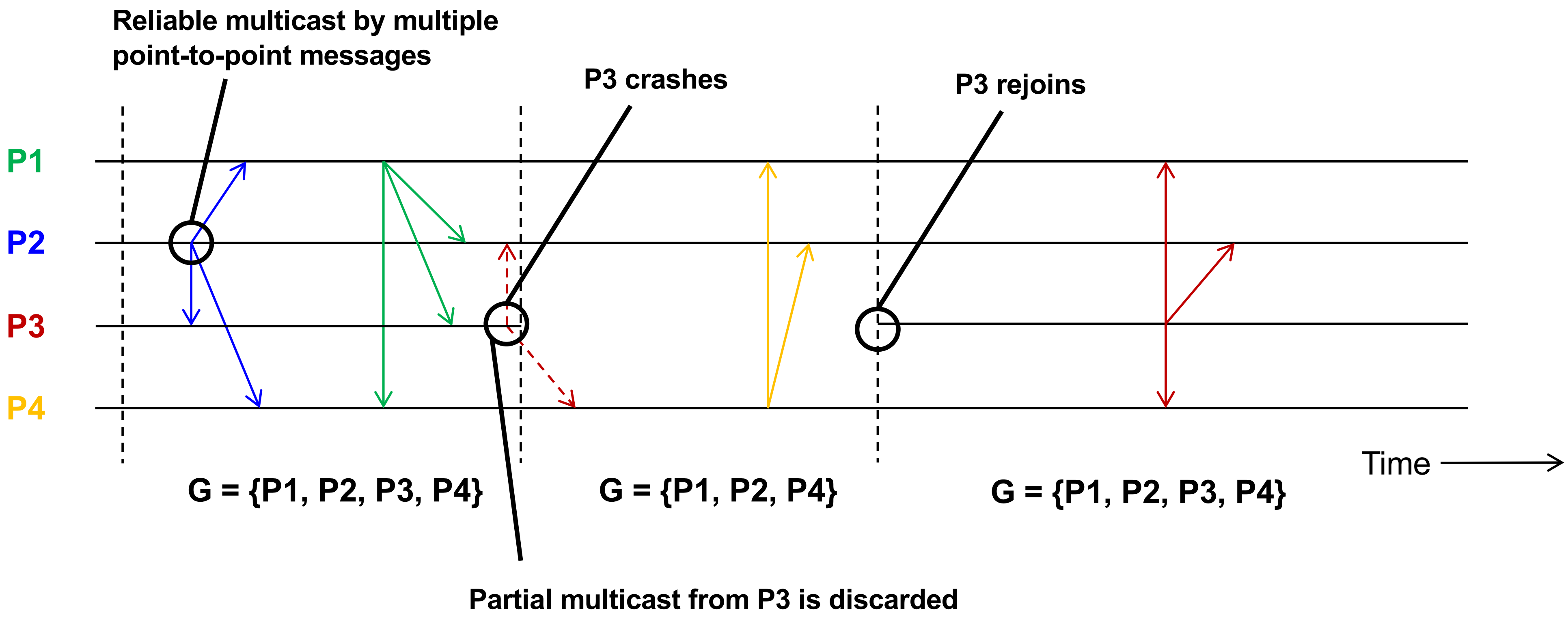
Virtual Synchrony (1)

- Virtual Synchrony provides atomic multicast
- In Virtual Synchrony
  - A multicast message *m* is uniquely associated with a list of processes to which it should be delivered
  - This delivery list corresponds to a *group view (G)*

- There is only one case in which delivery of *m* is allowed to fail:
  - When a group-membership-change is the result of the sender of *m* crashing

A reliable multicast with this property is said to be
virtually synchronous

Virtual Synchrony (2)



The Principle of Virtual Synchronous Multicast

Implementing Virtual Synchrony (1)

- Example
  - *Isis* [Birman et al. 1991]
  - One of many possible implementation of virtual synchrony
- Isis uses / assumes
  - assumes a FIFO-ordered multicast
  - uses TCP: each transmission is guaranteed to succeed
  - but: Using TCP does not guarantee that all messages sent to a view G are delivered to all non-faulty processes in G before any view change

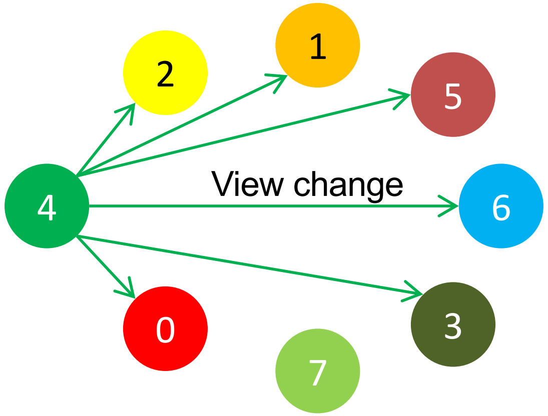
Implementing Virtual Synchrony (2)

- Idea
  - Node sends message m
  - every process in G buffers a message m until
    - it knows for sure that all members in G have also received it
  - If m has been received by all members in G
    - m is said to be stable
      - Unstable message: message that is not yet by all members
    - Only stable messages are allowed to be delivered

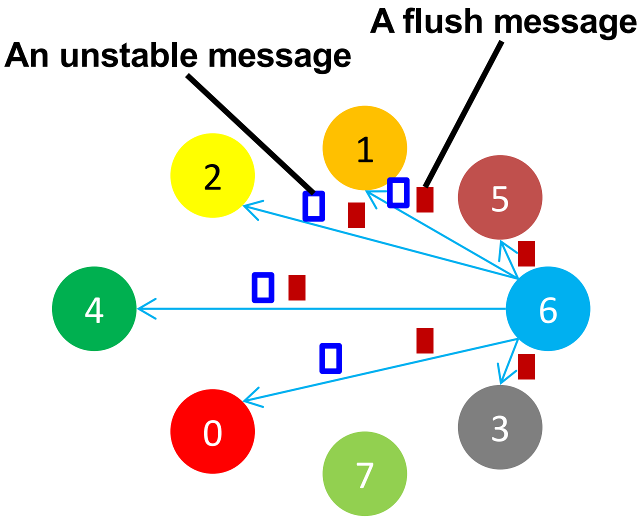
Implementing Virtual Synchrony (3)

- Node crash
  - How to make sure: that
    - *m* may either be delivered to all remaining processes
    - or ignored by each of them

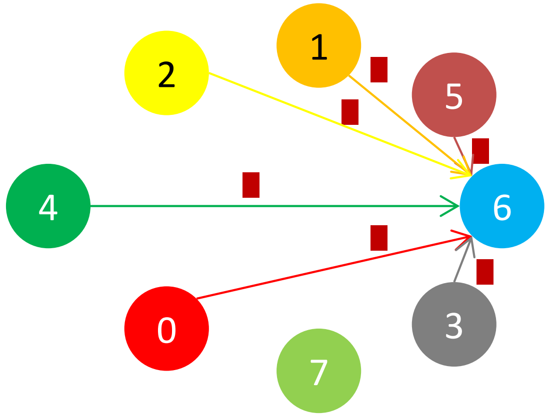
Implementing Virtual Synchrony (4)



Process 4 notices that process 7 has crashed and sends a view change



Process 6 sends out all its unstable messages, followed by a flush message



Process 6 installs the new view when it receives a flush message from everyone else

Implementing Virtual Synchrony (5)

- Why does this work?
  - FIFO properties
    - Upon receiving flush message: no previous message from this anymore in transit
  - Once flush msg received from all
    - Safe to assume: no other messages from any other node in transition
  - Where did we see this before?
    - Chandy-Lamport Protocol (Clocks and Time II)

Message Ordering

- Four virtually synchronous multicast orderings
  - *Unordered* multicasts
  - *FIFO-ordered* multicasts
  - *Causally-ordered* multicasts
  - *Totally-ordered* multicasts

Message Ordering

- Four virtually synchronous multicast orderings
  - *Unordered* multicasts
  - *FIFO-ordered* multicasts
  - *Causally-ordered* multicasts
  - *Totally-ordered* multicasts

FIFO order: TCP like communication channels

”Same order on all nodes only for causally related msg.”

“Same order on all nodes”

1. Unordered Multicasts

- What is a Unordered Multicast?
  – What do you expect?

1. Unordered Multicasts

- A reliable, unordered multicast
  - a virtually synchronous multicast
  - With no guarantees on the order in which received messages are delivered by different processes

Process P1	Process P2	Process P3
Sends m1	Receives m1	Receives m2
Sends m2	Receives m2	Receives m1

Three communicating processes in the same group

2. FIFO-Ordered Multicasts

- What is a FIFO-Ordered Multicast?
  - What do you expect?

2. FIFO-Ordered Multicasts

- A FIFO-Ordered multicast,
  - deliver incoming messages from the same process in the same order as they have been sent

Process P1	Process P2	Process P3	Process P4
Sends m1	Receives m1	Receives m3	Sends m3
Sends m2	Receives m3	Receives m1	Sends m4
	Receives m2	Receives m2	
	Receives m4	Receives m4	

Four processes in the same group with two different senders.

3-4. Causally-Ordered and Total-Ordered Multicasts

- What is a Causally-Ordered Multicast?
  - What do you expect?

- What is a Total-Ordered Multicast?
  - What do you expect?

3-4. Causally-Ordered and Total-Ordered Multicasts

- *Causally-ordered* multicast
  - preserves causality between messages
  - if message m1 causally precedes message m2,
    - regardless of whether they are by the same sender or not,
    - we will always deliver m1 before m2

- *Total-ordered* multicast
  - when messages are delivered, they are delivered in the same order to all group members
  - (regardless of whether message delivery is unordered, FIFO-ordered, or causally-ordered)

Virtually Synchronous Reliable Multicasting

- Atomic multicast
  - virtually synchronous reliable multicasting
  - with total-ordered delivery of messages

Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Six different versions of virtually synchronous reliable multicasting

Summary Virtual Synchrony & Atomic Multicast

- Atomic Multicast
  - Message to all or none
  - Can be realized by Virtual Synchrony

Objectives

Discussion on Fault Tolerance

General background on fault tolerance

Process resilience, failure detection and reliable communication

Atomicity and distributed commit protocols

Recovery from failures

Distributed Commit

- Atomic multicasting problem is an example of a more general problem, known as *distributed commit*

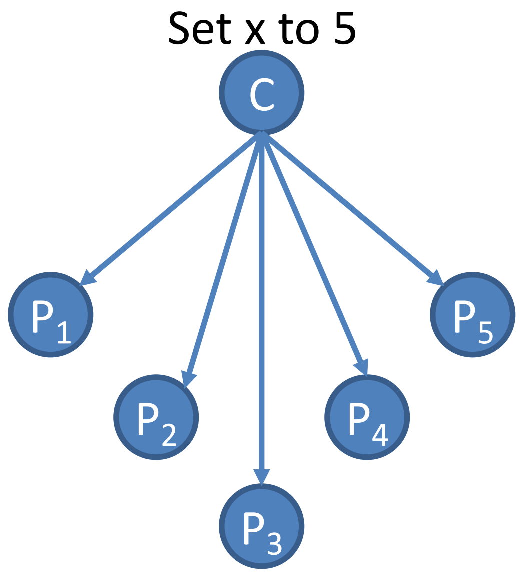
- Distributed commit: operation performed by
  - either each member of a process group
  - or none at all

- Example: Reliable multicasting
  - the operation is the delivery of a message

- Example: distributed transactions
  - the operation may be the commit of a transaction

Commit Protocols

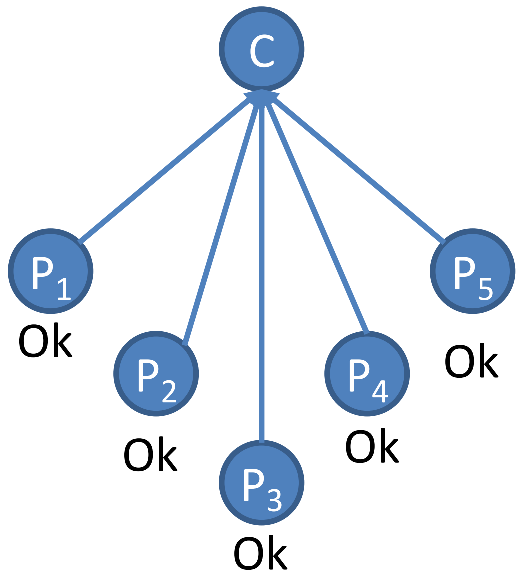
- Discuss Commit Protocols
  - One-Phase Commit (1PC)
  - Two-Phase Commit (2PC)
  - Three-Phase Commit (3PC)



- Common Setting
  - Coordinator (C), Participants (P_1 to P_n)

Commit Protocols

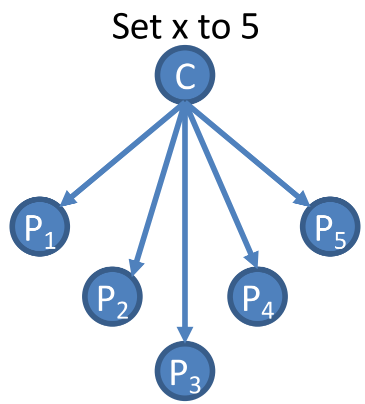
- Discuss Commit Protocols
  - One-Phase Commit (1PC)
  - Two-Phase Commit (2PC)
  - Three-Phase Commit (3PC)

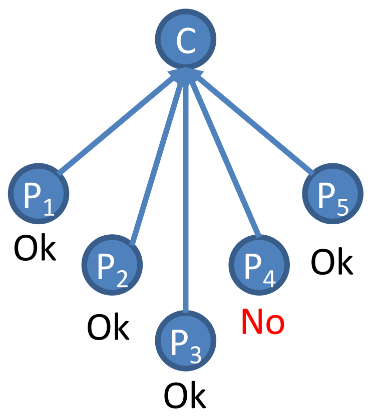


- Common Setting
  - Coordinator (C), Participants (P_1 to P_n)

One-Phase Commit Protocol

- Coordinator: tell participants to (locally) perform an operation
  - One-phase commit protocol
- Problem?
  - if one of the participants cannot perform the operation
- Possible reasons:
  - Write x to disk: out of memory, disk crashed, …
  - Set speed to 120: maximum speed of this engine of 100
- In practice, more sophisticated schemes are needed
  - The most common: Two-phase commit protocol

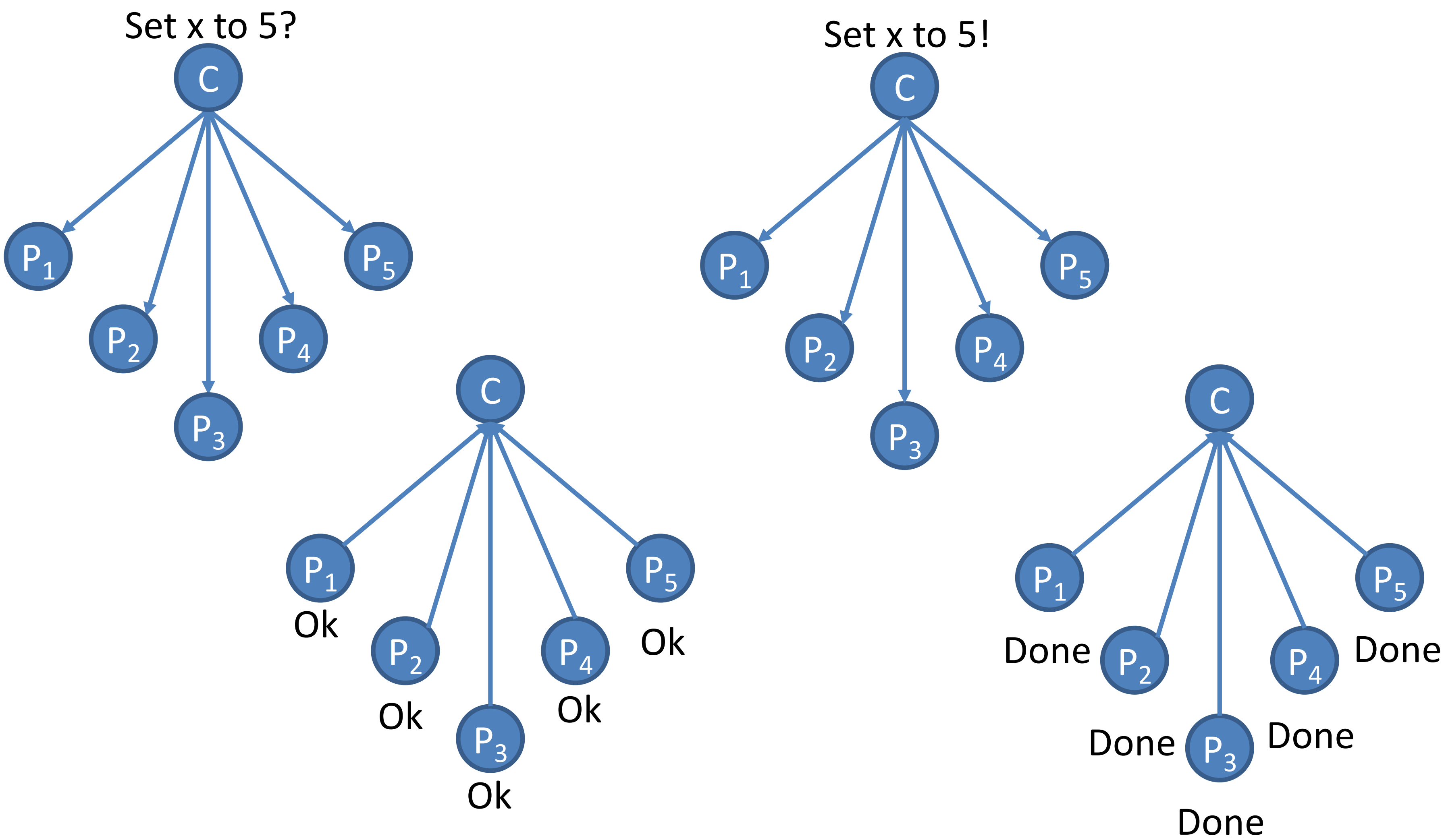




Two-Phase Commit Protocol

- Two phases
  - 1. Ask all if they are fine to commit to something
    - Prepare phase
  - 2. Commit (or not)

Two-Phase Commit Protocol



Two-Phase Commit Protocol

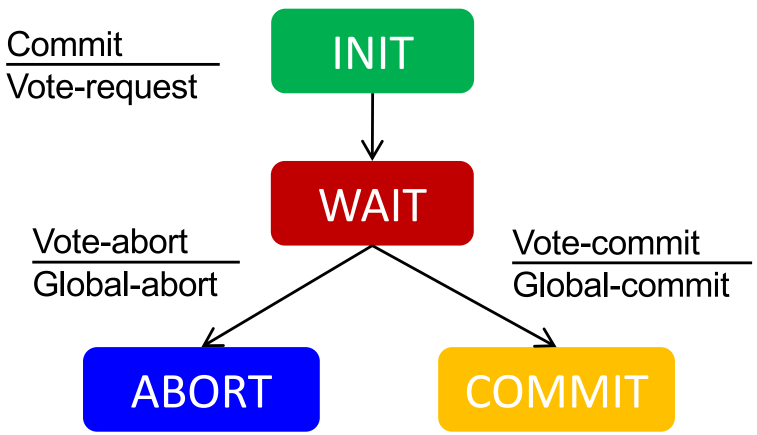
- Two-phase commit protocol (2PC)
  - two phases, each consisting of two steps (if no failures occur): Phase I and Phase II

Phase I: Voting Phase	
Step 1	• The coordinator sends a VOTE_REQUEST message to all participants.
Step 2	• When a participant receives a VOTE_REQUEST message, • it returns either a VOTE_COMMIT message to the coordinator indicating that it is prepared to locally commit its part of the transaction, • or otherwise a VOTE_ABORT message.

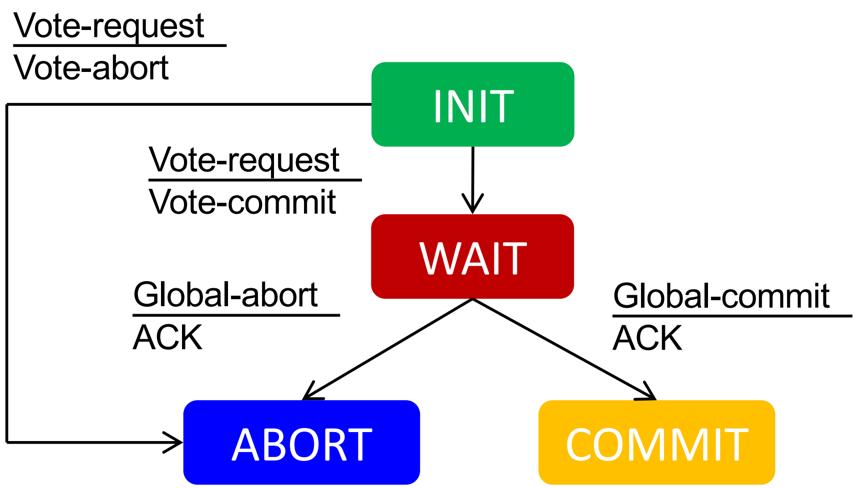
Two-Phase Commit Protocol

Phase II: Decision Phase	
Step 1	• The coordinator collects all votes from the participants. • If all participants have voted to commit the transaction, then so will the coordinator. In that case, it sends a GLOBAL_COMMIT message to all participants. • However, if one participant had voted to abort the transaction, the coordinator will also decide to abort the transaction and multicasts a GLOBAL_ABORT message.
Step 2	• Each participant that voted for a commit waits for the final reaction by the coordinator. • If a participant receives a GLOBAL_COMMIT message, it locally commits the transaction. • Otherwise, when receiving a GLOBAL_ABORT message, the transaction is locally aborted as well.

2PC Finite State Machines



The finite state machine for the *coordinator* in 2PC



The finite state machine for a *participant* in 2PC

Two-Phase Commit Protocol

- Ensures that all nodes can commit
  - Before asking for the actual commit

- Problem?
  - What happens if a node crashes between the two phases?

Two-Phase Commit Protocol

- What happens if?
  - A participant crashes
    - Before voting?
      - Coordinator will not get that vote, system will halt
    - After voting, before commit received?
      - Other nodes will commit, but not the crashed one
    - After commit?
      - Commit got executed…
  - Coordinator crashes
    - After participants voted, but before results are send out?
      - System will halt

Two-Phase Commit Protocol

Actions by coordinator:

```
write START_2PC to local log;
multicast VOTE_REQUEST to all participants;
while not all votes have been collected{
        wait for any incoming vote;
        if timeout{
                write GLOBAL_ABORT to local log;
                multicast GLOBAL_ABORT to all participants;
                exit;
        }
        record vote;
}
If all participants sent VOTE_COMMIT and coordinator votes COMMIT{
        write GLOBAL_COMMIT to local log;
        multicast GLOBAL_COMMIT to all participants;
}else{
        write GLOBAL_ABORT to local log;
        multicast GLOBAL_ABORT to all participants;
}
```

Two-Phase Commit Protocol

Actions by participants:

```
write INIT to local log;
Wait for VOTE_REQUEST from coordinator;
If timeout{
        write VOTE_ABORT to local log;
        exit;
}
If participant votes COMMIT{
        write VOTE_COMMIT to local log;
        send VOTE_COMMIT to coordinator;
        wait for DECISION from coordinator;
        if timeout{
                multicast DECISION_RQUEST to other participants;
                wait until DECISION is received; /*remain blocked*/
                write DECISION to local log;
        }
        if DECISION == GLOBAL_COMMIT { write GLOBAL_COMMIT to local log;}
        else if DECISION == GLOBAL_ABORT {write GLOBAL_ABORT to local log};
}else{
        write VOTE_ABORT to local log;
        send VOTE_ABORT to coordinator;
}
```

Two-Phase Commit Protocol

Actions for handling decision requests:

```
/*executed by separate thread*/

while true{
        wait until any incoming DECISION_REQUEST is received; /*remain blocked*/
        read most recently recorded STATE from the local log;
        if STATE == GLOBAL_COMMIT
                send GLOBAL_COMMIT to requesting participant;
        else if STATE == INIT or STATE == GLOBAL_ABORT
                send GLOBAL_ABORT to requesting participant;
        else
                skip; /*participant remains blocked*/
}
```

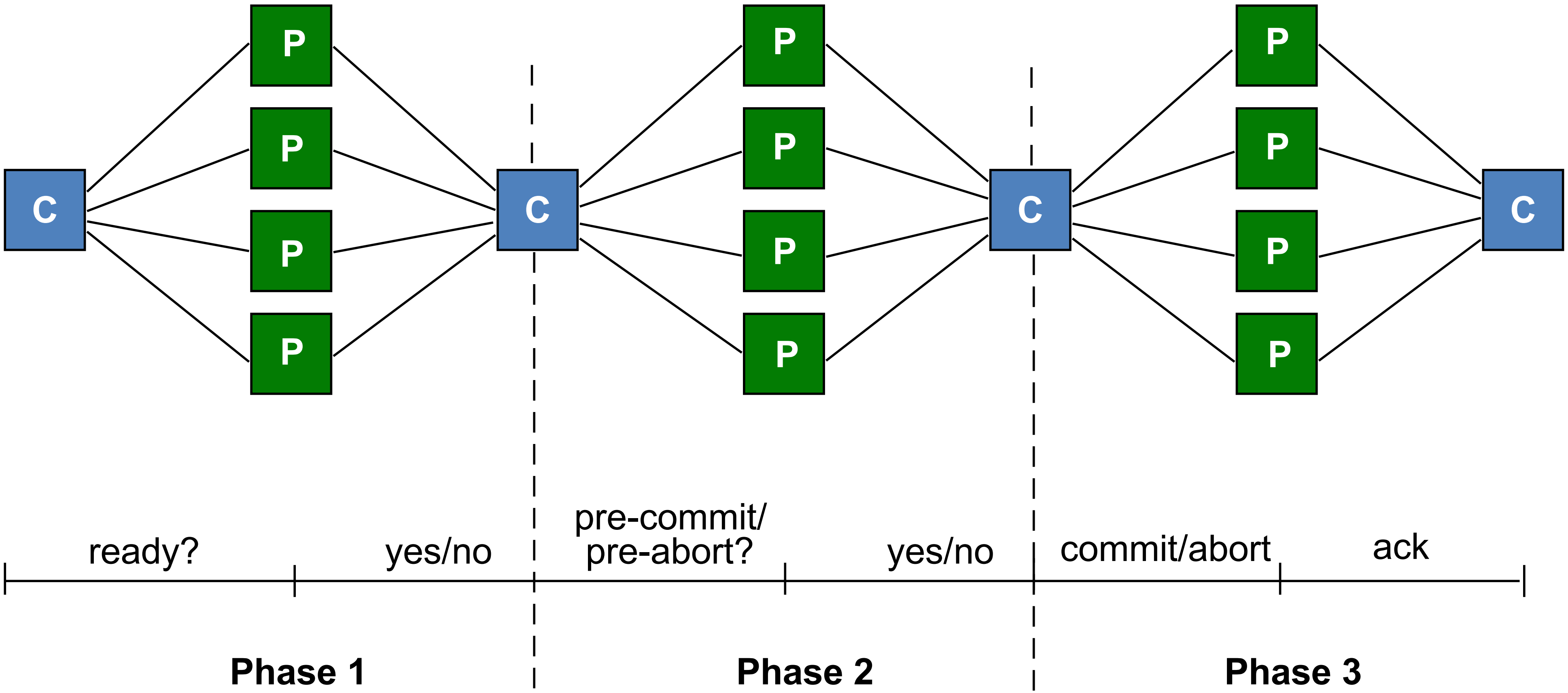
Problem With 2PC

- Blocking
  - Ready implies that the participant waits for the coordinator
  - If coordinator fails, site is blocked until recovery
  - Blocking reduces availability
- Independent recovery is not possible
- However, it is known that:
  - Independent recovery protocols exist only for single site failures; no independent recovery protocol exists which is resilient to multiple-site failures.

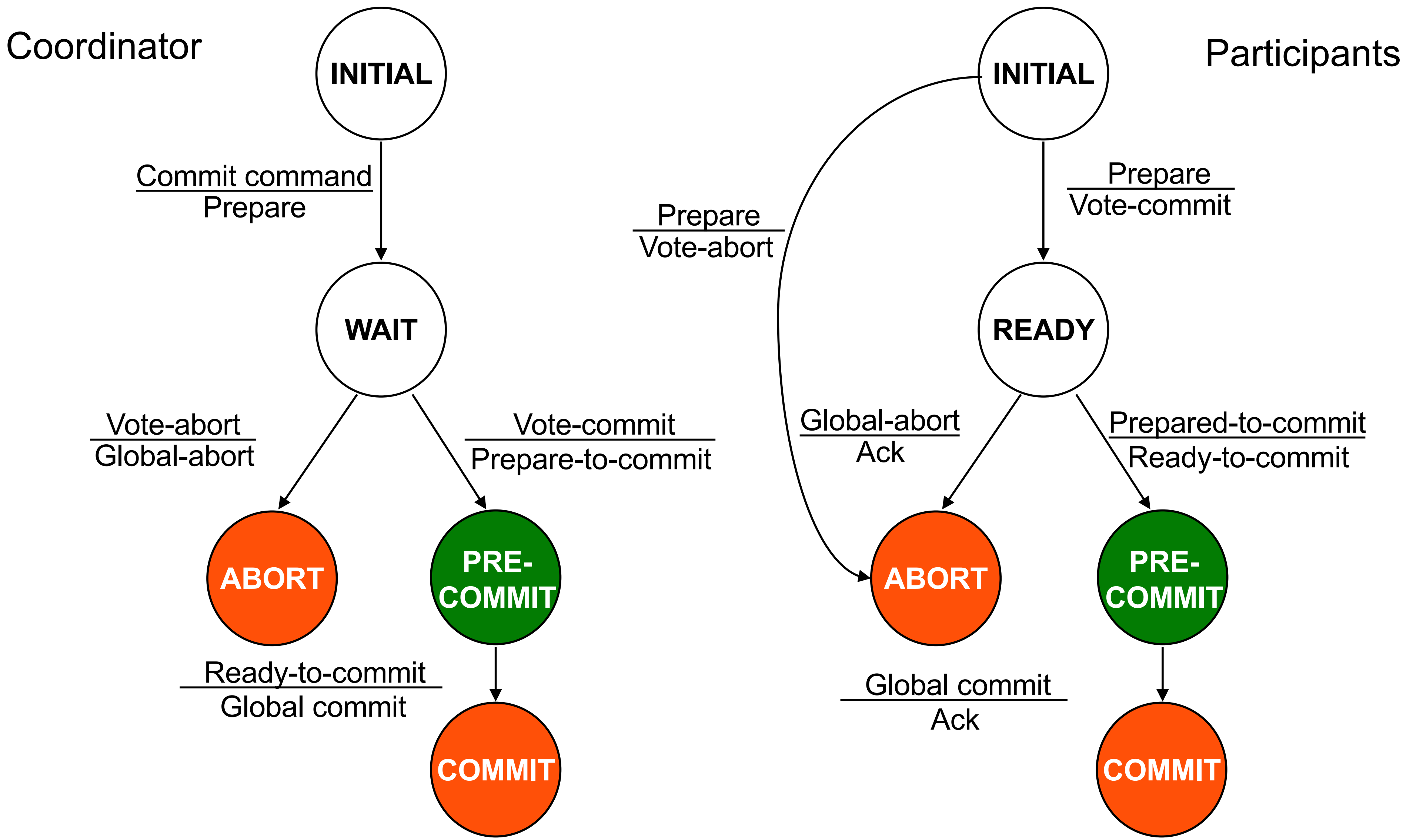
Three-Phase Commit (3PC)

- Three-Phase Commit (3PC)
  - Adds a third phase
    - 1. Ask all if they are fine to commit to something
      - Prepare phase
    - 2. Pre-Commit (or not)
    - 3. Commit
  - Removes the blocking problem
    - 3PC is non-blocking

Three-Phase Commit (3PC)



State Transitions in 3PC



Three-Phase Commit (3PC)

- If the coordinator fails before sending preCommit messages,
  - the cohort will agree that the operation was aborted.
- The coordinator will not send out a doCommit message until all cohort members have **ACK**ed that they are **Prepared to commit**.
  - This eliminates the possibility that any cohort member actually completed the transaction before all cohort members were aware of the decision to do so
  - (an ambiguity that necessitated indefinite blocking in the two-phase commit protocol).

But

- Network partition
  – imagine that all the replicas that received ‘prepare to commit’ are on one side of the partition,
  – and those that did not are on the other.
- Then both partitions will continue with recovery nodes that respectively commit or abort the transaction
  – and when the network merges the system will have an inconsistent state.
- 3PC has potentially unsafe runs, as does 2PC,
  – but will always make progress and therefore satisfies its liveness properties

Objectives

Discussion on Fault Tolerance

General background on fault tolerance

Process resilience, failure detection and reliable communication

Atomicity and distributed commit protocols

Recovery from failures

Recovery

- Focus so far,
  - concentrated on algorithms that allow us to tolerate faults
- After failure
  - Failed process should recover to a correct state
- We focus on:
  - What it actually means to recover to a correct state
  - When and how the state of a distributed system can be recorded and recovered
  - check-pointing and message logging

Recovery

- Error Recovery
- Check pointing
- Message Logging

Error Recovery

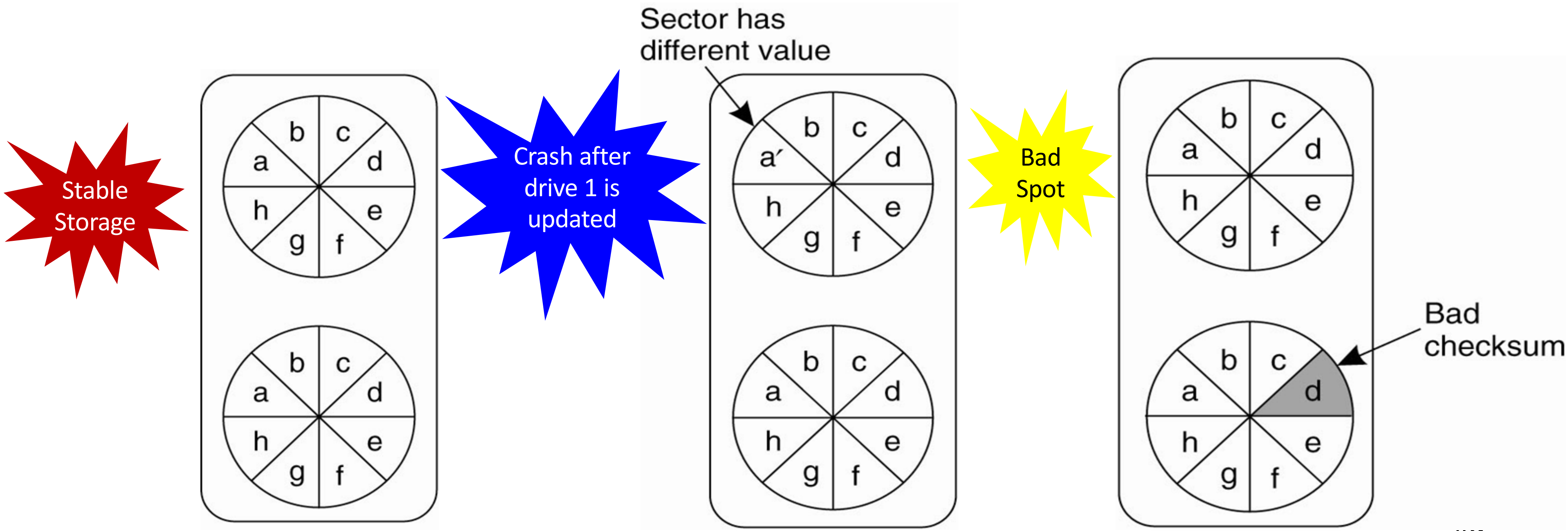
- Fundamental for fault tolerance: recovery from an error
- Once a failure has occurred
  - process where the failure has happened: recover to a correct state

- The idea of error recovery is to replace an erroneous state with an error-free state

- Two forms of error recovery:
  - Backward recovery
  - Forward recovery

1. Backward Recovery (1)

- In backward recovery:
  - from a erroneous state back to a previously correct state
- Record the system's state from time to time onto a stable storage,
  - restore such a recorded state when things go wrong



1. Backward Recovery (2)

- Checkpoint: each time (part of) the system's present state is recorded

- Problems with backward recovery:
  - Restoring a system or a process to a previous state is generally expensive in terms of performance
  - Some states can never be rolled back (e.g., typing in UNIX rm –fr *)

2. Forward Recovery

- When the system detects that it has made an error,
  - forward recovery reverts the system state to error time
  - corrects it, to be able to move forward
- Forward recovery is typically faster than backward recovery
  - but requires that it has to be known in advance which errors may occur: to be able to detect it
- Some systems make use of both forward and backward recovery for different errors or different parts of one error

Recovery

- Error Recovery
- Checkpointing
- Message Logging

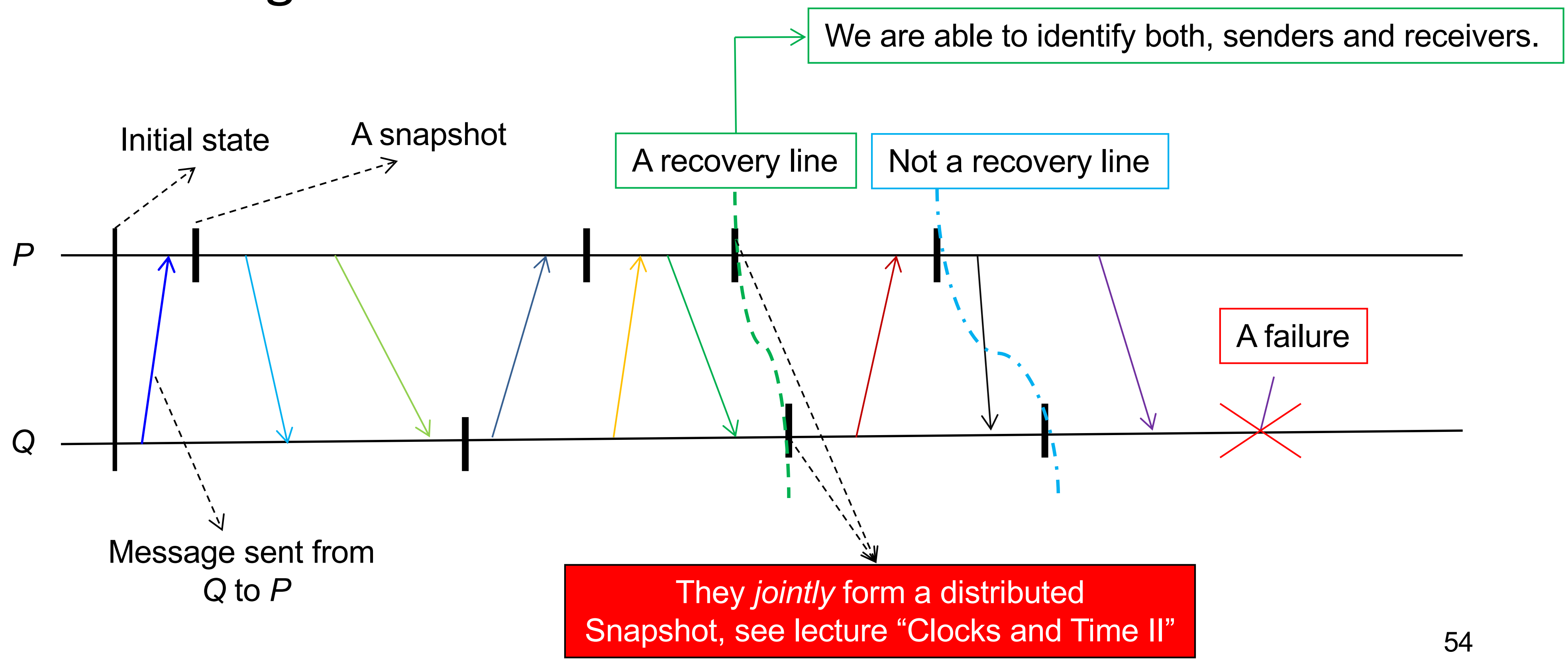
Why Checkpointing?

- In a fault-tolerant distributed system,
  - backward recovery requires that the system regularly saves its state onto a stable storage
- This process is referred to as checkpointing

- Checkpointing
  - storing a distributed snapshot of the current application state (i.e., a consistent global state),
  - and later on, use it for restarting the execution in case of a failure

Recovery Line

- In a distributed snapshot, if a process P has recorded the receipt of a message, then there should be also a process Q that has recorded the sending of that message

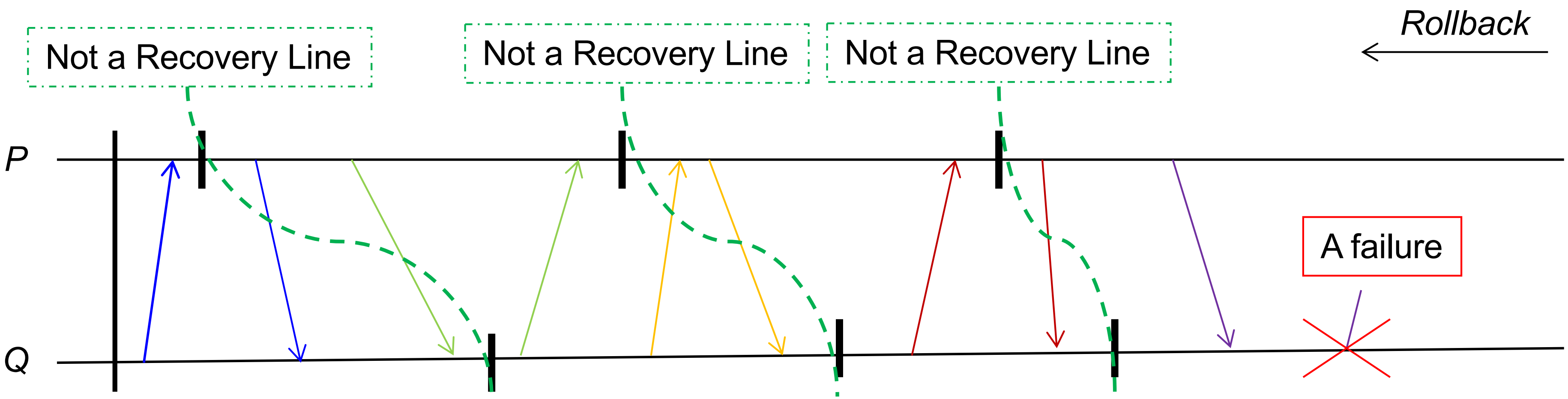


Checkpointing

- Checkpointing can be of two types:
  - Independent Checkpointing: each process simply records its local state from time to time in an uncoordinated fashion
  - Coordinated Checkpointing: all processes synchronize to jointly write their states to local stable storages

Domino Effect

- Independent checkpointing
  - difficult to find a recovery line, leading potentially to a domino effect resulting from cascaded rollbacks



- With coordinated checkpointing, the saved state is automatically globally consistent, hence, domino effect is inherently avoided

Recovery

- Error Recovery
- Checkpointing
- Message Logging

Why Message Logging?

- Checkpoint is expensive
  - Alternatives?

- Message logging
  - Log all messages,
  - replay when needed,
  - ->lead to the same state
- Result
  - Less checkpoint requires

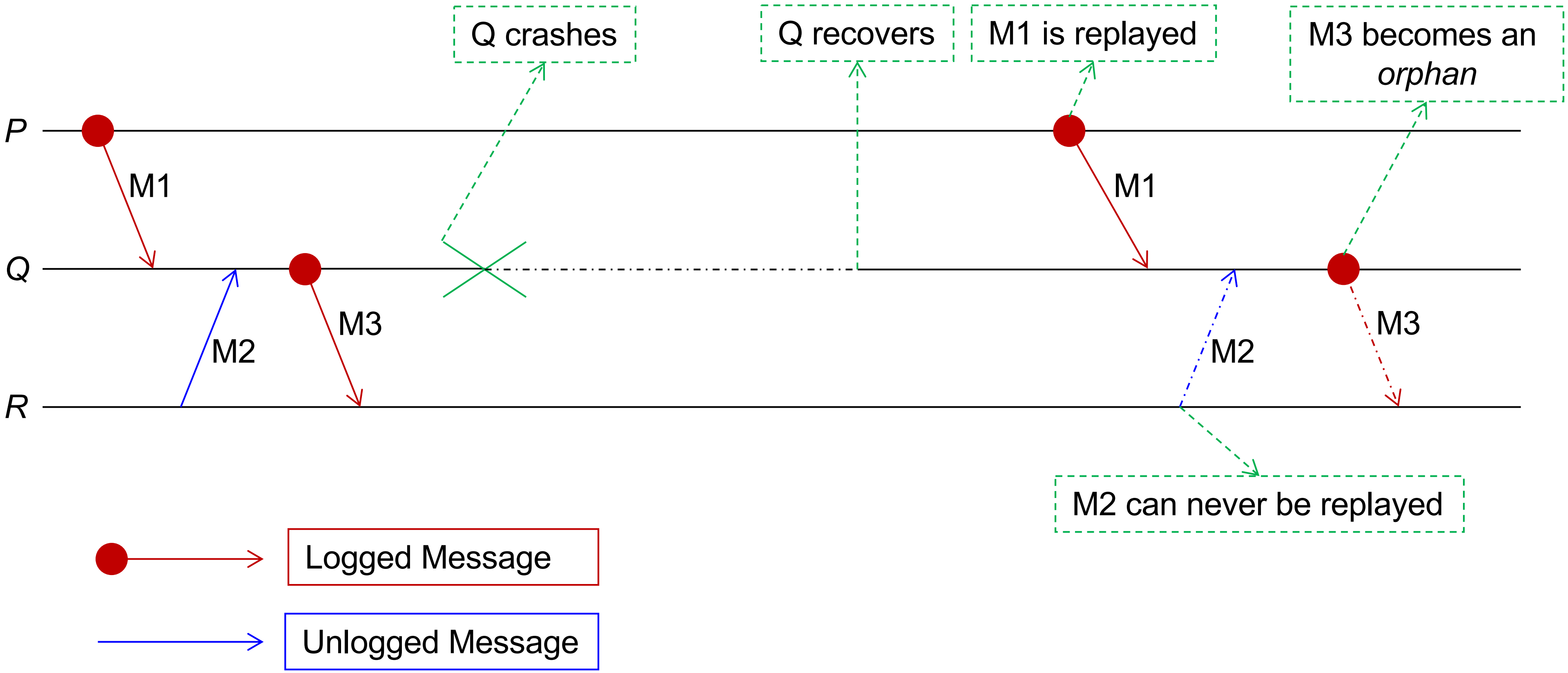
Message Logging

- Message Logging
  - Sender-based logging: A process can log its messages before sending them off
  - Receiver-based logging: A receiving process can first log an incoming message before delivering it to the application

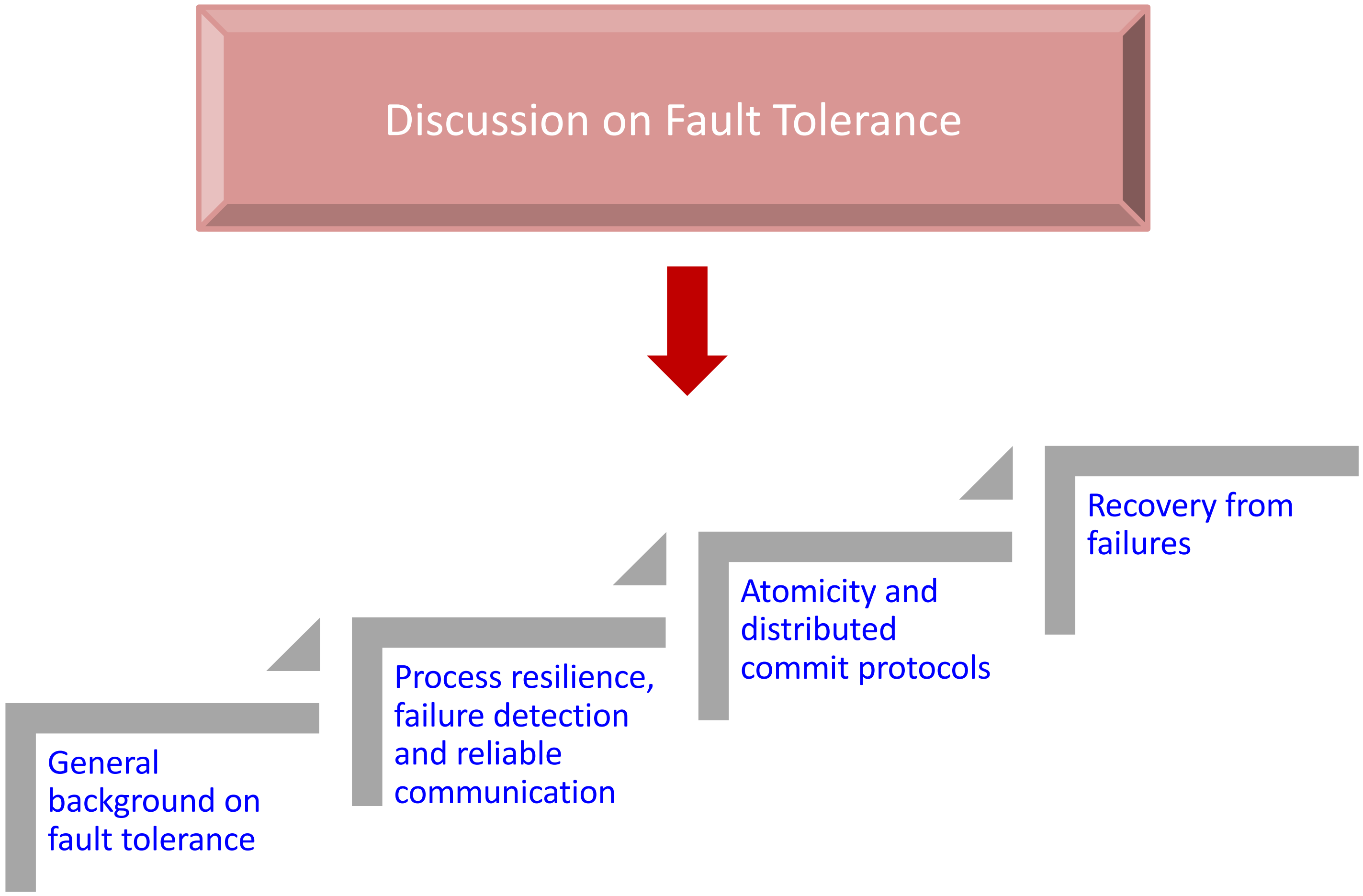
- When a sending or a receiving process crashes,
  1. restore the most recently checkpointed state,
  2. replay the messages logged after the checkpoint

Replay of Messages and Orphan Processes

- Incorrect replay of messages after recovery can lead to *orphan* processes. This should be avoided



Objectives


Discussion on Fault Tolerance
General background on fault tolerance
Process resilience, failure detection and reliable communication
Atomicity and distributed commit protocols
Recovery from failures

Next

- Applications
- Blockchains

- Come back to Fault Tolerance
  - When we discuss Paxos
  - Will one lecture just on this protocol

Questions?

In part, inspired from / based on slides from

- Mohammad Hammoud
- Muyuan Wang
- Philippas Tsigas