Distributed Systems – Fault Tolerance

Dr. Stefan Schulte
Distributed Systems Group
Vienna University of Technology

schulte@infosys.tuwien.ac.at

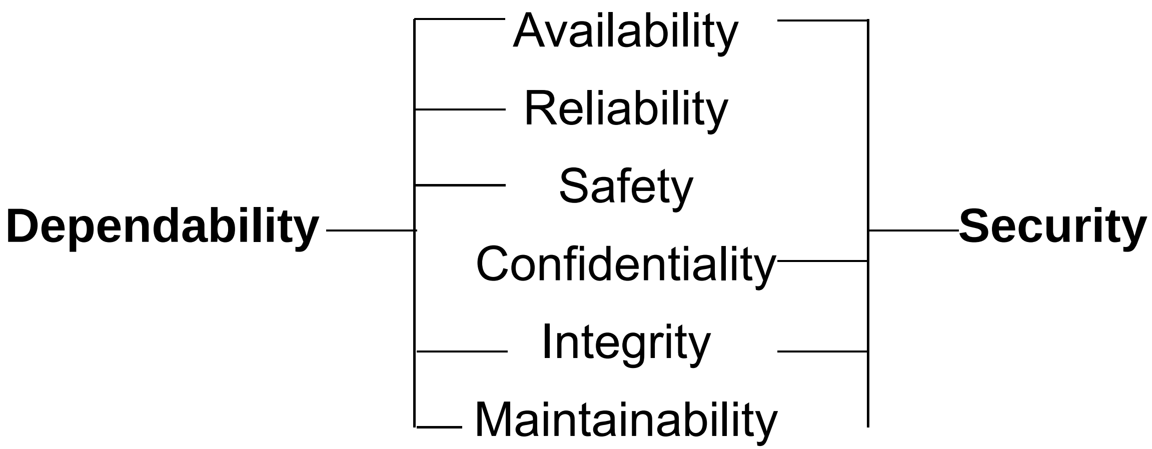
Outline

1. Introduction to Fault Tolerance
2. Process Resilience
3. Reliable Client-Server Communication
4. Recovery

Dependability

- Basics: In Distributed Systems (DS), *components* provide *services* to *clients*
  - To provide services, the component may require services from other components,
  - This means: It *depends* on some other component
  - More specific: The *correctness* of the component in question depends on the *correctness* of another component
- *Dependability* is therefore a core objective in DS

Dependability: Attributes



- Availability: Immediate readiness for correct service
- Reliability: Continuity of correct service
- Safety: Absence of catastrophic consequences
- Integrity: Absence of improper system alterations
- Maintainability: Ability to undergo modifications

Threats to Dependability

- *Failure*: Delivered service deviates from correct service, i.e., the system functionality is not delivered anymore
- *Error:* Deviation of the actual system state from the perceived one
- *Fault:* Cause of an error

Fault → Error → Failure

Example I: Failures, Errors, Faults

- Fault: Software bug in a particular method

  (so far, the fault is *dormant:* As long as nobody calls the method, it will not become *active*)

- Error: The method is called (fault becomes *active*), leading to calculation of wrong value

- Failure: If there is no mechanism to identify the error, it will lead to incorrect service of the component calling the method

Example II: Failures, Errors, Faults

- Fault: Defect USB port of external drive

  (as long as you don‘t make use of the drive: fault is dormant, your computer is still working)

- Error: Input/Output operation started; bit errors occur

- Failure: It is not possible to correctly copy files from/to the external drive

Fault Classes

- Development faults
- Operational faults
- Hardware faults
- Software faults
- Malicious faults
- Accidental faults
- Incompetence faults
- …

Failure Models

- Crash Failure: Component halts, but is working correctly until that moment.
- Omission Failure: Component fails to respond
- Timing Failure: Answer to request is too late
  (Performance Failure)
- Response Failure: Reproducible failures with correct input but wrong output
  (Common-Mode Failure)
- Arbitrary Failure: Arbitrary failures at arbitrary times

  (Byzantine Failures)

Fault Tolerance in DS

- Distributed Systems (DS) can become very complex:
  - The question is not *IF* something will go wrong, the question is *WHEN* this will happen: Faults are inevitable!
- However, a DS should not completely fail if a failure occurs:
  - *Partial Failure*: A failure in one component does not have to lead to a failure in another component or the whole system

What to do about Faults?

- Fault Prevention:
  - Prevent the occurence of a fault
- Fault Forecasting:
  - Estimate present and future faults and their consequences
- **Fault Tolerance:**
  - **Avoid that service failures occur from faults, i.e., *masks* the presence of faults**
  - **Service provision is continued!**
- Fault Removal:
  - Reduce the number and severity of faults

Approaches to Fault Tolerance

- "No Fault Tolerance Without Redundancy" (Gärtner, 1999)
  - Use redundancy to mask a failure, i.e., hide the occurence of a fault

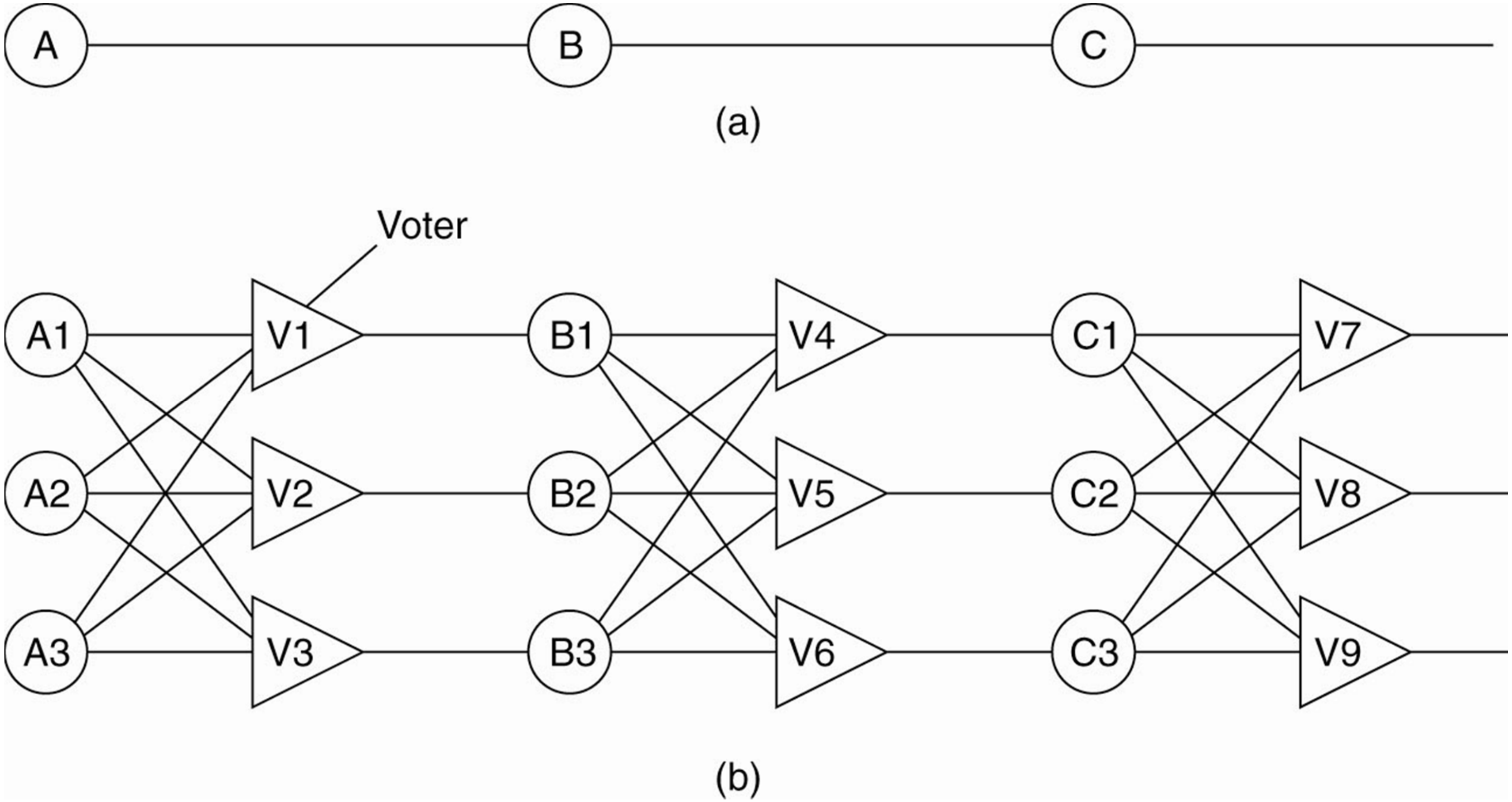
- Failure Masking by Redundancy:
  - Information Redundancy: Add extra information
  - Time Redundancy: Repeat request
  - Physical Redundancy: Add additional components

Redundancy – Examples

- Information Redundancy:
  - Add a parity bit
  - Error Correcting Codes (memory)
  - Hard disks in a RAID 4+5
- Time Redundancy:
  - Retransmissions in TCP/IP
  - Call method again
- Physical Redundancy:
  - Backup server
  - Hard disks in a RAID 1
  - But also: Different implementations of same functionality

Physical Redundancy – Example

Electronic circuit with Triple Modular Redundancy:

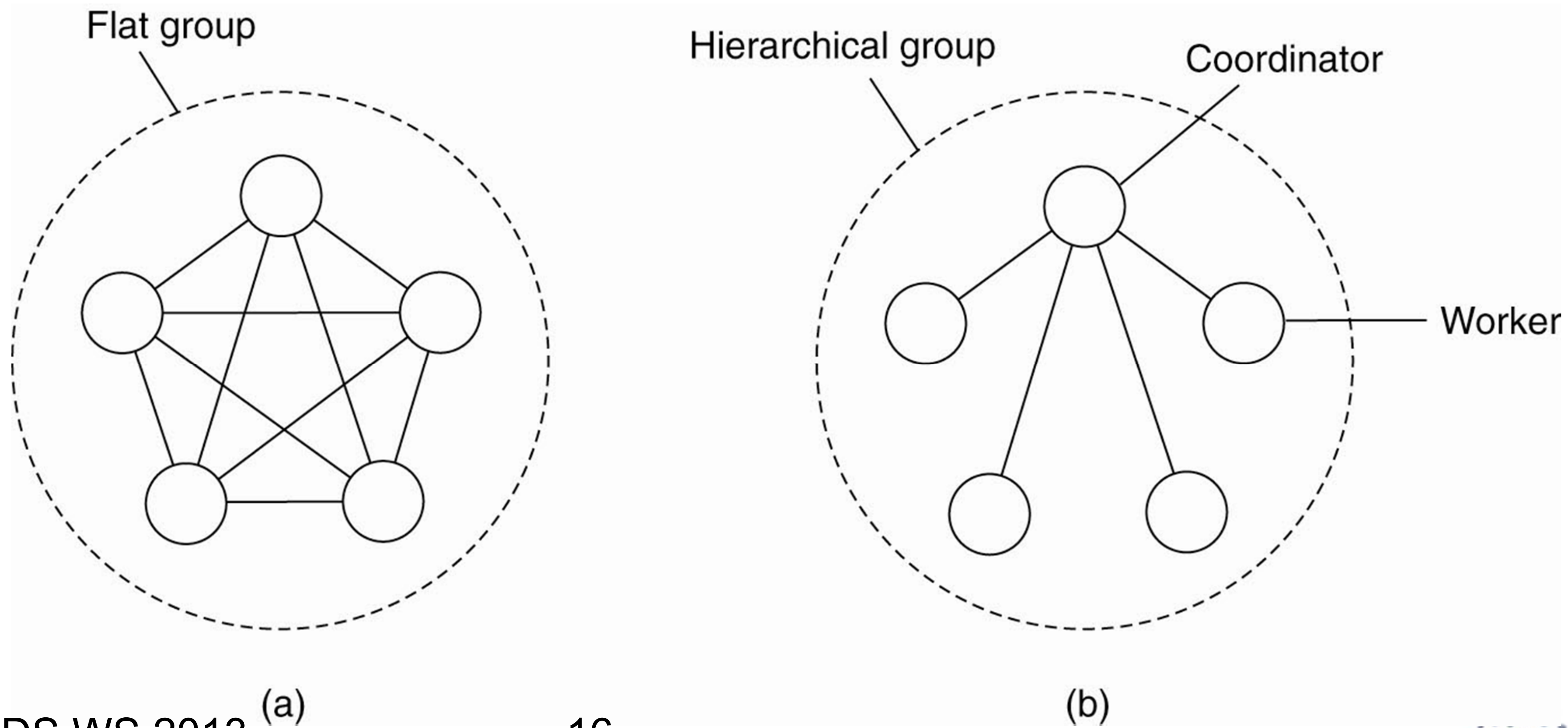


Outline

1. Introduction to Fault Tolerance
2. Process Resilience
3. Reliable Client-Server Communication
4. Recovery

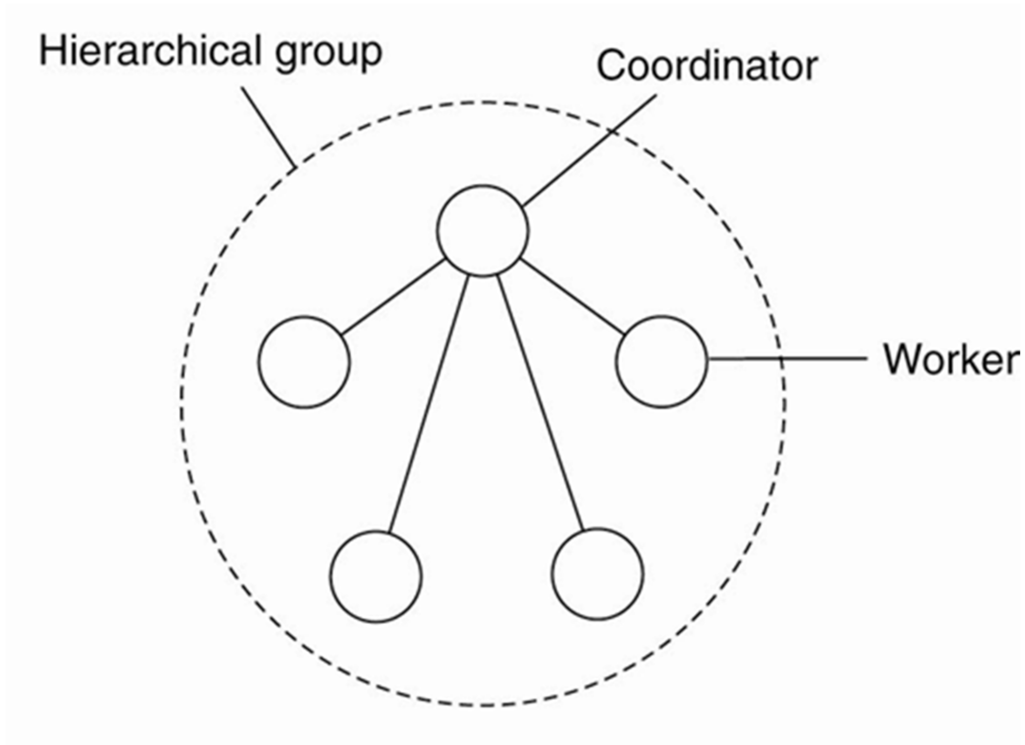
Basics

- How to tolerate faulty processes?
  - "No Fault Tolerance Without Redundancy"
    - ➢ Organize several identical processes into a group



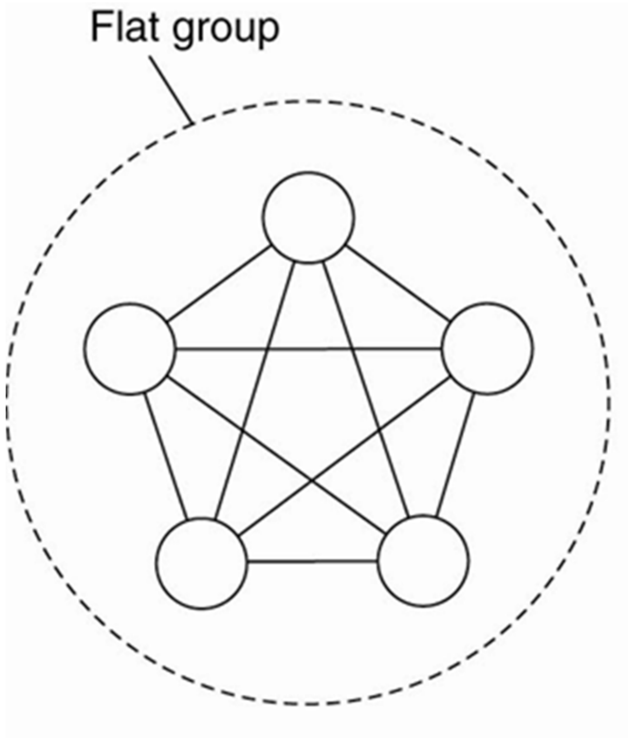
Communication in Hierarchical Groups

- Hierarchical Groups:
  - Communication through a single coordinator
  - Not really fault tolerant or scalable
  - However, easier to implement



Communication in Flat Groups

- Flat Groups:
  - Good for fault tolerance as information exchange immediately occurs with all group members
  - May impose overhead as control is completely distributed, and voting needs to be carried out
  - Harder to implement

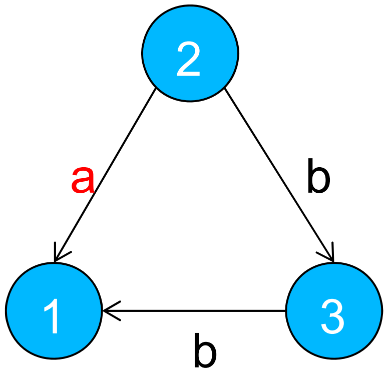


Groups and Failure Masking

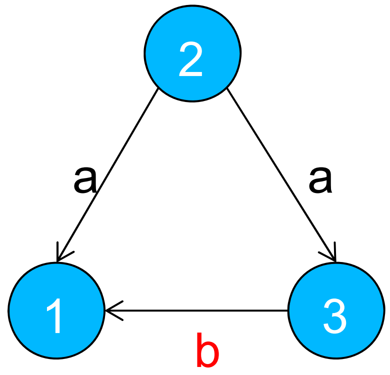
- k-fault tolerant group:
  - Group is able to mask any *k* concurrent member failures
- How large does a k-fault tolerant group need to be?
  - Crash/performance failure models (i.e., components don‘t answer anymore): k+1 are necessary
  - Arbitrary/Byzantine failure model: 2k+1 components are necessary
- Assumptions: All members are identical and process all input in the same order

Groups and Failure Masking II

- Scenario (distributed computation):
  - At least one group member different from the others
  - Non-faulty members should have to reach agreement on the same value



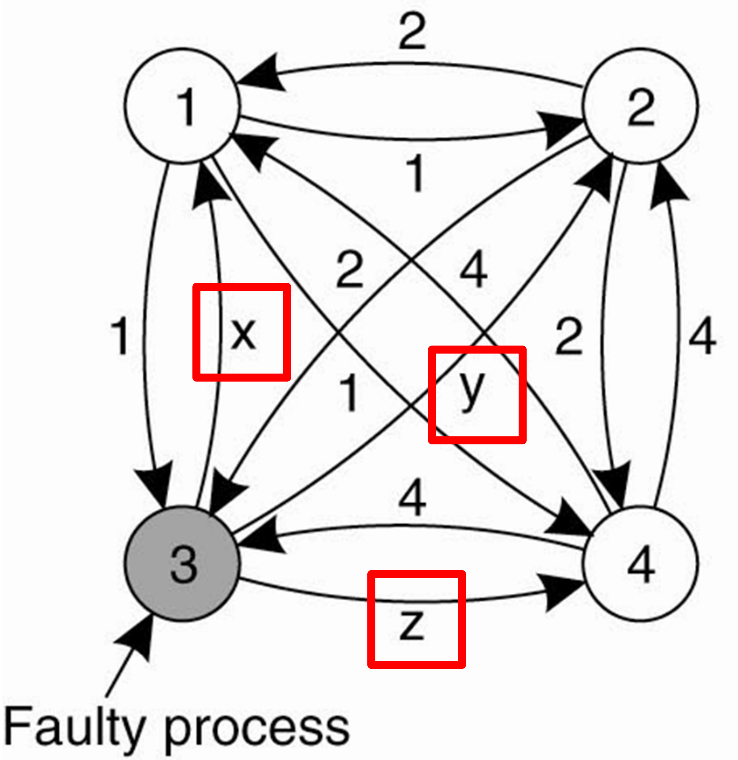
Process 2 tells different things



Process 3 passes a different value

Byzantine Agreement Problem I

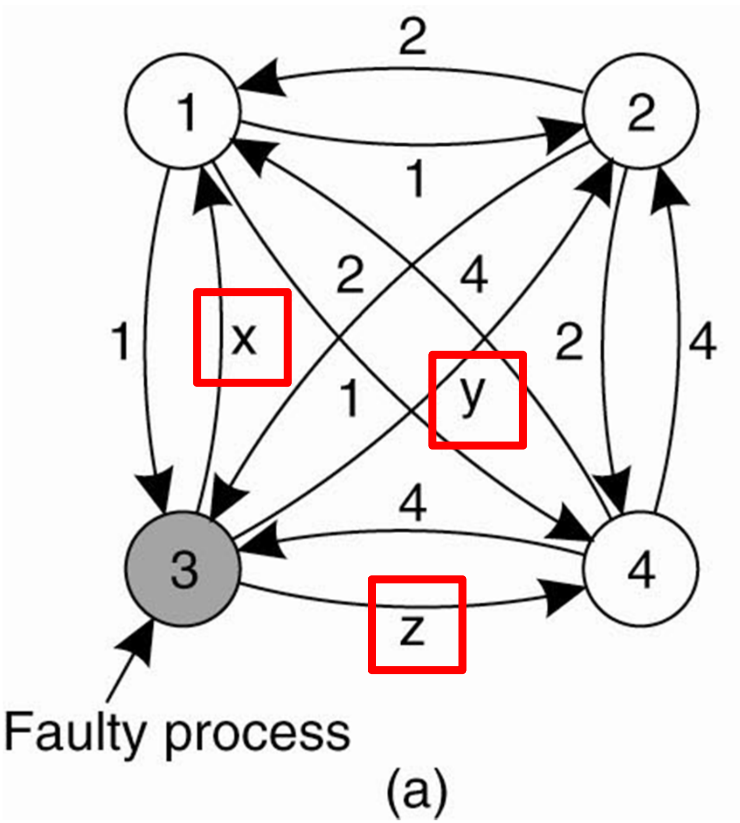
- Byzantine Agreement Problem:



(a)

- Assumptions:
  - Unicast messages
  - Ordered message delivery
  - Synchronous processes
  - Bounded communication delay

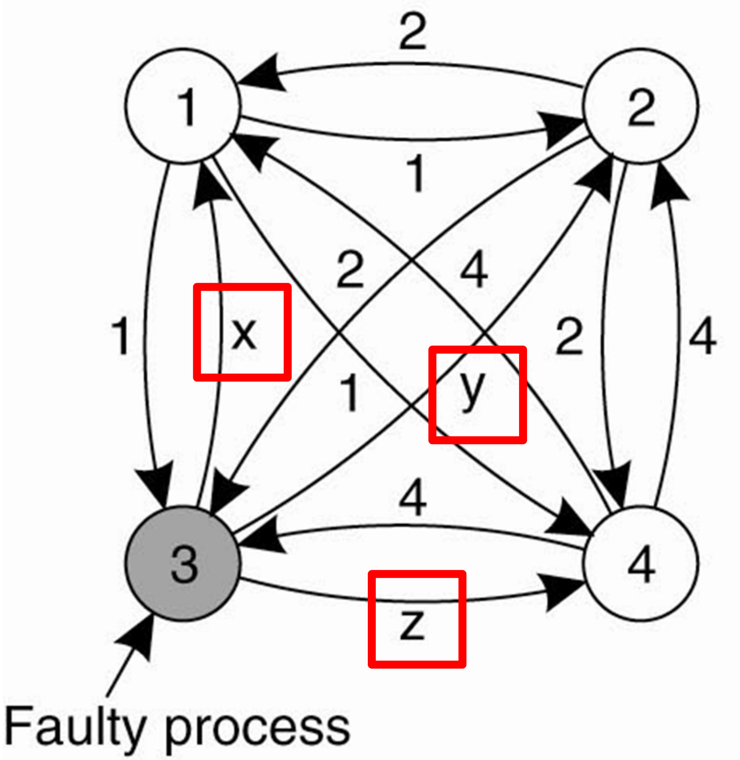
Byzantine Agreement Problem II



- *N* processes
- Each process sends value v_i to the others
- Each process builds a vector *V* from the values
- If process *i* is non-faulty, $V[i]=v_i$

Byzantine Agreement Problem III

- Step 1: Messages are sent



(a)

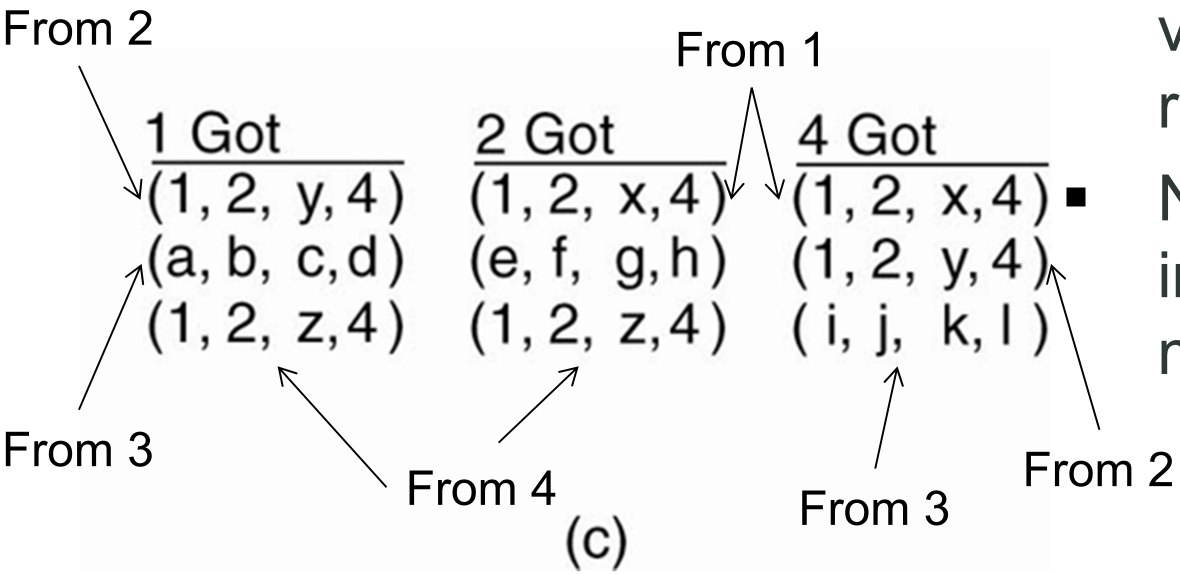
- Step 2: Results - Individual V

1 Got(1, 2, x, 4)
2 Got(1, 2, y, 4)
3 Got(1, 2, 3, 4)
4 Got(1, 2, z, 4)

(b)

Byzantine Agreement Problem IV

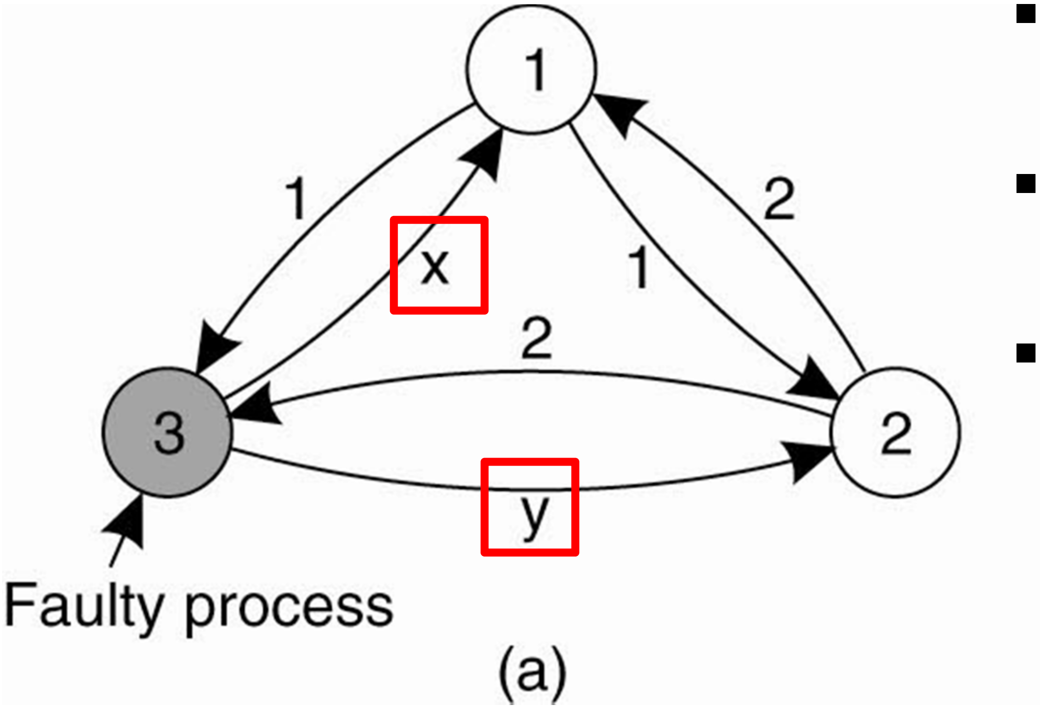
- Step 3:
  - Every process passes its vector V
  - Process 3 „lies“ to everyone



- Step 4:
  - Each process examines *i*th element of *received* vectors
  - If there is a majority, value is put into resulting vector
  - No majority: element in result vector is marked *UNKNOWN*

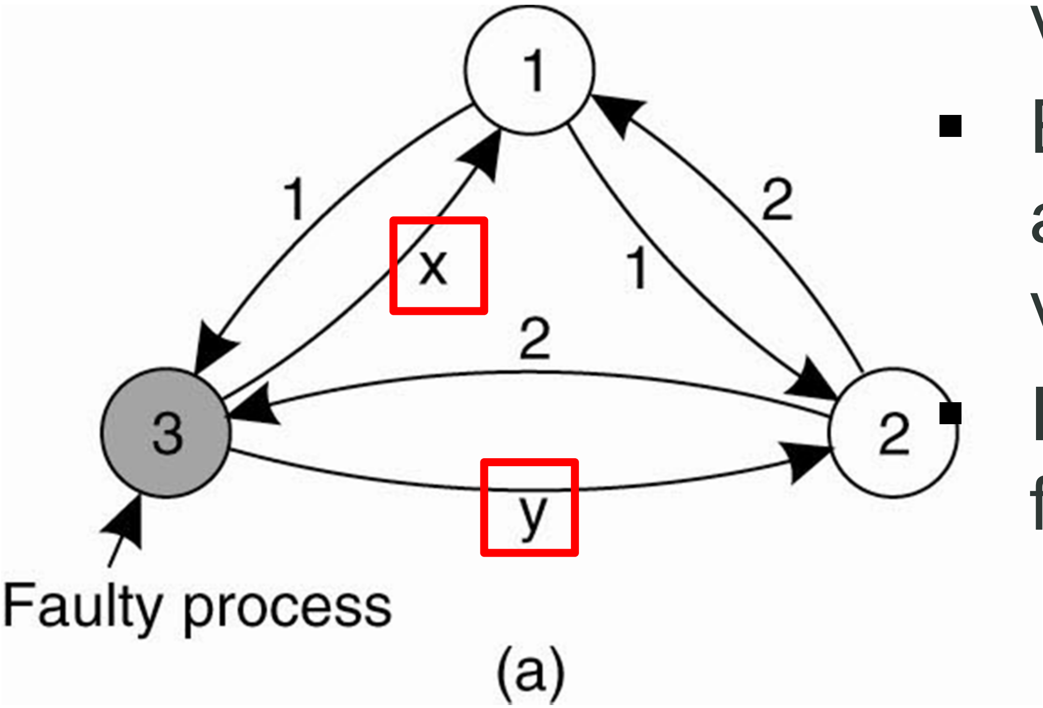
Byzantine Agreement Problem V

- Byzantine Agreement Problem:



- Assumptions:
  - Unicast messages
  - Ordered message delivery
  - Synchronous processes
  - Bounded communication delay

Byzantine Agreement Problem VI

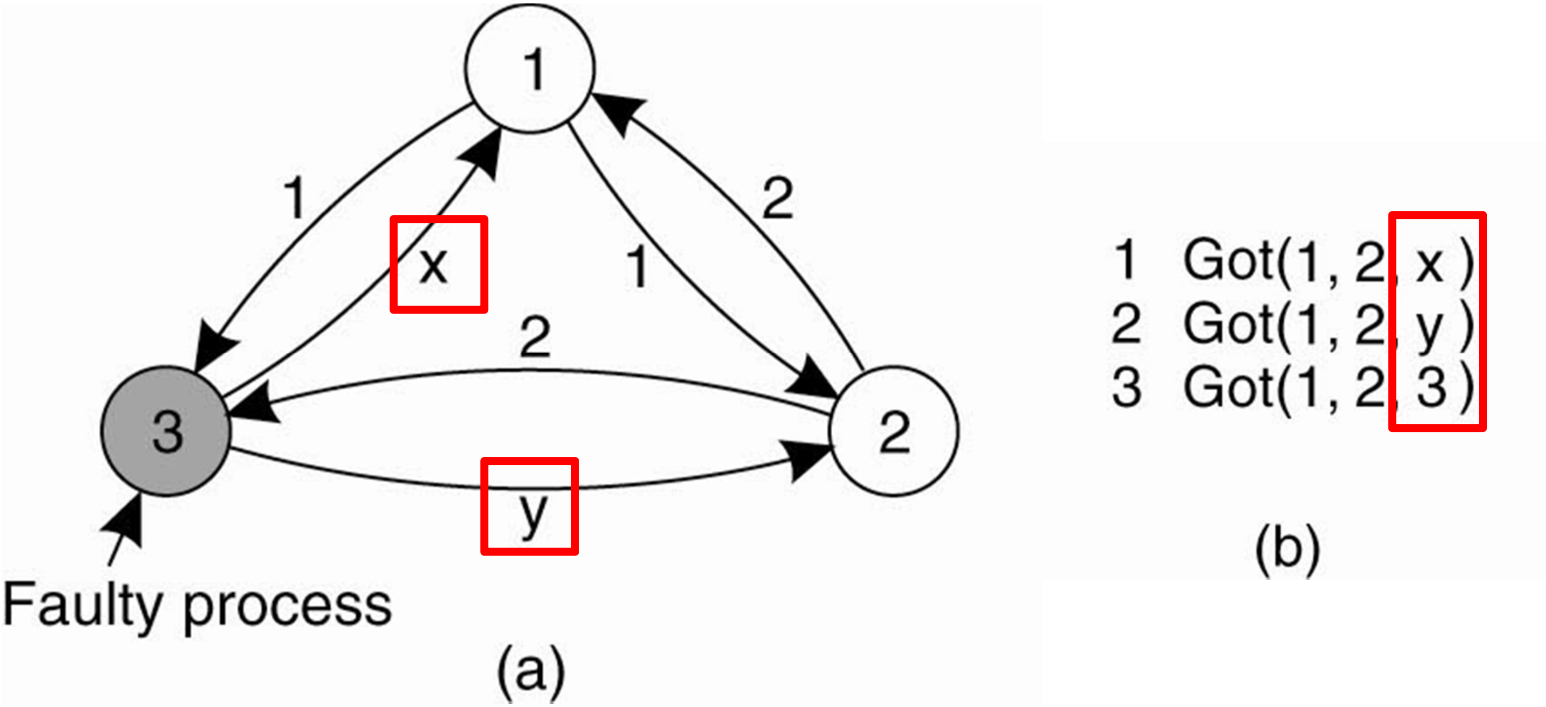


- *N* processes
- Each process sends value v_i to the others
- Each process builds a vector *V* from the values
- If process *i* is non-faulty, $V[i]=v_i$

Distributed Systems Group

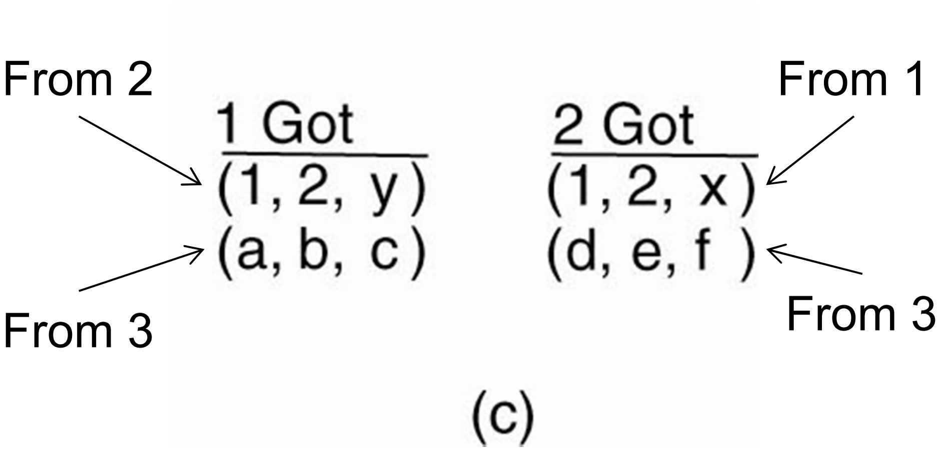
Byzantine Agreement Problem VII

- Step 1: Messages are sent
- Step 2: Results - Individual V



Byzantine Agreement Problem VIII

- Step 3:
  - Every process passes its vector V
  - Process 3 „lies“ to everyone



- Step 4:
  - Each process examines *i*th element of *received* vectors
  - If there is a majority, value is put into resulting vector
  - No majority: element in result vector is marked *UNKNOWN*

Byzantine Agreement Problem IX

- 4 Processes:

1 Got	2 Got	4 Got
(1, 2, y, 4)	(1, 2, x, 4)	(1, 2, x, 4)
(a, b, c, d)	(e, f, g, h)	(1, 2, y, 4)
(1, 2, z, 4)	(1, 2, z, 4)	(i, j, k, l)

- Agreement for v_1, v_2, v_4

- 3 Processes:

1 Got	2 Got
(1, 2, y)	(1, 2, x)
(a, b, c)	(d, e, f)

- No agreement possible!

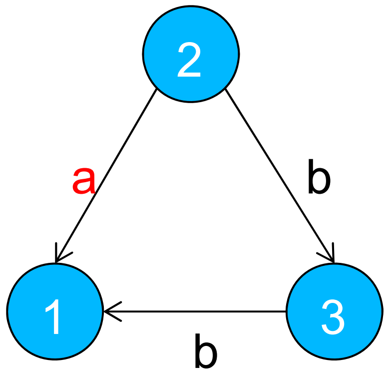
➢ 2k+1 non-faulty processes are necessary for k-fault tolerance

Outline

1. Introduction to Fault Tolerance
2. Process Resilience
3. Reliable Client-Server Communication
4. Recovery

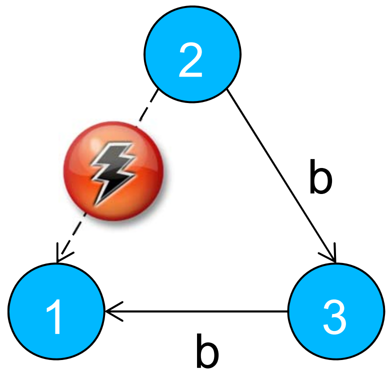
Reliable Client-Server Communication

- So far: Process Resilience



Process 2 tells different things

- But what about reliable communication channels?



Connection between Process 2 and Process 1 fails

Remote Procedure Calls: What can go wrong?

1. Client cannot locate server
2. Client request is lost
3. Server crashes
4. Server response is lost
5. Client crashes (after request has been sent)

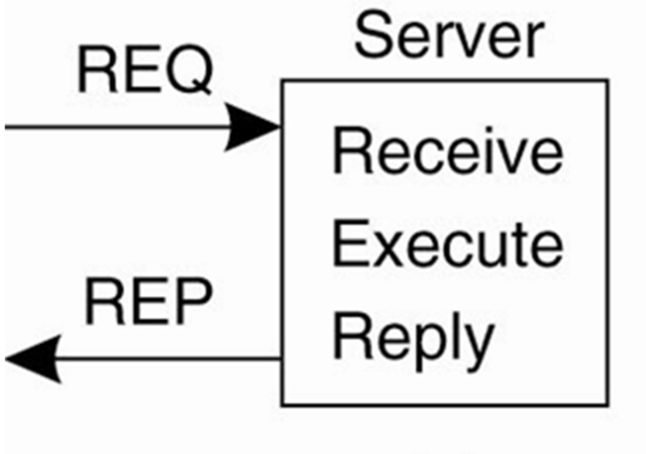
Remote Procedure Calls: Solutions I

1. Client cannot locate server
   → Just report back to client
   → Client has to take care of it (e.g., exception handling)

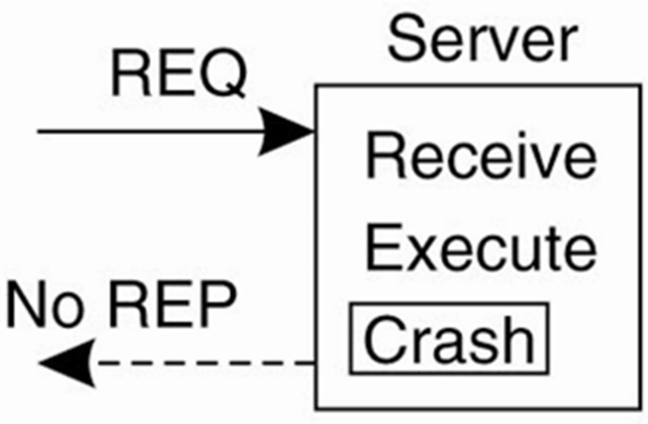
2. Client request is lost
   → Resend request message
   → Server won't know difference between original and retransmission

Remote Procedure Calls: Solutions II

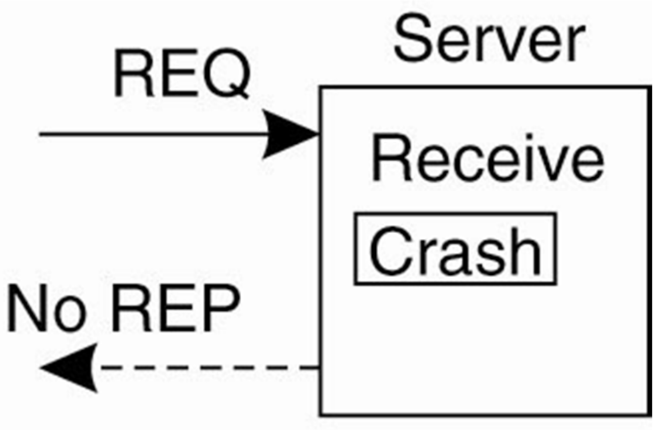
3. Server crashes
   a) Normal case
   b) Crash *after* execution
   c) Crash *before* execution



(a)



(b)

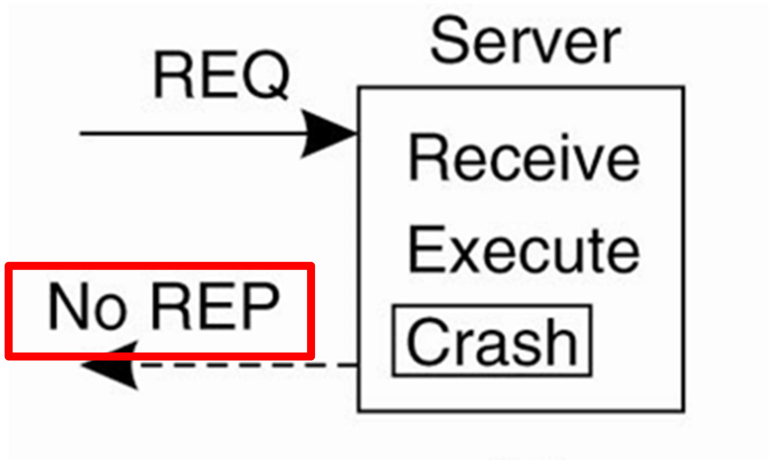


(c)

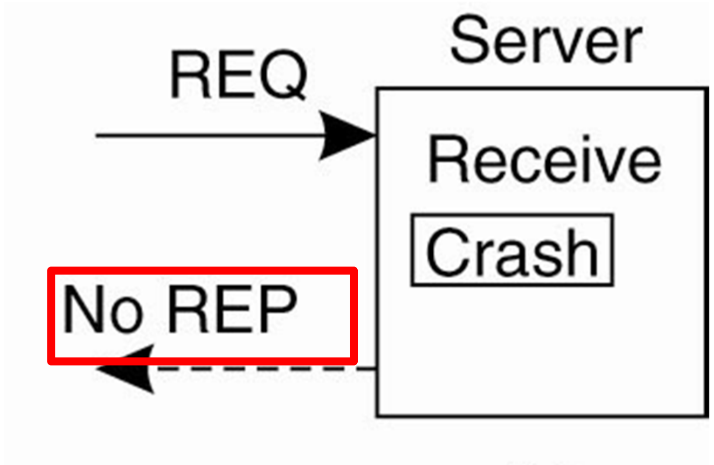
Remote Procedure Calls: Solutions II

3. Server crashes
   a) Normal case – no crash
   b) Crash *after* execution
   c) Crash *before* execution

The client is not able to see the difference



(b)



(c)

Remote Procedure Calls: Solutions III

3. Server crashes
   - Correct behaviour of Client depends on behaviour of Server
     1. At-*least*-once-semantics: The Server guarantees it will carry out an operation at *least* once, no matter what
     2. At-*most*-once-semantics: The Server guarantees it will carry out an operation at *most* once.
   - And the Client? (if not receiving a reply, but a message that the server has rebooted)
     1. *Always* reissues a request
     2. *Never* reissues a request
     3. Reissue a request only if it did not receive an ACK (that request has been delivered)
     4. Reissue a request only if it did receive an ACK

Remote Procedure Calls: Solutions IV

3. Server crashes
   - 8 possible combinations of strategies
   - Example: Client sends printing request to Print Server
     - Three events may happen at the Server:
       (M) Send the completion message (ACK)
       (P) Print the text
       (C) Crash
   - There is no combination of server and client strategies that will work correctly under all possible event sequences.

Remote Procedure Calls: Solutions V

3. Server crashes

- These events can occur in six different sequences:
  1. M →P →C: A crash occurs after sending the completion message and printing the text.
  2. M →C (→P): A crash happens after sending the completion message, but before the text could be printed.
  3. P →M →C: A crash occurs after sending the completion message and printing the text.
  4. P→C(→M): The text printed, after which a crash occurs before the completion message could be sent.
  5. C (→P →M): A crash happens before the server could do anything.
  6. C (→M →P): A crash happens before the server could do anything.

Remote Procedure Calls: Solutions VI

3. Server crashes

Client	Server					
	Strategy M → P			Strategy P → M		
Reissue strategy	MPC	MC(P)	C(MP)	PMC	PC(M)	C(PM)
Always	DUP	OK	OK	DUP	DUP	OK
Never	OK	ZERO	ZERO	OK	OK	ZERO
Only when ACKed	DUP	OK	ZERO	DUP	OK	ZERO
Only when not ACKed	OK	ZERO	OK	OK	DUP	OK

OK = Text is printed once
DUP = Text is printed twice
ZERO = Text is not printed at all

Example: Client wrongly assumes Print hasn‘t been carried out

M = Send the completion message, P = Print, C = Crash

Remote Procedure Calls: Solutions VII

4. Server response is lost
   - How do we know that the server has not crashed?
   - Once again: Has the server carried out the operation?
   - Repeat request:
     → In case of real-world impact? Transfer from your banking account carried out twice?
   - *No* real solution! Except making operations idempotent, i.e., repeatable without any harm

Remote Procedure Calls: Solutions VIII

5. Client crashes (after request has been sent)
   - Server executes requests anyway and sends response (called orphan computation)
   - Different Solutions:
     1. Orphan is killed by Client if it is received
     2. Reincarnation: Client tells Servers that it has rebooted; Server kills orphans
     3. Expiration: Require computations to complete in *T* time units. Old ones are simply removed.

Outline

1. Introduction to Fault Tolerance
2. Process Resilience
3. Reliable Client-Server Communication
4. Recovery

Recovery

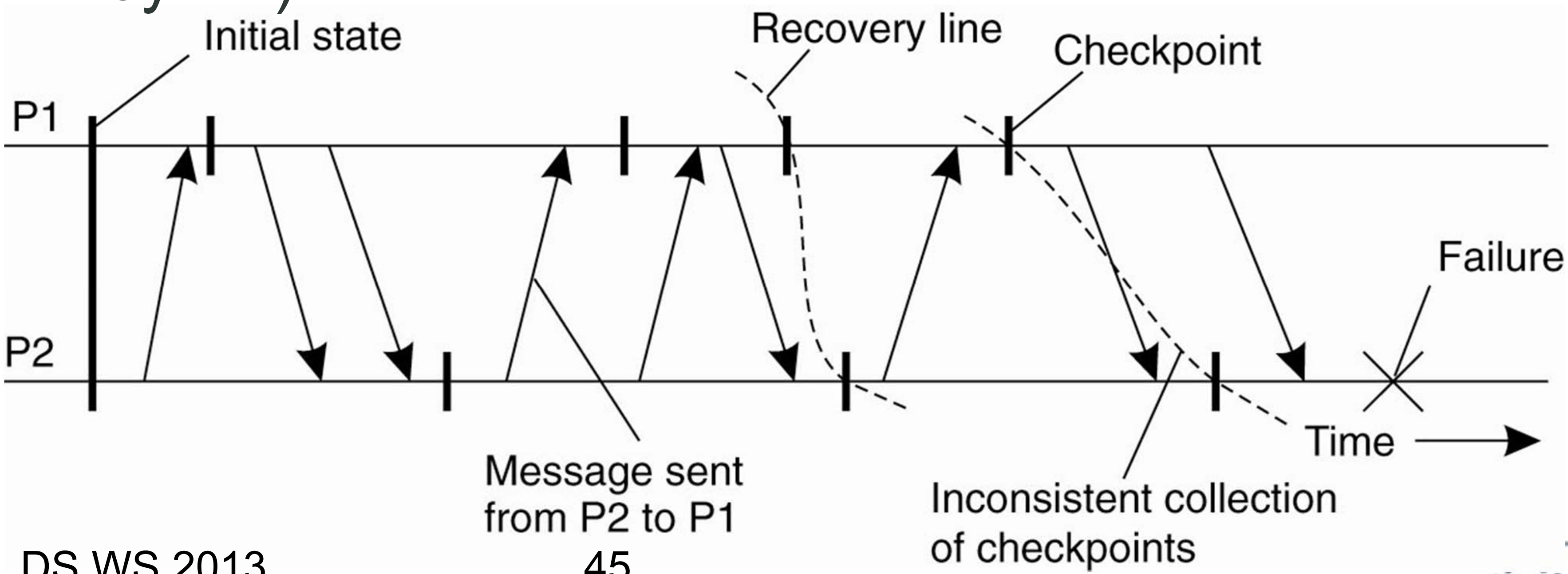
- So far: Tolerate faults
- But what if a failure occurs nevertheless?
  - Recovery is complicated as processes need to cooperate to identify a *consistent state* from where to recover
- Bring the system into an error-free state:
  - Forward error recovery: Find a new state from which the system can continue operation
  - Backward error recovery: Bring the system back into a previous error-free state
    - Usually applied

Backward Recovery

- Bring system from its present errornous state to a previously correct state:
  - Makes it necessary to record the system‘s state from time to time, i.e., *checkpointing*
- Benefit: Generally applicable method
- Drawbacks:
  - Relatively costly
  - No guarantee that the same failure won‘t happen again
  - Some things are simply irreversible

Checkpointing

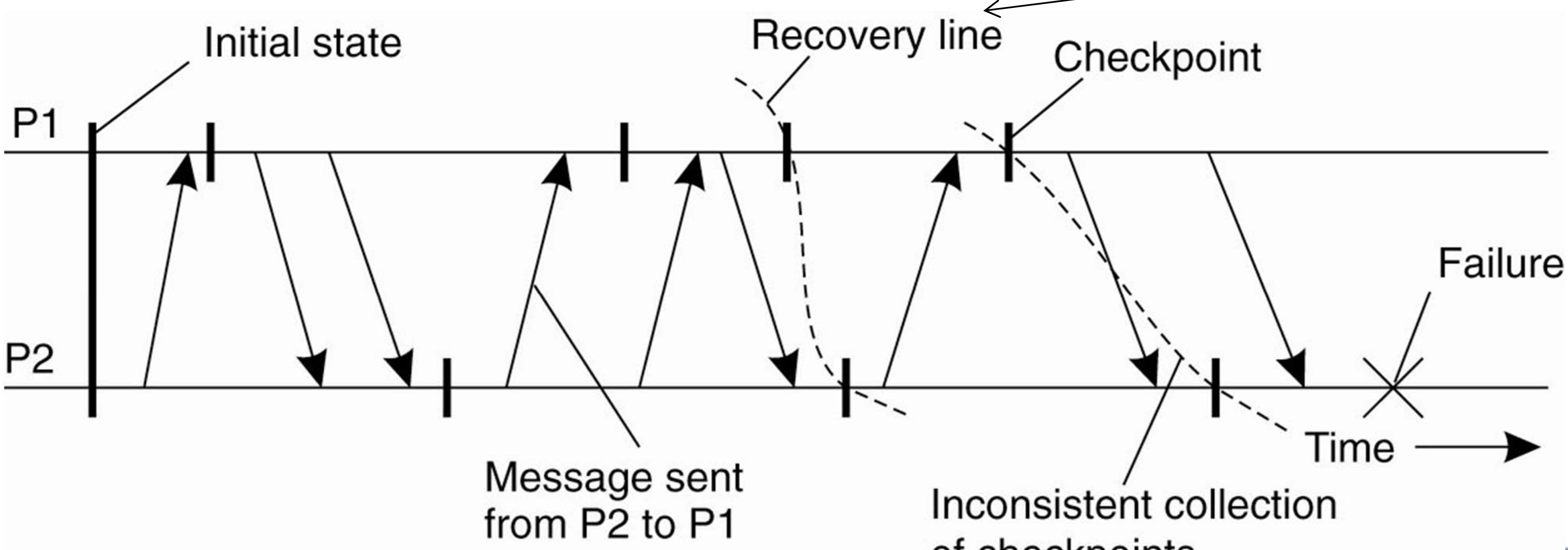
- Goal: Record a consistent global state (also known as *distributed snapshot*)
- Every message that has been received (here: by P2) is also shown to have been sent (here: by P1)



Independent Checkpointing

- Distributed nature of checkpointing (each process records local state from time to time) makes it difficult to find a *recovery line*

Distributed Snapshot
(Most recent consistent collection of snapshots)



Coordinate Checkpointing

- As the name implies: Each process takes a checkpoint after a globally coordinated action
  - Coordinator necessary!
- Two-phase blocking protocol:
  1. Coordinator multicasts a *checkpoint request* message
  2. When participant receives this message, it takes a checkpoint, stops sending (application) messages, and reports back that it has taken a checkpoint
  3. When all checkpoints have been confirmed at the coordinator, the latter broadcasts a *checkpoint done*
  4. Processes continue

Message Logging

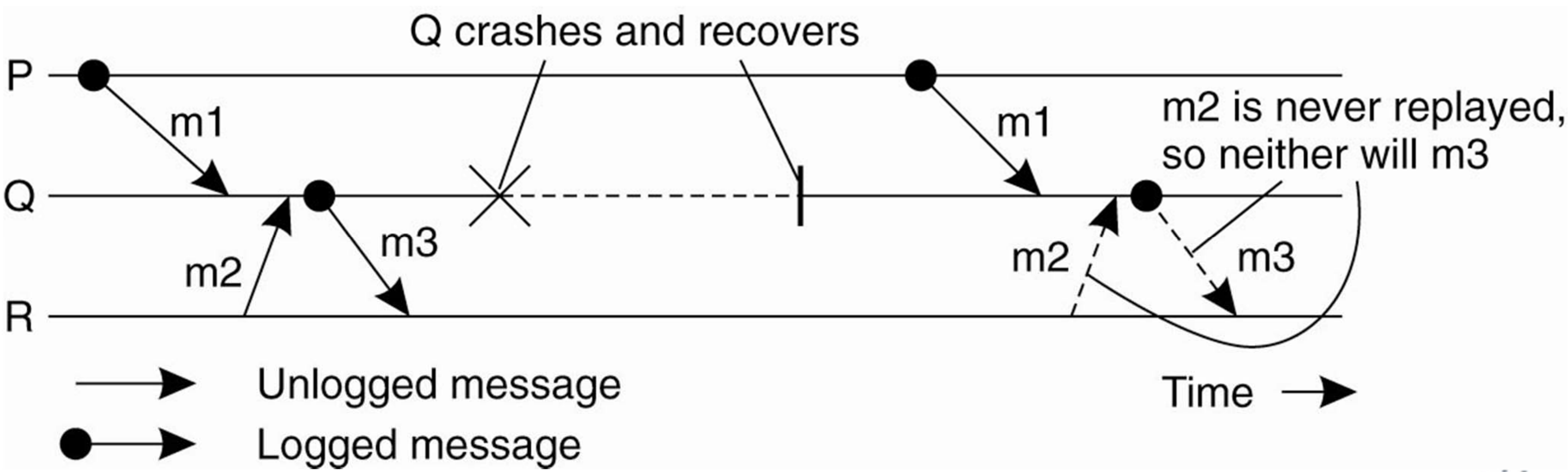
- Alternative to checkpointing
  - Less costly than checkpointing
  - Nevertheless needs some checkpoints
- Instead of taking a checkpoint, try to *replay* communication behaviour from the most recent checkpoint

Message Logging – Basic Assumption

- Piecewise deterministic execution model:
  - The execution of each process can be considered as a sequence of state intervals
  - Each state interval starts with a nondeterministic event (e.g., message receipt)
  - Execution in a state interval is completely deterministic
- If we record nondeterministic events (to replay them later), we obtain a deterministic execution model that will allow us to do a complete replay.

Message Logging – Avoid Orphans

- Example:
  - Process Q has just received and subsequently delivered messages m_1 and m_2.
  - Assume that m_2 is never logged.
  - After delivering m_1 and m_2, Q sends message m_3 to process R
  - Process R receives and subsequently delivers m_3



Further Readings

- Tanenbaum, van Steen: Distributed Systems – Principles and Paradigms, 2nd edition, 2007.
- Jalote: Fault Tolerance in Distributed Systems, 1998.
- Avizienis, Laprie, Randell, Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing, IEEE Transactions on Dependable and Secure Computing, 1(1), 2004.
- Gärtner: Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments, ACM Computing Surveys, 31(1), 1999.