Fault Tolerance
Part I

CS403/534

Distributed Systems

Erkay Savas
Sabanci University

Overview

- Basic concepts
- Process resilience
- Reliable client-server communication
- Reliable group communication
- Distributed commit
- Recovery from failure

Basic Concepts

- Concept of partial failure
- A distributed system consists of components (processes, communication channels, etc)
- Partial failure takes place when a component fails
  - Goal is to continue in the presence of partial failures.
  - and to automatically recover from partial failures without seriously affecting the overall performance
- Fault tolerance is closely related to dependability:
  - A dependable system must tolerate partial failures

Dependability

Dependability includes the following requirements:

1. Availability
   - *The system is ready to be used immediately*
2. Reliability
   - *A property that a system can run continuously without failure*
3. Safety
   - *When a system temporarily fails to operate correctly, nothing catastrophic happens*
4. Maintainability
   - *How easy a failed system can be repaired.*
5. Security

Terminology

- A system **fails** when it is not living up to its specifications
- An **error** is a a part of system's state that may lead to a failure
- The cause of an error is called a **fault**.
- Dependable system is where you can manage faults
  - Fault prevention: prevent the occurrence of a fault
  - Fault tolerance: build a system that can provide its services in the presence of faults.
  - Fault removal: reduce the presence, number, and seriousness of faults.
  - Fault forecasting: estimate the present number, future incidence, and the consequences of faults

Fault Types

- Transient faults
  - Occur once and then disappear.
  - If the operation is repeated, the fault goes away
- Intermittent faults
  - Occur, then vanishes of its own accord, then reappears, and so on
  - Difficult to diagnose
- Permanent faults
  - continues to exist until the faulty component is repaired

Failure Models

- Different types of failures.

Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure *Receive omission* *Send omission*	A server fails to respond to incoming requests A server fails to receive incoming messages A server fails to send messages
Timing failure	A server's response lies outside the specified time interval
Response failure *Value failure* *State transition failure*	The server's response is incorrect The value of the response is wrong The server deviates from the correct flow of control
Arbitrary failure **(Byzantine failure)**	A server may produce arbitrary responses at arbitrary times

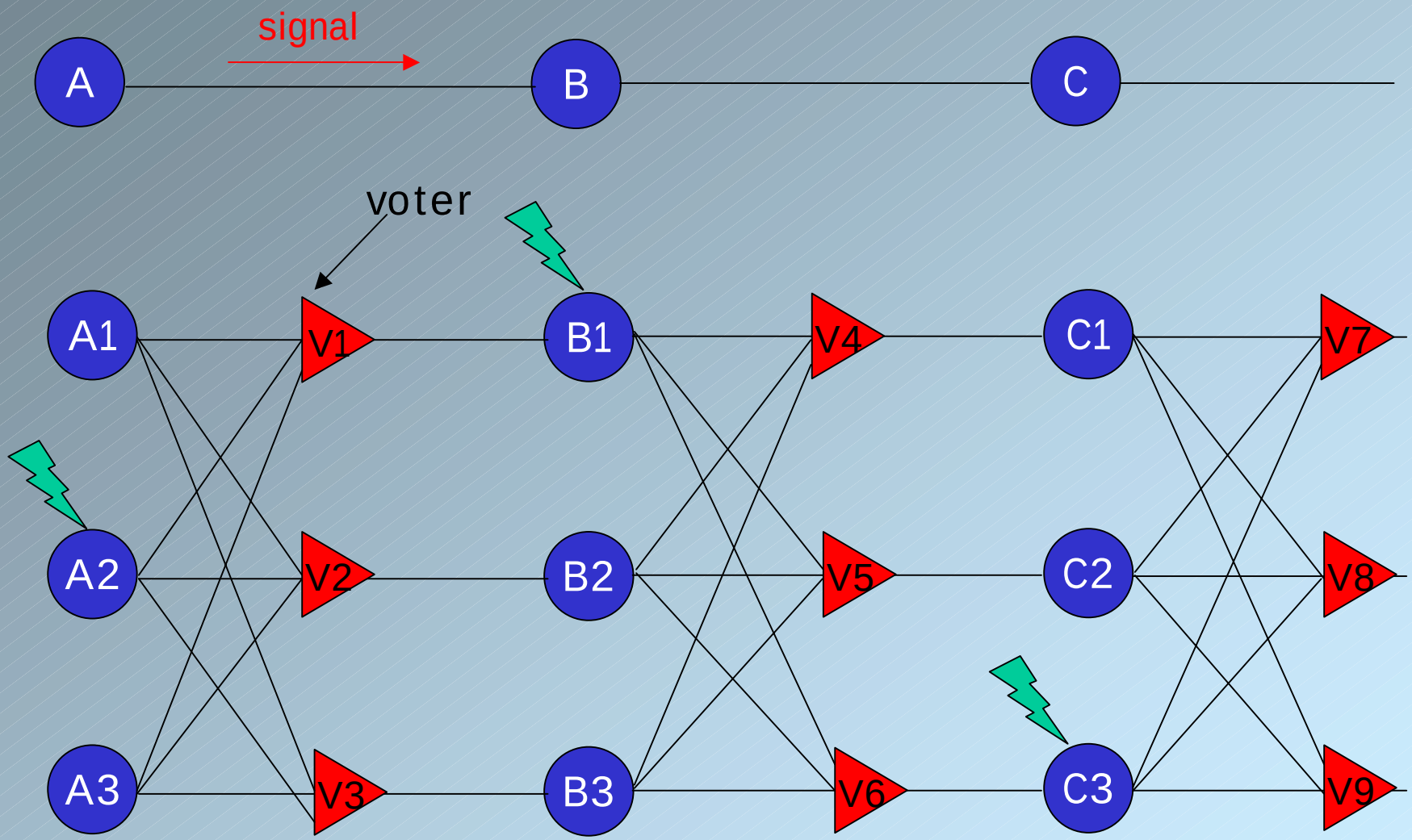
Crash Failures

- Problem:
  - Clients cannot distinguish between a crashed and slow server
  - Servers don't announce they are going to crash
- Fail-silent:
  - The server exhibits omission or crash failures; the client cannot tell what went wrong
- Fail-stop:
  - It is easy to detect that a server crashed; (e.g. it announces)
- Fail-safe:
  - The server exhibits arbitrary, but benign failures (they can't do any harm; clients can easily understand they are junk).

Failure Masking by Redundancy

- Information redundancy:
  - Extra bits are added to allow recovery from corrupted bits during transmission
- Time redundancy:
  - An action is repeated if needed.
  - Retransmitting
  - Helpful when the faults are transient or intermittent
- Physical redundancy
  - Extra equipment or processes are added to tolerate the loss or malfunctioning of some components

Triple Modular Redundancy.

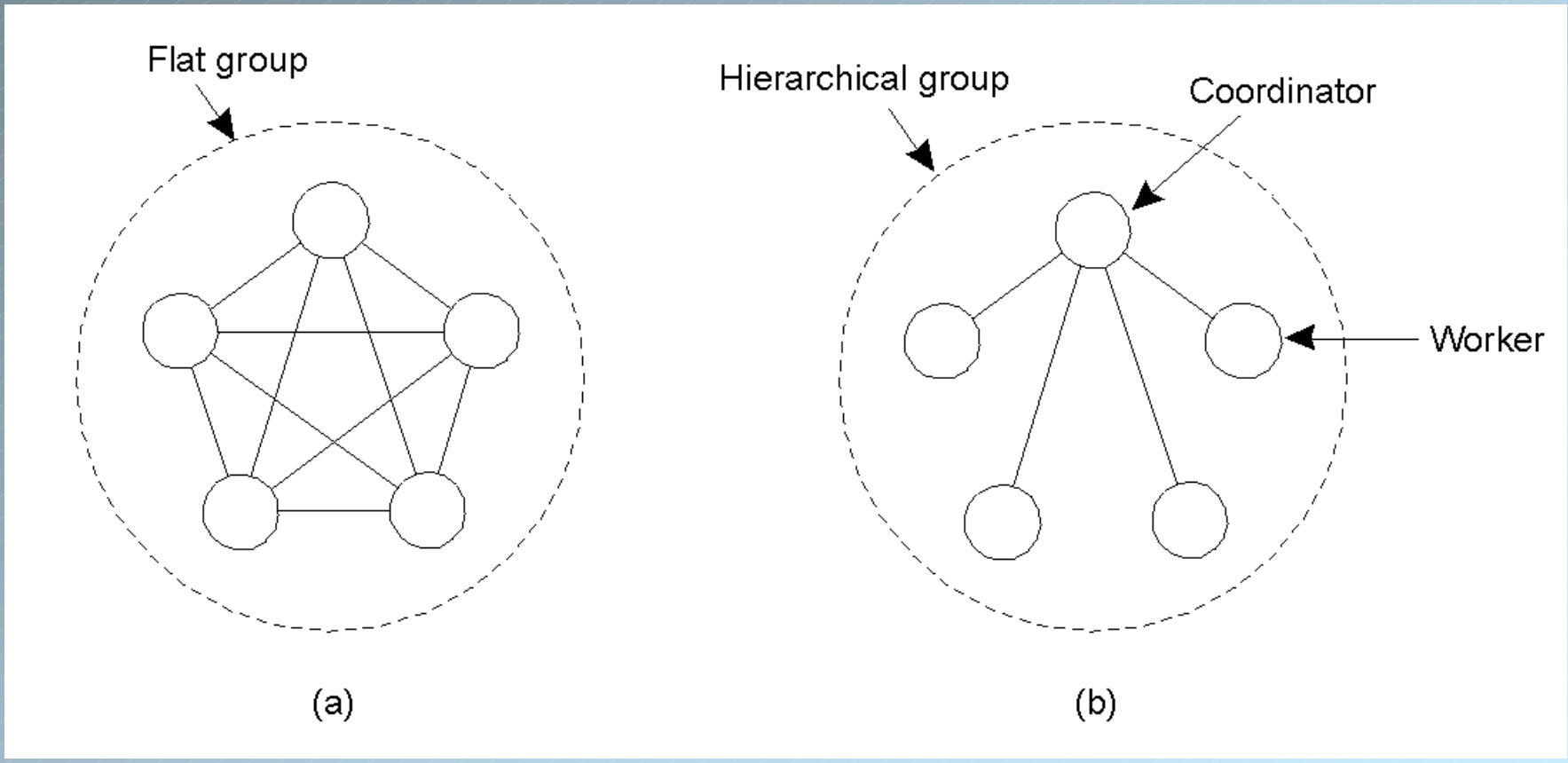


Process Resilience

- Basic approach
  - Organize several identical processes into a group; if a process in the group fails the others take over.
  - Group of processes act as if they were a single entity
- Flat groups:
  - All processes are equal; all decisions are made collectively
  - No single point of failure; hard to implement
  - May impose overhead as control is completely distributed
- Hierarchical groups:
  - All communications are done through a single coordinator; easy to implement
  - Loss of the coordinator brings the entire group to a halt

Flat Groups versus Hierarchical Groups

a) Communication in a flat group.
b) Communication in a simple hierarchical group



Groups & Failure Masking

- Assumption
  - All requests arrive at all servers in the same order (atomic multicast problem)
- When a group can mask any **k** concurrent member failures, it is said to be **k-fault tolerant** (**k** is called *degree of fault tolerance*)
- Question
  - How large should a **k**-fault tolerant group be?
- Answer partly depends on the fault type
  - If **k** processes *stop silently*, then we need **k+1** processes to provide **k**-fault tolerant system
  - If processes exhibit Byzantine failures, we need **2k+1** processes to achieve **k** degree of fault tolerance if the client base its decisions on majority

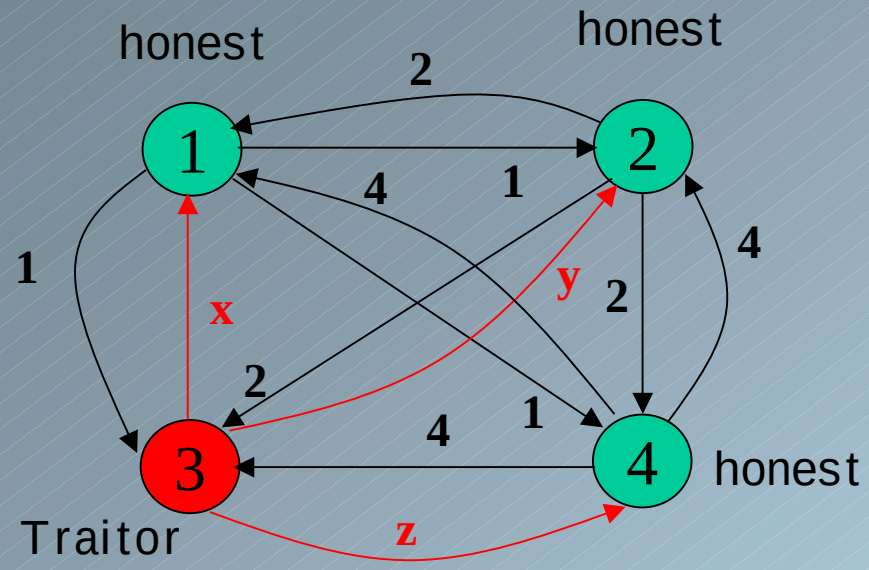
Agreement in Faulty Systems (1)

- Problem:
  - To reach an agreement in face of process failures
  - For example, electing a coordinator, deciding whether or not to commit a transaction, dividing up tasks among workers, and synchronization
  - The problem is more intricate since reaching an agreement needs majority of votes among correctly functioning processes.
  - Another issue is to reach this consensus within a finite number of steps.
- Lamport proved that in a system with **k** faulty processes, agreement can be reached only if **2k+1** correctly functioning processes are present, for a total of **3k+1**

Agreement in Faulty Systems (2)

- Byzantine generals problem:
  - There are **n** blue army generals who want to attack a red army
  - They want to exchange troop strengths
  - After finite number of steps each general will have a vector of length **n** corresponding to all blue armies
  - But, **k** generals are known to be *traitors*
  - Traitors lies to everyone
  - Traitors do not cooperate since it is unknown who the traitors are
  - There is reliable point-to-point communication channels between two generals.

Agreement With Four Processes



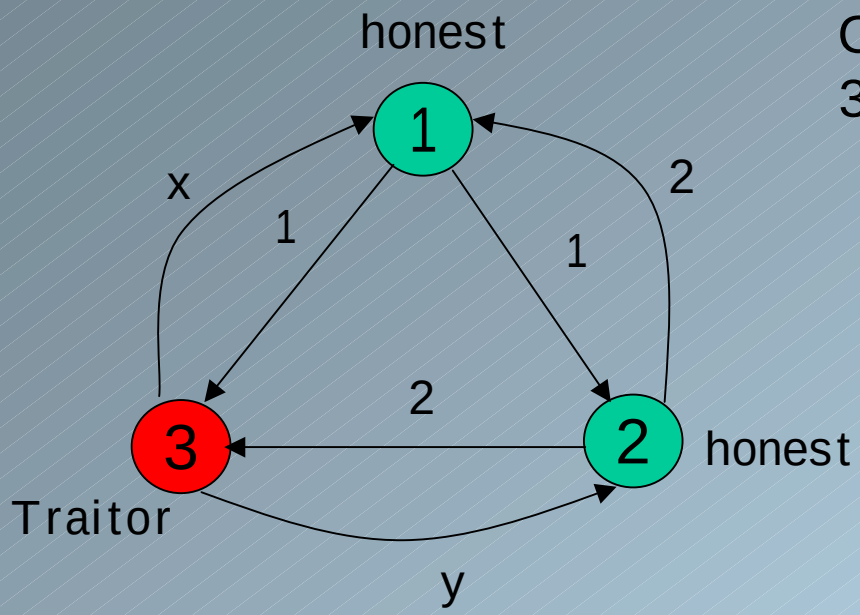
1 Got(1,2,x,4)
2 Got(1,2,y,4)
3 Got(1,2,3,4)
4 Got(1,2,z,4)

1 Got
(1,2,y,4)
(1,b,c,d)
(1,2,z,4)
Result:(1,2,U,4)

2 Got
(1,2,x,4)
(e,2,g,h)
(1,2,z,4)
Result:(1, 2, U, 4)

4 Got
(1,2,x,4)
(1,2,y,4)
(i,j,z,4)
Result:(1, 2, U, 4)

Agreement With Three Processes



Originally, they have 1000, 2000, and 3000 soldiers, respectively

P1: got(1,2,x)
P2: got(1,2,y)
P3: got(1,2,3)

P1 Got
(1,2,y)
(1,b,c)
Result: (1, 2, U)

P2 Got
(1,2,x)
(d,2,f)
Result: (1, 2, U)

Reliable C/S Communication

- So far:
  - Concentrated on process resilience. What about reliable communication channels?
  - A communication channel can crash (e.g. a TCP connection is abruptly broken)
  - Omission failures (a message can get lost)
  - Timing failures
  - Arbitrary failures (e.g. duplicate messages)

Point-to-Point Communication

- Point-to-point communication is established by making use of a reliable transport protocol, such as TCP.
- TCP masks omission failure (i.e. lost messages) by acknowledgements and retransmissions
- Crash failures are not masked (TCP connection is abruptly broken)
  - Distributed system lets the client know that the channel has crashed
  - and takes care of problem by setting up a new connection

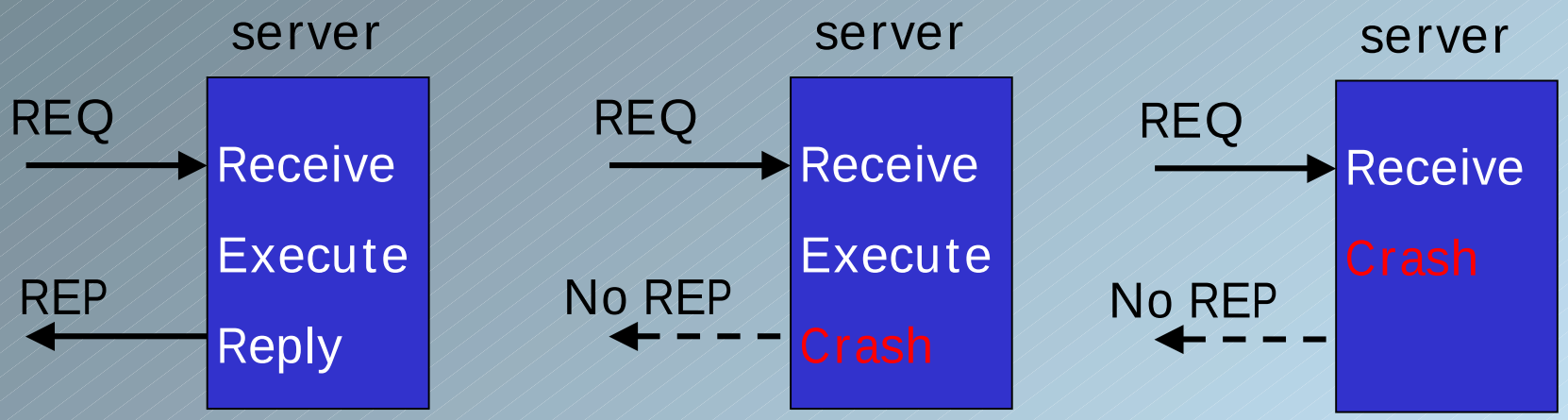
Reliable RPC (RMI)

- Transparency may be lost when an error occurs.
- What can go wrong?
  1. The client is unable to locate the server
  2. The request message from the client to the server is lost
  3. The server crashes after receiving a request
  4. The reply message from the server to the client is lost
  5. The client crashes after sending the request
- What to do?

  [1:] Relatively simple – just report back to client (raise an exception or use signals)

  [2]: Just resend the message (if an acknowledgement does not arrive after a certain time period elapsed)

Server Crashes (1)



- A server in client-server communication

a) Normal case

b) Crash after execution (report failure to the client)

c) Crash before execution (retransmit the request)

Server Crashes (2)

- Three schools of thought exist on what to do when the server crashes

1. **At least once semantics:** guarantees that the RPC has been carried out at least once, possibly more
2. **At most once semantics:** guarantees that the RPC has been carried out at most one time, possibly none at all.
3. No guarantee at all.

- What we want is **exactly once semantics**.

Server Crashes: Print Example (1)

- Example:
  - The client sends a (print) request to a server
  - The server crashes and subsequently recovers
  - The server announces it is up and running again
  - Client is not sure if its request has been carried out

- Server's Strategies:
  1. Send ACK after delivering the request to the printer
  2. Send ACK after the text is printed out

Server Crashes: Print Example (2)

- Client's strategies
  1. Never reissue the request
  2. Always reissue the request
  3. Reissue the request only if it did not receive an acknowledgement
     - client assumes that server crashed before the request is delivered to the printer
  4. Reissue the request only if it has received an acknowledgement
     - client assumes that server sent acknowledgement, and crashed before sending the document for printing

Server Crashes: Print Example (3)

- Situation: Two strategies for server and four for clients, there are eight combinations. None of them offers a satisfactory solution for every case
- There are three events that can happen at the server in different orders
  - (**M**) : send the completion message
  - (**P**) : print the message
  - (**C**): crash

Server Crashes: Print Example (4)

1. **M→P→C:** a crash occurs after sending 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

Server Crashes: Print Example (5)

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

- Different combinations of client and server strategies in the presence of server crashes.

[4:] Lost Reply Messages

- Detecting lost replies can be hard;
  - the client cannot know why there is no answer: the reply message is lost or the server crashed or it is just slow.
  - i.e. the client is not sure whether the request is carried out or not
- Solution
  - The client resend the request and does nothing to prevent duplicate execution if the request is an idempotent
  - Otherwise; it can assign each request a unique sequence number.
  - Or it indicates if the request is original or a reissue.
  - The last two solutions put extra burden on the server

[5:] Client Crashes (1)

- The problem
  - The server is processing the request and tying up the valuable resources for nothing (orphan computation)
- Solutions
  1. Extermination: the client keeps a log on persistent storage that survives the crashes; when the client recovers, it checks the log file and kills explicitly the orphan computation
  2. Reincarnation: divide time up into sequentially numbered *epochs*
     - when a client recovers, it announces the starting of a new epoch;
     - when announcement is received, the orphan computations are killed on behalf of the client. If the network is partitioned some orphans can survive