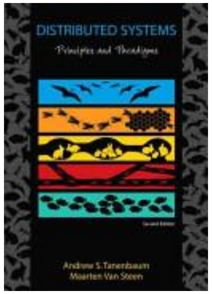
Chapter 8: FAULT TOLERANCE I

Continue to operate even when something goes wrong!



Thanks to the authors of the textbook [**TS**] for providing the base slides. I made several changes/additions.
These slides may incorporate materials kindly provided by Prof. Dakai Zhu.
So I would like to thank him, too.

Turgay Korkmaz
korkmaz@cs.utsa.edu

Chapter 8: FAULT TOLERANCE

- INTRODUCTION TO FAULT TOLERANCE
  - Basic Concepts, Failure Models
- PROCESS RESILIENCE
  - Design Issues, Failure Masking and Replication
  - Agreement in Faulty Systems, Failure Detection
- RELIABLE CLIENT-SERVER COMMUNICATION
  - Point-to-Point Communication, RPC Semantics
- RELIABLE GROUP COMMUNICATION
  - Basic Reliable-Multicasting Schemes, Scalability
  - Atomic Multicast
- DISTRIBUTED COMMIT
  - Two-Phase Commit, Three-Phase Commit
- RECOVERY
  - Introduction
  - Checkpointing
  - Message Logging
  - Recovery-Oriented Computing

Objectives

- To understand failures and their implications
- To learn about how to deal with failures
-

What is Fault Tolerance?

From Merriam-webster:

- **Failure** is a state of inability to perform a normal function (e.g., a received msg corrupted)
- **Error** is an act involving an unintentional deviation from truth or accuracy (e.g., reading 1 instead of 0)
- **Fault** is ….

From our textbook

- **Fault** is the *cause* of an error that may need to a failure (e.g., software bugs, broken line, or weather)
- It is important to find out what may cause an error and construct the system in such a way that it can tolerate faults (i.e., automatically recover and continue to operate (e.g., re-transmit damaged msg))

Failure in….

Distributed Systems

- Failure is **partial**
- Some components might be still working
- Entire system may still function

Non-Distributed systems

- Failure is **total**
- All components would be affected
- Entire system may be down

Questions:

Can we hide the effects of faults?

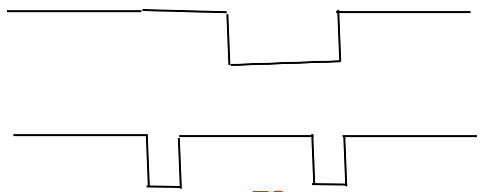
Can we recover from partial failures?

Answers are strongly related to what are called **dependable systems**

Dependable Systems

- A component provides services to clients. To provide services, the component may require the services from other components → a component may depend on some other component.
- Dependability implies the following:
  - Availability ready to be used
  - Reliability run continuously w/o failure
  - Safety temp failure should not cause catastrophic happens
  - Maintainability how easy to repair a failed system
  - Security (ch 9)?

High *availability* == high *reliability*?



How to build a dependable system?

How to control faults?

- Fault **prevention**
  - prevent the occurrence of a fault
- Fault **removal**
  - reduce the presence, number, seriousness of faults
- Fault **forecasting**
  - estimate the present number, future incidence, and the consequences of faults
- Fault **tolerance**
  - build a component in such a way that it can meet its specifications in the presence of faults (i.e., **mask** the presence of faults)

Types of Faults

- **Transient** faults
  - Occur once and then disappear
  - E.g., disturbance during wireless communication
  - Try it again, it will work next time!
- **Intermittent** faults
  - Disappear and reappear: **unpredictable (**and **notorious)**
  - E.g., loose contact on a connector
  - Hard to detect since it sometimes works or do not work!
- **Permanent** faults
  - Continue to exist until faulty components are repaired/replaced
  - E.g., software bugs or burnt out chips

Failure Models

In DS, we have a collection of **servers** and **channels.**
System may fail because servers, channels, or both are not working...

There are various types of failures:

- **Crash** failure
  - component simply halts, but behaves correctly before halting
- **Omission** failure
  - component fails to receive or send
- **Timing** failure
  - correct output, but lies outside a specified real-time interval
- **Response** failure
  - incorrect response (wrong value or state transition)
- **Arbitrary/Byzantine failure:**
  - Arbitrary/Malicious output
  - Cannot be detected easily

Failure Detection

- How can clients distinguish between a crashed component and one that is just a bit slow?
  - Consider a server from which a client is expecting output
    - Is the server perhaps exhibiting timing or omission failures?
    - Is the channel between client and server faulty?
- Assumptions we can make
  - **Fail-stop** : The component exhibits crash failures, but its failure can be detected (either through announcement or timeouts)
  - **Fail-silent** : The component exhibits omission or crash failures; clients cannot tell what went wrong
  - **Fail-safe** : The component exhibits arbitrary, but benign failures that cannot do any harm (e.g., junk output that can be recognized)

Fault Tolerance Techniques

- **Redundancy**: key technique to tolerate faults
  - Hiding failures and effect of faults

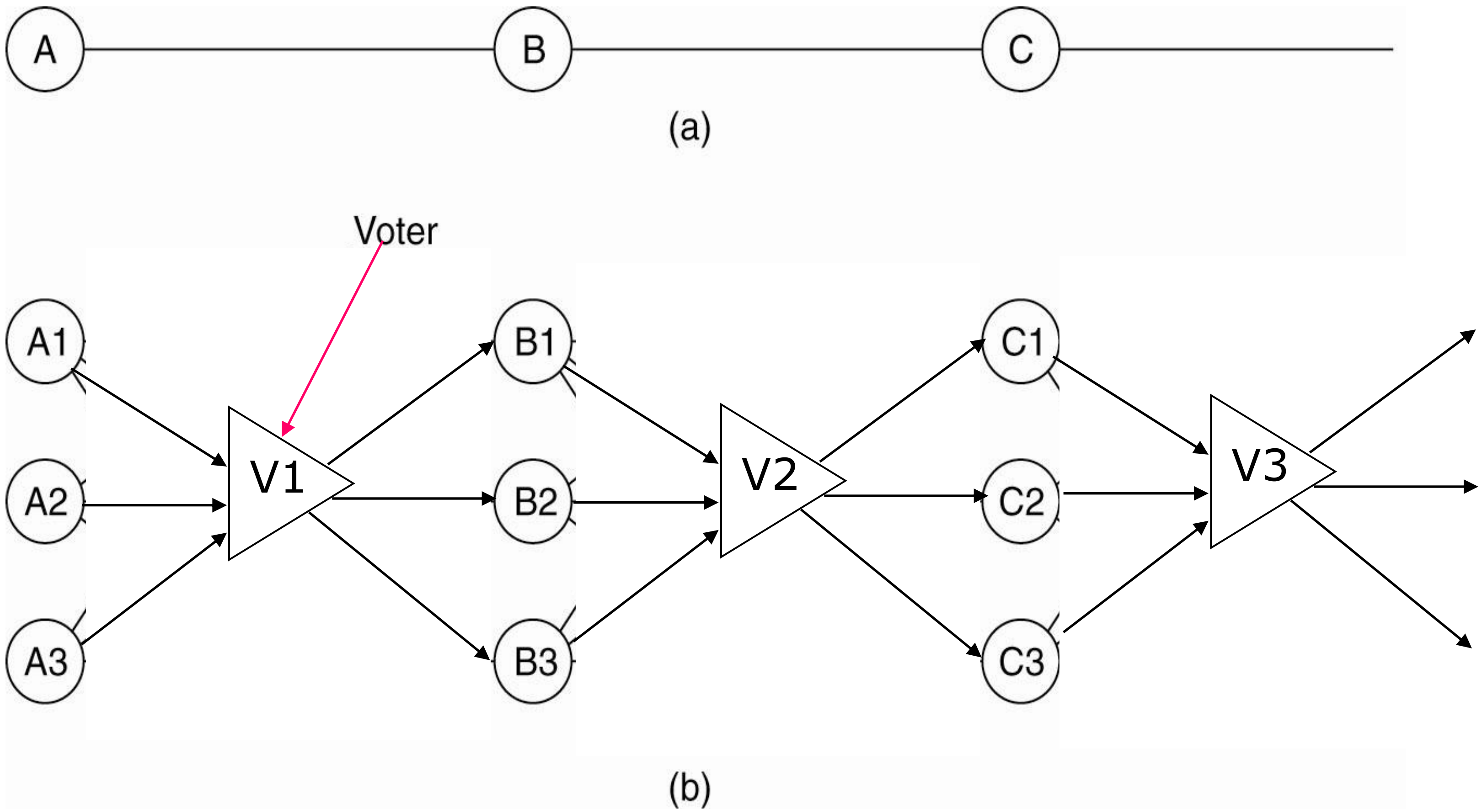
- **Recovery** and rollback (more later in Section 8.6)
  - Bringing system to a consistent state

Redundancy Techniques

- **Information** redundancy
  - e.g., parity bit and Hamming codes
- **Time** redundancy
  - Repeat action
  - e.g., re-transmit a msg
- **Physical** (software/hardware) redundancy
  - Replication
  - e.g., extra CPUs, multi-versions of a software

Physical Redundancy

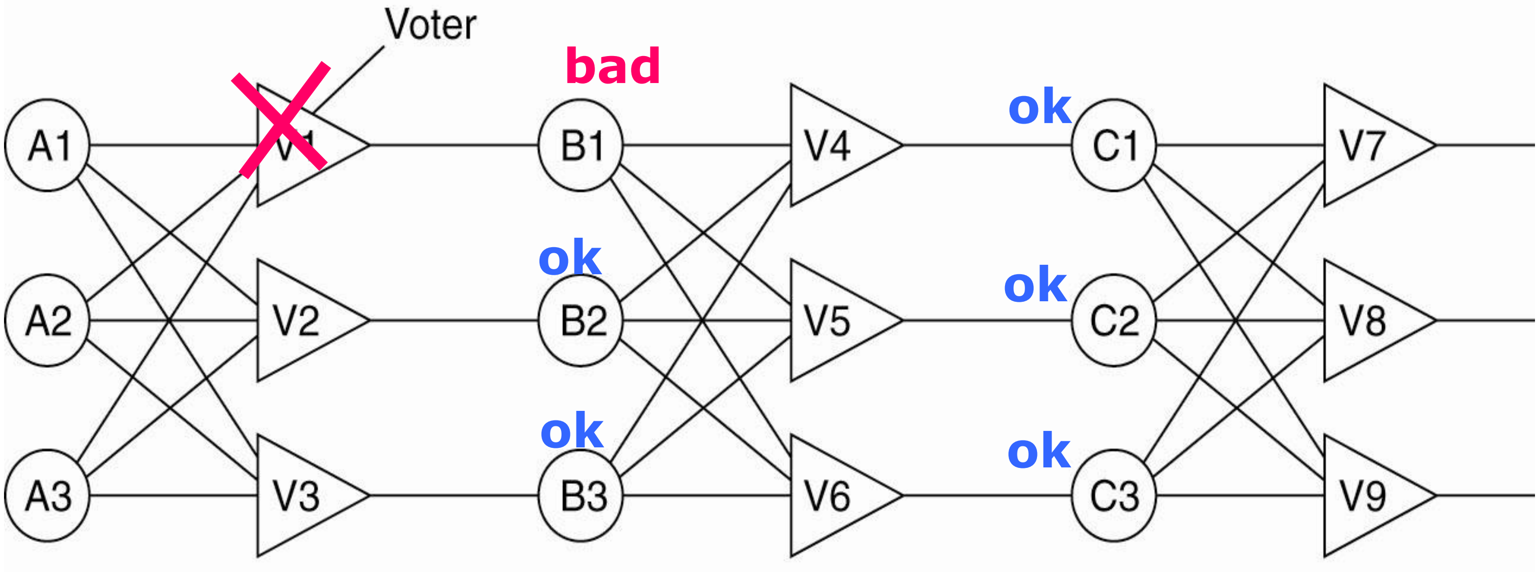
Triple Modular Redundancy (TMR)



- If A2 fails → V1: majority vote → B gets good result
- What if V1 fails?!

TMR (cont.)

- Correct results are obtain via **majority vote**
  - Mask **ONE** fault



(b)

Assume that prob Vx fails is 0.1
What is the probability that the above system fails?

Protect yourself against faulty processes by **replicating** and distributing computations in a group.

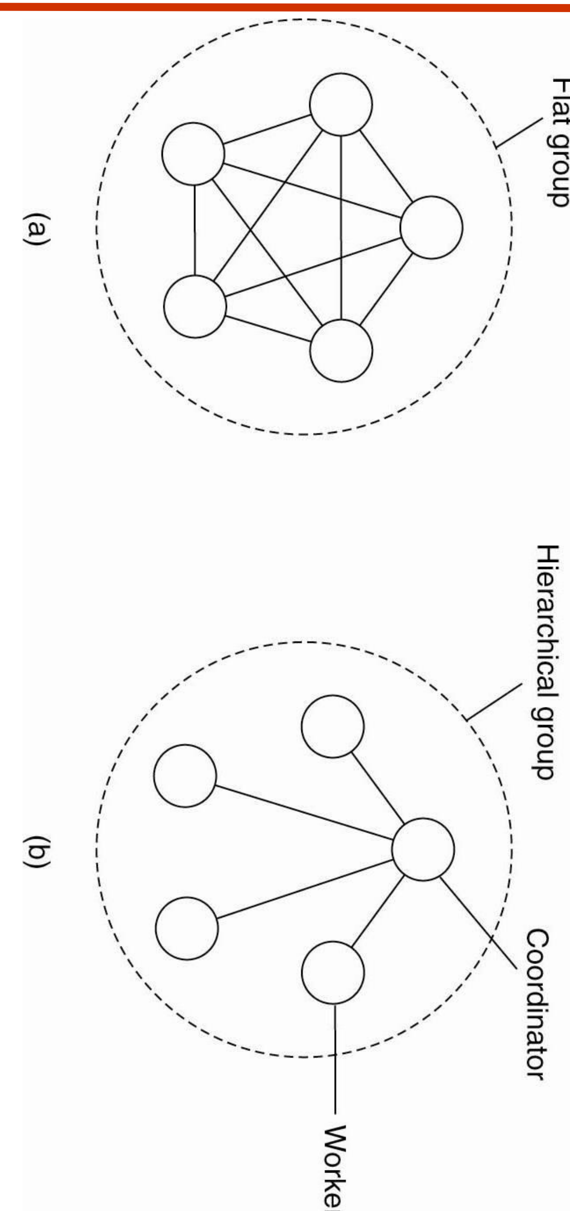
PROCESS RESILIENCE

Design Issues

- To tolerate a faulty process, organize several identical processes into a group
- A **group** is a *single abstraction* of a collection of processes
  - So we can send a message to a group without explicitly knowing who are they, how many are there, or where are they (e.g., e-mail groups, newsgroups)
  - *Key property:* When a message is sent, all members of the group must receive it. So if one fails, the others can take over for it.
- Groups could be dynamic
  - So we need mechanisms to manage groups and membership (e.g., join, leave, be part of two groups)

Flat vs. Hierarchical Groups

- Flat groups: information exchange immediately occurs with all group members
  - + good for fault tolerance,
  - + no single point of failure
  - - may impose more overhead as control is completely distributed
  - - hard to implement
- Hierarchical groups: All communication through a single coordinator
  - - not really fault tolerant or scalable,
  - + but relatively easy to implement.



Group Membership

How to add/delete groups and manage join/leave groups?

- **Centralized**: have a group server to maintain a database for each group and get these requests
  - Efficient, easy to implement, but single point of failure
- **Distributed**:
  - to join a group, a new process can send a message to all group members that it wishes to join the group *(Assume that reliable multicasting is available)*
  - To leave, a process can ideally send a goodbye msg to all, but if it crashes (not just slow) then the others should discover that and remove it from the group!
  - What if many leaves…. Re-build the group….

Failure masking by Replication

Use protocols from Ch 7:

- Primary-based
  - Organize processes in an hierarchical fashion
  - Primary coordinates all W operations
  - Primary is fixed but its role can be taken by a backup
  - If the primary fails, backups elect a new primary
- Replicated write protocols
  - Organize processes into flat group
  - W operations are performed using active replication or quorum-based protocols
  - No single point of failure, but distributed coordination cost
- How much replication is needed or enough?

Level of Redundancy

K-Fault Tolerance

- A system is said to be **k-fault tolerant** if it can survive faults in k components and still meet its specifications….
- How many components (processes) do we need to provide k-fault tolerance?
- Depends on what kind of faults can happen?

Level of Redundancy

- Assume crash failure semantics (i.e., **fail-stop**)
  - **k + 1** components are needed to survive *k* failures
    - if k of them stops, the last one can still take over
  - Ensure **at least one functional component** !
- Assume **arbitrary/Byzantine** (but **non-malicious**) failure semantics (i.e., continue to run when sick and send out random or erroneous replies)
  - Suppose group output is defined by voting and component failures are independent
  - **2k+1** components are needed
    - If k wrong then (k+1) must be good to have majority
  - Theoretically correct, but hard to convince**: k+1** vs. **k** (some statistical analysis is needed)

Level of Redundancy:

Agreement Problem

- Problem: Assume **Byzantine (malicious)** failure semantics and need **agreement** on non-faulty components
  - Faulty components cooperate to cheat!!!
  - 3*k*+1 components are needed to tolerate *k* failures
  - **Agreement** is possible only if more than **two-thirds** of components work properly.
  - In democracy, usually majority vote is enough but for certain things 2/3 is required (e.g., CS bylaws). Why do you think this might be the case?

Agreement in Faulty systems (1)

- A process group is required to reach an agreement for many things (e.g., electing a coordinator, deciding to commit a transaction or not, dividing tasks among workers, synchronization etc.),
- If all processes and communication channels are perfect, it is easy to reach an agreement.
- But not!
- *So the goal is to have all non-faulty processes reach consensus and establish this consensus within a finite number of steps!*
- Solutions differ under different assumptions.

Agreement in Faulty systems (2)

Reaching agreement is only possible for below cases

Process behavior	Communication delay	Message ordering: Unordered — Message transmission: Unicast	Unordered — Multicast	Ordered — Unicast	Ordered — Multicast
Synchronous	Bounded	X	X	X	X
Synchronous	Unbounded			X	X
Asynchronous	Bounded				X
Asynchronous	Unbounded	In practice			X

Sync: if any process has taken c+1 steps,
then every other has taken at least 1 step

Async: if not sync

Byzantine Agreement Problem

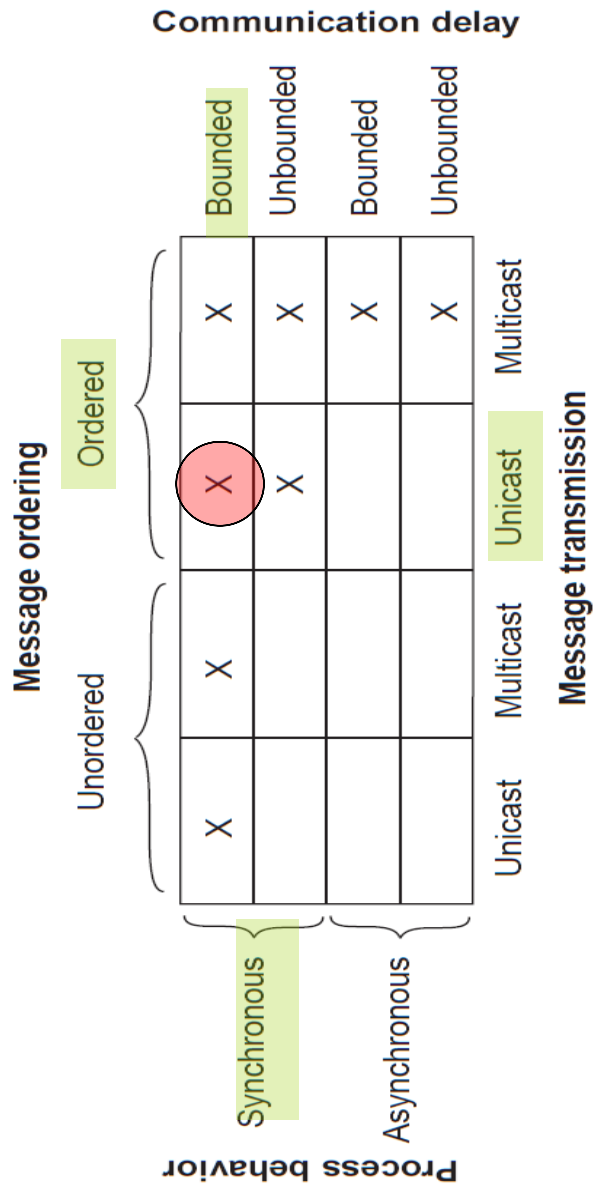
- N generals including k traitors

- **Problems**:

  Can trusted generals agree on their army sizes?

  What should be N and k?

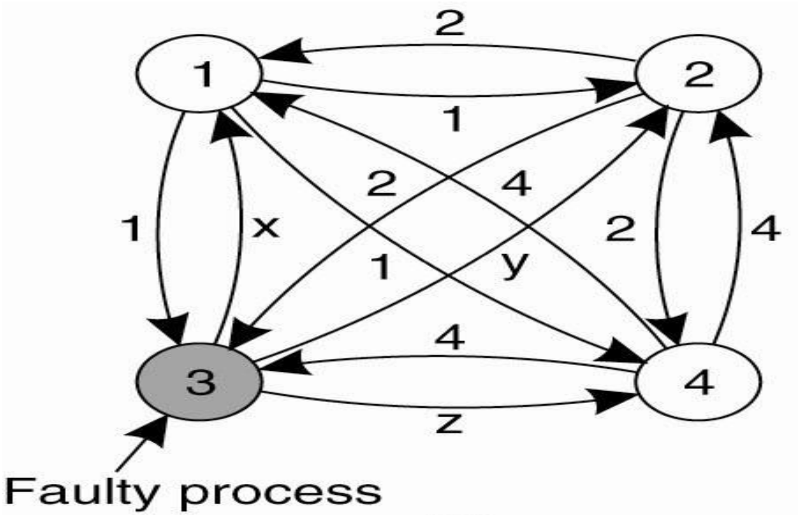
- **Assumptions**:
  - Traitors can lie, others don't know who the traitors are
  - Reliable communication channel more specifically …



Lamport's Agreement Algorithm

1. Each general i sends its army size v_i to others
   - Loyal generals tell the truth
   - Traitors can lie
2. Each general collects received information as a vector s.t. V[i] == v_i if general i is non-faulty
3. Each general sends its vector to others
   - Loyal generals send what they have
   - Traitors can change the vectors
4. Each general determines vector elements by voting among all vectors he/she receives

An Example: N=4, k=1



(a)

N = 3*k+1 for agreement

Majority vote?

<u>1 got</u>	<u>2 got</u>	<u>3 got</u>
1 2 ? 4	1 2 ? 4	1 2 ? 4

(d)

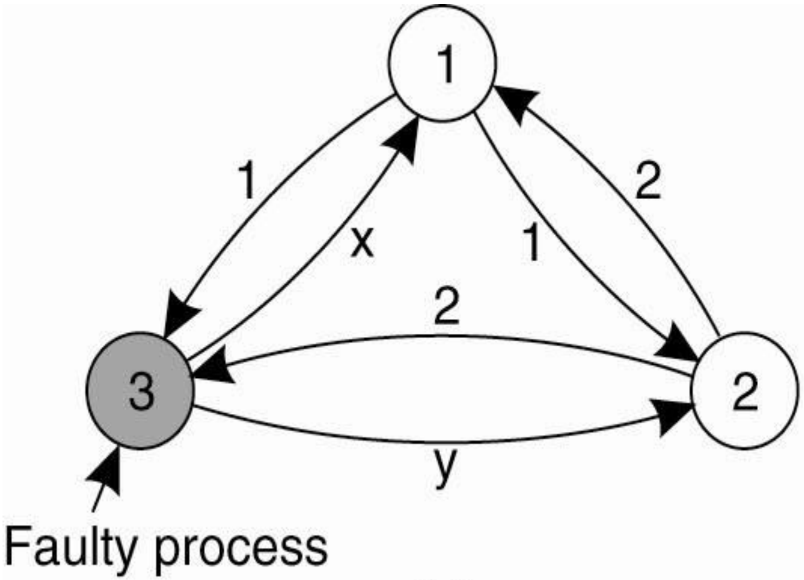
1	Got(1, 2, x, 4)
2	Got(1, 2, y, 4)
3	Got(1, 2, 3, 4)
4	Got(1, 2, z, 4)

(b)

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)

(c)

An Example: N=3, k=1



(a)

For agreement, we need at least 2k+1 correctly functioning nodes + k faulty ones so N is 3k+1….

Majority vote?

1 got	2 got
? ? ?	? ? ?

(d)

Fail to agree!

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

(b)

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

(c)

Failure detection

- How can we decide if a node is failed or just slow?
- There are essentially two mechanisms:
  - Actively send “Are you alive” and expect an answer or passively wait until messages come from others
  - Use timeouts:
    - Setting timeouts properly is difficult and application dependent
    - Premature timeouts generates false positives
    - You cannot distinguish process failures from network failures
- Also all non-faulty processes need to decide (agree on) who is failed and still a member or not!
  - Consider failure notification throughout the system:
    - Gossiping (i.e., proactively disseminate a failure detection)
    - On failure detection, pretend you failed as well to propagate it recursively