Fault Tolerance in Distributed Systems

Topics to be covered based on these questions

- What is a fault, an error and a failure?
- What is fault tolerance? How are systems made dependable?
- What are dependability requirements?
- Specify various types of faults
- How to develop a fault tolerant system?

  Masking failures?
- How to protect processes from failures?
- Agreement in faulty system – Byzantine Agreement

Basic Concepts

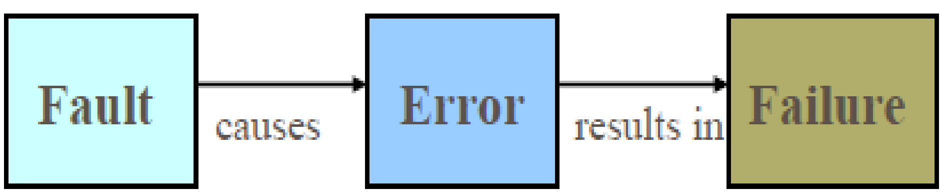
Fault – is a defect within the system

Error – is observed by a deviation from the expected behaviour of the system

Failure - occurs when the system can no longer perform as required

Erroneous state – It is a state which could lead to a system failure

Failure Recovery – is a process that involves restoring an erroneous state to an error-free state



Fault Tolerance - Dependability

- Availability – ready most of the time
- Reliability – running continuously without failure
- Safety – temporarily fails but no catastrophic happens
- Maintainability – how easily a system can be repaired

Types of faults

- Transient

  - occur once and disappears

    eg: transmitter cause loss of bits

- Intermittent

  - occurs, vanishes and then reappears and so on

  e.g: loose contact on a connector

- Permanent

  - continues to exist till faulty component is replaced e.g: burnt out chips, s/w bugs

Requirement Specification

- Failure types:
  - Some are more probable than others
  - Some are transient, others permanent
  - Some occur in hardware, others in software

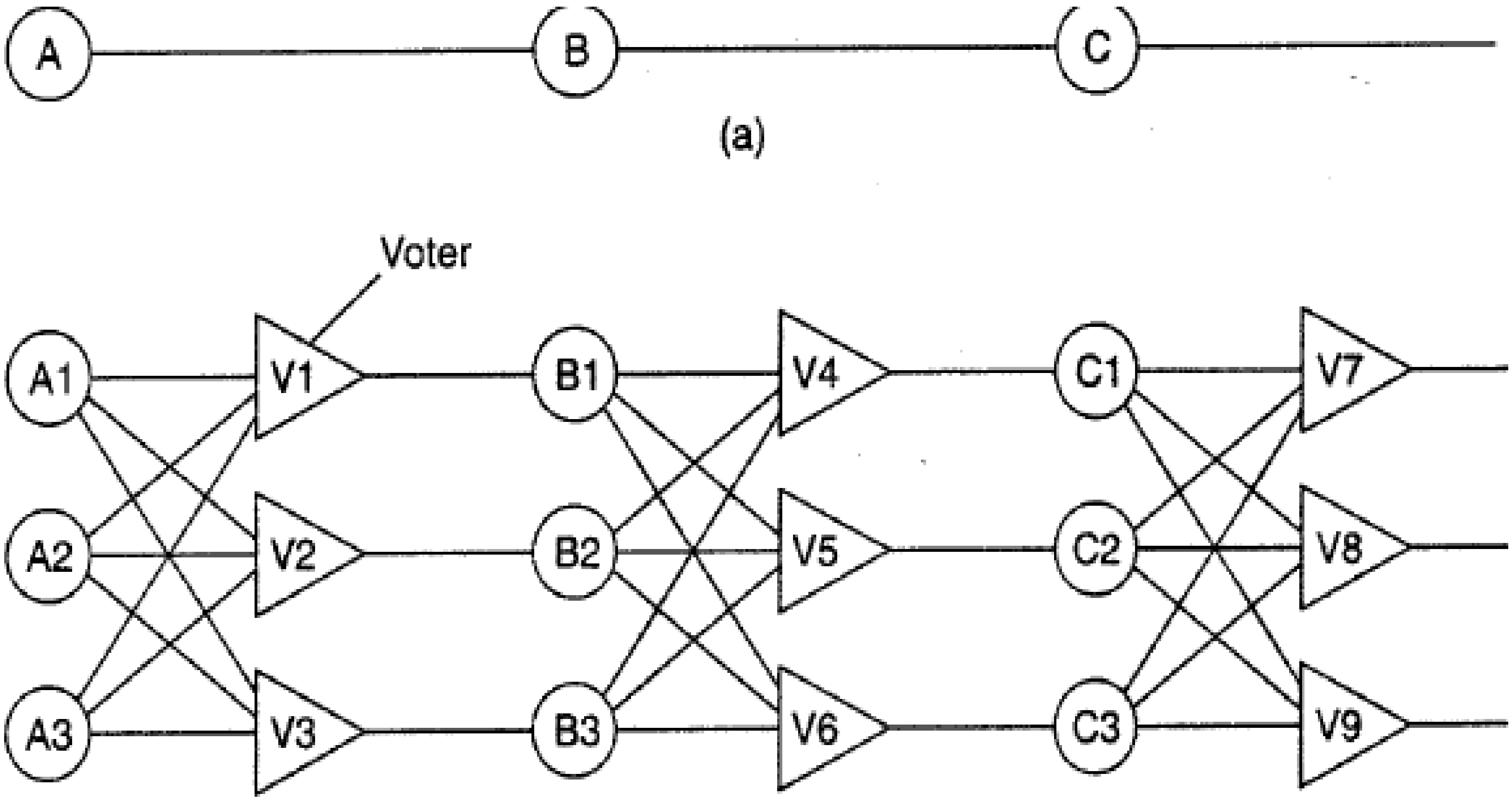
Design

- Fault Tolerant system: Hides the occurrence of failure from other processes
- Design of systems that tolerate faults that occur while system is in use
- Masking faults : Basic Principle – Redundancy
  - Spatial/Physical - redundant hardware/processes
  - Informational - redundant data structures – Hamming code
  - Temporal/Timing - redundant computation - transactions

- Redundancy costs money and time
  - Optimize the design by trading off amount of redundancy used against the desired level of fault tolerance
  - Temporal redundancy usually requires re-computation and it results in a slower recovery from failure
  - Spatial has faster recovery but increases hardware costs, space, power etc. requirements

- Commonly Used Techniques for Redundancy
  - Modular redundancy
    - Uses multiple, identical replicas of hardware modules and a voter mechanism
    - The outputs from the replicas are compared, and correct output is determined - majority vote
    - Can tolerate most hardware faults that can affect the minority of the hardware modules

Triple Modular Redundancy



- N- Version Programming
  - Write multiple versions of a software module
  - Outputs from these versions are received and correct output is determined via voting mechanism
  - Each version is written by different team, with the hope that they will not contain the same bugs
  - Can tolerate software bugs that affect a minority of versions
  - Cannot tolerate correlated fault - reason for failure is common to two (or more) modules eg two modules share a single power supply, failure of which causes both to fail

Classification of failures

- Process failures

    deadlocks, timeouts, wrong input by the user

- System failures

    process fails to execute

    caused by s/w errors and h/w problems

Failure models

- Omission Failure
  - A server omits to respond to a request or receive request
- Response Failure
  - Value failure - returns wrong value
  - State transition failure - has wrong effect on resources

- Timing Failure- any response that is not available to a client within a specified real time interval

- Server Crash Failure: a server repeatedly fails to respond to requests until it is restarted
  - Amnesia- crash - a server starts in its initial state, having forgotten its state at the time of the crash, ie loses the values of the data items
  - Pause- crash - a server restarts in the state before the crash
  - Halting- crash - server never restarts
- Arbitrary failure: A server may produce arbitrary responses at arbitrary times.

Handling Failures

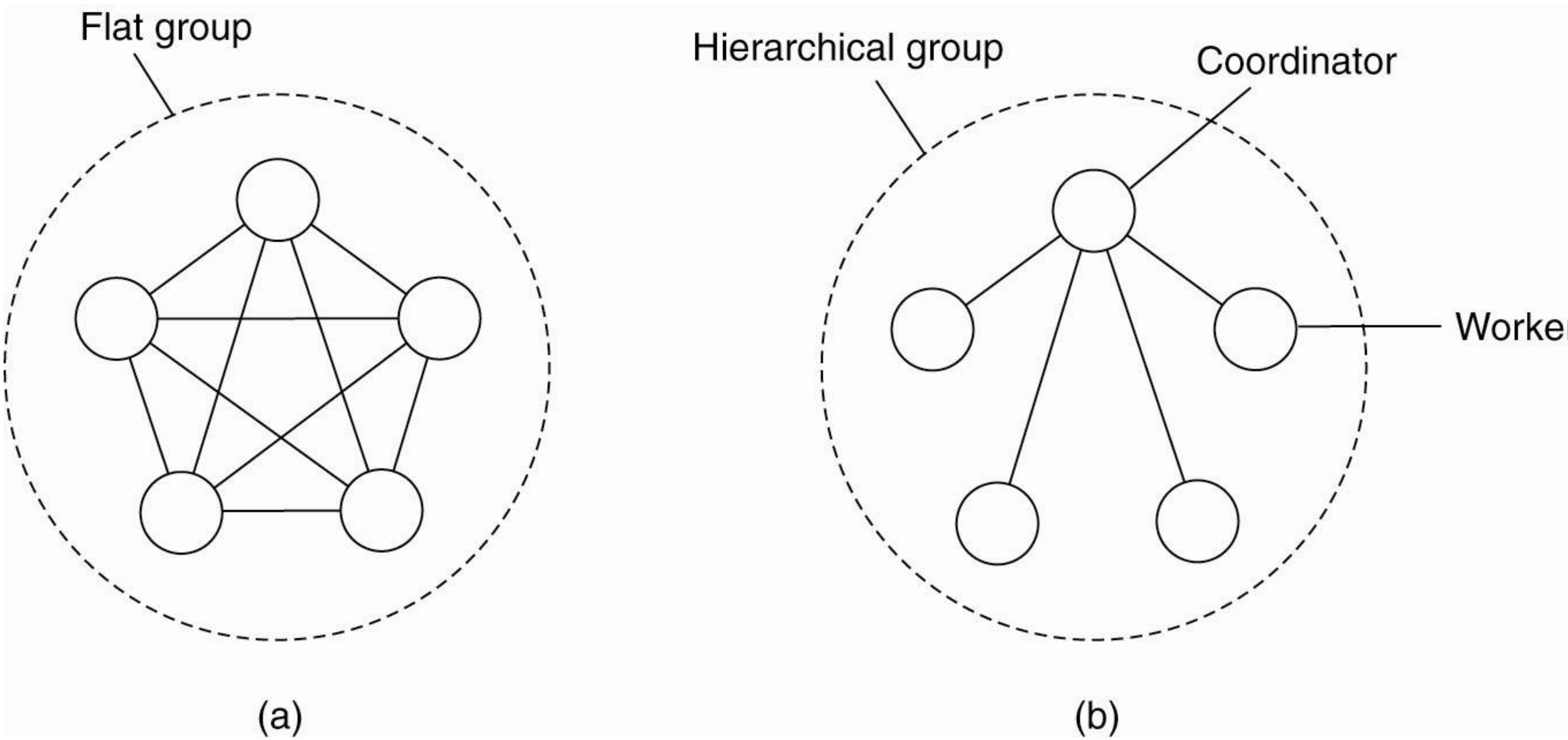
Process Resilience

- Processes can be made fault tolerant by arranging to have a group of processes, with each member of the group being identical .

- A message sent to the group is delivered to all of the “copies” of the process (the group members), and then only one of them performs the required service.

- If one of the processes fail, it is assumed that one of the others will still be able to function (and service any pending request or operation

Flat Groups versus Hierarchical Groups



- **Communication in a flat group** – all the processes are equal, decisions are made collectively.
  - **Note**: no single point-of-failure, however: decision making is complicated as consensus is required.
- **Communication in a simple hierarchical group** - one of the processes is elected to be the coordinator, which selects another process (a worker) to perform the operation.
  - **Note**: single point-of failure, however: decisions are easily and quickly made by the coordinator without first having to get consensus.

Failure masking and Replication

- By organizing a *fault tolerant group of processes* , we can protect a single vulnerable process.
- Two approaches to arranging the replication of the group:

Primary (backup) Protocols

  - A group of processes is organized in a hierarchical fashion in which a primary coordinates all write operations.
  - When the primary crashes, the backups execute some election algorithm to choose a new primary.

Replicated-Write Protocols

  - Replicated-write protocols are used in the form of active replication, as well as by means of quorum-based protocols.
  - Solutions correspond to organizing a collection of identical processes into a flat group.

Agreement in Faulty Systems

- Goal of distributed agreement algorithms to have all the non-faulty processes reach agreement on some issue, and to establish that agreement within a finite number of steps.

Agreement issue in Faulty Systems

Possible assumptions about the underlying system:

1. Synchronous versus asynchronous systems.
2. Communication delay is bounded or not.
3. Message delivery is ordered or not.
4. Message transmission is done through unicasting or multicasting.

Circumstances under which distributed agreement can be reached

		Message ordering			
		Unordered		Ordered	
Process behavior	Communication delay	Unicast	Multicast	Unicast	Multicast
Synchronous	Bounded			X	
	Unbounded			X	
Asynchronous	Bounded	X	X	X	X
	Unbounded			X	X

Message transmission

Agreement in Faulty Systems

- How should processes agree on results of a computation?
- *K-fault tolerant*: system can survive k faults and yet function
- Assume processes fail silently
  - Need (k+1) redundancy to tolerant k faults
- *Byzantine failures*: processes run even if sick
  - Produce erroneous, random or malicious replies
    - Byzantine failures are most difficult to deal with
  - Need ? Redundancy to handle Byzantine faults

Byzantine Agreement Problem: Lamport et al. 1982

- Assume reliable synchronous ordered unicast based message system. There are N process, k of which may act as faulty or even malicious. A faulty process may send different values to different processes.

Byzantine failure

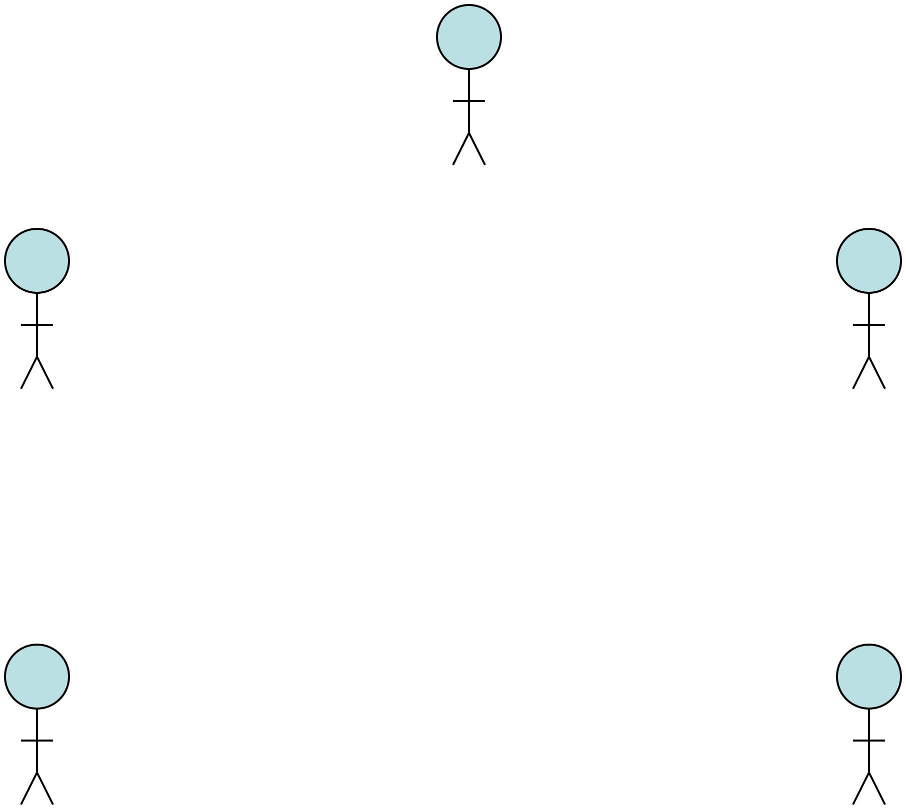
- In fault-tolerant distributed computing, a Byzantine failure is an arbitrary fault that occurs during the execution of an algorithm in a distributed system. When a Byzantine failure has occurred, the system may respond in any unpredictable way.
- These arbitrary failures may be loosely categorized as follows:
  - a failure to take another step in the algorithm, also known as a crash failure;
  - a failure to correctly execute a step of the algorithm; and
  - arbitrary execution of a step other than the one indicated by the algorithm.

Byzantine failure

- Byzantine refers to the Byzantine Generals' Problem, an agreement problem in which generals of the Byzantine Empire's army must decide unanimously whether or not to attack some enemy army.
- The problem is complicated by the geographic separation of the generals, who must communicate by sending messengers to each other, and by the presence of traitors amongst the generals.
- These traitors can act arbitrarily in order to force good generals into a wrong decision: trick some generals into attacking; force a decision that is not consistent with the generals' desires,
  - e.g. forcing an attack when no general wished to attack; or so confusing some generals that they never make up their minds. If the traitors succeed in any of these goals, any resulting attack is doomed, as only a concerted effort can result in victory.
  - Lamport et al., proved that with number of bad generals 1/3 or less, there is a solution.

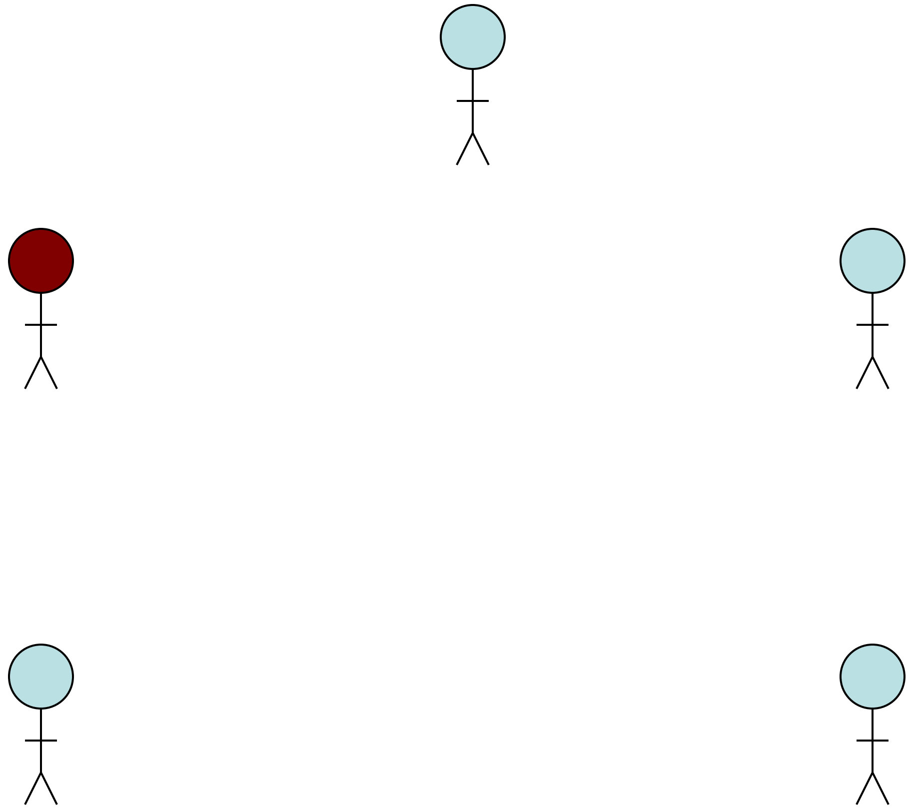
The Byzantine Generals Problem: Distributed Consensus

Let us assume we have five generals…



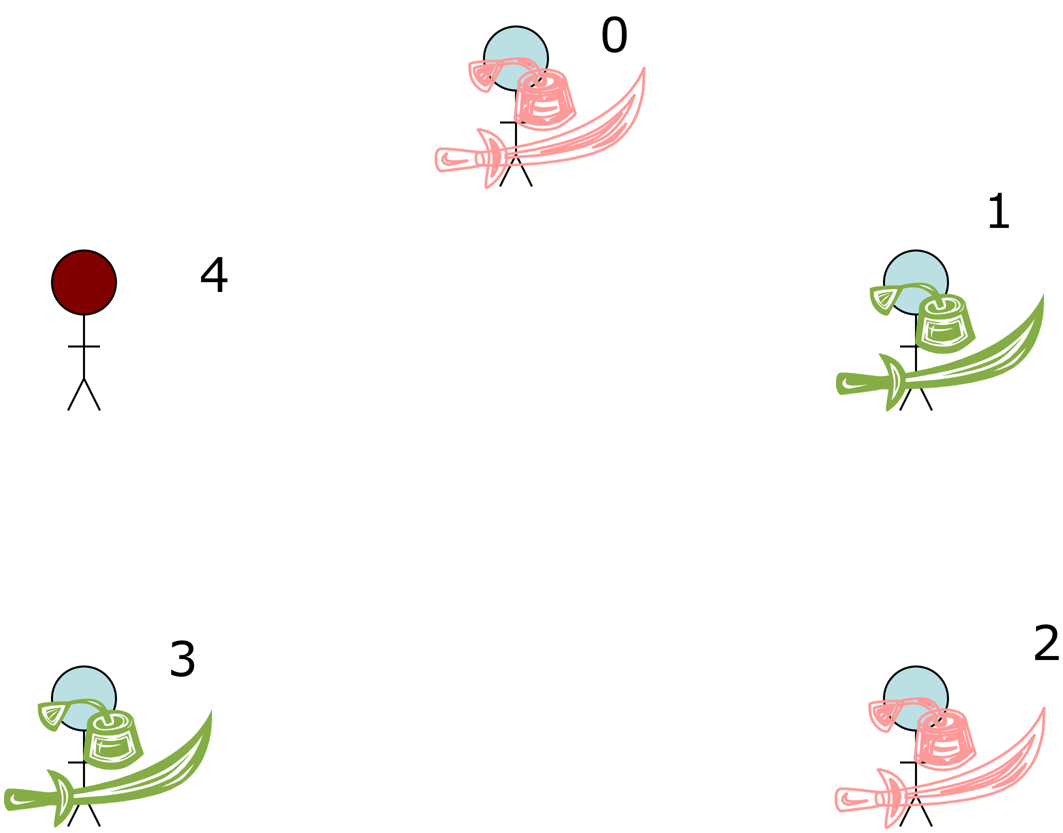
The Byzantine Generals

Let us assume one is malicious…



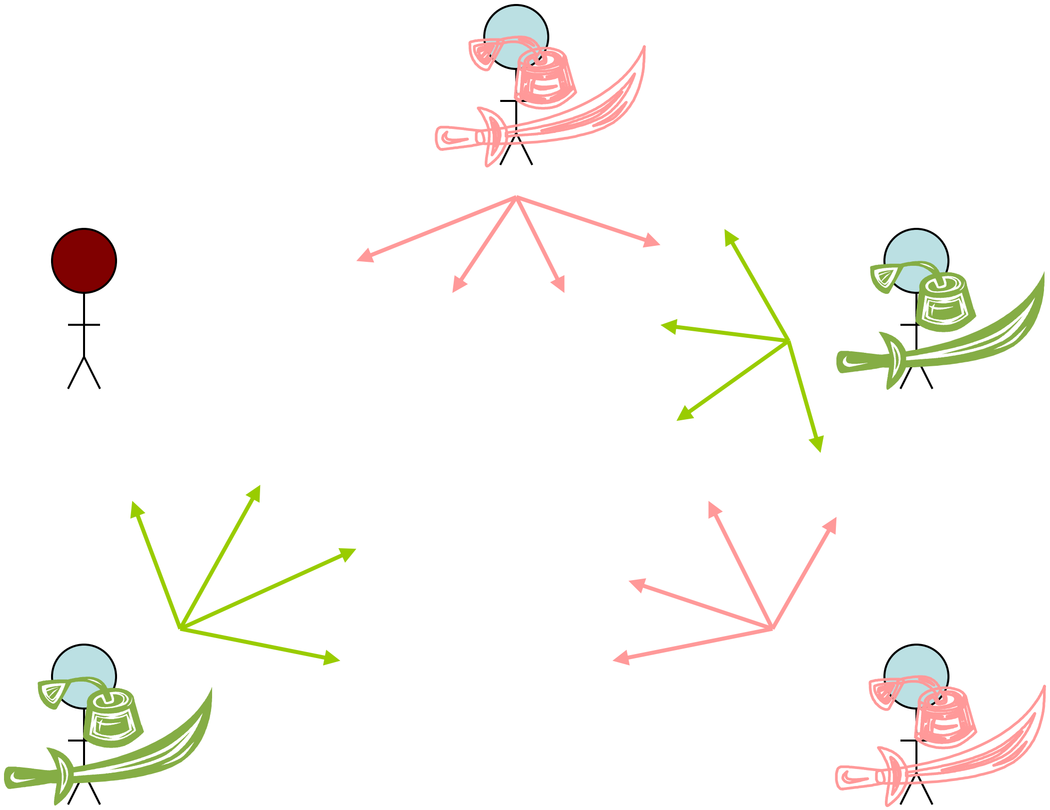
The Byzantine Generals

Each local general decides on an attack plan…



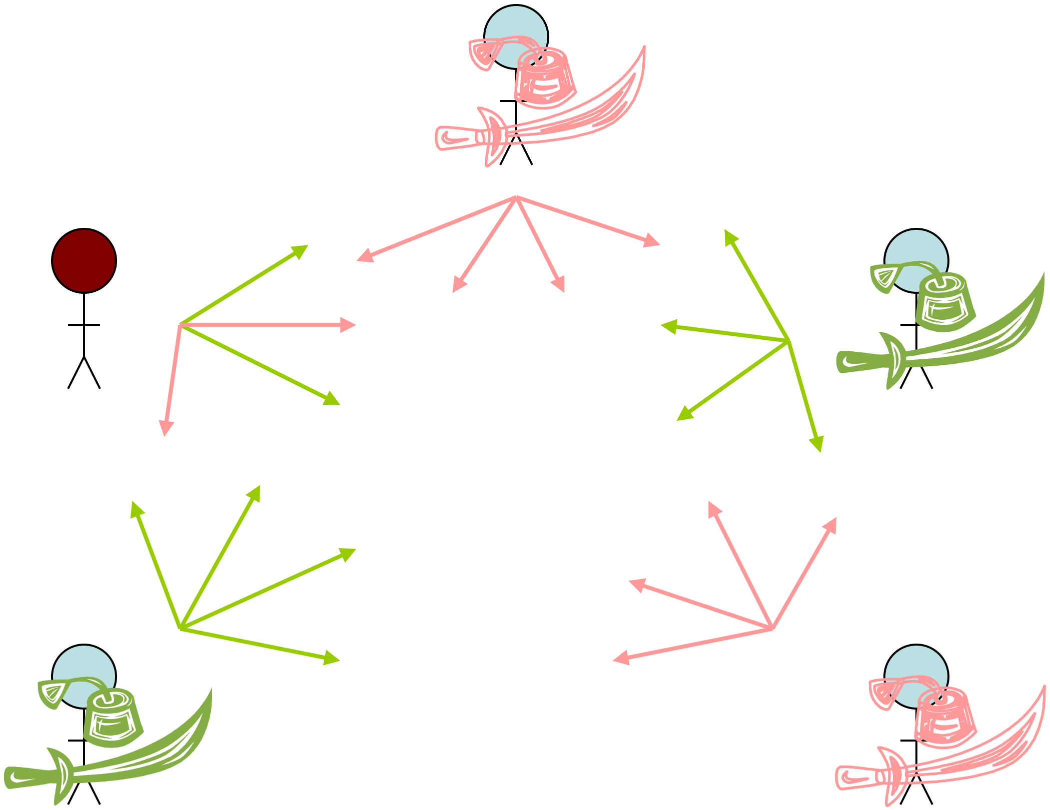
The Byzantine Generals

… and accurately relays their plan …



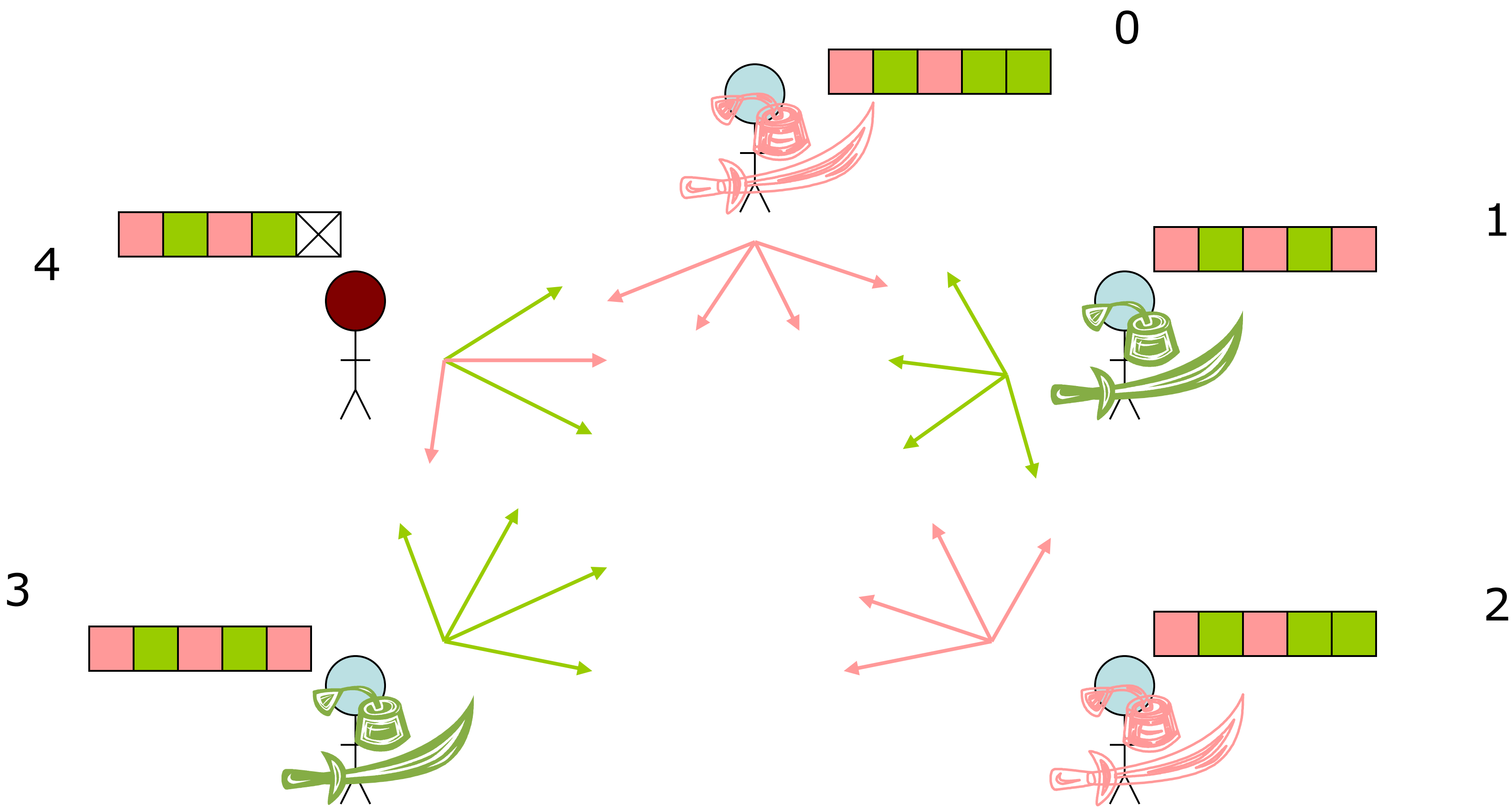
The Byzantine Generals

…except the random malicious one…



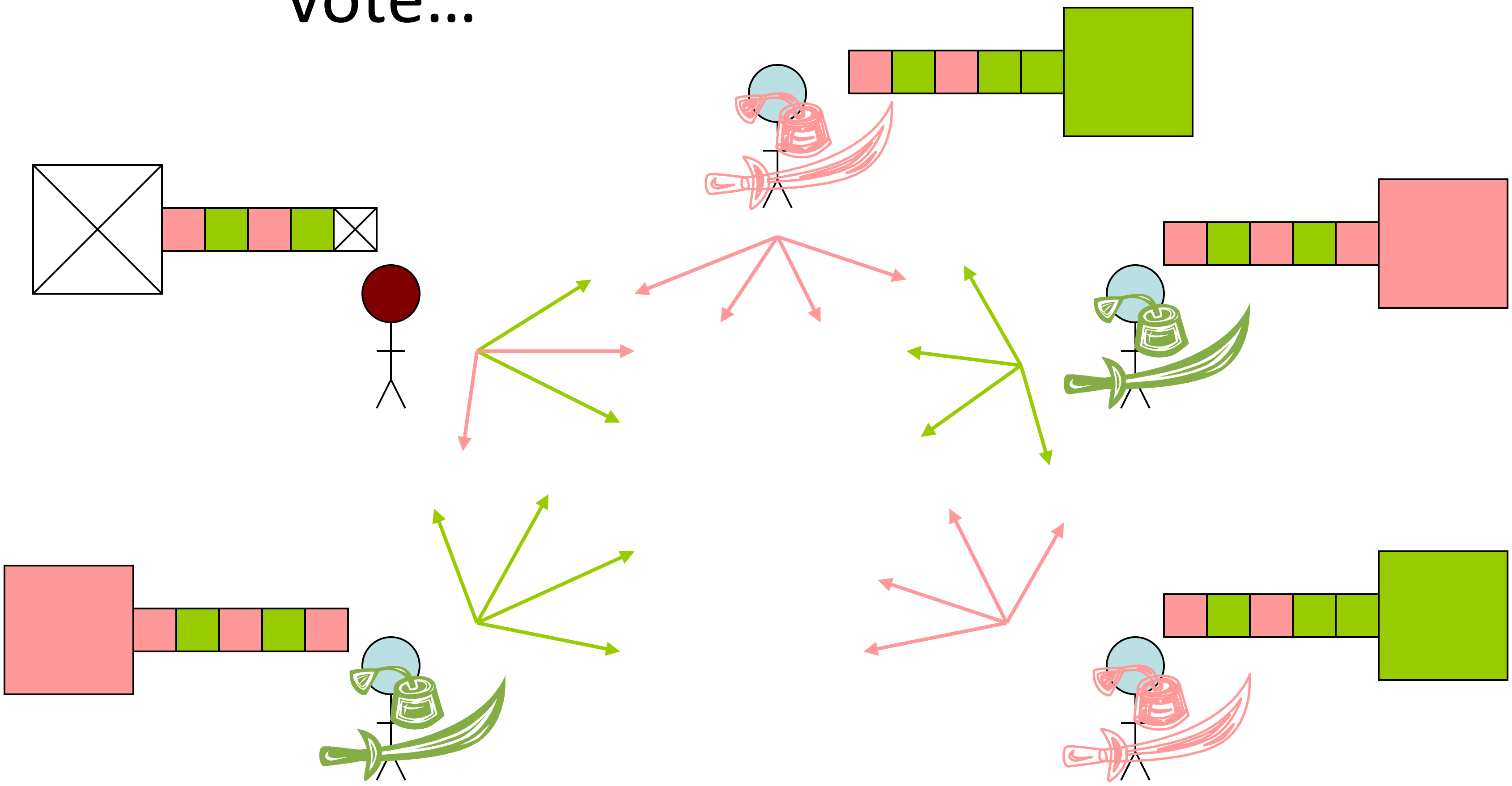
The Byzantine Generals

Each general collects his or her votes…



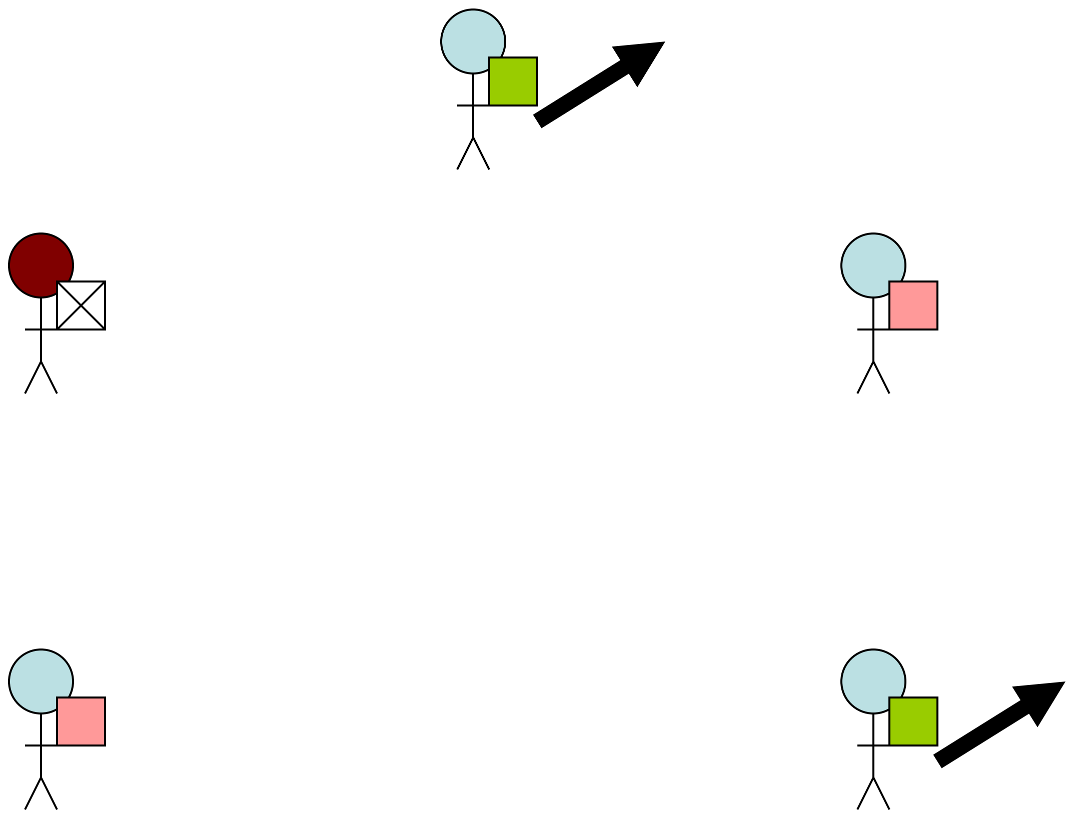
The Byzantine Generals

Assume each general takes the majority vote…



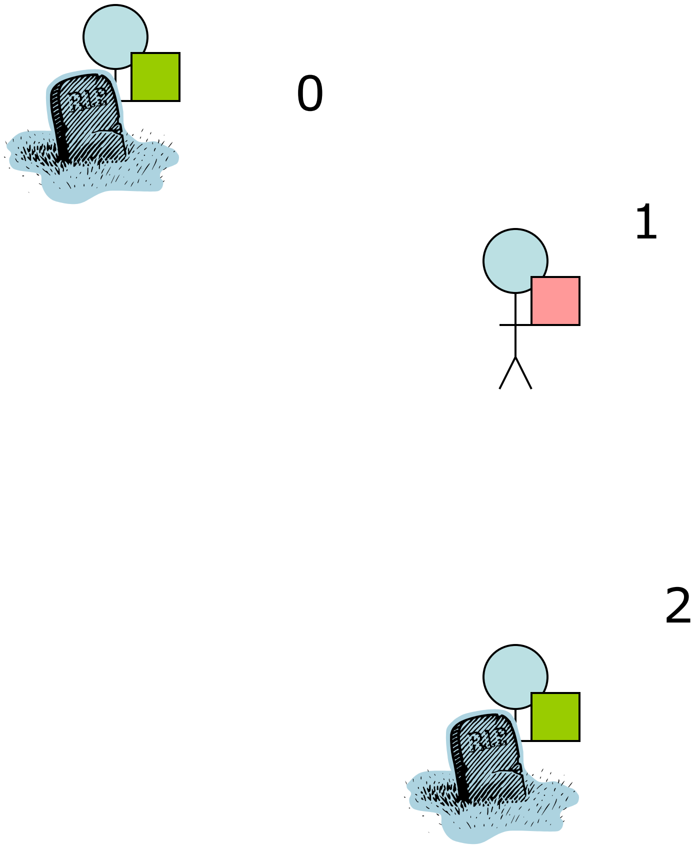
The Byzantine Generals

The generals now move based upon their 'agreed' orders…



The Byzantine Generals

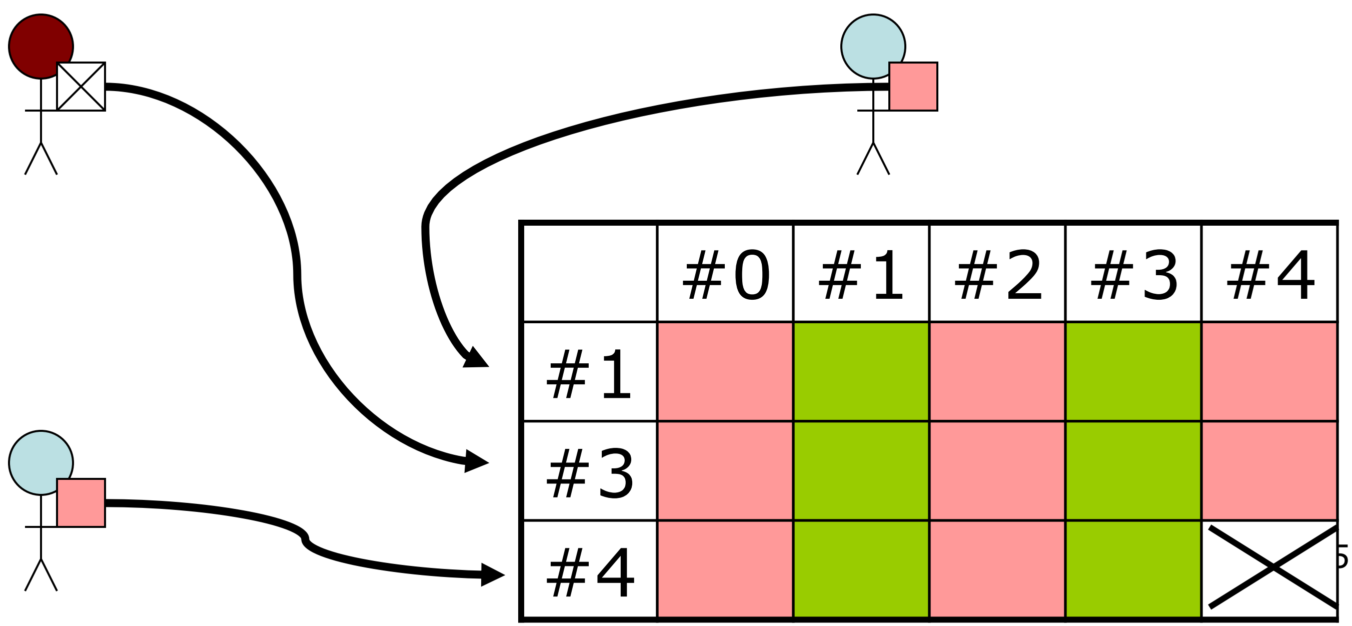
Since less than half of the military attacked, the military attack failed…



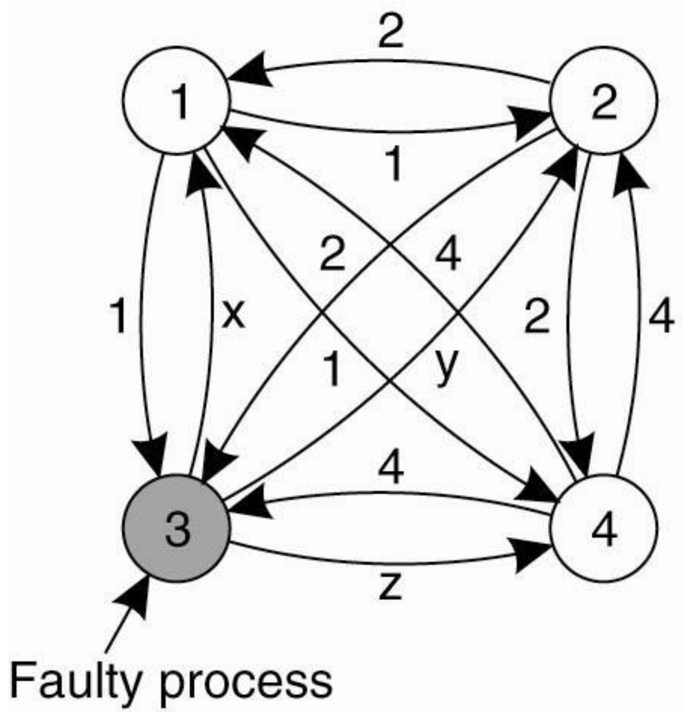
The Byzantine Generals

What’s more troubling is that:

the remaining loyal nodes do not know which node(s) among them are disloyal.



Example: Byzantine Agreement problem for four processes



(a)

- Three non-faulty and one faulty process. (a) Each process sends their value to the others. Process 3 lies, giving different values x, y,z to different processes.

Byzantine Agreement problem-2

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)

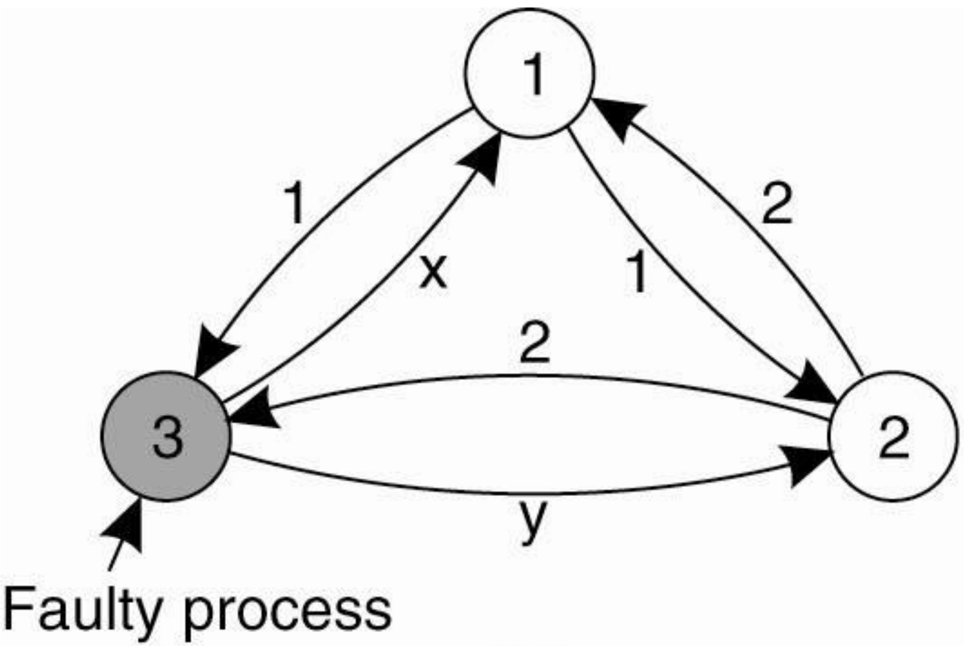
(c)

- (b) The vectors that each process assembles based on (a).
  (c) The vectors that each process receives in step 3. Process 3 sends different vectors to different processes.

Byzantine Agreement problem-3

- Three processes can agree on the values received from 1, 2, and 4. So that malicious process 3 value is irrelevant…
- If N=3 and k=1, that is only two non-faulty process this will not work!
- Lamport proved that, with 2k+1 nonfaulty processes, the system will survive k faulty processes, which makes a total of 3k+1 process…

BAP with two correct on faulty processes..



(a)

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)

Client Server Communication

- TCP-Transport Control Protocol as a connection oriented end-to-end communication protocol is a reliable protocol.
- But it does not prevent connection crash failures, which require searching for new connections…

RPC Semantics in the Presence of Failures

- Five different classes of failures that can occur in RPC systems:
  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. This can be handled with principles such as:-
     - At least once
     - At most once
     - The preferred principle is exactly once – not possible
  4. The reply message from the server to the client is lost.
  5. The client crashes after sending a request.
- Each case needs to be resolved properly to mask the failures

Failure Examples

- UDP service

  -has omission failures because it occasionally looses messages

- does not have value failures because it does not transmit corrupt messages.

  - UDP uses checksums to mask the value failures of the underlying IP by converting them to omission failures

Reliable Multicasting

- Use negative acknowledgement, known as scalable reliable multicasting-SRM
- Non-hierarchical and hierarchical solutions are possible
- Atomic multicasting requires all the replicas reaching agreement on the success or failure of multicast. This is known as distributed commit: two-phase or three-phase commit protocols can be used.

Recovery from a failure

- When and how the state of a distributed system be recorded and recovered to by means of check-pointing and logging.
- To be able to recover to a stable state, it is important that the state is safely stored..

Stable Storage

- Stable storage is an example of group masking at the disk block level
- Designed to ensure permanent data is recoverable after a system failure during a disk write operation or after a disk block has been damaged