Fault Tolerance

Fault Tolerance

- Fault tolerance is the ability of a distributed system to provide its services even in the presence of faults.
- A distributed system should be able to recover automatically from partial failures without seriously affecting availability and performance.

Fault Tolerance: Basic Concepts (1)

- Being fault tolerant is strongly related to what are called dependable systems Dependability implies the following:
  - Availability
  - Reliability
  - Safety
  - Maintainability
- **In-class Exercise**: How is is availability different from reliability? How about a system that goes down for 1 millisecond every hour? How about a system that never goes down but has to be shut down two weeks every year?

Fault Tolerance: Basic Concepts (2)

- A system is said to fail when it cannot meet its promises. For example, if it cannot provide a service (or only provide it partially).
- An error is a part of the system's state that may lead to failure. The cause of an error is called a fault.
- Types of faults:
  - Transient: Occurs once and then disappears. If a operation is repeated, the fault goes away.
  - Intermittent: Occurs, then vanishes of its own accord, then reappears and so on.
  - Permanent: Continues to exist until the faulty component is replaced.
- **In-class Exercise**: Give examples of each type of fault for your project!

Failure Models

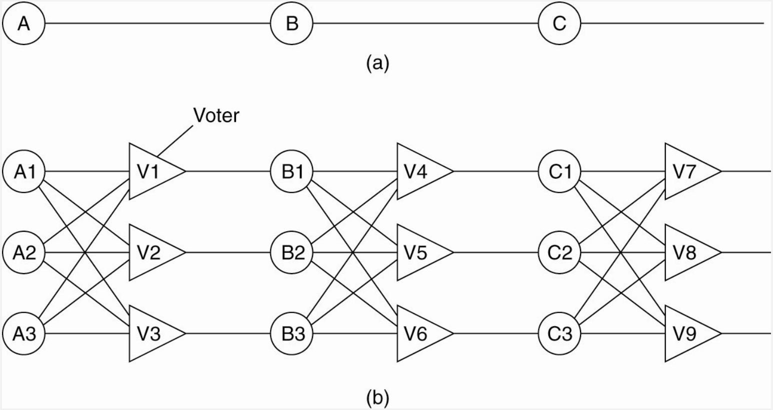
Type of failure	Description
Crash failure	A server halts, but is working correctly until it halts
Omission failure	A server fails to respond to incoming requests
Receive omission	A server fails to receive incoming messages
Send omission	A server fails to send messages
Timing failure	A server's response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

- Fail-stop versus fail-silent.
- Byzantine Failures:
  - Arbitrary failures where a server is producing output that it should never have produced, but which cannot be detected as being incorrect.
  - A faulty server may even be working with other servers to produce intentionally wrong answers!

Failure Masking by Redundancy

- Information Redundancy. For example, adding extra bits (like in Hamming Codes, see the book Coding and Information Theory) to allow recovery from garbled bits.
- Time Redundancy. Repeat actions if need be.
- Physical Redundancy. Extra equipment or processes are added to make the system tolerate loss of some components.

Failure Masking by Physical Redundancy

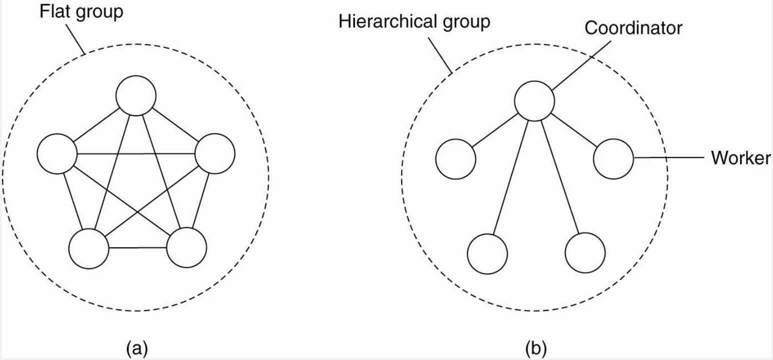


Process Resilience

Achieved by replicating processes into groups.

- How to design fault-tolerant groups?
- How to reach an agreement within a group when some members cannot be trusted to give correct answers?

Flat Groups Versus Hierarchical Groups



Failure Masking Via Replication

- *Primary-backup protocol*. A primary coordinates all write operations. If it fails, then the others hold an election to replace the primary
- *Replicated-write protocols*. Active replication as well as quorum based protocols. Corresponds to a flat group
- A system is said to be *k* fault tolerant if it can survive faults in *k* components and still meet its specifications.
  - For *fail-silent* components, $k + 1$ are enough to be *k* fault tolerant.
  - For *Byzantine failures*, at least $2k + 1$ extra components are needed to achieve *k* fault tolerance.
  - Requires atomic multicasting: all requests arrive at all servers in same order. This can be relaxed to just for write operations.

Agreement in Faulty Systems (1)

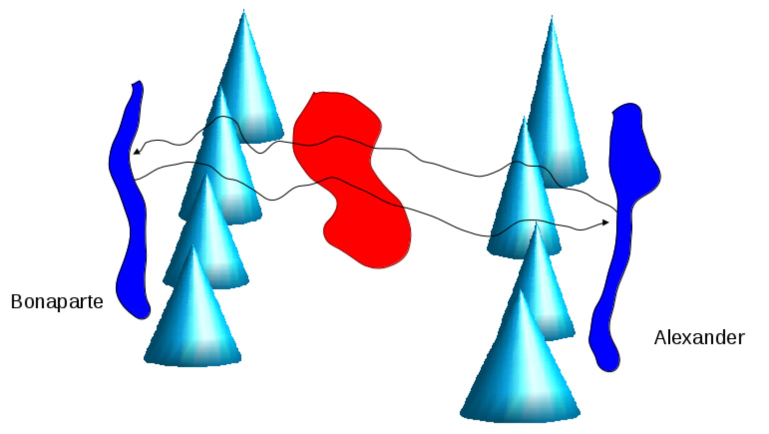
The general goal of distributed agreement algorithms is to have all the non-faulty processes reach agree consensus on some issue, and to establish that consensus within a finite number of steps. Possible cases:

- Synchronous versus asynchronous systems
- Communication delay is bounded or not
- Message delivery is ordered or not
- Message transmission is done through unicasting or multicasting

Agreement in Faulty Systems (2)

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

Agreement in Faulty Systems (3)



- Two Army Problem
- Non-faulty generals with unreliable communication.

Agreement in Faulty Systems (3)

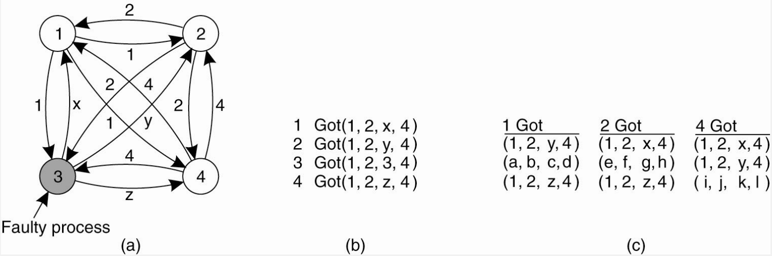
Byzantine Generals Problem (Lamport)

- Red army in the valley, n blue generals each with their own army surrounding them.
- Communication is pairwise, instantaneous and perfect.
- However m of the blue generals are traitors (faulty processes) and are actively trying to prevent the loyal generals from reaching agreement. The generals know the value m.

Goal: The generals need to exchange their troop strengths. At the end of the algorithm, each general has a vector of length n. If ith general is loyal, then the ith element has their correct troop strength otherwise it is undefined.
Conditions for a solution:

- All loyal generals decide upon the same plan of action
- A small number of traitors cannot cause the loyal generals to adopt a bad plan

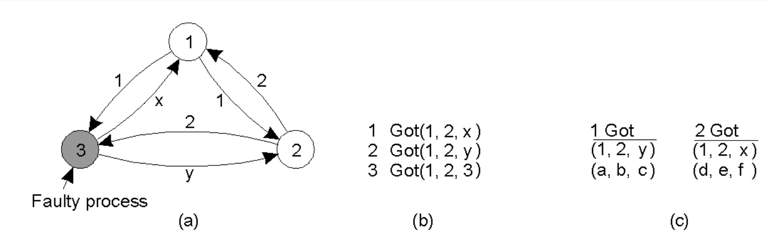
Byzantine Agreement Example (1)



The Byzantine generals problem for 3 loyal generals and 1 traitor:

- The generals announce their troop strengths (let's say, in units of 1 kilo soldiers)
- The vectors that each general assembles based on previous step
- The vectors that each general receives
- If a value has a majority, then we know it correctly, else it is unknown

Byzantine Agreement Example (2)



- The same as in previous slide, except now with 2 loyal generals and one traitor.
- For m faulty processes, we need a total of $3m+1$ processes to reach agreement.

Reliable Client-Server Communication

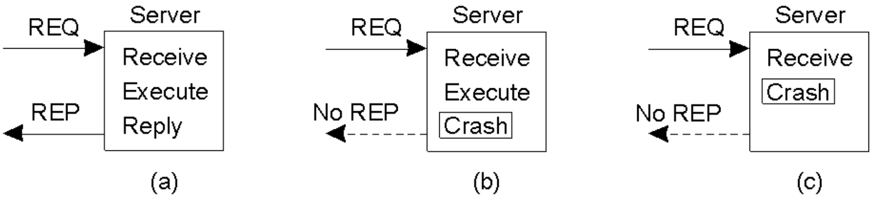
RMI semantics in the presence of failures.

- The client is unable to locate the server.
- The request message from the client to the server is lost.
- The server crashes after receiving a request.
- The reply message from the server to the client is lost.
- The client crashes after sending a request.

RPC Semantics in the Presence of Failures

- RPC failures
  - Client cannot locate server: Raise an exception or send a signal to client leading to loss in transparency
  - Lost request messages: Start a timer when sending a request. If timer expires before a reply is received, send the request again. Server would need to detect duplicate requests
  - Server crashes: Server crashes before or after executing the request is indistinguishable from the client side...
- Possible RPC semantics
  - Exactly once semantics
  - At least once semantics
  - At most once semantics
  - Guarantee nothing semantics

Server Crash (1)



- A server in client-server communication
  (a) Normal case
  (b) Crash after execution
  (c) Crash before execution

Server Crash (2)

- Server: Prints text on receiving request from client and sends message to client after text is printed.
  - Send a completion message just before it actually tells the printer to do its work
  - Or after the text has been printed
- Client:
  - Never to reissue a request.
  - Always reissue a request.
  - Reissue a request only if it did not receive an acknowledgment of its request being delivered to the server.
  - Reissue a request only if it has not received an acknowledgment of its print request.

Server Crash (3)

- Three events that can happen at the server:
  - Send the completion message (M)
  - Print the text (P)
  - Crash (C)
- These events can occur in six different orderings:

Server Crash (4)

1. $M \to P \to C$: A crash occurs after sending the completion message and printing the text.
2. $M \to C(\to P)$: A crash happens after sending the completion message, but before the text could be printed.
3. $P \to M \to C$: A crash occurs after sending the completion message and printing the text.
4. $P \to C(\to M)$: The text printed, after which a crash occurs before the completion message could be sent.
5. $C(\to P \to M)$: A crash happens before the server could do anything.
6. $C(\to M \to P)$: A crash happens before the server could do anything.

Server Crash (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

- M: send the completion message
- P: print the text
- C: server crash

RPC Semantics in the Presence of Failures (2)

- Lost Reply Messages. Set a timer on client. If it expires without a reply, then send the request again. If requests are idempotent, then they can be repeated again without ill-effects
- Client Crashes. Creates orphans. An orphan is an active computation on the server for which there is no client waiting. How to deal with orphans:
    - Extermination. Client logs each request in a file before sending it. After a reboot the file is checked and the orphan is explicitly killed off. Expensive, cannot locate grand-orphans etc.
    - Reincarnation. Divide time into sequentially numbered epochs. When a client reboots, it broadcasts a message declaring a new epoch. This allows servers to terminate orphan computations.
    - Gentle Reincarnation. A server tries to locate the owner of orphans before killing the computation.
    - Expiration. Each RPC is given a quantum of time to finish its job. If it cannot finish, then it asks for another quantum. After a crash, a client need only wait for a quantum to make sure all orphans are gone.

Idempotent Operations

- An idempotent operation is one that can be repeated as often as necessary without any harm being done. E.g. reading a block from a file.
- In general, try to make RPC/RMI methods be idempotent if possible. If not, it can be dealt with in a couple of ways.
  - Use a sequence number with each request so server can detect duplicates. But now the server needs to keep state for each client....
  - Have a bit in the message to distinguish between original and duplicate transmission