Fault Tolerance

Amir H. Payberah
amir@sics.se

Amirkabir University of Technology
(Tehran Polytechnic)

AMIRKABIR UNIVERSITY
OF TECHNOLOGY

Based on slides by Maarten Van Steen

What is the problem?

Dependability

- A component provides a services to a clients.

Dependability

- A component provides a services to a clients.
- To provide services, a component may require the services from other components.

Dependability

- A component provides a services to a clients.
- To provide services, a component may require the services from other components.
- A component C depends on C^* if the correctness of C's behavior depends on the correctness of C^*'s behavior.

Dependability

- A component provides a services to a clients.
- To provide services, a component may require the services from other components.
- A component C depends on C^* if the correctness of C's behavior depends on the correctness of C^*'s behavior.
- Components are processes or channels.

Terminology - Subtle Differences

- Failure: when a component is not living up to its specifications, a failure occurs.
- Error: that part of a component's state that can lead to a failure.
- Fault: the cause of an error.

Terminology - What To Do About Faults

- Fault prevention: prevent the occurrence of a fault.
- Fault tolerance: build a component such that it can mask the presence of faults.
- Fault removal: reduce presence, number, seriousness of faults.
- Fault forecasting: estimate present number, future incidence, and consequences of faults.

Terminology - Failure Models

- Crash failures: component halts, but behaves correctly before halting.
- Omission failures: component fails to respond.
- Timing failures: output is correct, but lies outside a specified real-time interval.
- Response failures: output is incorrect, e.g., wrong value is produced.
- Arbitrary failures: component produces arbitrary output and be subject to arbitrary timing failures.

Crash Failures (1/2)

- Clients cannot 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?

Crash Failures (2/2)

- Assumptions we can make:
  - Fail-silent: the component exhibits omission or crash failures; clients cannot tell what went wrong.
  - Fail-stop: the component exhibits crash failures, but its failure can be detected.
  - Fail-safe: the component exhibits arbitrary, but they can't do any harm.

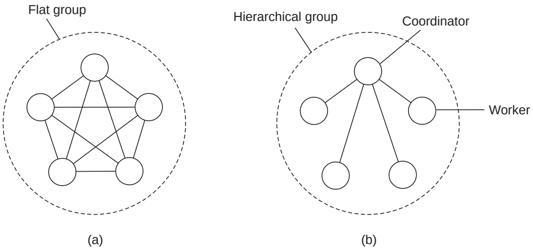
Process Resilience

Process Resilience (1/2)

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

Process Resilience (2/2)

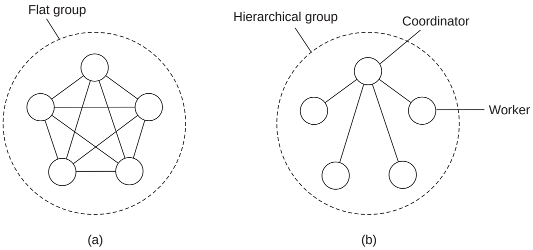
- Flat groups: good for fault tolerance as information exchange immediately occurs with all group members; however, may impose more overhead as control is completely distributed.



Process Resilience (2/2)

- Flat groups: good for fault tolerance as information exchange immediately occurs with all group members; however, may impose more overhead as control is completely distributed.

- Hierarchical groups: all communication through a single coordinator ⇒ not really fault tolerant and scalable, but relatively easy to implement.



Groups and Failure Masking (1/4)

- K-fault tolerant group: when a group can mask any k concurrent member failures (k is called degree of fault tolerance).

Groups and Failure Masking (1/4)

- K-fault tolerant group: when a group can mask any k concurrent member failures (k is called degree of fault tolerance).
- Assumption: all members are identical, and process all input in the same order.

Groups and Failure Masking (1/4)

- K-fault tolerant group: when a group can mask any k concurrent member failures (k is called degree of fault tolerance).
- Assumption: all members are identical, and process all input in the same order.
- How large does a k-fault tolerant group need to be?

Groups and Failure Masking (1/4)

- K-fault tolerant group: when a group can mask any k concurrent member failures (k is called degree of fault tolerance).
- Assumption: all members are identical, and process all input in the same order.
- How large does a k-fault tolerant group need to be?
  - In crash failure semantics $\Rightarrow$ a total of $k+1$ members are needed to survive k member failures.

Groups and Failure Masking (1/4)

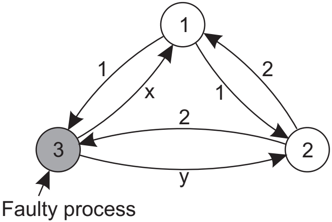
- K-fault tolerant group: when a group can mask any k concurrent member failures (k is called degree of fault tolerance).

- Assumption: all members are identical, and process all input in the same order.

- How large does a k-fault tolerant group need to be?
  - In crash failure semantics $\Rightarrow$ a total of $k+1$ members are needed to survive k member failures.
  - What about in arbitrary failure semantics? the group output defined by voting.

Groups and Failure Masking (2/4)

- (a) What they send to each other.
- (b) What each one got from the other.
- (c) What each one got in the second step.



(a)

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

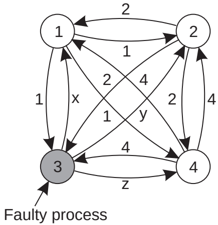
(b)

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

(c)

Groups and Failure Masking (3/4)

- (a) What they send to each other.
- (b) What each one got from the other.
- (c) What each one got in the second step.



(a)

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)

Groups and Failure Masking (4/4)

- In a system with K faulty processes, agreement can be achieved only if $2K + 1$ correctly functioning processes are present.

- Agreement is possible only if more than two-thirds of the processes are working properly: to achieve a majority vote among a group of nonfaulty processes.

Failure Detection

- We detect failures through timeout mechanisms.
- Setting timeouts properly is very difficult:
  - You cannot distinguish process failures from network failures.

Reliable Communication

Reliable Communication

- Client-Server communication
- Group communication

Client-Server Communication

Reliable Communication

- Concentrated on process resilience (by means of process groups).
- What about reliable communication channels?

Reliable RPC (1/6)

- RPC communication - 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

Reliable RPC (2/6)

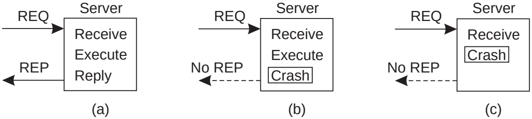
- Problem: client cannot locate server.
- Solution: report back to client.

Reliable RPC (3/6)

- Problem: client request is lost.
- Solution: resend message.

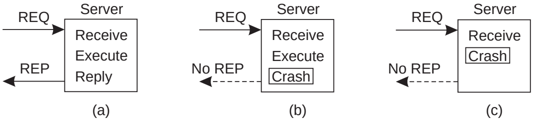
Reliable RPC (4/6)

- Problem: server crashes.
- It is hard as you don't know what it had already done.



Reliable RPC (4/6)

- Problem: server crashes.
- It is hard as you don't know what it had already done.



- We need to decide on what we expect from the server:
  - At-least-once-semantics: the server guarantees it will carry out an operation at least once, no matter what.
  - At-most-once-semantics: the server guarantees it will carry out an operation at most once.

Reliable RPC (5/6)

- Problem: server response is lost.
- Detecting lost replies can be hard, because it can also be that the server had crashed. You don't know whether the server has carried out the operation.
- Solution: none, except that you can try to make your operations idempotent: repeatable without any harm done if it happened to be carried out before.

Reliable RPC (6/6)

- Problem: client crashes.
- The server is doing work and holding resources for nothing (called doing an orphan computation).
- Solution:
  - Orphan is killed (or rolled back) by client when it reboots.
  - Broadcast new epoch number when recovering ⇒ servers kill orphans
  - Require computations to complete in a T time units. Old ones are simply removed.

Group Communication

Reliable Multicasting (1/2)

- We have a multicast channel c with two groups:
  - $SND(c)$: the sender group of processes that submit messages to channel c.
  - $RCV(c)$: the receiver group of processes that can receive messages from channel c.

Reliable Multicasting (1/2)

- We have a multicast channel c with two groups:
  - $SND(c)$: the sender group of processes that submit messages to channel c.
  - $RCV(c)$: the receiver group of processes that can receive messages from channel c.
- Simple reliability: if process $P \in RCV(c)$ at the time message m was submitted to c, and P does not leave $RCV(c)$, m should be delivered to P.

Reliable Multicasting (1/2)

- We have a multicast channel c with two groups:
  - $SND(c)$: the sender group of processes that submit messages to channel c.
  - $RCV(c)$: the receiver group of processes that can receive messages from channel c.
- Simple reliability: if process $P \in RCV(c)$ at the time message m was submitted to c, and P does not leave $RCV(c)$, m should be delivered to P.
- Atomic multicast: how can we ensure that a message m submitted to channel c is delivered to process $P \in RCV(c)$ only if m is delivered to all members of $RCV(c)$.

Reliable Multicasting (2/2)

- Let the sender log messages submitted to channel c:
  - If P sends message m, m is stored in a history buffer.
  - Each receiver acknowledges the receipt of m, or requests retransmission at P when noticing message lost.
  - Sender P removes m from history buffer when everyone has acknowledged receipt.

Reliable Multicasting (2/2)

- Let the sender log messages submitted to channel c:
  - If P sends message m, m is stored in a history buffer.
  - Each receiver acknowledges the receipt of m, or requests retransmission at P when noticing message lost.
  - Sender P removes m from history buffer when everyone has acknowledged receipt.
- Why doesn't this scale?
  - If $RCV(c)$ is large, P will be swamped with feedback (ACKs and NACKs).
  - Sender P has to know all members of $RCV(c)$.

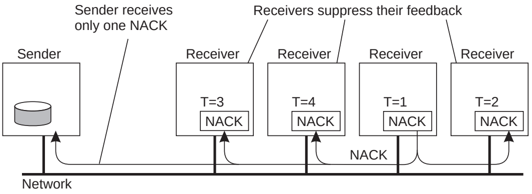
Scalable Reliable Multicasting

Scalable Reliable Multicasting

- Feedback suppression
- Hierarchical solutions

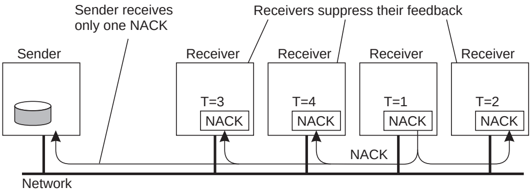
Feedback Suppression (1/2)

- Basic idea: let a process *P* suppress its own feedback when it notices another process *Q* is already asking for a retransmission.



Feedback Suppression (1/2)

- Basic idea: let a process P suppress its own feedback when it notices another process Q is already asking for a retransmission.
- Assumptions:
  - All receivers listen to a common feedback channel to which feedback messages are submitted.
  - Process P schedules its own feedback message randomly, and suppresses it when observing another feedback message.

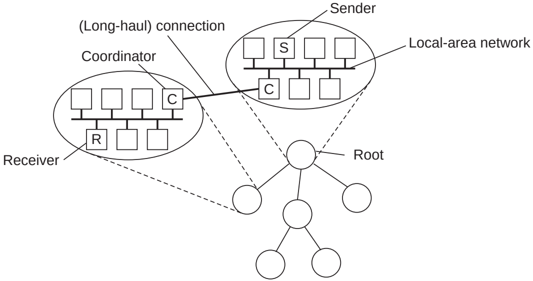


Feedback Suppression (2/2)

- Why is the random schedule so important? random schedule needed to ensure that only one feedback message is eventually sent.

Hierarchical Solutions (1/2)

- Basic idea: construct a hierarchical feedback channel in which all submitted messages are sent only to the root.
- Intermediate nodes aggregate feedback messages before passing them on.
- Intermediate nodes can easily be used for retransmission purposes.



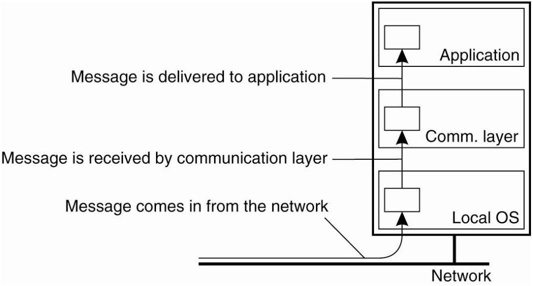
Hierarchical Solutions (2/2)

- What's the main problem with this solution? dynamically constructing the hierarchical feedback channel is the main problem.

Atomic Multicast

Receiving vs. Delivering

- The logical organization of a distributed system to distinguish between message receipt and message delivery.



Atomic Multicast

- A message is delivered only to the nonfaulty members of the current group.
- All members should agree on the current group membership: virtually synchronous multicast.
- We consider views $V \subseteq RCV(c) \cup SND(c)$.

Virtual Synchrony

- Suppose the message m is multicast at the time its sender has group view G.

Virtual Synchrony

- Suppose the message m is multicast at the time its sender has group view G.
- Assume that while the multicast is taking place, another process joins or leaves the group.

Virtual Synchrony

- Suppose the message m is multicast at the time its sender has group view G.
- Assume that while the multicast is taking place, another process joins or leaves the group.
  - The group membership change is announced to all processes in G: by multicasting a message vc.

Virtual Synchrony

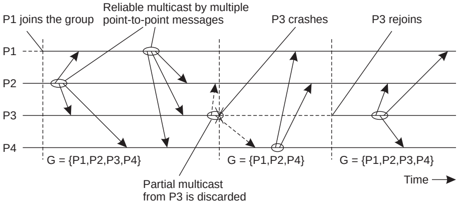
- Suppose the message m is multicast at the time its sender has group view G.
- Assume that while the multicast is taking place, another process joins or leaves the group.
  - The group membership change is announced to all processes in G: by multicasting a message vc.
- We now have two multicast messages simultaneously in transit: m and vc.

Virtual Synchrony

- Suppose the message m is multicast at the time its sender has group view G.
- Assume that while the multicast is taking place, another process joins or leaves the group.
  - The group membership change is announced to all processes in G: by multicasting a message vc.
- We now have two multicast messages simultaneously in transit: m and vc.
- We need to guarantee is that m is either delivered to all processes in G before each one of them is delivered message vc, or m is not delivered at all.

Virtual Synchrony - Implementation (1/3)

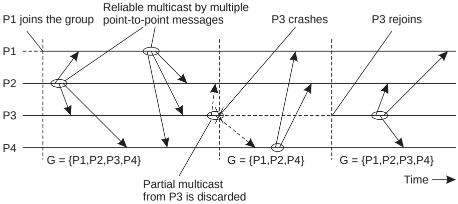
- How to guarantee that all messages sent to view G are delivered to all nonfaulty processes in G before the next group membership change takes place.



Virtual Synchrony - Implementation (1/3)

- How to guarantee that all messages sent to view G are delivered to all nonfaulty processes in G before the next group membership change takes place.

- Make sure that each process in G has received all messages that were sent to G.

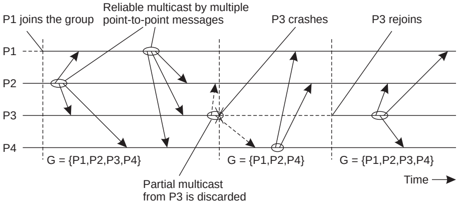


Virtual Synchrony - Implementation (1/3)

- How to guarantee that all messages sent to view G are delivered to all nonfaulty processes in G before the next group membership change takes place.

- Make sure that each process in G has received all messages that were sent to G.

- Because the sender of a message m to G may have failed before completing its multicast, there may be processes in G that will never receive m.

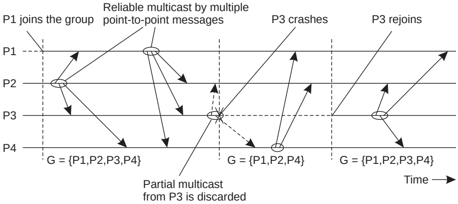


Virtual Synchrony - Implementation (1/3)

- How to guarantee that all messages sent to view G are delivered to all nonfaulty processes in G before the next group membership change takes place.

- Make sure that each process in G has received all messages that were sent to G.

- Because the sender of a message m to G may have failed before completing its multicast, there may be processes in G that will never receive m.
    - Because the sender has crashed, these processes should get m from somewhere else.

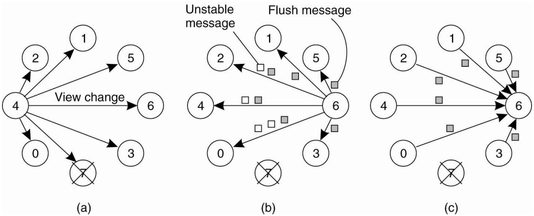


Virtual Synchrony - Implementation (2/3)

- Solution: let every process in G keep m until it knows for sure that all members in G have received it.
- If m has been received by all members in G, m is said to be stable.
- Only stable messages are allowed to be delivered.

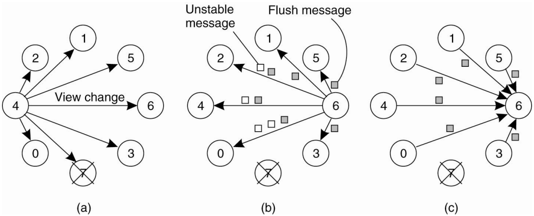
Virtual Synchrony - Implementation (3/3)

- (a) 4 notices that 7 has crashed and sends a view change.



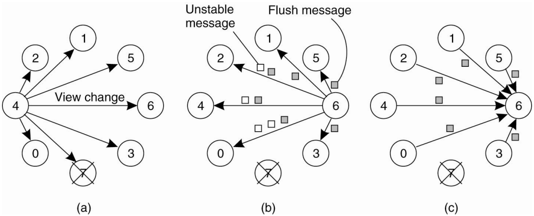
Virtual Synchrony - Implementation (3/3)

- (a) 4 notices that 7 has crashed and sends a view change.
- (b) 6 sends out all its unstable messages, followed by a flush message.



Virtual Synchrony - Implementation (3/3)

- (a) 4 notices that 7 has crashed and sends a view change.
- (b) 6 sends out all its unstable messages, followed by a flush message.
- (c) 6 installs the new view when it has received a flush message from everyone else.



Summary

Summary

- Failure
- Failure models: crash, omission, timing, response, arbitrary
- Crash failure: fail-silent, fail-stop, fail-safe
- Process resilience: flat group, hierarchical group
- K-fault tolerant group: more than two-thirds of the processes work properly
- Reliable communication: client-server, group
- Scalable reliable multicast: feedback suppression, hierarchical
- Atomic broadcast: virtual synchrony

Reading

- Chapter 9 of the Distributed Systems: Principles and Paradigms.

Questions?