Fault Tolerance

527950-1
Fall 2019
12/12/2019
Kyoung Shin Park
Applied Computer Engineering
Dankook University

Overview

- Fault Tolerance Basic Concepts
- Process Resilience
- Reliable Communications
- Distributed Commit
- Recovery

Failure

- In distributed systems,
  - **Partial failure may happen when one component fails**
  - This failure may affect some of the components
  - While at the same time, leaves other components totally unaffected
- In non-distributed systems (e.g., single machine system),
  - A failure often affects all components
  - This may easily bring down the entire system

Fault Tolerance and Recovery

- An important goal in distributed systems is,
  - Automatically recover from partial failures without seriously affecting overall performance
  - If a failure occurs, the system should continue to operate, while repair is being made.
  - In other words, it should **tolerate faults** and continue to operate

Dependability

- **Fault tolerance** is strongly related to what are called dependable systems
- **Dependability** implies the following:
  1. **Availability:**
     - A system is ready to be used immediately – System is up and running at any given moment (%)
  2. **Reliability**
     - A system can run continuously without failure – System continues to function for a long period of time (time)
  3. **Safety**
     - If a system fails, nothing catastrophic will happen - Low probability of catastrophes (effect)
  4. **Maintainability**
     - When a system fails, it can be repaired easily and quickly (and, sometimes, without its users noticing the failure)

What is Failure?

- A system is said to **"fail"** when it *cannot meet* its promises (specifications).
- A failure is brought about by the *existence* of **"errors"** in the system.
- The *cause* of an error is a **"fault".**

Types of Faults

- Faults can be
  - **Transient**
    - Occur once and then disappear
  - **Intermittent**
    - Occur, then vanish, then reappear occurs, but: follows no real pattern (worst kind)
  - **Permanent**
    - Continues to exist. Once it occurs, only the replacement/repair of a faulty component will allow the distributed system to function normally.

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

- Note: Crash failures are the least severe; arbitrary failures are the worst

Fault Handling Approaches

- **Fault prevention**
  - Prevent the occurrence of a fault
- **Fault tolerance**
  - **A system can provide its services in the presence of faults (i.e., mask the presence of faults)**
- **Fault removal**
  - Reduce the presence, number, seriousness of faults
- **Fault forecasting**
  - Estimate the present number, future incidence, and the consequences of faults

Approaches to Fault Tolerance

- **No fault tolerance without redundancy**
- **Failure masking by redundancy**
  - Use redundancy to mask a failure, i.e., hide the occurrence of a fault

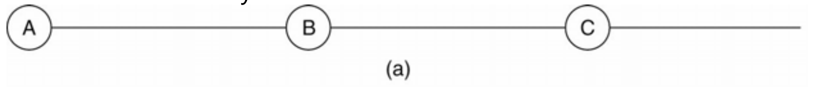
Failure Masking by Redundancy

- **Strategy**: **hide the occurrence of failure** from other processes using ***redundancy***.
- **3 types of failure masking by redundancy**
  - **Information redundancy**
    - E.g., FEC (Forward Error Correction): a code can be added to transmitted data to recover from packet error
  - **Time redundancy**
    - Performed again: retransmission in TCP/IP
    - Especially helpful for transient or intermittent faults.
  - **Physical redundancy**
    - Software (process replication), Hardware (biology, aircraft, sports, electronic circuits)
    - E.g., 747s have four engines but can fly on three

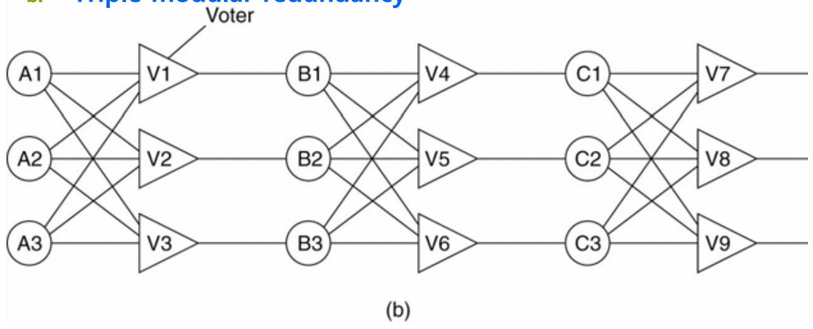
Physical Redundancy

- **Active replication** is a technique for achieving fault tolerance through physical redundancy.

a. No redundancy



b. **Triple modular redundancy**

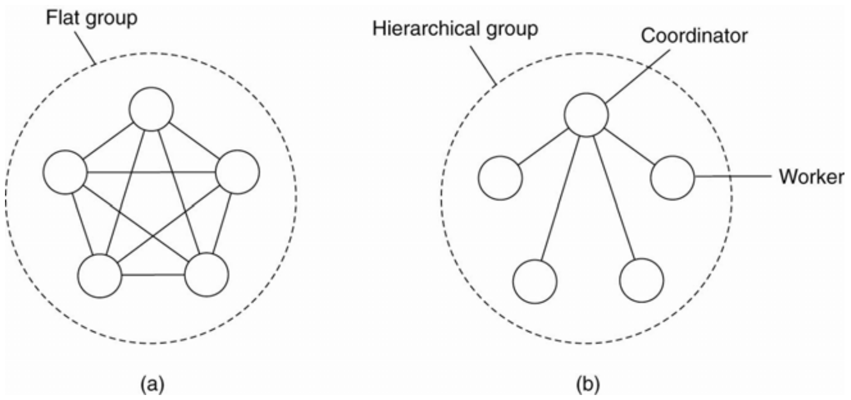


Process Resilience

- **Organizing replicated processes into a group**
- 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

- Group organization
  a. Communication in a **flat group**.
  b. Communication in a simple **hierarchical group**.



Flat Groups versus Hierarchical Groups

a. 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.
   - Good for fault tolerance as information exchange immediately occurs with all group members

b. 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.
   - Not really fault tolerant or scalable

Process Replication

- Replicate a process and group replicas in one group
- How many replicas do we create?
- A system is **k fault tolerant** if it can survive faults in k components and still meet its specifications.
  - For **crash failure model** (a faulty process halts, but is working correctly until it halts)

  ⇒ Need **k + 1** (k can fail and one will still be working)
  - For **arbitrary/Byzantine failure model** (a faulty process may produce arbitrary responses at arbitrary times) and **group output defined by voting** collected by the client

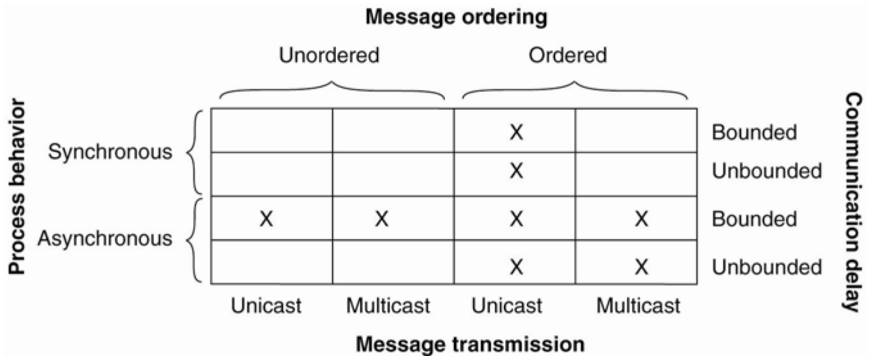
  ⇒ Need **2k + 1** (k can generate false replies and k + 1 will provide a majority vote)
  - Assume that **all members are identical** and process all input in **the same order**

Agreement in Faulty Systems

- Distributed agreement algorithms
  - **All non-faulty processes reach consensus on some issue, and to establish that consensus within a finite number of steps.**
- Possible cases
  1. Synchronous versus asynchronous systems
     - A system is synchronous if and only if the processes are known to operate in a lock-step mode.
  2. Communication delay is bounded or not.
     - Delay is bounded if and only if we know that every message is delivered with a globally and predetermined maximum time.
  3. Message delivery is ordered or not.
     - Messages from the same sender are delivered in the order that they were sent vs no such guarantees.
  4. Message transmission is done through unicasting or multicasting.

Agreement in Faulty Systems

- Circumstances under which distributed agreement can be reached.



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

Message ordering (columns: Unordered, Ordered); Message transmission (Unicast, Multicast).

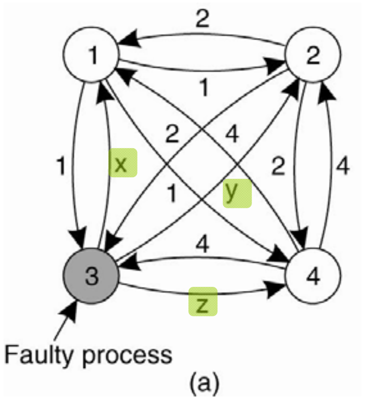
Byzantine Agreement Problem

- **Byzantine agreement problem** by Lamport et al. (1982)
  - A.k.a Byzantine generals problem
  - Assume that **processes are synchronous**, **messages are unicast** while **preserving ordering** and **communication delay is bounded.**
  - Proved that in a system with **k faulty processes**, agreement can be achieved only if **2k+ 1 correctly functioning processes** are present, for **a total of 3k + 1**.
  - Agreement is possible only if **more than two-thirds of the processes are working properly**.

⇒ **Need 3k + 1 members for k traitors (2k + 1 loyalists)** i.e., a majority vote among the group of loyalists, in the presence of k traitors

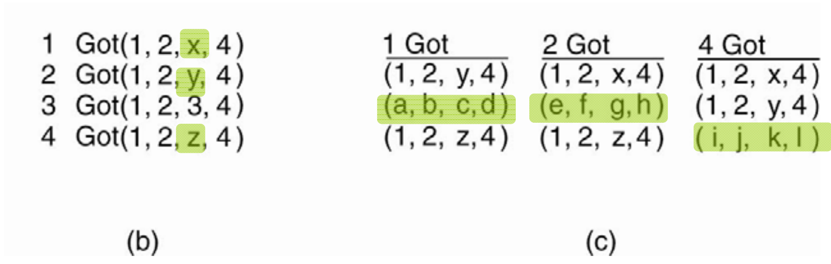
Byzantine Agreement Problem (1)

- **The Byzantine agreement problem** for three non-faulty and one faulty process.
  a. **Each process sends their value to the others.**



Byzantine Agreement Problem (1)

- **The Byzantine agreement problem** for three non-faulty and one faulty process.
  - b. **The vectors that each process assembles based on (a).**
  - c. **The vectors that each process receives in step 3.**



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

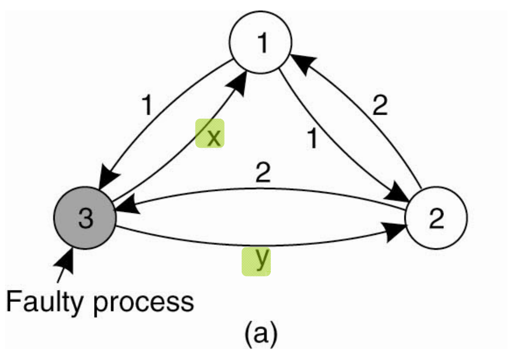
(b)

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)

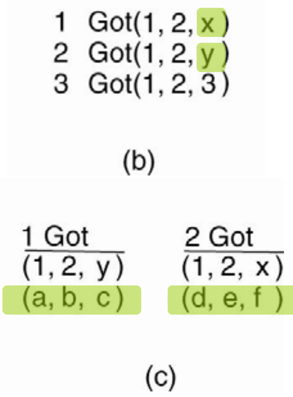
(c)

Byzantine Agreement Problem (2)

- The same as previous, except now with two correct process and one faulty process.
  - **Given three processes, if one fails, consensus is impossible.**



(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)

Agreement

- **Two generals problem**
  - **Perfect processes, faulty communication channels**
  - A.k.a. Two armies problem or Coordinated attack problem
  - Multiple acknowledgement problem
- **Byzantine generals problem**
  - **Faulty processes, perfect communication channels**
  - In essence, we are trying to reach a majority vote among the group of loyalists, in the presence of k traitors

  ⇒ **Need 3k + 1 members for k traitors (2k + 1 loyalists) (Lamport, 1982)**

Failure Detection

- Detecting (process) failures
  - Processes actively send "are you alive?" (**active pinging**) messages to each other (for which they obviously expect an answer)
  - Processes **passively wait** until messages come in from different processes.
- Detect failures through **timeout** mechanisms
  - Setting timeouts properly is difficult and application dependent
  - False positive: cannot distinguish process failures from network failures
  - Solutions?

Communication Failure

- Until now, we discussed only faulty processes
- However, communication channel may exhibit
  - Crash (system halts)
  - Omission (incoming request ignored)
  - Timing (responding too soon or too late)
  - Response (getting the order wrong)
  - Arbitrary/Byzantine (indeterminate, unpredictable) failures
- We need to mask these failures
- Arbitrary communication failures
  - Duplicate messages – why?

Reliable Communication

- Error detection
  - Framing of packets and using checksum (e.g., CRC) to allow for bit error detection
  - Use of frame numbering to detect which packet was lost
- Error correction
  - Add redundancy bits so that corrupted packets can be automatically corrected
  - Request retransmission of lost packets

Reliable RPC

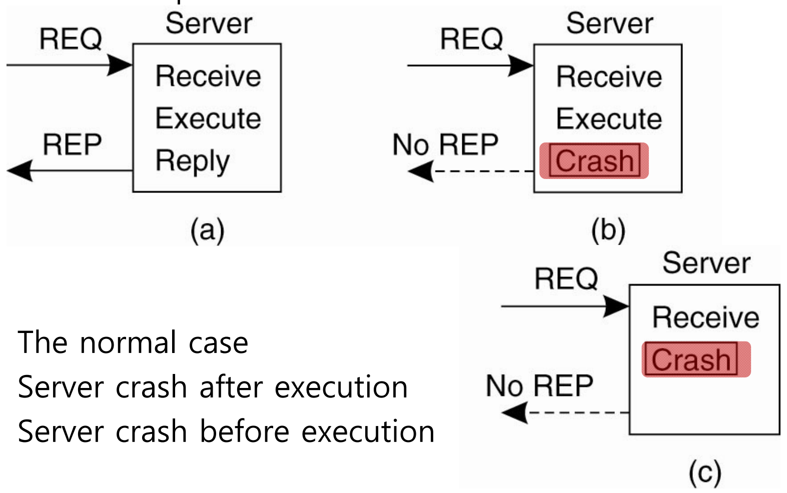
- RPC hides communication by making remote procedure calls look just like local ones
  - When errors occur it is not always easy to mask the difference between local and remote calls
- Five classes of failures can occur in RPC
  1. The client **cannot locate** the server, so no request can be sent.
  2. The **client's request** to the server is **lost**, so no response is returned by the server to the waiting client.
  3. The **server crashes** after receiving a request, and the service request is left acknowledged, but undone.
  4. The **server's reply** is **lost** on its way to the client, so the service has completed, but the results never arrive at the client.
  5. The **client crashes** after sending its request, and the server sends a reply to a newly-restarted client that may not be expecting it.

RPC Failures

1. **Client cannot locate server**
   - All servers might be down
   - Server is upgraded unknowingly to the client
   - Deal with the failure – have the error raise exceptions (e.g., Java) to solve the problem
2. **Client request is lost**
   - OS or client stub starts a timer after sending request
   - If timer expires before reply comes back, retransmit request
   - After so many requests are lost, the client gives up
     - Difference between first case?
3. **Server crashes**
4. **Server response is lost**
5. **Client crashes**

Server Crashes

- Client cannot tell if server crash occurred before or after the request is carried out.



a. The normal case
b. Server crash after execution
c. Server crash before execution

Server Crashes Semantics

- Server crashes are dealt with by implementing one of three possible philosophies
  - *At least once semantics*
    - A guarantee is given that the RPC occurred at least once, but (also) possibly more that once.
  - *At most once semantics*
    - A guarantee is given that the RPC occurred at most once, but possibly not at all.
  - *No semantics*
    - Nothing is guaranteed, and client and servers take their chances!
- It has proved difficult to provide *exactly once semantics*.

Printing Server Example

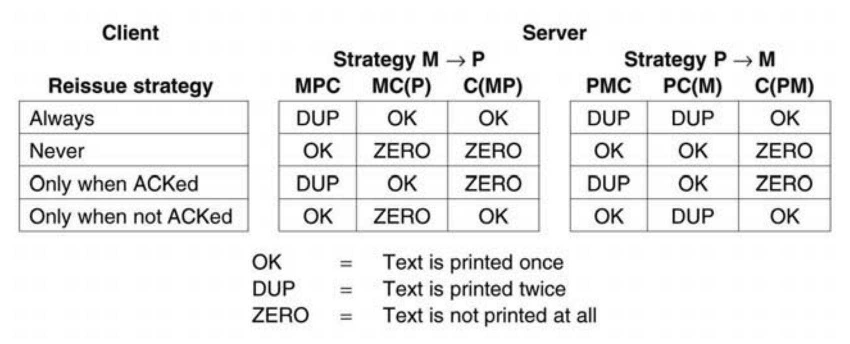
- Printing Server
  - Remote operation **prints** some text at server
  - Server sends **completion message** after text is printed
  - Client issues **request** and receives **acknowledgement**
- Assumption
  - Server crashes and subsequently recovers
  - Server announces to all clients that it just crashed but it is up and running again
  - Problem: Client doesn't know whether printing was actually carried out
- Three events that can happen at the server
  - **Send the completion message (M),**
  - **Print the text (P),**
  - **Crash (C).**

Printing Server Example

- These events can occur in six different orderings:
  1. **MPC** M →P →C: A crash occurs after sending the completion message and printing the text.
  2. **MC(P)** M →C (→P): A crash happens after sending the completion message, but before the text could be printed.
  3. **PMC** P →M →C: A crash occurs after sending the completion message and printing the text.
  4. **PC(M)** P→C(→M): The text printed, after which a crash occurs before the completion message could be sent.
  5. **C(PM)** C (→P →M): A crash happens before the server could do anything.
  6. **C(MP)** C (→M →P): A crash happens before the server could do anything.

Printing Server Example

- Different combinations of client and server



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

OK	=	Text is printed once
DUP	=	Text is printed twice
ZERO	=	Text is not printed at all

Server Response is Lost

- Lost replies
  - Detection is hard – only way is request timeout
  - It can also be that request is lost or the server had crashed.
  - Don't know whether the server has carried out the operation
- Examples:
  - File read(idempotent) vs. bank transfer(nonidempotent)
- Solutions
  - Try to make your operations **idempotent** - **repeatable without any harm done if it happened to be carried out before**
  - *Nonidempotent* requests are a little harder to deal with.
    - A common solution is to employ ***unique sequence numbers***
    - Another technique is the inclusion of ***additional bits*** in a retransmission to identify it as such to the server.

Client Crashes

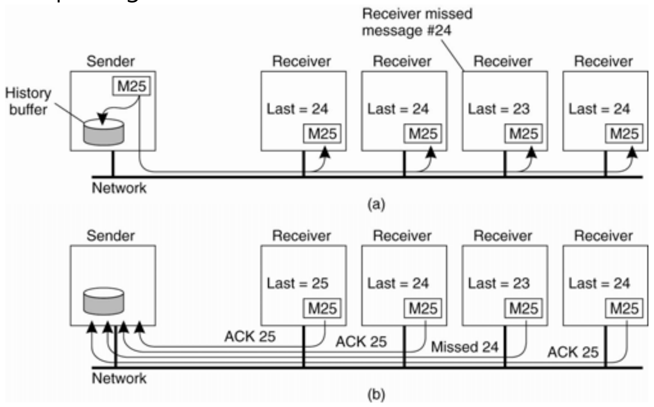
- Client crashes
  - When a client crashes, and when an 'old' reply arrives, such a reply is known as an ***orphan***.
- Orphan solutions
  - ***Extermination***
    - The client explicitly kills off the orphan if it is received
  - ***Reincarnation***
    - When a client reboots, it broadcasts a new epoch number. When server receives the broadcast, it kills the computations that were running on behalf of the client
  - ***Expiration***
    - Each RPC is associated with an expiration time T.
    - The call is aborted when the expiration time is reached
    - If RPC cannot finish within T, the client must asks for another quantum
    - If after a crash the client waits a time T before rebooting, all orphans are sure to be gone

Reliable Multicast

- Unicast vs. Multicast vs. Broadcast?
- Reliable unicast
  - Transport layers (e.g., TCP) offer reliable point-to-point communication
- Reliable multicast
  - **Guarantee that all messages are delivered to all members in a process group**
  - Multiple reliable point-to-point communications
    - Scalability?
  - Basic reliable-multicasting

Basic Reliable-Multicasting Schemes

- A simple solution to reliable multicasting when all receivers are known and are assumed not to fail.
  a. Message transmission – note that the 3rd receiver is expecting 24
  b. Reporting feedback – the 3rd receiver informs the sender

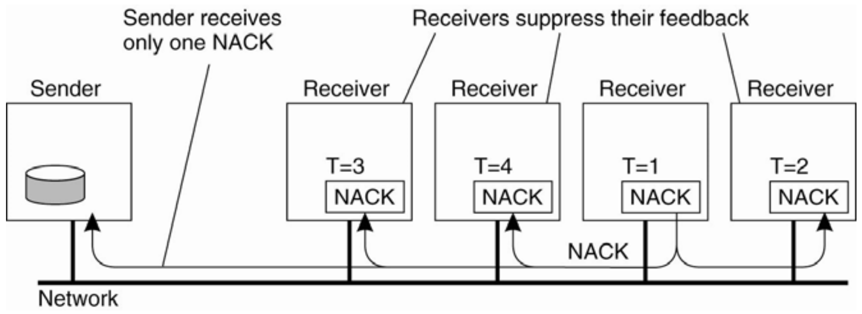


Reliable Multicasting Scalability

- Single sender, N receivers
  - Sender must accept at least N **ACK**s
  - If N is large, sender may be swamped with feedback messages – **feedback implosion**
- Using only **NACK**s
  - Receiver does not send ACK for message reception
  - Receiver only sends NACK for missing message
  - Better scalability (since ACKs are not sent)
  - Problems?
- One step further: Scalable reliable multicasting

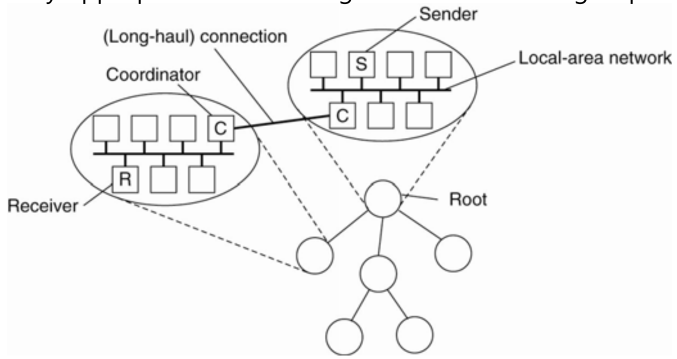
Nonhierarchical Feedback Control

- Several receivers have scheduled a request for retransmission, but the first retransmission request leads to the **suppression** of other **feedbacks**.



Hierarchical Feedback Control

- Hierarchical reliable multicasting
  - The essence is that it supports the creation of **very large groups**
  - Each local coordinator forwards the message to its children and later handles retransmission requests.
  - A local coordinator handles retransmission requests *locally*, using any appropriate multicasting method for small groups.



Reliable Multicasting Scalability

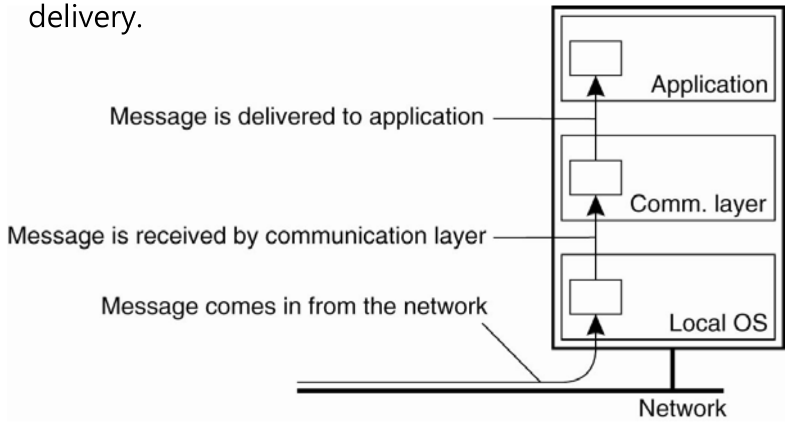
- Simple reliability
  - No messages lost
- In the presence of process failures
  - What does reliable delivery mean in the presence of process failures?
  - If a process fails it cannot receive the message
- **Atomic Multicast**
  - Atomic multicasting ensures that non-faulty processes maintain a consistent view of the database, and forces reconciliation when a replica recovers and rejoins the group.
  - **Virtually synchronous**
    - Guarantee that message is delivered to either **all non-faulty processes** or **none** at all
  - **Message ordering**
    - All messages are delivered **in the same order** to all processes

Virtual Synchrony

- Model for **group management** and **group communication**
  - A process can join or leave a group
  - A process can send a message to a group
    - Ordering requirements defined by programmer
- **Atomic multicast**
  - “A message is either delivered to **all processes** in the group or to **none**”

Virtual Synchrony System Model

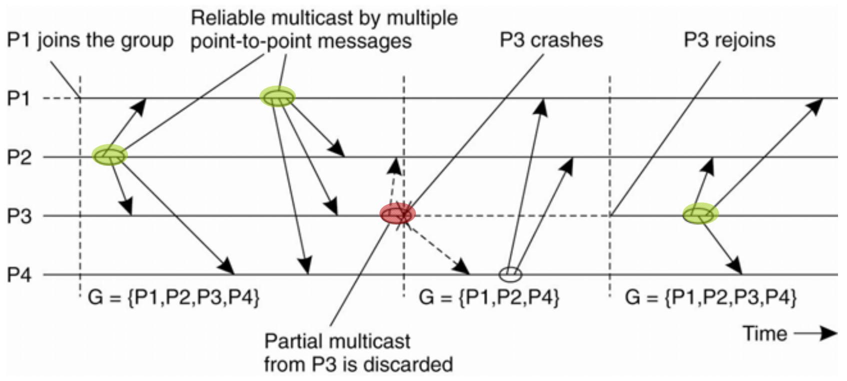
- The logical organization of a distributed system to distinguish between message receipt and message delivery.



Application
Message is delivered to application
Comm. layer
Message is received by communication layer
Message comes in from the network
Local OS
Network

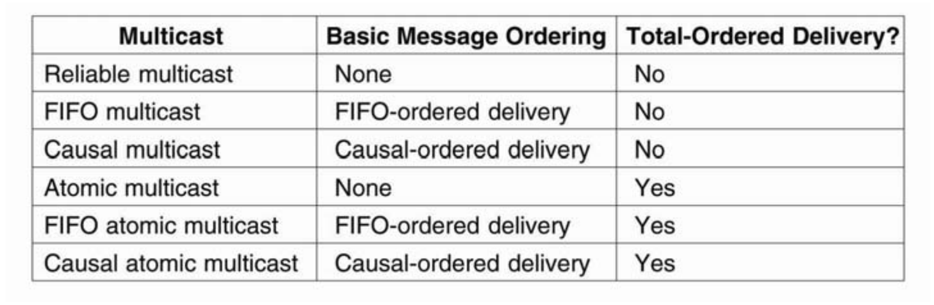
Virtual Synchrony

- The principle of virtual synchronous multicast



Implementing Virtual Synchrony

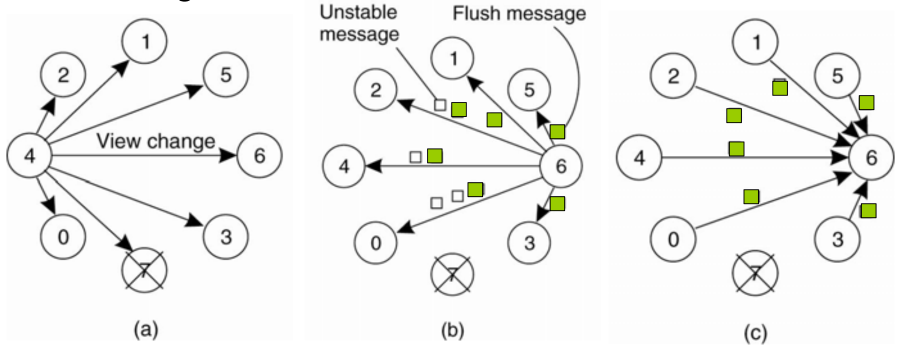
- Six different versions of virtually synchronous reliable multicasting.



Multicast	Basic Message Ordering	Total-Ordered Delivery?
Reliable multicast	None	No
FIFO multicast	FIFO-ordered delivery	No
Causal multicast	Causal-ordered delivery	No
Atomic multicast	None	Yes
FIFO atomic multicast	FIFO-ordered delivery	Yes
Causal atomic multicast	Causal-ordered delivery	Yes

Implementing Virtual Synchrony

a. P4 notices that P7 has crashed and sends a **view change**
b. P6 sends out all its **unstable messages**, followed by a **flush message**
c. P6 installs the **new view** when it has received a flush message from everyone else

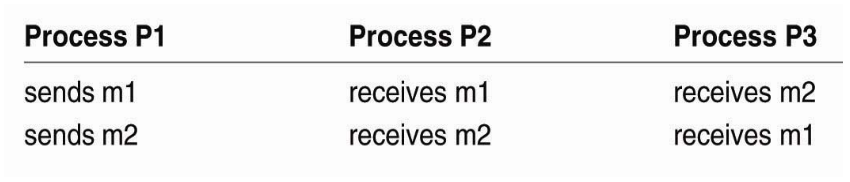


Message Ordering

- Virtually synchronous behavior is independent from the ordering of message delivery.
  - The only issue is that messages are delivered to an agreed upon group of receivers.
- Four different orderings
  1. **Unordered multicasts**
     - No guarantees are given concerning the order in which received messages are delivered by different processes
  2. **FIFO-ordered multicasts**
     - The communication layer is forced to deliver incoming messages from the same process in the same order as they have been sent
  3. **Causally-ordered multicasts**
     - Delivers messages so that potential causality between different messages is preserved
  4. **Totally-ordered multicasts**
     - It is required additionally that when messages are delivered, they are delivered in the same order to all group members.

Un-Ordered Multicast

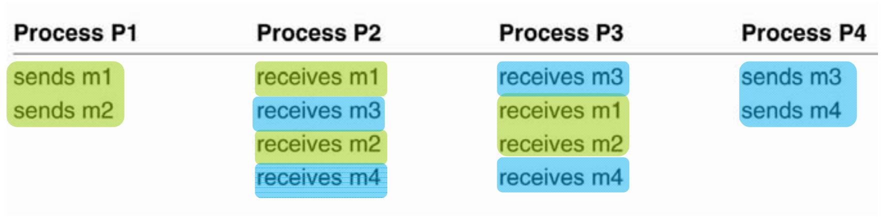
- Three communicating processes in the same group.
- The ordering of events per process is shown along the vertical axis.



Process P1	Process P2	Process P3
sends m1	receives m1	receives m2
sends m2	receives m2	receives m1

FIFO-Ordered Multicast

- Four processes in the same group with two different senders, and a possible delivery order of messages under FIFO-ordered multicasting



Process P1	Process P2	Process P3	Process P4
sends m1	receives m1	receives m3	sends m3
sends m2	receives m3	receives m1	sends m4
	receives m2	receives m2	
	receives m4	receives m4	

Message Ordering

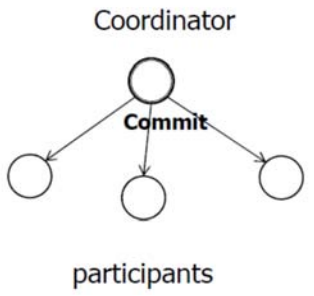
- Causally-ordered multicasts
  - If **m1->m2 (causally related)**, then always deliver m2 after it has delivered m1
  - Use logical clocks (vector clocks)
- Totally-ordered multicasts
  - The message is delivered in the same order to all group members (regardless of the sending order)
  - Somewhat analogous to **sequential consistency**
- Virtually synchronous reliable multicasting offering totally-ordered delivery is called **atomic multicasting**.

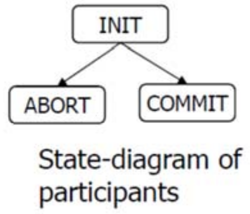
Distributed Commit

- A more general problem of **atomic multicast**
- Definition:
  - Having an operation performed by **all group members** or **none at all**
  - In reliable multicast, operation is delivery of message
- There are three types of "commit protocol"
  - Single-phase, Two-phase and Three-phase commit.

One Phase Commit

- **One phase commit** protocol
  - An elected co-ordinator tells all the other processes whether or not to locally perform the operation in question.
  - But, what if a process cannot perform the operation?
  - There's no way to tell the coordinator!
  - **The solutions**: T*wo-Phase* and *Three-Phase Commit Protocols*





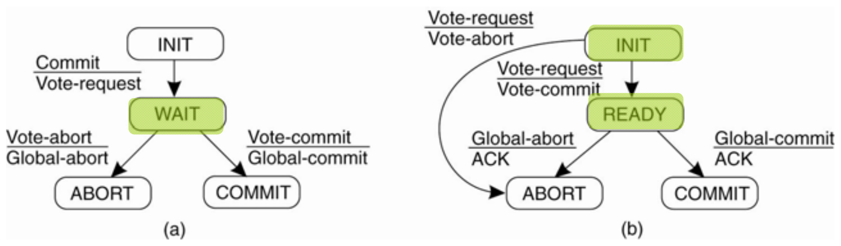
Two Phase Commit

- Goal:
  - Reliably agree to **commit** or **abort** a collection of sub-transactions
- **All** processes in the transaction will agree to **commit** or **abort**
- One transaction manager is *elected* as a **coordinator** – the rest are **participants**
- Assume
  - **Write-ahead log** in stable storage
  - No system dies forever
  - Systems can always communicate with each other

Two Phase Commit

- Model:
  - The client who initiated the computation acts as **coordinator**; processes required to commit are the **participants**
- Phase 1: Voting Phase
  - Coordinator sends **vote-request** to participants (also called a **pre-write**)
  - When participant receives vote-request it returns either **vote-commit** or **vote-abort** to coordinator. If it sends vote-abort, it aborts its local computation
- Phase 2: Commit Phase
  - Coordinator collects all votes; **if all are vote-commit**, it sends **global-commit** to all participants, **otherwise** it sends **global-abort**
  - Each participant waits for global-commit or global-abort and handles accordingly.

Two-Phase Commit

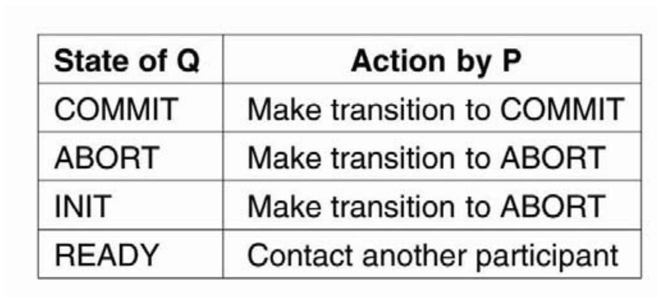


The finite state machine for the **coordinator** in 2PC

The finite state machine for a **participant**

Two-Phase Commit

- Actions taken by a **participant** P when residing in state READY and having contacted another participant Q.



State of Q	Action by P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Two-Phase Commit

- The steps taken by the **coordinator** in a two-phase commit protocol.

```
Actions by coordinator:

    write START_2PC to local log;
    multicast VOTE_REQUEST to all participants;
    while not all votes have been collected {
        wait for any incoming vote;
        if timeout {
            write GLOBAL_ABORT to local log;
            multicast GLOBAL_ABORT to all participants;
            exit;
        }
        record vote;
    }
```

Two-Phase Commit

- The steps taken by the **coordinator** in a two-phase commit protocol.

```
if all participants sent VOTE_COMMIT and coordinator votes COMMIT {
    write GLOBAL_COMMIT to local log;
    multicast GLOBAL_COMMIT to all participants;
} else {
    write GLOBAL_ABORT to local log;
    multicast GLOBAL_ABORT to all participants;
}
```

Two-Phase Commit

- The steps taken by a **participant** process in 2PC.

```
actions by participant:

    write INIT to local log;
    wait for VOTE_REQUEST from coordinator;
    if timeout {
        write VOTE_ABORT to local log;
        exit;
    }
    if participant votes COMMIT {
        write VOTE_COMMIT to local log;
        send VOTE_COMMIT to coordinator;
        wait for DECISION from coordinator;
        if timeout {
            multicast DECISION_REQUEST to other participants;
            wait until DECISION is received; /* remain blocked */
            write DECISION to local log;
        }
        if DECISION == GLOBAL_COMMIT
            write GLOBAL_COMMIT to local log;
        else if DECISION == GLOBAL_ABORT
            write GLOBAL_ABORT to local log;
    } else {
        write VOTE_ABORT to local log;
        send VOTE_ABORT to coordinator;
    }
```

(a)

Two-Phase Commit

- The steps for handling incoming decision requests..

```
Actions for handling decision requests: /* executed by separate thread */

    while true {
        wait until any incoming DECISION_REQUEST is received; /* remain blocked */
        read most recently recorded STATE from the local log;
        if STATE == GLOBAL_COMMIT
            send GLOBAL_COMMIT to requesting participant;
        else if STATE == INIT or STATE == GLOBAL_ABORT
            send GLOBAL_ABORT to requesting participant;
        else
            skip; /* participant remains blocked */
    }
```

(b)

2PC Dealing with Failure

- 2PC assumes a ***fail-recover model***
  - Any failed system will eventually recover
- A recovered system cannot change its mind
  - If a node agreed to commit and then crashed, it must be willing and able to commit upon recovery
- Each system will use a **write-ahead (transaction) log**
  - Keep track of where it is in the protocol (and what it agreed to)
  - As well as values to enable commit or abort (rollback)
  - This enables fail-recover

2PC Dealing with Failure

- Failure during Phase 1 (**voting**)
  - Coordinator dies
    - Some participants may have responded; others have no clue

      ⇒ Coordinator restarts; checks log; sees that voting was in progress

      ⇒ Coordinator **restarts voting**
  - Participant dies
    - The participant may have died before or after sending its vote to the coordinator

      ⇒ If the coordinator received the vote, wait for other votes and **go to phase 2**

      ⇒ Otherwise: **wait for the participant** to recover and respond (**keep querying** it)

2PC Dealing with Failure

- Failure during Phase 2 (**commit**)
  - Coordinator dies
    - Some participants may have given commit/abort instructions

      ⇒ **Coordinator restarts**; checks log; **informs all participants** of chosen action
  - Participant dies
    - The participant may have died before or after getting the commit/abort request

      ⇒ Coordinator **keeps trying to contact the participant** with the request

      ⇒ **Participant recovers**; checks log; gets request from coordinator
      - If it committed/aborted, acknowledge the request
      - Otherwise, process the commit/abort request and send back the acknowledgement

Adding a Recovery Coordinator

- Another system can take over for the coordinator
  - Could be a participant that detected a timeout to the coordinator
- Recovery node needs to find the state of the protocol
  - Contact ALL participants to see how they voted
  - If we get voting results from all participants
    - We know that Phase 1 has completed
    - If all participants voted to commit, send commit request
    - Otherwise send abort request
  - If ANY participant states that it has not voted
    - We know that Phase 1 has not completed, restart the protocol
  - But … if a participant node also crashes, we're stuck!
    - Have to wait for recovery

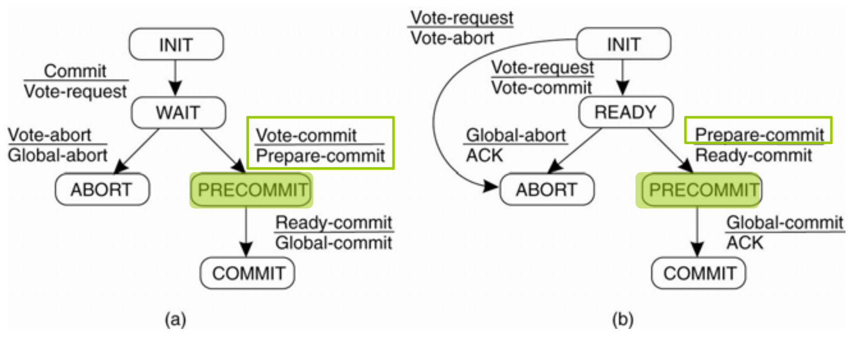
Blocking Commit Protocol

- What's wrong with the 2PC protocol?
- Biggest problem: **It's a blocking protocol**
  - If the coordinator crashes, the group members may not be able to *reach a final decision*, and they may, therefore, block until the coordinator *recovers*
    - A recovery coordinator helps in some cases
  - A non-responding participant will also result in **blocking**
- When a participant gets a commit/abort message, it does not know if every other participant was informed of the result.
- **The solution**: *Three-Phase Commit Protocol*

Three-Phase Commit

- Same setup as the two-phase commit protocol:
  - **Coordinator** & **Participants**
- Enable the use of a **recovery coordinator**
  - **Propagate the result of the commit/abort vote** to each participant *before* telling them to act on it
  - This will allow us to recover the state if any participant dies
- Add **timeouts** to each phase that result in an abort
- If the coordinator crashes – A **recovery** node can query the state from *any available participant*

Three-Phase Commit



The finite state machine for the **coordinator** in 3PC

The finite state machine for a **participant**

Three Phase Commit

- Phase 1: Voting phase
  - Coordinator sends **canCommit?** queries to participants & gets responses
  - Purpose: **Find out if everyone agrees to commit**
  - If the coordinator gets a ***timeout*** from any participant, or any ***NO*** replies are received, send an **abort** to all participants
  - If a participant **times out** waiting for a request from the coordinator, it **aborts** itself (assume coordinator crashed)
  - Else continue to phase 2

Three-Phase Commit

- Phase 2: **"Prepare to commit" phase**
  - Send **Prepare** message to all participants when it received a yes from all participants in phase 1
  - Participants can prepare to commit but cannot do anything that cannot be undone
  - Participants **reply with an acknowledgement**
  - Purpose: ***Let every participant know the state of the result of the vote so that state can be recovered if anyone dies***
  - If the coordinator gets a ***timeout*** (assume participant crashed), send an **abort** to all participants - The coordinator cannot count on every participant having received the Prepare message

Three-Phase Commit

- Phase 3: **"Commit" phase (same as in 2PC)**
  - If coordinator gets ACKs for all **prepare** messages, it will send a **commit** message to all participants
  - Else it will abort – It will send an **abort** message to all participants
  - If participant times out, contact any other participant and move to that state (**commit** or **abort**)
  - If coordinator times out, that's ok

3PC Recovery

- Possible states that the participant may report
  - **Already committed**
    - That means that *every* other participant has received a *Prepare to Commit*
    - Some participants may have committed
    - Send **Commit** message to all participants (just in case they didn't get it)
  - **Not committed but received a Prepare message**
    - That means that all participants agreed to commit; some may have committed
    - Send **Prepare to Commit** message to all participants (just in case they didn't get it)
    - Wait for everyone to acknowledge; then **commit**
  - **Not yet received a Prepare message**
    - This means no participant has committed; some may have agreed
    - Transaction can be **aborted** or the commit protocol can be **restarted**

3PC Weaknesses

- Main weakness of 3PC
  - May have problems when the network gets partitioned
  - Partition A: nodes that received Prepare message
    - Recovery coordinator for A: allows commit
  - Partition B: nodes that did not receive Prepare message
    - Recovery coordinator for B: aborts
  - Either of these actions are legitimate as a whole
    - But when the network merges back, the system is inconsistent
- Not good when a crashed coordinator recovers
  - It needs to find out that someone took over and stay quiet
  - Otherwise it will mess up the protocol, leading to an inconsistent state

3PC Coordinator Recovery Problem

- Suppose
  - A coordinator sent a **Prepare** message to all participants
  - All participants acknowledged the message
  - BUT the coordinator died before it got all acknowledgements
- A recovery coordinator queries a participant
  - Continues with the commit: Sends **Prepare**, gets **ACKs**, sends **Commit**
- Around the same time…*the original coordinator recovers*
  - Realizes it is still missing some replies from the Prepare
  - Times out and decides to send an **Abort** to all participants
- Some processes may commit while others abort!
- 3PC works well when servers crash (fail-stop model)
- **3PC is not resilient against fail-recover environments**

References

- http://csis.pace.edu/~marchese/CS865/Lectures/Chap8/New8/Chapter8.html
- https://www.cs.rutgers.edu/~pxk/rutgers/notes/content/fault-tolerance-slides.pdf
- https://www.cs.rutgers.edu/~pxk/417/notes/content/10-virtual_synchrony-slides.pdf
- https://www.cs.rutgers.edu/~pxk/417/notes/content/11-transactions-slides.pdf
- https://www.cs.rutgers.edu/~pxk/417/notes/content/12-paxos-slides.pdf