Fault-Tolerant Computer System Design
ECE 60872

Distributed Algorithm Primitives: Broadcast, Agreement, Commit

Saurabh Bagchi
ECE/CS
Purdue University

Outline

- Specific issues in design and implementation of networked/distributed systems
- Broadcast protocols
- Agreement protocols
- Commit protocols

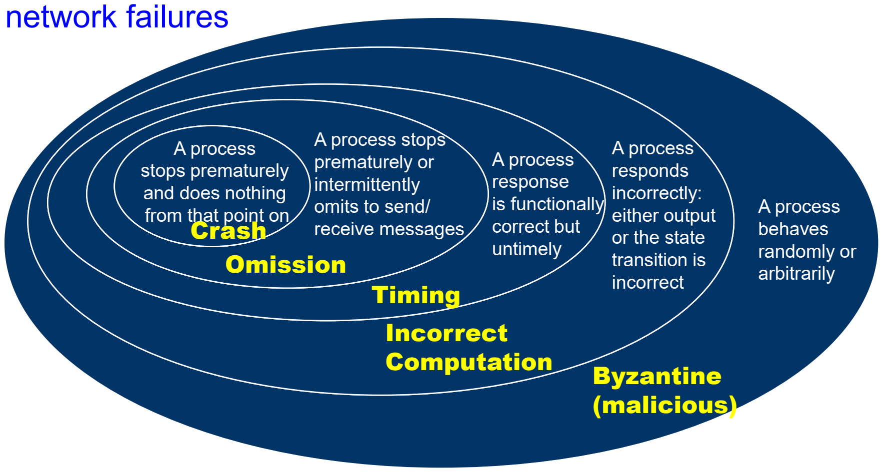
Networked/Distributed Systems Key Questions

How do we integrate components (often heterogeneous) with varying fault tolerance characteristics into a coherent high availability networked system?

- ***How do you guarantee reliable communication (message delivery)?***
- ***How do you synchronize actions of dispersed processors and processes?***
- ***How do you ensure that replicated services with independently executing components have a consistent view of the overall system?***
- ***How do you contain errors (or achieve fail-silent behavior of components) to prevent error propagation?***
- ***How do you adapt the system architecture to changes in availability requirements of the application(s)?***

Failure Classification

- Necessity to cope with machine (node), process, and network failures



What Do We Need in Approaching the Problems?

- Understand and provide solution to *replication problem* (in its broad meaning)
  - process/data replication
  - replica consistency and replica determinism
  - replica recovery/reintegration
  - redundancy management
- Provide efficient techniques capable of supporting a consistent data and coherent behavior between system components despite failures

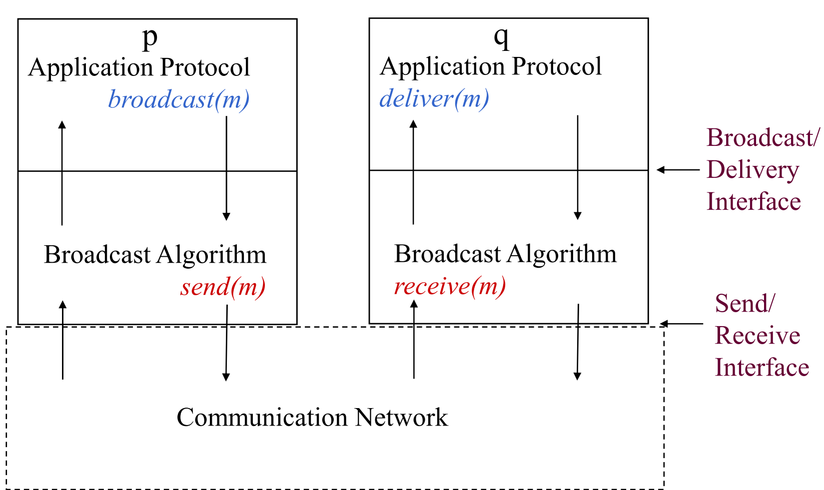
What Do We Need in Approaching the Problems?

- Problems posed by replication
  - Replication of processes
  - Replication of data
- Techniques include:
  - ***Broadcast protocols*** (e.g., atomic broadcast, causal broadcast), which ensure reliable message delivery to all participants (replicas)
  - ***Agreement protocols***, which ensures all participants have a consistent system view
  - ***Commit protocols***, which implement atomic behavior in transactional types of systems

Broadcast Protocols

- Cooperating processes in networked /distributed systems often communicate via broadcast
- A failure during a broadcast can lead to inconsistency and can compromise the integrity of the system
- Need for supporting reliable broadcast protocols that provide strong guarantee on message delivery
- Example protocols include
  - reliable broadcast
  - FIFO broadcast
  - causal broadcast
  - atomic broadcast

Application/Broadcast Layering



What Do We Assume?

- The system consists of a set of sites interconnected through a communication network
- Computation processes communicate with each other by exchanging messages
- Process failures can be detected by timeouts
  - Processes suffer crash or omission failures
  - Communication is synchronous and each message is received within a bounded time interval

What Do We Assume?

- The network is not partitioned
- Conventional Message-Passing Technologies
  - Unreliable datagrams (e.g., UDP)
  - Remote procedure call (RPC)
  - Reliable data streams (e.g., TCP)

Goal: **Provide robust techniques/algorithms for supporting consistent data and reliable communications in a networked environment**

Reliable Broadcast

- **Reliable broadcast** guarantees the following properties:
  - ***Validity:*** if a correct process broadcasts a message *m*, then all correct processes eventually deliver *m* (all messages broadcast by correct processes are delivered)
  - ***Agreement:*** if a correct process delivers a message *m*, then all correct processes eventually deliver *m* (all correct processes agree on the set of messages they deliver),
  - ***Integrity:*** for any message *m*, every correct process delivers *m* at most once and only if *m* was previously broadcast by a sender (no spurious messages are ever delivered)
- Reliable broadcast imposes no restrictions on the order of messages delivery

Reliable Broadcast by Message Diffusion

- Consider an asynchronous system where every two correct processes are connected via a path of processes and links that never fail

Every process p executes the following:
To execute **broadcast(*R, m*)**
  tag *m* with *sender(m)* and *seq#(m)* *//these tags make m unique*
  send(*m*) to all neighbors including *p*

deliver(*R, m*) *occurs as follows:*
  upon receive(*m*) do
    if *p* has not previously executed **deliver(*R, m*)**
    then
      if *sender(m) != p* **then send(*m*) to** all neighbors
      deliver(*R, m*)

Reliable Broadcast by Message Forwarding

- Consider the network as a tree
  - Root is the initiator of the broadcast, call it *S*
  - If edge from node *P* to node *Q* in the tree, then *P* will forward the message to *Q*
  - Tree is a logical structure and has no relation to the physical structure of the network

1. Upon receiving a message, node *i* sends the message to all $j \in$ CHILD(*i*)
2. Node *j* sends ACK to node *i*
3. Node *j* sends message to all its children nodes
4. If node *i* does not get an ACK from *j*, it assumes *j* has failed and takes over the responsibility of forwarding message to all $k \in$ CHILD(*j*)
5. Each node eliminates duplicates using (*S*, *m*.seq_no)

Reliable Broadcast by Message Forwarding (Cont'd)

- How to handle failure of root node *S*?
- Case 1: *S* fails after sending *m* to all its children
  - No problem – protocol takes care of it
- Case 2: *S* fails before sending *m* to any of its children
  - No problem – broadcast has not even started
- Case 3: S fails after sending *m* to some, but not all, of its children
  - A child of *S* has to take over responsibility
  - Multiple children can take over responsibility – each node just eliminates duplicates
  - When *S* completes sending to all its children, it can inform its children

    OR
  - A child receiving the next broadcast message m_2 serves as indication that S has completed sending m_1 to all its children

FIFO Broadcast

- **FIFO Broadcast** is a Reliable Broadcast that satisfies the following requirement on message delivery

  FIFO order: if a process broadcasts a message *m* before it broadcasts a message *m'*, then no correct process delivers *m'*, unless it has previously delivered *m* (messages sent by the same sender are delivered in the order they were broadcast)

ECE 60872 15 PURDUE UNIVERSITY

Build FIFO Broadcast Using Reliable Broadcast

Every process p executes the following:
Initialization:
msgBag := ∅ *//set of messages that p R-delivered*
// but not yet F-delivered
next[q] := 1 for all q *//sequence number of next message from q*
//that p will F-deliver

To execute **broadcast(*F, m*)**
broadcast(*R, m*)

deliver(*F, m*) *occurs as follows:*
upon deliver(*R, m*) do
q := sender(m)
msgBag := msgBag ∪ *{m}*
while *(∃ m' ∈ msgBag: sender (m')== q* **and** *seq#(m')== next[q])* **do**
deliver(*F, m'*)
next[q] := next[q] +1
msgBag := msgBag – {m'}

ECE 60872 16 PURDUE UNIVERSITY

FIFO Broadcast (cont.)

- The FIFO Order is not sufficient if a message m depends on messages that the sender of m delivered before broadcasting m, e.g., let consider a network news application where users distribute their articles with FIFO broadcast
  - *user_1* broadcast an article
  - *user_2* delivers that article and broadcasts a response that can only be properly handled by a user who has the original article
  - *user_3* delivers *user_2's* response before delivering the original article from *user_1* and consequently misinterprets the response

- ***Causal broadcast*** prevents the above problem by introducing the notion of a *message depending on another* one and ensuring that a message is not delivered until all the messages it depends on have been delivered

Causal Broadcast

- **Causal Broadcast** is a Reliable Broadcast that satisfies the following requirement on message delivery

  Causal Order: if the broadcast of message *m* causally precedes the broadcast of a message *m'*, then no correct process delivers *m'* unless it has previously delivered *m*

Causal Broadcast Using FIFO Broadcast

Every process p executes the following:
Initialization:
prevDlvrs := ∅ *//sequence of messages that C-delivered // since its previous C-broadcast*

To execute **broadcast(*C, m*)**
  broadcast(*F*, <*prevDlvrs* || *m*>)
  prevDlvrs := ∅

deliver(*C, m*) *occurs as follows:*
  upon deliver(*F*, <m_1, m_2, …, m_l>) for some *l* **do**
    for *i* := *1…l* **do**
      if *p* has not previously executed **deliver(*C*, m_i)**
      then
        deliver(*C*, m_i)
        prevDlvrs := *prevDlvrs* ∪ {m_i}

Causal Broadcast (cont.)

- *Causal Broadcast* does not impose any order on those messages that are not causally related
  - consider a replicated database with two copies of a bank account *client_account* residing at different sites. Initially *client_account* has an amount of $1000.
  - A user deposits $150 triggering a broadcast of *msg1* = {add $150 to *client_account* } to the two copies of *client_account.*
  - At the same time, at other site, the bank initiates a broadcast of *msg2* = {add 8% interest to *client_account* }
  - the two broadcasts are not causally related, the Causal Broadcast allows the two copies of *client_account* to deliver these updates in different order and creates inconsistency in the database
- *Atomic Broadcast* prevents such problem by providing strong message ordering or total order

Atomic Broadcast

- **Atomic Broadcast** is a Reliable Broadcast that satisfies the following condition

  Total Order: if correct processes *r* and *s* both deliver messages *m* and *m'*, then *r* delivers *m* before *m'* if and only if *s* delivers *m* before *m'* (messages sent concurrently are delivered in identical order to the selected destinations)

Atomic Broadcast Protocol using Message Queues

- Two phase protocol
- Each process has a queue in which it stores received messages
- Phase I

1. A sender has a group of receivers to send a message to. It multicasts the message to the group, with the receiver ids in the message.
2. On receiving a message, a receiver:

- Assigns a priority (highest among all buffered messages), marks it undeliverable, and buffers it in the message queue.
- Informs the sender of the message priority.

Atomic Broadcast Protocol using Message Queues

- Phase II

1. When sender receives responses from all receivers:
   - Chooses the highest priority as the final message priority.
   - Multicasts the final priority to all receivers.
2. When a receiver receives the final priority:
   - Assigns priority to corresponding message.
   - Marks the message as deliverable.
   - Orders messages in increasing order of priorities.
   - Message is delivered when it reaches head of the queue and is marked deliverable.

Atomic Broadcast Protocol using Message Queues: Failure Scenario

- A receiver detects it has a message marked undeliverable and sender has failed. It becomes the new sender/coordinator.

1. It asks all receivers about status of message. Three possible answers:
   I. Message is marked undeliverable and its associated priority.
   II. Message is marked deliverable and the final priority of the message.
   III. It has not received the message.
2. After receiving responses from all receivers:
   I. If message marked deliverable at any receiver, it assigns that as the final priority and multicasts it. On receiving this, receivers execute phase II.2 actions.
   II. Otherwise, the coordinator reinitiates the protocol from phase I.

Remarks on Broadcasts

- **Inconsistency and contamination**
  - suppose that a process *p* fails by omitting to deliver a message that is delivered by all the correct processes
  - state of *p* might be inconsistent with other correct processes
  - *p* continues to execute and *p* broadcasts a message *m* that is delivered by all the correct processes
  - *m* might be corrupted because it reflects p's erroneous state
  - correct processes get contaminated by incorporating p's inconsistency into their own state.

  Observation: Broadcast can lead to the corruption of the entire system

Remarks on Broadcasts (cont.)

- To prevent contamination a process can refuse to deliver messages from processes whose previous deliveries are not compatible with its own
  - a message must carry additional information , so that the receiving process can determine whether it is safe to deliver the message
- To prevent inconsistency requires techniques that ensure that the faulty process will immediately stop to execute (i.e., the process is fail-silent)

Remarks on Broadcasts (cont.)

- A fault-tolerant broadcast is usually implemented by a broadcast algorithm that uses lower-level communication primitives, such as point-to-point message *sends* and *receives*
- The failure models are usually defined in terms of failures that occur at the level of *send* and *receive* primitives, e.g., omission to receive messages
- How do these failures affect the execution of higher-level primitives, such as *broadcast* and *delivery*? For example, if a faulty process omits to receive messages, will it simply omit to deliver messages?
- In general broadcasts algorithms are likely to amplify the severity of failures that occur at the low level communication primitives (*sends* and *receives*).
  - e.g., the omission to receive messages may cause a faulty process to deliver messages in the wrong order

Primitives for Fault-Tolerance in Distributed/Networked Systems

- Techniques include:
  - ***Broadcast protocols*** (e.g., atomic broadcast, causal broadcast), which ensure reliable message delivery to all participants (replicas)
  - ***Agreement protocols***, which ensures all participants have a consistent system view
  - ***Commit protocols***, which implement atomic behavior in transactional types of systems

Agreement Protocols

- In a distributed system, it is often required that processes reach a mutual agreement.
- Faulty processes can send conflicting values to other processors preventing them from reaching an agreement
- In the presence of faults, processes must exchange their values and relay the values received from other processes several times to isolate the effects of faulty processes.
- System model
  - There are *n* processes in the system and at most *m* of them can be faulty.
  - Processes communicate with one another by message passing and the receiver process always knows the identity of the sender process of the message.
  - The communication network is reliable, i.e., only processes are prone to failures.

Synchronous vs. Asynchronous Computation

- In ***synchronous computation***, processes in the system run in lockstep:
  - In each step/round, a process receives messages (sent to it in the previous step), performs computation, and sends messages to other processes (received in the next step).
  - A process knows all the messages it expects to receive in a step/round.
- In ***asynchronous computation***, processes do not execute in lockstep:
  - A process can send and receive messages and perform computation at any time
- The **synchronous model** of computation is assumed in further discussion

Model of Processor Failures

- Three modes of failures
  - Crash fault
  - Omission fault
  - Byzantine fault
- Crash fault: Processor stops functioning and never resumes operation
- Omission fault: Processor "omits" to send messages to some processors
- Malicious fault: Processor behaves randomly and arbitrarily (Byzantine fault)
- In synchronous model, omission can be detected

Authenticated vs. Non-Authenticated Messages

- To reach an agreement, processes need to exchange their values and relay the received values to other processors.
- A faulty process can distort a message received from other processes.

Two Types of Messages:

- ***Authenticated (signed)***
  - A faulty process cannot forge a message or change the contents of a received message (before it relays the message to other processes).
  - A process can verify the authenticity of the received message.
- ***Non-authenticated (oral)***
  - A faulty process can forge a message and claim to have received it from another processor or change the contents of the received message before it relays it to other processes.
  - A process has no way to verify the authenticity of the received message.

Agreement Problems - Classification

- ***The Byzantine Agreement Problem***
  - **A single value** is **initialized by any arbitrary process,** and all nonfaulty processes have to agree on that value
- ***The Consensus Problem***
  - Every process has its **own initial value,** and all correct processes must agree on **a single, common value.**
- ***The Interactive Consistency Problem***
  - Every process has its **own initial value,** and all nonfaulty process must agree on **a set of common values.**

The Byzantine Agreement Problem

- An **arbitrarily chosen process** - *the source process* - broadcasts its initial value to all other processes.
- ***Agreement*** - All nonfaulty processes agree on the same value.
- ***Validity*** - If the source process is nonfaulty then the common value agreed on by all nonfaulty processes should be the initial value of the source.

The Consensus Problem

- **Every process** broadcasts its initial value to all other processes.
  - *Initial values of the processes may be different.*
- ***Agreement*** - All nonfaulty processes agree on the same single value.
- ***Validity*** - if the initial value of every nonfaulty process is υ, then the common value agreed upon by nonfaulty processes must be υ.

The Interactive Consistency Problem

- **Every process** broadcasts its initial value to all other processes.
  - *Initial values of the processes may be different.*
- ***Agreement*** - All nonfaulty processes agree on the same vector:
  $(\upsilon_1, \upsilon_2, \ldots, \upsilon_n)$
- ***Validity*** - If the i_{th} process is nonfaulty and its initial value is υ_i, then the i_{th} value to be agreed on by all nonfaulty processes must be υ_i

Relations Among the Agreement Problems

1. Given an algorithm to solve Byzantine agreement, how would you solve Interactive Consistency?
2. Given an algorithm to solve Interactive Consistency, how would you solve Consensus?
3. Given an algorithm to solve Consensus, how would you solve Byzantine Agreement?

Byzantine Agreement Problem: Solution

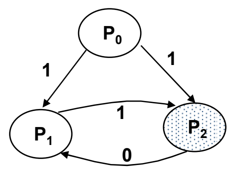
The upper bound on the number of faulty processes

- It can be shown that in a fully connected network it is impossible to reach a consensus if the number of faulty processes, *m* , exceeds $\lfloor (n-1)/3 \rfloor$,
  - For example, if n = 3, than m = 0, i.e., having three processes, we cannot solve the Byzantine agreement problem even in the event of a single error.
  - The protocol requires *m+1 rounds of message exchange* (*m* is the maximum number of faulty processes)
  - This is also the lower bound on the number of rounds of message exchanged.
- Using authenticated messages, this bound is relaxed, and a consensus can be reached for any number of faulty processes.

Impossibility Results

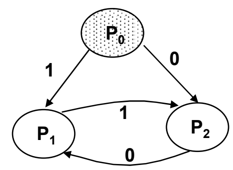
- Consider a system with three processes p_1, p_2, p_3
- There are two values, 0 and 1, on which processes agree.
- p_0 initiates the algorithm.

Case one - p_0 is not faulty



assume p_2 is faulty
suppose p_0 broadcast 1 to p_1 and p_2
p_2 acts maliciously and sends 0 to p_1
p_1 must agree on 1 if algorithm is to be satisfied
p_1 receives two conflicting values
no agreement is possible

Case one - p_0 is faulty



suppose p_0 sends 1 to p_1 and 0 to p_2
p_2 communicates 0 to p_1
p_1 receives two conflicting values
no agreement is possible

- No solution exists for the Byzantine agreement problem for three processes, which can work under a single failure

Oral Messages Algorithm OM(m)

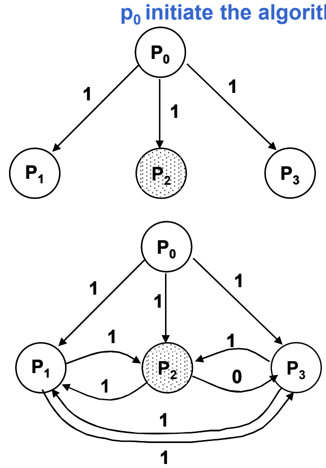
- A recursive algorithm solves the Byzantine agreement problem for **3m+1** or more processes in the presence of at most ***m*** faulty processes.
- **Algorithm OM(0)**
- 1. The source process sends its value to every process.
- 2. Each process uses the value it receives from the source (if it receives no value, then it uses a default value of 0).

Oral Messages Algorithm OM(m)

- **Algorithm OM(m), m > 0**
- 1. The source process sends its value to every process.
- 2. For each *i*, let υ_i be the value processor *i* receives from the source.
  - Process *i* acts as a new source and initiates *Algorithm OM(m-1)* wherein it sends the value υ_i to each of the n-2 other processes.
- 3. For each *i* and each $j \neq i$ let υ_j be the value process *i* received from *j* in step (2) using *Algorithm OM(m-1).* (If no value is received then default value 0 is used). Process *i* uses the value *majority* $(\upsilon_1, \upsilon_2, \ldots, \upsilon_{n-1})$.
- The algorithm is complex
  - Message complexity?
  - Time complexity?

Oral Messages Algorithm OM(m): An Example

Consider a system with four processes p_0, p_1, p_2, p_3
p_0 initiate the algorithm; p_2 is faulty



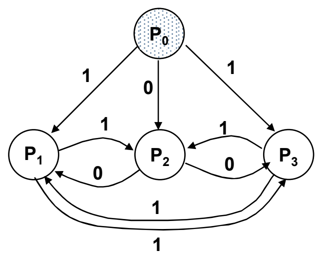
To initiate the agreement p_0 executes OM(1) wherein it sends 1 to all processes

At step 2 of the OM(1) algorithm, p_1, p_2, p_3 execute the algorithm OM(0)
p_1 and p_3 are nonfaulty and
p_1 sends 1 to $\{p_2, p_3\}$
p_3 sends 1 to $\{p_1, p_2\}$
p_2 is faulty and sends 1 to p_1 and 0 to p_3

After receiving all messages p_1, p_2, p_3 execute step 3 of the OM(1) to decide the majority value
p_1 received $\{1, 1, 1\} \Rightarrow 1$
p_2 received $\{1, 1, 1\} \Rightarrow 1$
p_3 received $\{1, 1, 0\} \Rightarrow 1$
Both conditions of the Byzantine agreement are satisfied

Oral Messages Algorithm OM(m): An Example (cont.)

Consider a system with four processes p_0, p_1, p_2, p_3
p_0 initiate the algorithm; p_0 is faulty



P_0 send conflicting values to p_1, p_2, p_3

Under step 2 of OM(0) p_1, p_2, p_3 send the received values to the other two processes

p_1, p_2, p_3 execute step 3 of OM(1) to decide on the majority value

p_1 received {1, 0, 1} $\Rightarrow$ 1
p_2 received {0, 1, 1} $\Rightarrow$ 1
p_3 received {1, 1, 0} $\Rightarrow$ 1

Both conditions of the Byzantine agreement are satisfied

Protocol with Signed Messages

- Transmitter sends a "signed" message (use digital signature from asymmetric cryptography)
- If a node changes the content of message from transmitter before forwarding it, the receiver can detect the forgery
- With signed messages, agreement can be reached between $n=m+2$ processes, where m is the number of faulty processes
- Each process maintains a set V_i (for process i) that has all the unique values that it has received

Protocol with Signed Messages

- Algorithm SM(m)

1. The transmitter (process 0) signs its value and sends to other nodes
2. For each process *i*:
   A. If process *i* received message *v*: 0 (i) it sets V_i to {*v*}; (ii) it sends *v*: 0: *i* to every other process
   B. If process *i* received message *v*: 0: j_1: … : j_k and $v \notin V_i$, then (i) it adds *v* to V_i; (ii) if $k < m$, it sends *v*: 0: j_1: …: j_k : *i* to every process other than j_1, …, j_k
3. For each process *i*, when it receives no more message, it considers the final value as *choice*(V_i)

Application of Agreement Algorithms

Fault-Tolerant Clock Synchronization Example

- In distributed systems, it is often necessary for processes to maintain synchronized physical clocks.
- Drift of the physical clock requires the clocks at different processes to be periodically resynchronized.
- It is assumed that
  – All clocks are initially synchronized to approximately the same value.
  – A nonfaulty process's clock runs approximately at the correct rate (i.e., one second of clock time per second of real time).
  – A nonfaulty process can read the clock value of another nonfaulty process with a small error ε

Fault-Tolerant Clock Synchronization

Interactive Convergence Algorithm

- The clocks are:
  - Initially synchronized
  - Resynchronized often enough so that two nonfaulty clocks never differ by more than δ
- Each process reads the value of all other processes' clocks and sets its clock value to the average of these values.
- If a clock value differs from a process's own value by more than δ, the process replaces that value by its own clock value when taking the average.

Fault-Tolerant Clock Synchronization

Interactive Convergence Algorithm (cont.)

- Let two processes *p* and *q*, use c_{pr} and c_{qr} as the clock values of a third process *r* when computing their averages.
- If *r* is nonfaulty, then $c_{pr} = c_{qr}$. *Actually* $|c_{pr} - c_{qr}| \leq \varepsilon$
- If *r* is faulty then $|c_{pr} - c_{qr}| \leq 3\delta$
- If *p* and *q* computes their averages for the *n* clocks values:
  - use identical values for clocks of *n-m* nonfaulty processes.
  - The difference in the clock values of *m* faulty processes used is bounded by 3δ
- The averages computed by *p* and *q* differ by at most $(3m/n)\delta$

$$n > 3m \Rightarrow (3m/n)\delta < \delta$$

Resynchronization brings the clocks closer by a factor of *(3m/n)*

Fault-Tolerant Clock Synchronization
Interactive Convergence Algorithm (cont.)

- In the algorithm, it was assumed that:
  - All processes execute the algorithm instantaneously at exactly the same time.
  - The error in reading another process's clock is zero.
- A process may read other processes' clocks at different time instances

Solution:

- A process computes the average of the difference in clock values and increments its clock by the average increment.
  - Clock differences larger than δ are replaced by 0.

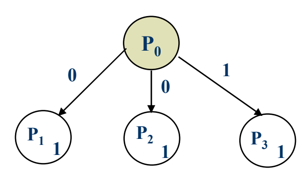
Interactive Consistency by Running the Byzantine Agreement Protocol, Example 1

Consider a system, which consists of four processes: p_0, p_1, p_2, p_3
Initial values in the processes: v_1=1, v_2=1, v_3=1, v_0=1

	Src p_0	p_1	p_2	p_3
P_0 sender : P_1 received	0		0	1
P_2	0	0		1
P_3	1	0	0	
P_1 sender : P_1				
P_2	0/1	1		1
P_3	0/1	1	1	
P_2 sender : P_1	0/1		1	1
P_2				
P_3	0/1	1	1	
P_3 sender : P_1	0/1		1	1
P_2	0/1	1		1
P_3				

majority

own value

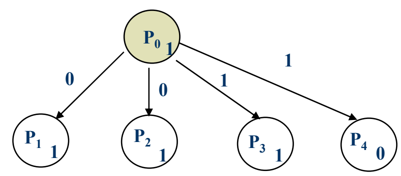


P_1={0,1,1,1} ==> 1
P_2={0,1,1,1} ==> 1
P_3={0,1,1,1} ==> 1

Vectors in each process **Final decision**

Interactive Consistency by Running the Byzantine Agreement Protocol, Example 2

		p0	p1	p2	p3	p4
P_0 sender :	P_1 received	0		0	1	1
	P_2	0	0		1	1
	P_3	1	0	0		1
	P_4	1	0	0	1	
P_1 sender :	P_1					
	P_2	0/1	1		1	1
	P_3	0/1	1	1		1
	P_4	0/1	1	1	1	
P_2 sender :	P_1	0/1		1	1	1
	P_2					
	P_3	0/1	1	1		1
	P_4	0/1	1	1	1	
P_3 sender :	P_1	0/1		1	1	1
	P_2	0/1	1		1	1
	P_3					
	P_4	0/1	1	1	1	
P_4 sender :	P_1	0/1		0	0	0
	P_2	0/1	0		0	0
	P_3	0/1	0	0		0
	P_4					

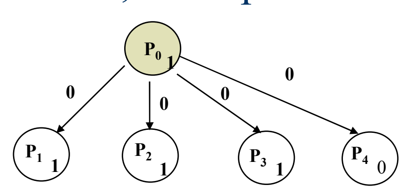


P_1={0,1,1,1,0} ==> 1
P_2={0,1,1,1,0} ==> 1
P_3={0,1,1,1,0} ==> 1
P_4={0,1,1,1,0} ==> 1

Vectors in each process **Final decision**

Interactive Consistency by Running the Byzantine Agreement Protocol, Example 3

		p0	p1	p2	p3	p4
P_0 sender :	P_1 received	0		0	0	0
	P_2	0	0		0	0
	P_3	0	0	0		0
	P_4	0	0	0	0	
P_1 sender :	P_1					
	P_2	0/1	1		1	1
	P_3	0/1	1	1		1
	P_4	0/1	1	1	1	
P_2 sender :	P_1	0/1		1	1	1
	P_2					
	P_3	0/1	1	1		1
	P_4	0/1	1	1	1	
P_3 sender :	P_1	0/1		1	1	1
	P_2	0/1	1		1	1
	P_3					
	P_4	0/1	1	1	1	
P_4 sender :	P_1	0/1		0	0	0
	P_2	0/1	0		0	0
	P_3	0/1	0	0		0
	P_4					



P_1={0,1,1,1,0} ==> 1
P_2={0,1,1,1,0} ==> 1
P_3={0,1,1,1,0} ==> 1
P_4={0,1,1,1,0} ==> 1

Vectors in each process **Final decision**

Primitives for Fault-Tolerance in Distributed/Networked Systems

- Techniques include:
  - ***Broadcast protocols*** (e.g., atomic broadcast, causal broadcast), which ensure reliable message delivery to all participants (replicas)
  - ***Agreement protocols***, which ensures all participants have a consistent system view
  - ***Commit protocols***, which implement atomic behavior in transactional types of systems

Commit Protocols

- The commit problem occurs when a set of processes need to agree on whether or not to perform some action that may not be possible for some of the participants
- The initial uncertainty is overcome by:
  - determine whether or not all the participant will be able to perform the operation
  - communicate the outcome of the decision to the participants in a reliable way
- The operation can be ***committed*** if the participants can all perform it
- Once a commit is reached, this requirements will hold even if some participants fail and later recover
- If one or more participants are unable to perform the operation, the operation as a whole ***aborts***, i.e, no participant should perform it

Atomic Actions – Process Interaction Example

Process P_1	Process P_2
….	….
…..	…..
Lock(X)	Lock(X)
X := X + Z;	X := X + Y;
Unlock(X);	Unlock(X);
….	….
…..	…..
failure	

1. Suppose P_1 and P_2 share a memory location X and both modify X
2. Suppose P_1 locks X before P_2
3. P_1 updates X and releases the lock
4. If P_1 fails after P_2 has seen the change made to X by P_1
   then
   P_2 must be aborted or rolled back to recover the correct system state

a) P_2 should not interact with P_1 until this can be done safely
b) P_1 should be atomic, i.e., the effect of P_1 on the system should look like an uninterrupted operation

Two-Phase Commit Protocol - Assumptions

- The system consist of a set of sites/nodes interconnected through a communication network
- Computation processes communicate with each other by exchanging messages
- Processes suffer crash or omission failures
- Communication is reliable and each message is received within δ time units after being sent
- One of the cooperating processes acts as a ***coordinator***
- Coordinator cooperates with other processes called ***cohorts***
- Stable storage is available at each site/node

Two-Phase Commit Protocol (2PCP)

- At the beginning of a transaction, the coordinator sends a start transaction message to every cohort.

Phase 1

- ***Coordinator***
  - send a ***Commit_Request*** to every cohort
  - wait *with a timeout* for replies from all cohorts
- ***Cohorts***
  - on receiving ***Commit_Request***

    if the transaction execution is successful

    write ***Undo*** and ***Redo*** log on the stable storage

    send an ***Agreed*** message to the coordinator

    otherwise send an ***Abort*** to the coordinator
  - wait forever for ***Commit*** or ***Abort*** from the coordinator

Two-Phase Commit Protocol (cont.)

Phase 2

- ***Coordinator***
  - if all cohorts reply ***Agreed***

    write ***Commit*** into the log

    send ***Commit*** to all cohorts and wait *forever* for ***Acknowledgments*** from cohorts

    if all cohorts respond with ***Acknowledgment*** write a *Complete* record to the log
  - if some cohort responds with *ABORT* or *timeouts* (does not respond within a timeout interval)

    send ***Abort*** to all the cohorts, undo database changes (using UNDO log) and log *Complete* record
  - when the *Complete* record is written, delete the live transaction state.

Two-Phase Commit Protocol (cont.)

Phase 2

- ***Cohorts***
  - on receiving a ***Commit*** write a *Complete* record and send an ***Acknowledgment*** to the coordinator
  - on receiving an ***Abort*** undo the transaction (using the ***Undo*** log) and log the *Complete* record
  - when the *Complete* record is written, delete the live transaction state.

Site/Node Failures

- **Coordinator crashes before having written the *Commit* record:**
  - On recovery, the coordinator broadcasts an *Abort* message to all cohorts.
  - All cohorts who agreed to commit undo the transaction using *Undo* log.
  - Others *Abort* the transaction.
  - Cohorts are blocked until they receive an *Abort* message.
- **Coordinator crashes after writing *Commit* but before writing the *Complete* record: ???**
- **Coordinator crashes after writing *Complete* record:**
  - On recovery, there is nothing to be done for the transaction.

Site/Node Failures (cont.)

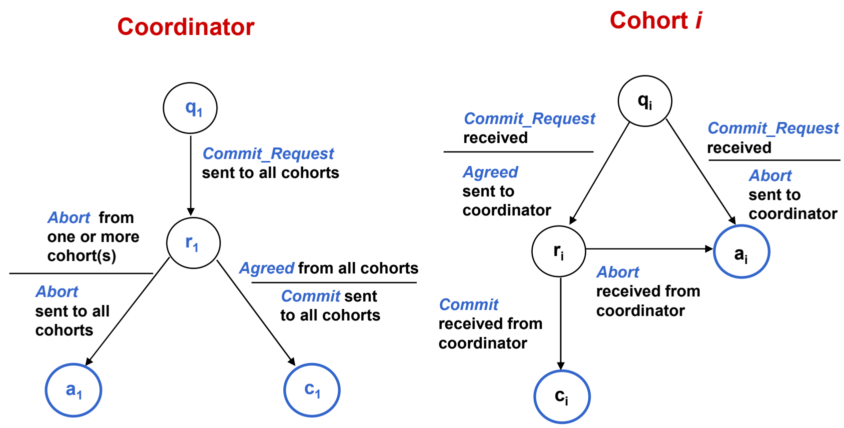
- **A cohort crashes in Phase 1:**
  - The coordinator can abort the transaction, as it did not receive a reply from the crashed cohort.
- **A cohort crashes in Phase 2 (after writing *Undo* and *Redo* log):**
  - ???
- **The two-phase commit protocol guarantees global atomicity.**
- **The protocol can block: How?**

Non-blocking Commit Protocol

- Need a commit protocol that:
  - Is non-blocking
  - Tolerates site failures
- Implies independent recovery
  - Operational sites should agree on outcome of transaction by examining local state
  - Failed sites upon recovery should reach same decision as operational sites
- Assumptions:
  - The network is reliable.
  - Point-to-point communication is possible between any two operational nodes.
  - The network can detect a node failure (e.g., by a timeout) and report it to the site trying to communicate with the failed site (fail-stop failure mode)

Two-Phase Commit Protocol

Finite State Automate



How Is Blocking of a 2PCP Eliminated?

- **Concurrency Set**
  - Let s_i depict the state of the node *i*
  - A *concurrency set* of s_i ($C(s_i)$) is the set of all the states of every node that can be concurrent with s_i
  - Consider a system with two nodes (one coordinator and one cohort):
    $C(r_2) = \{c_1, a_1, r_1\}$, and **$C(q_2) = \{q_1, r_1\}$**

- If a protocol contains the local state of a site with both abort and commit states in its concurrency set, then under independent recovery conditions it is **not resilient to an arbitrary single failure.**

How Is Blocking of a 2PCP Eliminated?

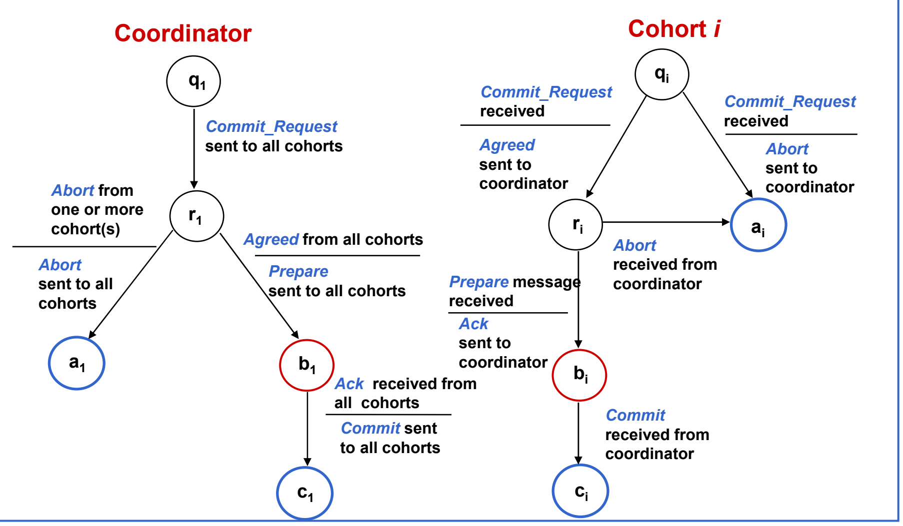
- In the 2PCP, only states r_i $(i \neq 1)$ have both abort and commit in their $C(r_i)$
- This can be resolved by introducing a *buffer state* b_i in the finite state automaton representing 2PCP, e.g., a system containing only two sites:

$$C(r_1) = \{q_2, a_2, r_2\} \text{ and } C(r_2) = \{a_1, b_1, r_1\}$$

The extended 2PCP is non-blocking in case of a single site failure and a failed site can perform independent recovery

How Is Blocking of a 2PCP Eliminated?

Three-Phase Commit Protocol (3PCP)



Failure Transitions

- The failed site should be able to reach a final decision based solely on its local state.
- The decision-making process is modeled using ***failure transitions.***
- A failure transition occurs at a failed site/node at the instant it fails or immediately after it recovers from the failure.

The Rule:

- For each nonfinal state *s* (i.e., q_i, r_i, b_i) in the protocol,
  - If *C(s)* contains a ***Commit***, then assign ***a failure transition from s to a commit state.***
  - Otherwise, assign ***a failure transition from s to an abort state.***

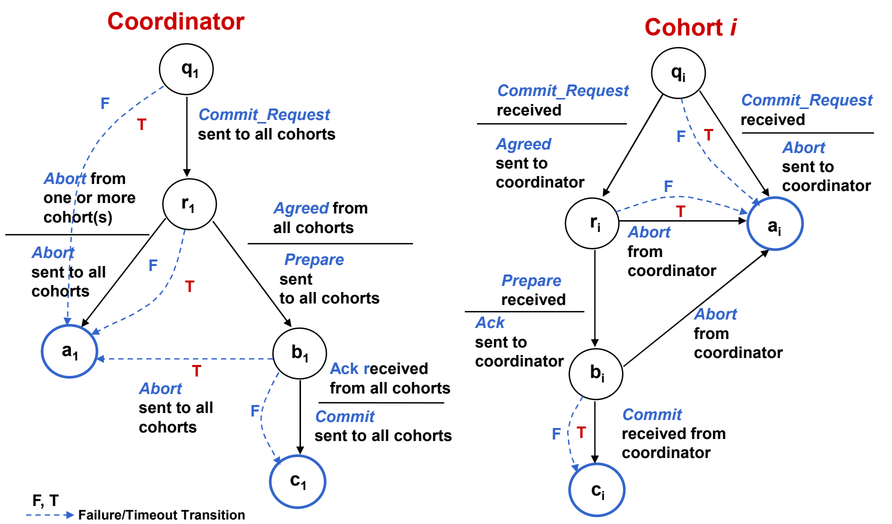
Timeout Transitions

- Consider what an operational site does in the event of another site's failure.
- If site/node *i* is waiting for a message from site *j (i.e., $j \in S(i)$)* and *j* has failed, then site/node *i* times out.
- Based on the expected message type, site/node *i* can determine in what state site *j* failed.

The Rule:

- For each non-final state *s*,
  - If site *j* is in *S(s)* and node *j* has *a failure transition to a commit (abort)* state, then ***assign the time out transition from state s to a commit (abort) state.***

Finite State Automate to Illustrate Timeout and Failure Transitions



Three-Phase Commit Protocol (3PCP)

- All nodes are in the state *q*.
- If the coordinator fails while in state q_1 all cohorts
  - Time out (waiting for the ***Commit_Request*** message)
  - Perform time out transition and abort the transaction
- Upon recovery, the coordinator performs the failure transition from state q_1 and aborts the transaction.

Three-Phase Commit Protocol (3PCP): *Phase 1*

- Error-free execution identical to the Phase 1 of the 2PCP
- In the event of a site/node failure (the coordinator is in state r_1 each cohort is either in state *a*, *r,* or *q),*
  - ***In state a*** - a cohort has already sent an ***Abort*** message to the coordinator
  - ***In states r*** or ***q*** - if a cohort fails, the coordinator
    - times out waiting for the ***Agreed*** message from the failed cohort
    - aborts the transaction and sends abort message to all cohorts

Three-Phase Commit Protocol (3PCP): *Phase 2*

- Coordinator sends a ***Prepare*** message to all the cohorts if all the cohorts sent ***Agreed*** messages in *Phase 1.*
- Otherwise the coordinator sends an ***Abort*** message to all cohorts.
- Upon receiving a ***Prepare*** message a cohort sends an ***Acknowledgment*** **(*Ack*)** message to the coordinator.
- If coordinator fails **before sending the *Prepare* message** (in state r_1)
  - It aborts the transaction on recovery.
  - The cohorts time out waiting for the ***Prepare*** message and abort the transaction.

Three-Phase Commit Protocol (3PCP): *Phase 3*

- On receiving ***Ack*** messages from all cohorts, the coordinator sends a ***Commit*** message to all the cohorts.
- A cohort on receiving a ***Commit*** message, commits the transaction.
- If the coordinator fails **before sending the *Commit* message** (in state b_1)
  - It commits the transaction upon recovery.
  - The cohorts time out waiting for commit message and commit the transaction from state b_1.
- If the cohort fails **before sending an *Ack* message**
  - The coordinator times out, aborts the transaction, and sends the ***Abort*** to all cohorts.
  - The failed cohort aborts the transaction on recovery.

Why Cohorts Need a Buffer State (b_i)

- Assuming that b_i is not present,
  - Let the coordinator be in b_1 waiting for ***Ack*** form cohorts.
  - Cohort 2 (in state r_2) sends an ***Ack*** and commits the transaction.
  - Cohort 3 (in state r_3) fails, then both the coordinator and cohort 3 (upon recovery) abort the transaction.
- The result is an inconsistent outcome for the transaction.
- Adding b_i ($i \neq 1$), we ensure that no state has both ***Abort*** and ***Commit*** states in its concurrency set.

Reference

- Material for the topic from:
  - "Fault Tolerance in Distributed Systems" by Pankaj Jalote, Prentice Hall. Chapter 4 – Broadcast.
  - "Advanced Concepts in Operating Systems" by Singhal and Shivaratri, McGraw Hill. Chapter 8 – Agreement.
  - "Advanced Concepts in Operating Systems" by Singhal and Shivaratri, McGraw Hill. Chapter 13 – Commit Protocols.