CSED601
Dependable Computing
Lecture 15

Jong Kim

Dept. of CSE

POSTECH

Distributed Systems

- Basic building block
  - Node, communication
  - Many assumptions : implicit and explicit
- Frequent assumptions
  - The relationship between the clocks of the different processors
  - The reliability of the communication network
  - The behavior of nodes during failure
  - Stable storage

Byzantine agreement

- Background
  - Belief : When a component fails, it behaves in a certain well-defined manner, though its behavior may be different its failure-free behavior.
  - Symptom : The behavior of failed component is totally arbitrary.
- Problem
  - Reaching an agreement

Byzantine agreement

- Problem description
  - A system with many components in which components exchange information with each other.
  - Obtain a consensus among all non-faulty nodes in this context.
  - Each node has to make a decision based on values it gets from the other nodes in the system.
  - It requires all non-faulty nodes get the same set of values.
  - Complication: a faulty node may send different values to different nodes.
  - Every non-faulty node in the system uses the same value for a node j for decision making.
  - The general problem of consensus is reduced to agreement by nodes in a system on the value for a particular node.

Byzantine agreement

- Formal statement on requirements
  - All non-faulty nodes use the same value v(i) for a node i.
  - If the sending node I is non-faulty, then every non-faulty node uses the value I sends.
- Other facts
  - Also known as, Interactive consistency problem
  - The need of information that is received by others
  - The problem can be solved only if the # of faulty nodes in the system is limited.
  - If the system is asynchronous, the agreement is impossible if even one processor can fail.
    - No progress → cannot distinguish slow and no progress.

Byzantine agreement

- Two scenarios
  - One of receiving node is faulty, sender sends it a 1, it transmits to node i that a 0 was sent by the sender.
  - Sender is faulty and it send a 1 to node i and a 0 to node j.
  - Two scenarios are indistinguishable.
  - Conclusion ➔ With ordinary messages it is impossible to solve this problem unless more than 2/3 of the nodes are non-faulty
- Simple solution
  - Message signing.
    - When a non-faulty sender sends a message to other nodes, a faulty node cannot tamper with its message and forward it to other nodes.

Byzantine agreement

- Protocols with ordinary message
  - Consensus can be reached in the n faulty nodes if the total number of nodes is at least 3n + 1
  - Assumptions
    - Every message that is sent by a node is delivered correctly by the message system to the receiver.
    - The receiver of a message knows which node has sent the message
    - The absence of a message can be detected.
    - No interference of other messages
    - A faulty node cannot masquerade as another node
    - Cannot fail the consensus attempt as required by the protocol.
  - Algorithm by Lamport, Shostak and Pease in [CSP82]

Byzantine agreement

- Protocols with ordinary message
  - Algorithm by Lamport, Shostak and Pease in [CSP82]
    - Idea : Each receiver has to communicate the value it receives from the transmitter to others.
    - Algorithm ICA(0)
      - The transmitter sends its value to all other n-1 nodes
      - Each node uses the value it receives from the transmitter, or uses the default value, if it receives no value.
    - Algorithm ICA(m)
      - The transmitter sends its value to all the other n-1 nodes
      - Let v(i) be the value the node I receives from the transmitter, or else be the default value if it receives no value. Node I acts as the transmitter in algorithm ICA(m-1) to send the value v(i) to each of the other n-2 nodes.
      - For each node i, let v(j) be the value received by the node j (i <> j). Node I uses the value majority (v(1), v(2),…, v(n-1))

Byzantine agreement

- Protocols with Signed message
  - The problem of messages tampering make it more complex
  - What about the blocking of message tampering
  - Algorithm by Lamport [lsp82]
    - Send message with signature
    - Received message is forwarded with adding signature

Practical Byzantine Fault Tolerance

Slides from MIT

System Model

- Asynchronous distributed system where nodes are connected by a network
- Byzantine failure model
  - faulty nodes behave arbitrarily
  - independent node failures
- Cryptographic techniques to prevent spoofing and replays and to detect corrupted messages
- Very strong adversary

Service Properties

- Any deterministic replicated service with a state and some operations
- Assuming less than one-third of replicas are faulty
  - safety (linearizability)
  - liveness (assuming delay(t) >> t)
- Access control to guard against faulty client
- The resiliency (3f+1) of this algorithm is proven to be optimal for an asynchronous system

The Algorithm

- Basic setup:

  - $|\mathcal{R}| = 3f + 1$

  - A view is a configuration of replicas (a primary and backups): $p = v \ mod \ |\mathcal{R}|$
  - Each replica is deterministic and starts with the same initial state
  - The state of each replica includes the state of the service, a message log of accepted messages, and a view number

The Algorithm

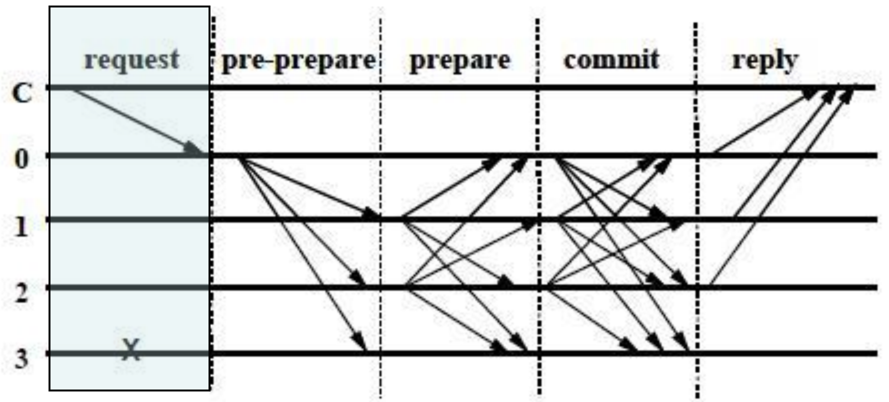

Figure 1: Normal Case Operation

- 1. A client sends a request to invoke a service operation to the primary

$$\langle \text{REQUEST}, o, t, c \rangle_{\sigma_c}$$

o= requested operation
t= timestamp
c= client
σ= signature

The Algorithm

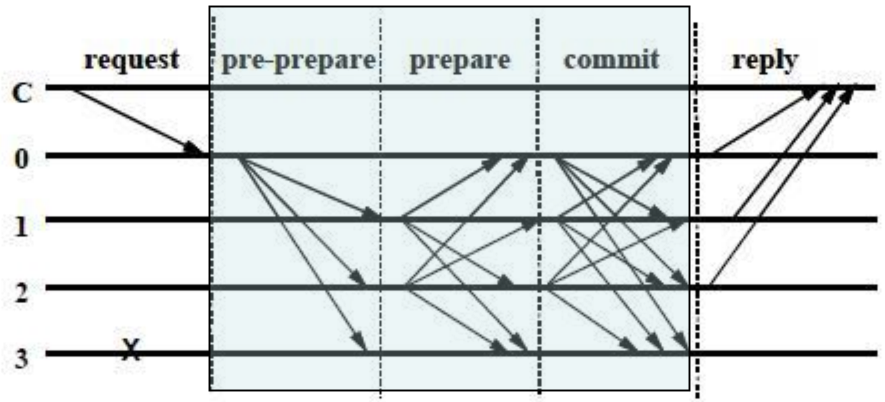

Figure 1: Normal Case Operation

- 2. The primary multicasts the request to the backups (three-phase protocol)

The Algorithm

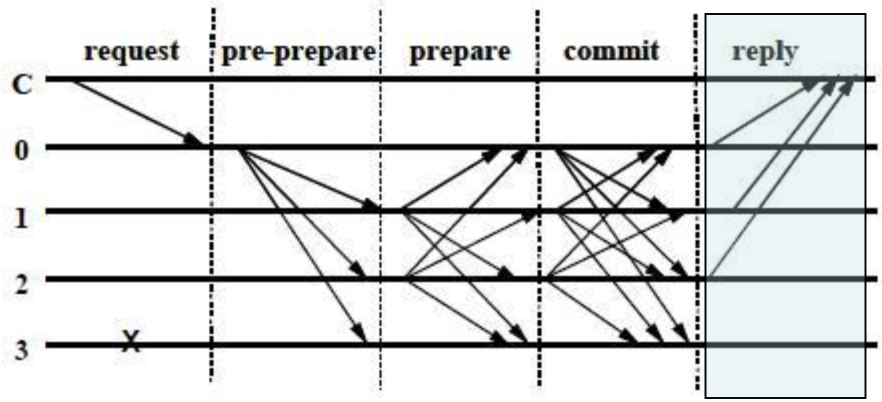

Figure 1: Normal Case Operation

- 3. Replicas execute the request and send a reply to the client

$$\langle \text{REPLY}, v, t, c, i, r \rangle_{\sigma_i}$$

o= requested operation
t= timestamp
c= client
σ= signature

v= view
i= replica
r= result

The Algorithm

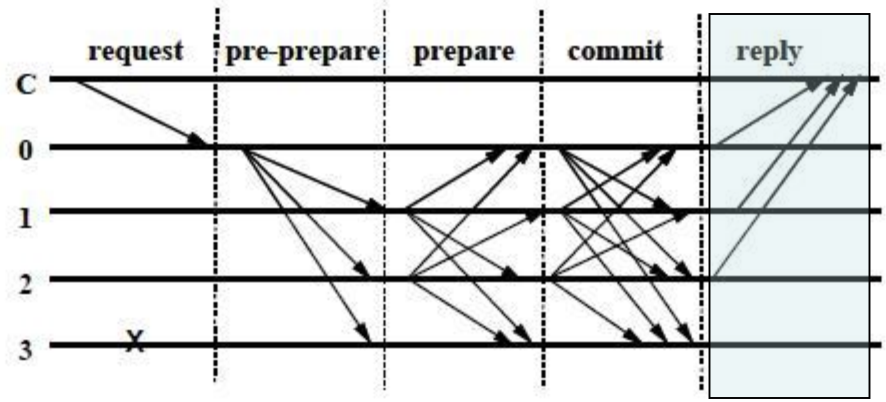

Figure 1: Normal Case Operation

- 4. The client waits for f+1 replies from different replicas with the same result; this is the result of the operation

Three-phase Protocol

- 1.pre-prepare
  - primary assigns n to the request; multicasts pp
  - request message m is piggy-backed (request itself is not included in pp)
  - accepted by backup if: $\langle\langle \text{PRE-PREPARE}, v, n, d\rangle_{\sigma_p}, m\rangle$
    - the messages are properly signed;
    - it is in the same view v;
    - the backup has not accepted a pp for the same v and n with different d
    - h <= n <= H
  - if accepted, then replica i enters prepare phase

Three-phase Protocol

- 2.prepare
  - if backup accepts pp, multicasts p
  - accepted by backup if: $\langle \text{PREPARE}, v, n, d, i \rangle_{\sigma_i}$
    - message signature is correct;
    - in the same view;
    - h<= n<= H
  - prepared(m,v,n,i) is true if i has logged:
    - request message m
    - pp for m in v
    - 2f matching prepares with the same (v,n,d)
  - if prepared becomes true, multicasts commit message and enters commit phase

Three-phase Protocol

- Pre-prepare – prepare phases ensure the following invariant:

  - *if prepared(m,v,n,i) is true then prepared(m',v,n,j) is false for any non-faulty replica j (inc. i=j) and any m' such that D(m') != D(m)*

- i.e. ensures requests in the same view are totally ordered (over all non-faulty replicas)

Three-phase Protocol

- 3.commit
  - accepted by backup if: $\langle \text{COMMIT}, v, n, D(m), i \rangle_{\sigma_i}$
    - message signature is correct;
    - in the same view;
    - h<= n<= H
  - committed(m,v,n) is true iff prepared(m,v,n,i) is true for all i in some set of f+1 non-faulty replicas
  - committed-local(m,v,n,i) is true iff prepared(m,v,n,i) is true and i has accepted 2f+1 matching commits
  - replica i executes the operation requested by m after committed-local(m,v,n,i) is true and i's state reflects the sequential execution of all requests with lower n

Three-phase Protocol

- Commit phase ensures the following invariant:

  - *if committed-local(m,v,n,i) is true for some non-faulty i then committed(m,v,n) is true*
- i.e. any locally committed request will eventually commit at f+1 or more non-faulty replicas
- The invariant and view change protocol ensure that non-faulty replicas agree on the sequence numbers of requests that commit locally even if they commit in different views at each replica
- Prepare – commit phases ensure requests that commit are totally ordered across views

The Algorithm

- The algorithm provides safety if all non-faulty replicas agree on the sequence numbers of requests that commit locally
- To provide liveness, replicas must change view if they are unable to execute a request
  - avoid view changes that is too soon or too late; the fact that faulty replicas can't force frequent view changes will guarantee liveness unless message delays grow faster than the timeout period indefinitely

Consistent State

Recovering a Consistent State

- Fault recovery
  - When a process stops due to a fault, recovery is required.
  - Recovery to a consistent state before the fault occurs.
  - Checkpointing of running process is used to save a consistent state of the process.
  - Recovery (rollback) is a restoration of the saved consistent state.

Checkpointing

- Types of checkpointing
  - Single process checkpointing
    - Used for long-running scientific process
    - Issues
      - How to minimize the information to be saved with having a consistent state.
        - » Incremental checkpointing
        - » User-defined checkpointing
      - How frequently the checkpoint to be performed.
        - » Periodic checkpoiting
        - » Program induced checkpointing

Checkpointing

- Types of checkpointing
  - Multi process checkpointing
    - Need a global consistency on all communicating multi processes
    - Have a problem of rollback & domino effect
      - Messages require consistency.
      - Lost messages : messages before failure will be disappeared by the failure
      - Orphan messages : some messages do not have an origin.
    - Two approaches
      - Coordinated checkpointing / Synchronous checkpointing
      - Independent checkpointing / Asynchronous checkpointing

Coordinated checkpointing

- Idea
  - When one process is to checkpoint, it requests other processes to participate the checkpointing.
  - Request message itself is a communication load.
  - Use a synchronization primitive for coordination.
  - Issues
    - How to minimize the number of processes to be checkpointed together.
    - How to reduce the synchronization load.
  - Two phase commit algorithm
    - A starting process P1 does a tentative checkpoint and requests other processes to do a tentative checkpoint.
    - Processes received a tentative checkpoint request does a tentative checkpoint if P1 $\leftarrow$ P2 and report to the requesting process.
    - If all processes do the tentative checkpont, request to convert it to permanent.

Coordinated checkpointing

- Idea
  - Checkpointing using synchronized clock
    - Clock difference is β and communication delay δ.
    - Choose a period larger than $\beta + \delta$
    - Can have an orphan message
      - Block to send a message for β time interval after checkpoint.
      - Do an earlier checkpoint
    - Can have a lost message
      - Use a message logging

Independent checkpointing

- Why?
  - Can reduce a load in checkpoint
- Problems
  - Messages
    - Lost message / Orphan message
  - Domino effect to recover communication
- Solution?
  - Lost message : Use logging
  - Logging types
    - Optimistic message logging : asynchronous no-wait message logging, but will increase the rollback distance.
    - Pessimistic message logging : process is stopped while logging messages. Recovery is easier. But will have a delay in execution.
    - Sender-based logging
    - Receiver-based logging
  - How to reduce the number of writes in stable storage
    - Logging on volatile storage and do write on a stable storage when checkpoint

Independent checkpointing

- Solution for orphan message?
  - Prevention of orphan message
  - How to?
    - Detect an inconsistent checkpoint state when a message is received.
    - Insert a checkpoint to avoid an inconsistent checkpoint.
  - How to find an inconsistent checkpoint state?
    - If there is a zig-zag cycle, the system has an inconsistent state.
    - If there is a zig-zag path on checkpoint C to itself, the checkpoint C is involved in a zigzag cycle.

Independent checkpointing

- Solution for orphan message?
  - How to know checkpoint state of others?
    - Use piggyback
      - Carry checkpoint index number
      - Carry time stamp
      - Carry other processes' checkpoint status
  - How to find a recovery line