6. Replication

TFD

Outline

- Replication Management in Partition-Free Systems
  - Active Replication (State Machine Replication)
  - Passive Replication (Primary-backup approach)
    - Weak Passive Replication
    - Strong Passive Replication
  - Semi-Active Replication
  - Lazy Replication
- Replication Management in Partitionable Systems
  - Primary partition
  - Quorum Systems
    - ROWA
    - Majority Quorums
    - Weighted Quorums

TFD

Replicated computations

- Client-server model
- Service provided is replicated in several servers…
- …but client remains under the illusion that it contacts a single server
- If replicas fail independently, the service tolerates these faults (to a certain extend)

- Clearly, replicated computations can be considered under different fault and synchrony models
  – The specific protocols, and the needed replication degree depend of the considered fault model

Replicated computations

- Fault-tolerant applications may run replicated pieces of code which should behave in the same way
- Replica determinism:
  – Two replicas, starting in the same initial state and subject to a same sequence of inputs reach the same final state and produce the same sequence of outputs
- Atomic broadcast:
  – Ensures Validity, Integrity, **Agreement** and **Total Order**
  – Guarantees “same sequence of inputs” objective
  – The rest lies with the replica itself

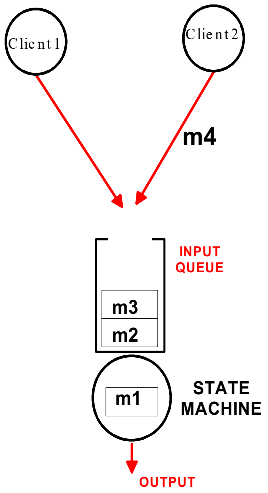
Replicated Computations
(issues)

- State of two non-deterministic replicas may diverge even when they execute same sequence of inputs
- Non-deterministic component
  - State and behavior depend not only on the sequence of commands it executes but also on local parameters that cannot be controlled
- Many mechanisms can lead to non-determinism:
  - Resource sharing with other processes
  - Multithreading scheduling
  - Readings from clocks or random number generators
  - Non-deterministic constructs in programming languages (e.g., Ada 'select' statement)

Replication management
(partition-free systems)

Replicas as State Machines

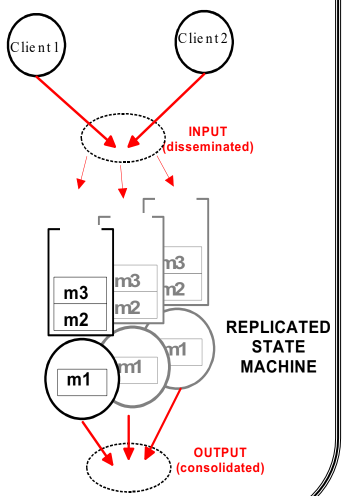
- Execution model
  - state = set of state variables
  - commands modify state variables and produce outputs (I/O or return results)
- Characteristics
  - confinement - atomic commands
  - fault tolerance - easy replication (why?)



Replicated State Machine

(active replication or state machine approach)

- Replicated state machine:
  - Servers start in same **state**
  - All replicas execute same sequence of input commands, in same order
  - THEN: all follow same sequence of state/outputs
  - Achieves error masking
  - Determinism is mandatory
- Message ordering:
  - Total order of commands to replicas
  - Same commands in same order => same results



Replicated State Machine
(**active replication** or **state machine approach**)

- Requirements of the problem:
  - **SMA1 Initial state -** All servers start in the same state
  - **SMA2 Agreement -** All servers execute the same commands
  - **SMA3 Total order -** All servers execute the commands in the same order

Replicated State Machine
(**active replication** or **state machine approach**)

- Properties of a Replicated State Machine:
  - **Safety:** all correct replicas execute the same sequence of operations
    - This means strong consistency, or Linearizability, in which a replicated system perfectly simulates a non-replicated one
  - **Liveness:** all correct clients operations are executed
    - Some protocols explicitly assume that this property is valid only if the environment is “synchronous enough” (why?)

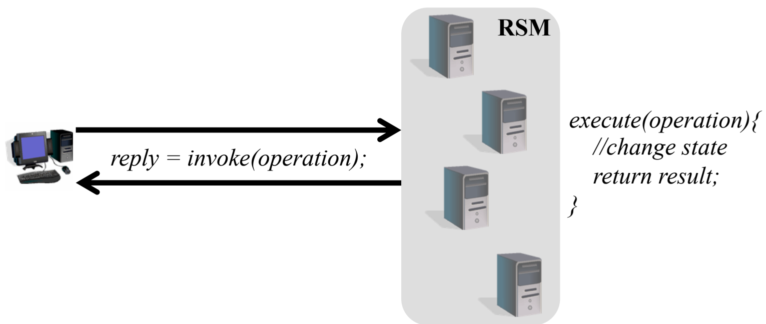
Replicated State Machine

(**active replication** or **state machine approach**)

- Client-servers protocol
  - Send to all servers; then all servers broadcast (using atomic/total order broadcast) OR
  - Send to one server; then server atomically broadcasts; if no reply obtained in a timeout, send to *f* other servers etc.
  - After executing the commands, one or more servers send output to client that consolidates it, i.e., votes

Replicated State Machine

(**active replication** or **state machine approach**)



- In summary:
  - It's a basic RPC model
  - Servers do not initiate communication
  - Usually, read-only operations are implemented and invoked in separate calls

Replicated State Machine

(active replication or state machine approach)

- How many servers do we need?
  - Constrained by the requirements of the atomic broadcast algorithm (system model) and the fault model

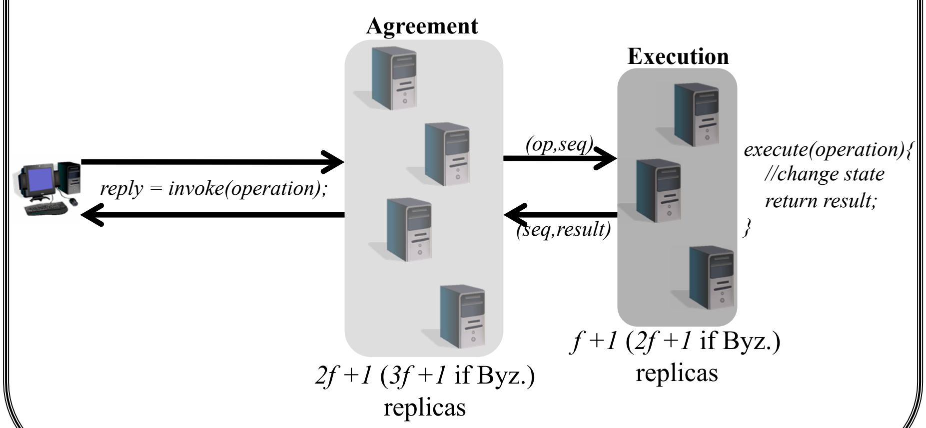
	Ordering		Execution
Failure type	Synchronous	Partially synchronous	Replicated Execution
Fail-stop	$f+1$	$2f+1$	$f+1$
Authenticated Byzantine	$f+1$	$3f+1$	$2f+1$
Byzantine	$3f+1$	$3f+1$	$2f+1$

Byzantine faults with digital signatures.

Replicated State Machine

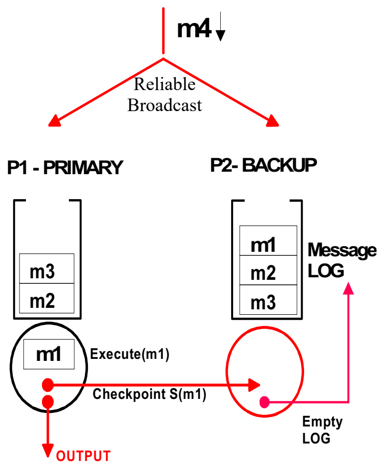
(active replication or state machine approach)

- Separation of agreement and execution



Replicated State Machine
(passive replication or primary-backup approach)

- **(Weak) Passive replication**
  - Only Primary executes (it decides the order)
  - Uses leader-election algorithm (no atomic broadcast is needed)
  - Supports preemption by urgent commands and non-determinism (active replication does not)
- State transferred to Backup(s)
  - Inter-replica deferred state-level synchronization (**checkpoints, stored in shared space or sent to backups**)
  - Backups log commands until a checkpoint is received
  - Primary fails: Backup takes over
  - Potentially long takeover-glitch
- Message ordering:
  - Non-ordered message diffusion



Replicated State Machine
(passive replication or primary-backup approach)

- Alternative algorithm (**Strong passive replication**):

// Part executed by the primary only
Upon receiving a request from a client
Execute the request
Forward the state update (if any) to the backups
Send reply to the client

Only *f+1* servers needed for leader election, if system is synchronous or there is a **perfect failure detector**

//part executed by the backups only
Upon receiving a request
Store it in the log // allows error masking
Upon receiving a state update
Update the state and clean the corresponding request from the log
Upon detection of faulty primary
Elect a new primary
If it is the new primary, execute and reply to any requests in the log

Zookeeper?

Replicated State Machine

(**passive replication** or **primary-backup approach**)

- How to make this algorithm work on non-synchronous systems?

// Part executed by the <u>primary</u> only
Upon receiving a request from a client
Execute the request
Forward the state update (if any) to the backups
Send reply to the client

With $2f+1$ servers for atomic multicast and leader election, doesn't require perfect failure detection, only the **eventually strong FD**

//part executed by the <u>backups</u> only
Upon receiving a request
Store it in the log // allows error masking
Upon receiving a state update
Update the state and clean the corresponding request from the log
Upon detection of faulty primary
Elect a new primary **(HOW?)**
If it is the new primary, execute and reply to any requests in the log

Replicated State Machine

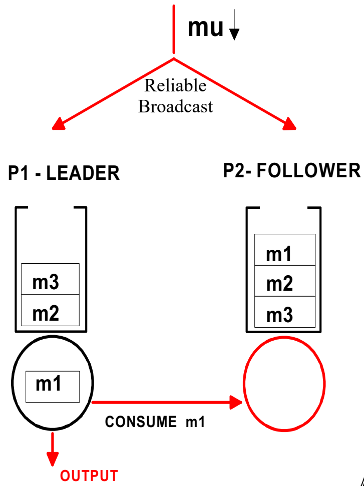
(**passive replication** or **primary-backup approach**)

- Fault model is important:
  - If the network is reliable (i.e., if primary sends *m* to backup and the backup eventually receives *m*, even if the primary fails), the algorithm is non-blocking (reply to client sent without delay)
    - This is assumed in previous slide, together with FIFO links
  - If the network commits omission faults (state update messages may get lost), the primary must make sure that the backup(s) receive(s) the state update, before a reply is sent to the client
    - Explicit confirmation from backups to primary (two round trip delays)
    - Alternative: backup sends reply to client after receiving update (one round trip delay, but more complex to implement)

RAFT?

Replicated State Machine
(semi-active replication)

- Semi-active replication
  - All execute, but the leader executes first
    - In the order it decides
  - Supports per-message deterministic preemption, e.g., urgent command message m_i overtakes (preempts) others
  - Leader instructs preemption point to Followers
- Commands transferred to Follower(s)
  - Leader informs Followers of next command or command sequence
  - Leader fails: Follower takes over
  - Short takeover-glitch
- Message ordering:
  - Non-ordered message diffusion (but needs a leader-election algorithm)



Lazy replication

- Also called **optimistic replication**
  - If everything goes well, replicas will eventually become consistent
- Trades some consistency for performance
  - Client requests can be made to different replicas
  - Replicas are allowed to diverge
  - Strong consistency is only guaranteed when the systems is idle
  - Guarantee: weak or eventual consistency
- What about conflicts?
  - Using knowledge about the semantic of the service
    - Vector clocks can be used to ensure causal ordering of commands
    - Synchronization among replicas when total order is needed
  - Causal requests are propagated in background

Optimizations for reads

- Active Replication
  - Reading from any single replica may return stale values
  - Reading from the leader may return stale values (how?)
  - Reading from a (majority) quorum, without atomic broadcast, may return updated and stale values (how? How to detect?)
- Passive Replication
  - Reading from a backup (if possible) may return significantly stale values (e.g., from the last installed checkpoint)
- Semi-active Replication
  - Reading from a follower may return stale values
- Staleness can bring noticeable performance gains…

These optimizations are fundamental in practical systems

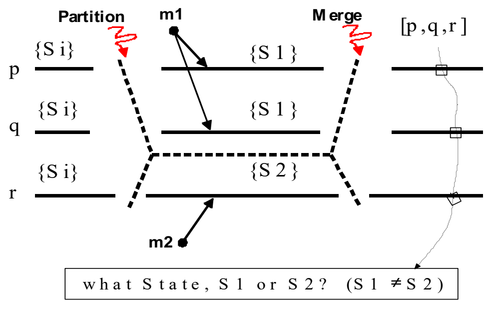
Replication management

(partitionable systems)

Rationale

- One should ensure a consistent service even when some replicas become mutually unreachable
  - For networks where partitions can occur

State Divergence with Partitioning

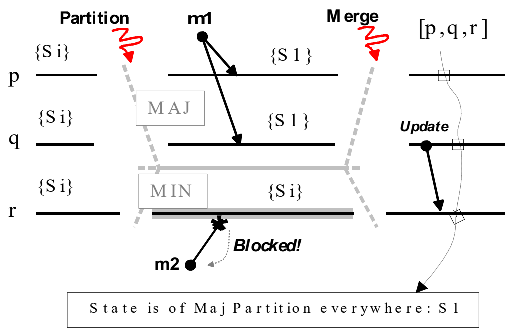


- Partitioning occurs, *p* and *q* execute command *m1* and assume state *S1*
- *r* executes command *m2* and assumes state *S2*
- What is the system state after a merge?
  - E.g., if *m1* and *m2* produced conflicting results, it is impossible to find a coherent common state without application-dependent reconciliation

Replicated computations
(issues)

- State divergence with partitioning
  - With partitioning, cliques of replicas may mutually think they are dead, and continue computations independently
- One way to solve this is to ensure computations only proceed in a primary partition
  - Consistent but also less available way
- Another common way is to gather votes or quorums of a minimum number of replicas that guarantee progress, potentially losing some consistency
  - Nothing to do with value masking but with progress
- It is important to remember that keeping all of it (Consistency, Availability, Partition tolerance) is impossible
  - **Brewer's CAP Theorem**

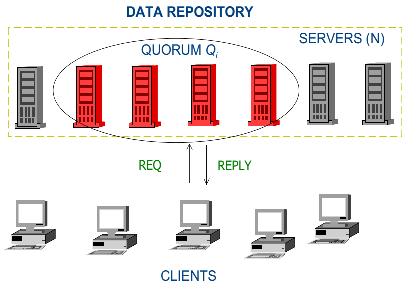
Avoiding State Divergence
primary partition



- As before, but now only primary partition continues executing
- Primary partition has majority of replicas, i.e., *p* and *q*
- *r* stays blocked in state *Si*
- *p* and *q* continues, processing *m1*, and reaching state *S1*
- After a merge, *r* requests state update to *p* and *q*
- Since *Si* (of *r*) is a prefix of *S1*, there is no divergence

Static voting and quorums

- An operation should only be allowed to proceed if a minimum quorum of replicas can perform it
- Quorum formation rules:
  - Two conflicting operations must always intersect in at least one correct replica
  - This common replica ensures the outcome of first operation is seen by any quorum of replicas executing the next operation
  - Most up to date state is identified by version numbers (incremented by each replica upon an update)



Quorum algorithms
(read-one write-all)

- Read-one write-all (ROWA)
  - *n* the total number of replicas
  - *r* and *w* the quorums required for read and write operations
  - Read operations are allowed to read any single replica ($r = 1$)
  - Write operations are required to write all replicas ($w = n$)
- Advantages:
  - Read operations with a high degree of availability
  - Write operations with a high degree of consistency
- Disadvantages:
  - Write operations with a potentially low degree of availability, since they cannot be executed after failure of a single copy (unless the system reconfigures)
- Variant: Read-one write-all-available
  - Require a perfect failure detector, or may lose consistency

Quorum algorithms
(majority quorums)

- Majority quorums for asynchronous systems
  - *n* the total number of replicas
  - *r* and *w* the quorums required for read and write operations
  - Read and write operations are required to access any majority of replicas ($w = r > n/2$)
- Advantages:
  - Both operations require a primary partition to be successful
  - Does not require synchrony or system reconfigurations to work
- Disadvantages:
  - Operations only complete on majority partitions

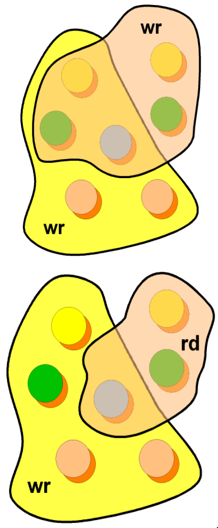
Quorum algorithms
(weighted voting)

- Each copy is assigned a number of votes
- Quorums are defined based on the number of votes instead of the number of replicas
- Overlapping guarantee rule: $2w > n$ and $w + r > n$
- Why?
  - *n*, the total number of votes
  - Sum of quorums for conflicting operations on an item should exceed the total number of votes for that item (to yield a common replica)

Quorum algorithms
(Weighted voting intuition)

n=7, w=5, r=3

- 2w > n
  - Suppose n = 7, w = 5 and r = 3
  - Write to partition containing replicas summing at least 5 votes
  - 2 votes left, not enough to write divergently in other partitions
- w + r > n
  - Reads and writes to same item, different partitions, are serialized
  - E.g., write occurs first (5 votes), so read must wait (2 votes left, read needs 3 votes)
  - So, read is sure to include at least one of the replicas that have seen previous write
  - This replica can update the others, ensuring sequential consistency of the history of operations



Quorum algorithms
(Weighted voting intuition)

- Advantages of the separation between the number of replicas and the number of votes:
  - Careful vote assignment taking into account node/network properties may yield improved availability and performance
  - Example:
    - Replica *A* runs in a node with better connectivity and/or reliability is given 3 votes while all other four replicas are given 1 vote
    - Partition with *A* and any other replica secures the majority of votes, allowing progress even if a majority of replicas fail or partition