TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Programming Distributed Systems

7 Consensus

Annette Bieniusa

FB Informatik
TU Kaiserslautern

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Motivation

- Replication is a core problem in distributed systems[2, Sec 15.1-15.3]
- Why do we want to replicate services or data?
  - **Performance**: If there are many clients issuing operations, a single process might not be enough to handle the whole load with adequate response time. Further, keeping data close to clients reduces the network latency when handling requests.
  - **Availability**: Despite server failures and network partitions, clients can still interact with the system (potentially operating with stale or conflicting data).
  - **Fault-tolerance**: Despite faults, the systems continues to behave correctly; e.g. it does not loose information.
- We can replicate computations and **state** (focus of this lecture)

Goals of this Learning Path

In this learning path, you will learn how

- to classify replication strategies
- to model replicated data storage systems as replicated state machines
- to reduce total-order broadcast to consensus (and vice versa)
- to argue about the impossibility of reaching consensus in asynchronous systems with crash-faults
- to use quorum systems to implement consensus algorithms
- to implement fault-tolerant consensus for replicated state machines using the Raft algorithm

State Machine Replication

State Machine Replication[10]

- Generic model for replicated services
- A state machine has a state S and a set of commands/requests/operations $Ops = \{Op_1, Op_2, \dots\}$ that
  - potentially take some input and/or
  - transform the state deterministically and/or
  - return some response
- Clients invoke operations from the set Ops on the service
- The process implementing the state machine is replicated, i.e. there are multiple copies / instances of the same process.

Replication Algorithm

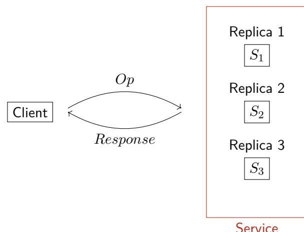
A replication algorithm is responsible for managing the multiple replicas of a state machine

- under a given fault model
- under a given synchronization model

In essence, the replication algorithm will enforce properties on the effects of operations observed by clients given the evolution of the system (potentially including the evolution the clients).

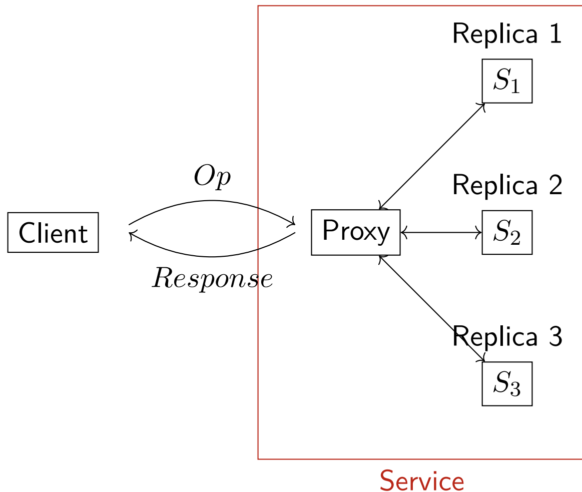
Desirable properties: Transparency + Consistency

- Clients should not be aware that multiple replicas (might) exist.
- When interacting with the system, a client should only observe a single logical state.
- The behavior of this logical state must be in accordance with its correctness specification.

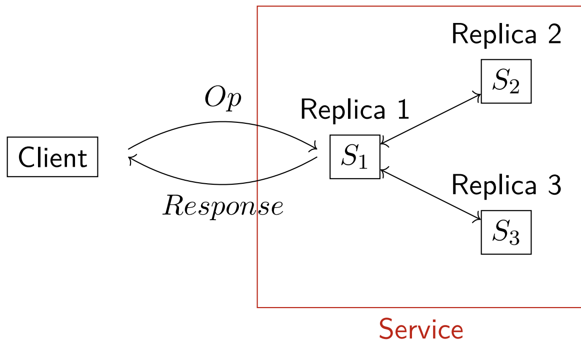


$\Rightarrow$ Need to restrict the state that can be observed by a client!

Option 1: Coordinating proxy



Option 2: One of the replicas interacts with the client



Replication strategies

- **Active Replication:** Operations are executed by *every* replica.
- **Passive Replication:** Operations are executed by a *single* replica, results are shipped to other replicas.

- **Synchronous Replication:** Replication takes place *before* the client gets a response.
- **Asynchronous Replication:** Replication takes place *after* the client gets a response.

- **Single-Master**: A *specific replica* receives operations from clients.
- **Multi-Master**: *Any replica* can process operations from clients.

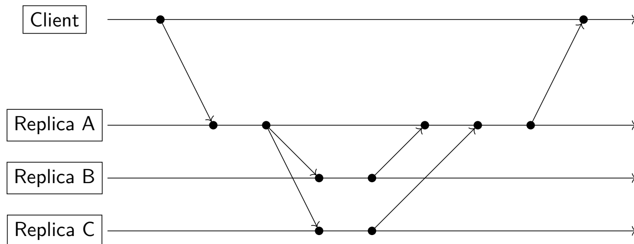
Active Replication

- All replicas execute operations.
- State is continuously updated at every replica
  - Lower impact of a replica failure
- Can only be used when operations are deterministic
  - i.e. not dependent on non-deterministic input, such as local time or randomly generated values
- If operations are not commutative (i.e., execution of the same set of operations in different orders lead to different results), then all replicas must agree on the order in which operations are executed.

Passive Replication

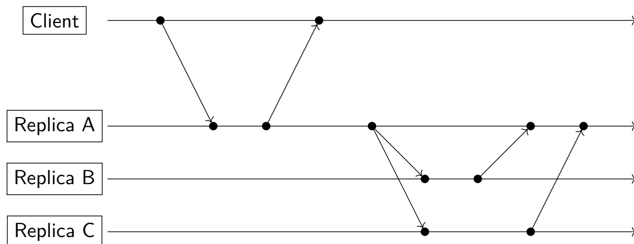
- Required when operations depend on non-deterministic data or inputs
- Load across replicas is not balanced
  - Only one replica effectively executes an operation and computes the result
  - Other replicas only observe results to update their local state

Synchronous Replication



- Strong durability guarantees
  - Tolerates faults of $N-1$ servers
- Request will be served as fast as the slowest server
- Response time is bound by network latency

Asynchronous replication



- One replica immediately sends back response and propagates the updates later
- Client does not need to wait
- Tolerant to network latencies
- *Problem:* Data loss if replica A goes down before forwarding the update!

Single-master (~~Master-slave~~, Primary-backup, Log Shipping)

- Only a single replica, called the master/leader/coordinator, processes operations that modify the state.
- Other replicas can process client operations that only observe the state.
- Problems:
  - Clients might observe stale values
  - Susceptible to lost updates or incorrect updates if nodes fail at inopportune times
- When the master fails, another node has to take over the role of master.
- If two processes believe themselves to be the master, safety properties might be violated.

Multi-master Systems

- Any replica can process any operation (i.e, both read and update operations).
- All replicas have the same role $\Rightarrow$ Better load balancing
- *Problem:* Divergence
    - Multiple replicas might attempt to perform conflicting operations at the same time
    - Requires coordination (e.g. distributed locks or other coordination protocols)

On the Equivalence of Total-order Broadcast and Consensus

Preventing divergence in multi-master systems

- *Idea*: Execute all operations in the same order on all replicas

⇒ **Total-order broadcast** (aka Atomic broadcast)

Preventing divergence in multi-master systems

- *Idea*: Execute all operations in the same order on all replicas

$\Rightarrow$ **Total-order broadcast** (aka Atomic broadcast)

Properties of Total-Order Broadcast

- *Validity*: If a correct process to-broadcasts message m, then it eventually to-delivers m.
- *Agreement*: If a correct process to-delivers message m, then all correct processes eventually to-deliver m.
- *Integrity*: For any message m, every process to-delivers m at most once, and only if m was previously to-broadcast.
- *Total order*: If some process to-delivers message m before message m', then every process to-delivers m' only after it has to-delivered m.

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Implementing Atomic Broadcast

We rely on the **consensus** abstraction to implement total-order broadcast.

- Each process p_i has an initial value v_i ($propose(v_i)$).
- All processors have to agree on common value v that is the initial value of some p_i ($decide(v)$).

Properties of Consensus

- *Uniform Agreement*: Every correct process must decide on the same value.
- *Integrity*: Every correct process decides at most one value, and if it decides some value, then it must have been proposed by some process.
- *Termination*: All processes eventually reach a decision.
- *Validity*: If all correct processes propose the same value v, then all correct processes decide v.

Total-Order Broadcast using Consensus: Idea

- Every process executes sequence of consensus problems, numbered 1, 2, . . .
- Initial value for each consensus for process p is the set of messages received by p that have not been to-delivered, yet
- msg^k is the set of messages decided by consensus numbered k
  - Each process to-delivers the messages in msg^k before the messages in msg^{k+1}
  - More than one message may get to-delivered by one instance of consensus!
    - Need to ensure deterministic order to-delivery for messages in msg^k

Atomic Broadcast using Consensus: Algorithm

```
State:
  k                // consensus number
  delivered        // messages to-delivered by process
  received         // messages received by process

Upon Init do:
  k <- 0;
  delivered <- ∅;
  received <- ∅;
Upon to-broadcast(m) do
  trigger rb-broadcast(m);
Upon rb-deliver(q, m) do
  if m ∉ received then received <- received ∪ {m};
Upon received \ delivered ≠ ∅ do
  k <- k + 1;
  undelivered <- received \ delivered;
  propose(k, undelivered);
  wait until decide(k, msg^k)
  ∀ m in msg^k in deterministic order do trigger to-deliver(m)
  delivered <- delivered ∪ msg^k
```

Equivalence of Total-Order Broadcast and Consensus

- As the previous algorithm shows, we can implement Total-Order Broadcast using Consensus.
- Similarly, we can build Consensus using Total-Order Broadcast ($\Rightarrow$ Exercise).

Consensus and Total-Order Broadcast are equivalent problems in a system with reliable channels.

Consensus in the Asynchronous System Model

The Consensus Problem in “Real Life”

Assume you and your two other flatmates want to hire a fourth person for your shared apartment.

Process:

- Each of you separately interviews the candidate
- Afterwards, you pass each other messages under the door regarding your vote
- If the vote is unanimous, the new flatmate may move in
- Otherwise, you look for a new candidate

But:

- You or your flatmates might leave the apartment for an unspecified amount of time

When can you inform a candidate about your common decision?

Question

How do you solve consensus in

- an asynchronous model
- with crash-stop
- and (at least) one failing process?

Intuition:

- In an asynchronous system, a process p cannot tell whether a non-responsive process q has crashed or is just slow
- If p waits, it might do so forever
- If p decides, it may find out later that q came to a different decision

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

The FLP Theorem [4]

There is no deterministic protocol that solves consensus in an asynchronous system in which a single process may fail by crashing.

- 2001 Dijkstra prize for the most influential paper in distributed computing
- Proof Strategy
  - Assume that there **is** a (deterministic) protocol to solve the problem
  - Reason about the properties of **any** such protocol
  - Derive a contradiction ⇒ Done :)

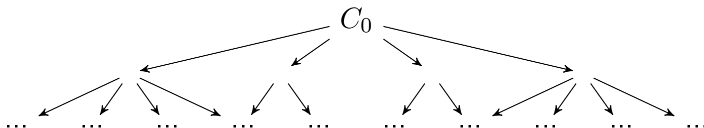
FLP: System model

We will use here a slightly different model that simplifies the proof.

- $N \geq 2$ processes which communicate by sending messages
- Without loss of generality, binary consensus (i.e. proposed values are either 0 or 1)
- Message are stored in abstract *message buffer*
  - $send(p, m)$ places message m in buffer for process p
  - $receive(p, m)$ randomly removes a message m from buffer and hands it to p or hands "empty message" ϵ to p
- This model describes an asynchronous message delivery with arbitrary delay
- Every message is eventually received (i.e. no message loss)

FLP: Configurations

- A **configuration** C is the internal state of all processes + contents of message buffer.
- In each step, one process p
  - performs a $receive(p, m)$, updates its state deterministically, and potentially sends messages (**event** e)
  - or crashes
- An **execution** is a (possibly infinite) sequence of events, starting from some initial configuration C_0.
- A **schedule** S is a finite sequence of events.



FLP: Disjoint schedules are commutative

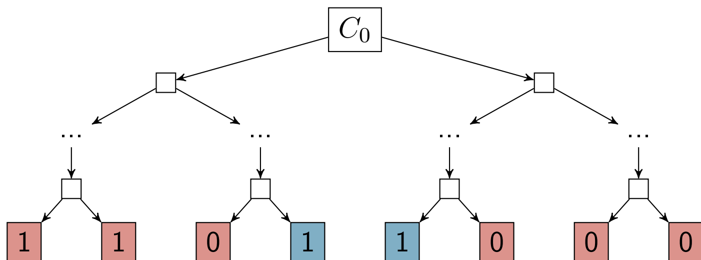
Lemma 1

Disjoint schedules are commutative.

- Schedules S_1 and S_2 are both applicable to configuration C
- S_1 and S_2 contain disjoint sets of receiving processes

FLP: Assumptions

- All correct nodes eventually decide.
- In every config, decided nodes have decided on the same value (here: 0 or 1).

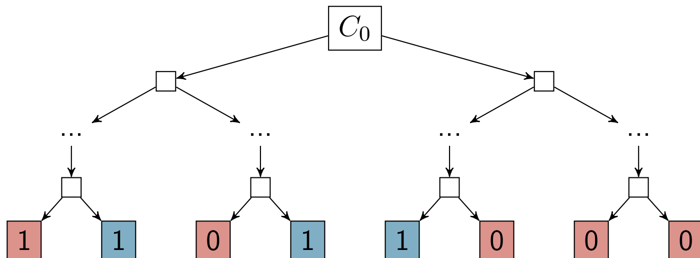


- *0-decided configuration*: A configuration with decision for 0 on *some* process
- *1-decided configuration*: A configuration with decision for 1 on *some* process

FLP: Bivalent Configurations

- *0-valent configuration*: A config in which *every* reachable decided configuration is a 0-decide
- *1-valent configuration*: A config in which *every* reachable decided configuration is a 1-decide

- **Bivalent configuration**: A configuration which can reach a 0-decided **and** 1-decided configuration



FLP: Bivalent Initial Configuration

Lemma 2

Any algorithm that solves consensus with at most one faulty process has at least one bivalent initial configuration.

- This means that there is some initial configuration in which the decision is not predetermined by the proposed values, but is a result of the steps taken and the occurrance of failures.

Proof idea for two processes A and B

- Assume that all executions are predetermined and there is no bivalent initial configuration
- If A and B both propose 0:
  - All executions must decide on 0, including the solo execution by A
- If A and B both propose 1:
  - All executions must decide on 1, including the solo execution by B
- If A proposes 0 and B proposes 1:
  - Solo execution by A decides on 0
  - Solo execution of B decides on 1

$\Rightarrow$ Bivalent initial configuration!

Proof idea for N processes

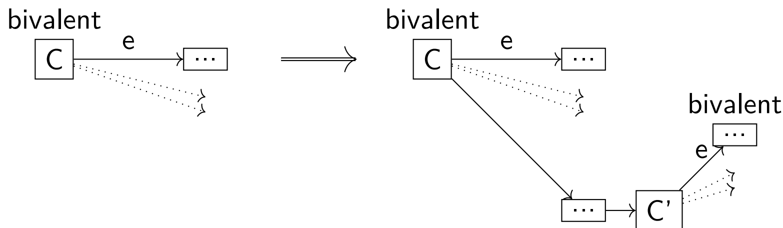
- Assume that all executions are predetermined and there is no bivalent initial configuration
- For N processes, there are 2^N different initial configurations for binary consensus
- Arrange configurations in a line such that adjacent initial configurations only differ by proposed value for *one* process
- There must exist an adjacent pair $C_{0,0}$ and $C_{0,1}$ of 0-valued and 1-valued configurations
  - Let's assume they differ in the proposed value of process p
- Assume that p crashes (i.e. doesn't make steps in the executions)
- Both initial configs will lead to the same configs when applying schedules without p

$\Rightarrow$ $C_{0,0}$ and $C_{0,1}$ are actually bivalent

FLP: Staying Bivalent

Lemma 3

Given *any* bivalent config C and *any* event e applicable in C, there exists a reachable config C' where e is applicable, and $e(C')$ is bivalent.



- If you delay a pending event for some number of steps, there will be a configuration in which you trigger this event and still end up in a bivalent state.

FLP: Proof of Theorem

1. Start in an initial bivalent config. [This configuration must exist according to Lemma 2.]
2. Given the bivalent config, pick an event e that has been applicable longest.
   - Pick the path which takes the system to another config where e is applicable (might be empty).
   - Apply e, and get a bivalent config [applying Lemma 3].
3. Repeat Step 2.

Termination violated.

What now?

- In reality, scheduling of processes is rarely done in the most unfavorable way.
- The problem caused by an unfavorable schedule is transient, not permanent.
- Re-formulation of consensus impossibility:

Any algorithm that ensures the safety properties of consensus can be delayed indefinitely during periods with no synchrony.

Circumventing FLP in Theory

Obviously, by relaxing the specification of consensus ...

- *Idea 1:* Use a probabilistic algorithm that ensures termination with high probability.
- *Idea 2:* Relax on agreement and validity, e.g. by allowing disagreement for transient phases.
- *Idea 3:* Only ensure termination if the system behaves in a synchronous way.

Summary

- Replication is one of the key problems in distributed systems[1].
- Characterization of replication schemes
  - active/passive
  - synchronous/asynchronous
  - single-/multi-master
- Problem: Divergence of replicas
- Total-order Broadcast and Consensus
- FLP Theorem: Impossibility of Consensus in asynchronous distributed systems with crash-stop

Quorum-based Systems

Consensus in Parliament

!

Motivation

- A **quorum** is the minimum number of members of an assembly that is necessary to conduct the business of this assembly.
- In the German Bundestag at least half of the members (355 out of 709) must be present so that it is empowered to make resolutions.

Idea

Can we apply this technique also for reaching consensus in distributed replicated systems?

Problem revisited: Register replication

Registers

- A **register** stores a single value.
- *Here:* Integer value, initially set to 0.

- Processes have two operations to interact with the register: **read** and **write** (aka: put/get).
- Processes invoke operations sequentially (i.e. each process executes one operation at a time).

- Replication: Each process has its own local copy of the register, but the register is shared among all of them.
- Values written to the register are uniquely identified (e.g, the id of the process performing the write and a timestamp or monotonic value).

Properties of a register

Liveness: Every operation of a correct process eventually completes.

Safety: Every read operation returns the last value written.

Properties of a register

Liveness: Every operation of a correct process eventually completes.

Safety: Every read operation returns the last value written.

What does **last** mean?

Properties of a register

Liveness: Every operation of a correct process eventually completes.

Safety: Every read operation returns the last value written.

What does last mean?

Each operation has an start-time (invocation) and end-time (return).
Operation A **precedes** operation B if $end(A) < start(B)$.

We also say: operation B is a subsequent operation of A

Different types of registers (1 writer, multiple readers)

(1,N) Safe register

A register is safe if every read that doesn't overlap with a write returns the value of the last preceding write. A read concurrent with writes may return any value.

(1,N) Regular register

A register is regular if every read returns the value of one of the concurrent writes, or the last preceding write.

(1,N) Atomic register

If a read of an atomic register returns a value v and a subsequent read returns a value w, then the write of w does not precede the write of v.

Different types of registers (multiple writers and readers)

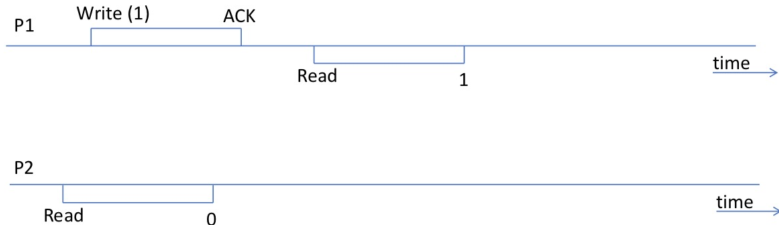
(N,N) Atomic register

Every read operation returns the value that was written most recently in a hypothetical execution, where every operation appears to have been executed at some instant between its invocation and its completion (linearization point).

Equivalent definition: An atomic register is linearizable with respect to the sequential register specification.

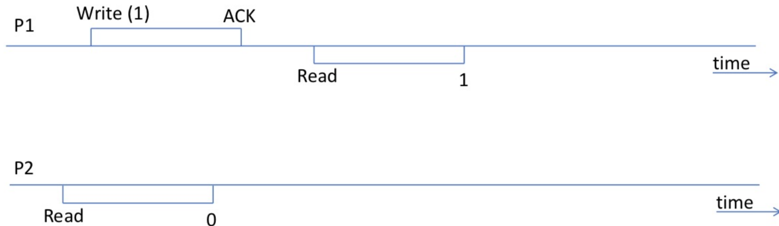
Example execution 1

Is this execution possible for a safe/regular/atomic register?



Example execution 1

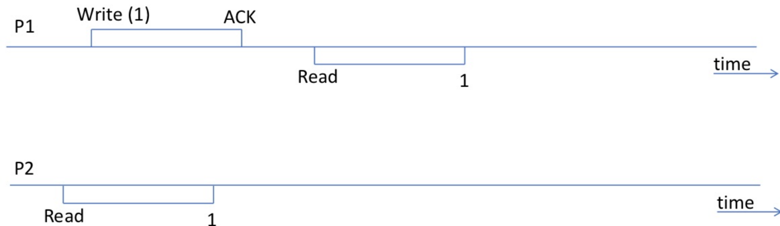
Is this execution possible for a safe/regular/atomic register?



Valid for all!

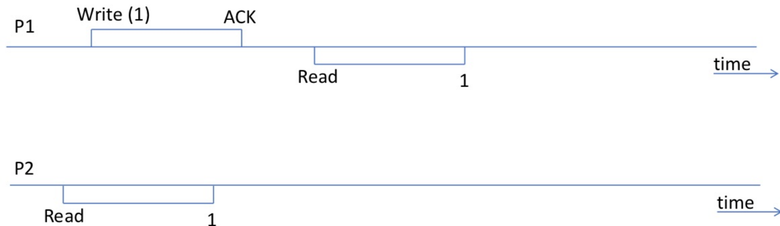
Example execution 2

Is this execution possible for a safe/regular/atomic register?



Example execution 2

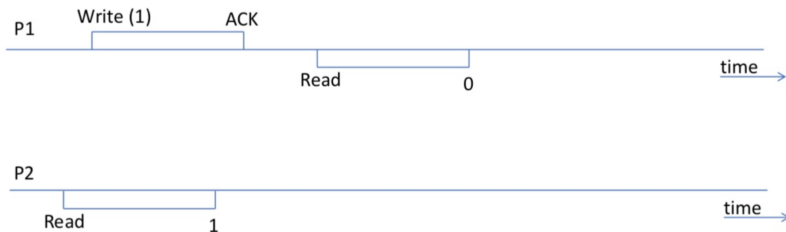
Is this execution possible for a safe/regular/atomic register?



Valid for all!

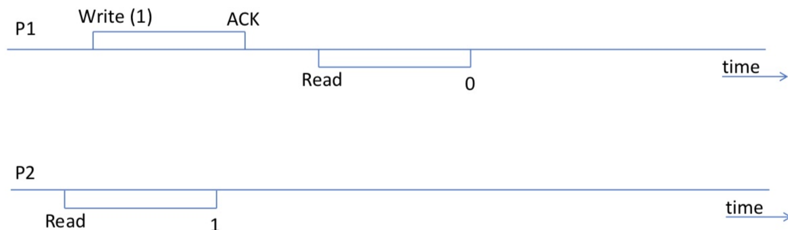
Example execution 3

Is this execution possible for a safe/regular/atomic register?



Example execution 3

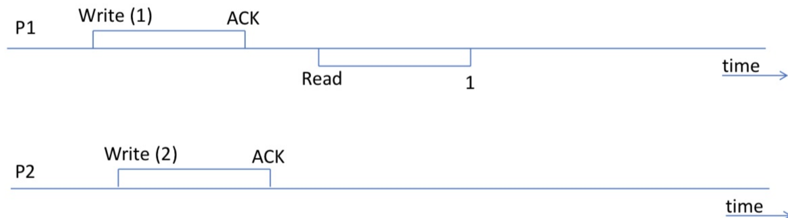
Is this execution possible for a safe/regular/atomic register?



Not valid!

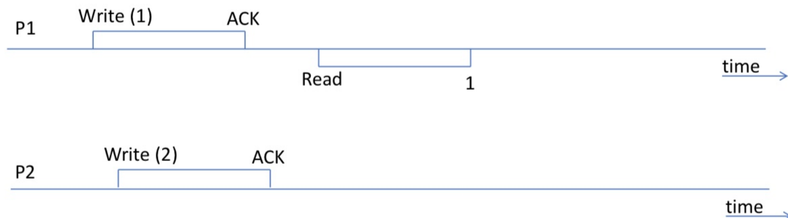
Example execution 4

Is this execution possible for an (N,N) atomic register?



Example execution 4

Is this execution possible for an (N,N) atomic register?



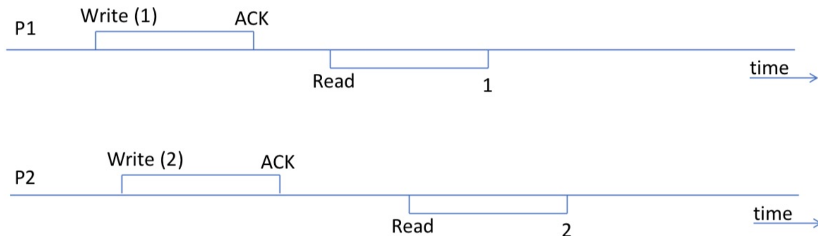
Write operations are concurrent, we have to define linearization points to arbitrate their order.

Example execution 5

Is this execution possible for an (N,N) atomic register?

Example execution 5

Is this execution possible for an (N,N) atomic register?



Not a valid execution, there are no linearization points that explain the return of those two reads.

Your turn!

- We use the replicated regular register to build a replicated key-value store.
- 5 processes replicate one register; at most 2 replicas can fail (i.e. the majority processes will not fail).
- Assumptions: Writers assigns a unique sequence number to each write (i.e. given two written values you can determine the most recent one)

Define an algorithm for reading and writing the register value!

- No update should be lost even if 2 of the 5 replicas fail
- Every read returns the value of one of the potential concurrent writes, or the last preceding write.
- How many acknowledgements from the replicas does a writer need to be sure that the write succeeded despite potential replica fault?
- How many replies does a reader need to obtain the last written value?

Intuition

- We wait for at least 3 processes to reply to the writer; this ensures that our writes will be successful even if 2 replicas fail.
- But when I read, how can I be sure that I am reading the last value?
- If I read from just one replica, I might have missed the last write(s).

- A reader needs to read from at least 3 processes; this ensures that it will read at least from one process that knows the last write.
- If several different values are returned when reading, we just need to figure out which one is the last write ($\Rightarrow$ sequence number!).

Why is this correct?

- *Liveness*: Operations always terminate because you only wait for a number of processes that will never fail (since there are at most 2 failures).
- *Safety*: Any write and read operation will intersect in one correct process. The read will either return the previous or the currently written value in case of concurrency.

This intersection is the basis for quorum-based replication algorithms.

Quorum system

Definition

Given a set of replicas $P = \{p_1, p_2, \ldots, p_N\}$, a **quorum system** $Q = \{q_1, q_2, \ldots, q_M\}$ is a set of subsets of P such that for all $1 \leq i, j \leq M, i \neq j$:

$$q_i \cap q_j \neq \emptyset$$

- Examples: $P = \{p_1, p_2, p_3\}$
  - $Q_1 = \{\{p_1, p_2\}, \{p_2, p_3\}, \{p_3, p_1\}\}$
  - $Q_2 = \{\{p_1\}, \{p_1, p_2, p_3\}, \{p_1, p_3\}\}$
- A quorum system Q is called *minimal* if $\forall q_i, q_j \in Q : q_i \not\subset q_j$

Definition: Read-Write Quorum systems

Definition

Given a set of replicas $P = \{p_1, p_2, \ldots, p_N\}$, a **read-write quorum system** is a pair of sets $R = \{r_1, r_2, \ldots, r_M\}$ and $W = \{w_1, w_2, \ldots, w_K\}$ of subsets of P such that for all corresponding i, j:

$$r_i \cap w_j \neq \emptyset$$

- Choose quorums $w, r \subseteq P$ with $|w| = W$ and $|r| = R$ such that $W + R > N$
- Typically, reads and writes are always sent to all N replicas in parallel and the first responding replicas determine than the quorum for the operation
- Parameters W and R determine how many nodes need to reply before we consider the operation to be successful.

Quorum Types: Read-one/write-all

Replication strategy based on a read-write quorum system

- Read operations can be executed in any (and a single) replica ($R = 1$).
- Write operations must be executed in all replicas ($W = N$).

Properties:

- Very fast read operations
- Heavy write operations
- If a single replica fails, then write operations can no longer be executed successfully.

Quorum Types: Read-all/write-one

Replication strategy based on a read-write quorum system

- Read operations can be executed in all replicas ($R = N$).
- Write operations must be executed in one replica ($W = 1$).

Properties:

- Very fast write operations
- Slow read operation
- If a single replica fails, then read operations can no longer be executed successfully.

Quorum Types: Majority

Replication strategy based on a quorum system

- Every operation (either read or write) must be executed across a majority of replicas (e.g. $\lfloor \frac{N}{2} \rfloor + 1$).

Properties:

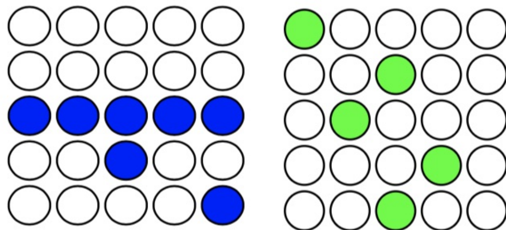
- Best fault tolerance possible from a theoretical point of view
  - Can tolerate f faults with $N = 2f + 1$
- Read and write operations have a similar cost

Quorum Types: Grid

Processes are organized (logically) in a grid to determine the quorums

Example:

- *Write Quorum:* One full line + one element from each of the lines below that one
- *Read Quorum:* One element from each line



Properties:

- Size of quorums grows sub-linearly with the total number of replicas in the system: $O(\sqrt{N})$
  - This means that load on each replica also increases sub-linearly with the total number of operations.
- It allows to balance the dimension of read and write quorums (for instance to deal with different rates of each type of request) by manipulating the size of the grid (i.e, making it a rectangle)
- Complex

How can we compare the different schemes?[8]

Load

The load of a quorum system is the minimal load on the busiest element.

An **access strategy** Z defines the probability $P_Z(q)$ of accessing a quorum $q \in Q$ such that $\sum_{q \in Q} P_Z(q) = 1$.

The **load** of an access strategy Z on a node p is defined by

$$L_Z(p) = \sum_{q \in Q, p \in q} P_Z(q)$$

The load on a quorum system Q induced by an access strategy Z is the maximal load on any node:

$$L_Z(Q) = \max_{p \in P} L_Z(p)$$

The load of a quorum system Q is the minimal load on the busiest element:

$$L(Q) = \min_{Z} L_Z(Q)$$

Resilience and failure probability

If any f nodes from a quorum system Q can fail such that there is still a quorum $q \in Q$ without failed nodes, then Q is f-**resilient**.

The largest such f is the **resilience** $R(Q)$.

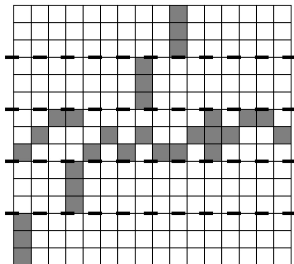
Assume that every node is non-faulty with a fixed probability (here: $p > 1/2$). The **failure probability** $F(Q)$ of a quorum system Q is the probability that at least one node of every quorum fails.

Analysis

- The majority quorum system has the highest resilience ($\lfloor \frac{N-1}{2} \rfloor$); but it has a bad load ($1/2$). Its asymptotic failure probability ($N \rightarrow \infty$) is 0.
- One can show that for *any* quorum system S, the load $L(S) \geq 1/\sqrt{N}$.
- Can we achieve this optimal load while keeping high resilience and asymptoatic failure probability of 0?

Quorum Types: B-Grid[8]

- Consider $N = dhr$ nodes.
- Arrange the nodes in a rectangular grid of width d, and split the grid into h bands of r rows each.
- Each element is represented by a square in the grid.
- To form a quorum take one “mini-column” in every band, and add a representative element from every mini-column of one band $\Rightarrow d + hr - 1$ elements in every quorum.

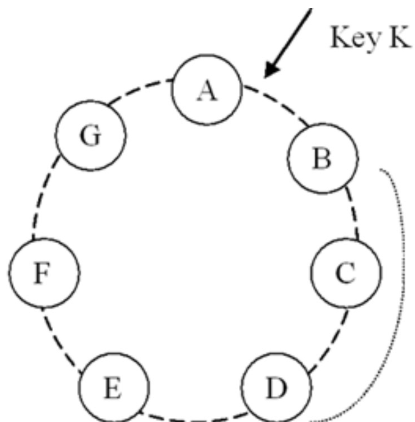


Case study: Dynamo

Amazon Dynamo[3]

- Distributed key-value storage
- Dynamo was one of the first successful non-relational storage systems (a.k.a. NoSQL)
  - Data items accessible via some primary key
  - Interface: `put(key, value)` & `get(key)`
- Used for many Amazon services, e.g. shopping cart, best seller lists, customer preferences, product catalog, etc.
  - Several million checkouts in a single day
  - Hundreds of thousands of concurrent active sessions – Available as service in AWS (DynamoDB)
- Uses quorums to achieve partition- and fault-tolerance

Ring architecture



- Consistent hashing of keys with “virtual nodes” for better load balancing
- Replication strategy:
  - Configurable number of replicas (N)
  - The first replica is stored regularly with consistent hashing
  - The other $N-1$ replicas are stored in the $N-1$ successor nodes (called preference list)
- Typical Dynamo configuration: $N=3, R=2, W=2$
  - But e.g. for high performance reads (e.g., write-once, read-many): $R=1, W=N$

Sloppy quorums

If Dynamo used a traditional quorum approach, it would be unavailable during server failures and network partitions, and would have reduced durability even under the simplest of failure conditions. To remedy this, it does not enforce strict quorum membership and instead it uses a "sloppy quorum"; all read and write operations are performed on the first N healthy nodes from the preference list, which may not always be the first N nodes encountered while walking the consistent hashing ring. [3]

Why are sloppy quorums problematic?

- Assume $N = 3, R = 2, W = 2$ in a cluster of 5 nodes (A, B, C, D, and E)
- Further, let nodes A, B, and C be the top three preferred nodes; i.e. when no error occurs, writes will be made to nodes A, B, and C.
- If B and C were not available for a write, then a system using a sloppy quorum would write to D and E instead.
- In this case, a read immediately following this write could return data from B and C, which would be stale because only A, D, and E would have the latest value.

Dynamos' solution: Hinted handoff

- If the system needs to write to nodes D and E instead of B and C, it informs D that its write was meant for B and informs E that its write was meant for C.
- Nodes D and E keep this information in a temporary store and periodically poll B and C for availability.
- Once B and C become available, D and E send over the writes.

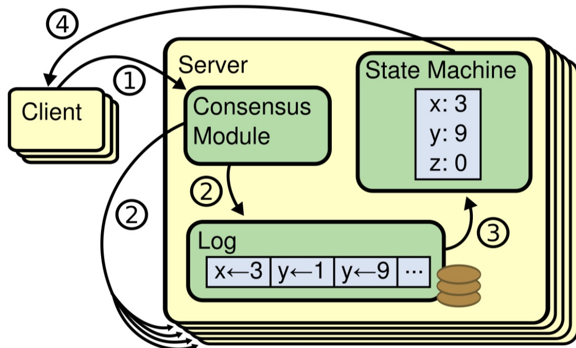
Summary

- Quorums are essential building blocks for many applications in distributed computing (e.g. replicated databases).
- Essential property of quorum systems is the pairwise non-empty intersection of quorums.
- Majority quorums are intuitive and comparatively easy to implement, but far from optimal.
- Small quorums are not necessarily better
  - Compare loads and availability instead of size!

Protocols for Replicated State Machines

Protocols for Replicated State Machines

Motivation: Replicated state-machine via Replicated Log



All figures in these slides are taken from [9].

- Replicated log $\Rightarrow$ State-machine replication
  - Each server stores a log containing a sequence of state-machine commands.
  - All servers execute the same commands in the same order.
  - Once one of the state machines finishes execution, the result is returned to the client.
- Consensus module ensures correct log replication
  - Receives commands from clients and adds them to the log
  - Communicates with consensus modules on other servers such that every log eventually contains same commands in same order
- *Failure model:* Nodes may crash, recover and rejoin, delayed/lost messages

Practical aspects

- **Safety**: Never return in incorrect result despite network delays, partitions, duplication, loss, reordering of messages
- **Availability**: Majority of servers is sufficient
  - Typical setup: 5 servers where 2 servers can fail
- **Performance**: (Minority of) Slow servers should not impact the overall system performance

Approaches to consensus

- *Leader-less (symmetric)*
  - All servers are operating equally
  - Clients can contact any server
- *Leader-based (asymmetric)*
  - One server (called leader) is in charge
  - Other server follow the leader's decisions
  - Clients interact with the leader, i.e. all requests are forwarded to the leader
  - If leader crashes, a new leader needs to be (s)elected
  - Quorum for choosing leader in next epoch (i.e. until the leader is suspected to have crashed)
  - Then, overlapping quorum decides on proposed value $\Rightarrow$ Only accepted if no node has knowledge about higher epoch number

Classic approaches I

- Paxos[6]
  - The original consensus algorithm for reaching agreement on a **single value**
  - Leader-based
  - Two-phase process: Promise and Commit
    - Clients have to wait 2 RTTs
  - Majority agreement: The system works as long as a majority of nodes are up
  - Monotonically increasing version numbers
  - Guarantees safety, but not liveness

Classic approaches II

- Multi-Paxos
  - Extends Paxos for a stream of a agreement problems (i.e. total-order broadcast)
  - The promise (Phase 1) is not specific to the request and can be done before the request arrives and can be reused
  - Client only has to wait 1 RTT
- View-stamped replication (revisited)[7]
  - Variant of SMR + Multi-Paxos
  - Round-robin leader election
  - Dynamic membership

The Problem with Paxos

> *[...] I got tired of everyone saying how difficult it was to understand the Paxos algorithm.[...] The current version is 13 pages long, and contains no formula more complicated than n1 > n2. [5]*

Still significant gaps between the description of the Paxos algorithm and the needs or a real-world system

- Disk failure and corruption
- Limited storage capacity
- Effective handling of read-only requests
- Dynamic membership and reconfiguration

In Search of an Understandable Consensus Algorithm: Raft[9]

- Yet another variant of SMR with Multi-Paxos
- Became very popular because of its understandable description

In essence

- Strong leadership with all other nodes being passive
- Dynamic membership and log compaction

Consensus Algorithms in Real-World Systems

- Paxos made live - or: How Google uses Paxos
  - Chubby: Distributed coordination service built using Multi-Paxos and MSR
- Spanner: Paxos-based replication for hundreds of data centers; uses hardware-assisted clock synchronization for timeouts
- Apache Zookeeper: Distributed coordination service using Paxos
  - Typically used as naming service, configuration management, synchronization, priority queue, etc.
- etcd: Distributed KV store using Raft
  - Used by many companies / products (e.g. Kubernetes, Huawei)
- RethinkDB: JSON Database for realtime apps
  - Storing of cluster metadata such as information about primary

Summary

- Consensus algorithms are an important building block in many applications
- Replicated log via total-order broadcast
- Raft as alternative to classical Paxos
  - Leader election
  - Log consistency
  - Commit

Further reading I

[1] Bernadette Charron-Bost, Fernando Pedone und André Schiper, Hrsg. *Replication: Theory and Practice*. Bd. 5959. Lecture Notes in Computer Science. Springer, 2010. ISBN: 978-3-642-11293-5. DOI: 10.1007/978-3-642-11294-2. URL: https://doi.org/10.1007/978-3-642-11294-2.

[2] George Coulouris u. a. *Distributed Systems: Concepts and Design*. 5th. USA: Addison-Wesley Publishing Company, 2011.

[3] Giuseppe DeCandia u. a. „Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. Stevenson, Washington, USA: ACM, 2007, S. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281. URL: http://doi.acm.org/10.1145/1294261.1294281.

Further reading II

[4] Michael J. Fischer, Nancy A. Lynch und Mike Paterson. „Impossibility of Distributed Consensus with One Faulty Process“. In: *J. ACM* 32.2 (1985), S. 374–382. DOI: 10.1145/3149.214121. URL: http://doi.acm.org/10.1145/3149.214121.

[5] Leslie Lamport. „Paxos Made Simple“. In: *SIGACT News* 32.4 (Dez. 2001), S. 51–58. ISSN: 0163-5700. DOI: 10.1145/568425.568433. URL: http://research.microsoft.com/users/lamport/pubs/paxos-simple.pdf.

[6] Leslie Lamport. „The Part-Time Parliament“. In: *ACM Trans. Comput. Syst.* 16.2 (1998), S. 133–169. DOI: 10.1145/279227.279229. URL: http://doi.acm.org/10.1145/279227.279229.

Further reading III

[7] Barbara Liskov und James Cowling. *Viewstamped Replication Revisited (Technical Report)*. MIT-CSAIL-TR-2012-021. MIT, Juli 2012.

[8] Moni Naor und Avishai Wool. „The Load, Capacity, and Availability of Quorum Systems". In: *SIAM J. Comput.* 27.2 (1998), S. 423–447. DOI: 10.1137/S0097539795281232. URL: https://doi.org/10.1137/S0097539795281232.

[9] Diego Ongaro und John K. Ousterhout. „In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference, USENIX ATC '14, Philadelphia, PA, USA, June 19-20, 2014.* Hrsg. von Garth Gibson und Nickolai Zeldovich. USENIX Association, 2014, S. 305–319. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

Further reading IV

[10] Fred B. Schneider. „Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial“. In: *ACM Comput. Surv.* 22.4 (Dez. 1990), S. 299–319. ISSN: 0360-0300. DOI: 10.1145/98163.98167. URL: https://doi.org/10.1145/98163.98167.