TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Programming Distributed Systems

05 Quorums

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Summer Term 2018

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: 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.

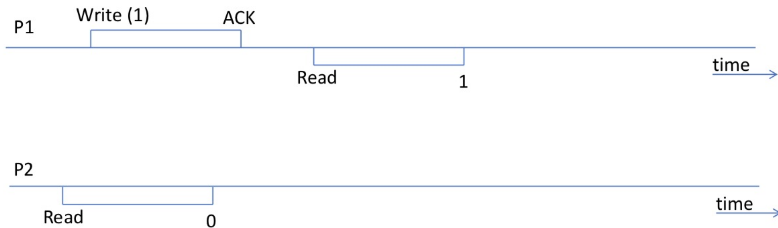
Different types of registers (multiple writers and readers)

(N,N) Atomic register

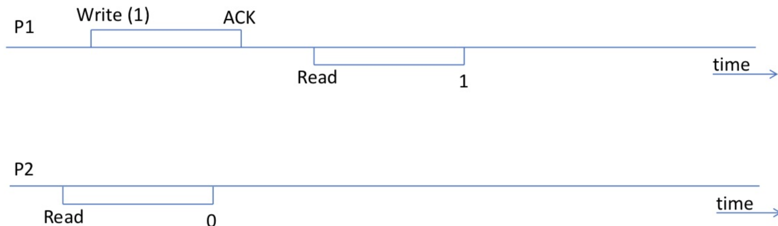
Every read operation returns the value that was written most recently in a hypothetical execution, where every operation appears to have been executed at some instant between its invocation and its completion (linearization point).

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

Example execution 1

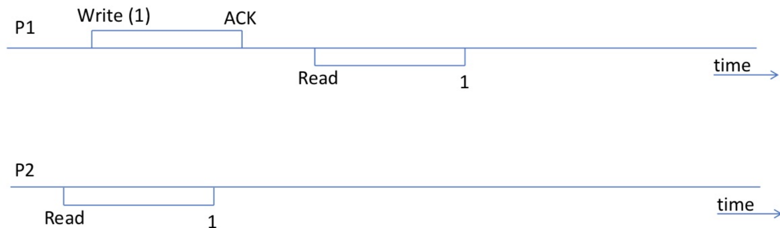


Example execution 1

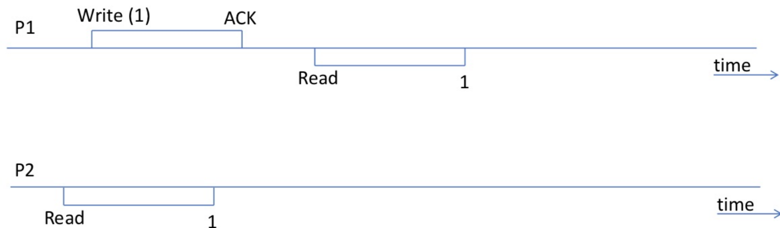


Valid!

Example execution 2

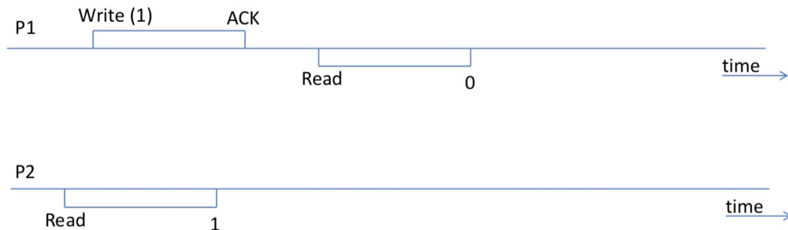


Example execution 2



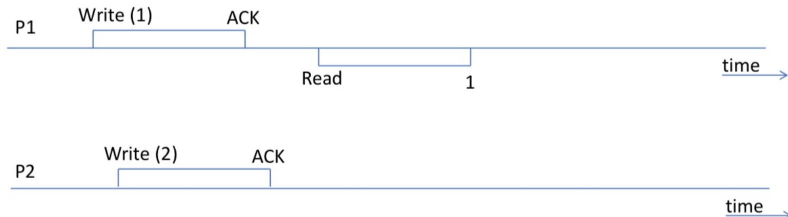
Valid!

Example execution 3



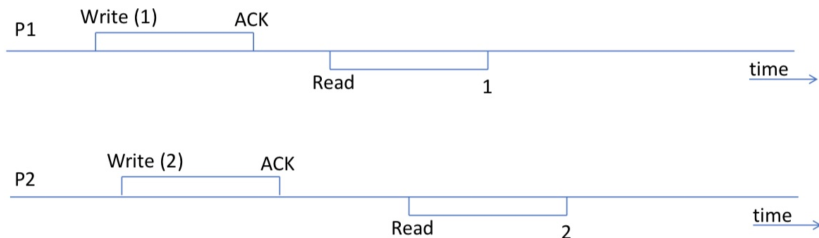
Not valid!

Example execution 4



Write operations are concurrent, we have to define serialization points to arbiter their order.

Example execution 5



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

Your task!

- Assume that one writer and a reader operate on a shared regular register.
- The writer assigns a unique sequence number to each write (i.e. given two written values you can determine the most recent).
- 5 processes replicate this register; at most 2 replicas can fail (i.e. the majority processes will not fail).

Questions

- How many acknowledgements does the writer need to be sure that the write succeeded?
- How many replies does a reader need to obtain the last written value?
- Can you optimize the algorithms for fast reads? And for fast writes?
- How does your scheme work for N replicas, where f replicas may fail and $N \geq 2f + 1$?

Intuition

- We wait for at least $N/2+1$ processes to reply to the writer, that ensures our writes will be successful even if f 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 $N/2+1$ 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?

- Operations always terminate because you only wait for a number of processes that will never fail (since there are at most f failures).
- Any write and read operation (more generally: any pair of operations) will intersect in one correct process.

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

Read repair and anti-antropy

- We need to ensure that eventually all updates are applied at every replica even if nodes are temporarily unavailable (e.g. due to network partitions)
- When a read receives different replies, the reader can forward the newest value to the replicas with stale values (**read repair**).
  - Works well with registers that are frequently read
- A background process can check for differences in the values on each replica and forward missing updates from one replica to another (**anti-antropy**).
  - Needed for registers that are rarely read

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 $1 \leq i, j \leq M, i \neq j$:

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

- 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:

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

- Also called *asymmetric* quorum system
- Typically, reads and writes are always sent to all N replicas in parallel and choose quorums $w, r \subseteq P$ with $|w| = W$ and $|r| = R$ such that $W + R > N$
- W and R determine how many nodes need to reply before we consider the operation to be successful.
  - Why is this a quorum system?

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.
- Write operations must be executed in all replicas.

Properties:

- Very fast read operations
- Heavy write operations
- If a single replica fails, then write 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:

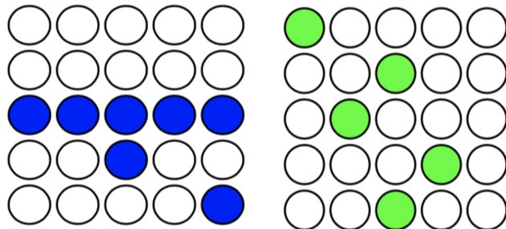
- Best fault tolerance possible from a theoretical point of view
  - Can tolerate f faults with $N = 2f + 1$
- Read and write operations have a similar cost

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?(Naor and Wool 1998)

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** $R(Q)$.

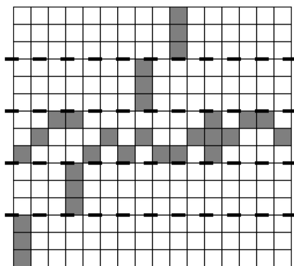
Assume that every node is non-faulty with a fixed probability (here: $p > 1/2$). The **failure probability** $F(Q)$ of a quorum system Q is the probability that at least one node of every quorum fails.

Analysis

- The majority quorum system has the highest resilience ($\lfloor \frac{N-}{2} \rfloor$); but it has a bad load ($1/2$). Its asymptotic failure probability ($N \to \infty$) 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(Naor and Wool 1998)

- 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.

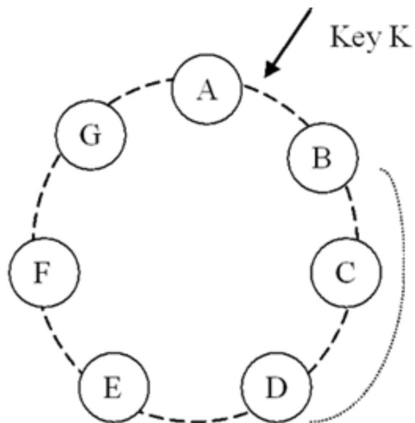


Case study: Dynamo

Amazon Dynamo(DeCandia et al. 2007)

- Distributed key-value storage
- Dynamo marks one of the first non-relational storage systems (a.k.a. NoSQL) – Data items only accessible via some primary key
  - Interface: `put(key, value)` & `get(key)`
- Used for many Amazon services ("applications"), 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 now also as service in AWS as well (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. (DeCandia et al. 2007)

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 inconsistent 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!
- More on quorum theory: (Vukolic 2010)

Next week: Consensus algorithms in Paxos-style

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Further reading

DeCandia, Giuseppe, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. "Dynamo: Amazon's Highly Available Key-Value Store." In *Proceedings of Twenty-First Acm Sigops Symposium on Operating Systems Principles*, 205–20. SOSP '07. New York, NY, USA: ACM. https://doi.org/10.1145/1294261.1294281.

Naor, Moni, and Avishai Wool. 1998. "The Load, Capacity, and Availability of Quorum Systems." *SIAM J. Comput.* 27 (2): 423–47. https://doi.org/10.1137/S0097539795281232.

Vukolic, Marko. 2010. "The Origin of Quorum Systems." *Bulletin of the EATCS* 101: 125–47. http://eatcs.org/beatcs/index.php/beatcs/article/view/183.