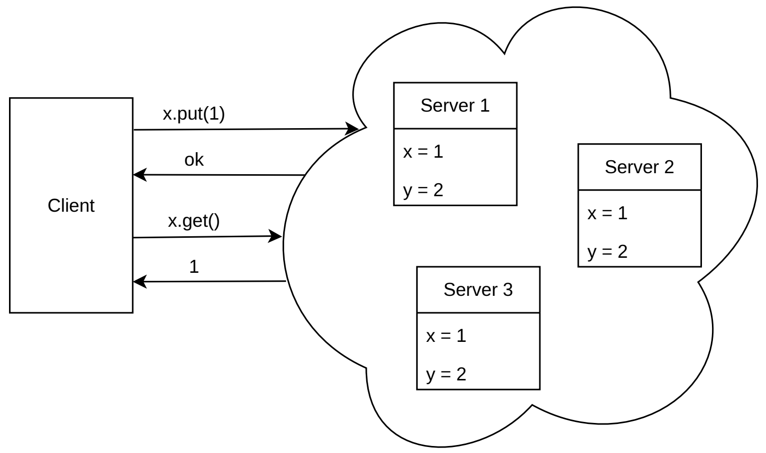
Replicated Data

Distributed Data

> A distributed system is a collection of independent computers that appears to its users as a single coherent system.

Andrew S. Tanenbaum and Maarten van Steen. 2006. Distributed Systems: Principles and Paradigms (2nd Edition).



When to use it

Reason 1: reliability, i.e., fault tolerance.

Reason 2: performance, i.e., very large data.

With retries we execute operations at most once.

Now we need to execute operations exactly once.

Later we will execute operations at least once.

Examples: distributed databases, key-value stores, collaborative editors, blockchains, ...

State Machine Replication

Fred B. Schneider. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial.

Mealy machine: states, operations, outputs, deterministic transition.

Goal: all replicas execute the same operations in the same order.

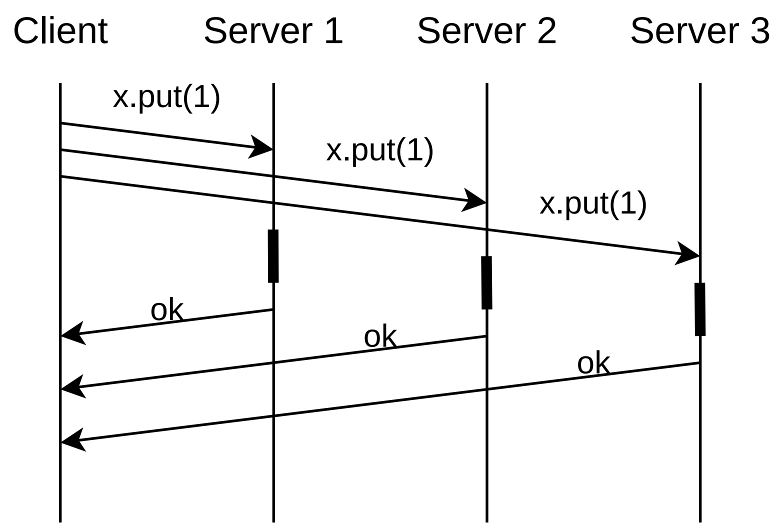
Reliability: every live process reliably receives every operation.

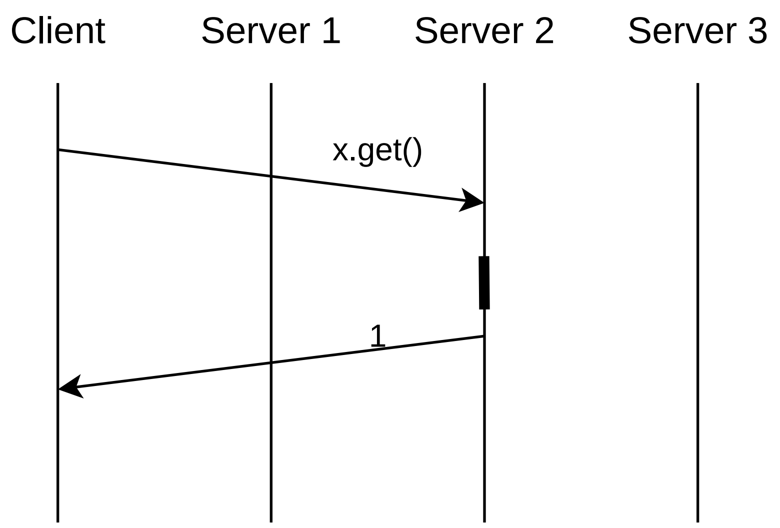
Ordering: every live process applies them in the same order.

Running example: a key-value store with operations `put` and `get`.

Active Replication

Client sends puts to all replicas and waits for acknowledgment and sends gets to any replica.



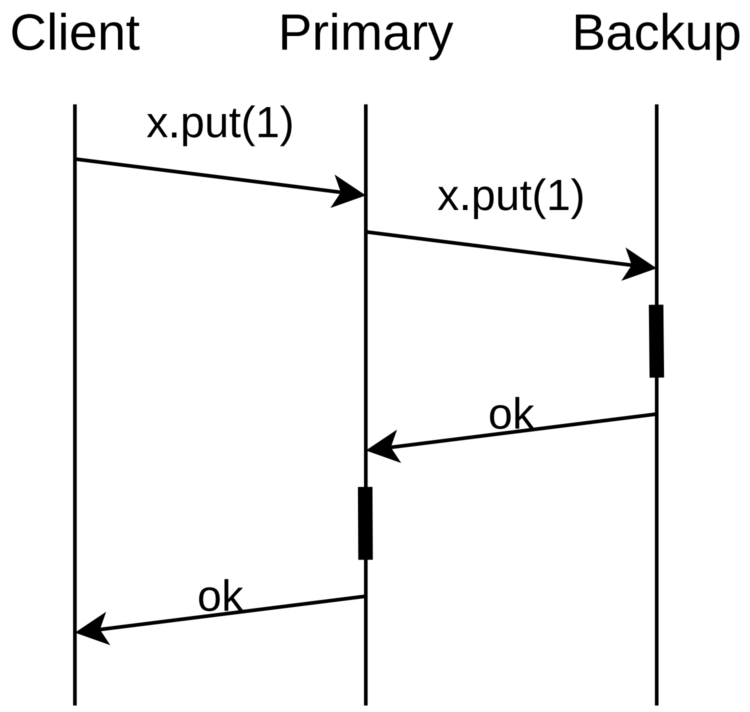


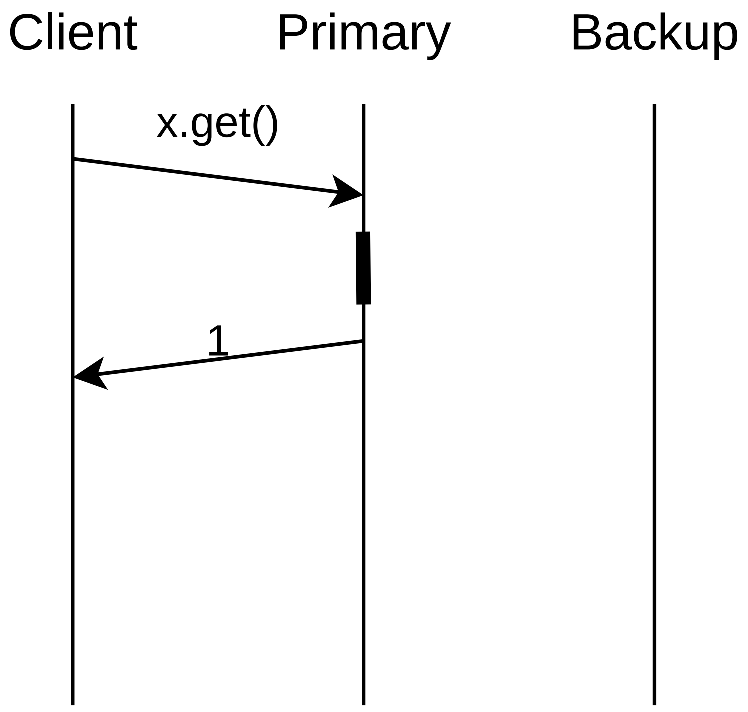
Put: write and acknowledge.

Get: read and respond.

Primary-Backup Replication

Client sends puts and gets to primary.



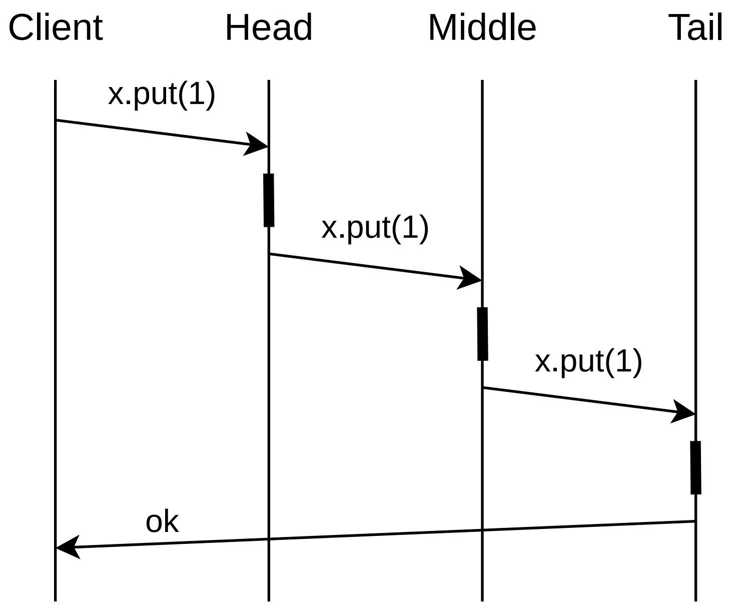


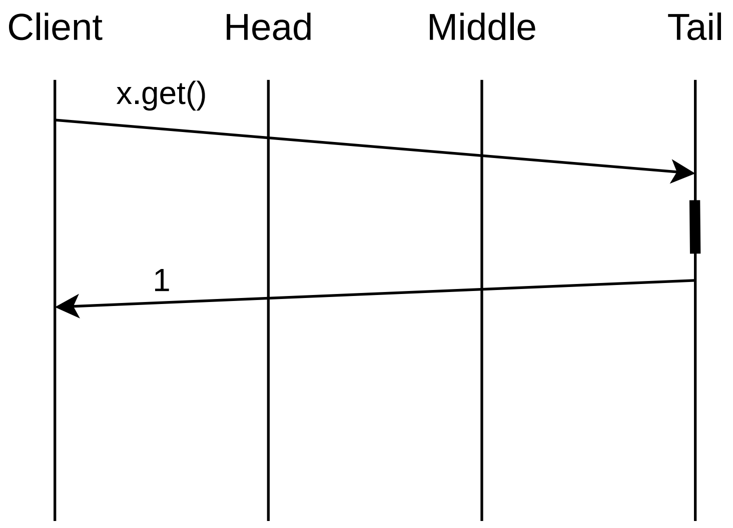
Put: write and wait for acknowledgment from backup.

Get: read and respond.

Chain Replication

Client sends puts to head and gets to tail.





Put: write and send on.

Get: read and respond.

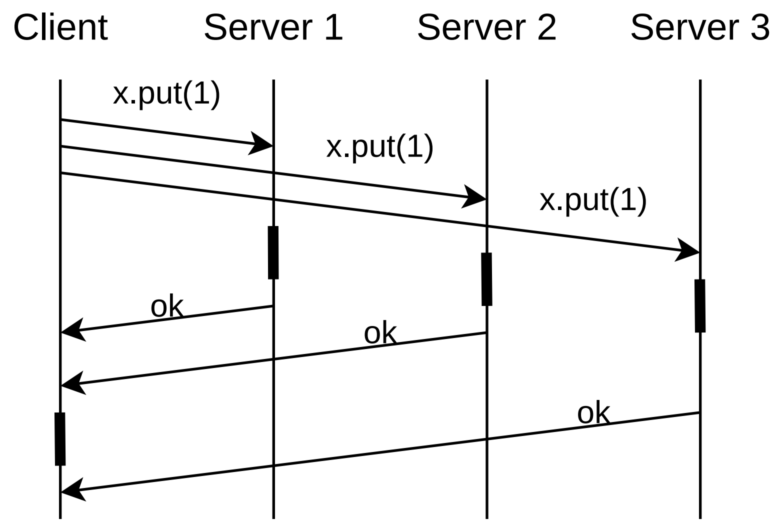
Quorum Protocols

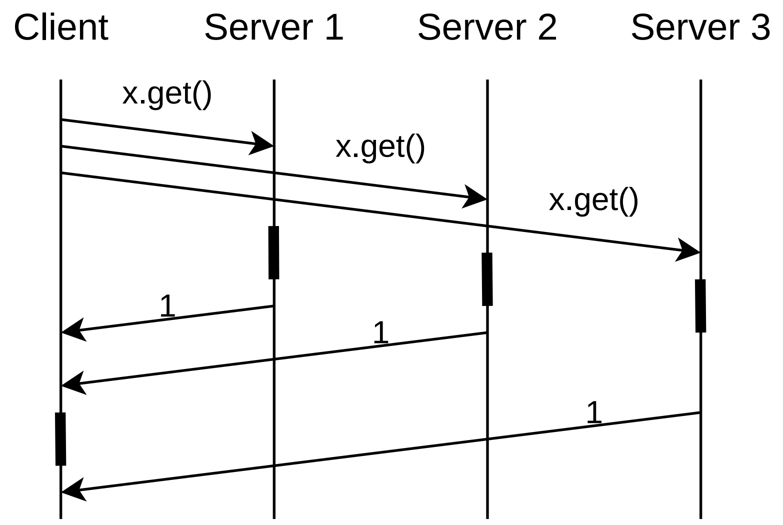
Client waits for enough acknowledgments. Tunable: R + W > N versus not.

N: number of replicas

W: votes required for write

R: votes required for read



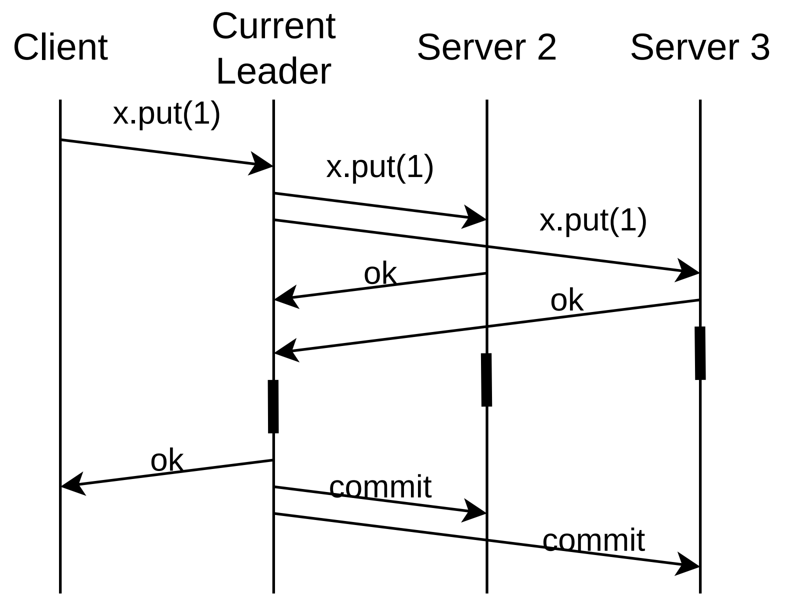


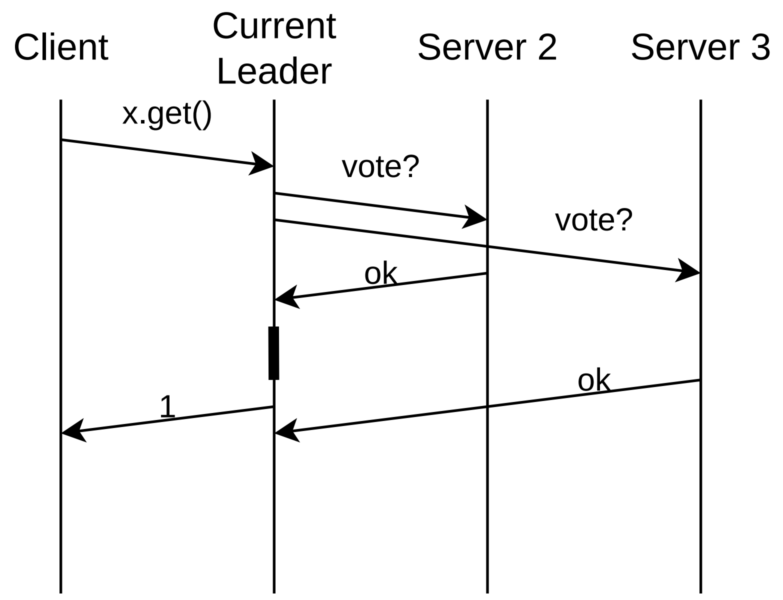
Put: write and acknowledge.

Get: read and respond.

Consensus Protocols

Client sends operation to any server.





Very complicated.

Paxos: Leslie Lamport. 1998. The part-time parliament.

Raft: Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm.

Machine Crashes

Active replication: tolerates 0 crashes for writes (N - 1) crashes for reads.

Primary backup replication: tolerates 1 crash with backup taking over.

Chain replication: tolerates (N - 1) crashes with bypassing in chain.

Quorum replication: tolerates (N - W) crashes for writes and (N - R) crashes for reads.

Consensus protocol: tolerates F failures when N = 2 * F + 1

Concurrent Operations

Only possible with mutliple cients.

Active replication: replicas receive and execute operations in different order.

Primary backup replication: all clients talk to same server which orders operations.

Chain replication: clients talk to the current head which orders operations.

Quorum replication: replicas receive and execute operations in different order.

Consensus protocol: clients talk to any server but current leader decides order.

Ordering Schemes

etcd (CoreOS): consensus-based total order via Raft

Diego Ongaro and John Ousterhout. 2014. In search of an understandable consensus algorithm.

Dynamo (Amazon): version vectors and reconciliation

Giuseppe DeCandia, et al. 2007. Dynamo: amazon's highly available key-value store.

Spanner (Google): real time with global positioning system and atomic clocks

James C. Corbett, et al. 2013. Spanner: Google’s Globally Distributed Database.

Calvin (Yale): sequencer processes

Alexander Thomson, et al. 2012. Calvin: fast distributed transactions for partitioned database systems.

Performance

Read Latency

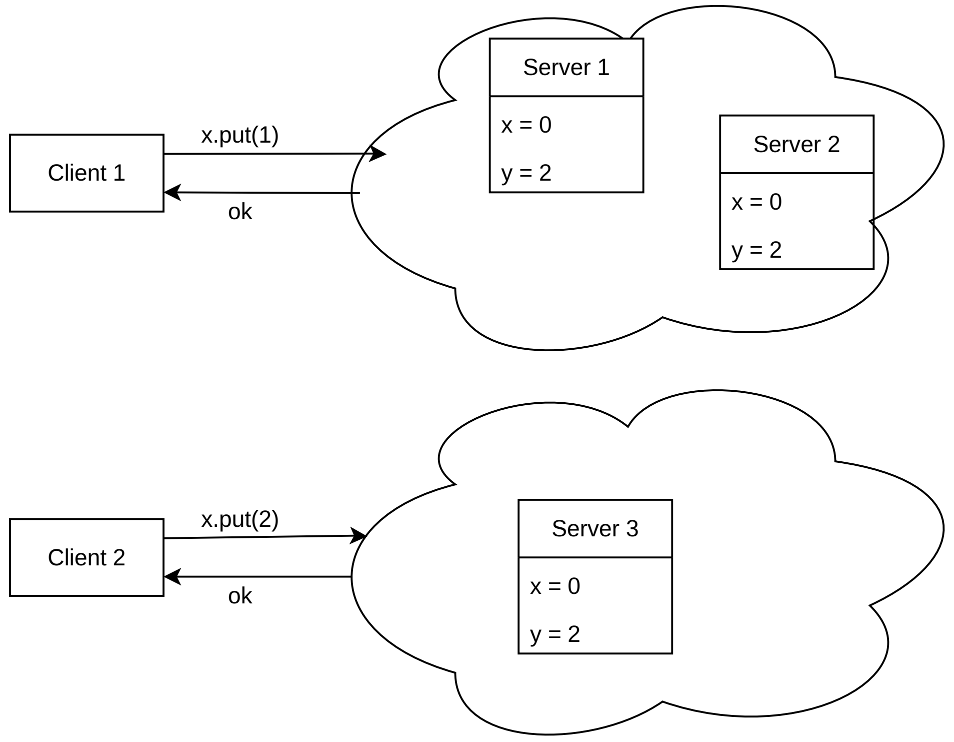
Active replication: low
Primary backup replication: low
Chain replication: low
Quorum replication: tunable
Consensus protocol: medium

Write Latency

Active replication: low
Primary backup replication: medium
Chain replication: high
Quorum replication: tunable
Consensus protocol: high

Network Partitions

The network splits into multiple parts that can not communicate anymore.



CAP theorem: during the partition, choose between consistency and availability.

Consistency Levels

Linearizability: Every operation appears to execute atomically at some point between invocation and response, respecting real-time order.

Sequential Consistency: All operations appear in the same total order to all processes, preserving each process's program order.

Causal Consistency: Operations that are related via happened-before are seen in the same order by all processes, concurrent operations may be seen differently.

Consistency

Active replication: causal consistency by using version vectors.

Primary backup replication: linearizable by single sequencer.

Chain replication: linearizable by single sequencer.

Quorum replication: causal consistency by using version vectors.

Consensus protocol: linearizable by using lots of things.

Temporal Logic

Propositional logic extended with two modalities: □ (always) and ◇ (eventually).

Axioms:

```
□(P → Q) → (□P → □Q)
□P → P
□P → □□P

◇(P → Q) → (◇P → ◇Q)
P → ◇P
◇◇P → ◇P
```

Safety properties: bad things never happen.
Liveness properties: good things eventually happen.

https://lamport.azurewebsites.net/tla/science-book.html

TLA+

https://lamport.azurewebsites.net/tla/tla.html

```
----------- MODULE HourClock ---------------
EXTENDS Naturals

VARIABLES hour

Init == hour = 1

Next == hour' = IF hour = 12 THEN 1 ELSE hour + 1

Spec == Init /\ [][Next]_hour
=============================================
```

Designed by Leslie Lamport, first appeared 1999.

Quint

https://quint-lang.org

```
var balances: str -> int

pure val ADDRESSES = Set("alice", "bob", "charlie")

action withdraw(account, amount) = {
  balances' = balances.setBy(account, curr => curr - amount)
}

val no_negatives = ADDRESSES.forall(addr =>
  balances.get(addr) >= 0
)
```

Designed by the Informal Systems team, first released 2022.

Modern syntax and tooling for TLA+.

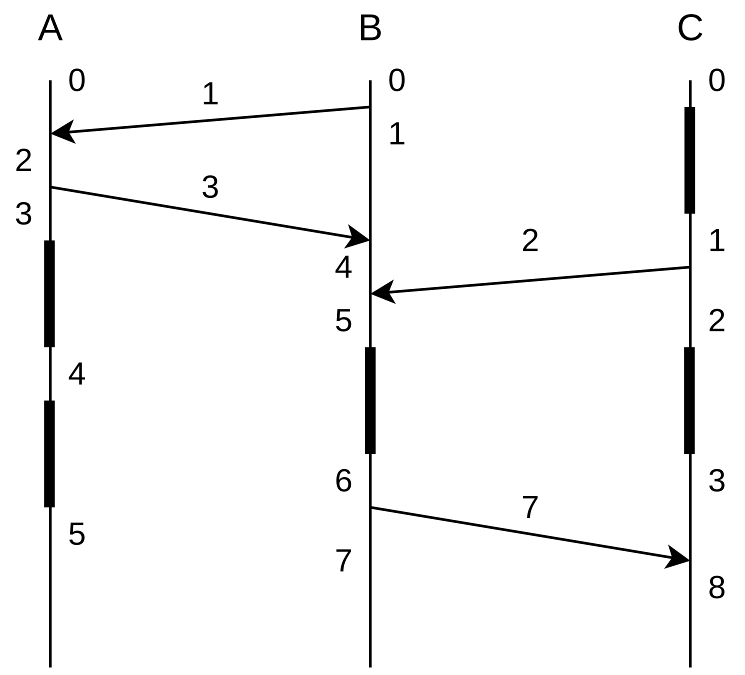
demo/basics.qnt

demo/network.qnt

Logical Time

The happened-before relation is only a partial order.

A logical clock provides a total order that respects this patial order.



Two implementation rules:

> IR1. Each process Pi increments Ci between any two successive events.

> IR2. (a) If event a is the sending of a message m by process Pi, then the message m contains a timestamp Tm = Ci(a).
> (b) Upon receiving a message m, process Pi sets Ci greater than or equal to its present value and greater than Tm.

Use process names to break ties.

Vector Clocks

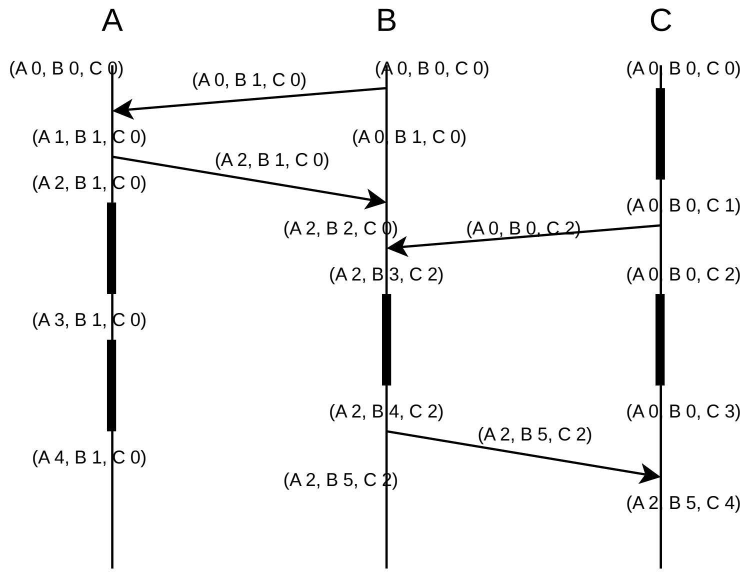
Colin J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial order.

Friedemannn Mattern. 1989. Virtual time and global states of distributed systems.

Vector clocks form a partial order that is precisely happened-before.

Makes it possible to detect concurrent events.

Each process maintains a vector of counters, one entry for each process.



Each process increments its own entry as with Lamport clocks.

Processes attach the vector when sending messages.

Processes update the vector with the pointwise maximum when receiving messages.

demo/causal.qnt

Local-first Data

> However, by centralizing data storage on servers, cloud apps also take away ownership and agency from users. If a service shuts down, the software stops functioning, and data created with that software is lost.

> In this article we propose “local-first software”: a set of principles for software that enables both collaboration *and* ownership for users. Local-first ideals include the ability to work offline and collaborate across multiple devices, while also improving the security, privacy, long-term preservation, and user control of data.

Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. Local-first software: you own your data, in spite of the cloud.

https://www.inkandswitch.com/essay/local-first/

Coordination-free Computation

> Coordination protocols enable autonomous, loosely coupled machines to jointly decide how to control basic behaviors, including the order of access to shared memory. These protocols are among the most clever and widely cited ideas in distributed computing.

> The issue is not that coordination is tricky to implement, though that is true. The main problem is that coordination can dramatically slow down computation or stop it altogether.

Joseph Hellerstein and Peter Alvaro. 2020. Keeping CALM: when distributed consistency is easy.

https://cacm.acm.org/research/keeping-calm/

Semilattices

Associativity: `(a + b) + c = a + (b + c)` (for composition).

Commutativity: `a + b = b + a` (for permutation).

Idempotence: `a + a = a` (for duplication).

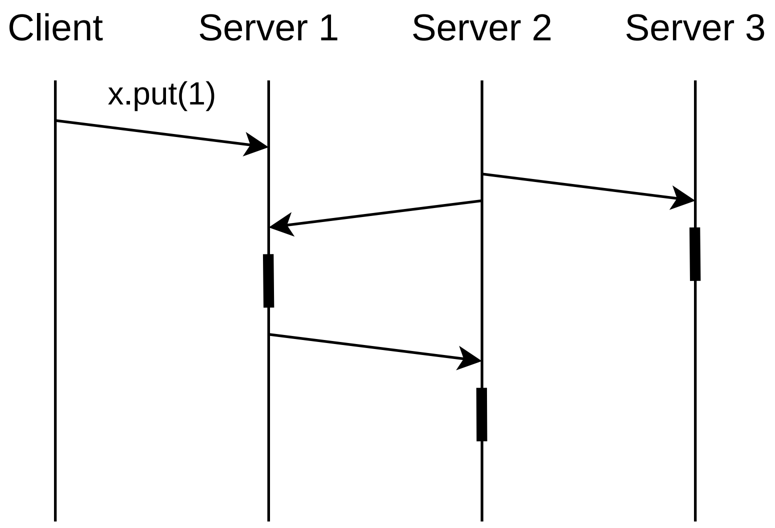
These properties induce an order: `a ≤ b` is defined as `a + b = b`.

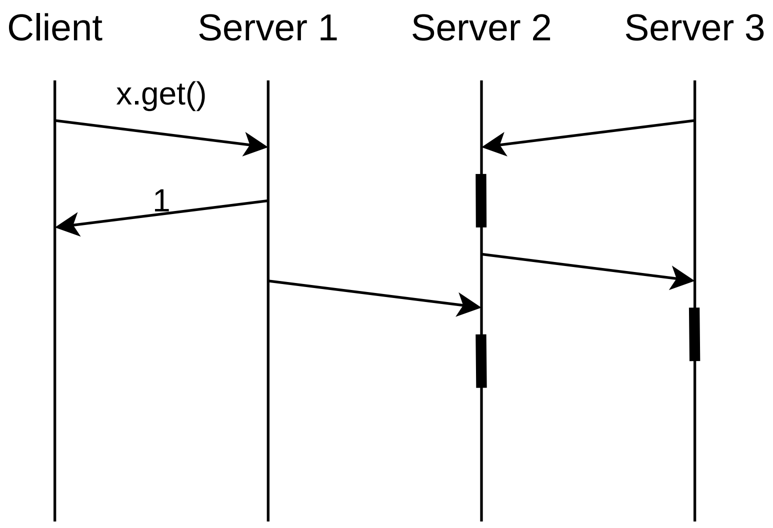
Inflation: `a ≤ f(a)`(for operations).

Monotonicity: when `a ≤ b` then `f(a) ≤ f(b)` (not needed).

Gossip Protocol

Client sends operation to any server.





Servers exchange and merge their states independent of clients.

Conflict-free Replicated Datatypes (CRDTs)

Paulo Sérgio Almeida. 2024. Approaches to Conflict-free Replicated Data Types.

```
type S

merge : (S × S) -> S

update : U -> (S -> S)

view : S -> V
```

Where `merge` forms a semilattice and `update u` is inflationary.

Properties of Gossip Protocol with CRDT

Network Partitions: no problem.
Machine Crashes: no problem.

Read Latency: low
Write Latency: low

Eventual Consistency: If no new updates are made, all reads eventually return the same value.

When to use it

Mobile applications

Collaborative editing

Edge computing

Geo-distributed databases

Examples:

- Automerge: https://automerge.org/
- Yjs: https://yjs.dev/

demo/counter.qnt

Grow-only Counter

A counter with an operation `increment`.

```
type S = Vector[Nat]

merge s1 s2 = zipWith(max, s1, s2)

increment s = s[i] += 1

view s = sum(s)
```

Where `i` is index of executing replica.

Positive-Negative Counter

A counter with operations `increment` and `decrement`.

```
type S = (Vector[Nat], Vector[Nat])

merge (p1, n1) (p2, n2) = (zipWith(max, p1, p2), zipWith(max, n1, n2))

increment (p, n) = (p[i] += 1, n)
decrement (p, n) = (p, n[i] += 1)

view (p, n) = sum(p) - sum(n)
```

Where `i` is index of executing replica.

Observed-Reset Counter

A counter with operations `increment` and `reset`.

A replica only resets those increments it has observed.

```
type S = (Vector[Nat], Vector[Nat])

merge (p1, n1) (p2, n2) = (zipWith(p1, p2, max), zipWith(n1, n2, max))

increment (p, n) = (p[i] += 1, n)
reset (p, n) = (p, zipWith(max, p, n))

view (p, n) = sum(p) - sum(n)
```

Where `i` is index of executing replica.

Grow-only Set

A set with an operation `add`.

```
type S[A] = Set[A]

merge s1 s2 = union(s1, s2)

add s x = insert(s, x)

view s = s
```

Elements, once inserted, can never be removed.

Two-Phase Set

A set with operations `add` and `remove`.

```
type S[A] = (Set[A], Set[A])

merge (a1, r1) (a2, r2) = (union(a1, a2), union(r1, r2))

add (a, r) x = (insert(a, x), r)

remove (a, r) x = (a, insert(r, x))

view (a, r) = difference(a, r)
```

Elements, once inserted and removed, can never be inserted again.

We keep a set of tombstones.

Multi-value Register

A register with an operation `put`.

Reading from the register might result in multiple values.

```
type Dot = (ID, Nat)
type S[T] = (Set[(T, Dot, Set[Dot])], Set[Dot])

merge (w1, o1) (w2, o2) = (union(w1, w2), union(o1, o2))

put (w, o) v = (insert(w, (v, (i, n), o)), insert(o, (i, n)))
  where `i` is index of executing replica
  where `n` is the next fresh version at this replica

view (w, o) = filter overwritten writes from w
```

Observed writes are overwritten, concurrent writes are kept.

Text

> The key idea of most text CRDTs is to represent the text as a sequence of characters, and to give each character a unique identifier.

> To insert a character into a document, we generate an insert operation of the following form:
>
> ```
> { action: "insert", opId: "2@alice", afterId: "1@alice", character: "x" }
> ```
>
> This operation inserts the character “x” after the existing character whose ID is 1@alice.

> To delete a character from a document, we generate a remove operation of the following form:
>
> ```
> { action: "remove", opId: "5@alice", removedId: "2@alice" }
> ```
>
> This operation removes the existing character whose ID is 2@alice (i.e. the “x” we inserted above).

Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. 2022. Peritext: A CRDT for Collaborative Rich Text Editing.

Summary and Outlook

Replicating data and keeping it consistent is hard.

Next week: Collective Computation.