Distributed systems
Lecture 14: Elections, distributed transactions, and replication

Michaelmas 2019
Dr Martin Kleppmann
(With thanks to Dr Robert N. M. Watson and Dr Steven Hand)
Last time

• **Vector clocks** and **consistent global state**
  • Saw how we can build **ordered multicast**
    – Messages between processes in a group
    – Need to distinguish **receipt** and **delivery**
    – Several ordering options: **FIFO**, **causal** or **total**

• Considered **distributed mutual exclusion**:  
  – Want to limit one process to a **critical section** at a time
  – Central server OK; but bottleneck & **Single Point of Failure (SPoF)**
  – Token passing OK: but traffic, repair, token loss
  – Totally-Ordered Multicast: OK, but high number of messages and problems with failures
Leader election

• Many schemes are built on the notion of having a well-defined "leader" (master, coordinator)
  – Examples seen so far include the Berkeley time synchronization protocol, and the central lock server

• An election algorithm is a dynamic scheme to choose a unique process to play a certain role
  – Assume process \( P_i \) contains state variable \( \text{elected}_i \)
  – When \( P_i \) first joins the group, \( \text{elected}_i = \text{UNDEFINED} \)

• By the end of the election, for every \( P_i \)
  – \( \text{elected}_i = P_x \) where \( P_x \) is the winner of the election, or
  – \( \text{elected}_i = \text{UNDEFINED}, \) or
  – \( P_i \) has crashed or otherwise left the system

• i.e. everyone agrees who the leader is
Difficulties with leader election

• “Processes agree who the leader is.”
• However: notion of “agreement” is hazy
  – When did the processes agree?
  – When did processes find out about leader change?
  – Is the purported leader still the leader, or has it been demoted since we last checked?
• Better to be cautious
  – In general: any action taken by a leader must be accompanied by a check that the process still is, in fact, the leader.
  – Prevent action from taking effect if leadership has changed
Ring-based election

- Leader crashes: some process notices, starts election
  - Find node with highest ID who will be new leader
  - Puts its ID into a message, and sends to its successor
  - On receipt, a process acks to sender (not shown), and then appends its ID and forwards the election message
  - Finished when a process receives message containing its ID
- Depends on accurately detecting crashes — realistic?
The Bully Algorithm

• Algorithm proceeds by attempting to elect the process still alive with the highest ID
  – Assume that we know the IDs of all processes
  – Assumes we can reliably detect failures by timeouts

• If process $P_i$ sees current leader has crashed, sends election message to all processes with higher IDs, and starts a timer
  – Concurrent initiation by multiple processes is fine
  – On receiving an election message reply OK to sender, start their own election (if not in progress)
  – If a process hears nothing back before timeout, it declares itself the winner, and multicasts result

• A recovering dead process (or new process joining) starts an election – the [new] highest ID will be elected
Problems with elections

- Algorithms rely on timeouts to reliably detect failure
- However, networks also fail: a network partition
  - Some processes can speak to others, but not all
- Can lead to split-brain syndrome:
  - Each partition independently elects a leader → too many bosses!
- To fix, need some secondary (& tertiary?) communication scheme
  - e.g. secondary network, shared disk, serial cables, ...
- This is important because we want to implement distributed algorithms dependent on the invariant that the leader is unique
Aside on consensus

• Elections are a specific example of a more general problem: **consensus**
  – Given a set of $N$ processes in a distributed system, how can we get them all to agree on something?

• Classical treatment has every process $P_i$ propose something (a value $V_i$)
  – Want to arrive at some deterministic function of $V_i$’s (e.g. ‘majority’ or ‘maximum’ will work for election)

• A correct solution to consensus must satisfy:
  – **Agreement**: all nodes arrive at the same answer
  – **Validity**: answer is one that was proposed by someone
  – **Termination**: all nodes eventually decide

• e.g. Paxos + variants, Raft, Viewstamped Replication, ...
“Consensus is impossible”

• Famous result due to Fischer, Lynch & Patterson (1985)
  – Focuses on an **asynchronous network** (unbounded delays) with at least one process failure
  – Shows that it is possible to get an infinite sequence of states, and hence **never terminate**
  – Given the Internet is an asynchronous network, then this seems to have major consequences!

• Not really:
  – Result actually says we can’t **always guarantee** consensus, **not** that we can **never achieve** consensus
  – And in practice, we can use tricks to mask failures (such as reboot, or replication), and to ignore asynchrony
  – Have seen solutions already, and will see more later
Transaction processing systems

• Earlier looked at transactions:
  – ACID properties
  – Support for composite operations (i.e. a collection of reads and updates to a set of objects)
• A transaction is atomic (“all-or-nothing”)
  – If it commits, all operations are applied
  – If it aborts, it’s as if nothing ever happened
• A committed transaction moves system from one consistent state to another
• Transaction processing systems also provide:
  – isolation (between concurrent transactions)
  – durability (committed transactions survive a crash)
• Q: Can we bring the {scalability, fault tolerance, ...} benefits of distributed systems to transaction processing?
Distributed transactions

• Scheme described earlier was client/server:
  – E.g., a program (client) accessing a database (server)
• However **distributed transactions** are those which span multiple transaction processing servers
• e.g. exactly-once message processing
  – Processing a message has side-effects: updating data in a database
  – Want changes in database to take effect **iff** message is marked as processed
  – Atomically commit side-effects (in database) and message-delivery status (in message broker)
  – If either fails, transaction is aborted in both systems, and message processing can be safely retried
A model of distributed transactions

- Multiple servers \((S_1, S_2, S_3, \ldots)\), each holding some objects which can be **read** and **written** within client transactions
- Multiple concurrent clients \((C_1, C_2, \ldots)\) who perform transactions that interact with one or more servers
  - E.g. **T1** reads \(x, z\) from \(S_1\), writes \(a\) on \(S_2\), reads+writes \(j\) on \(S_3\)
  - E.g. **T2** reads \(i, j\) from \(S_3\), then writes \(z\) on \(S_1\)
- A successful **commit** implies agreement at all servers
Implementing distributed transactions

• Can build on top of solution for single server:
  – e.g. use locking or shadowing to provide isolation
  – e.g. use write-ahead log for durability

• Need to coordinate to either commit or abort
  – Assume clients create unique transaction ID: \( T_{\text{XID}} \)
  – Uses \( T_{\text{XID}} \) in every read or write request to a server \( S_i \)
  – First time \( S_i \) sees a given \( T_{\text{XID}} \), it starts a tentative transaction associated with that transaction ID
  – When client wants to commit, must perform atomic commit of all tentative transactions across all servers
Atomic commit protocols

• A naïve solution would have client simply invoke `commit(TxID)` on each server in turn
  – Will work only if no concurrent conflicting clients, every server commits (or aborts), and no server crashes

• To handle **concurrent clients**, introduce a **coordinator**:
  – A designated machine (can be one of the servers)
  – Clients ask coordinator to commit on their behalf... and hence coordinator can **serialize** concurrent commits

• To handle **inconsistency/crashes**, the coordinator:
  – Asks all involved servers if they *could* commit `TxID`
  – Servers $S_i$ reply with a vote $V_i = \{ \text{COMMIT}, \text{ABORT} \}$
  – If all $V_i = \text{COMMIT}$, coordinator multicasts `doCommit(TxID)`
  – Otherwise, coordinator multicasts `doAbort(TxID)`
Two-phase commit (2PC)

- This scheme is called **two-phase commit (2PC)**:
  - First phase is **voting**: collect votes from all parties
  - Second phase is **completion**: either abort or commit
- Doesn’t require ordered multicast, but needs reliability
  - If server fails to respond by timeout, implicit vote to **abort**
- Once all ACKs received, inform client of commit success
2PC: additional details

- Client (or any server) can abort during execution: simply multicasts `doAbort(TxID)` to all servers
  - E.g., if client transaction explicitly aborts or server fails
- If a server votes **NO**, can abort at once locally
- If a server votes **YES**, it **must** be able to commit if subsequently asked by coordinator:
  - Before voting to commit, server will **prepare** by writing entries into log and flushing to disk
  - Records all requests from/responses to coordinator
  - Hence even if crashes **after** voting to commit, will be able to recover on reboot
2PC: coordinator crashes

• Coordinator must also **persistently log** events:
  – Including initial message from client, requesting votes, receiving replies, and final decision made
  – Lets it reply if (restarted) client or server asks for outcome
  – Also lets coordinator recover from reboot, e.g. re-send any vote requests without responses, or reply to client

• One additional problem occurs if coordinator crashes **after phase 1, but before initiating phase 2**:
  – Servers will be uncertain of outcome...
  – If voted to commit, will have to continue to hold locks, etc

• Can implement fault-tolerant distributed coordinator using consensus algorithm (e.g. Paxos)
Replication

Many distributed systems involve replication – Multiple copies of some object stored at different servers – Multiple servers capable of providing some operation(s)

Three key advantages:

- **Load-Balancing**: if have many replicas, then can spread out work from clients between them
- **Lower Latency**: if replicate an object/server close to a client, will get better performance
- **Fault-Tolerance**: can tolerate the failure of some replicas and still provide service

Examples include DNS, web & file caching (& content-distribution networks), replicated databases, ...
Replication in a single system

• A good single-system example is **RAID**:
  – RAID = Redundant Array of Inexpensive Disks
  – Disks are cheap, so use several instead of just one
  – If replicate data across disks, can tolerate disk crash
  – If don’t replicate data, appearance of a single larger disk

• A variety of different configurations (levels)
  – RAID 0: **stripe** data across disks, i.e. block 0 to disk 0, block 1 to disk 1, block 2 to disk 0, and so on
  – RAID 1: **mirror** (replicate) data across disks, i.e. block 0 written on disk 0 and disk 1
  – RAID 5: **parity** – write block 0 to disk 0, block 1 to disk 1, and (block 0 XOR block 1) to disk 2

• Improved performance as can access disks in parallel
• With RAID 1, 5 also get fault-tolerance
• NB: More disks **increase risk of single-disk failure** while **reducing probability of fatal multi-disk failure**
Distributed data replication

• Have some number of servers ($S_1$, $S_2$, $S_3$, …)
  – Each holds a copy of all objects
• Each client $C_i$ can access any replica (any $S_i$)
  – E.g. clients can choose closest, or least loaded
• If objects are read-only, then trivial:
  – Start with one primary server $P$ having all data
  – If client asks $S_i$ for an object, $S_i$ returns a copy
  – ($S_i$ fetches a copy from $P$ if it doesn’t already have a fresh one)
• Can easily extend to allow updates by $P$
  – When updating object $O$, send invalidate($O$) to all $S_i$
• In essence, this is how web caching / CDNs work today
• But what if clients can perform updates?
Replication and consistency

• More challenging if clients can perform updates
• For example, imagine $x$ has value 3 (in all replicas)
  – $C_1$ requests $\text{write}(x, 5)$ from $S_4$
  – $C_2$ requests $\text{read}(x)$ from $S_3$ (after $C_1$'s request has completed)
  – What should occur?

• With strong consistency/linearizability, the system behaves as if there was no replication:
  – That is, $C_2$ should read the value 5
  – Requires coordination between all servers

• With weak consistency, $C_2$ may get 3 or 5 (or ...?)
  – Harder to reason about, but better performance
  – Recall close-to-open consistency in NFS
Replication for fault tolerance

- Replication for **services**, not just data objects
- Easiest is for a **stateless service**:
  - Simply duplicate functionality over $k$ machines
  - Clients use any (e.g. closest), fail over to another
- Very few totally stateless services
  - But e.g. many web apps have per-session soft state
  - State generated per-client, lost when client leaves
- For example: multi-tier web farms (Facebook, ...):

![Diagram of a multi-tier web farm](image)
Passive replication

- Stateful services can use **primary/backup**:
  - Backup server takes over in case of failure

- Based on **persistent logs, system checkpoints**:
  - Periodically (or continuously) checkpoint primary
  - If detect failure, start backup from checkpoint

- A few variants trade-off fail-over time:
  - **Cold-standby**: backup server must start service (software), load checkpoint & parse logs
  - **Warm-standby**: backup server has software running in anticipation, must load primary state
  - **Hot-standby**: backup server mirrors primary work, but output is discarded; on failure, enable output
Active replication

- **Alternative**: each of the replicas independently executes each operation

- **Use** **total order multicast**:
  - Client (or frontend server) sends each request to all replicas by total order multicast
  - All replicas receive operations in the same order, apply them in the same order, then respond

- **This is known as** **state machine replication**:
  - Replicas must act **deterministically** based on input
  - Same input + same processing = same state
  - Beware of sources of nondeterminism: random numbers, current time, result order...
  - Any errors/transaction aborts must also be made deterministic. What if a replica crashes?
Summary + next time

• Leader elections + distributed consensus
• Distributed transactions + atomic commit protocols
• Replication + consistency

• (More) replication and consistency
  – Strong consistency
  – Quorum-based systems
  – Weaker consistency
• Consistency, availability and partitions
• Further replication models
• Amazon/Google case studies