Distributed systems
Lecture 6: Elections, distributed transactions, and replication

Dr Robert N. M. Watson

Last time

• 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 CS 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 $P_i$ contains state variable $\text{elected}_i$
  – when a process 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
• Common theme: live node with the highest ID wins

Ring-based election

• System has coordinator who crashes
• Some process notices, and starts an 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
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 election initiation by multiple processes is fine
  - Processes receiving an election message reply OK to sender, and start an election of their own (if not already in progress)
  - If a process hears nothing back before timeout, it declares itself the winner, and multicasts result
- A dead process that recovers (or new process that joins) also starts an election: can ensure highest ID always elected

Problems with elections

- Algorithms rely on timeouts to reliably detect failure
- However it is possible for networks to fail: a network partition
  - Some processes can speak to others, but not all
- Can lead to split-brain syndrome:
  - Every 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

“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

- Last term 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 last term 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. booking a complex trip from London to Vail, CO
  - Could fly LHR -> LAX -> EGE + hire a car...
  - … or fly LHR -> ORD -> DEN + take a public bus
- Want a complete trip (i.e. atomicity)
  - Not get stuck in an airport with no onward transport!
- Must coordinate actions across multiple parties
A model of distributed transactions

- Multiple servers ($S_1, S_2, S_3, ...$), each holding some objects which can be read and written within client transactions
- Multiple concurrent clients ($C_1, C_2, ...$) who perform transactions that interact with one or more servers
  - e.g. $T_1$ reads $x, z$ from $S_1$, writes $a$ on $S_2$, and reads & writes $j$ on $S_3$
  - e.g. $T_2$ 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: $TxID$
  - Uses $TxID$ in every read or write request to a server $S_i$
  - First time $S_i$ sees a given $TxID$, 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 \texttt{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 \texttt{TxID}
  – Servers \( S_i \) reply with a vote \( V_i = \{ \text{COMMIT, ABORT} \} \)
  – If all \( V_i = \text{COMMIT} \), coordinator multicasts \texttt{doCommit(TxID)}
  – Otherwise, coordinator multicasts \texttt{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, treat as a vote to abort

• Once all Acks received, inform client of successful commit
2PC: additional details

- Client (or any server) can abort during execution: simply multicasts `doAbort(TxID)` to all servers
  - E.g., if client transaction throws exception or server fails
- If a server votes **NO**, can immediately abort 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
  - Also 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
- Other schemes (3PC, Paxos, ...) can deal with this
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
• Get improved performance since can access disks in parallel
• With RAID 1, 5 also get fault-tolerance
Distributed data replication

- Have some number of servers \((S_1, S_2, S_3, \ldots)\)
  - 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
  - \((S_i, P)\) 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

- Gets more challenging if clients can perform updates
- For example, imagine \(x\) has value 3 (in all replicas)
  - \(C_1\) requests write\((x, 5)\) from \(S_4\)
  - \(C_2\) requests read\((x)\) from \(S_3\)
  - What should occur?
- With **strong consistency**, the distributed system behaves as if there is no replication present:
  - i.e. in above, \(C_2\) should get the value 5
  - Requires coordination between all servers
- With **weak consistency**, \(C_2\) may get 3 or 5 (or …?)
  - Less satisfactory, but much easier to implement
Replication for fault tolerance

- Replication for services, not just data objects
- Easiest is for a **stateless services**:  
  - 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 multi-tier web farm](image)

**Passive replication**

- A solution for stateful services is **primary/backup**:  
  - Backup server takes over in case of failure
- Based around **persistent logs** and **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 – just needs to load primary state  
  - Hot-standby: backup server mirrors primary work, but output is discarded; on failure, enable output
Active replication

• **Alternative**: have $k$ replicas running at all times

• Front-end server acts as an *ordering node*:
  – Receives requests from client and forwards them to all replicas using **totally ordered multicast**
  – Replicas each perform operation and respond to front-end
  – Front-end gathers responses, and replies to client

• Typically require replicas to be “state machines”:
  – i.e. act deterministically based on input
  – Idea is that all replicas operate ‘in lock step’

• **Active replication** is expensive (in terms of resources)...
  – ... and not really worth it in the common case.
  – However valuable if consider *Byzantine failures*

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
• Start of Google case studies