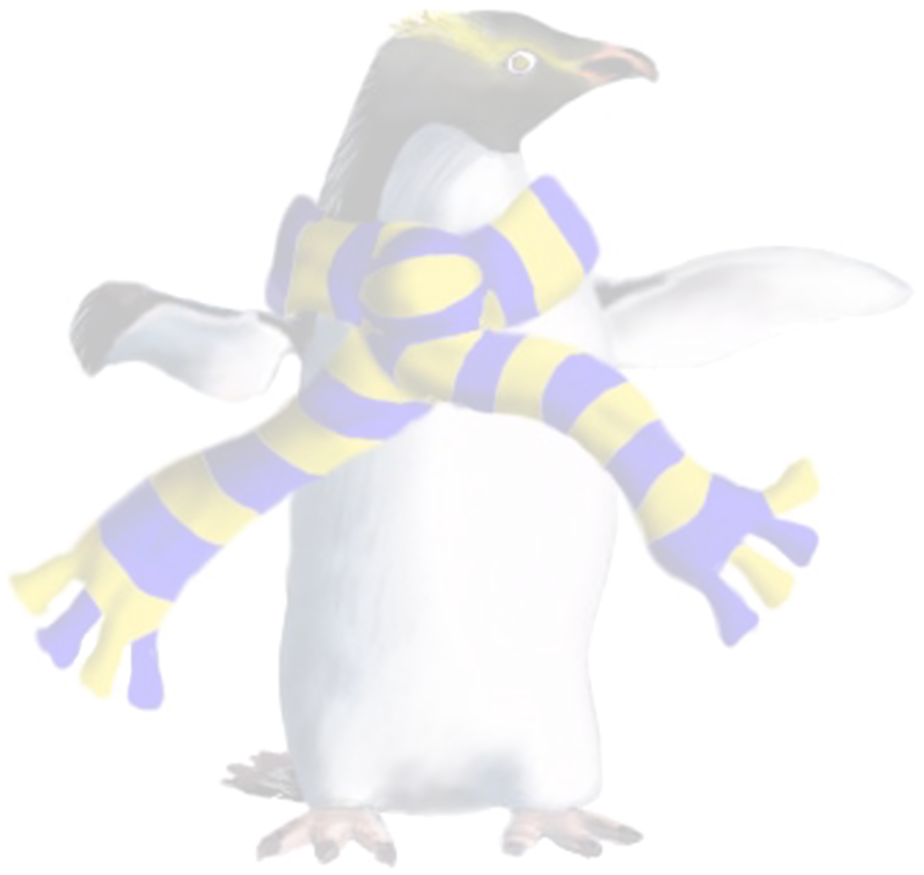

Distributed consensus and integrity checking

COSC349—Cloud Computing Architecture

David Eyers

Learning objectives

- Explain that scale-out design must **avoid contention**
  - Application workflow must be analysed to identify contention
- Describe how **atomic broadcast effects coordination**
  - Must allow coordination to run on multiple servers that can fail
  - Tools like Apache ZooKeeper provide app. coordination needs
  - … but this work is often done for you by cloud providers' services
- Sketch how Merkle trees allow **data integrity checking**
  - Specifically that they are more efficient than sets of hash values

Scale-out design (recall elasticity lecture)

- Consider software design for issuing concert tickets
  - Assume that a flash crowd of 100,000 customers arrives
  - **Ticket count** and **ticket allocations** need to be consistent
- Traditional relational DB? Many **contending transactions**
  - Locking will serialise customers' requests (likely causing timeouts)
- Try to create designs that avoid contention:
  - Allocate **batches of tickets** to servers; or hash customers to seats
  - Note: increment & decrement of ticket count is **commutative**

Building scale-out systems

- Need to characterise parts of workflow carefully: e.g.,
  - **Embarrassingly parallel**—coordination of workers not required
  - **Partitionable**—workers can be coordinated within partitions
  - **Tightly coupled**—whole system needs coordination
- Large scale usually needs highly concurrent operation
  - Can’t require serialisation, but typically must be **serialisable**
  - Systems requiring coordination must handle **machine failures**
  - … also must operate without **software race conditions**

Challenges / solutions for scale-out systems

- Computers used in data centres are unreliable devices
  - **Electronic malfunctions**: *e.g.*, cosmic radiation bit-flips in RAM
  - **Software malfunctions**: *e.g.*, operating system crashes
  - Scaling out over multiple machines: more likely to see failures
  - Also, assessment of failure might be wrong & **device recovers**
- Use **quorum** over set of machines: reduce risk of failures
  - A set of machines carries out computation redundantly
  - Determine that **a majority agree** before proceeding
  - Expensive to maintain redundancy, but its value is high

Core tool for reliability: atomic broadcast

- **Atomic broadcast**—all correct instances receive same set of messages in the same order (AKA total order)
  - Total order does not imply order matches order messages sent
    - (Partial order just provides a set of "X is before Y" clauses)
  - Equivalent to distributed consensus: agree on message order
- General async. distributed consensus with faulty node?
  - Proven to be **impossible to achieve**—Fischer, Lynch, & Paterson
  - … but can make practical systems if requirements are relaxed
  - Are synchronous solutions: the 'Byzantine Generals' problem

Apache ZooKeeper

- ZooKeeper gives **safe, high-performance coordination**
  - Although technically it is 'just' a **hierarchical key-value store**
  - Key protocol: **ZooKeeper Atomic Broadcast** (ZAB)
  - Set of ZK servers maintain in-memory database of all state
    - Snapshots written to persistent storage for faster server recovery
    - All ZK servers have to know about all other ZK servers
- ZooKeeper was developed as part of Hadoop
  - Hadoop needed to coordinate distributed work being done
  - Early developments ran into subtle coordination failures

ZooKeeper’s guarantees and simple API

- **Sequential consistency**—clients’ updates are in order
- **Atomicity**—clients’ updates apply entirely or not at all
- **Single view**—all servers provide same view of system
  - *i.e.*, clients can connect to any ZooKeeper server
- **Reliable**—updates persist once committed
- **Timely**—all clients’ views up to date within time bound

- Very simple API: create node; delete node; node exists?; get data; set data; get children; sync

Establishing integrity of application's data

- Failure-free system? Components—thus data—is correct
  - However this also means no protection from malicious agents
- Consider **verifying integrity** of files for malicious changes
  - Not sufficiently safe or precise to look at modification times
  - Need to look at the contents of the data in the files
  - Typical approach: summarise files with a **secure hash code**
- Special case: checking append-only log of transactions
  - Related to distributed ledger technology (DLT), *e.g.*, blockchain

Merkle trees: a useful type of hash tree

- Consider data divided up into fixed-sized blocks
  - (Covered in more detail in COSC312 / COSC412 …)
- Rather than hashing each block and sending hash list:
  - **Hash data blocks** (leaves), then **hash concatenated hashes**
  - Binary tree proceeds up to the **root hash**—the handle for data
- Can quickly check blocks within individual branches
  - Do not need to have whole tree: **can reconstruct branch hash**
  - Then can check if new block is **consistent with the root hash**

Merkle trees are widely used

- Can verify **BitTorrent downloads**—the root hash is file ID
  - (currently many torrents are actually a flat list of block hashes)
  - any malicious block manipulations can be easily detected
- Check integrity of **Git repositories**—track modifications
  - (FYI: some Git data is not protected, *e.g.*, branch pointers)
- Verify state of **data in filesystems**, *e.g.*, BTRFS and ZFS
- Used in **bitcoin's blockchain** system—light clients
- Within **NoSQL DBs**: cheaply locate data inconsistencies

Checking consistency of distributed ledgers

- Ledger tracks state of system—e.g., account balances
  - Ledgers are typically **append-only data structures**
    - Immutable history is useful widely, such as auditing DB changes…
  - State of ledger can be checked effectively using **Merkle trees**
    - Newest transaction block checked against hash tree and root hash
- Distributed ledgers (DLT) have multiple copies of ledger
  - Can **quickly & efficiently check all ledgers** are consistent
  - Most DLTs rely on peer-to-peer network: avoid central servers
  - (Large download when starting to mine bitcoins is the ledger)

FYI: Blockchain: types and cloud role

- Public, permissionless blockchains in Ethereum; bitcoin
  - **No central control** over set of participants
  - Need a consensus system such as **proof-of-stake**:
    - compete to solve a hash-puzzle: winner is randomised and verifiable
- Private, permissioned systems more typical in enterprise
  - Understand set of participants and who is allowed to act
  - Can **facilitate BFT consensus** which is stricter than ZooKeeper
- Can use blockchain to **check cloud applications' state**
  - Cloud providers also happily sell (distributed) ledger systems

Amazon Quantum Ledger Database: QLDB

- QLDB: an **append-only** DB with **verified transaction log**
  - Hash records (SHA-256) provided over transaction history
  - **Not a DLT**: QLDB is centralised infrastructure; one data owner
  - API is server-agnostic: Amazon will scale server-side as needed
- Pricing: based on **I/O against data**, and **data storage**
  - I/O: writes—$0.70/mil; reads—$0.136/mil
  - Storage: journal—$0.03/GB/month; index—$0.25/GB/month
- **PartiQL** allows querying of transaction records
  - PartiQL extends SQL to handle semi-structured & nested data