Chapter 1

Scalable Distributed Systems

HPC and mainframes.

- Pros: Single solution, powerful hardware, reliable design
- Cons: £££, failure models, vendor lock-in, adaptability

Distributed Computing. Want to achieve linear scalability, but synchronization, communication (**overheads**) prevent it. Ideally, perfect partition of data and compute (e.g. distribute client/server model on **cheap commodity hardware** (best £/resource) and account for failures and manage the network speed bottleneck in software (parallel algorithms and systems).

Data centers Usually DC will have non-commodity network, but it will still the slowest part in memory hierarchy. Focus on **scaling-out** not up by buying cheap hardware in bulk therefore achieve economy of scale. Elasticity: Resources on-demand (illusion of infinite resource).

1.1 Properties of DS

- Scalability: By aggregation of many resources
  - Compute
  - Storage: distributed file systems
  - Memory: cluster memory (in-memory key/value stores, caches)
  - Bandwidth: DC networks, CDN
- location transparency from the user's perspective (black box API)
- High availability: Mask hardware and software failures
- Composition of services

1.1.1 Answering a google search request

1. Load-balancer routes request to lightly-loaded Google Web Server (GWS)
2. GWS routes search to one Index Server for each shard through load-balancer
3. Results are aggregated (IDs for matching documents), ie ordered by relevance
4. GWS sends appropriate IDs to Doc Servers to retrieve URL, title, summary
5. Results are aggregated (+ad, spell), producing search result page

1.2 Design principles

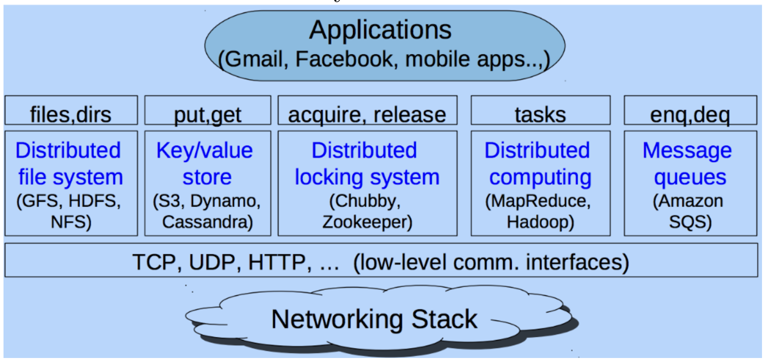
Types of scalable systems

1. Online systems (<100 ms) (OLTP)
2. Batch processing systems (>1 hours) (OLAP)
3. Nearline systems (<1 secs or mins)

Distributed Computing Challenges

- **Scalability**: Independent parallel processing of sub-requests/tasks. More servers permits serving more concurrent requests.
- **Fault tolerance**: Mask and recover from HW/SW failure. Replication data and service.
- **High availability**
- **Consistency**: $/neq$ services but = results.
- **Performance**: **Predictable** low-latency processing with high throughput

Abstractions : Each layer should be itself scalable.



Design principles

- **Stateless services**: Avoid data inconsistency. Separation of data + meta-data, Use lease.
- **Caching**: Latency.
- **Partition/aggregration pattern** (see google search above)
- **Consistency models**: (Most apps use a mix of strong and others). Switch to weaker for latency reasons. Decide what matters (e.g. Order of posts in LinkedIn news feed? Access from multiple devices?)
- **Efficient failure recovery**: Failure is very common, but full redundancy too expensive $\rightarrow$ failure recovery $\Rightarrow$ Rather reduce the cost of failure recovery.
  - Replication: Need to replicate data and service (consistency issues)
  - Recomputation: Use stateless protocols, form data lineage for compute jobs

Chapter 2

Data Center and cloud

Chapter 3

Bigtable

Chapter 4

Dynamo

Chapter 5

Spanner: Globally distributed database

Spanner is a scalable, multi-version, globally-distributed, synchronously-replicated database.

External Consistency/Linearizability If transaction T1 commits before another transaction T2 starts, then T1's commit timestamp is smaller than T2's

5.1 Issues addressed

- Wanted strong consistency (which via CAP theorem meant less availability)
- Wanted ACID transactions
- Wanted schema
- Spanner is an example of NewSQL (SQL like model, but scalability and performance)
- Built for F1 (important app for Google). Was MySQL cluster, had big problems of resharding (done manually) and schema migration.
- Wanted easy geodistribution (coping with whole datacentre failure)
- Wanted to automate the process of replication

5.2 Main Ideas

- TrueTime API
- General-purpose transactions (ACID). Support different transaction types which are optimised e.g. non- blocking reads
  - Write transactions guarantees 'external consistency' (strong form of consistency)
    - Writes buffered.
    - Write locks acquired at commit time (when Paxos prepare is done.
    - But reducing availabilty for this (CAP) - block to wait out the uncertainty.
    - Writes must initiate the Paxos protocol at the leader.
  - Reads access state directly from the underlying tablet at any replcia that is sufficiently up-to-date.
  - Read-Write - requires locks
  - Read-Only - lock free
    - Requires declaration before start of transaction
    - Reads information that is up-to-date

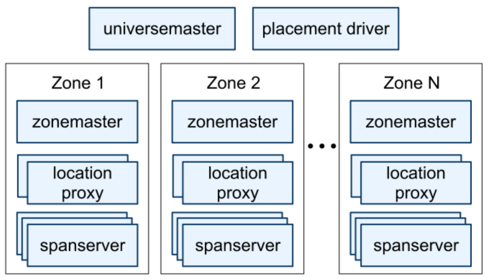
  - Snapshot-Read - read information from past by specifying timestamp or bound
    - Lock free (non-blocking)
    - Globally consistent
    - Use specific timestamp from past or timestamp bound so that data until that point will be read

- Atomic schema changes
- Temporal database. Transaction serialization via global timestamps
- Schematised, semi-relational (tabular) data model
  - SQL-like query interface
- Data is versioned. Each version is automatically timestamped at commit time - while locks are held
- Data chunks - directory/bucket
  - Unit of data movement and for defining replication properties
  - Split into fragments
  - Set of contiguous keys that share a common prefix.
  - Locality encoded in to it - pick keys to get better locality
- Shards data across many sets of Paxos groups
  - Want multiple groups to be able to partition data so you have smaller set of nodes to run Paxos on so its quicker.
  - Also, with multiple paxos groups they can have different ways of replication (nr replicas, location, etc).
- Layering of how transactions are executed.
  - If transaction can be done in one Paxos group then just run on that group.
  - When the transaction involves multiple paxos groups, then use the top layer (which uses strict two phase commit). This means users can do cross-row transactions
- Automatic resharding, rebalancing, failure response
- Globally distributed for high availability and geographic locality.

5.2.1 TrueTime API

- Global timestamps
- Exposes uncertainty in time and guarantees a bound on it
- Timestamp as a range
- Truetime timestamp getting is local, but these timestamps can be globally ordered (by reasoning about clock uncertainty/clock skew and waiting out the uncertainty to avoid overlapping timestamps)
- TrueTime has masters that globally synchronize and decide on the global bound.
- If network partition you lose the synchronization between the truetime masters. So must assume the clock skew increases over time, so over time the wait time will increase to where the system is very slow. (smaller clock uncertainty, larger throughput rate)
- Uses GPS and atomic clocks to get accurate time.

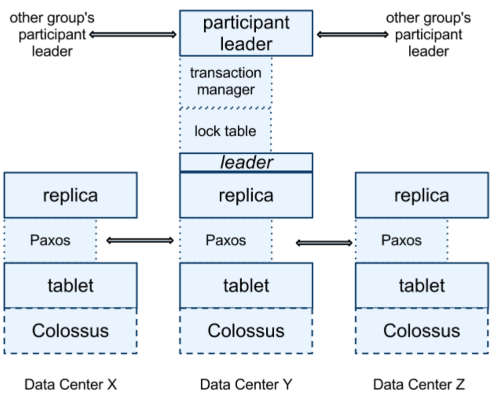
5.3 Additional points



- Universe = Spanner deployment
- Zone = unit of administrative deployment; unit of physical isolation
- Zonemaster = assigns data to spanservers
- Spanservers = serve data to clients
- Location proxies = used by clients to locate the spanservers assigned to serve their data.
- Universe master = console that displays status info about all zones for interactive debugging
- Placement driver = handles automated movement of data across zones
- Tablet: data structure. Bag of key-value mappings (key, timestamp) -¿ string. A container that may encapsulate multiple partitions of the row space, hence its possible to colocate mutiple directories that are frequently accessed together. Tablets are stored on top of the Colossus distributed file system.
- Paxos state machine on each tablet.
- Long-lived leaders with time-based leader leases. If network partition happens, Paxos waits for 10 secs to get new leader.
- Lock table at each replica that is a leader. Contains state for 2-phase locking.
- Transaction manager = used to support distributed transactions
- Participant leader = built on top of transaction manager; used when transactions involve multiple Paxos groups.

As the number of replicas increases, the latency to achieve a quorum becomes less sensitive to slowness at one slave replica.

Shorter lease times would reduce the effect of server deaths on availability, but would require gretaer amounts of lease-renewal network traffic.



Chapter 6

Zookeeper: Wait-free coordination for large scale systems

Coordination examples: Group membership • Leader election • Dynamic Configuration • Status monitoring • Queuing • Critical sections

What is Zookeeper: Highly available, scalable, distributed, configuration, consensus, group membership, leader election, naming, and coordination service.

6.1 Main contributions

- Coordination kernel – Wait-free coordination
- Coordination recipes – Build higher primitives
- Experience with coordination – Some application use ZooKeeper

6.2 Zookeeper properties

- Simplified file system type API
- No partial reads/writes
- Ordered updates and strong persistence guarantees
- Conditional updates (version)
- Watches for data changes
- Ephemeral nodes
- Generated file names
- No renames

6.3 Zookeeper guarantees

- Linearisable writes – Writes serialisable + respect precedence
- FIFO client order
  - Clients never detect old data
  - Clients get notified of change to watched data within bounded time
  - All requests from client processed in order
  - All results received by client consistent with results received by other clients

6.4 Zookeeper model

- Znode
  - In-memory data node
  - Hierarchical namespace
  - Manipulated through the Zookeeper API
  - Types: Regular/Ephemeral (can be automatically removed by system)
  - Flags: Sequential
  - Designed to store only meta-data or configuration, not general data storage
  - CAn store information such as timestamps or version counters to track updates. Support conditional writes based on timestamp/version.
- Watch mechanism
  - Get notification upon update to data
  - One time triggers
- Client sessions
  - Session = connection to server from client
  - Timeout mechanism

Chapter 7

Map reduce

Data flow model. Share data through stable storage - replication, I/O, serialization costs.

Chapter 8

(Spark) Resilient Distributed Datasets: A Fault Tolerant Abstraction for In-Memory Cluster Computing

8.1 Motivation

They want to leverage distributed memory to make it more efficient to resuse intermediate results across multiple computations. Iterative MapReduce runtimes etc perform data sharing implicitly for the pattern of computation they support, and do not provide a general that the user can employ to share data of their choice among operations of their choice. Wanted to allow big data analysis but with:

- More complex, multi-stage applications that reuse intermediate results (e.g. iterative ml, graph processing)
- More interactive ad-hoc queries (interactive data mining where a user runs many ad-hoc queries on the same data)

Hence needed efficient primitives for data sharing (in MapReduce the only way to share data across jobs is stable storage - slow due to replication and disk I/O, but necessary for fault tolerance). Goal: fault-tolerant and efficient distributed memory abstraction. In-memory data sharing.

With fine-grained updates to mutable state, one can only provide fault tolerance through replicating the data across machines or logging updates across machines. These are expensive for data-intensive workloads cause they require copying large amounts of data over the cluster network and they incur substantial storage overhead.

8.2 Resilient Distributed Datasets(RDDS)

- Distributed memory abstraction that lets programmers perform in-memory computations on large clusters in a fault-tolerant manner.
- Enables efficient data reuse. Lets users explicitly persist intermediate results in memory, control their partitioning to optimize data placement and manipulate them using a rich set of operators.
- Store data lineage ins

8.3 Representing RDDs

- partitions
- dependencies on pa

8.4 Representing RDDs

  * partitions
  * dependencies on parent RDDs
  * function for computing the dataset based on its parents
  * metadata about its partitioning scheme - hash/range partitioned
  * preferredLocations(p) - list nodes where partition p can be accessed faster due to data locality

  Dependencies are narrow(each partition of parent is used by at most one partition of the child RDD) or wide(multiple partitions may depend on it). Narrow good for pipelined execution on one node and less recomputing needed upon recovery.rent RDDs

- function for computing the dataset based on its parents
- metadata about its partitioning scheme - hash/range partitioned
- preferredLocations(p) - list nodes where partition p can be accessed faster due to data locality

Dependencies are narrow(each partition of parent is used by at most one partition of the child RDD) or wide(multiple partitions may depend on it). Narrow good for pipelined execution on one node and less recomputing needed upon recovery.tead of data

- Recompute data based on lineage. Used in fault recovery - much quicker and less expensive than using replication. Only the lost partitions of an RDD need to be recomputed upon failure and can be recomputed in parallel on different nodes.
- Partitioned across nodes
- Immutable to simplify lineage tracking
- Can only be built (created, written) through coarse-grained, deterministic transformations (map, filter, join etc). This restricts RDDs to apps that perform bulk writes, but allows for more efficient fault-tolerace.
- Note that reads can be coarse- or fine-grained.
- Checkpointing to disk to avoid unbounded lineage
- Can efficiently express many parallel algos (these apply the same operator to many items). Unify many programming models and support new apps too.
- Best for batch workloads (coarse granularity, memory bandwidth levels of write throughput(so quite high))
- Straggler mitigation: Since RDDs are immutable the system can run backup copies of slow tasks (like in MapReduce). Hard to do with distributed shared memory cause two copies of a task would access the same memory locations and interfere with each others' updates.

8.5 Representing RDDs

- partitions
- dependencies on parent RDDs
- function for computing the dataset based on its parents
- metadata about its partitioning scheme - hash/range partitioned
- preferredLocations(p) - list nodes where partition p can be accessed faster due to data locality

Dependencies are narrow(each partition of parent is used by at most one partition of the child RDD) or wide(multiple partitions may depend on it). Narrow good for pipelined execution on one node and less recomputing needed upon recovery.

8.6 Spark programming interface

- DryadLINQ-like API in Scala
- Usable interactively from scala interpreter
- Provides:
  - RDDs
  - Operations on RDDs: transformations (build new RDDs), actions (compute and output results to app/storage system)
  - Control each RDDs partitioning (layout across nodes) and persistence (storage in RAM, on disk, replicating across machines etc.). Can co-partition (e.g. hash both on the joining field) RDDs that are e.g. repeatedly joined to avoid shuffles. Persistent RDDs can be storage in-memory as deserialized Java objects, in-memory as serialized data or on-disk.

8.7 Spark scheduler

In bulk operations on RDDs, a runtime can schedule tasks based on data locality to improve performance. Creates DAG of stages to execute (narrow, wide dependencies, shuffle stages). Then launches tasks to compute missing partitions from each stage until it has computed the target RDD.

Bibliography