Topics covered in this lecture

- Scalable Distributed Data Structures for Internet Service Construction
  - Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler

Properties that internet services must satisfy

- **Scale** to large, continually growing entities
- **High availability** in the face of partial failures
- **Consistency** of user data
- **Operational manageability**

Challenges to achieving these properties

- Manage **large amounts** of persistent state
- State must be available & consistent despite
  - **Failures** of disks, processes or processors
- Consequences of failing
  - Lost data, angry users, financial liability

Issues with traditional databases and file systems

- Not designed for internet service workloads
- Required service properties are not accounted for
- As a result they fail to provide required guarantees for
  - **Scaling**, consistency, or availability

Distributed Data Structure

- **Reusable, cluster-based** storage layer
- Provides a **single-site** data structure interface to applications
  - Durably manage data behind this layer
  - Distribute and replicate the data transparently
- **DHT** (Distributed Hash Table)
Clusters as a natural platform for internet services

- Each cluster node is an independent failure boundary
- Replicate computations & data for fault tolerance
- Incremental scalability
- Add nodes to linearly increase service capacity
- Parallelism
- If properly balanced all CPUs, disks, and links can be used simultaneously
- High throughput

Internet service workloads that are considered

- Hundreds of millions of tasks per day
- Task is usually small with low compute footprint
- Small amount of data transferred
- Yahoo
  - 625 million page views per day (in 2000)

DDS is a self managing storage layer

- Services interact with DDS as a conventional data structure
- DDS hides mechanisms to:
  - Access, partition, replicate, scale, and recover data
- Developers only worry about service-specific logic

DDS

But ...

- The idea of durable storage layer is not new
- Databases and file systems have accomplished this for decades

ACID

- Atomicity: All or nothing
  - If any part of the transaction fails, abort it
- Consistency
  - Each transaction takes it from one consistent state to another
- Isolation
  - Transactions do not interfere with each other
- Durability
  - Once you commit a transaction, it will remain so
Relational databases (1/2)

- Extremely strong durability and consistency guarantees
- ACID properties derived from transactions
  - ACID come at a high cost in terms of complexity and overhead
- Internet services relying on databases go to great lengths to reduce presented workloads
  - Query caching in the front ends

Relational databases (2/2)

- Layers of RDBMS permit decoupling of logical structure of the data from physical structure
  - SQL parsing, query optimization, access path selection
- From the perspective of service properties
  - RDBMS will choose consistency over availability
  - May be unavailable till failure is resolved

Distributed file systems

- File systems expose low-level interface with little data independence
  - File system = hierarchical directory of files
  - Files = variable-length array of bytes
- Elements (directories & files) are directly exposed to clients
- Clients must structure and interpret application data in terms of
  - Directories, files, bytes inside those files

Design principles in DDS (1/2)

- Separate service code from storage management
  - Decoupling simplifies architecture
- Cluster
  - Must be physically secure and well-administered
  - Low latency SAN (systems area network)
    - In the order of 100 microseconds as opposed to 100 milliseconds over the internet

Design principles in DDS (2/2)

- High throughput and high concurrency
  - Asynchronous, event-driven style of control
    - Process-per-task or thread-per-task do not scale well
  - Use of queues to absorb bursts in data traffic
Assumptions

- If a node cannot communicate with another
  - The second node is considered to have failed
- Network partitions do not occur in the cluster
- Fail-stop behavior in software
  - Terminate if you encounter problems
- Table’s key space is 64-bit integers

Clients

- Client runs service-specific software
  - Client could run inside a web-browser
- Clients are unaware of DDS

Service

- Set of cooperating software services
  - Each of which is called a service instance
- Services have:
  - Soft state
    - May be lost and recomputed if needed
  - Persistent state
    - Managed by the distributed hash table

API has methods for

- put()
- get()
- remove()
- create()
- destroy()

Brick

- Manage durable data
- Manages a set of network accessible single-node hash tables
  - Set of key, value pairs
  - Typically, 1 brick per CPU in the cluster

Partition tables to spread operations and data across bricks

- Each brick stores some number of partitions
  - Of each table in the system
- When new nodes are added?
  - Partitioning is altered
  - Data is migrated/moved to the new node
- If a cluster node fails?
  - A portion of the hash table becomes unavailable
Coping with failures

- Each partition is `replicated` on more than one cluster node
- Replica group: set of nodes for a partition
- Replicas in a group are `strictly coherent`
  - Any replica can be used for a `get`
  - All replicas must be updated during a `put` or `remove`

Synchronous updates of all replicas of a partition

- Two-phase commit protocol
  - `Prepare` message sent first
  - `Commit` messages thereafter
- Operation is aborted
  - If problems encountered during prepare phase
  - Timeout counter elapses

Metadata maps

- Two metadata maps replicated on each node in the cluster
  - `Data partitioning` map
    - Trie, prefix tree, over hash table keys
    - Given a hash table key, return name of partition
  - `Replica group membership` map
    - Given a partition name, return list of bricks serving as replicas for that partition

Recovery (1/3)

- When a brick fails, all replicas on it become unavailable
- Remove failed brick from all replicas
- When brick recovers (or an alternate is found)
  - Must `catch-up` to all missed operations
- Partitions are relatively small
  - 100 MB
  - Possible to transfer entire partitions over the wire

Recovery (2/3)

- Incrementally copy entire partitions to recovering node
  - Avoid undo/redo logs maintained by databases during recovery
- When a node initiates a recovery, it grabs a `write lease` on that partition
  - All state changing operations on that replica will start to fail
Recovery (3/3)

- Recovering node
  - Copies partition over the network
  - Updates replica group for partition
  - Releases write lock
  - Once lock is released, write operations on the partition will succeed

Convergence of recovery (1/2)

- During recovery state changing operations fail on the recovering partition
  - Allows surviving replicas to be consistent
  - Recovering nodes have stable image for recovery
  - Recovery node joins replica group
  - After copying partition, but before releasing lock

Convergence of recovery (2/2)

- If recovery takes longer?
  - Recovering node needs to renew lease
- If recovering node crashes during recovery?
  - Write lease will expire

Asynchrony

- Asynchronous, event-driven processing underlies DDS
  - Layers are separated by FIFO queues
    - I/O requests and responses are placed here

Use of single thread

- Only a single thread ever executes within a hash table layer
  - Eliminates need for data locks and coping with overheads
- Inefficiencies?
Performance (1/2): in-core

- Throughput (ops/sec) scales linearly with number of bricks/nodes
- Graceful degradation of reads
  - Excess traffic absorbed in queues and processed with greater latency
  - Rejected in cases where queue sizes exceeded limits

Performance (2/2): in-core

- Ungraceful degradation of writes
  - Writes go to replicas
  - Even if one of the replicas slows down (e.g. garbage collection) it becomes a bottleneck
- Objects created during prepare phase must wait for one network-round trip
  - For abort or commit
  - Number of live objects is proportional to the delay x bandwidth product

Performance: out-of-core (1/2)

- 130 minutes to fill table with 1.3 TB of data
  - 22,015 ops/sec with 8 KB data elements
- Random read/write throughputs
  - 319 minutes to populate 1.3 TB of data

Performance: out-of-core (2/2)

- Performance also limited by network ceiling
  - Higher bandwidth delays saturation

Paper this lecture was based on