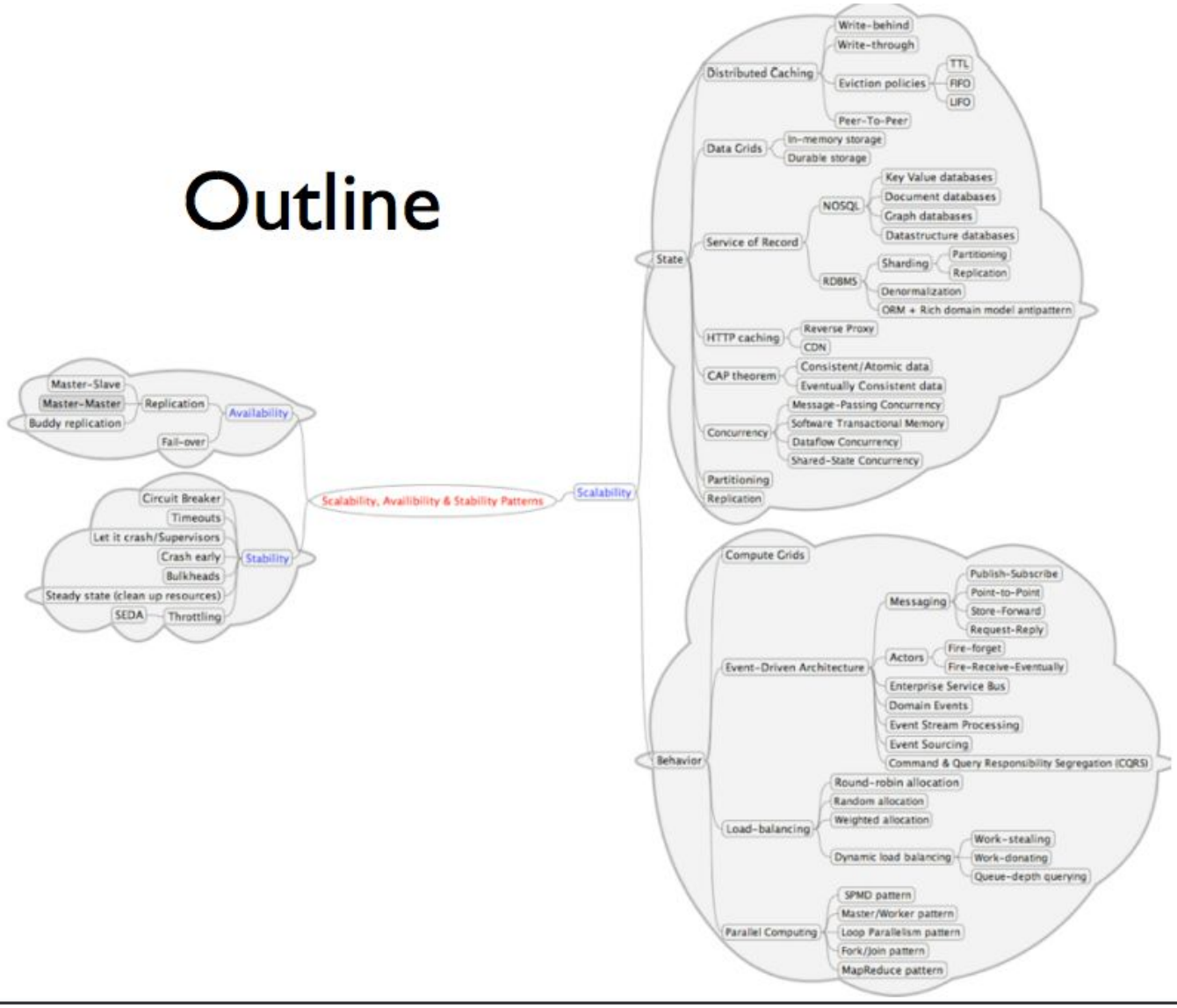
Scalability, Availability, and Stability Patterns

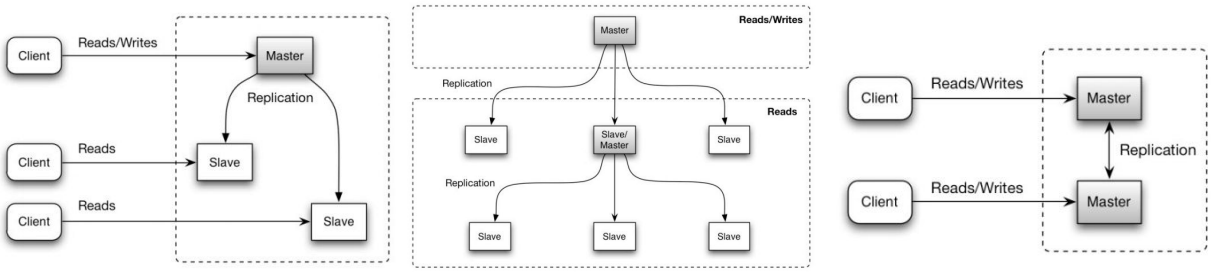
Scalability - Trade Offs

1. Performance vs. scalability
    a. Performance - system is slow for a single user
    b. Scalability - system is fast for a single user, but slow under heavy load
2. Latency vs. Throughput - strive for maximum throughput with acceptable latency
    a. Latency - time for operation
    b. Throughput - number of operations per unit of time
3. Availability vs. Consistency - CAP theorem
    a. Centralized system -
        i. no partitions, hence consistent and available
        ii. ACID
            - Atomic - single transaction with multiple updates is atomic iff either all updates are committed as one, or if any of the updates fail, all updates are rolled back.
            - Consistent - transactions creates a new a valid state of data, or maintains prior state in case of a failure
            - Isolated - uncommitted transaction in process is isolated from other transactions
            - Durable - committed data is saved by the system and recoverable even in case of system failure or system restart
    b. Distributed System -
        i. Always partitioned
        ii. Only consistency or availability is possible. Hence 2 types of systems - CP or AP
        iii. In case of a n/w partition, what do you prioritize C or A.
        iv. BASE - Basically available, Soft-state, Eventually Consistent
        v. Trade-off - highly available with eventual consistency



Availability Patterns

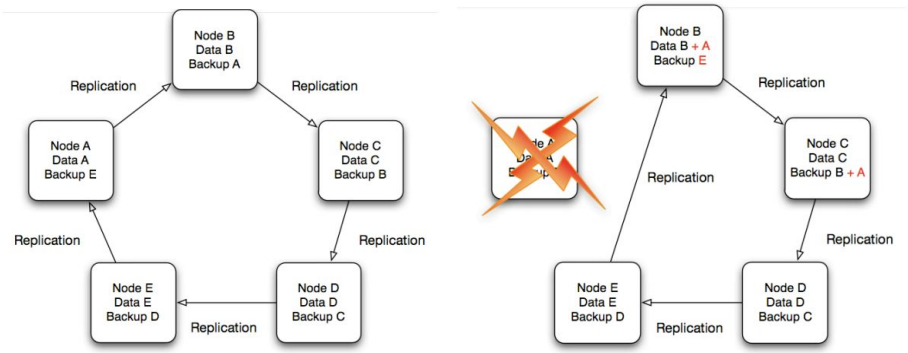
1. Fail over
    a. Fail over
        i. Downtime for passive node to be brought up
        ii. Traffic redirection to passive node
    b. Fail back
        i. Data resynch to primary node
        ii. Traffic redirection to primary node
2. Replication
    a. Active (push) vs. Passive (pull)
    b. Types
        i. Master-slave - Client reads/writes with Master. Master replicated to Slaves. Client Reads from Slaves



ii. Tree replication - Reads/writes with Master. Replicated to Slave. Reads from Slave. A slave can then replicate to more slaves, hence forming a tree structure.

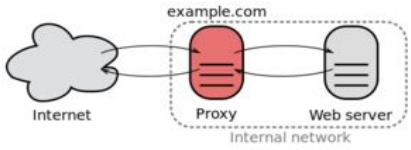
iii. Master-master replication - client reads/writes with multiple masters

iv. Buddy replication - every node has a replica of a peer. When a node (say A) crashes, Node B now becomes responsible for A's data, and also acts a backup for a node (say E), that A was a backup for.

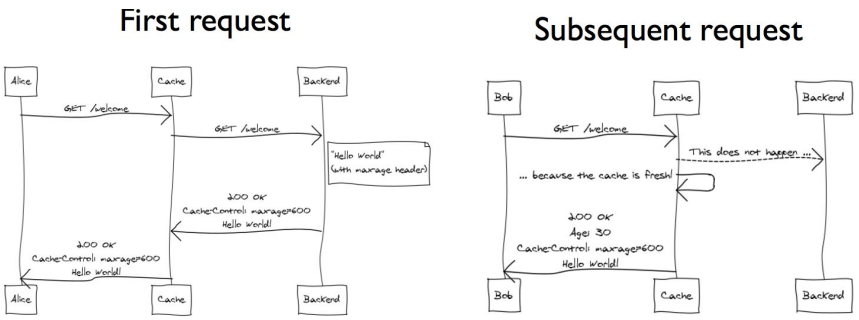


Scalability Patterns - State

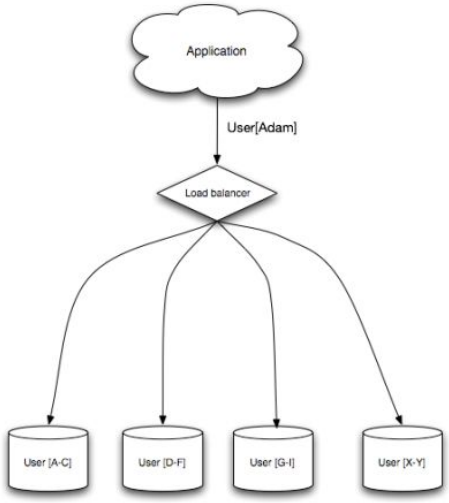
- HTTP Caching
    a. Reverse Proxy - similar to forward proxy - i.e. proxy b/w accessing internet from within an intranet (like at GS). But reverse direction - Reverse proxies are helpful for load balancing, security, caching static content, spoon feeding (i.e. return data to the client a little bit at a time, to allow client to receive and process the data) etc.



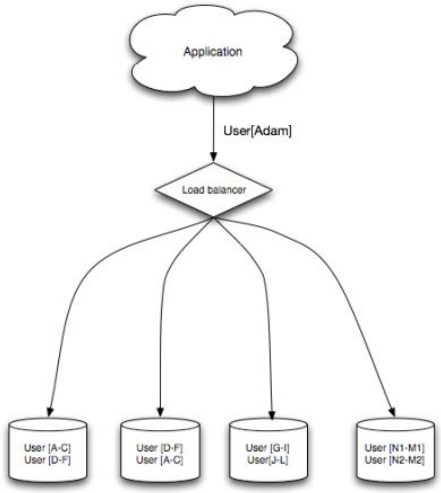
A reverse proxy taking requests from the Internet and forwarding them to servers in an internal network. Those making requests to the proxy may not be aware of the internal network.



- Service of Record (SOR)
  a. RDMS
     i. Partitioning



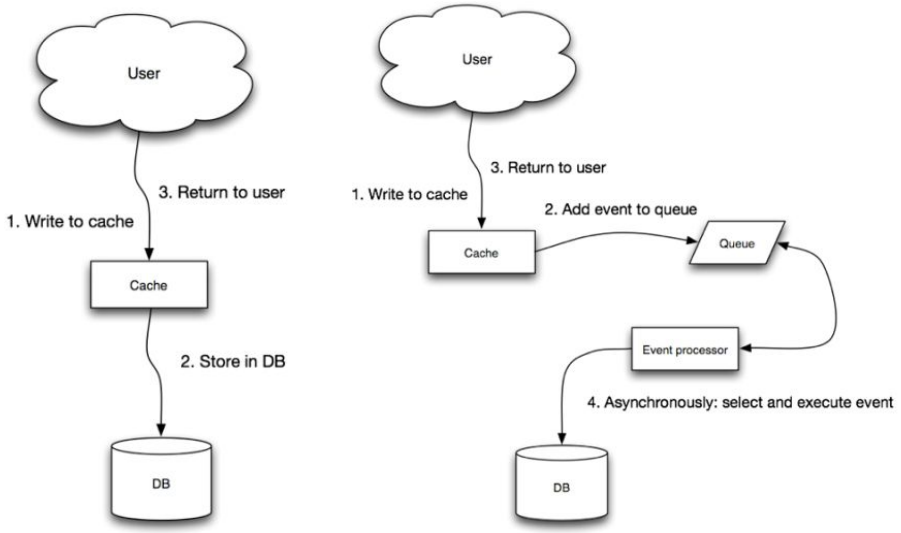
     ii. Replication



     iii. Scaling READS to an RDBMS is hard
     iv. Scaling WRITES to an RDBMS is impossible
  b. NOSQL - Not Only SQL
     i. Key-Value; Column; Document; Graph; Data Structure databases
     ii. Distributed Hash Tables

iii. Consistent Hashing

- Distributed Caching
  - Write through - updates write to the DB through the cache, only then return to the user
  - Write behind - write to cache, return to user. Asynch process writes back to DB



  - Eviction policies
    - TTL
    - Bounded FIFO
    - Bounded LIFO
    - Explicit cache invalidation
  - Peer-to-peer
    - Decentralized
    - Nodes can leave and join as they wish
- Data Grids/Clustering
  - Parallel data storage
  - CP in CAP
- Concurrency
  - Shared-State concurrency
    - Use of locks - i.e. java.util.concurrent.*
  - Message-passing concurrency
    - “Actors” - share nothing, and communicate through messages (actor-based concurrency)
    - Asynchronous and non-blocking
    - No shared state
    - Use of a “mailbox”
  - Dataflow Concurrency
    - Threads block until data is available
  - STM - Software Transactional Memory
    - Similar to DB transactions - begin, commit, abort/rollback
    - Transactions are retried automatically upon collision

Scalability Patterns - Behavior

- Event-Driven Architecture
- Compute Grids
    - Divide and Conquer
    - Map/Reduce
- Load Balancing
- Parallel Computing

Stability Patterns

- Timeouts
    - Always use timeouts
        - Thread.wait(timeout)
        - reentrantLock.tryLock
        - blockingQueue.poll(timeout)
- Circuit Breaker
- Fail Fast
- Steady State
    - Clean up after you; for ex: logging - RollingFileAppender
- Throttling

<u>References</u>

Scalability, Availability & Stability Patterns Jonas Bonér CTO Typesafe

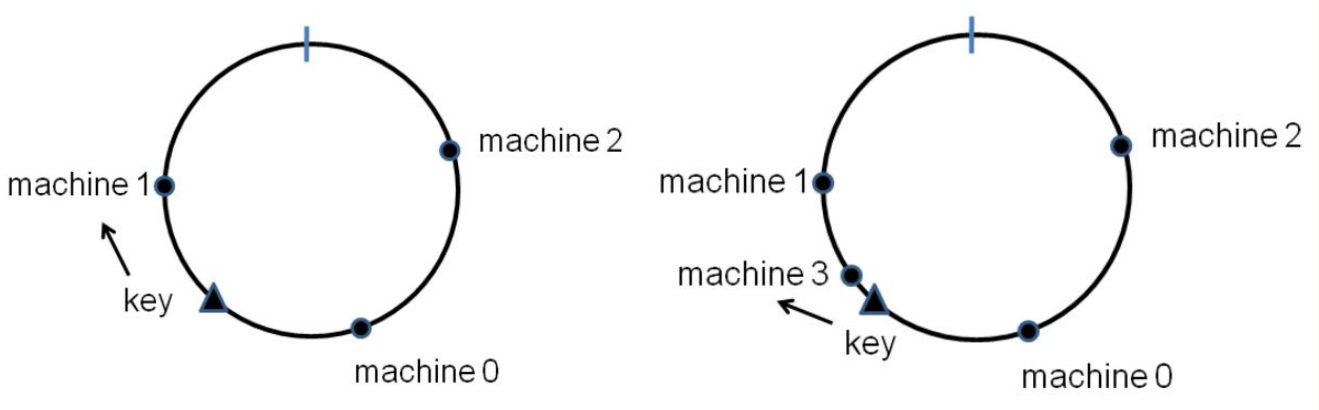
Distributed Hash Tables

- Distributed storage of things with known names
- Highly scalable - automatically distributes load to new nodes
- Robust against node failure - data automatically migrated away from failed nodes
- Self-organizing - no need for centralized server
- Use of consistent hashing, for ex: Chord protocol

Consistent Hashing - mechanism to support a distributed dictionary

- Naive method involves storing key-values across n machines using hash(key)%n. However, this has a problem if machines get added for scaling (or removed due to failures) because adding machines will involve reallocation of keys across n machines + new machines added (or removed). For ex: if 1 new machine is added, this will need relocation of n/(n+1) keys across the cluster.
- Consistent hashing solves this problem.
- Spread machines along a circle [0, 1) using a hash function on the machine id
- If R is the range of hash function, hash key x, and store in the machine closest to hash(x)/R.

- Adding a new machine means only 1/(n+1) keys need to be remapped.



- Drawback -
    - 2 machines can get mapped pretty close to each other, meaning one gets very less keys
    - When a machine gets added to a cluster, all keys come from only one machine
- Solution - each machine j gets mapped to r points (as replicas). Hence the cluster contains j*r machines.
    - Replicas for a machine j get mapped by hashing (j, 0), (j, 1),..., (j, r - 1) across the circle.
    - Increases uniformity with which keys get mapped to machines.
    - Smaller number of keys to be remapped i.e. O(1/(rn)).
- Assumptions
    - Uniform hash function

Server side consistency

N = number of nodes that store replicas of the data
W = number of replicas that need to acknowledge receipt of update before update completed
R = number of replicas that are contacted when a data object is accessed through a read operation

W + R > N ⇒ strong consistency
W + R <= N ⇒ eventual consistency

SQL vs. NoSQL

SQL - related tables, with integrity constraints such as unique key definitions. ACID
NoSQL - data stored as documents, or key-value, or graphs, or data structures. BASE

Where SQL is ideal	**Where NoSQL is ideal**
Logical related discrete data requirements that can be identified up front	Unrelated, indeterminate, or evolving data requirements
Data Integrity is essential	Simpler or looser project objectives, ability to start coding immediately
Harder to scale - partitioning and replication	Speed and scale is imperative

Scalable Web Architecture and Distributed Systems

Principles of Web Distributed Systems Design

- Availability
- Performance and Stability
- Reliability and Consistency
- Manageability
- Cost

Basics

- Services
    - Start off with a single service eg. read and write service for uploading images
    - Break out Read service from Write service. Write usually takes a long time than Read, so scaling is different
    - Web servers usually have upper limits on concurrent connections (e.g. apache allows only 500 simultaneous connections), hence more sense to break out the reads and writes
    - Benefits of using SOA
        - Decouples operation of different functions
        - Clearly establishes relationships b/w them
        - Helps isolate problems
        - Allows independent scaling
- Redundancy/Replicas
    - Replicas for both servers and data storage
    - Helps handle failure gracefully
    - System can fail over to healthy copy
    - Shared-nothing architecture - each nodes behaves independently from each other. No dependency on state of another node
    - Replicas usually in different geographical regions
- Partitions
    - Large data sets cannot fit in a single server
    - Scale vertically - add more resources to an existing server
    - Scale horizontally - add more nodes to the cluster
    - Partitions can be based on different factors, e.g. - geographical locations; paying vs. non-paying customers
    - Distributed Hash Tables, and use Consistent Hashing
    - Downsides
        - Lack of data locality
        - Inconsistency (CAP theorem)

Building Blocks

- Scale access to app/web server
- Scale access to database

- Caches
    - Locality of reference - recently queried data is likely to be queried again
    - Example: storing food in refrigerator (cache), and more food available at the grocery store (database)
    - Global cache vs. distributed cache
    - Global Cache
        - All request nodes pass through the global cache. If data is not present in the cache, the cache itself is responsible for fetching the data from database
        - Or request nodes can query DB, if data not present in the cache.
        -
        - Being single cache, it could get overwhelmed
    - Distributed Cache
        - Each node has its own cache
        - Same data could get cached on multiple nodes
        - Remedy a missing node - definitely needs same data to be cached on multiple nodes
    - Worst case if the cache fails, requests just hit the DB directly. Impacts performance, but not availability.
- Proxies
    - Coordinate requests from multiple servers
    - Collapsed forwarding - batching similar requests to DB. Also collapse requests for data that are spatially close together
    - Doesn't store results, but optimizes the incoming requests
    - Proxies and Caches can be used together, but in general cache is in front of proxy
- Indexes
    - Optimize data access performance
    - Finding small payload in large data set
    - Indexes do occupy space and can get expensive to store
    - Creating indexes for multiple attributes can add very quickly in terms of disk space usage
- Load Balancer
    - How to manage user-session-specific data - needs to be smart enough to route to a node where the latest user update might be present. This is because the user update might not have propogated to all the nodes yet.
- Queues
    - Enables async processing of data
    - Eventual consistency
    - Support retry of requests in case of any transient failures.

Tips for a system design problem

1. Define the use cases
    a. MoS's for functionality to support
    b. SLAs for performance, availability, and consistency
2. Start with a single user use case
    a. Focus on correctness, and improving performance

    b. Service based architecture
3. High Level Architecture Diagram
    a. Services
    b. Components
    c. Data Storage Layer
4. Component Design
    a. Specific APIs
    b. OO Design
    c. Database schema design
5. Understand Bottlenecks and Scale to more users
    a. Break out reads vs writes in to different services; also potentially by client type i.e. web vs mobile.
    b. Introduce redundancy for data and processing through replication - improves consistency and availability
    c. Introduce partitioning to support large data sets by breaking them out across servers
    d. Talk about CAP theorem - decide on supporting availability vs. consistency
    e. Leverage building blocks
        i. Caches
        ii. Proxies
        iii. Load Balancers
        iv. DB Indexes
        v. Queue for async persisting the data into a DB or a filesystem
6. Other aspects - Code, Documentation, Testing, SDLC process