Scalable Web Architectures(SWA)

COMP 599 - Graduate Seminar, Fall 2018

Rihan Pereira, MSCS

Department of Computer Science
California State University, Channel Islands

March 2, 2019

Roadmap

1. The landscape of SWA
   - Why I switched my topic
   - Why bother studying SWA?
2. Vertical & Horizontal scaling
3. Beyond single node deployment
   - Replication
   - Sharding/Partitions
4. Some strategies to scale up high-traffic web systems
   - Using Caches
   - Proxies
   - Indexes
   - Load Balancers
   - Queues
   - Naive Hashing
   - Consistent Hashing
5. References
6. Thats it

- These systems are designed to handle heavy web traffic, tolerate faults and yet continue to run

- These systems are designed to handle heavy web traffic, tolerate faults and yet continue to run
- Uses lot of research from Distributed Systems

- These systems are designed to handle heavy web traffic, tolerate faults and yet continue to run
- Uses lot of research from Distributed Systems
- SWA has a solid plan when failure happens - unreliable network, power outage, etc.

- These systems are designed to handle heavy web traffic, tolerate faults and yet continue to run
- Uses lot of research from Distributed Systems
- SWA has a solid plan when failure happens - unreliable network, power outage, etc.
- There is no right/wrong answer when building such systems; uses design principles to benchmark solutions to the system design problems it encounters.

- These systems are designed to handle heavy web traffic, tolerate faults and yet continue to run
- Uses lot of research from Distributed Systems
- SWA has a solid plan when failure happens - unreliable network, power outage, etc.
- There is no right/wrong answer when building such systems; uses design principles to benchmark solutions to the system design problems it encounters.

- Availability
- Partial Tolerance
- Consistency
- scalability
- Maintenance

Why I switched my topic

- At first, I had decided to present concepts in blockchain, I have barely scratched the surface of blockchain from technical standpoint.
- I am still learning, its huge, exciting.

Why I switched my topic

- At first, I had decided to present concepts in blockchain, I have barely scratched the surface of blockchain from technical standpoint.
- I am still learning, its huge, exciting.
- Its still early, this space doesn't have a strong talent yet - smart people flock to study Machine Learning :)

Why I switched my topic

- At first, I had decided to present concepts in blockchain, I have barely scratched the surface of blockchain from technical standpoint.
- I am still learning, its huge, exciting.
- Its still early, this space doesn't have a strong talent yet - smart people flock to study Machine Learning :)
- Much of the innovation is happening outside academia

Why I switched my topic

- At first, I had decided to present concepts in blockchain, I have barely scratched the surface of blockchain from technical standpoint.
- I am still learning, its huge, exciting.
- Its still early, this space doesn't have a strong talent yet - smart people flock to study Machine Learning :)
- Much of the innovation is happening outside academia
- demand exceeds supply

- Thinking about scaling in advance before its a requirement makes systems unnecessarily complex without any benefit.

- Thinking about scaling in advance before its a requirement makes systems unnecessarily complex without any benefit.
- Althought, some forethought into the design can save substantial time and resources in the future.

- Thinking about scaling in advance before its a requirement makes systems unnecessarily complex without any benefit.
- Althought, some forethought into the design can save substantial time and resources in the future.
- As a software developer, especially backend engineer, will be dealing with Distributed Systems at some point in their career path

- Thinking about scaling in advance before its a requirement makes systems unnecessarily complex without any benefit.
- Althought, some forethought into the design can save substantial time and resources in the future.
- As a software developer, especially backend engineer, will be dealing with Distributed Systems at some point in their career path
- A large population of backend engineer wont have the opportunity to build such systems from scratch, but you still need to tune knobs(configure) correctly, take advantage of features to achieve that sweet spot.

- Thinking about scaling in advance before its a requirement makes systems unnecessarily complex without any benefit.
- Althought, some forethought into the design can save substantial time and resources in the future.
- As a software developer, especially backend engineer, will be dealing with Distributed Systems at some point in their career path
- A large population of backend engineer wont have the opportunity to build such systems from scratch, but you still need to tune knobs(configure) correctly, take advantage of features to achieve that sweet spot.

Vertical & Horizontal scaling

Horizontal

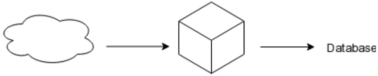
- ability to redirect request to another nodes (in short, resiliency) *
- load balancing
- network calls RPC/REST
- data consistency issues
- system scales proportional to variable data sets *

Vertical

- has a single point of failure
- N/A
- inter-process communication (IPC) *
- consistent *
- hardware limit

Beyond single node deployment

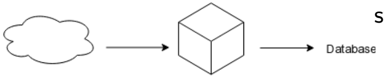
Imagine a simple image upload service using a central server



Very soon, your appln is a hit!
You start getting lot of traffic

Beyond single node deployment

Imagine a simple image upload service using a central server

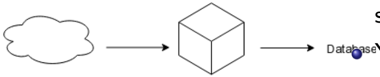


Very soon, your appln is a hit! You start getting lot of traffic

- Your service dont have high profit margins yet(bad reason, couldnt think of something else), so need to make sure the system design is cost-effective

Beyond single node deployment

Imagine a simple image upload service using a central server

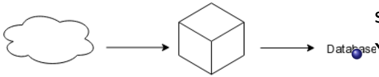


Very soon, your appln is a hit! You start getting lot of traffic

- Your service dont have high profit margins yet(bad reason, couldnt think of something else), so need to make sure the system design is cost-effective
- Your system should be high available - no zero tolerance

Beyond single node deployment

Imagine a simple image upload service using a central server

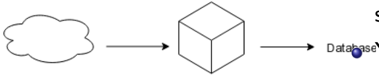


Very soon, your appln is a hit! You start getting lot of traffic

- Your service dont have high profit margins yet(bad reason, couldnt think of something else), so need to make sure the system design is cost-effective
- Your system should be high available - no zero tolerance
- Upon image upload, image should be always there (you get what you put)

Beyond single node deployment

Imagine a simple image upload service using a central server



Very soon, your appln is a hit! You start getting lot of traffic

- Your service dont have high profit margins yet(bad reason, couldnt think of something else), so need to make sure the system design is cost-effective
- Your system should be high available - no zero tolerance
- Upon image upload, image should be always there (you get what you put)
- low-latency request-response

Replication

Replication

keeps a copy of the same data on multiple machines connected via a network.

Why you want to do it ?

Replication

Replication

keeps a copy of the same data on multiple machines connected via a network.

Why you want to do it ?

- keep data geographically close to your users (reduce latency)

Replication

Replication

keeps a copy of the same data on multiple machines connected via a network.

Why you want to do it ?

- keep data geographically close to your users (reduce latency)
- Availability

Replication

Replication

keeps a copy of the same data on multiple machines connected via a network.

Why you want to do it ?

- keep data geographically close to your users (reduce latency)
- Availability
- Increased read throughput

Replication

Replication

keeps a copy of the same data on multiple machines connected via a network.

Why you want to do it ?

- keep data geographically close to your users (reduce latency)
- Availability
- Increased read throughput

Challenge lies in handling changes to replicated data.

Replication

Replication

keeps a copy of the same data on multiple machines connected via a network.

Why you want to do it ?

- keep data geographically close to your users (reduce latency)
- Availability
- Increased read throughput

Challenge lies in handling changes to replicated data.

1. Single-leader based replication

Replication

Replication

keeps a copy of the same data on multiple machines connected via a network.

Why you want to do it ?

- keep data geographically close to your users (reduce latency)
- Availability
- Increased read throughput

Challenge lies in handling changes to replicated data.

1. Single-leader based replication
2. Multi-leader replication

Replication

Replication

keeps a copy of the same data on multiple machines connected via a network.

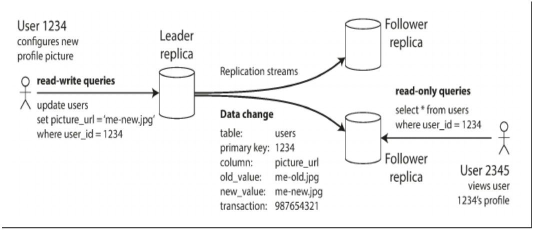
Why you want to do it ?

- keep data geographically close to your users (reduce latency)
- Availability
- Increased read throughput

Challenge lies in handling changes to replicated data.

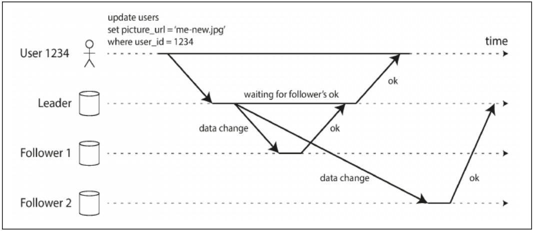
1. Single-leader based replication
2. Multi-leader replication
3. Leaderless replication

Single-Leader based replication



- Also called master-slave replication
- Considering synchronous/asynchronous config in databases.
- Using synchronous configuration is bad
- Async configuration is widely used in production deployments.

leader follower copying strategy



Follower setup without downtime

Follower setup without downtime

- Take a snapshot of the leader database at a specific time intervals

Follower setup without downtime

- Take a snapshot of the leader database at a specific time intervals
- copy the snapshot to the new follower node.

Follower setup without downtime

- Take a snapshot of the leader database at a specific time intervals
- copy the snapshot to the new follower node.
- followers connect to leaders & retrieves data changes happened since the snapshot was taken

Follower setup without downtime

- Take a snapshot of the leader database at a specific time intervals
- copy the snapshot to the new follower node.
- followers connect to leaders & retrieves data changes happened since the snapshot was taken
- when followers has processed the backlog of data changes since the last snapshot was taken, it is in sync.

Follower setup without downtime

- Take a snapshot of the leader database at a specific time intervals
- copy the snapshot to the new follower node.
- followers connect to leaders & retrieves data changes happened since the snapshot was taken
- when followers has processed the backlog of data changes since the last snapshot was taken, it is in sync.

Handling node outage in Leader-based replication

How do you achieve high availability in this architecture ?

Handling node outage in Leader-based replication

How do you achieve high availability in this architecture ?

- Follower failure: Catch-up recovery

Handling node outage in Leader-based replication

How do you achieve high availability in this architecture ?

- Follower failure: Catch-up recovery
  - Each follower maitains a backlog

Handling node outage in Leader-based replication

How do you achieve high availability in this architecture ?

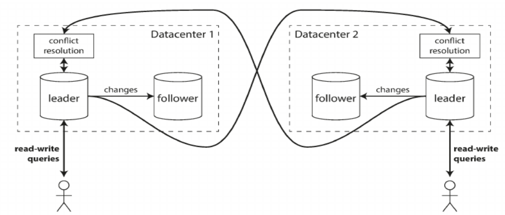
- Follower failure: Catch-up recovery
    - Each follower maitains a backlog
- Leader failure

Handling node outage in Leader-based replication

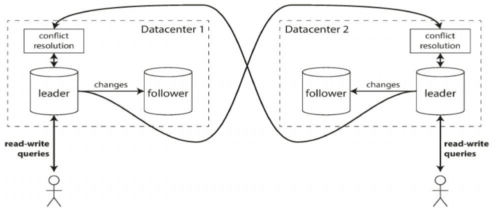
How do you achieve high availability in this architecture ?

- Follower failure: Catch-up recovery
  - Each follower maitains a backlog
- Leader failure
  - one of its followers takes the leader
  - clients need to be reconfigured to send writes to new leader
  - other followers need to start consuming data changes from new leader.
  - old leader joins back as normal follower

Multi-leader based replication

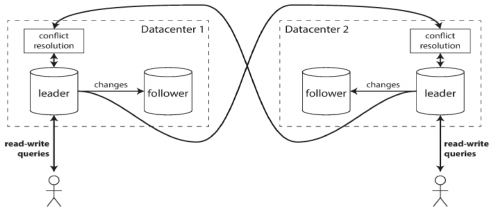


Multi-leader based replication



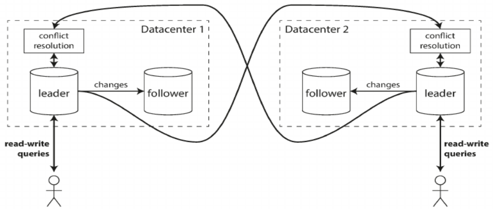
- natural extension of leader-based replica, allows more than 1 node to accept writes

Multi-leader based replication



- natural extension of leader-based replica, allows more than 1 node to accept writes
- each leader simultaneously acts as follower to each leader

Multi-leader based replication



- natural extension of leader-based replica, allows more than 1 node to accept writes
- each leader simultaneously acts as follower to each leader
- mostly used in multi-data center operations

Conflict resolution problem

Conflict resolution problem

- all replicas must arrive at the same final value.
- avoid conflict all-together

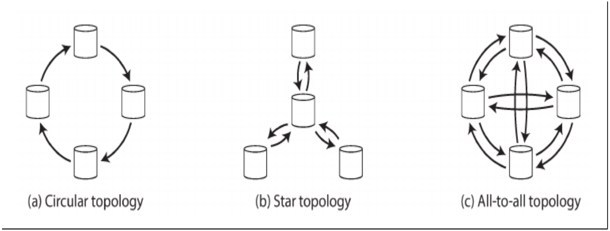
Conflict resolution problem

- all replicas must arrive at the same final value.
- avoid conflict all-together
  - all writes for a particular record go through the same leader.
- In worst case, you have to deal with concurrent writes

Conflict resolution problem

- all replicas must arrive at the same final value.
- avoid conflict all-together
  - all writes for a particular record go through the same leader.
- In worst case, you have to deal with concurrent writes
- There is some research on conflict resolving
  - Conflict-free replicated data types
  - Mergeable persistent data structures
  - Operation transformation

Multi-leader topologies

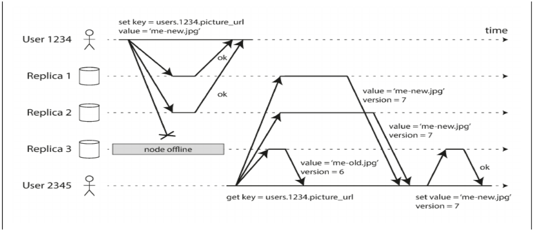


(a) Circular topology

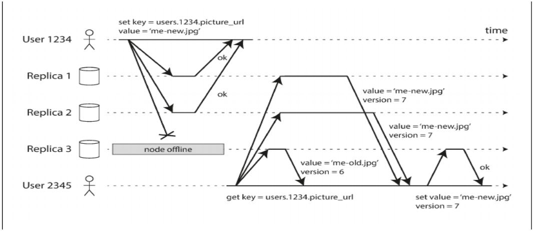
(b) Star topology

(c) All-to-all topology

Leaderless replication

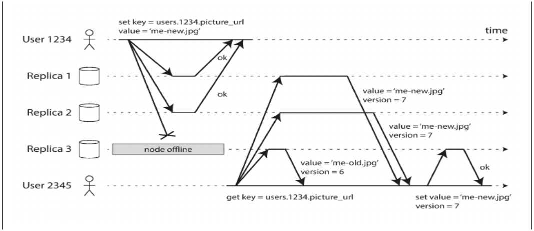


Leaderless replication



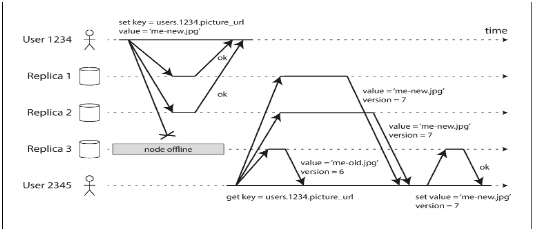
- takes different approach instead of developing leader-follower concept

Leaderless replication



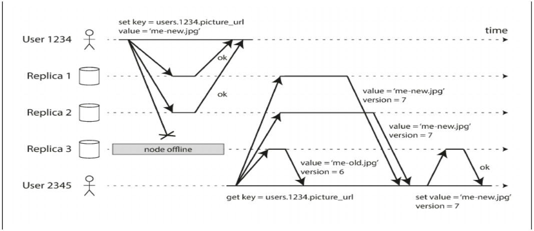
- takes different approach instead of developing leader-follower concept
- clients sends writes to replicas (can use broker to do this)

Leaderless replication



- takes different approach instead of developing leader-follower concept
- clients sends writes to replicas (can use broker to do this)
- No restriction on ordering of writes

Leaderless replication



- takes different approach instead of developing leader-follower concept
- clients sends writes to replicas (can use broker to do this)
- No restriction on ordering of writes
- Amazon's DynamoDB is built using this concept
- examples include Cassandra, Riak

Leaderless write mechanisms

- 3 replicas, 1 unavailable. how do you make up for unavailable replica ?

Leaderless write mechanisms

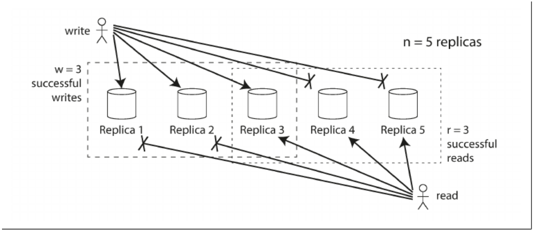
- 3 replicas, 1 unavailable. how do you make up for unavailable replica ?
- **Read Repair**
    - version 6 value from replica 3, version 7 value from replica 1 & 2 write version 7 to replica 3

Leaderless write mechanisms

- 3 replicas, 1 unavailable. how do you make up for unavailable replica ?
- **Read Repair**
  - version 6 value from replica 3, version 7 value from replica 1 & 2 write version 7 to replica 3
- **Anti-entropy process**
  - does reconcilation in background process
  - writes are copied in unordered way

Quorums for reading and writing

Write processed on 2 out of 3 replicas. What if only 1 of 3 replicas accepted write ? How far can we push ?

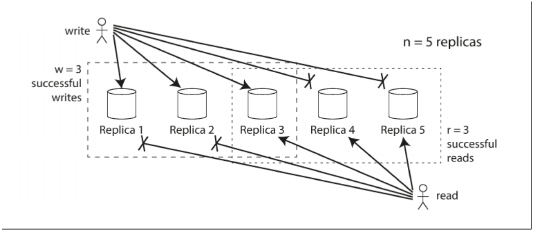


Quorums for reading and writing

Write processed on 2 out of 3 replicas. What if only 1 of 3 replicas accepted write ? How far can we push ?

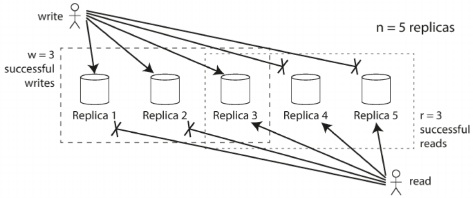
Corollary

$w + r > n$



Quorums for reading and writing

Write processed on 2 out of 3 replicas. What if only 1 of 3 replicas accepted write ? How far can we push ?



Corollary

$w + r > n$

- If w $<$ n, we can still process writes if a node is missing
- If r $<$ n, we can still process reads if a node is missing
- n $=$ 3, w $=$ 2, r $=$ 2, we can tolerate 1 unavailable node
- n $=$ 5, w $=$ 3, r $=$ 3, we can tolerate 2 unavailable nodes

Sharding/Partitioning

Very large datasets, having high query throughput are broken down into partitions or shards.

Sharding/Partitioning

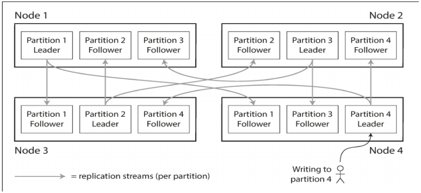
Very large datasets, having high query throughput are broken down into partitions or shards.

- think of each partition as a small database of its own
- approaches for partitioning large datasets
  - Partitioning by key range
  - Partitioning by hash of a key

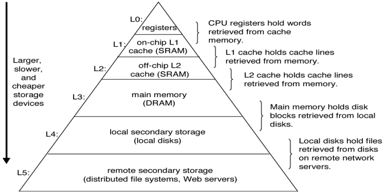
Sharding/Partitioning

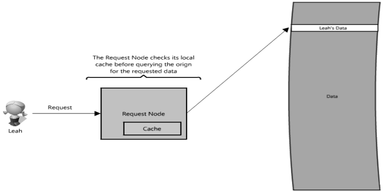
Very large datasets, having high query throughput are broken down into partitions or shards.

- think of each partition as a small database of its own
- approaches for partitioning large datasets
  - Partitioning by key range
  - Partitioning by hash of a key



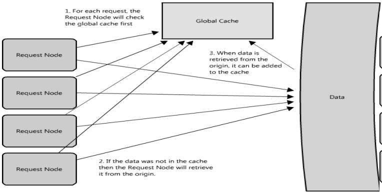
Using Caches


L0:
registers
CPU registers hold words retrieved from cache memory.
L1:
on-chip L1 cache (SRAM)
L1 cache holds cache lines retrieved from memory.
L2:
off-chip L2 cache (SRAM)
L2 cache holds cache lines retrieved from memory.
L3:
main memory (DRAM)
Main memory holds disk blocks retrieved from local disks.
L4:
local secondary storage (local disks)
Local disks hold files retrieved from disks on remote network servers.
L5:
remote secondary storage (distributed file systems, Web servers)
Larger, slower, and cheaper storage devices


The Request Node checks its local cache before querying the orign for the requested data
Leah
Request
Request Node
Cache
Leah's Data
Data

- any serious app will deploy a cache server
- usually placed in front of original data source
- algorithms like LRU(online algorithms) are widely used
- problem - If your system design uses load balancer, you will need to overcome high cache misses

Overcoming cache misses


1. For each request, the Request Node will check the global cache first
Global Cache
3. When data is retrieved from the origin, it can be added to the cache
Request Node
Request Node
Request Node
Request Node
Data
2. If the data was not in the cache then the Request Node will retrieve it from the origin.

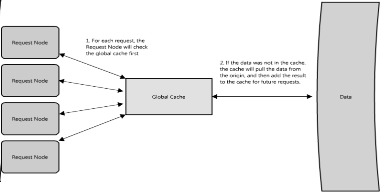

Request Node
Request Node
Request Node
Request Node
1. For each request, the Request Node will check the global cache first
Global Cache
2. If the data was not in the cache, the cache will pull the data from the origin, and then add the result to the cache for future requests.
Data

figure 1

figure 2

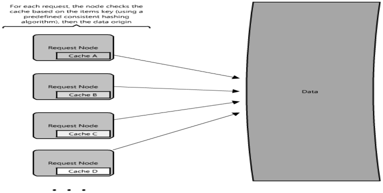

For each request, the node checks the cache based on the items key (using a predefined consistent hashing algorithm), then the data origin
Request Node
Cache A
Request Node
Cache B
Request Node
Cache C
Request Node
Cache D
. . .
Data

figure 3

1st figure handles eviction and data retrieval on its own

2nd figure application handles eviction

3rd figure each cache holds a portion of cached data

3rd figure uses consistent hashing to lookup data across nodes

Proxies

basic role is to receive requests from client and relay them to next node in line.

Proxies

basic role is to receive requests from client and relay them to next node in line.

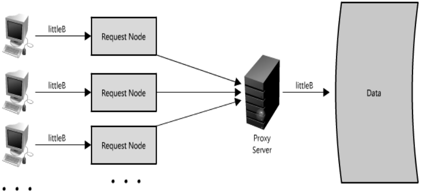


figure 1

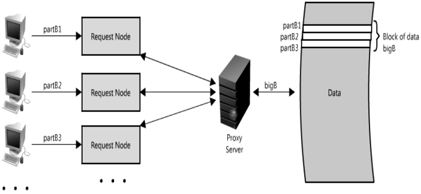


figure 2

1st figure collapse similar requests into a single request

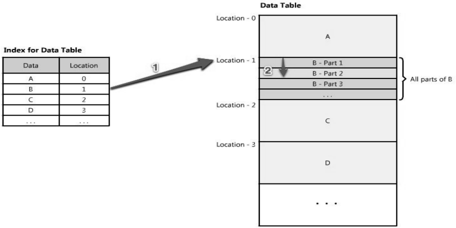
2nd figure collapse requests for data that is spatially close together

Indexes

A very common, popular and important technique used to speedup data access

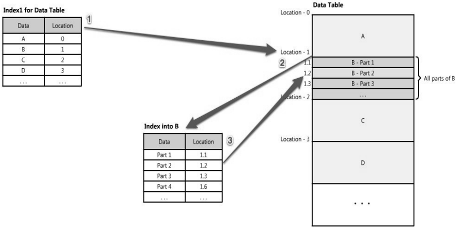
Indexes

A very common, popular and important technique used to speedup data access



Index for Data Table

Data	Location
A	0
B	1
C	2
D	3
...	...



Index1 for Data Table

Data	Location
A	0
B	1
C	2
D	3
...	...

Index into B

Data	Location
Part 1	1.1
Part 2	1.2
Part 3	1.3
Part 4	1.6
...	...

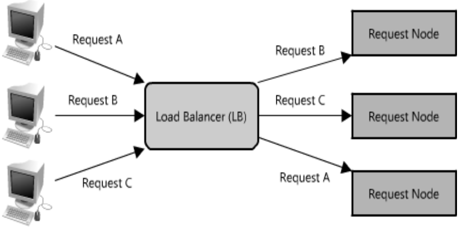
- the trick is to index your database based on users data access patterns
- In return for faster data access they do add write overhead and requiring updating indexes on each write

Load Balancers

Their role is to distribute incoming requests evenly, fairly among available servers. They serve as a brokers between client and logical nodes, handling lot of simultaneous connections.

Load Balancers

Their role is to distribute incoming requests evenly, fairly among available servers. They serve as a brokers between client and logical nodes, handling lot of simultaneous connections.



Algorithms used -

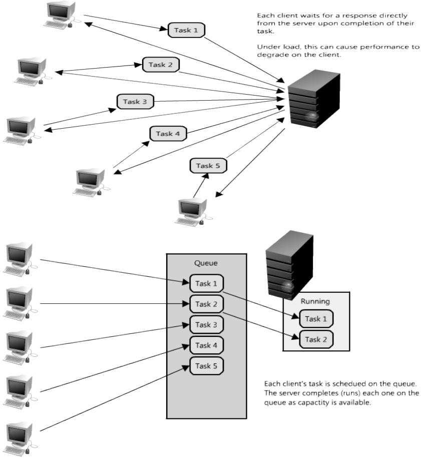
- round robin
- just pick a random node
- selecting a node on a criteria - CPU, memory

Queues

Queues solve a unique problem that load balancing, adding/removing servers cant solve.

Queues

Queues solve a unique problem that load balancing, adding/removing servers cant solve.



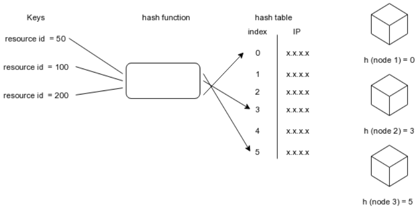
Each client waits for a response directly from the server upon completion of their task.

Under load, this can cause performance to degrade on the client.

Each client's task is scheduled on the queue. The server completes (runs) each one on the queue as capactity is available.

- work in async manner
- client is given acknowledgement that request is served
- tasks range from as simple as write to a data store to as complex as extracting pdf from text

Naive Hashing



index	IP
0	x.x.x.x
1	x.x.x.x
2	x.x.x.x
3	x.x.x.x
4	x.x.x.x
5	x.x.x.x

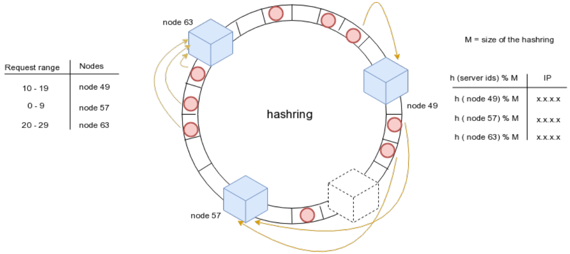
General hashing

some assumptions -

- no. of nodes serving request never change
- repeated request can be served using cache
- requires rehashing every single key, caches get obselete.

Consistent Hashing

Incoming requests and serving nodes are placed onto a virtual ring structure called *hashring*



Request range	Nodes
10 - 19	node 49
0 - 9	node 57
20 - 29	node 63

h (server ids) % M	IP
h (node 49) % M	x.x.x.x
h (node 57) % M	x.x.x.x
h (node 63) % M	x.x.x.x

Consistent Hashing

Graphic By: Rihan Pereira

- placement of server nodes is not fixed on the ring, instead are placed at random locations
- each server owns a range of hashring
- No worries on adding new servers or server disruptions
- only rehashing of affected portion of requests is required

- Martin Kleppmann,
  Designing Data Intensive applications, **2017**
- James Hamilton,
  On Designing and Deploying Internet-Scale Services, **2007**.
- Twitter,
  Automatic Management of Partitioned, Replicated Search Services, **2011**.
- http://blog.acolyer.org/,
  The morning Paper.
- http://aosabook.org,
  The architecture of open source applications.
- http://book.mixu.net/distsys,
  Distributed Systems: fun and profit.

Thank you! Questions ?