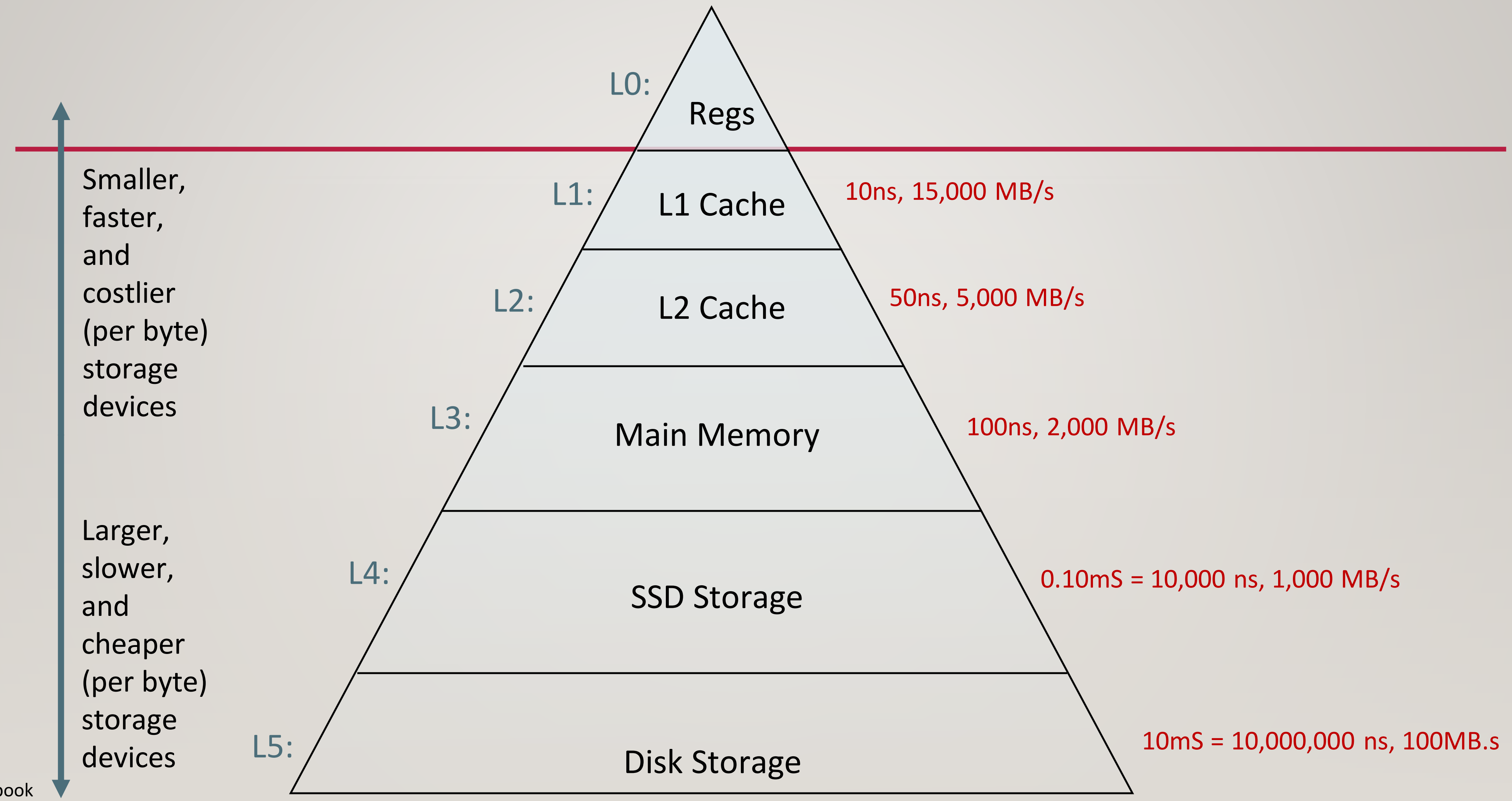
MEMORY, MEMCACHED, REDIS, ELASTICACHE

14-848 CLOUD COMPUTING, FALL 2019

KESDEN

REMEMBER THE MEMORY HIERARCHY



L0:
Regs
L1:
L1 Cache
10ns, 15,000 MB/s
L2:
L2 Cache
50ns, 5,000 MB/s
L3:
Main Memory
100ns, 2,000 MB/s
L4:
SSD Storage
0.10mS = 10,000 ns, 1,000 MB/s
L5:
Disk Storage
10mS = 10,000,000 ns, 100MB.s
Smaller, faster, and costlier (per byte) storage devices
Larger, slower, and cheaper (per byte) storage devices

Adapted from:
15/18-213, CS:APP Textbook

THE ROLE OF MEMORY

- Not so Big Data?
  - In many cases, if we pre-process it, filter it, and use a denser representation, it is not so big
- In other cases, our old friend (Caching) comes to the rescue
  - Spatial locality
  - Temporal locality
- In other cases, data isn't persistent
  - Soft values

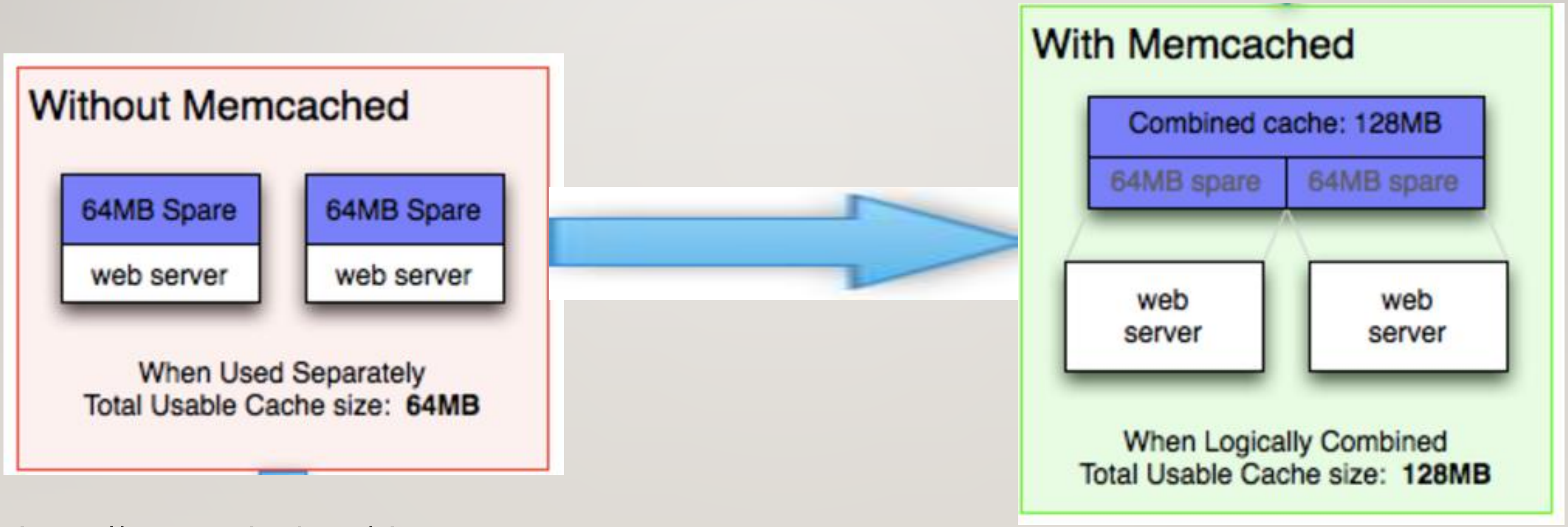
MEMCACHED

- A venerable solution
- Simple and effective

CLASSIC PROBLEMS

- Web servers query from disk
  - Main memory is wasted
- Underlying databases queries can be redundant and slow
- Distributing load among Web servers partitions main memory
  - Redundant memory cache, not larger memory cache
- Dedicated caching can often maintain whole, or large portions of, the working set, limiting the need to go to disk

COMMON USE OF MEMCACHED (BUT, DEDICATED SERVERS ARE ALSO COMMON)



https://memcached.org/about

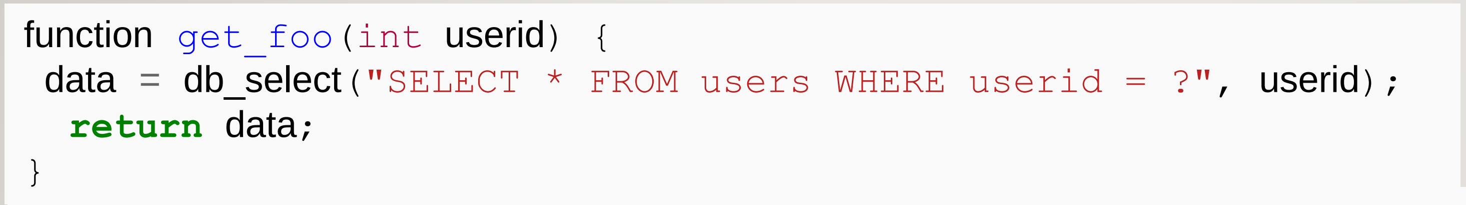
MEMCACHED OVERVIEW

- Distributed hashtable (Key-Value Store)
- Except that "Forgetting is a feature"
  - When full – LRU gets dumped
- Excellent for high-throughput servers
  - Memory is much lower latency than disk
- Excellent for high-latency queries
  - Caching results can prevent the need to repeat these big units of work

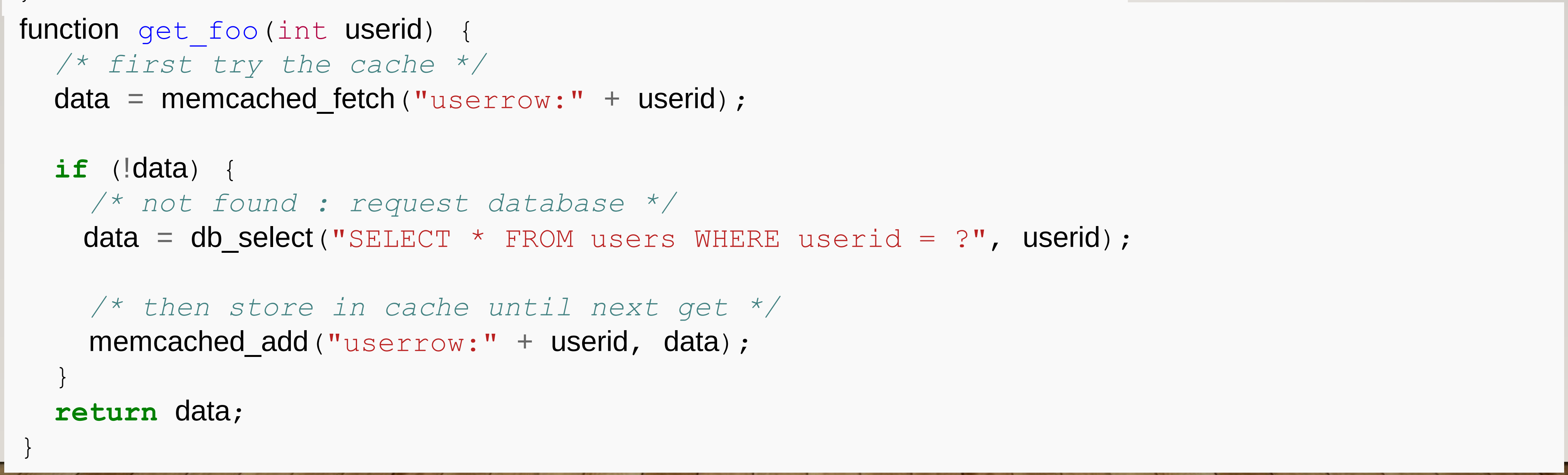
MEMCACHED ARCHITECTURE

- Servers maintain a “key-value” store
- Clients know about all servers
- Clients query server by key to get value
- Two hash functions
  - Key--->Server
  - Key-->Associative Array, within server
- All clients know everything.

CODE, FROM WIKIPEDIA



```
function get_foo(int userid) {
 data = db_select("SELECT * FROM users WHERE userid = ?", userid);
  return data;
}
```



```
function get_foo(int userid) {
   /* first try the cache */
   data = memcached_fetch("userrow:" + userid);

   if (!data) {
     /* not found : request database */
     data = db_select("SELECT * FROM users WHERE userid = ?", userid);

     /* then store in cache until next get */
     memcached_add("userrow:" + userid, data);
   }
   return data;
}
```

MEMCACHED: NO REPLICATION

- Designed for volatile data
  - Failure: Just go to disk
  - Recovery: Just turn back on and wait
- Need redundancy?
  - Build above memcached
  - Just build multiple instances

REDIS
REMOTE DICTIONARY SERVER

- Like Memcached in that it provides a key value store
- But, much richer
  - Lists of strings
  - Sets of strings (collections of non-repeating unsorted elements)
  - Sorted sets of strings (collections of non-repeating elements ordered by a floating-point number called score)
  - Hash tables where keys and values are strings
  - HyperLogLogs used for approximated set cardinality size estimation.
  - (List from Wikipedia)

REDIS
REMOTE DICTIONARY SERVER

- Each data type has associated operations, e.g. get the one in particular position in a list, intersect sets, etc.
- Transaction support
- Configurable cache policy
  - LRU-ish, Random, keep everything, etc.

REDIS: PERSISTENCE

- Periodic snapshotting to disk
- Append-only log to disk as data is updated
  - Compressed in background
- Updates to disk every 2 seconds to balance performance and risk
- Single process, single thread

REDIS CLUSTERING: OLD SCHOOL

- Old School
  - Divide up yourself
    - Clients hash
    - Clients divide range
    - Clients Interact with Proxy which does the same
    - Hard to handle queries involving multiple keys

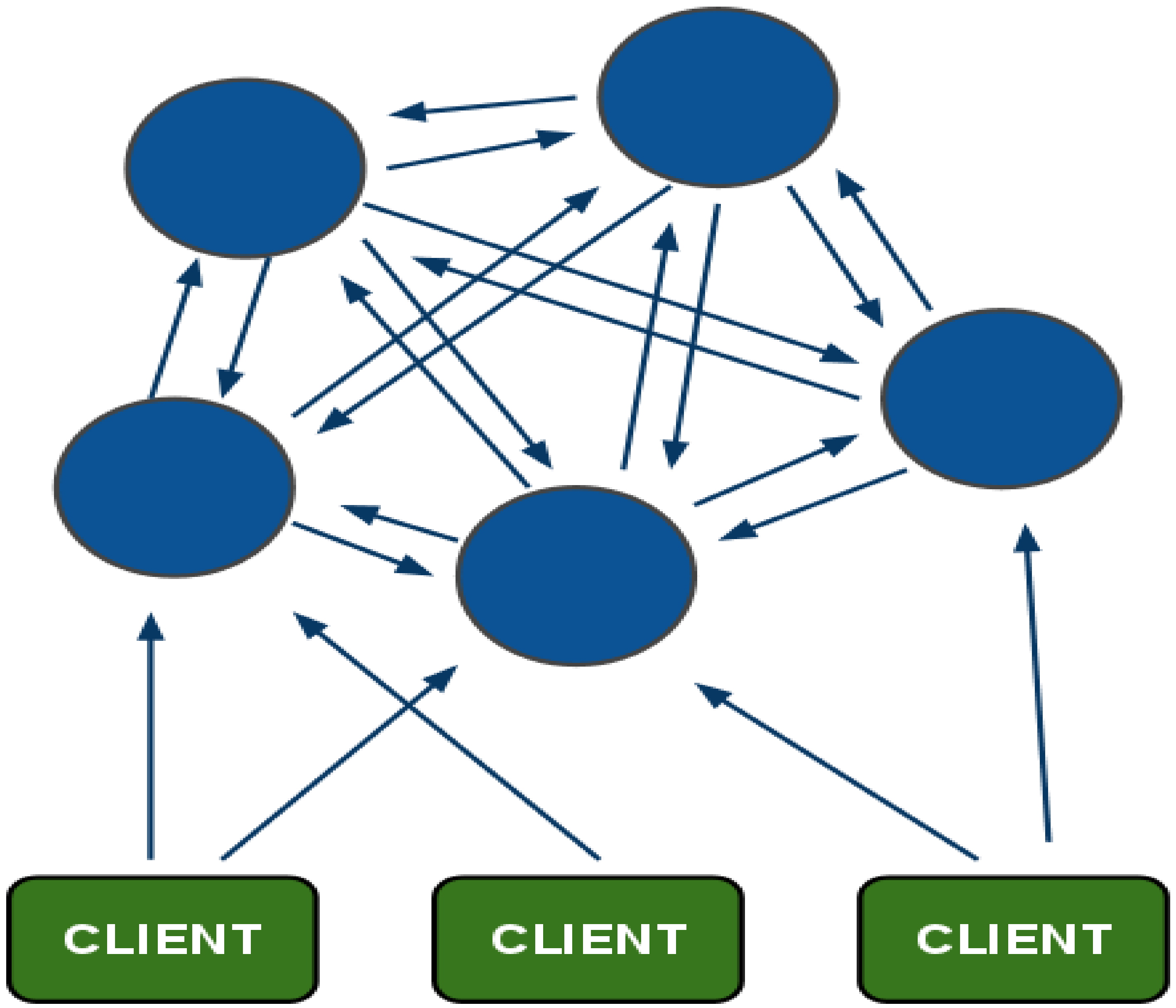
REDIS CLUSTERING: NEW SCHOOL

- REDIS Cluster
  - Distribute REDIS across nodes
  - Multiple key queries okay, so long as all keys in query in same slot
  - Hash tags used to force keys into the same slot.
  - this{foo}key and that{foo}key in the same slot
    - Only {foo} is hashed

Redis Cluster

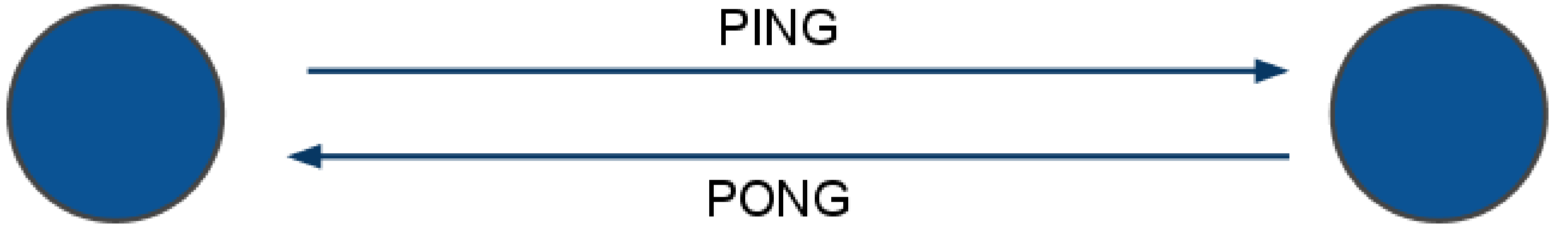
a pragmatic approach to distribution

https://redis.io/presentation/Redis_Cluster.pdf



- All nodes are directly connected with a service channel.
- TCP baseport+4000, example 6379 -> 10379.
- Node to Node protocol is binary, optimized for bandwidth and speed.
- Clients talk to nodes as usually, using ascii protocol, with minor additions.
- Nodes don't proxy queries.

What nodes talk about?



PING: are you ok dude?
I'm master for XYZ hash slots.
Config is FF89X1JK

Gossip: this are info about other nodes I'm in touch with:

A replies to my ping, I think its state is OK.
B is idle, I guess it's having problems but I need some ACK.

PONG: Sure I'm ok dude!
I'm master for XYZ hash slots.
Config is FF89X1JK

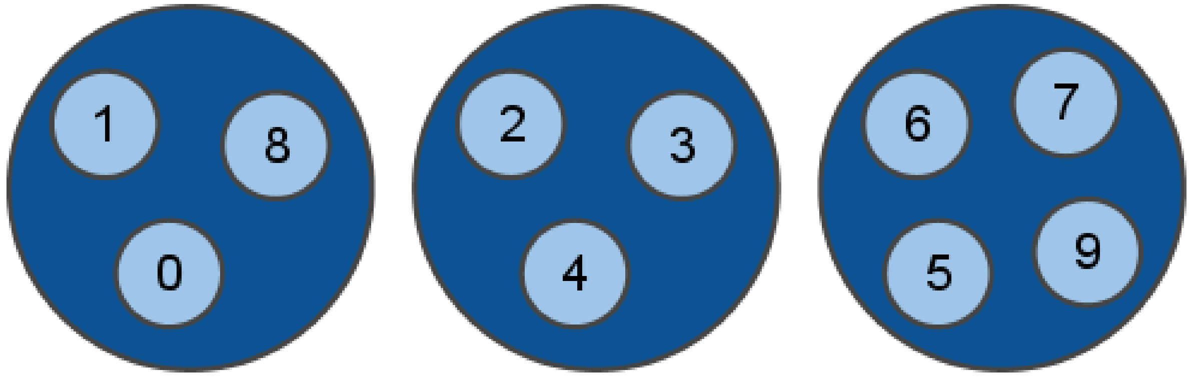
Gossip: I want to share with you some info about random nodes:

C and D are fine and replied in time.
But B is idle for me as well!
IMHO it's down!.

Hash slots

keyspace is divided into 4096 hash slots. But in this example we'll assume they are just ten, from 0 to 9 ;)

Different nodes will hold a subset of hash slots.

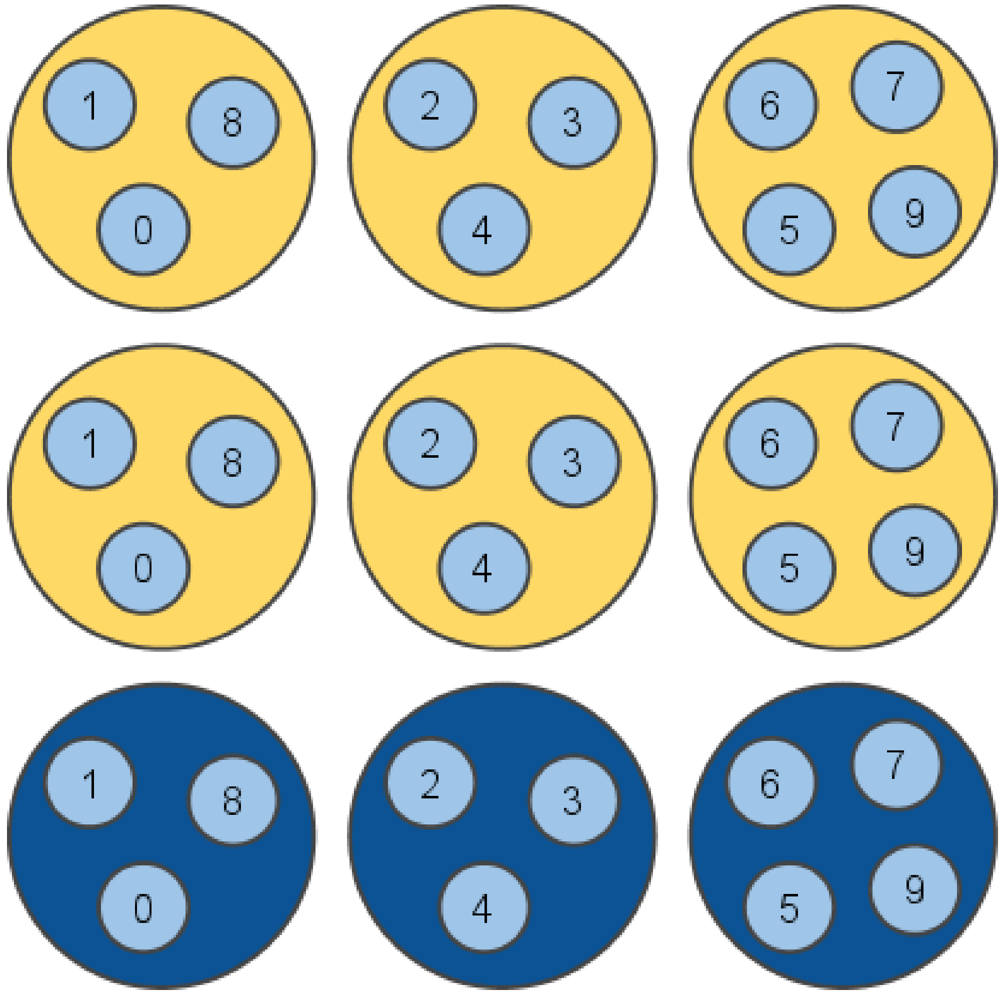


A given key "foo" is at slot:

slot = crc16("foo") mod NUMER_SLOTS

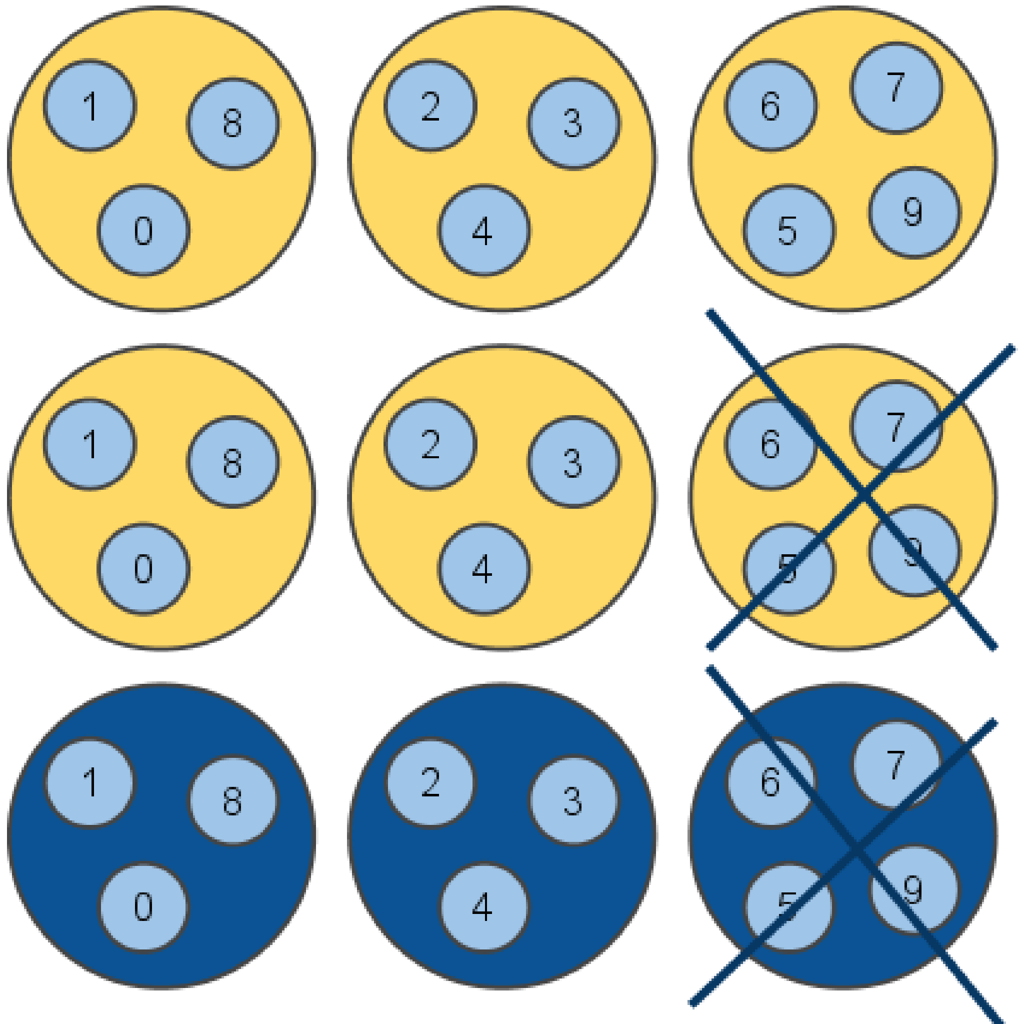
Master and Slave nodes

Nodes are all connected and functionally equivalent, but actually there are two kind of nodes: slave and master nodes:



Redundancy

In the example there are two replicas per every master node, so up to **two random nodes can go** down without issues.

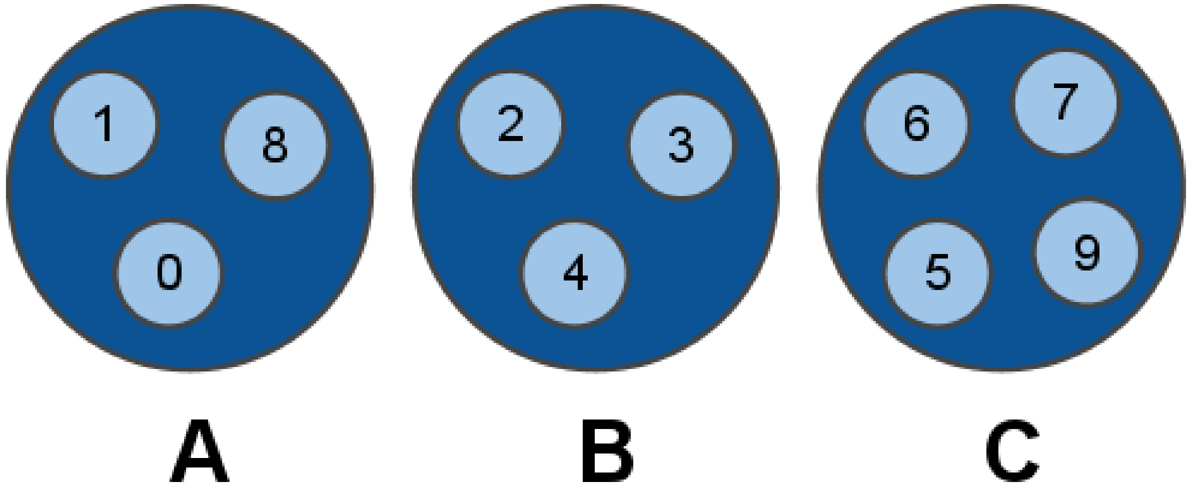


Working with two nodes down is guaranteed, but in the best case the cluster will continue to work as long as there is at least one node for **every hash slot**.

What this means so far?

- Every key only exists in a single instance, plus N replicas that will never receive writes. So there is **no merge, nor application-side inconsistency resolution**.
- The price to pay is not resisting to net splits that are bigger than *replicas-per-hashslot* nodes down.
- Master and Slave nodes use the Redis Replication you already know.
- Every physical server will usually hold multiple nodes, both slaves and masters, but the *redis-trib* cluster manager program will try to allocate slaves and masters so that the replicas are in different physical servers.

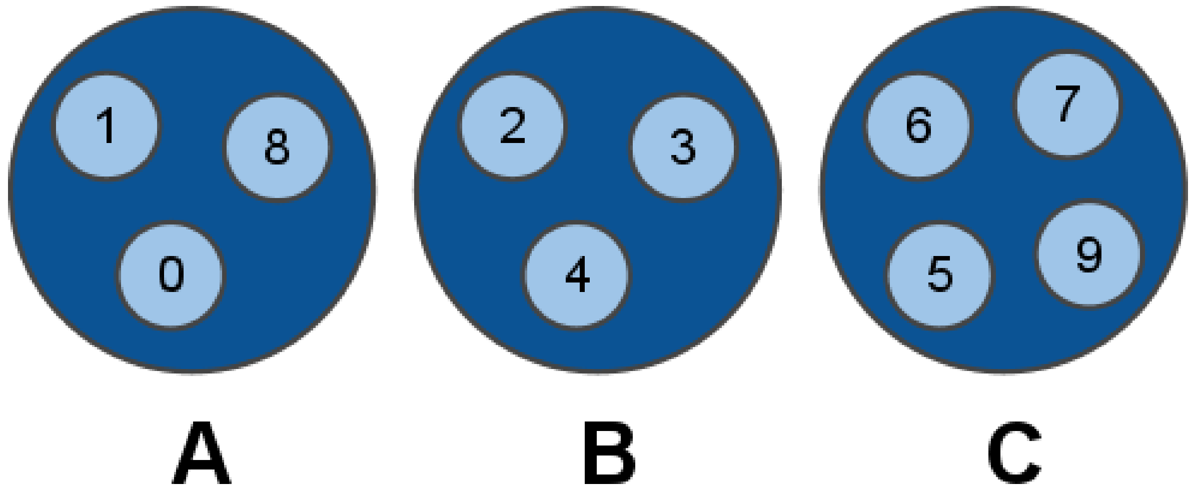
Client requests - dummy client



1. Client => A: **GET foo**
2. A => Client: **-MOVED 8 192.168.5.21:6391**
3. Client => B: **GET foo**
4. B => Client: **"bar"**

-MOVED 8 ... this error means that hash slot 8 is located at the specified IP/port, and the client should reissue the query there.

Client requests - smart client



1. Client => A: **CLUSTER HINTS**
2. A => Client: **... a map of hash slots -> nodes**
3. Client => B: **GET foo**
4. B => Client: **"bar"**

Client requests

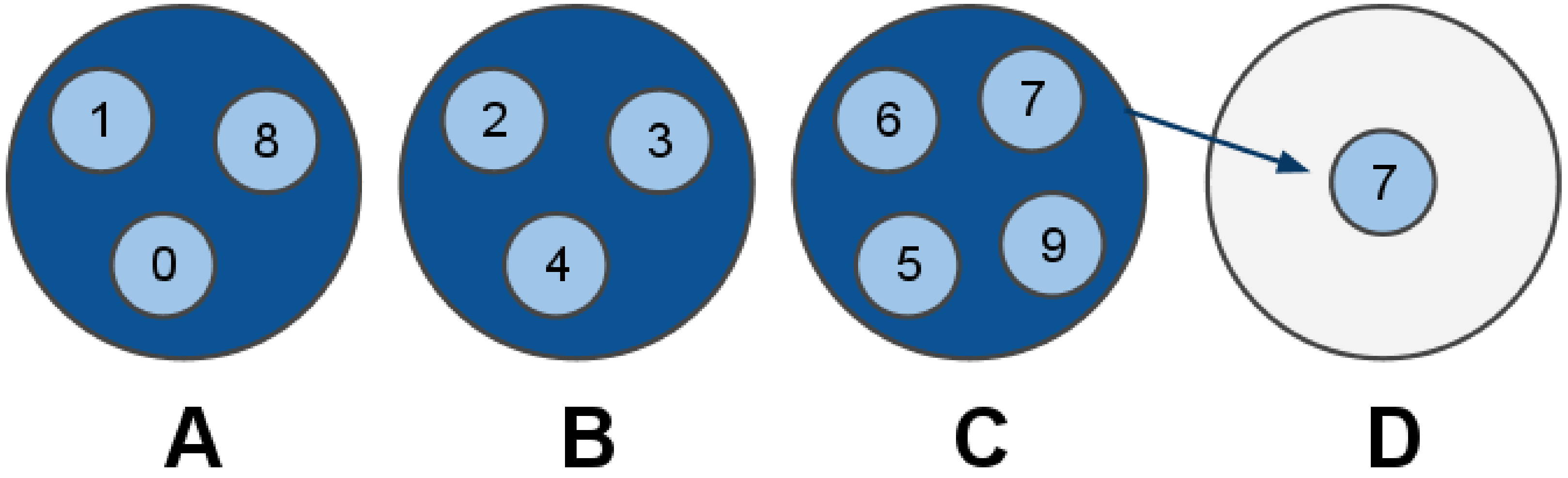
- Dummy, single-connection clients, will work with minimal modifications to existing client code base. Just try a random node among a list, then reissue the query if needed.

- Smart clients will take persistent connections to many nodes, will cache *hashslot -> node* info, and will update the table when they receive a -MOVED error.

- This schema is always horizontally scalable, and low latency if the clients are smart.

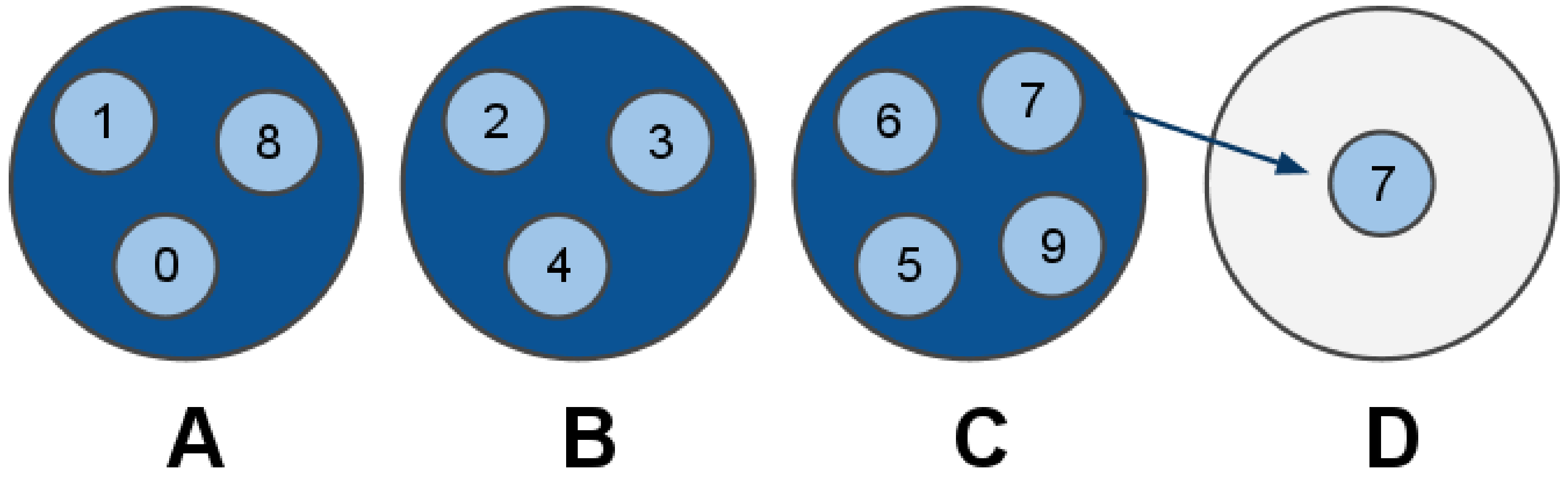
- Especially in large clusters where clients will try to have many persistent connections to multiple nodes, the Redis client object should be shared.

Re-sharding



- We are experiencing too much load. Let's add a new server.
- Node C marks his slot 7 as "MOVING to D"
- Every time C receives a request about slot 7, if the key is actually in C, it replies, otherwise it replies with -ASK D
- -ASK is like -MOVED but the difference is that the client **should retry against D only this query**, not next queries. That means: smart clients should not update internal state.

Re-sharding - moving data

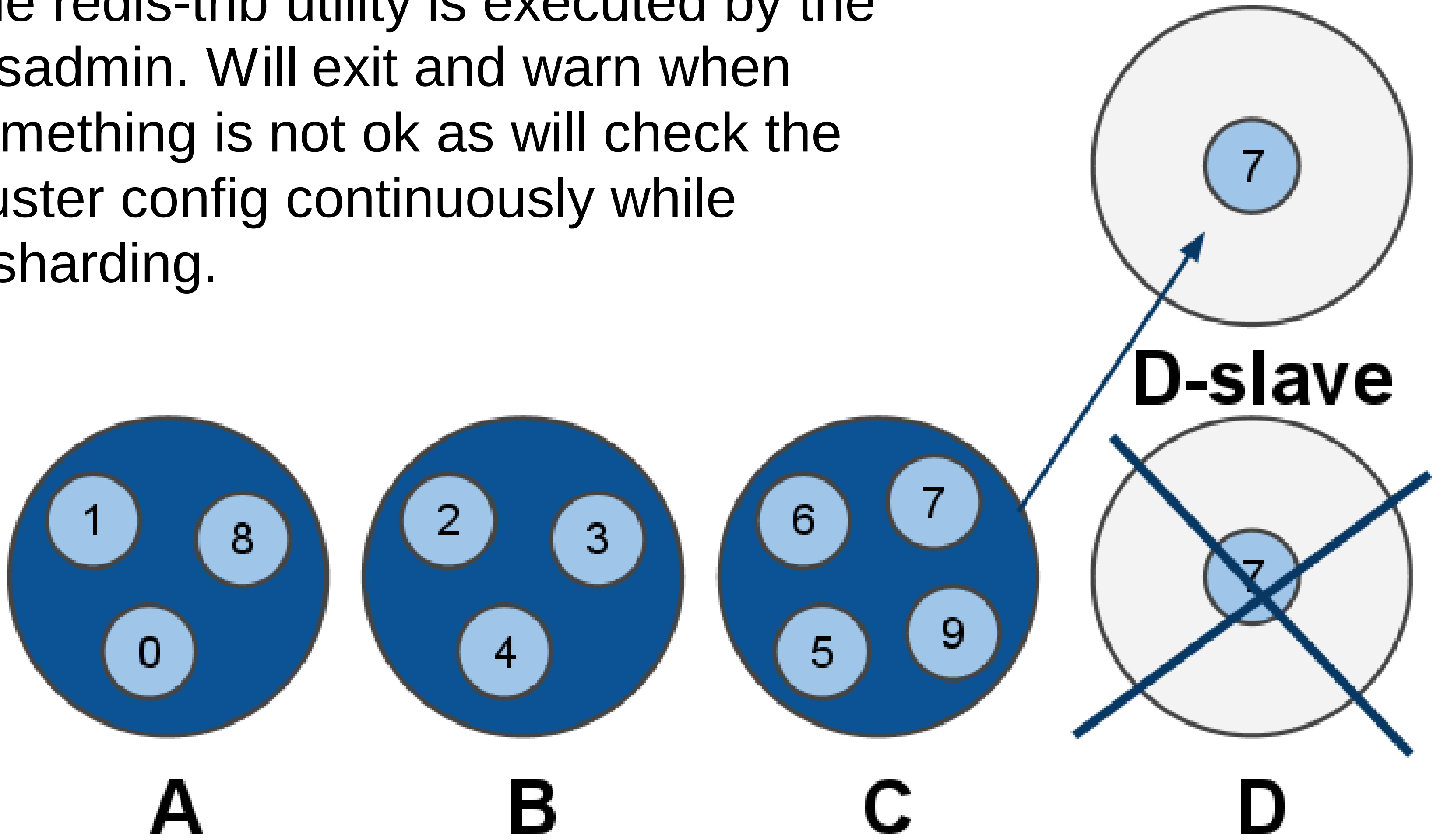


- All the new keys for slot 7 will be created / updated in D.
- All the old keys in C will be moved to D by redis-trib using the MIGRATE command.
- MIGRATE is an atomic command, it will transfer a key from C to D, and will remove the key in C when we get the OK from D. So no race is possible.
- p.s. MIGRATE is an exported command. Have fun...
- Open problem: ask C the next key in hash slot N, efficiently.

Re-sharding with failing nodes

- Nodes can fail while resharding. It's slave promotion as usually.

- The redis-trib utility is executed by the sysadmin. Will exit and warn when something is not ok as will check the cluster config continuously while resharding.

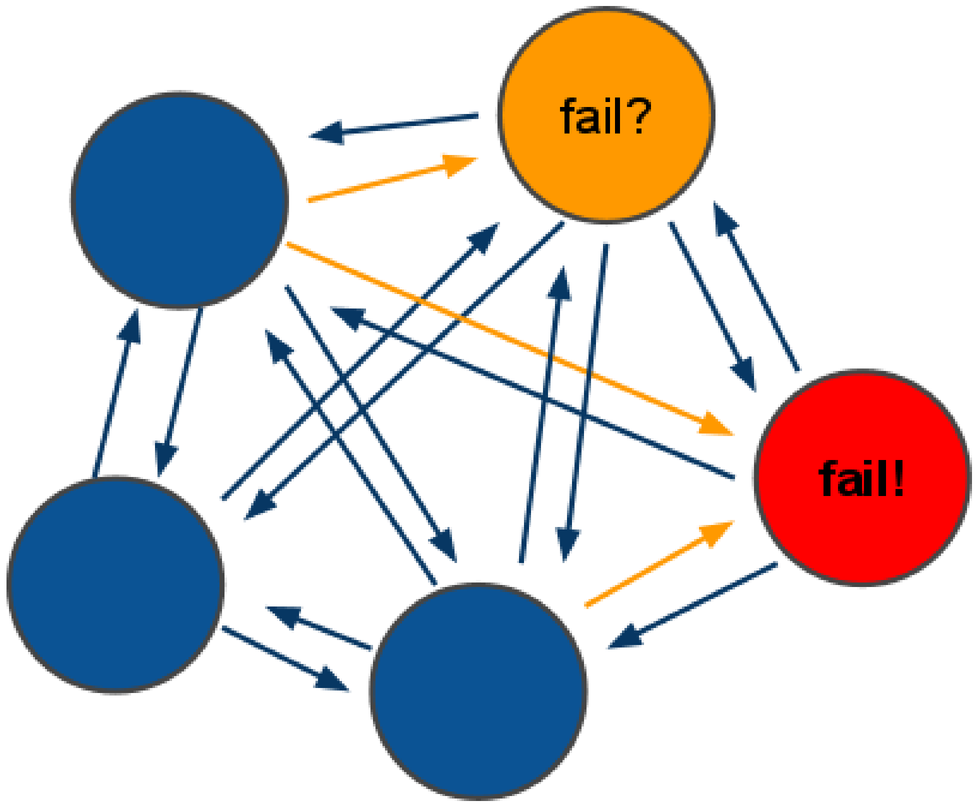


Fault tolerance

- All nodes continuously ping other nodes...

- A node marks another node as *possibly* failing when there is a timeout longer than N seconds.

- Every PING and PONG packet contain a **gossip section:** information about other nodes idle times, from the point of view of the sending node.



Fault tolerance - failing nodes

- A guesses B is failing, as the latest PING request timed out. A will not take any action without any other hint.
- C sends a PONG to A, with the gossip section containing information about B: C also thinks B is failing.
- At this point A marks B as failed, and notifies the information to all the other nodes in the cluster, that will mark the node as failing.
- If B will ever return back, the first time he'll ping any node of the cluster, it will be notified to **shut down** ASAP, as intermitting clients are not good for the clients.
- **Only way to rejoin a Redis cluster after massive crash is: redis-trib by hand**.

Redis-trib - the Redis Cluster Manager

- It is used to setup a new cluster, once you start N blank nodes.
- it is used to check if the cluster is consistent. And to fix it if the cluster can't continue, as there are hash slots without a single node.
- It is used to add new nodes to the cluster, either as slaves of an already existing master node, or as blank nodes where we can re-shard a few hash slots to lower other nodes load.

It's more complex than this...

- there are many details that can't fit a 20 minutes presentation...
- Ping/Pong packets contain enough information for the cluster to restart after graceful stop. But the sysadmin can use CLUSTER MEET command to make sure nodes will engage if IP changed and so forth.
- Every node has a unique ID, and a cluster config file. Everytime the config changes the cluster config file is saved.
- The cluster config file can't be edited by humans.
- The node ID never changes for a given node.
- **Questions?**

AMAZON ELASTICACHE

This part of today’s lecture is a very slight adaptation of the first part of the following paper, from which it borrows most language and illustrations:

- Wiger, Nate, *Performance at Scale With Amazon ElastiCache*, Amazon, May 2015.
- https://d0.awsstatic.com/whitepapers/performance-at-scale-with-amazon-elasticache.pdf

OVERVIEW

- In-memory caching improves application performance by storing frequently accessed data items in memory, so that they can be retrieved without access to the primary data store.
- Properly leveraging caching can result in an application that not only performs better, but also costs less at scale.
- Amazon ElastiCache is a managed service that reduces the administrative burden of deploying an in-memory cache in the cloud.
- Beyond caching, an in-memory data layer also enables advanced use cases, such as analytics and recommendation engines.

ROLE OF AMAZON ELASTICACHE

- The Amazon ElastiCache architecture is based on the concept of deploying one or more cache clusters for your application.
- Once your cache cluster is up and running, the service automates common administrative tasks such as resource provisioning, failure detection and recovery, and software patching.
- Amazon ElastiCache provides detailed monitoring metrics associated with your cache nodes, enabling you to diagnose and react to issues very quickly. For example, you can set up thresholds and receive alarms if one of your cache nodes is overloaded with requests.

ALTERNATIVES

- Use a CDN, such as Amazon CloudFront
  - Generally caches whole objects, but not individual components that are only useful to the application
- Read-only replicas, such as of databases
  - Still not in-memory
- On host caching
  - Difficult to coordinate among many hosts serving application

KEY-VALUE ENGINES

- These should look familiar:
- Memcached
  - Because Memcached is designed as a pure caching solution with no persistence, ElastiCache manages Memcached nodes as a pool that can grow and shrink, similar to an Amazon EC2 Auto Scaling group. Individual nodes are expendable, and ElastiCache provides additional capabilities here such as automatic node replacement and Auto Discovery.
- Redis
  - Because of the replication and persistence features of Redis, ElastiCache manages Redis more as a relational database. Redis ElastiCache clusters are managed as stateful entities that include failover, similar to how Amazon RDS manages database failover.

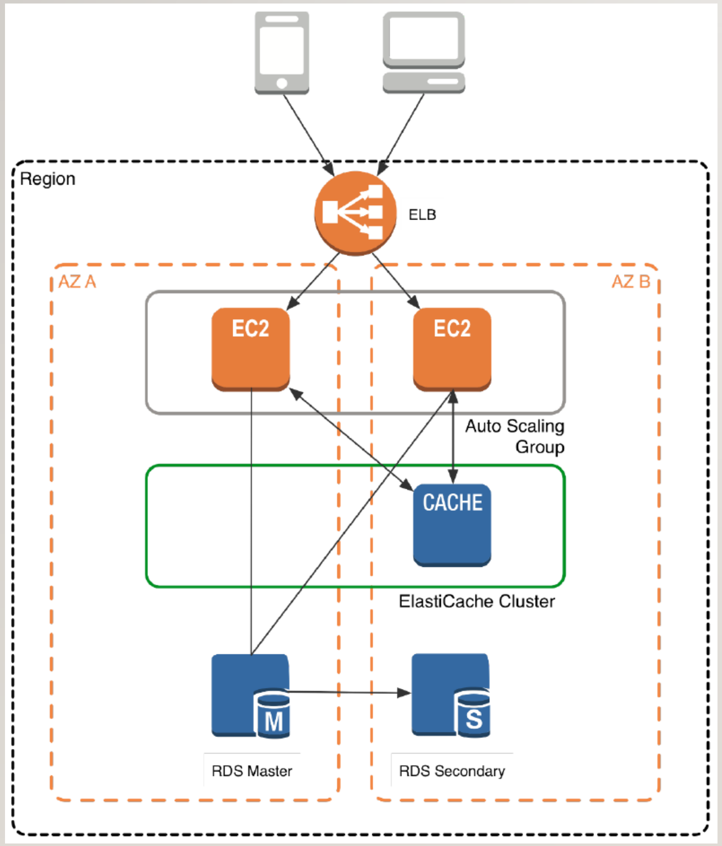
REDIS VS MEMCACHED

- Is object caching your primary goal, for example to offload your database? If so, use Memcached.
- Are you interested in as simple a caching model as possible? If so, use Memcached.
- Are you planning on running large cache nodes, and require multithreaded performance with utilization of multiple cores? If so, use Memcached.
- Do you want the ability to scale your cache horizontally as you grow? If so, use Memcached.
- Does your app need to atomically increment or decrement counters? If so, use either Redis orMemcached.
- Are you looking for more advanced data types, such as lists, hashes, and sets? Ifso, use Redis.
- Does sorting and ranking datasets in memory help you, such as with leaderboards? If so, use Redis.
- Are publish and subscribe (pub/sub) capabilities of use to your application? If so, use Redis. • Is persistence of your key store important? If so, useRedis.
- Do you want to run in multiple AWS Availability Zones (Multi-AZ) with failover? If so, use Redis.

REDIS VS MEMCACHED COMMON USE CASES

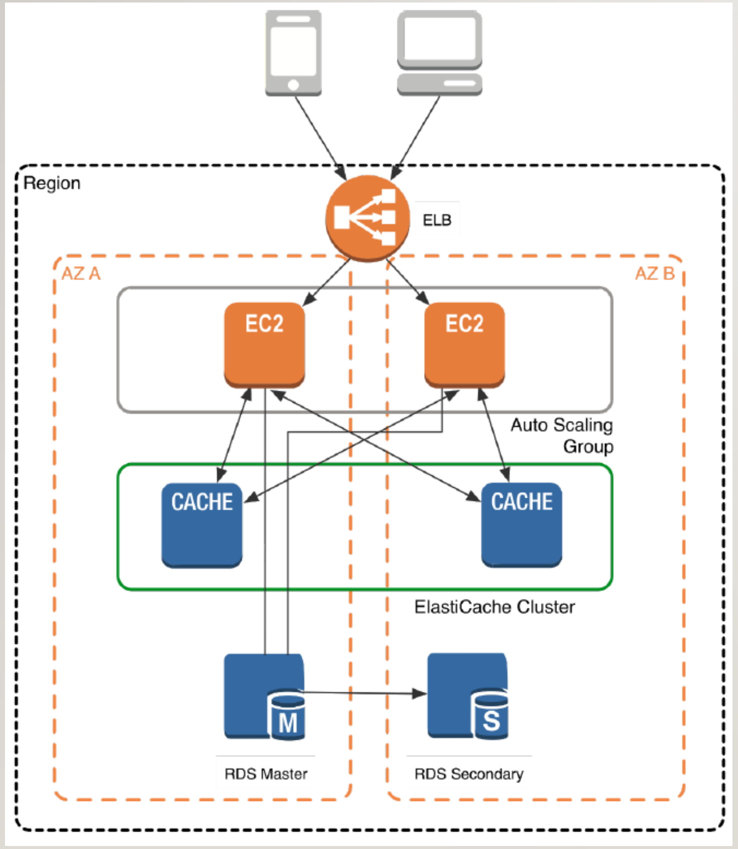
- Memcached as an in-memory cache pool
- Redis for advanced datasets such as game leaderboards and activity streams.

ARCHITECTURE (SIMPLIFIED) ELASTICACHE FOR MEMCACHED



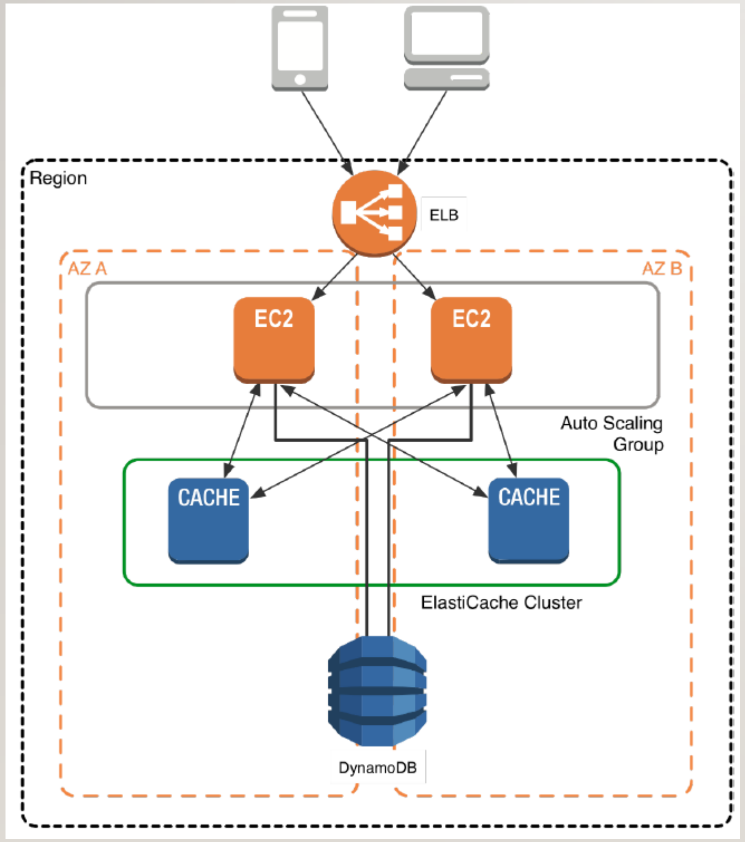
- ELB load balances among ECS application instances
- Application in EC2
- RDS Database
- ElastiCache between application and database
  - Coordinated by application in EC2
- Two availability zones (locations)

ARCHITECTURE (2 CACHE CLUSTERS) ELASTICACHE FOR MEMCACHED



- Notice this diagram has two cache clusters, one per AZ
- Cache clusters are coordinated by application in EC2
    - This means application is handing sharding

ARCHITECTURE (2 CACHE CLUSTERS + DYNAOMDB) ELASTICACHE FOR MEMCACHED



- Notice this solution uses DynamoDB
  - DynamoDB is not the same as Dynamo
- DynamoDB is Amazon's NoSQL Database
  - Implementation is black-boxed
  - Key-Value store
  - JSON-style documents

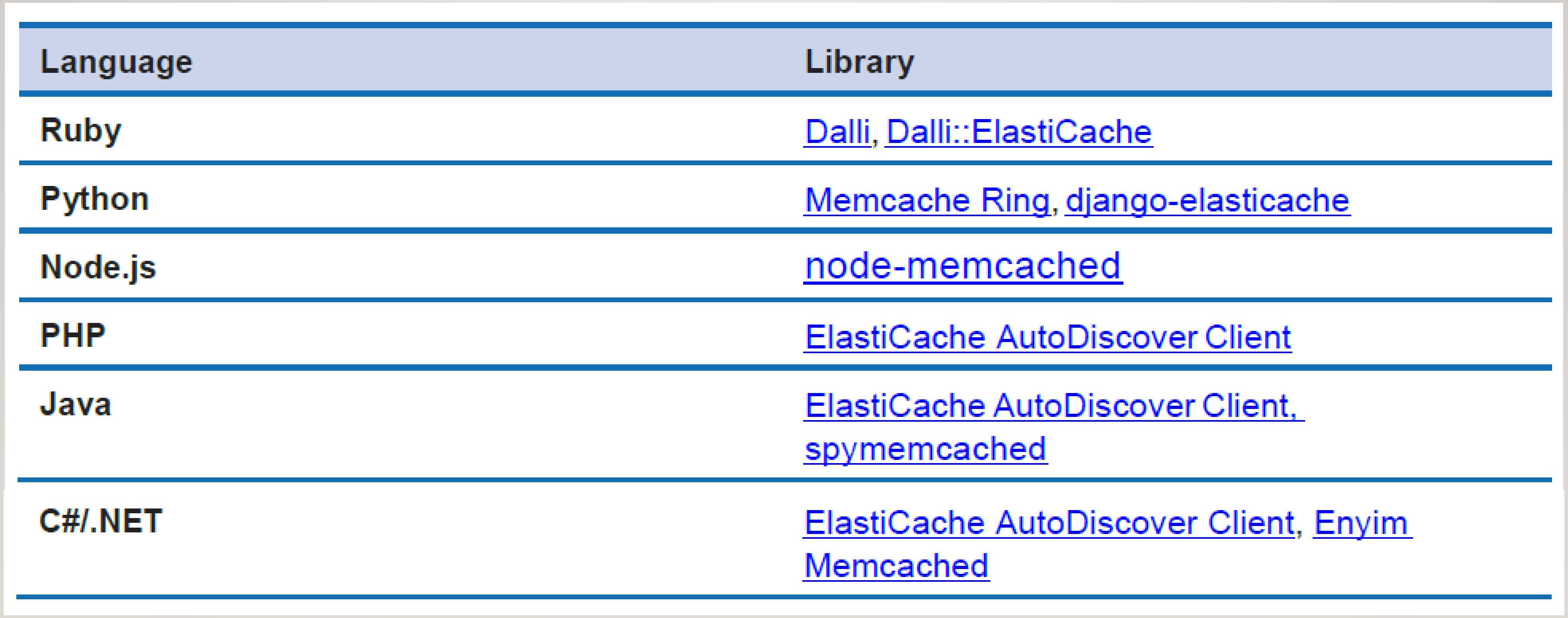
ACCESS CONTROL

- Neither Memcached nor Redis has any serious authentication or encryption capabilities.
- Like other Amazon web services, ElastiCache supports security groups.
  - You can use security groups to define rules that limit access to your instances based on IP address and port.
- ElastiCache supports both subnet security groups in Amazon Virtual Private Cloud (Amazon VPC) and classic Amazon EC2 security groups.
  - We strongly recommend you deploy ElastiCache and your application in Amazon VPC, unless you have a specific need otherwise (such as for an existing application).
  - Amazon VPC offers several advantages, including fine-grained access rules and control over private IP addressing.
- When launching your ElastiCache cluster in VPC, launch it in a private subnet with no public connectivity for best security.

HOW TO APPLY CACHING

- Is it safe to use a cached value? The same piece of data can have different consistency requirements in different contexts. For example, during online checkout, you need the authoritative price of an item, so caching might not be appropriate. On other pages, however, the price might be a few minutes out of date without a negative impact on users.
- Is caching effective for that data? Some applications generate access patterns that are not suitable for caching—for example, sweeping through the key space of a large dataset that is changing frequently. In this case, keeping the cache up to date could offset any advantage caching could offer.
- Is the data structured well for caching? Simply caching a database record can often be enough to offer significant performance advantages. However, other times, data is best cached in a format that combines multiple records together. Because caches are simple key-value stores, you might also need to cache a data record in multiple different formats, so you can access it by different attributes in the record.

MEMCHACHED LIBRARIES FEW EXAMPLES



Language	Library
Ruby	Dalli, Dalli::ElastiCache
Python	Memcache Ring, django-elasticache
Node.js	node-memcached
PHP	ElastiCache AutoDiscover Client
Java	ElastiCache AutoDiscover Client, spymemcached
C#/.NET	ElastiCache AutoDiscover Client, Enyim Memcached

CACHING STRATEGIES

- Simple Hashing
    - Like sophomore year (Ouch)
- Consistent Hashing
    - Complex to implement – but many good libraries
- Lazy Caching, e.g. LRU
    - Data pulled in by reads and expires
- Write-Through
    - Store data when written
    - Good if likely to be used again soon, not so good if it pushes out what will be
    - If not also doing Lazy caching, hot items won't be pulled back into cache after failure

THUNDERING HERDS

- The situation
  - Item not in cache (maybe brand new cache node)
  - TTL expires and item pushed out of cache
  - Many clients race to query underlying data in parallel
- One solution
  - Pre-warm cache

ELASTICACHE WITH REDIS

- Redis data structures cannot be horizontally sharded.
  - As a result, Redis ElastiCache clusters are always a single node, rather than the multiple nodes we saw with Memcached.
- Redis supports replication, both for high availability and to separate read workloads from write workloads.
  - A given ElastiCache for Redis primary node can have one or more replica nodes.
  - A Redis primary node can handle both reads and writes from the app.
  - Redis replica nodes can only handle reads, similar to Amazon RDS Read Replicas.
- Because Redis supports replication, you can also fail over from the primary node to a replica in the event of failure.
  - You can configure ElastiCache for Redis to automatically fail over by using the Multi-AZ feature.
- Redis supports persistence, including backup and recovery.
  - However, because Redis replication is asynchronous, you cannot completely guard against data loss in the event of a failure.

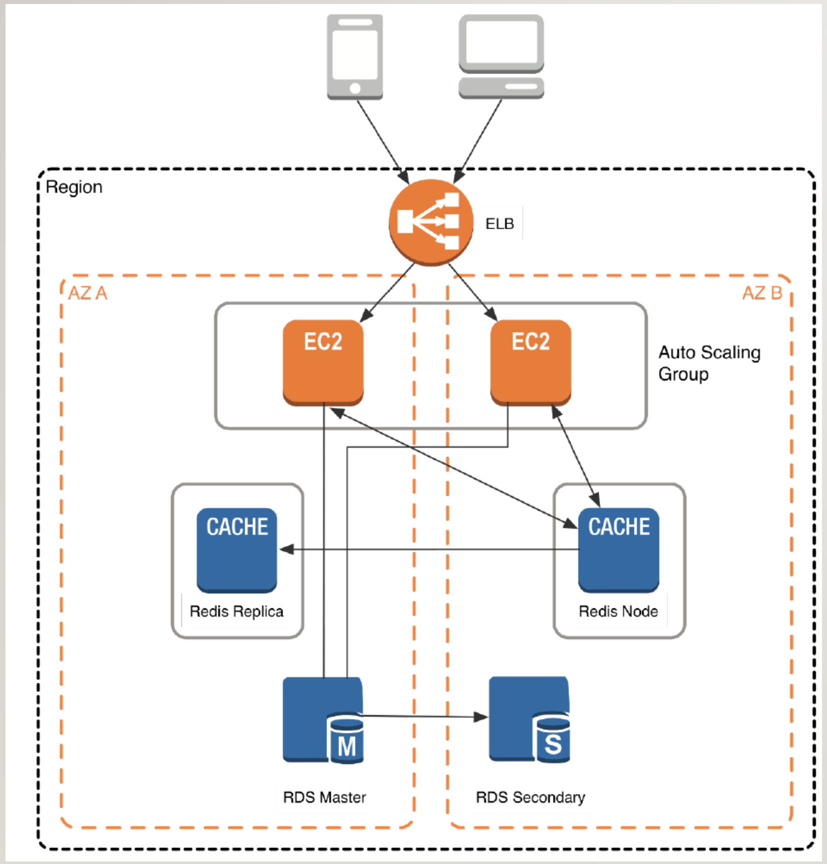
ELASTICACHE WITH REDIS: ARCHITECTURE

- As with Memcached, when you deploy an ElastiCache for Redis cluster, it is an additional tier in your app.
- Unlike Memcached, ElastiCache clusters for Redis only contain a single primary node.
- After you create the primary node, you can configure one or more replica nodes and attach them to the primary Redis node.
- An ElastiCache for Redis replication group consists of a primary and up to five read replicas.
    - Redis asynchronously replicates the data from the primary to the read replicas.

ELASTICACHE WITH REDIS: ARCHITECTURE NOTE

- Because Redis supports persistence, it is technically possible to use Redis as your only data store.
- In practice, customers find that a managed database such as Amazon DynamoDB or Amazon RDS is a better fit for most use cases of long-term data storage.
  - Redis provides rich types and simple data structures
  - Databases provide much stronger ability to structure data and organize and validate constraints

ELASTICACHE WITH REDIS: ARCHITECTURE (SIMPLE)

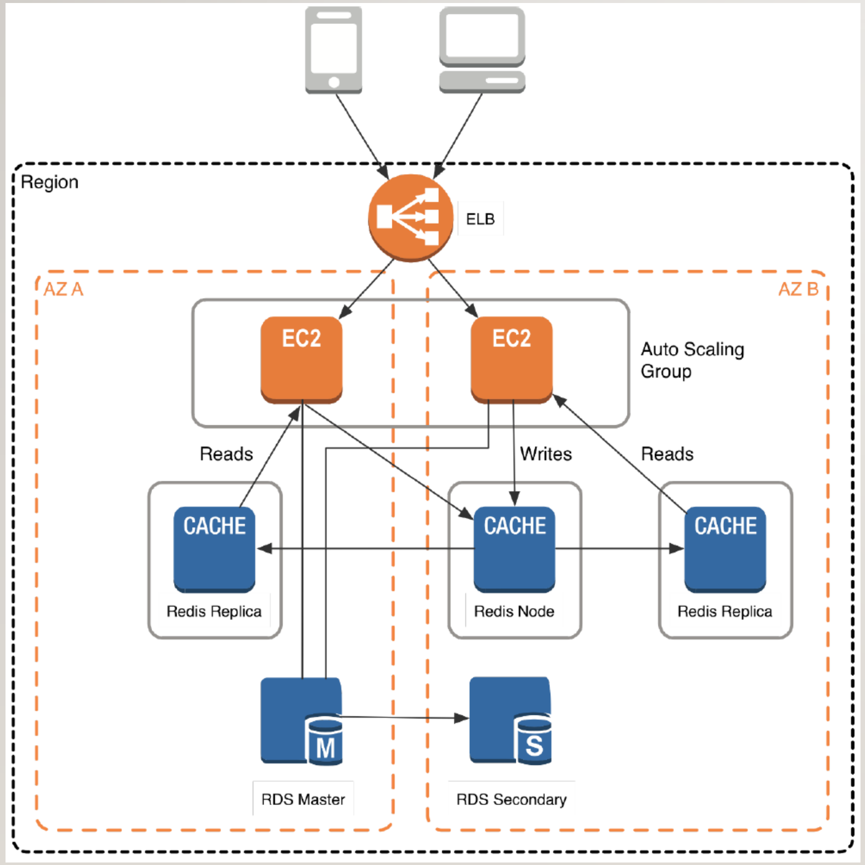


- Notice Simple replication across Azs
- ElastiCache for Redis has the concept of a primary endpoint, which is a DNS name that always points to the current Redis primary node.
- If a failover event occurs, the DNS entry will be updated to point to the new Redis primary node.

ELASTICACHE WITH REDIS: ARCHITECTURE (DISTRIBUTING READS AND WRITES)

- Using read replicas with Redis, you can separate your read and write workloads.
- This separation lets you scale reads by adding additional replicas as your application grows.
- In this pattern, you configure your application to send writes to the primary endpoint.
- Then you read from one of the replicas.
- With this approach, you can scale your read and write loads independently, so your primary node only has to deal with writes.

ELASTICACHE WITH REDIS: ARCHITECTURE (DISTRIBUTING READS AND WRITES)

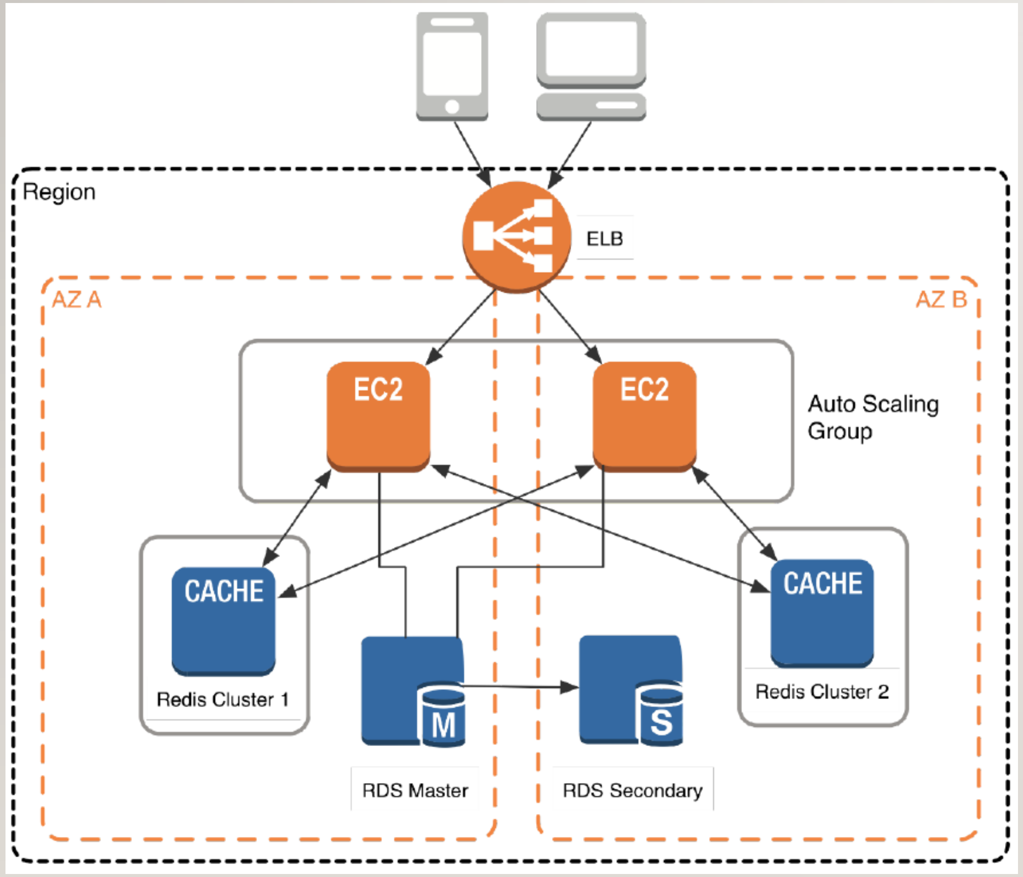


- Notice that reads are only from the replicas
- And the primary node absorbs the initial write and then replicates it.

ELASTICACHE WITH REDIS: ARCHITECTURE (DISTRIBUTING READS AND WRITES)

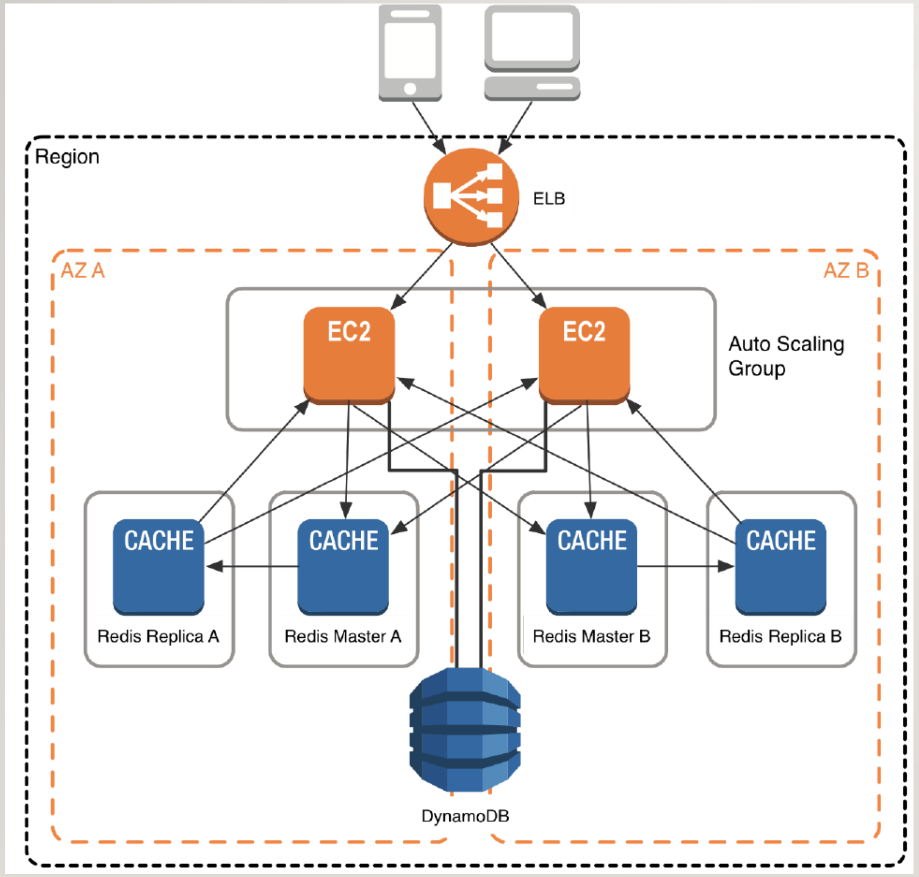
- **The main caveat to this approach is that reads can return data that is slightly out of date compared to the primary node, because Redis replication is asynchronous.**
- Is the value being used only for display purposes? If so, being slightly out of date is probably okay.
- • Is the value a cached value, for example a page fragment? If so, again beingslightly out of date is likely fine.
- • Is the value being used on a screen where the user might have just edited it? In this case, showing an old value might look like an application bug.
- • Is the value being used for application logic? If so, using an old value can be risky.
- • Are multiple processes using the value simultaneously, such as a lock or queue?If so, the value needs to be up-to-date and needs to be read from the primarynode.

ELASTICACHE WITH REDIS: ARCHITECTURE (SHARDING)



- Redis has two categories of data structures: simple keys and counters, and multidimensional sets, lists, and hashes.
  - The bad news is the second category cannot be sharded horizontally.
  - But the good news is that simple keys and counters can.
    - Range partitioning, Hash partitioning
- In the simplest case, you can treat a single Redis node just like a single Memcached node.
  - Just like you might spin up multiple Memcached nodes, you can spin up multiple Redis clusters, and each Redis cluster is responsible for part of the sharded dataset.

ELASTICACHE WITH REDIS: ARCHITECTURE (SHARDING + SPLIT READS AND WRITES)



- Notice two masters
    - Each with a replica