Practical Lab

Cloud Systems Engineering

(cloud-lab)

Chair of Decentralized Systems Engineering
https://dse.in.tum.de/

Distributed KVS

Sharding

Task #2:

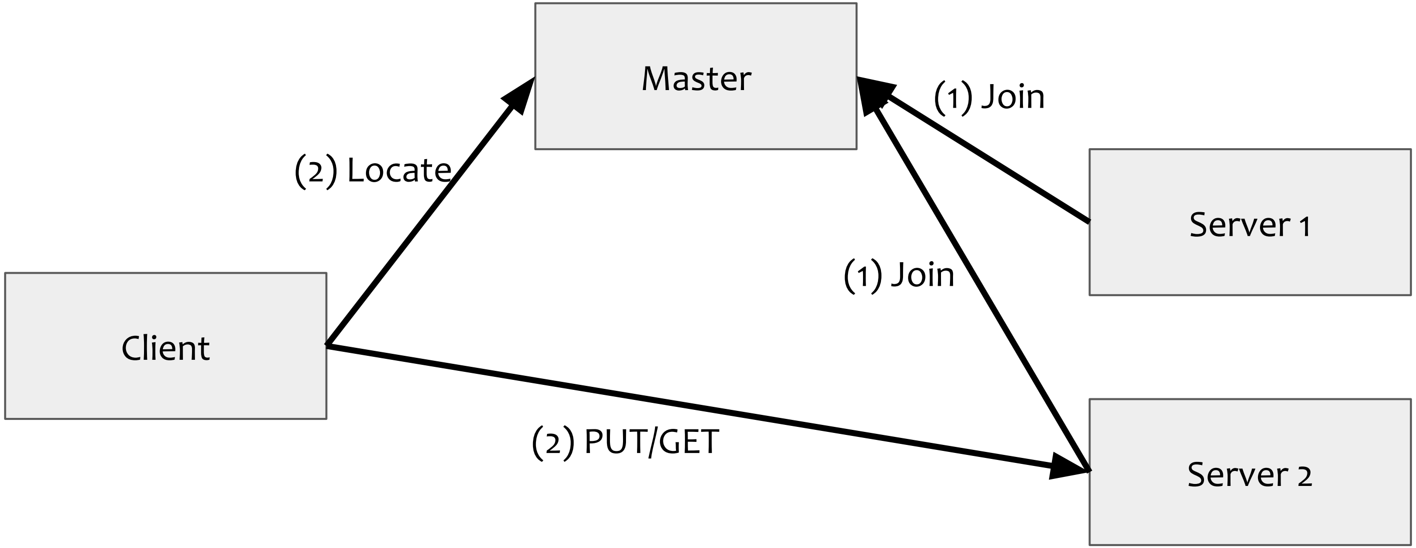
Task #2

Building on top of the single-node Key-Value store of task #1:

- Distribute it across a number of nodes by the way of sharding
- The sharding will be based on a simple hash function to map keys to their shards
  - Static sharding (shards not added once the client performs a few PUTs)
  - Dynamic sharding (shards added at any time)
- Implement a failure-detection system
  - Detect if a server has crashed
  - Not tested in this task, but required for task #3

https://classroom.github.com/a/0JR80WR-

Task #2: Basic design



Task #2: Master requirements

- A simple hash function that maps keys to their respective shards (KV servers in this case)
  - Shard = (key % num_servers) + 1
- Support two operations
  - Server join
  - Client locate
- Hint: Implement this as a single threaded server from the first task

Task #2: Client requirements

- Single threaded
- Protocol steps
  - First connect to master to know the location of the shard
  - Connect directly to the shard server to perform a PUT or GET

Task #2: Server requirements

- Single threaded
- Protocol steps
  - First connect to master server and JOIN the cluster
  - Respond to client requests

Task #2: Static vs dynamic sharding

- Static sharding
  - Keys need not be moved as servers/shards not added after clients have stored keys in the distributed kv store
- Dynamic sharding
  - Master has to redistribute the keys between the servers, including the new one

Task #2: Failure detection

- Implement within the master server
- Heartbeat thread
    - Pings all servers in the cluster to see if they are still alive

Background

Learning goals

In this task you will learn about:

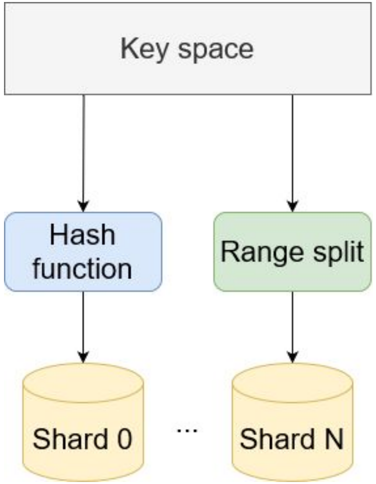
- Distributing a single-node KVS
- Key sharding & sharding strategies
- Consistent hashing
- Faults & Reconfiguration

Why distribute a single-node KVS?

- Fault-tolerance
  - A single node is a single point of failure
  - If this node fails, the system become unavailable
- Performance
  - A single node serves all PUTS/GETS for all keys
  - The system’s throughput = the node’s throughput
  - A single node is a bottleneck by definition
- Solutions:
  - Replication (next task !)
  - Sharding (this task)

Sharding (½)

- Is partitioning the key-space over a set of shards
  - Shards are mapped to groups of nodes, one-to-one
- Equivalently, a shard contains a subset of key-value pairs
  - And serves PUT/GET requests for this subset only
- Ideally, the partitioning should be uniform
  - They keys are uniformly distributed across the shards



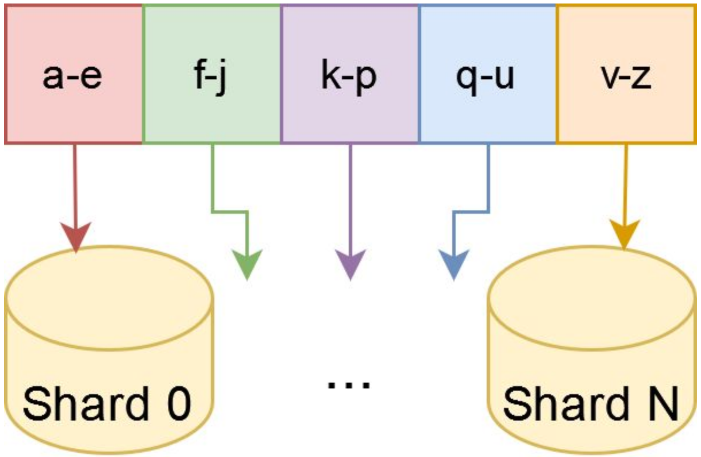
Sharding (2/2)

- Sharding directly enables *horizontal* scaling
  - More shards -> more throughput (linear scaling ideally)
  - Load-balancing across shards -> lower operation latency
  - Storage space scales linearly with the number of shards
- Also provides *partial* fault-tolerance
  - If some shards fail, the rest can still serve requests
  - A subset of the key-space remains available
- The sharding strategy should support flexible reconfiguration
  - Adding/removing shards affects a few shards
  - The key-subspace for the active shards should be automatically re-configured

Sharding strategies (½)

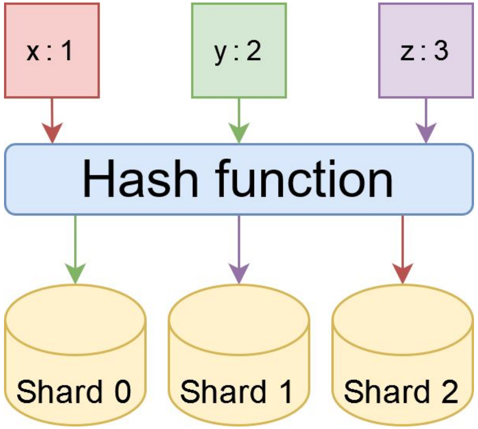
There are mainly two popular sharding strategies: Range and Hash based

- Range strategy:
  - The key-space is divided into equal ranges in a sequential manner
  - Each range is then mapped to a shard
  - Ideal for range-queries (e.g. fetch all values with keys in (*a-j*))
  - Suffers from key-hotspots (e.g. keys *k-p* are always accessed while keys *v-z* never)



Sharding strategies (2/2)

- Hash strategy
  - Maps keys to shards by applying a hash function on them
  - Simplest hash function: *shard = key % number_of_shards + 1*
  - Solves the hotspot problem, if the hash function provides uniform distribution
  - Inefficient for range-queries (sequential keys are scattered in different shards)
- In this lab we focus on simple PUT/GET operations and scalability, thus we choose a hash-based sharding strategy



Consistent hashing

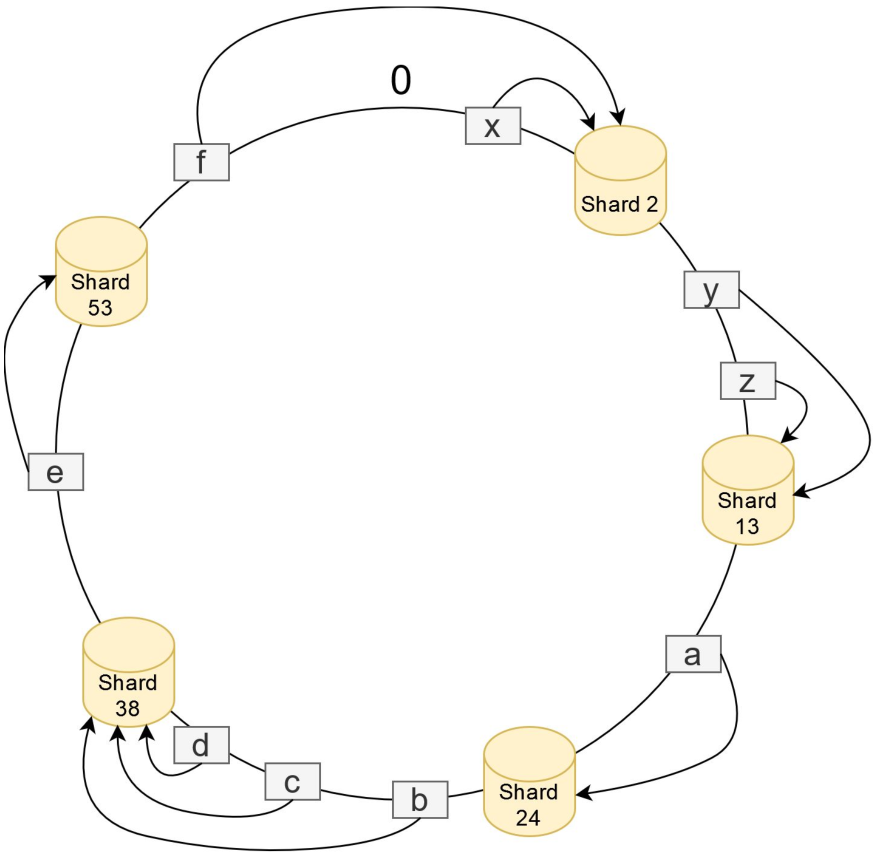
-Consistent hashing aims to operate independently of the number of shards or keys

-This allows shards and keys to scale without affecting the overall system

How it works:

- Imagine the shards placed on the edges of a circle
  - The shards are placed based on their unique IDs, ordered clockwise
- The keys are hashed to some values which now lie on the circle as well
  - Some keys may be placed on the same spot with shards, others not
- For each key that doesn't overlap with a shard, assign it to the next shard in the circle (clockwise)
  - Now all keys map to a shard

Consistent hashing

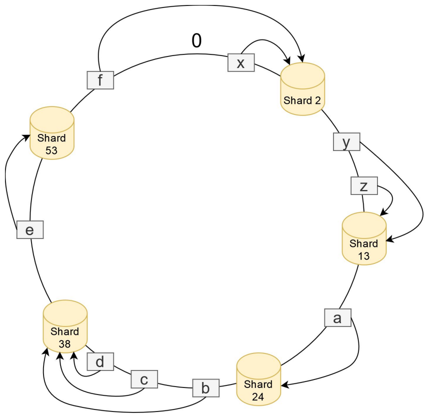


Finger tables

-If a shard only knew its successor node in the circle, a key lookup would cost O(n) network hops for *n* shards

-Solution: Finger tables

- Hold the IDs of m = identifier bits (identifier: 0 to $2^m - 1$)
- The *j*-th entry of node *i* contains the ID of successor $(i + 2^{j-1}) \bmod 2^m$
  - Where *successor* is a function that returns the closest node clockwise



S2 + 1	13	S2 + 8	13
S2 + 2	13	S2 + 16	24
S2 + 4	13	S2 + 32	38

Finger table for Shard 2

Key placement

The keys are assigned to shards based on the trivial hash function:

- Shard = successor(*key.hashCode()* **mod** 2^m)

Where *hashCode*() is the function:

- hash(s) = s[0]*31^(m - 1) + s[1]*31^(m - 2) + ... + s[m - 1]

  Where s[*i*] is the i^{th} character of the string and *m* its length

Key lookup

Lookup algorithm for key *k*:

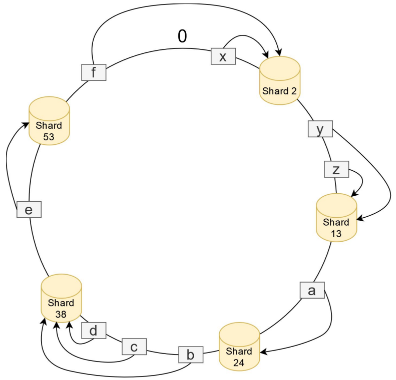
1. Query any random shard
2. If shard *ID* == successor(*k.hashCode()* mod 2^m), then this shard is responsible for *k*
3. Else, look in the finger table for the closest successor *s* of *k*
   3.1. Of course this may wrap around in the circle
4. Forward request to *s*

Steps 2-4 may be repeated O(log *n*) times, in each contacted shard in the path to the key, n is the number of shards

Key lookup example (1/2)

Say we want to lookup for a key with hashCode = 123

1. We pick Shard 13, randomly
2. 123 mod 64 = 59 (and successor(59) != 13)
3. The closest successor in the finger table is shard 53
4. We forward the lookup request to shard 53

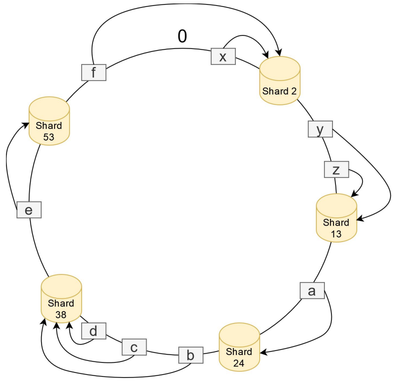


S13 + 1	24	S13 + 8	24
S13 + 2	24	S13 + 16	38
S13 + 4	24	S13 + 32	53

Key lookup example (2/2)

Now we repeat for shard 53

1.
2. 123 mod 64 = 59 (and successor(59) != 53)
3. The first entry of the finger table is the closest successor of 123
4. We forward the lookup request to shard 2, which will satisfy condition (2) and serve the value

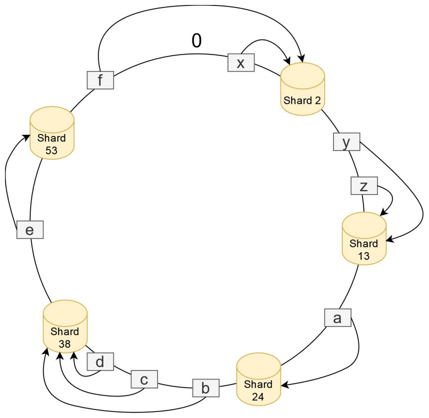


S53 + 1	2	S53 + 8	2
S53 + 2	2	S53 + 16	13
S53 + 4	2	S53 + 32	24

Reconfiguration

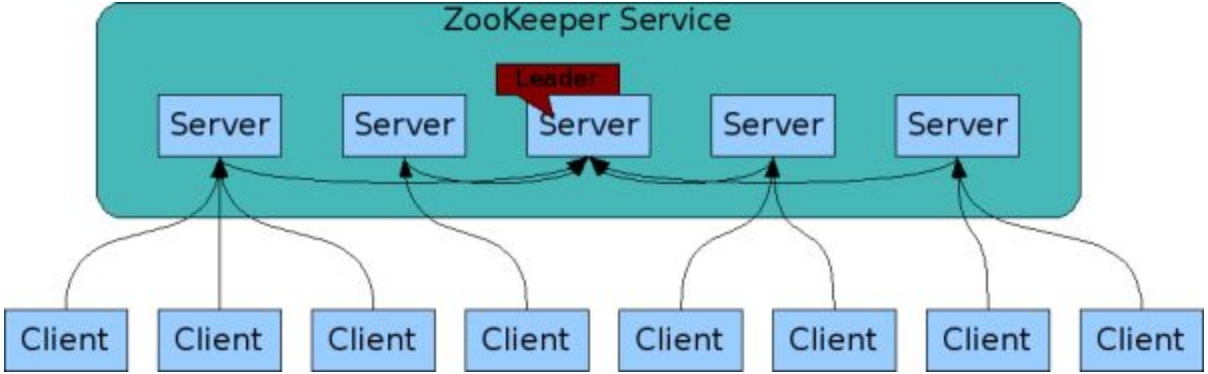
1. The newly added node is placed between nodes n_1 and n_2
2. The keys lying on the arc between n_1 and n_2 are splitted between the new node and n2
3. n_1 and n_2 are informed for this

Similarly, when a node leaves (either gracefully or by crash), a coordination service (e.g. ZooKeeper) will detect this and inform the node's predecessor and successor



Zookeeper

- Apache Zookeeper is an open source distributed coordination service that helps to manage a large set of hosts.
  - Zookeeper follows a Client-Server Architecture
  - All systems store a copy of the data
  - Leaders are elected at startup
- Used by Reddit, Meta, Twitter…



Zookeeper

- Naming service
  - Identifying the nodes in a cluster by name. It is similar to DNS, but for nodes.
- Configuration management
  - Latest and up-to-date configuration information of the system for a joining node.
- Cluster management
  - Joining / leaving of a node in a cluster and node status at real time.
- Leader election
  - Electing a node as leader for coordination purpose.
- Locking and synchronization service
  - Locking the data while modifying it.
- Highly reliable data registry
  - Availability of data even when one or a few nodes are down.

Chord

- CH: https://drive.google.com/file/d/1W2ikH7vXfoa7LBUnq9GzaX6xwL8vlGbW/view?usp=sharing
- DHTs: https://drive.google.com/file/d/1t6wlczVNBBeSiktnUFurZK7-eE1EBDrX/view?usp=sharing
- Chord: https://ieeexplore.ieee.org/abstract/document/1180543
- Go: https://github.com/arriqaaq/chord
- C++: https://github.com/sit/dht
- Paper: https://pdos.csail.mit.edu/papers/ton:chord/paper-ton.pdf
- Simplified explanation: https://resources.mpi-inf.mpg.de/d5/teaching/ws03_04/p2p-data/11-18-writeup1.pdf

Zookeeper

- Official introduction
  - https://zookeeper.apache.org/doc/r3.4.13/zookeeperOver.html
- CLI tutorial
  - https://www.tutorialspoint.com/zookeeper/index.htm
- Zookeeper C client github repository
  - https://github.com/apache/zookeeper/tree/master/zookeeper-client/zookeeper-client-c
- A book chapter on Zookeeper C client
  - https://www.oreilly.com/library/view/zookeeper/9781449361297/ch07.html

Revisiting task requirements

- Master
  - Implement hash function to determine shard
  - Implement failure detection system
  - Dynamic sharding and redistribution of keys and values
  - Acts similar to zookeeper
- Server
  - Register with master on creation
  - Respond to master heartbeat request
- Client
  - Connect to master to know the location of shard
  - Connect to shard for a GET/PUT request

Thank you for listening!

See you in the Q&A session