Practical Lab

Cloud Systems Engineering

(cloud-lab)

Lecture #1: Single-node KVS

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

Layered architecture

#4: Distributed TXs: w/ and w/o replication

#3: Replicated distributed KVS

#2: Distributed KVS

#1: Single-node KVS | #1: Single-node KVS | … | #1: Single-node KVS

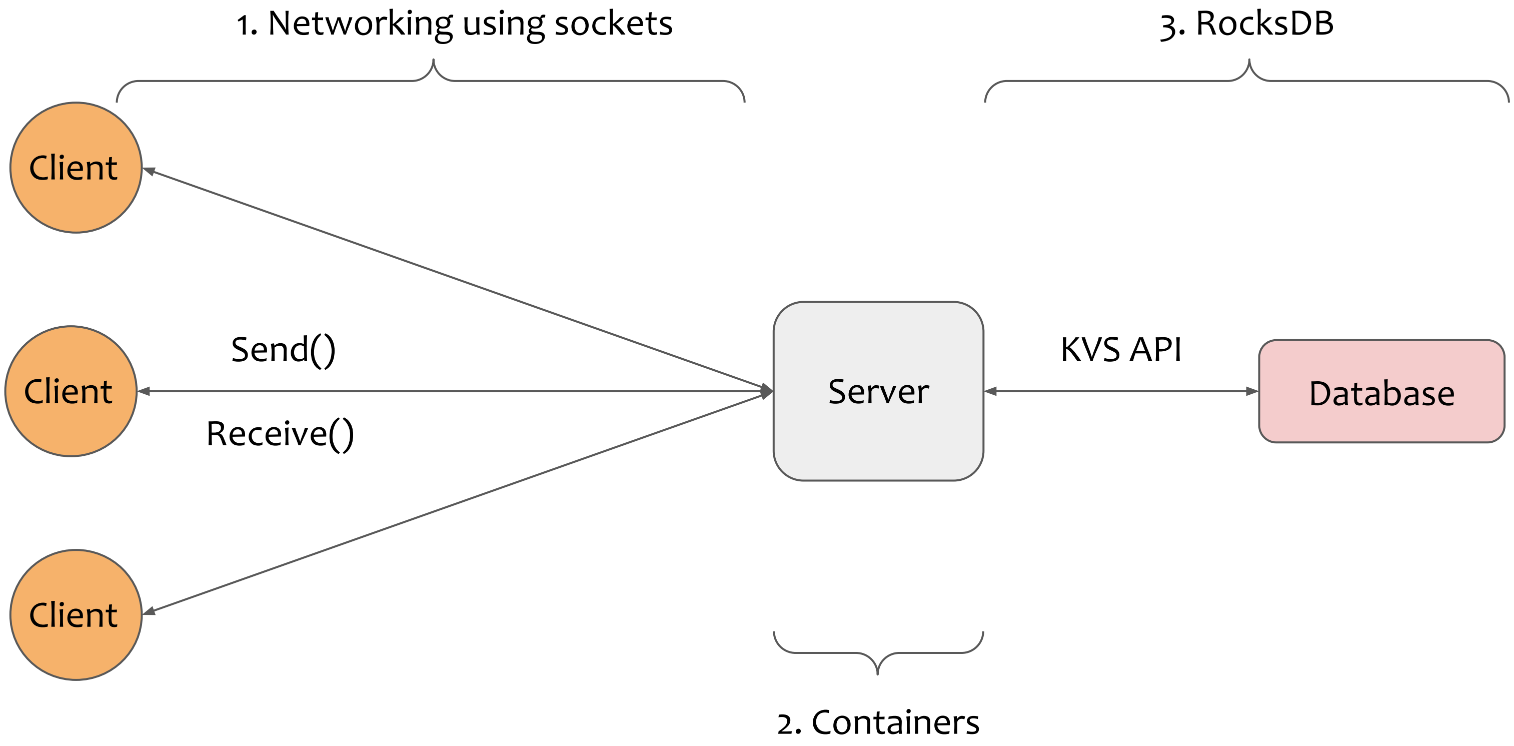
Cloud Spanner

raft

APACHE ZooKeeper™

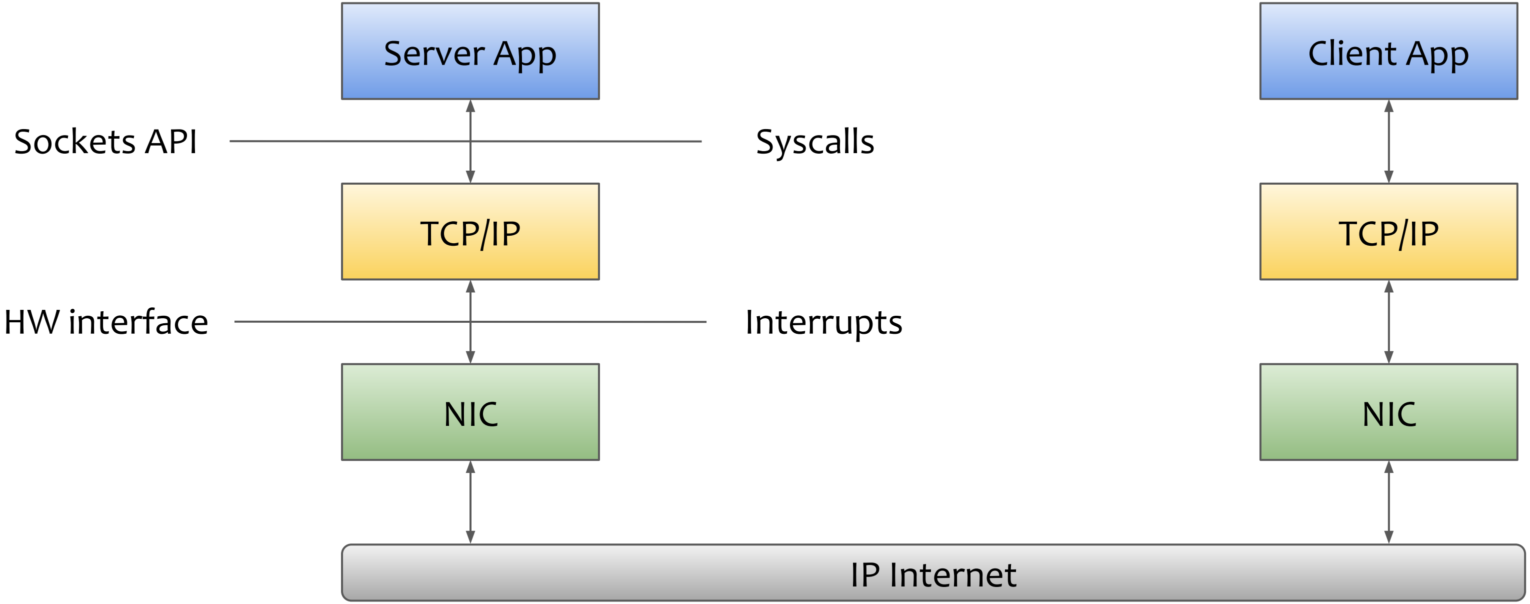
kubernetes

Single-node KV store



Socket networking

Network stack



Source: https://users.cs.duke.edu/~chase/cps196/slides/sockets.pdf

Sockets

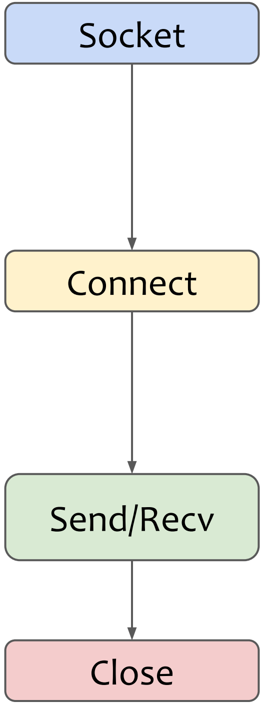
- A socket is essentially an **endpoint** for communication.
- Can be used both for IPC and communication through the **network.**
- Support both for **TCP** and **UDP**.
- Can operate in **blocking** or **non-blocking** mode.
- select(), poll(), epoll() for multiplexing and monitoring I/O to improve scalability

TCP Server

Create a socket	Socket	int **socket**(int domain, int type, int protocol);
Bind to an address (address+port)	Bind	int **bind**(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);
Listen for incoming connection(s)	Listen	int **listen**(int sockfd, int backlog);
Accept incoming connection(s)	Accept	int **accept**(int sockfd, struct sockaddr *addr, socklen_t *addrlen)
Send and receive messages	Send/Recv	ssize_t **send**(int sockfd, const void *buf, size_t len, int flags); ssize_t **recv**(int sockfd, void *buf, size_t len, int flags);
Close the socket descriptor	Close	int **close**(int sockfd);

Source: https://www.cs.cmu.edu/~dga/15-441/S08/lectures/03-socket.pdf

TCP Client



Create a socket — int **socket**(int domain, int type, int protocol);

Connect to a socket — int **connect**(int sockfd, const struct sockaddr *saddr, socklen_t addrlen);

Send and receive messages — ssize_t **send**(int sockfd, const void *buf, size_t len, int flags);
ssize_t **recv**(int sockfd, void *buf, size_t len, int flags);

Close the socket descriptor — int **close**(int sockfd);

Source: https://www.cs.cmu.edu/~dga/15-441/S08/lectures/03-socket.pdf

References

Talks & documentation:

- Sockets: https://users.cs.duke.edu/~chase/cps196/slides/sockets.pdf
- Protobufs: https://developers.google.com/protocol-buffers

Useful tutorials:

- Protobufs tutorial: https://developers.google.com/protocol-buffers/docs/cpptutorial
- Socket tutorial: https://www.linuxhowtos.org/C_C++/socket.htm
- Libevent tutorial: http://www.wangafu.net/~nickm/libevent-book/

Containers

Motivation

Create, package, and deploy software across different environments

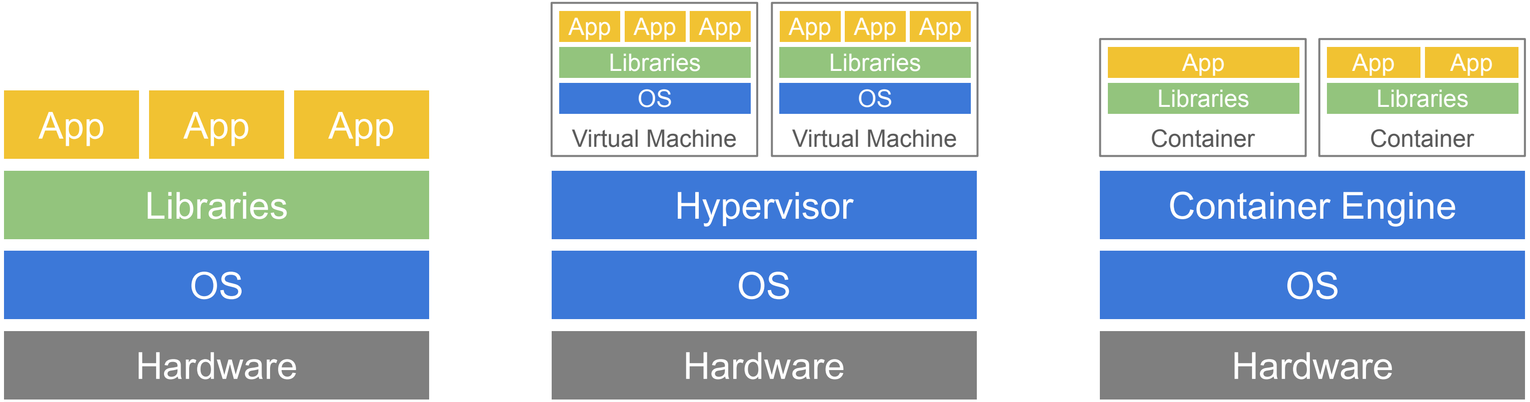
Share resources in a flexible and cost-effective way

→ Split applications into microservices using containerization

Advantages:

- Greater hardware resource utilization
- Infrastructure management for container systems can be standardized
- Scalable - Lightweight image sizes, quick start up times, easy testing, etc.
- Compatibility - old apps or OS-specific apps run on newer/different hosts

Containers vs Virtual Machines



Traditional system

Virtual machines

Containers

Containers

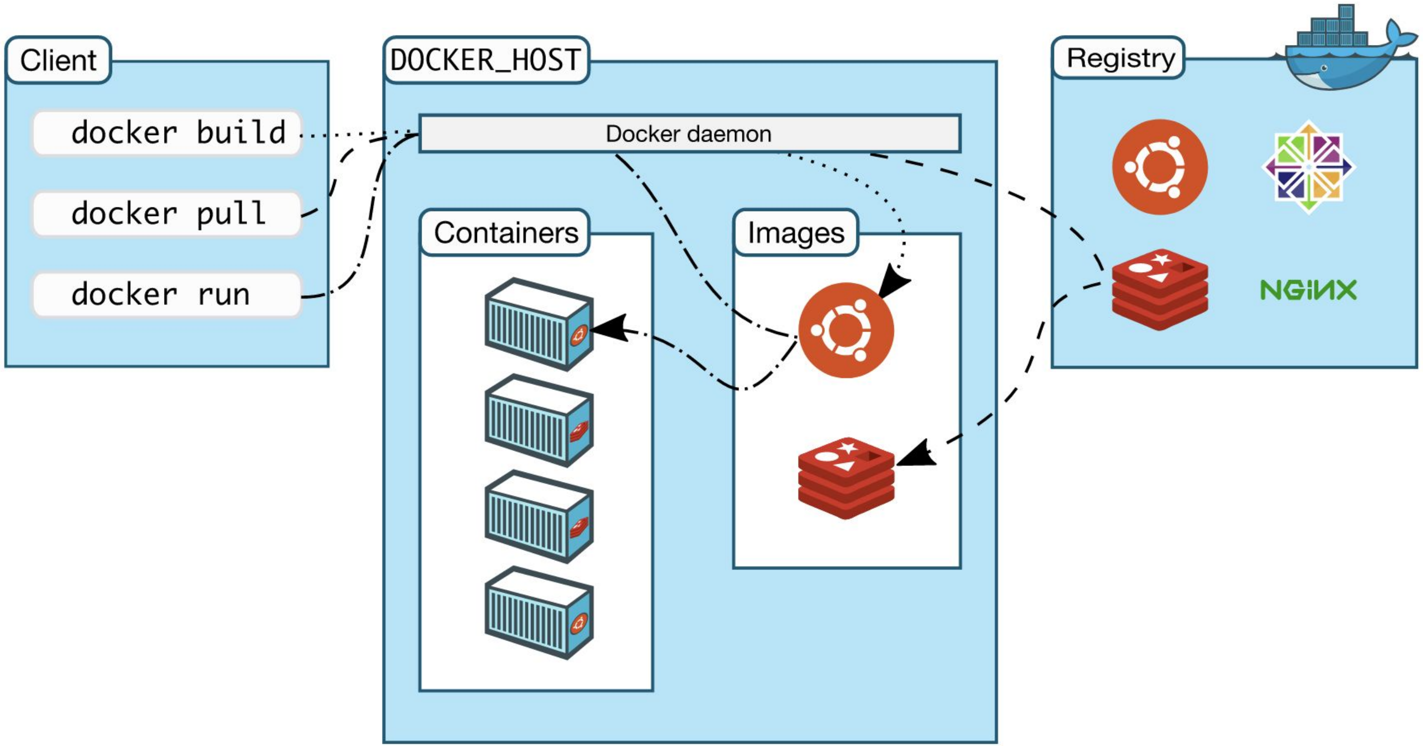
Unlike virtual machines ...

- containers often contain only one application
- containers use the operating system (OS) of the host
- containers offer process-focused virtualization
- containers provide a good balance of flexibility and speed
- containers limit access to resources through host OS mechanisms

Docker

- Application container engine
    - create container images, push or pull images
    - manage containers in many different environments
- Uses the resource isolation mechanisms of the Linux kernel
    - Namespaces - limit what a process can see in a system
    - Cgroups - group processes and their resources
    - Layered filesystems - filesystem separation
- Consists of three components: Software, objects and registry

Docker



Client
docker build
docker pull
docker run
DOCKER_HOST
Docker daemon
Containers
Images
Registry
NGINX

Getting started

1. Install Docker on your system, e.g.: `# apt install docker-ce`
2. Run some container: `$ docker run -it ubuntu /bin/bash`
3. Docker run provides [further useful options](), such as mounting a volume (*-v*), setting a memory limit (*-m*), connecting to a network (*--network*) etc.

Hints:

- List all containers: `$ docker ps -a`
- Get a shell for a running container: `$ docker exec -it <container name> /bin/sh`

Getting started

1. Create a Dockerfile (see Docker’s docs)
2. Build an image based on the Dockerfile: $ docker build -t <image name> .
3. Run the container: $ docker run -d <image name>
4. Create and run multiple containers using Docker compose

Container debugging hints:

- Force clean rebuild of image: $ docker build --no-cache -t <image name> .
- To prevent a container from exiting early when debugging, add CMD tail -f /dev/null to the end of the Dockerfile

References

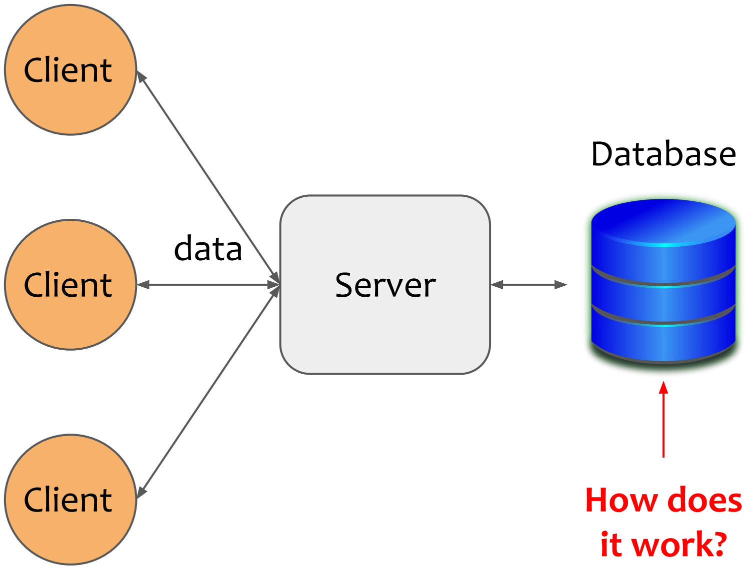
Talks & documentation:

- Intro to containers: https://www.chpc.utah.edu/presentations/images-and-pdfs/containers22s.pdf
- SUSE intro to containers: https://www.suse.com/c/rancher_blog/an-introduction-to-containers/#container-terminology
- Docker: https://www.docker.com/

KV store - RocksDB

Client-server architecture recap

- Server
  - Usually a long running process (*daemon process*)
  - Manages resources
  - Receives and processes requests
- Client(s)
  - Sends one or more requests to the server
  - Wait for the server's reply
- Transport layer
  - Network medium
  - Transfers the data

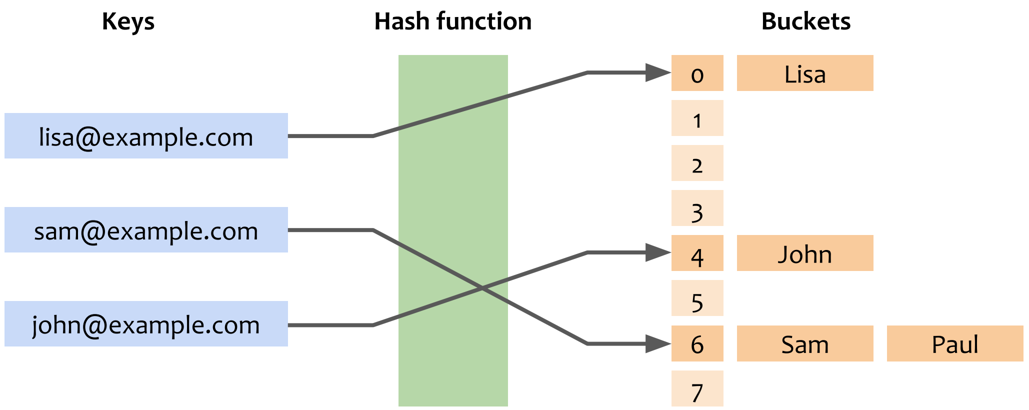


Key-Value store (KVs)

- Data structure
  - stores, retrieves and manages data
  - e.g., dictionaries, hash-tables

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2022
K5	3,ZZZ,5623

dictionary



hash map

Key-Value store importance

Key-value stores play an important role at tech giants:

memcached	Redis	Voldemort	Dynamo
Facebook	GitHub	LinkedIn	Amazon
Twitter	Digg		
Zynga	Blizzard Interactive		

Research interest

Key-value stores play an important role in the scientific community:

- FASTER: A Concurrent Key-Value Store with In-Place Updates [SIGMOD '18]
- KVell: Design and Implementation of a Fast Persistent Key-value Store [SOSP '19]
- Nova-LSM: A Distributed, Component-based LSM-tree Key-value Store [SIGMOD '21]
- And many, many more…

Challenges - design goals

- Performance
  - lock contention, significant write-traffic, complex memory management
  - low-latency operations and high throughput (I/O, batching)
  - parallelism (e.g., keys hashing)

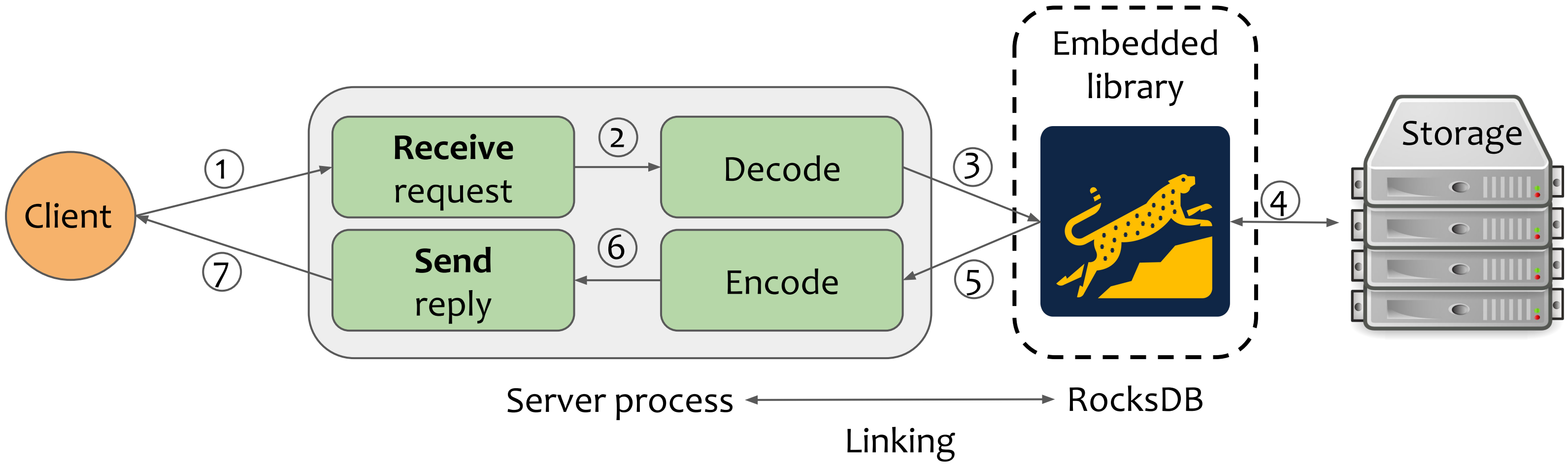
- Data properties
  - Persistency, e.g., persistent KVs or in-memory KVs
  - Consistency, e.g., linearizability or sequential consistency
  - Durability or crash consistency (for persistent KVs)

Key-Value store operations

Key-value stores typically implement a set of operations:

- GET
  - Retrieve a value by key
- PUT
  - Insert or update a key-value pair
- DELETE
  - Deletes a key-value pair (if it exists)
- Range Queries
  - Queries applying to a range of KV pairs

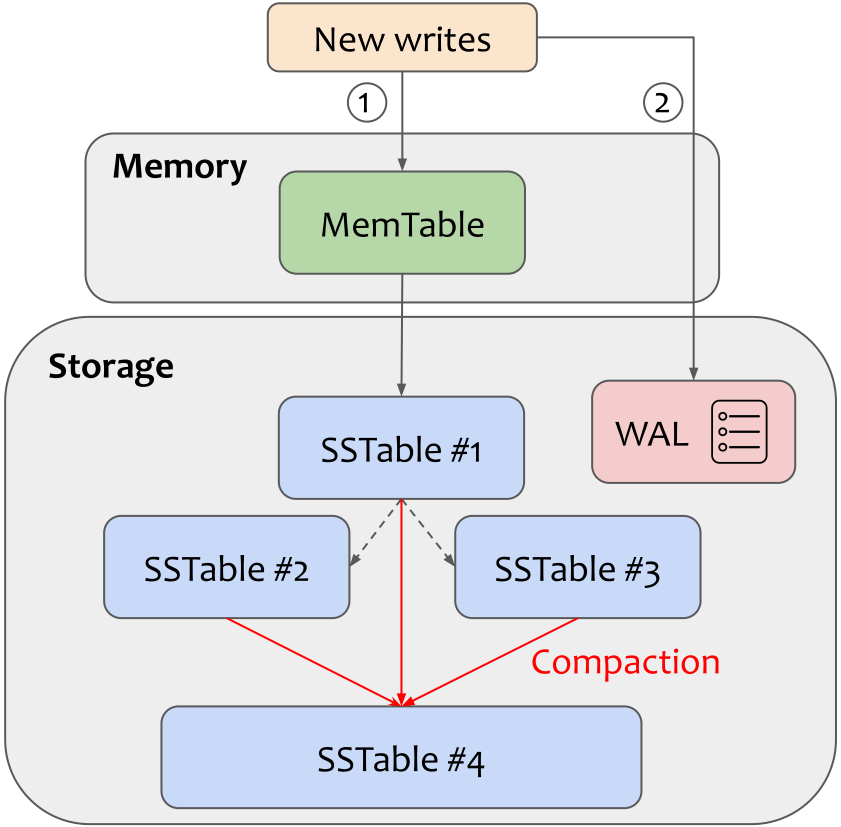
Server-Client architecture w/ RocksDB



RocksDB architecture

RocksDB components:

- Based on LSM (Log-structured merge-tree)
- In-memory skiplist (MemTable)
- SSTable files organized in levels with KV pairs
- Write-Ahead-Log (WAL)
- Compaction (background thread)



RocksDB data structures

RocksDB properties

- RocksDB data properties
  - Linearizable reads – a “Read” retrieves the latest write
  - Durability – SSTables are persistent
  - Crash-consistency – Write-Ahead-Log (WAL)
  - No replication out of the box – helper functions for replication system implementation

- RocksDB Operations
  - supports PUT, GET, DELETE queries
  - supports range scans
  - supports multi-operational transactions

References

Talks & documentation:

- Rocksdb: https://github.com/facebook/rocksdb
- Rocksdb 101: https://www.youtube.com/watch?v=V_C-T5S-w8g
- Rocksdb talk: https://www.youtube.com/watch?v=tgzkgZVXKB4
- Rocksdb overview: https://github.com/facebook/rocksdb/wiki/RocksDB-Overview

Useful tutorials:

- Rocksdb tutorial: https://rocksdb.org/docs/getting-started.html