Summary of Cloud Computing at University of Bristol 2018 / 2019*

Florian Bauer

January 19, 2019

*This is just a simple summary. I am not responsible for the provided content or anything which belongs to this. If there are any questions please contact me at bauerflorian13@gmail.com .

Contents

Lecture 01: Introduction

Comparison of the internet and electricity network

- starts with everyone has his own (electricity/ computationally power)
- connection between every single users grows
- ends in an all connected world with only a few big services provided by a small number of providers (computationally power goes from the device of the endusers to the cloud, electricity comes from big providers)

Normal Failure

- cloud data centre with 99.999% survival rate
- 500000 server, probability of 100% of the servers are still running after 3 years is 1%.
- **solution**: modular data centres, *servers in container boxes*

Essential Characteristics of Cloud Computing

This definition belongs to NIST's characteristics of Cloud Computing

- **On-demand self service**
- **Broad network access**
- **Ressource pooling**
- **Rapid elasticity**
- **Measured service**

A common stratification: *aaS

Everything as a Service.

- **SaaS**: *Software as a Service*, for instance: everyone
- **PaaS**: *Platform as a Service*, for instance: *Google App Engine*, *Amazon Appstream*
- **IaaS**: *Infrastructure as a Service*, for instance: *Amazon EC2, S3*, *Google Compute Engine*

A small number of companies providing IaaS/PaaS s services. Convergence to an oligopoly of less than five providers seems certain.

Lecture 02: Coursework

Just a few informations about the coursework and programming project. May be hopefully not important for the exam...

L03: Economics of Cloud

The basic Economics

- **Cap**ital **Ex**penditure: *Capex*
- **Op**erating **Ex**panditure: *Opex*
- Capex vs Opex: *Why buy a cow if all you need is the milk?*

A typical warehouse scale computer

- *pizzabox* in a *refrigerator* is a server rack
- multiple server racks together are a cluster
- see Figure 1

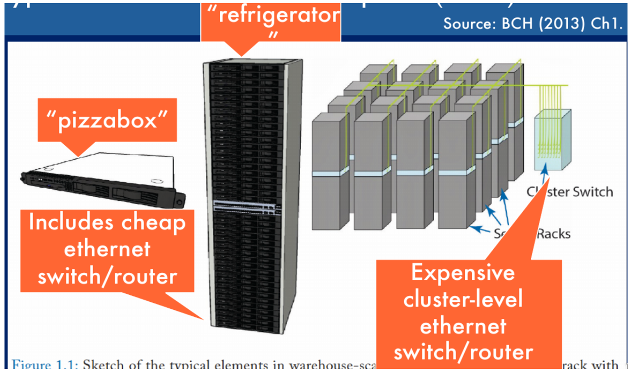


Figure 1: WSC - Warehouse-scale Computer

Energy & Power Efficiency

- cooling cost are around 42%
- optimizing the cooling efficiency will lower the overall costs massivley

Resume

- there is a lot going on *under the hood of a WSC* (WSC = **Warehouse-scale Computer**)
- *prod>>dev*: The innovations are made by and in companies not universitys

L05: *aaS

Definiton see in the Introduction section (Everything as a Service).

Why Xaas or *aaS

- avoiding of **Undiffertiated Heavy Lifting**
- the cloud is an ideal environment providing *scale*, *low cost*, *automation via Infrastructure-as-Code*

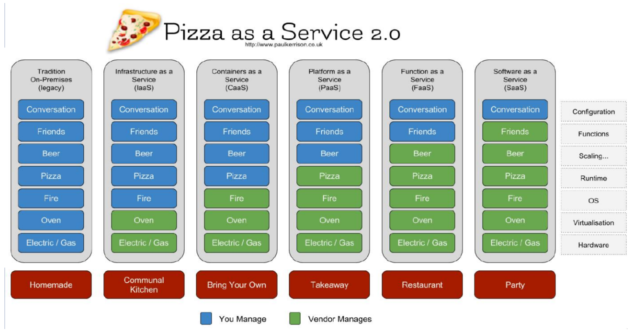


Figure 2: Pizza as a Service Example for *aaS

Structure of AWS Cloud

- **Availability Zones**: cluster of independent data centres, enables **fault isolation** and **high availability**
- **Regions**: entirely independent clouds, consists of a least two *AZ*s, interconnection on global backbone, different regions have different costings

Which Region should I choose?

- **Data souvereignty and compilance**: where to store user data?
- **Proximity of users to data**: where are the most of my users? -¿ lowest latency
- **Services and feature availability**: services and features may vary
- **Cost effectiveness**: each region has different costs (Europe and US are the cheapest)

High Availability & Fault Tolerance

High Availability:

- minimise service downtime by using redundant components
- require components in at least two AZs
- IaaS may have HA, PaaS usually will have HA

Fault Tolerance

- ensure no service disruption by using active-active architecture
- requires service components in at least three AZs
- Iaas is unlikely to offer FT, PaaS some offers FT

AWS Storage options

- Elastic Block Storage: SSDs, Magnetic, NAS, Use: OS, Apps
- S3: durable object storage, very cheap and big
- Instance Storage: on-host storage, very fast, caching
- Elastic File Store: shared storage across AZs

IaaS vs PaaS

- IaaS mainly used by SysAdmins, PaaS mainly used by Developers
- IaaS provides e.g. *VMs*,*Storage Services*,*Networking*, PaaS provides e.g. hosted databases, *App deployment and managment env.*,*test suites*
- IaaS lower cloud costs, PaaS lower human costs

L07: Virtualisation, Containers and Container Orchestration

Virtualisation Basics

- server hardware should be hidden from the user, → user sees only guest OS in a VM and not the host OS
- Amazon offers different VMs (*AMIs*) with Linux or Windows
- VMs are created and run by the *Virtual Machine Monitor (VMM)* aka the **hypervisor**
- VMs can stopped, copied, paused and resumed, which enables **server consolidation**: compress VMs to freeup servers

Types of Virtualisation

Have a look at Figure 3

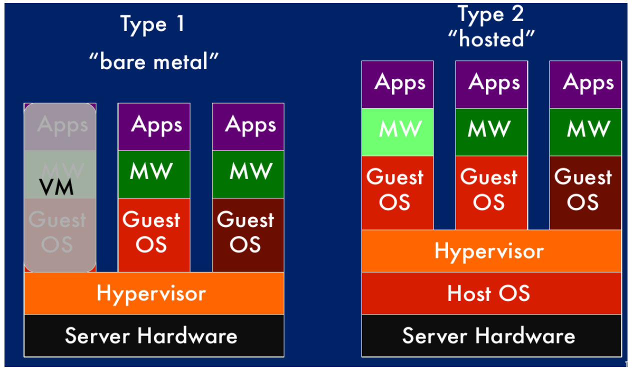


Figure 3: The two different virtualisation types

Xen is an example for Type 1 VMs.

- **Full virtualisation**: complete simulation of underlying guest machine hardware

- **Paravirtualisation**: guest OS can make Syscalls via the hypervisor's API, hypervisor does not simulate hardware

Containerisation: Docker

- package and run application in lightweight, isolated environment
- Docker runs user processes in a super-isolated execution mode
- *operating system level virtualisation* with shared kernel
- Advantage: No need to boot a whole VM
- Disadvantage: Potentially more insecure than complete virtualisation

Docker Objects

- **Images**: read only template with instructions how to create a Docker Container
- **Container**: runnable instance of an image, but ephemeral → all changes not mounted to persistend storage will be lost

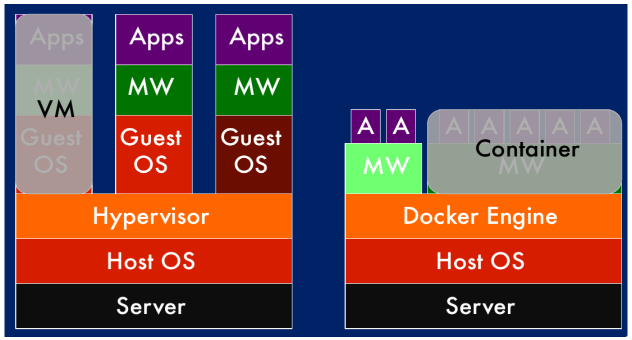


Figure 4: VMs vs Docker architecture schema

Container Orchestration: Kubernetes

Motivation

- To run containers at scale needs managment tools
- **(Horizontal) Auto-scaling on demand**
- **Fault Tolerance**
- **Manage Accessibility from the web**
- **update/rollback without downtime**

Featues of Kubernetes

- **Automated scaling**
- **Self healing**
- **Horizontal scaling**
- **Service discovery and Load Balancing**
- **Automated Rollbacks/Rollouts**

Kubernetes Components

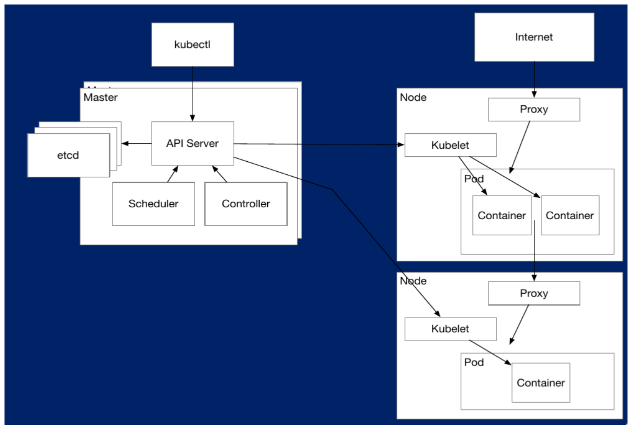


Figure 5: Components of the Kubernetes architecture

- **Master**: manages the cluster state, subcomponents: **API Server**, **Controller**, **Scheduler**, writes to *etcd*
- **Nodes**: run work in pods, **Pods** are the scheduling unit, **Kubelet** is the agent to communicates with master, **Kube-proxy** is the network agent
- **Kubeclt**: local cli to controll cluster
- **Etcd**: distributed key-value store
- **Deployments**: **Replica Sets**, balances the number of running and scheduled pods; deployments provide update to Pods or ReplicaSets
- **Services**: groupings of pods which can be referred by a name, Unique IP and DNS name; Pods in Services are load balanced

L09: Serverless

Definiton: *The essence of the serverless trend is the **absence** of the server concept during software development.*

Abstractions of App Deployment

- *More Abstraction*: more control and trust to given platform
- *Less Abstraction*:more undifferentiated heavy lifting

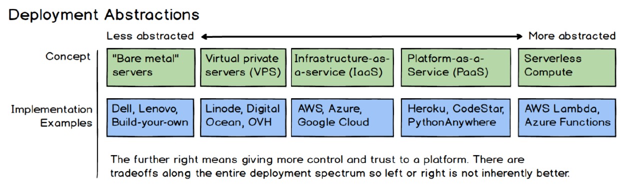


Figure 6: Deployment abstractions: More vs less abstraction

The four pillars of serverless

- No server managment
- Flexible Scaling
- High Availability
- Never Pay for Idle

Serverless FaaS: AWS Lambda

- Triggered by an event
- typically invoked in a few ms (warm start)
- Cold start issue: code that hasn't been used for a while takes longer to start

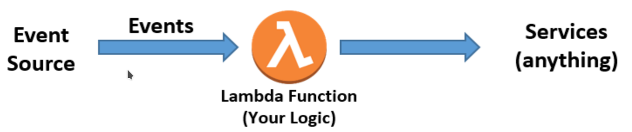


Figure 7: AWS Lambda: Event Triggers

The four stumbling blocks of serverless

- Performance Limitations
- Vendor Lock-in
- Monitoring and Debugging
- Security and Privacy

Serverless usecases

- Event-driven data processing (resize uploaded images)
- Serverless webapplication (simple 3-tier app)
- Mobile and IoT Apps (Airbnb smart home)
- Application Ecosystem (Alexa Skill)
- Event Workflow (image recognition and processing)

L11: Scalable Systems

The Scale Cube

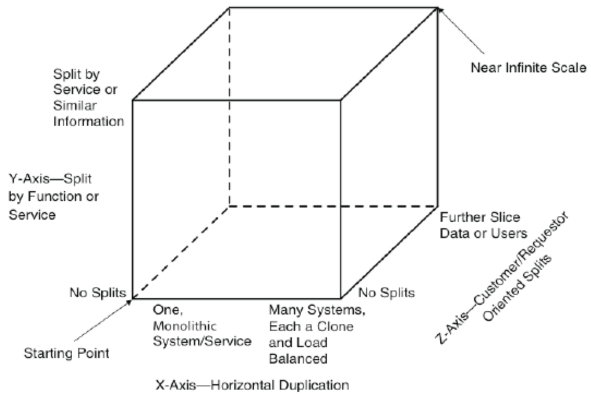


Figure 8: The Scale Cube

- x-axis: **Horizontal Duplication**, unbiased cloning of services and data
- y-axis: **split by function or service**:refers to isolation (making different services)
- z-axis: **partitioning the domain of incoming requests**:data-partitioning, split relevant to client (example: All customers from A-F are together processed, all customers from G-M, etc)

Software architectures

- set of structures needed to reason about the system
- might be implicit

Architectural Components and Patterns for scalable systems

- **Decoupled Components**: allows independent scalability of components; mechanisms to decouple:
  - load balancers
  - message queues
  - message topics
  - service registry
- **Load Balancers**: distributing requests, hiding the server from client access, manage availability (HA),session affinity/sticky sessions
- **Session affinity/sticky sessions**:cookies managed by load balancer(duration based), cookies managed by application cookie
- **LB Algorithms**:(Weighted) Round Robin, Least connections
- **Message Topics**:messages are immeditaley pushed to subscribers, decouple producers and subscribers, concurrent processing
- **Message Queues**: Asynchronous: queue it now but run it later; seperates application logic; introduces latency
- **Service Registries**: resolve addresses for names, Leader voting (*Byzantine Generel*)
- **Automation**: autoscaler as sclaing can not be done manually (Metrics are CPU, RAM, Memory)
- Architectural Patterns: Service oriented architectures; APIs are cloud requirement

L13: MapReduce and GFS/HDFS

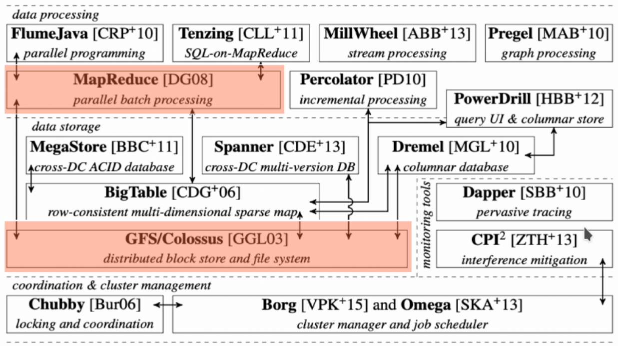


Figure 9: The Google Technology Stack

MapReduce: Basics

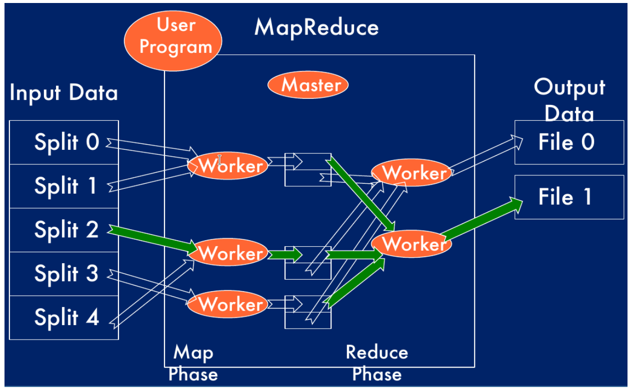


Figure 10: The MapReduce Technology

- we have some input data
- *Map phase*: master process assigns worker processes their part of the data, the data is than processed
- *Reduce phase*: other worker processes collect the processed data and reduce them

AS the master pings the worker and a failure would be noticed really fast. This can now be handled by assigning other processes the task of the failed process.

GFS - Google File System

GFS Objects

- TODO

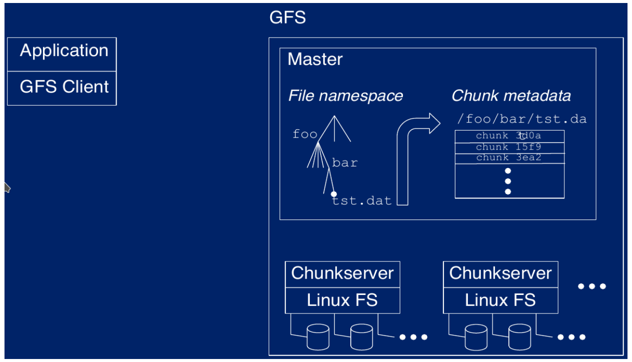


Figure 11: The GFS Architecture

TODO

L14: CAP, Paxos, BGP

CAP Theorem

A good cloud might seek to achieve these three things, but it is only able to select two of them. And as partition tolerance is mandatory for cloud applications we can only choose one of the other two.

- **C**onsistency
- **A**vailability
- **P**artition Tolerance

Paxos Approach

- **Paxos** is an approach to ensuring agreement of a series of asynchronous operations in distributed systems.
- achieve consensus, and ensure agreed actions can not be forgotten anymore
- dspite system messages being duplicate, lost, etc
- Paxos assumes messages are not deliberately malicious (BGP does)

Three paxos rules

- Proposers: learn already accepted values
- Acceptors: let proposers know already accepted values, accept or reject proposals, reach consensus on chosing a particular proposal/value
- Learners: become aware of the chosen proposal/value and action it

Types of Consistency

- **Strong Consistency**: after an update completes, every access will return the same updated value
- **Weak Consistency**: after an update completes, accesses are not guaranteed to return the updated value
- **Eventual Consistency**: eventually all access return the updated value (e.g. updates propagate in a lazy fashion)

Byzantine Generals Problem

Byzantine Faults

- **Byzantine Fault**: different symptoms to different observers

- **Byzantine Failure**: loss of a system service due to Byzantine Fault

Theoretical Problem

- there are a number of generals, each of them with one vote
- some of the generals are traitors and try to foil the other ones, by sending different votes to different generals (instead of the same vote to different generals)
- Key results of BG paper: BG can achieve consensus when $n \geq 3m + 1$ with n loyal generals and m traitors; To do so they must engange in $m + 1$ rounds of message passing
- **Oral Message algorithm**: solves the problem but preventing BFs is very expensive in term of more bandwith and redundancy

L15: The Hadoop Ecosystem

Important Components

- **Pig**: platform for batchmode analysis and large datasets, *Pig Latin* is compiled to use MapReduce
- **Hive**: datawarehouse-software, allows big queries over distributed storage via SQL
- **YARN**: cluster resource managment and job scheduling, middleware layer between HDFS and various application listed here
- **Mahout**: scalable MachineLearning platform, runs on Hadoop/Spark
- **Hoyal/HBase**: HBase is non-relational distributed database (NoSQL), similar to *Google BigTable*
- **Storm**: a distributed real-time stream-processing system
- **Giraph**: graph database, running MapReduce to process graphs
- **Spark**: analytics engine/framework for largescale dataprocessing that runs on YARN

Hadoop Versions

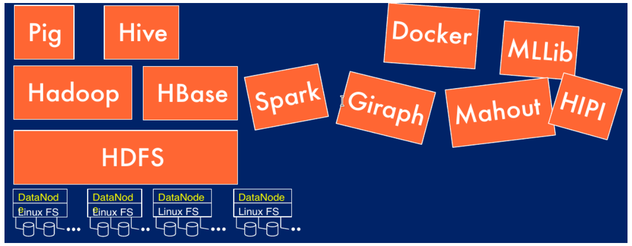


Figure 12: The architecture of Hadoop 1.0

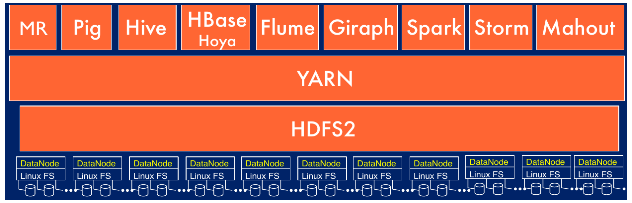


Figure 13: The architecture of Hadoop 2.0

Hive

TODO

Pig

TODO

L16: Spark and In-Memory Methods

-

L17: NoSQL Databases

Main classes of NoSQL databases

- **KeyValue DBs**: e.g. DynamoDB, Redis
- **Document DBs**: e.g. CouchDB, MongoDB
- **Column-Family DBs**: e.g. Cassandra, HBase
- **Graph DBs**: e.g. Giraph

ACID & CRUD

- **ACID**: offered by RDBMSs, **A**tomacity **C**onsistency **I**solation **D**urablility
- **CRUD**: often CRUD is enough, **C**reate, **R**ead, **U**pdate, **D**elete

If you only want CRUD and do not care about the lack of ACID, you can choose NoSQL DBs instead of RDBMSs (which where engineered to run on a single server, which is hard to scale)

KeyValue DBs

- very simple, schemaless
- often very fast
- DBs have different constraints (ACID, object limit size, etc)

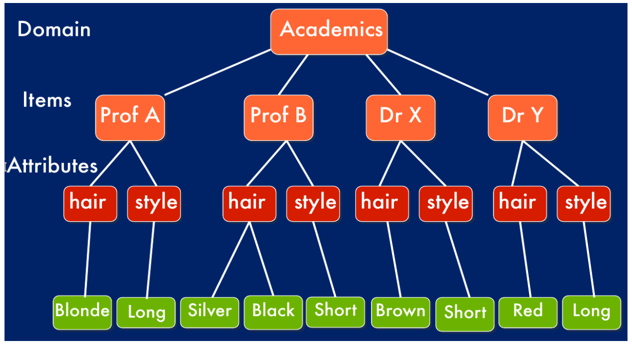


Figure 14: Example of a KeyValue DB: Amazon Simple DB

Document DBs

- manage more complex data structures than KV datastores
- do not require you to define common structure for all datasets (like KV stores)
- a *document* is a structured object, most commonly used are *XML* and *JSON*

Column-Family DBs & Columnar DBs

- for *big data*, so *VLDB* (Very Large Database)
- frequently used columns can be grouped together
- *families* are like relational tables, inividual columns are more like key-value pairs

Choices

- **Relational**:
  - *good for*: when layout of data is known in before, but exact queries are not
  - *less good for*: when data is highly variable or deeply hierarchical
- **Key Value**:
  - *good for*: data largely independant, horizontal scaling, CRUD
  - *less good for*: perform non trivial queries
- **Document**:
  - *good for*: highly variable data, storing redundant data is not a problem
  - *less good for*: when data needs to be normalized
- **Column-family**:
  - *good for*: Big Data, data compression, have an idea how the queries will look like
  - *less good for*: when you don't know how data will be quiered
- **Graph**:
  - *good for*: applications with *networks of relationships*
  - *less good for*: large scale situations where partitioning across nodes is necessary

⇒ *polyglot persistence model*: Use more databases, each playing a different role

L18: Graph Databases

Network Basics

- a network/**Graph** consists of nodes/points/**Vertices**, which are connected with lines/**Edges**
- **edges** are *directed or undirected*, may be *self-connections*, may be *weighted*, may be *multi-edges or hyper-edges*
- a **clicque** is a subgraph that is complete
- a **path** is a sequence of edges connecting two or more vertices, a **cycle** is a closed path
- node's *indegree* counts ways to arrive it
- node's *outdegree* counts ways to leave
- on a directed graph indegree and outdegree may differ
- Representing Graphs:
  - *Adjacency List*
  - *Adjacency Matrices*: the power of the matrix indicates the lenght of the path

PageRank

- PageRank for ranking webpage, where often linked pages are higher ranked
- *Transition Matrix*: indicates the weight of each outgoing link/ edge (each page has 100% for all his links together)
- *Dangling nodes problem*: A node with no outgoing edges simply jumps to a random site
- *Damping*: off-network transitions appear now not with 0 but with ϵ (very small) probability, (there is a dumping factor to specify this)

Graph-processing

Pregel

- **Pregel** is a Google framework for network analysis
- it provides high scalability, fault-tolerance, flexibility in expressing arbitrary graph algorithms
- operates over a directed graph

Apache Giraph

- open source equivalent of *Pregel*
- integration in Hadoop ecosystem, efficiently loading data from HBase

Bulk Sychronous Parallel Model

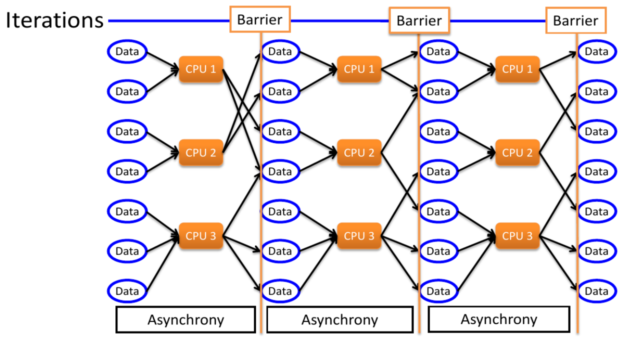


Figure 15: Bulk Synchronous Parallel Model

- 3 steps:
  - *Concurrent Computation*: computation exclusively local to each node
  - *Communication*: messaging between nodes
  - *Barrier synchronisation*: checkpoints, block globally nodes before processing signalling received in a *Superstep*
- a sequence of *supersteps* are run analogous to MapReduce
- during a superstep nodes can send messages, receive message sent in the last superstep, the function can modify the edges/state of verteces
- *Algorithm Termination*: initinally every node is active, after finishing computation a node sets his state to inactive (until it receives a message and becomes active again), the algorithm terminates when all nodes are simultaneously inactive and there is no message in transit
- can be run distributed, then there is a master copy of the program managing/advicing all worker copies
- fault tolerance: if a worker fails the master reassigns its vertices to another worker and restart the last superstep

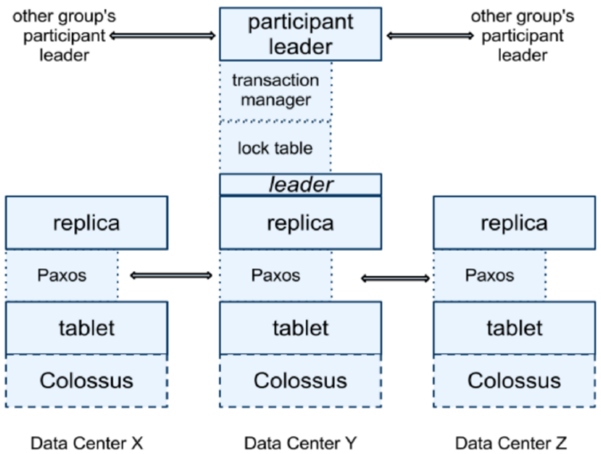
L19: NewSQL & Event Stream Processing

NewSQL properties

- flexibility and scalability of NoSQL, support for SQL and/or ACID as RDBMS
- SQL as the primary interface
- ACID support for transactions
- Nonlocking concurrency controll
- high per node performance
- parallel shared nothing architecture
- an example for this might be *Google Spanner*

Google Spanner

- lockfree distributed read-only transactions
- external consistency (ACID transactions)
- SQL queries
- officially AP (in terms of CAP)
- scales horizontally
- Spaner Zone Server Organisation:
  - *Zonemaster* assigns data to *span servers*, 1 per zone
  - *Span server* serves data to clients, 300 - 1000 per zone
  - *Location proxies* are used by clients to locate span servers with correct data
  - *Universe master* has a status console, 1 per universe
  - Zone Software Stack is part of **??**
- Spaner Application Model: supports unlimited tables in database within the universe, data actually stored on 3 replicas
- Open Source DBs based on Spanner: *CockroachDB*, *YugaByte*



Google TrueTime

- used in and makes **Google Spanner** possible
- every Google datacentre has an atomic clock
- clients poll TimeMasters (GPS and atomic clocks)
- TrueTime API is Google proprietary

Google F1 Database

- database built on top of Spanner
- five replicas needed for HA
- Geography: replicas spread around the world to survive regional disasters
- High Throughput, but different approaches have to be done to deal with high latency
- One System for *OLAP* and *OLTP* (=*HTAP*)

Real Time Event Streaming

- different use cases (e.g. network and Infrastructure monitoring, sensor data, commerce and retail marketing)
- Requirements: data arrives in real time (online), High throughput, low latency, FT

Systems for RTSP

- General Architecture: Components(Source, Processing, Sinks), Framework provides message queueing, ressoure manager to allocate nodes tasks

Apache Storm

- pioneer in realtime processing
- works on unbounded stream of messages
- appropriate for low latency
- *Storm provides the transformation of streams, so to transform one stream into another one.*
- topology runs forever or until it gets killed. No data loss is guaranteed even in the case of machine failures or messages get dropped
- Problems for Storm: no exactly once-guarantees, consistency, difficult to combine batch and stream processing, straggler tolerance

Discetrized Streams and Spark Streaming

- *Buffer and Compute* pattern
- near real time
- avoid overhead from Disk I/O that Hadoop faced

TODO

Apache Kafka

- TODO

L20: Cloud Security

-

L21: DevOp

Todo...

Possible exam Questions

Example Exam Questions

- 1) Contrast Self-hosting, IaaS, PaaS and SaaS using different modes of transportation as a metaphor.
- 2) Name the four factors that can influence the decision of which AWS Region to use for an application.
- 3) On AWS, what is a critical requirement to enable High Availability (HA)?
- 4) What is the difference between High Availability and Fault Tolerance?
- 5) On AWS EC2, contrast the EBS and Instance Store volume types.
- 6) Mention one AWS example service from IaaS, PaaS and SaaS.

Exam Question

- What is another name for operating system – level virtualization?
- What is the role of hypervisor?

Example Exam Questions

- 1) Describe 2 benefits and 2 drawbacks to serverless technologies.
- 2) List the serverless architectural layers.
- 3) Explain the concept of ‘cold start’ in a serverless application.
- 4) “You should avoid AWS due to vendor lock-in!”. Discuss.
- 5) Name 4 parameters that must be specified when configuring a Lambda function.
- 6) Name three serverless use cases.

- Why might a high-degree node be considered important? *[1 mark]*
- If the cell *i,j* in adjacency matrix *A* contains the value 0, but the same cell in A^4 (i.e., *A* raised to the power 4) has value 1, what do we know about the network that *A* represents? *[2 marks]*
- What do teleportation and damping achieve for the PageRank algorithm? *[8 marks]*
- Write a Pregel pseudocode that assigns the lowest node ID in each network component to all nodes in that component. *[7 marks]*

Example Exam Questions

- 1) What are two significant issues with traditional NoSQL databases?
- 2) Name two use cases where a NewSQL database would be preferred to a traditional NoSQL type?
- 3) Describe what makes Google Spanner a compelling alternative to MySQL?
- 4) Name three open source frameworks that can be used for Event Stream Processing.
- 5) Explain two disadvantages when using Apache Storm for ESP.
- 6) Name four main components of Kafka, with a brief description of the function of each.