Topic 9

Cloud Application Paradigms

University of Cyprus
Department of Computer Science

Readings

- **"MapReduce: Simplified Data Processing on Large Clusters" (2004)** Jeffrey Dean and Sanjay Ghemawat (Google). Usenix OSDI.
- **"The Google File System"** (2003) Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. SOSP.
- **"The rise of serverless computing"** (2019) P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, Commun. ACM, vol. 62, no. 12, pp. 44–54, Nov. 2019.
- **"Cloud Programming Simplified: A Berkeley View on Serverless Computing"** (2019) E. Jonas et al., Feb. 2019.

Cloud Applications Paradigms

Big-Data Processing: MapReduce & GFS

University of Cyprus
Department of Computer Science

Batch Processing

- Processing large amounts of data at-once, in one-go to deliver a result according to a query on the data.

Motivation: Processing Large Data-sets

- Need for many computations over large/huge sets of data:
  - Input data: crawled documents, web request logs
  - Output data: inverted indices, summary of pages crawled per host, the set of the most frequent queries in a given day, …
- Most of these computation are relatively straight-forward
- To speedup computation and shorten processing time, we can distribute data across 100s of machines and process them in parallel
- But, parallel computations are difficult and complex to manage:
  - Race conditions, debugging, data distribution, fault-tolerance, load balancing, etc
- Ideally, we would like to process data in parallel but not deal with the complexity of parallelisation and data distribution

MapReduce

- "A new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library."
- Programming model:
  - Provides abstraction to express computation
- Library:
  - To take care the runtime parallelisation of the computation.

Count the number of occurrences of each word in a text file:

"Cloud computing is a recently evolved computing terminology or metaphor based on utility and consumption of computing resources. Cloud computing involves deploying groups of remote servers and software networks that allow centralized data storage and online access to computer services or resources."

Identify each word per line. For each word identified emit <word, 1>:

'Cloud computing is a recently evolved computing terminology or metaphor based on utility and consumption of computing resources. Cloud computing involves deploying groups of remote servers and software networks that allow centralized data storage and online access to computer services or resources."

cloud	1
computing	1
is	1
a	1
recently	1
evolved	1
computing	1
terminology	1
or	1
metaphor	1

Identify each word per line. For each word identified emit <word, 1>:

"Cloud computing is a recently evolved computing terminology or metaphor based on utility and consumption of computing resources. Cloud computing involves deploying groups of remote servers and software networks that allow centralized data storage and online access to computer services or resources."

cloud	1	based	1
computing	1	on	1
is	1	utility	1
a	1	and	1
recently	1	consumption	1
evolved	1	of	1
computing	1	computing	1
terminology	1	resources	1
or	1	cloud	1
metaphor	1	computing	1

Identify each word per line. For each word identified emit <word, 1>:

"Cloud computing is a recently evolved computing terminology or metaphor based on utility and consumption of computing resources. Cloud computing involves deploying groups of remote servers and software networks that allow centralized data storage and online access to computer services or resources."

cloud	1	based	1	involves	1
computing	1	on	1	deploying	1
is	1	utility	1	groups	1
a	1	and	1	of	1
recently	1	consumption	1	remote	1
evolved	1	of	1	servers	1
computing	1	computing	1	and	1
terminology	1	resources	1	software	1
or	1	cloud	1	networks	1
metaphor	1	computing	1	that	1
				allow	1

Identify each word per line. For each word identified emit <word, 1>:

"Cloud computing is a recently evolved computing terminology or metaphor based on utility and consumption of computing resources. Cloud computing involves deploying groups of remote servers and software networks that allow centralized data storage and online access to computer services or resources."

cloud	1	based	1	involves	1	centralized	1
computing	1	on	1	deploying	1	data	1
is	1	utility	1	groups	1	storage	1
a	1	and	1	of	1	and	1
recently	1	consumption	1	remote	1	online	1
evolved	1	of	1	servers	1	access	1
computing	1	computing	1	and	1	to	1
terminology	1	resources	1	software	1	computer	1
or	1	cloud	1	networks	1	services	1
metaphor	1	computing	1	that	1	or	1
				allow	1	resources	1

Sort key-value pairs <word, 1> per key (word):

a	1	data	1	recently	1
access	1	deploying	1	remote	1
allow	1	evolved	1	resources	1
and	1	groups	1	resources	1
and	1	involves	1	servers	1
and	1	is	1	services	1
based	1	metaphor	1	software	1
centralized	1	networks	1	storage	1
cloud	1	of	1	terminology	1
cloud	1	of	1	that	1
computing	1	on	1	to	1
computing	1	online	1	utility	1
computing	1	or	1		
computing	1	or	1		
computer	1				
consumption	1				

Aggregate the counts per each word in the sorted <k, v> list below:

a	1
access	1
allow	1
and	3
based	1
centralized	1
cloud	2
computing	4
computer	1
consumption	1
data	1
deploying	1
evolved	1
groups	1
involves	1
is	1
metaphor	1
networks	1
of	2
on	1
online	1
or	2
recently	1
remote	1
resources	2
servers	1
services	1
software	1
storage	1
terminology	1
that	1
to	1
utility	1

Generalizing the Computation

- Apply a **map** operation to each logical "record" in the input to compute a set of intermediate key/value pairs:
  - In this case, **each text file** is mapped to a set of <w, 1>, where the keys correspond to words found in the file.
- Apply a **reduce** operation to all the values that share the same key, combining the derived data, with an appropriate reduce function.
  - In this case, for each word w collect a list **<w, {1, 1,…}>** and reduce it to **<w, $count_w$>**

Count the number of occurrences of each word in the text below:

"Cloud computing is a recently evolved computing terminology or metaphor based on utility and consumption of computing resources. Cloud computing involves deploying groups of remote servers and software networks that allow centralized data storage and online access to computer services or resources."

How do you solve this problem if the **file is very large** and you have **multiple CPUs** in your disposal?

Say you have 4 nodes.

Partition the input data to 4 nodes and run the algorithm in parallel

"Cloud computing is a recently evolved computing terminology or metaphor based on utility and consumption of computing resources. Cloud computing involves deploying groups of remote servers and software networks that allow centralized data storage and online access to computer services or resources."

cloud	1
computing	1
is	1
a	1
recently	1
evolved	1
computing	1
terminology	1
or	1
metaphor	1

based	1
on	1
utility	1
and	1
consumption	1
of	1
computing	1
resources	1
cloud	1
computing	1

involves	1
deploying	1
groups	1
of	1
remote	1
servers	1
and	1
software	1
networks	1
that	1
allow	1

centralized	1
data	1
storage	1
and	1
online	1
access	1
to	1
computer	1
services	1
or	1
resources	1

Sort each list

"Cloud computing is a recently evolved computing terminology or metaphor based on utility and consumption of computing resources. Cloud computing involves deploying groups of remote servers and software networks that allow centralized data storage and online access to computer services or resources."

a	1
cloud	1
cloud	1
computing	1
evolved	1
is	1
metaphor	1
or	1
recently	1
terminology	1

and	1
based	1
cloud	1
computing	1
computing	1
consumption	1
of	1
on	1
resources	1
utility	1

and	1
allow	1
deploying	1
groups	1
involves	1
networks	1
of	1
remote	1
servers	1
software	1
that	1

access	1
and	1
centralized	1
computer	1
data	1
storage	1
to	1
online	1
or	1
resources	1
services	1

Observe that some words appear in lists on different nodes

a	1
cloud	1
cloud	1
computing	1
evolved	1
is	1
metaphor	1
or	1
recently	1
terminology	1

and	1
based	1
cloud	1
computing	1
computing	1
consumption	1
of	1
on	1
resources	1
utility	1

and	1
allow	1
deploying	1
groups	1
involves	1
networks	1
of	1
remote	1
servers	1
software	1
that	1

access	1
and	1
centralized	1
computer	1
data	1
storage	1
to	1
online	1
or	1
resources	1
services	1

What is the next step?

a	1
cloud	1
cloud	1
computing	1
evolved	1
is	1
metaphor	1
or	1
recently	1
terminology	1

and	1
based	1
cloud	1
computing	1
computing	1
consumption	1
of	1
on	1
resources	1
utility	1

and	1
allow	1
deploying	1
groups	1
involves	1
networks	1
of	1
remote	1
servers	1
software	1
that	1

access	1
and	1
centralized	1
computer	1
data	1
storage	1
to	1
online	1
or	1
resources	1
services	1

Collect all lists to one node

a	1
cloud	1
cloud	1
computing	1
evolved	1
is	1
metaphor	1
or	1
recently	1
terminology	1

and	1
based	1
cloud	1
computing	1
computing	1
consumption	1
of	1
on	1
resources	1
utility	1

and	1
allow	1
deploying	1
groups	1
involves	1
networks	1
of	1
remote	1
servers	1
software	1
that	1

access	1
and	1
centralized	1
computer	1
data	1
storage	1
to	1
online	1
or	1
resources	1
services	1

a	1	and	1	and	1	access	1
cloud	1	based	1	allow	1	and	1
cloud	1	cloud	1	deploying	1	centralized	1
computing	1	computing	1	groups	1	computer	1
evolved	1	computing	1	involves	1	data	1
is	1	consumption	1	networks	1	storage	1
metaphor	1	of	1	of	1	to	1
or	1	on	1	remote	1	online	1
recently	1	Resources	1	servers	1	or	1
terminology	1	utility	1	software	1	resources	1
				that	1	services	1

Sort by word (key)

a	1	data	1	recently	1
access	1	deploying	1	remote	1
allow	1	evolved	1	resources	1
and	1	groups	1	resources	1
and	1	involves	1	servers	1
and	1	is	1	services	1
based	1	metaphor	1	software	1
centralized	1	networks	1	storage	1
cloud	1	of	1	terminology	1
cloud	1	of	1	that	1
computing	1	on	1	to	1
computing	1	online	1	utility	1
computing	1	or	1		
computing	1	or	1		
computer	1				
consumption	1				

Aggregate counts per each word

a	1	data	1	recently	1
access	1	deploying	1	remote	1
allow	1	evolved	1	resources	2
and	3	groups	1	servers	1
based	1	involves	1	services	1
centralized	1	is	1	software	1
cloud	2	metaphor	1	storage	1
computing	4	networks	1	terminology	1
computer	1	of	2	that	1
consumption	1	on	1	to	1
		online	1	utility	1
		or	2		

What if the lists do not fit in one node?

Partition the lists to R nodes

(Use a partitioning function e.g. hash(key) mod R)

a	1
cloud	1
cloud	1
computing	1
evolved	1
is	1
metaphor	1
or	1
recently	1
terminology	1

and	1
based	1
cloud	1
computing	1
computing	1
consumption	1
of	1
on	1
resources	1
utility	1

and	1
allow	1
deploying	1
groups	1
involves	1
networks	1
of	1
remote	1
servers	1
software	1
that	1

access	1
and	1
centralized	1
computer	1
data	1
storage	1
to	1
online	1
or	1
resources	1
services	1

a	1	and	1	and	1	access	1
cloud	1	based	1	allow	1	and	1
cloud	1	cloud	1	deploying	1	centralized	1
computing	1	computing	1	groups	1	computer	1
evolved	1	computing	1	involves	1	data	1
is	1	consumption	1				

metaphor	1	of	1	networks	1	storage	1
or	1	on	1	of	1	to	1
recently	1	resources	1	remote	1	online	1
terminology	1	utility	1	servers	1	or	1
				software	1	resources	1
				that	1	services	1

Sort the lists in parallel

a	1
access	1
allow	1
and	1
and	1
and	1
based	1
centralized	1
cloud	1
cloud	1
computing	1
computing	1
computing	1
computing	1
computer	1
consumption	1
data	1
deploying	1
evolved	1
groups	1
involves	1
is	1

metaphor	1
networks	1
of	1
of	1
on	1
online	1
or	1
or	1
recently	1
remote	1
resources	1
resources	1
servers	1
services	1
software	1
storage	1
terminology	1
that	1
to	1
utility	1

Aggregate counts

a	1	data	1
access	1	deploying	1
allow	1	evolved	1
and	3	groups	1
based	1	involves	1
centralized	1	is	1
cloud	2		
computing	4		
computer	1		
consumption	1		

metaphor	1	recently	1
networks	1	remote	1
of	2	resources	2
on	1	servers	1
online	1	services	1
or	2	software	1
		storage	1
		terminology	1
		that	1
		to	1
		utility	1

Programming Model

- **Input**: a set of key/value pairs
- **Output**: a set of key/value pairs
- Computation is expressed using the two functions:
  - Map task: single pair → list of intermediate pairs
    - **map(input-key, input-value)** → **list(out-key, intermediate-value)**
    - e.g. <k1, v1> → { < k2, v2 >,… }
  - Reduce task: all intermediate pairs with the same **k2** → a list of values
    - **reduce(out-key, list(intermediate-value))** → **list(out-values)**
    - e.g. < k2, {v2,…} > → {v2,… }

Programming Model

- The **Map** function, *written by the user*, takes an input pair <*k1,v1*> and produces a set of intermediate key/value pairs {<*k2, v2*>,…}
- The **MapReduce library** groups together all intermediate values associated with the same intermediate key <*k2, {v2,…}*> and passes them to the Reduce function.
- The **Reduce** function, *also written by the user*, accepts an intermediate key *k2* and a set of values for that key *{v2,…}*
  - It merges together these values to form a possibly smaller set of values.
  - Typically just zero or one output value is produced per Reduce invocation.
  - The intermediate values are supplied to the user's reduce function via an **iterator**. This allows to handle lists of values that are too large to fit in memory.

Example: Counting the number of occurrences of each word in a collection of documents

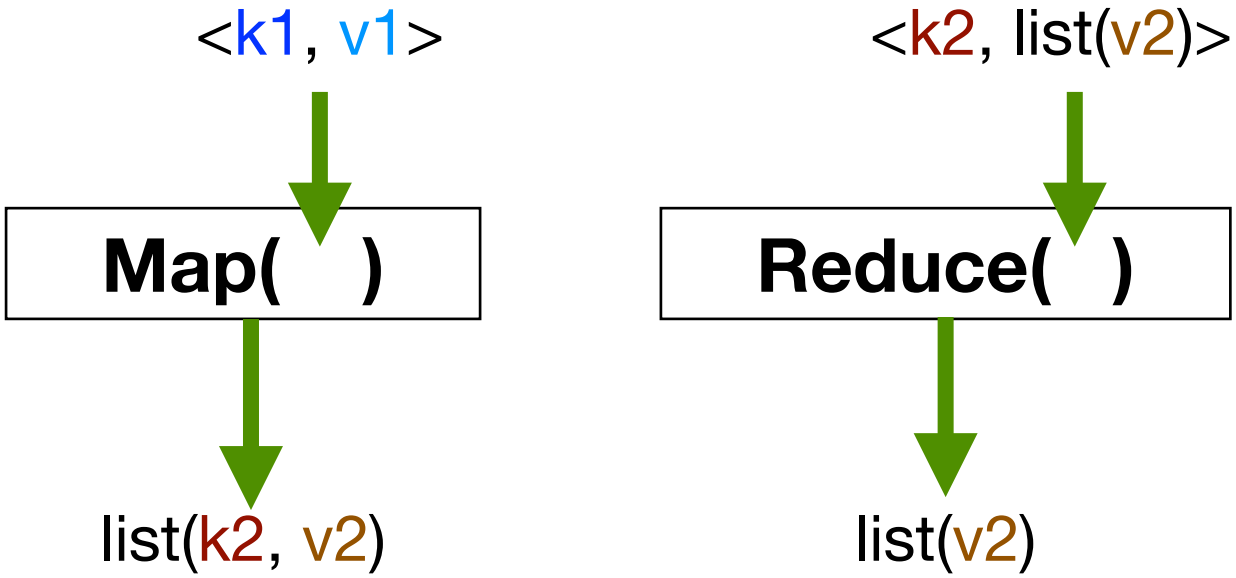
```
map(String input_key, String input_value):
    // input_key: document name
    // input_value: document contents
    for each word w in input_value:
      EmitIntermediate(w, "1");
reduce(String output_key, Iterator intermediate_values):
    // output_key: a word
    // output_values: a list of counts
    int result = 0;
    for each v in intermediate_values:
      result += ParseInt(v);
    Emit(AsString(result));
```

Abstraction & Implementation

- The Map-Reduce programming model provides a simple abstraction that captures the essence of the computation and allows programmers to express simple computations while…
- … hiding the messy details of data partitioning, parallelization, fault-tolerance, data distribution and load balancing in a library implementation.
- Main contribution: a **simple and powerful interface** that enables **automatic parallelization** and **distribution** of **large-scale computations**, combined with an **implementation** of this interface that achieves high performance on large clusters of commodity PCs.

What is the abstraction provided by MapReduce?


Takeaway


<k1, v1>
Map( )
list(k2, v2)
<k2, list(v2)>
Reduce( )
list(v2)

MapReduce Example Applications

- The MapReduce model can be applied to many applications:
  - Distributed grep:
    - map: emits a line, if line matched the pattern
    - reduce: identity function
  - Count of URL access Frequency
  - Reverse Web-Link Graph
  - Inverted Index
  - Distributed Sort
  - ….

Reverse Web-Link Graph

- For every web page in a given set of Web pages return a list of the origin pages of its **incoming links.**
- Give the code in the map-reduce abstraction.

MapReduce Implementation

- MapReduce implementation for (2006):
  - Large cluster of commodity PCs connected via switched Ethernet
  - Machines are typically dual-processor x86, running Linux, 2-4GB of mem! (slow machines for today's standards)
  - A cluster of machines, so failures are anticipated
  - Storage with (GFS) Google File System (2003) on IDE disks attached to PCs. GFS is a distributed file system, uses replication for availability and reliability.
- Scheduling system:
  - Users submit jobs
  - Each job consists of tasks; scheduler assigns tasks to machines

Parallel Execution on Google Cloud

- User specifies:
  - **M**: number of map tasks
  - **R**: number of reduce tasks
- Map:
  - MapReduce library splits the input file into **M** pieces
  - Typically 16-64MB per piece
  - Map tasks are distributed across the machines
- Reduce:
  - Partitioning the intermediate key space into **R** pieces
  - hash(intermediate_key) **mod** R
- Typical setting:
  - 2,000 machines, M = 200,000, R = 5,000

Execution Flow

1. The MapReduce library in the user program splits the input files into M pieces of typically 16 megabytes to 64 megabytes (MB) per piece (parameterized). It then starts up many copies of the program on a cluster of machines.
2. One of the copies of the program is special – the **master**. The rest are **workers** (assigned work by the master).
   - There are M map tasks and R reduce tasks to assign.
   - The master picks idle workers and assigns each one a map task or a reduce task.
3. A worker who is assigned a **map task**:
   - Reads the contents of the corresponding input split.
   - Parses key/value pairs out of the input data.
   - Passes each pair to the user-defined Map function.
   - The intermediate key/value pairs produced by the Map function are buffered in memory.

Execution Flow

4. Periodically, the buffered pairs are written to local disk, partitioned into R regions by the partitioning function.
   - The locations of these buffered pairs on the local disk are passed back to the master, who is responsible for forwarding these locations to the reduce workers.
5. When a **reduce worker** is notified by the master about these locations, it uses remote procedure calls to read the buffered data from the local disks of the map workers. When a reduce worker **has read all intermediate data,** it sorts it by the intermediate keys so that all occurrences of the same key are grouped together.
   - **Sorting is needed** because typically many different keys map to the same reduce task.
   - If the amount of intermediate data is too large to fit in memory, an **external sort** is used.
6. The **reduce worker** iterates over the sorted intermediate data and
   - For each unique intermediate key encountered, it passes the key and the corresponding set of intermediate values to the **user's Reduce function**.
   - The output of the Reduce function is appended to a final output file for this reduce partition.

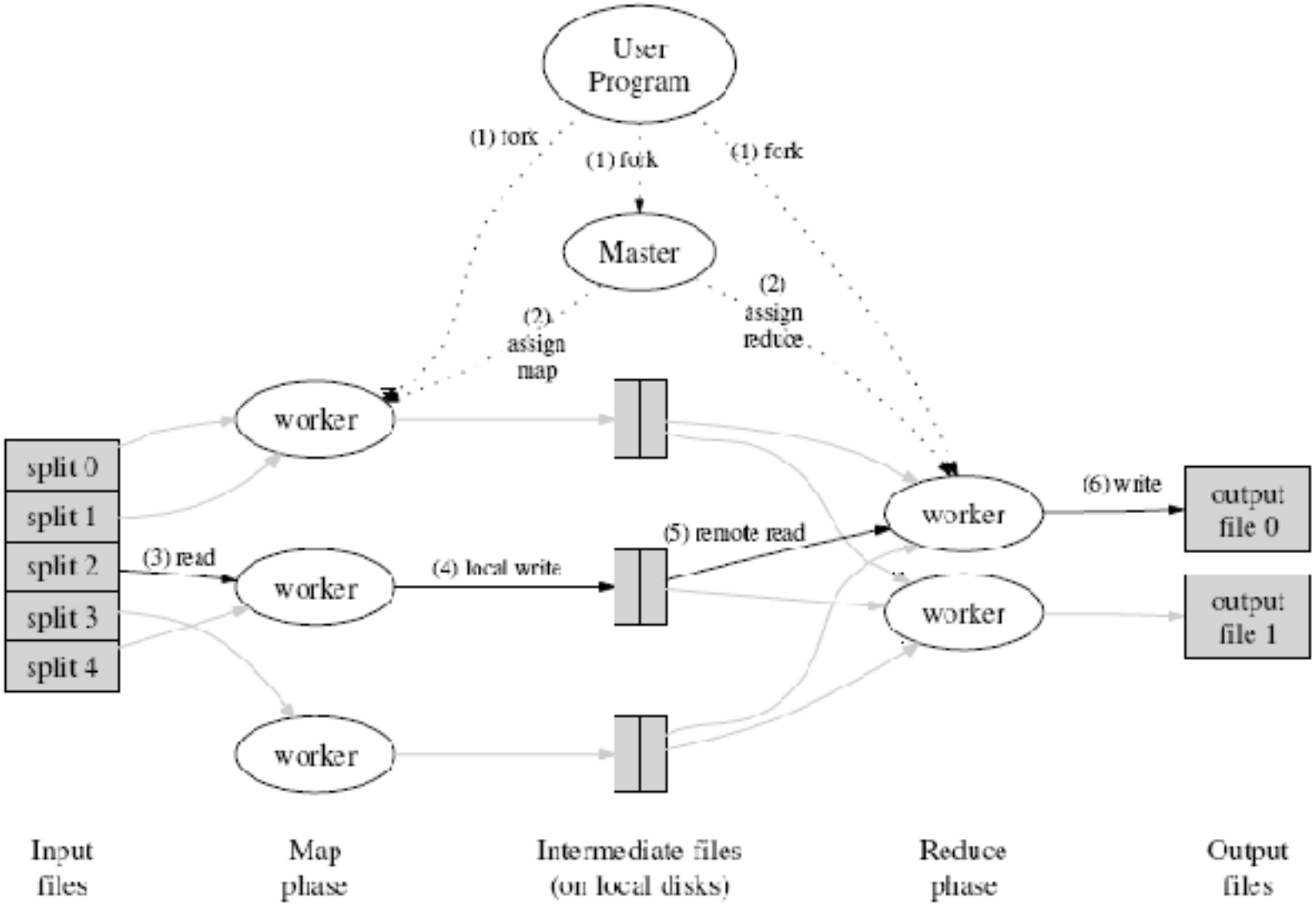
Execution Flow

7. When all map tasks and reduce tasks have been completed, the master wakes up the user program. At this point, the MapReduce call in the user program returns back to the user code.

8. After successful completion, the output of the MapReduce execution is available in the **R output files** (one per reduce task, with file names as specified by the user).

- Typically, users do not need to combine these R output files into one file:
  - they often pass these files as input to another MapReduce call, or
  - use them from another distributed application that is able to deal with input that is partitioned into multiple files.

Execution Flow



Master Data Structures

- For each map/reduce task:
  - State status {**idle**, **in-progress**, **completed**}
  - Identity of the worker machine (for non-idle tasks)
- The location of intermediate file regions is passed from maps to reducers tasks through the master.
  - This information is pushed incrementally (as map tasks finish) to workers that have in-progress reduce tasks.

Fault-Tolerance

Two types of failures:

- Worker failures:
  - Identified by sending heartbeat messages by the master.
  - If no response within a certain amount of time, then the worker is dead.
  - **In-progress** and **completed** map tasks are re-scheduled → **idle**
  - **In-progress** reduce tasks are re-scheduled → **idle**
  - Workers executing reduce tasks affected from failed map/workers are notified of re-scheduling
- Question: **Why completed map tasks have to be re-scheduled?**
- **Answer**: Map output is stored on local fs, while reduce output is stored on GFS
- Master failure:
  - Rare
  - Can be recovered from checkpoints
  - Solution: aborts the MapReduce computation and starts again

Disk Locality

- Network bandwidth is a relatively scarce resource and also increases latency
- The goal is to save network bandwidth
- Use of **GFS** that stores typically three copies of the data block on different machines
- Map tasks are scheduled "close" to data
  - On nodes that have input data (local disk)
  - If not, on nodes that are nearer to input data (e.g., same switch)

Task Granularity

- Number of map tasks > number of worker nodes
  - Better load balancing.
  - Better recovery.
- But, this, increases load on the master
  - More scheduling.
  - More states to be saved.
- **M** could be chosen with respect to the block size of the file system
  - For locality properties.
- Number **R** of partitions of intermediate results is usually specified by the user
  - Each reduce task produces one output file.

Stragglers

- Slow workers delay overall completion time → stragglers
  - Bad disks with soft errors
  - Other tasks using up resources
  - Machine configuration problems, etc
- Very close to end of MapReduce operation, master schedules backup execution of the remaining in-progress tasks.
- A task is marked as complete whenever either the primary or the backup execution completes.
- Example: a sort operation of an example given in the Map-Reduce paper takes **44% longer** to complete when the **backup** task mechanism is **disabled**.

Refinements: Partitioning Function

- Partitioning function identifies the reduce task
  - Users specify the desired output files they want, R
  - But, often there are more keys than R
  - Use the intermediate key and R for partitioning: hash(key) mod R
- Important to choose well-balanced partitioning functions. E.g, for output keys that are URLs:
  - hash(hostname(urlkey)) mod R

Refinements: Combiner Function

- Introduce a mini-reduce phase **before** intermediate data is sent to reduce
- When there is significant repetition of intermediate keys
  - Merge values of intermediate keys before sending to reduce tasks
  - Example: word count, many records of the form <word_name, 1>. Merge records with the same word_name
  - Similar to reduce function
- Saves network bandwidth

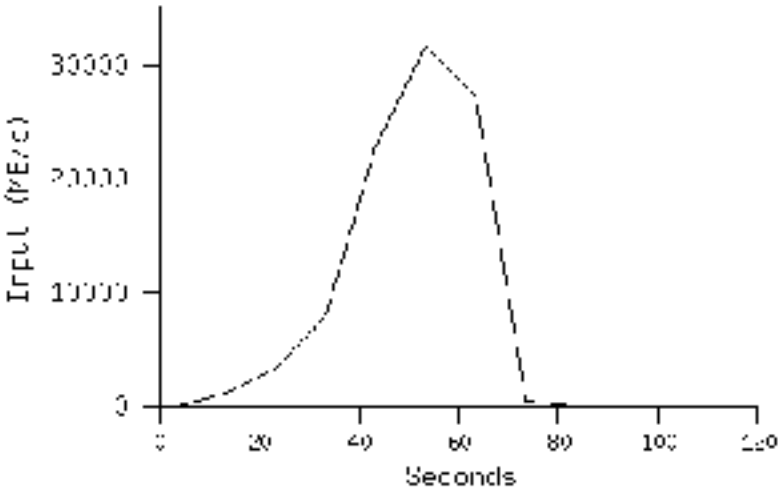
Evaluation - Setup

- Evaluation on two programs running on a large cluster and processing 1 TB of data:
  1. **grep:** search over 10^{10} 100-byte records looking for a rare 3-character pattern
  2. **sort:** sorts 10^{10} 100-byte records

- Cluster configuration:
  - 1,800 machines
  - Each machine has 2 GHz Intel Xeon proc., 4GB mem, 2 160GB IDE disks
  - Gigabit Ethernet link
  - Hosted in the same facility

Grep

- M = 15,000 of 64MB each split
- R = 1
- Entire computation finishes at 150s
- Startup overhead ~60s
  - Propagation of program to workers
  - Delays to interact with GFS
  - …
- Picks at 30GB/s with 1,764

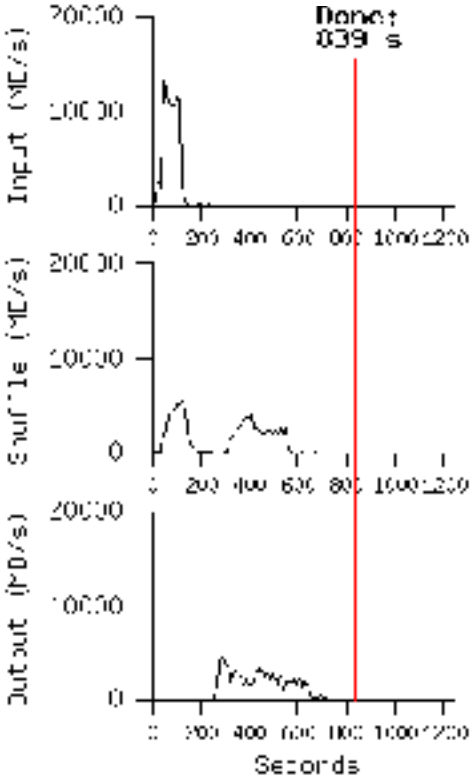


Sort

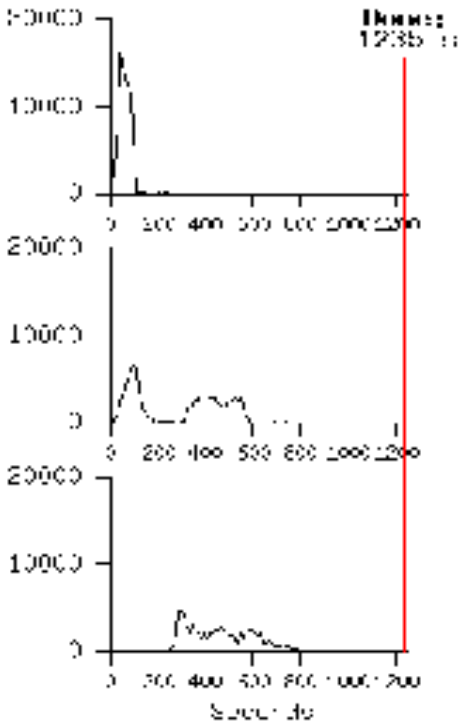
- M = 15,000 splits, 64MB each
- R = 4,000 files
- Workers = 1,700
- Evaluated on three executions:
  - With backup tasks
  - Without backup tasks
  - With machine failures

Sort Results

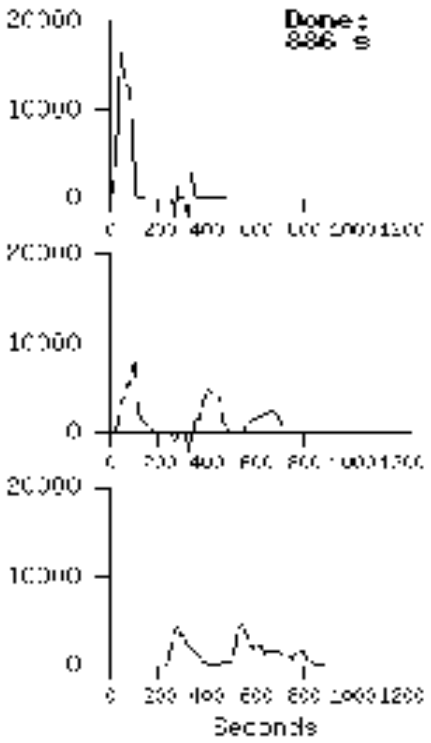
Top: rate at which input is read
Middle: rate at which data is sent from mappers to reducers
Bottom: rate at which sorted data is written to output file by reducers



Normal execution with backup



Without backup tasks, 5 reduce tasks stragglers, 44% increase



With machine failures, 200 out of 1746 workers,

- a 5% increase over normal execution time

Implementation

- First MapReduce library in 02/2003
- Use cases (back then):
  - Large-scale machine learning problems
  - Clustering problems for the Google News
  - Extraction of data for reports Google zeitgeist
  - Large-scale graph computations

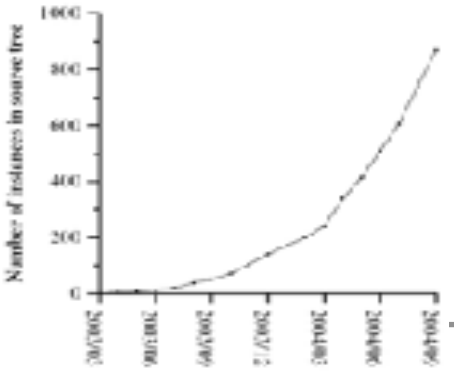


Figure 4: MapReduce instances over time

MapReduce jobs run in 8/2004

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

Summary

- MapReduce is a simple, very powerful and expressive model.
- MapReduce libraries hide behind their simple programming abstractions, important and complex implementation details regarding management of faults, scheduling and load-balancing.
- Performance depends a lot on implementation details.
- Implementation becomes more challenging when dealing with heterogeneous clusters.

Apache Hadoop

- The Apache Hadoop software library is a framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models.
- It is designed to scale up from single servers to 1000s of machines, each offering local computation and storage.
- Rather than rely on hardware to deliver high-availability, the library itself is designed to detect and handle failures at the application layer, so delivering a highly-available service on top of a cluster of failure-prone computers.

Hadoop Modules

- Hadoop Common: The common utilities that support the other Hadoop modules.
- Hadoop Distributed File System (HDFS™): A distributed file system that provides high-throughput access to application data.
- Hadoop YARN: A framework for job scheduling and cluster resource management.
- Hadoop MapReduce: A YARN-based system for parallel processing of large data sets.

Hadoop Project Components

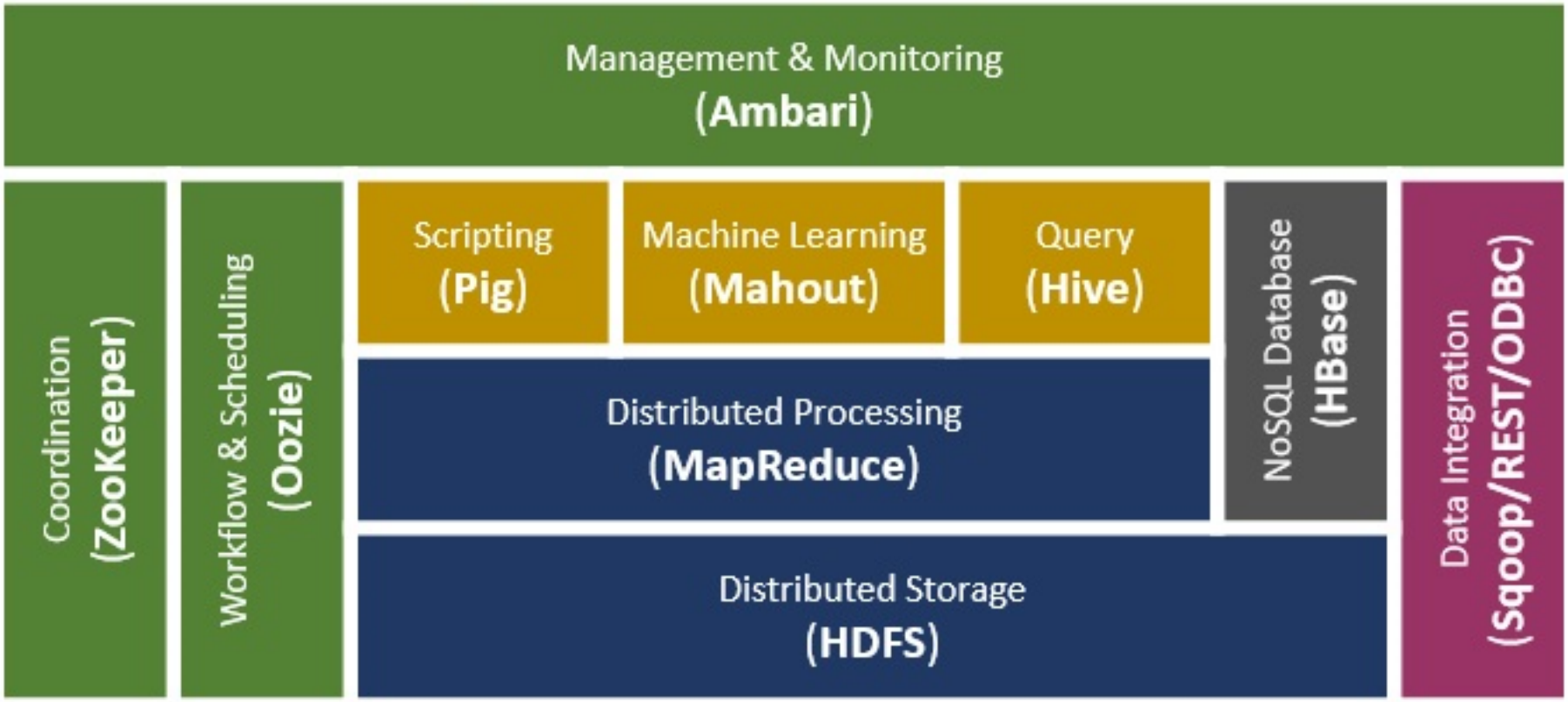


Image Credit: mssqltips.com

Cloud Storage

GFS: The Google File System

University of Cyprus
Department of Computer Science

GFS in a Nutshell

- Developed by Google as scalable distributed file system for large distributed **data intensive applications**:
  - Fault tolerance while running on inexpensive commodity hardware.
  - High aggregate performance to a large number of clients.
  - A distributed file storage.
  - Efficient, reliable access to data.
  - Co-designing the applications and the file system API benefits the overall system by increasing GFS flexibility (relaxed GFS's consistency model to vastly simplify the file system without imposing an onerous burden on the applications).
- Runs on clusters with 100,000s of commodity servers comprising:
  - Commodity hardware (known as off-shelf hardware, inexpensive hardware such as standard-issue PC or commodity hard disk RAID)
  - Application servers to write and read

GFS Goals

- Performance
- Scalability
  - The ease to add capacity to the system
- Reliability
  - Main concern of a search engine
- Availability
- Three different points in the design space
  - Component failure
  - Large files
  - File operation

Assumptions & Requirements

- Work for cheap servers, who come with failures:
  - Disk /network or server
  - OS bugs
  - Human errors
- Optimized for:
  - Storing and reading modest number of large files (expect few million) (100MB to multi GB).
  - Batch processing
  - High sustained bandwidth chosen over low latency
- File operations:
  - **Most files are mutated** by appending new data rather than overwriting existing data.
  - 2 kinds of reads – **large streaming read** (1MB), **small random reads** (batch and sort)

GFS Interface

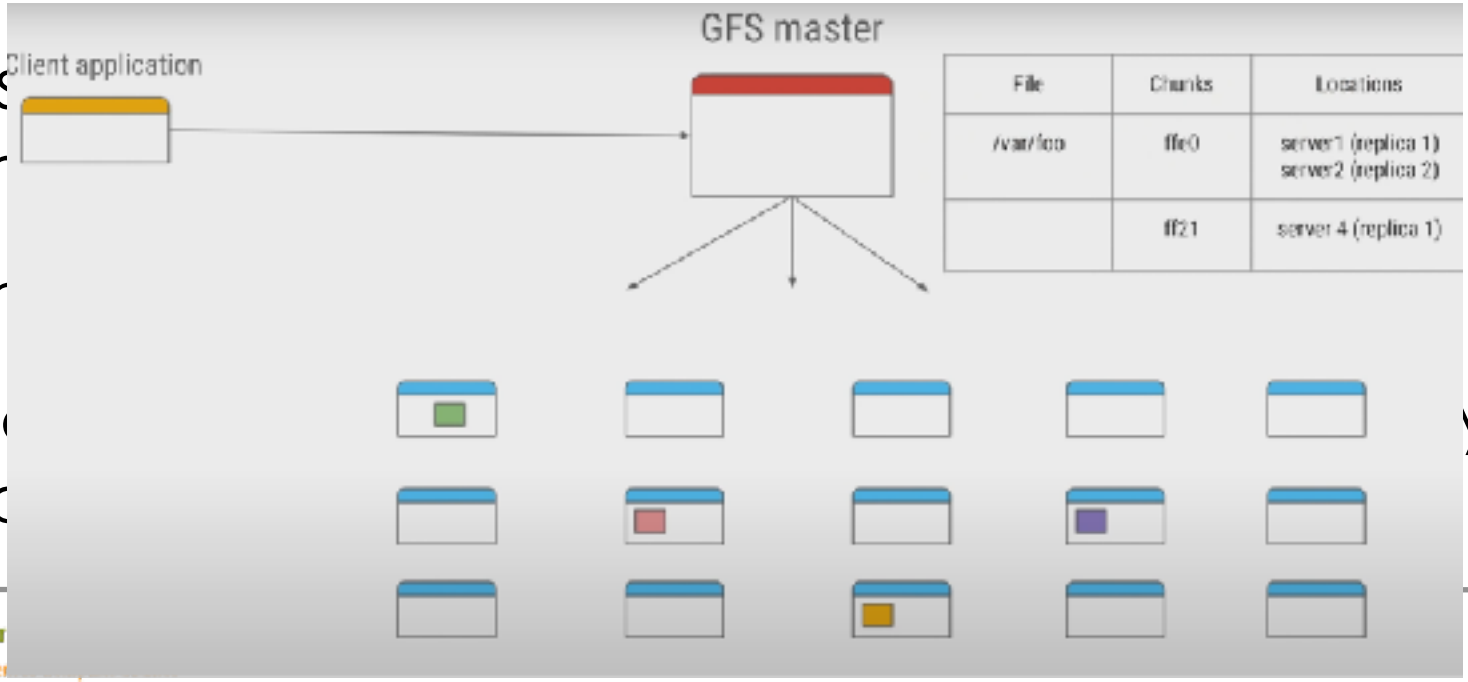
- Familiar **file system interface;** not an implementation of a standard API such as POSIX.
- Files organized **hierarchically** in directories and identified by path-names.
- Operations supported:
  - **Create**, **delete**, **open**, **close**, **read**, and **write** files.
  - **Snapshot** and record **append** operations:
    - Snapshot creates a copy of a file or a directory tree at low cost.
    - Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client's append.
    - Useful for implementing multi-way merge results and producer-consumer queues that many clients can simultaneously append to without additional locking.

GFS: The Google File System

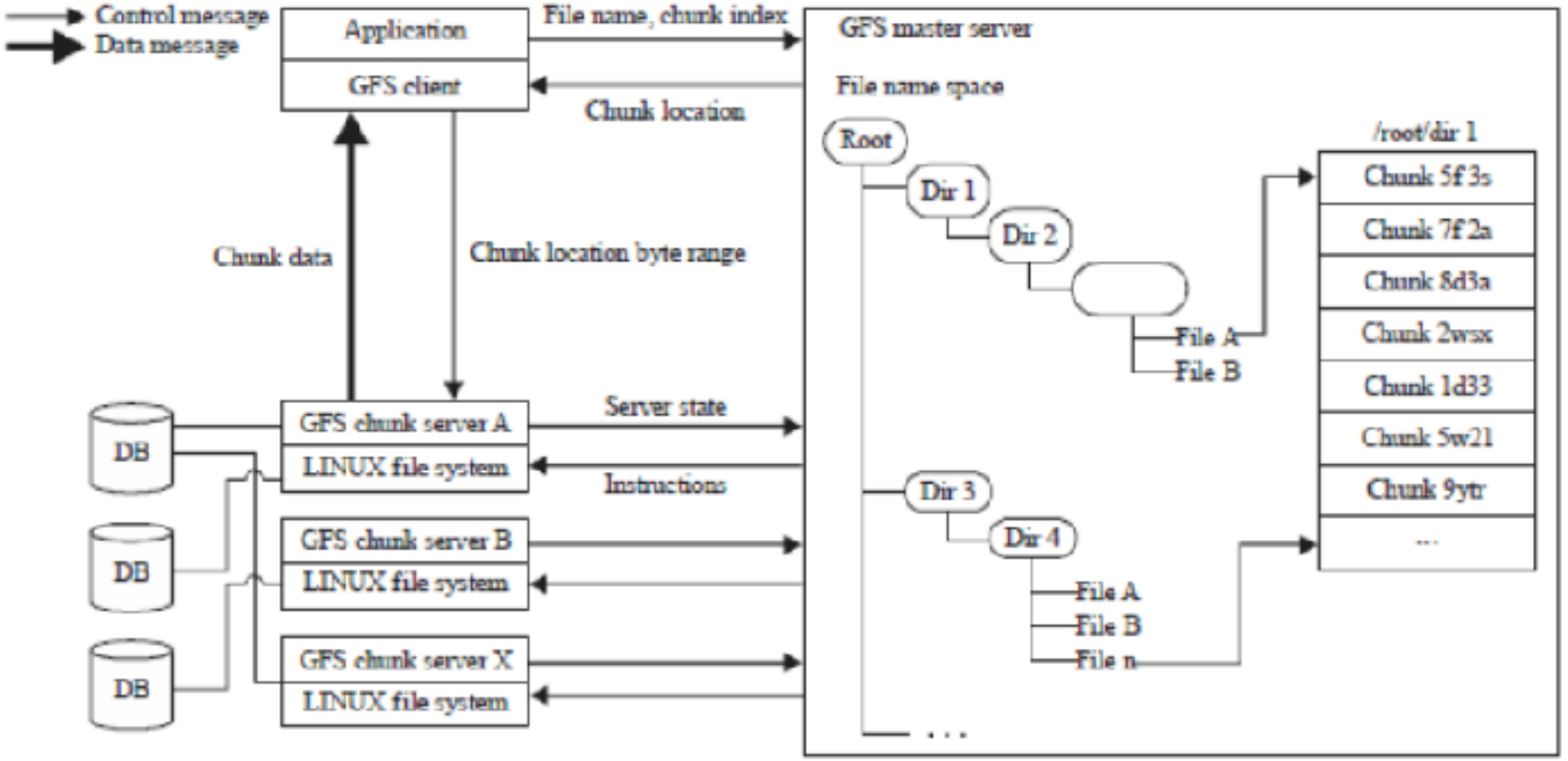
GFS Architecture & Operation

GFS Architecture

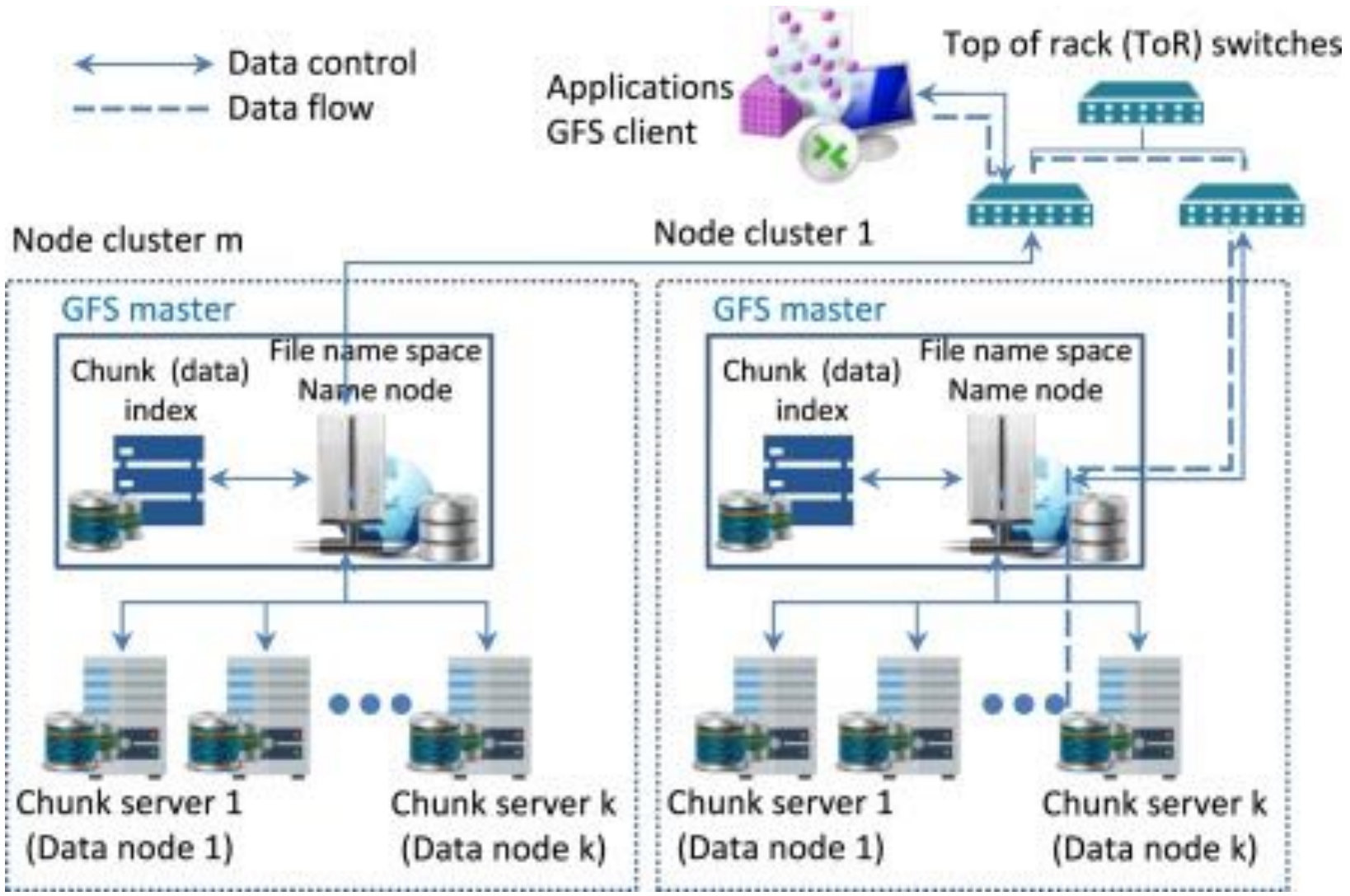
- A GFS cluster consists of a single master and multiple chunk-servers and is accessed by multiple clients.
- Each of these is typically a commodity Linux machine running a user-level server process.
- Eas n the sar
  - r
  - l y



GFS Architecture

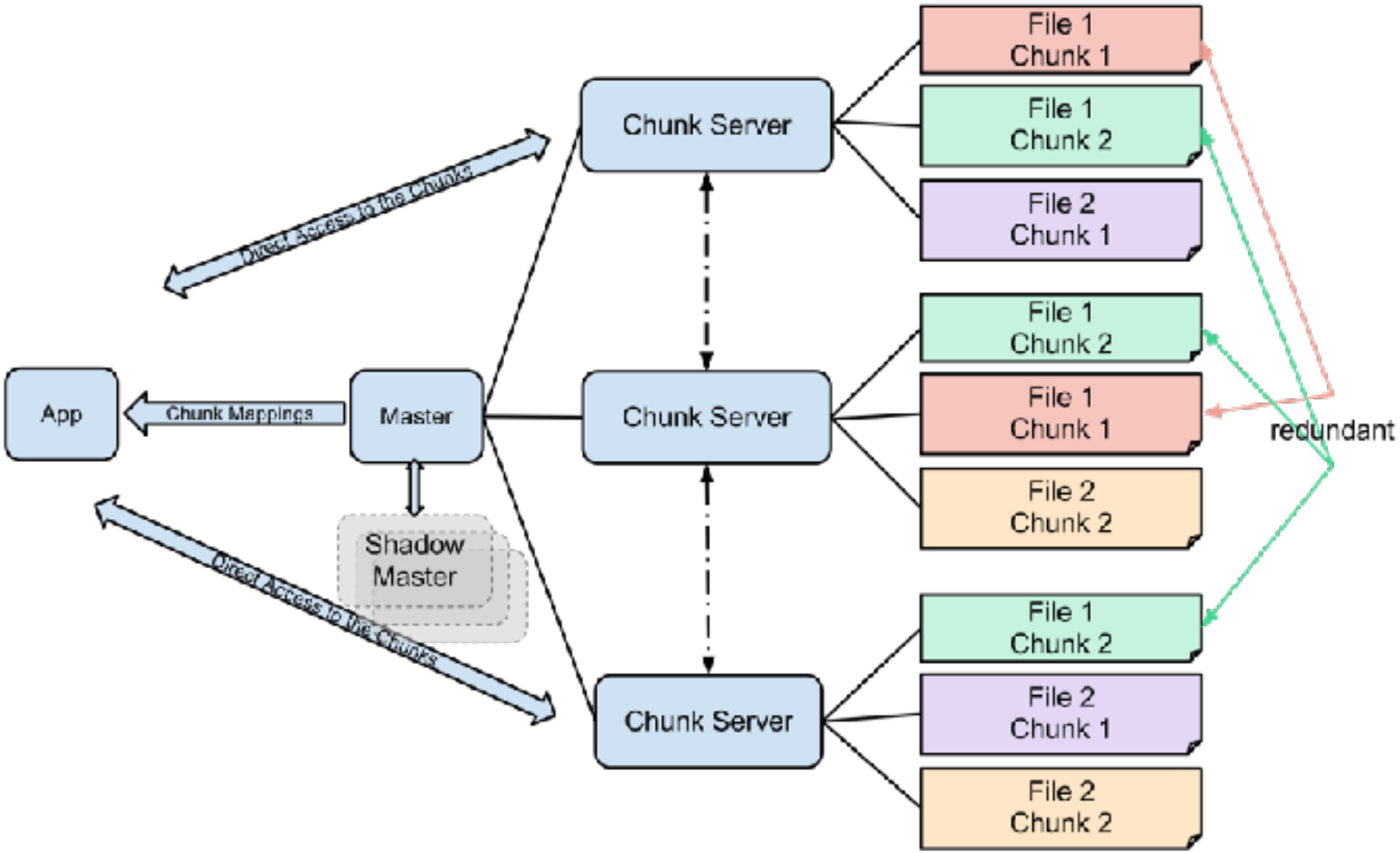


GFS Architecture



Chunks

- Files split into chunks:
  - Chunk size is 64MB - much larger than typical file system block size; large size reduces client's need to interact with master, overhead of interaction, size of metadata in memory.
  - Each chunk assigned at creation an immutable and globally unique 64 bit chunk handle (chunk ID).
  - Stored as plain Linux Files in chunk servers on local disks.
- Replicas
  - Ensure durability of data (if chunk server goes down)
  - Replica count by GFS client
  - Each chunk replicated on multiple chunk-servers – default is 3
- R or W chunk data specified by chunk handle and byte range


Direct Access to the Chunks
Chunk Server
File 1
Chunk 1
File 1
Chunk 2
File 2
Chunk 1
App
Chunk Mappings
Master
Chunk Server
File 1
Chunk 2
File 1
Chunk 1
File 2
Chunk 2
redundant
Shadow
Master
Direct Access to the Chunks
Chunk Server
File 1
Chunk 2
File 2
Chunk 1
File 2
Chunk 2

Master Node

- Responsible for all system-wide activities: managing chunk leases, reclaiming storage space, load-balancing, consistency protocol
- Master maintains all file system metadata
- Controls garbage collection of chunks
- Communicates with each chunk-server through HeartBeat messages:
  - this let's master determines chunk locations and assesses state of the overall system
- Clients interact with master for metadata
  - chunk-severs do the rest, e.g. R/W on behalf of applications
- No caching
- Important: The chunkserver has the final word over what chunks it does or does not have on its own disks – not the master

Client Read (1)

1. **Client** calculates address:
   - Translates file name/byte offset specified by application into a chunk index within the file.
2. **Client** sends the master:
   - Request with file name/chunk index.
3. **Master's** reply with chunk handle/locations of replicas.
4. **Client** caches received information
   - Uses file name/chunk index as key.
5. **Client** sends request to "closest" chunk server with replicas:
   - The request specifies the chunk handle/byte range within that chunk.
   - "Closeness" determined by IP address on simple rack-based topology.
6. **Chunkserver** replies with data

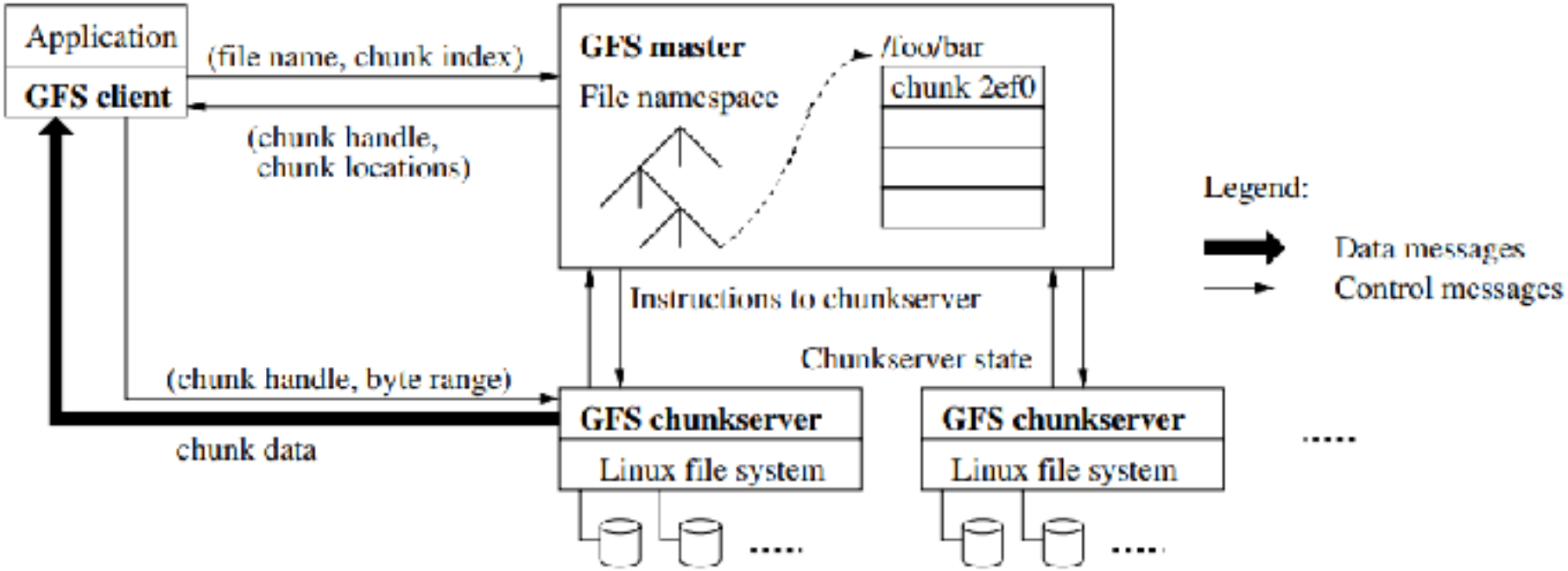


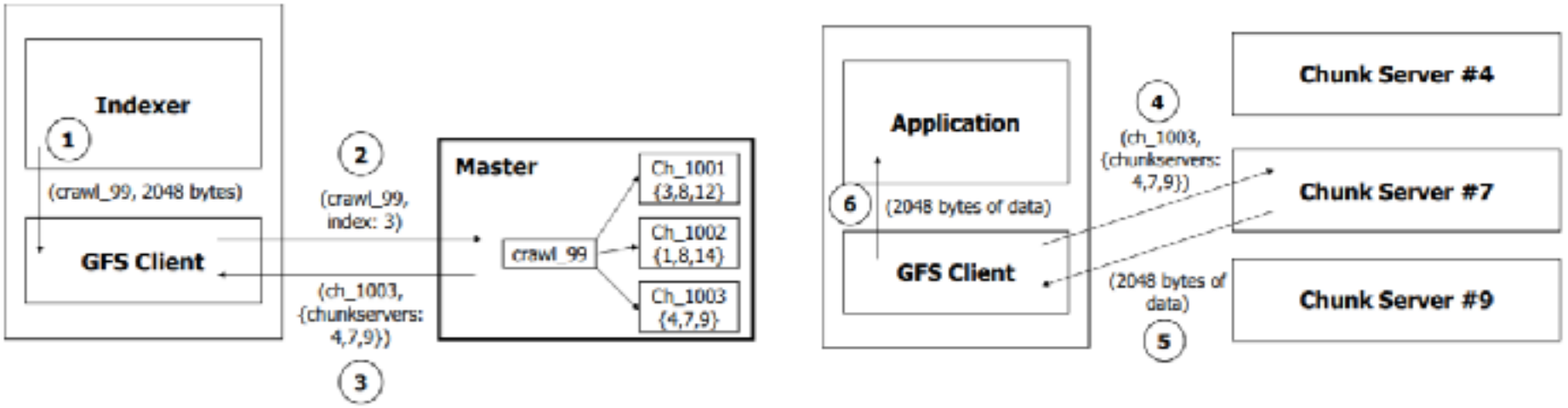
Figure 1: GFS Architecture

Client Read (2)

- Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened.
- Clients typically ask for multiple chunks in the same request.
- Master can also include the information for chunks immediately following those requested.
- This extra information sidesteps several future client-master interactions at practically no extra cost.

Chunk index calculation

- Calculating chunk index from byte range:
- (Assumption: File position is 201,359,161 bytes)
- Chunk size = 64 MB = 1024 *1024 * 64 bytes = 67,108,864 bytes.
- 201,359,161 bytes = 67,108,864 * 2 + 32,569 bytes.
- So, client translates 2048 byte range -> chunk index 3.

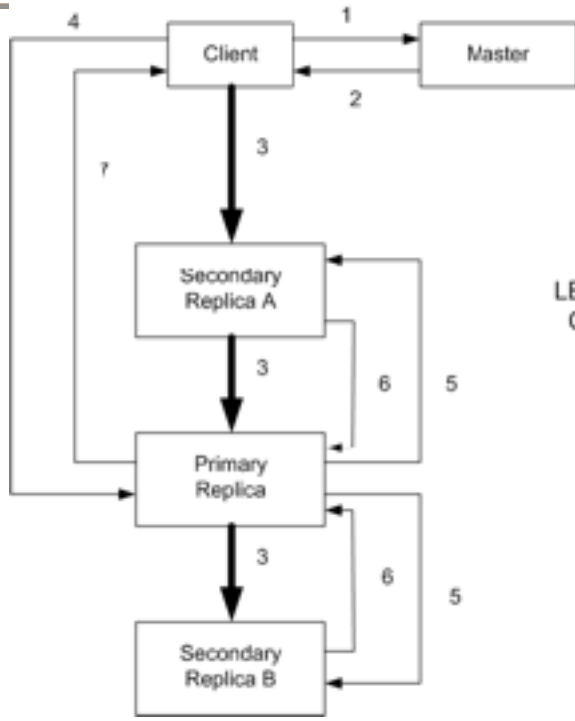


Client Write (synopsis)

- Client asks master for chunkserver with lease
- Ask for location to write
- Get replication locations
- Write data to closest replica
- Request commit to primary
- Primary instructs order of writes to secondaries
- Secondaries acknowledge
- Primary ack to client

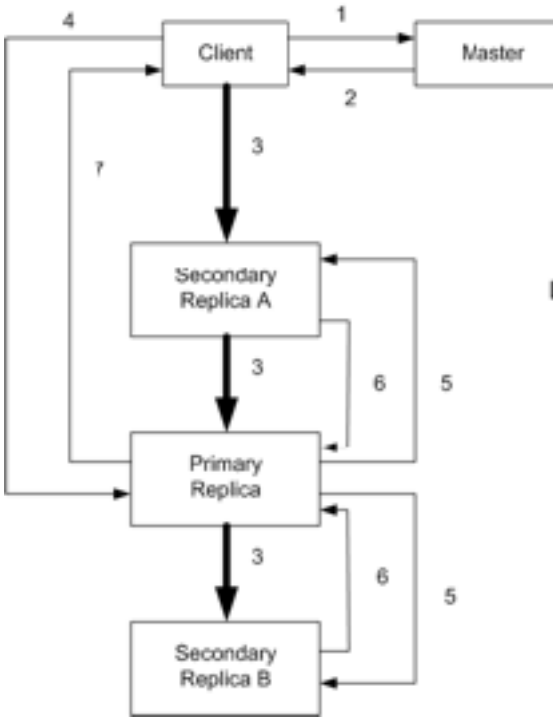
Client Write (1)

1. **Client asks the master** which chunkserver holds the current lease for the chunk and the locations of the other replicas.
   - Some chunkserver is **primary** for each chunk
   - Master grants lease to primary (typically for 60 sec.)
   - Leases renewed using periodic heartbeat messages between master and chunkservers
   - If no one has a lease, the master grants one to a replica it chooses.
2. **Master replies** with the identity of the primary and the locations of the other (secondary) replicas.
   - Client caches this data for future mutations.
   - Client needs to contact the master again only when the primary becomes unreachable or replies that it no longer holds a lease.
3. The client **pushes the data** to **all replicas**.
   - A client can do so in any order. Each chunkserver will store the data in an internal LRU buffer cache until the data is used or aged out.
   - By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which chunkserver is the primary.



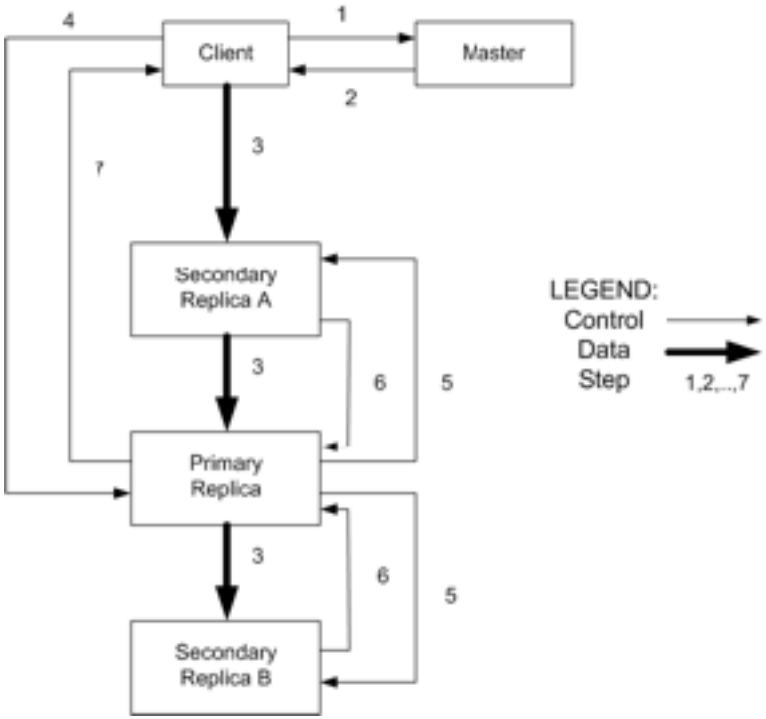
Client Write (2)

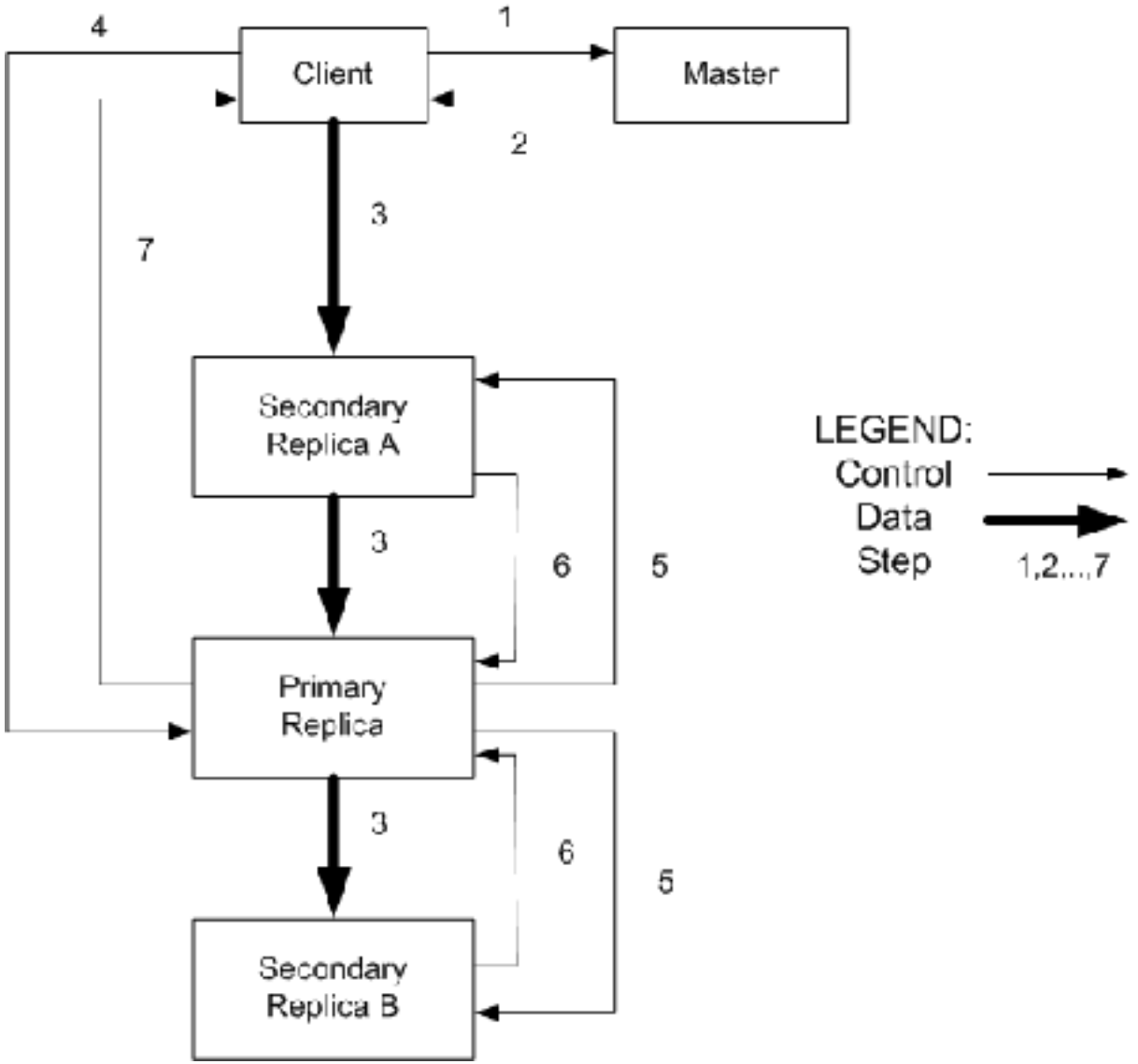
4. Once all the replicas have acknowledged receiving the data, the **client sends a write request** to the **primary**.
   - The request identifies the data pushed earlier to all of the replicas.
   - The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides the necessary serialization.
   - It applies the mutation to its own local state in serial number order.
5. The **primary forwards the write request** to all **secondary replicas**.
   - Each secondary replica applies mutations in the same serial number order assigned by the primary.
6. The **secondaries all reply** to the primary indicating that they have completed the operation.



Client Write (3)

7. The **primary replies to the client**.
   - Any errors encountered at any of the replicas are reported to the client.
   - In case of errors, the write may have succeeded at the primary and an arbitrary subset of the secondary replicas.
   - The client request is considered to have failed, and the modified region is left in an *inconsistent state*.
   - Client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the beginning of the write.




4
1
Client
Master
2
3
7
Secondary
Replica A
3
6
5
Primary
Replica
3
6
5
Secondary
Replica B
LEGEND:
Control
Data
Step
1,2...,7

Client Write: Record Append

- In a traditional write, the client specifies the offset at which data is to be written.
- Concurrent writes to the same region are **not serializable**: the region may end up containing data fragments from multiple clients.
- **Record append - what is it?** An atomic append operation offered by GFS:
  - Client specifies only the data it wants to write.
  - GFS appends it to the file at least once atomically (i.e., as one continuous sequence of bytes) at an offset of GFS's choosing and returns that offset to the client.
- Record append is heavily used in Google's distributed applications, in which many clients on different machines append to the same file concurrently:
  - Such files often serve as multiple-producer/single-consumer queues or contain merged results from many different clients.
- Without record append, if using traditional writes, clients would need additional complicated and expensive synchronization, for example through a distributed lock manager in such workload scenarios.

Record Append (2)

- The client pushes the data to all replicas of the last chunk of the file.
- Then, it sends its request to the primary.
- The primary checks to see if appending the record to the current chunk would cause the chunk to exceed the maximum size (64 MB).
  - If so, it pads the chunk to the maximum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk.
  - If the record fits within the maximum size, which is the common case, the primary appends the data to its replica, tells the secondaries to write the data at the exact offset where it has, and finally replies success to the client.

Record Append (3)

- If a record append **fails at any replica**, the client retries the
- operation.
  - As a result, replicas of the same chunk may contain different data possibly including duplicates of the same record in whole or in part.
- GFS does not guarantee that all replicas are bytewise identical. It only guarantees that the data is written at least once as an atomic unit. Note that:
  - For the operation to report success, the data must have been written at the same offset on all replicas of some chunk.
  - After this, all replicas are at least as long as the end of record and any future record will be assigned a higher offset or a different chunk.
- In terms of consistency guarantees, the regions in which successful record append operations have written their data are defined (hence consistent), whereas intervening regions are inconsistent (hence undefined).

Metadata

- Master stores three types of metadata stored in main memory for fast access:
  - File and chunk namespaces
  - Mapping from file to chunks
  - Locations of the replicants
- Namespaces and Mapping info are kept persistent by logging mutations to operation log stored on master's disk and replicated.
- In memory-data structures **scanned** periodically to implement:
  - Garbage collection
  - Re-replication if chunkserver failure
  - Chunk migration for better load balancing and disk space allocation across chunkservers
- File namespace < 64B for each chunk (64MB) – not serious problem to add more memory to Master

Operation Log

- Central to GFS:
  - Contains historical record of critical metadata changes.
  - The only persistent record of metadata.
  - Serves as a logical time line that defines the order of concurrent operations: Files and chunks, as well as their versions are uniquely and eternally identified by the logical times at which they were created.
- Must be stored reliably; its changes are not visible to clients until metadata changes are made persistent:
  - Replicated on multiple remote machines and respond to a client operation only after flushing the corresponding log record to disk both locally and remotely.
  - Master batches several log records together before flushing for performance.
- Master recovers its file system state by replaying the operation log.
- Log kept small to minimize startup time: master checkpoints its state whenever the log grows beyond a certain size so that it can recover by loading the latest checkpoint from local disk and replaying only the

Consistency Model

- GFS has a relaxed consistency model that supports highly distributed applications well but remains relatively simple and efficient to implement.
- The consistency protocol determines how **mutations** are handled. There are two kinds of mutations:
- File namespace mutations (e.g., file creation):
  - Atomic.
  - Handled exclusively by the master: namespace locking guarantees atomicity and correctness.
  - Master's operation log defines a global total order of these operations.
- Data mutations may be: **writes** or **record appends:**
  - A write causes data to be written at an application-specified file offset.
  - A record append causes data (the "record") to be appended atomically **at least once** even in the presence of concurrent mutations, but at an offset of GFS's choosing
  - A regular append is merely a write at an offset that the client believes to be the current end of file.

Consistency w/ Data Mutations

- The **state** of a **file region** after a data mutation depends on:
  - the type of mutation
  - whether it succeeds or fails, and
  - whether there are concurrent mutations.

States after Mutations

Possible file-region states after mutation(s):

- Consistent: all clients always see the same data, regardless of which replicas they read from.
- Defined (and by implication consistent): when the mutation **succeeds without interference** from concurrent writers, and clients see what the mutation writes in its entirety.
- Undefined but consistent: after concurrent successful mutations where all **clients see the same data**, but it may not reflect what **any one mutation has written**.
  - Typically, consists of mingled fragments from multiple mutations.
- Inconsistent (and undefined): after a **failed mutation,** where different clients may see different data at different times.

Consistency w/ Record Appends

- A record append causes the record to be appended atomically at least once even in the presence of concurrent mutations, but at an offset of GFS's choosing.
- The offset marks the beginning of a defined region that contains the record.
  - In addition, GFS may insert padding or record duplicates in between, which occupy regions considered to be inconsistent and are typically dwarfed by the amount of user data.
- After a sequence of successful mutations, the mutated file region is guaranteed to be defined and contain the data written by the last mutation.
- GFS achieves this by:
  - Applying mutations to a chunk in the same order on all its replicas,
  - Using chunk version numbers to detect any replica that has become stale because it has missed mutations while its chunkserver was down.
  - Stale replicas will never be involved in a mutation or given to clients asking the master for chunk locations. They are garbage collected at the earliest opportunity.

Namespace Management & Locks

- Master ops can take time, e.g. revoking leases – allow multiple ops at same time, use locks over regions for serialization.
- Each absolute file name or absolute directory name has an associated **read-write lock**
- GFS does not have per directory data structure listing all files – Instead lookup table mapping full pathnames to metadata
- Each name in tree has R/W lock
- If accessing: /d1/d2/ ../dn/leaf, **R lock** on /d1, /d1/d2, etc., **W lock** on /d1/d2 …/leaf
- Locks are required to prevent deadlock:
  - First ordered by level in the namespace tree
  - Lexicographically ordered within the same level

Shadow Master

- Master Replicated for reliability
  - One master remains in charge of all mutations and background activities
- If fails, start instantly
  - If machine or disk fails, monitor outside GFS starts new master with replicated log
  - Clients only use canonical name of master
- Shadow master read replica of operation log, applies same sequence of changes to data structures as the primary does
  - Polls chunk-server at startup, monitors their status, etc.
  - Depends only on primary for replica location updates

Replica Placement

- GFS cluster distributed across many machine racks
- Need communication across several network switches: Challenge to distribute data
- Chunk replica – Maximize data reliability
- Maximize network bandwidth utilization: Spread replicas across racks (survive even if entire rack offline)
- R can exploit aggregate bandwidth of multiple racks
- W traffic has to go through multiple racks

Garbage Collection

- The system has a unique approach for this. Once a file is deleted its resources are not reclaimed immediately instead they are renamed with hidden namespace. Such files are removed if they exist for 3 days during the regular scan.
- The advantages offered by it are:
  - Simplicity
  - Deleting of files can take place during master's idle periods
  - Safety against accidental deletion

Summary

- Support large-scale data processing workload
- Component failures as the norm rather than the exception
- Optimize for huge files mostly append to and then read sequentially
- Fault tolerance by constant monitoring, replicating crucial data and fast automatic recovery(+checksum to detect data corruptions)
- Delivers high aggregate throughput to many concurrent readers and writers

Cloud Storage

The Hadoop File System

Goal

"Hadoop provides a distributed file system
and a
framework for the analysis and
transformation of very large data sets"

Hadoop Project Components

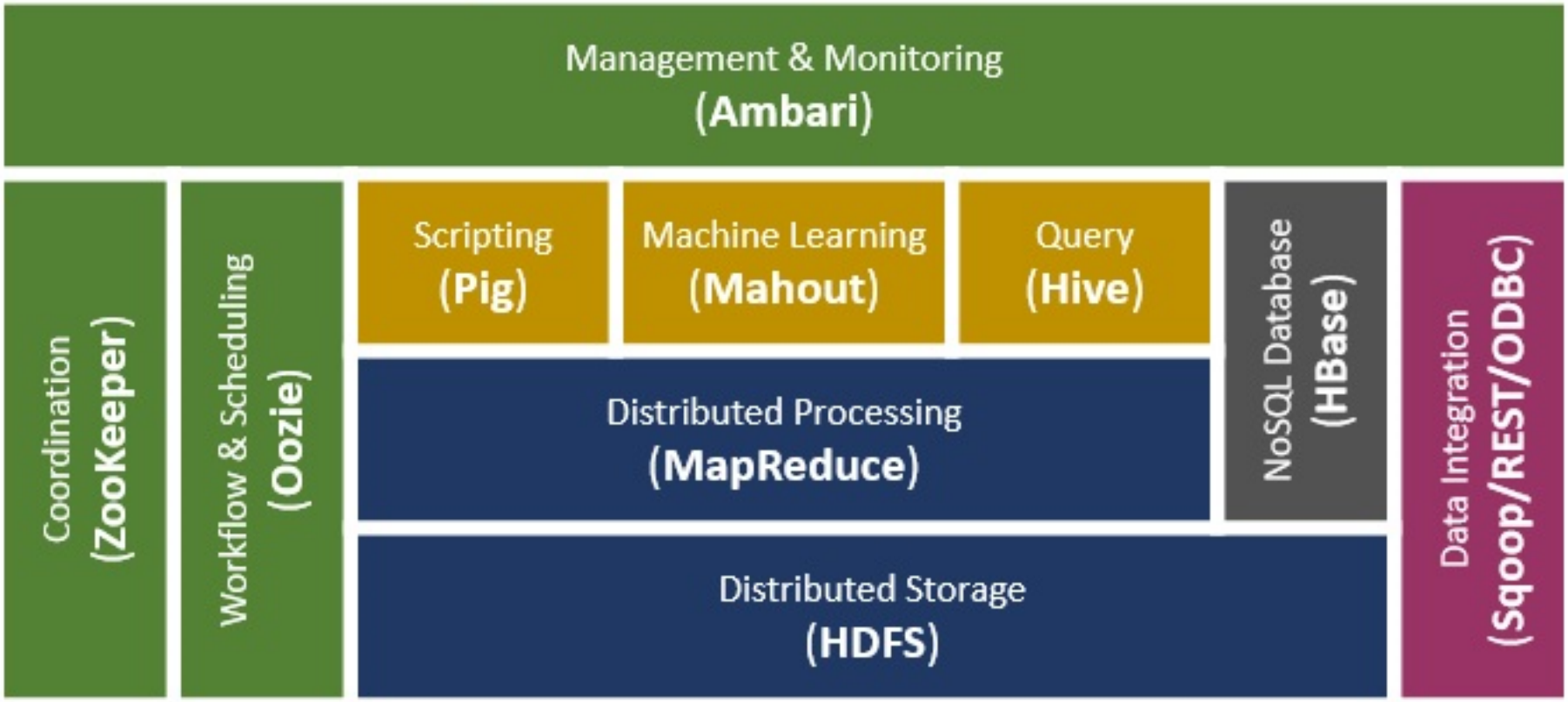
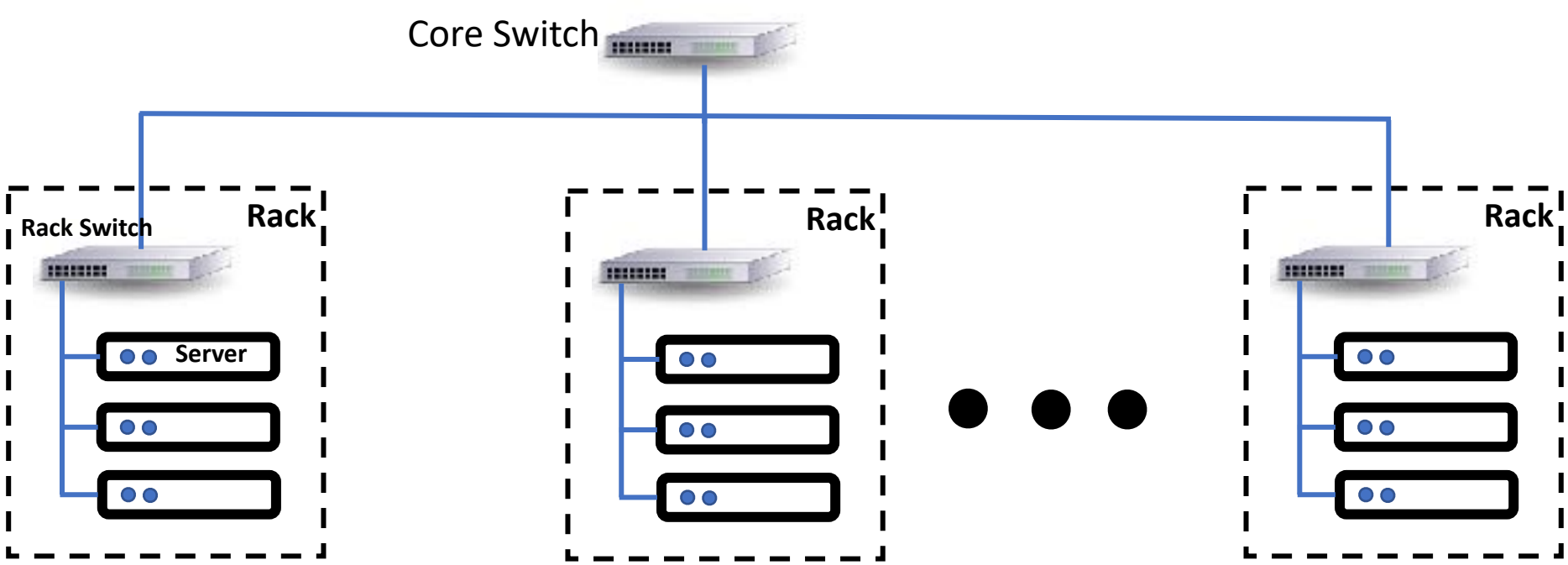
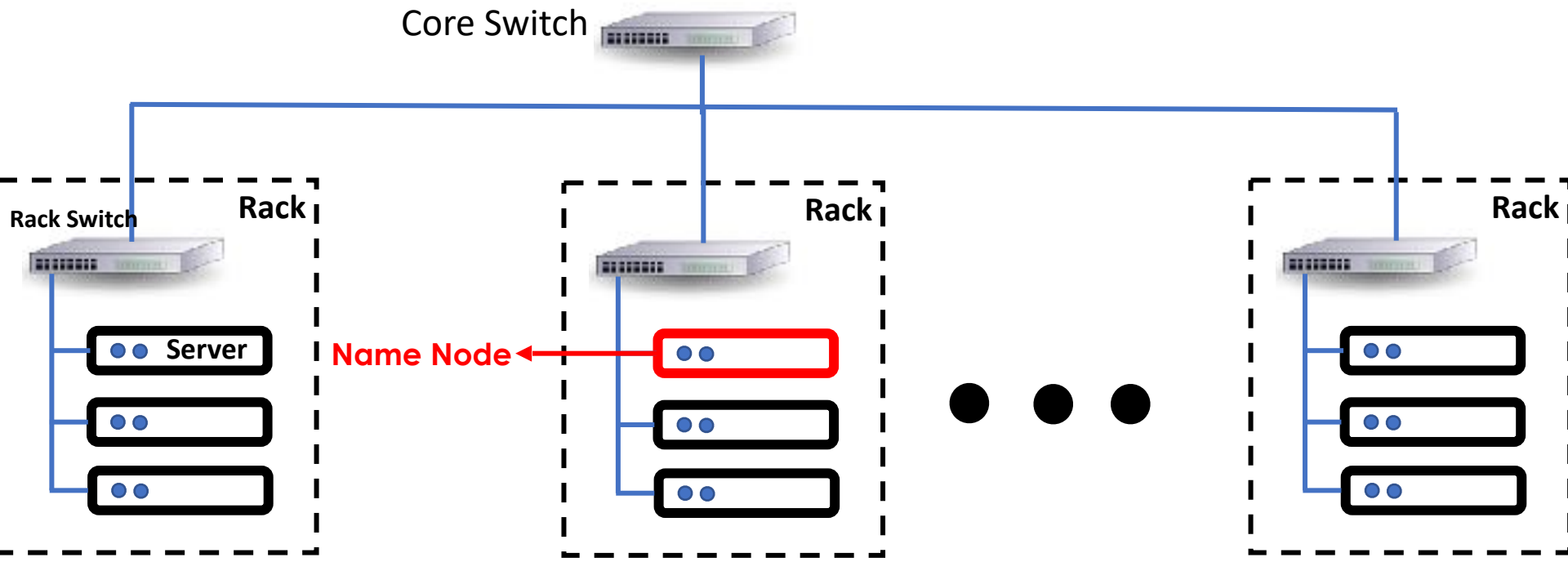


Image Credit: mssqltips.com

HDFS Architecture Overview

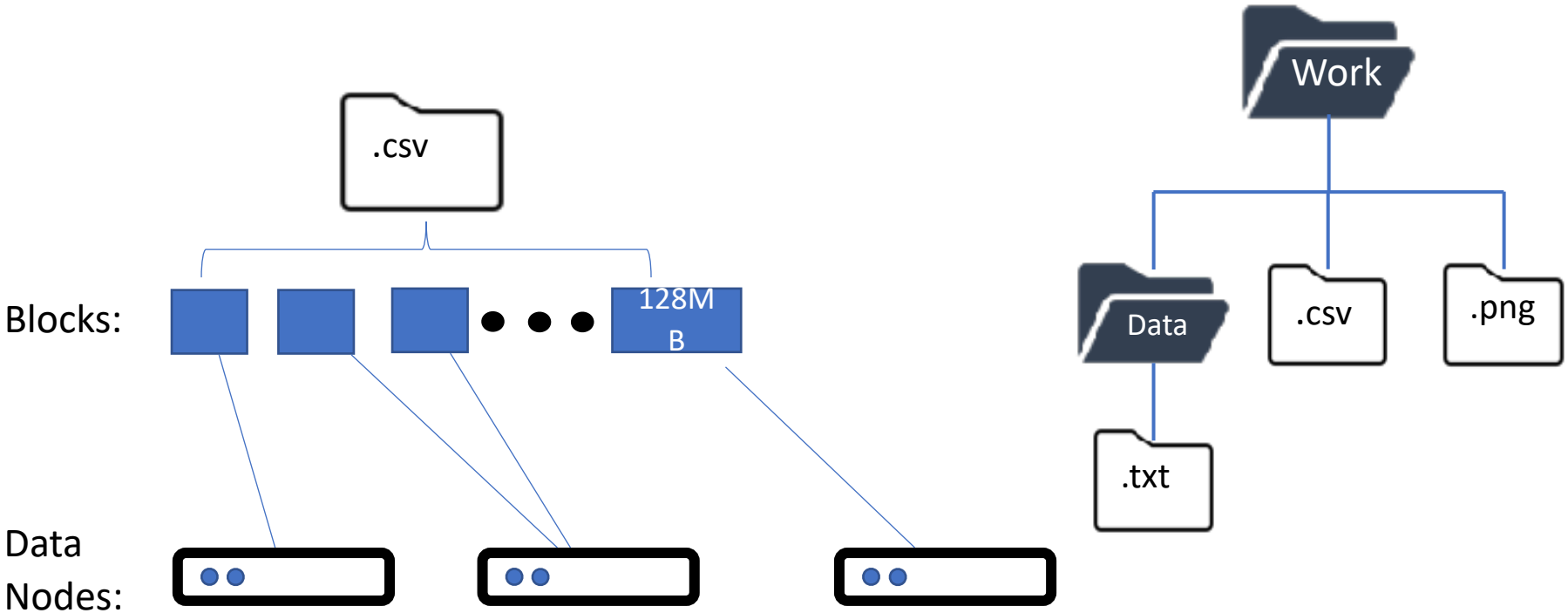


HDFS Architecture Overview



NameNode

- NameNode maintains the name space tree and the mapping of file blocks to DataNodes
- HDFS Keeps the entire namespace in RAM



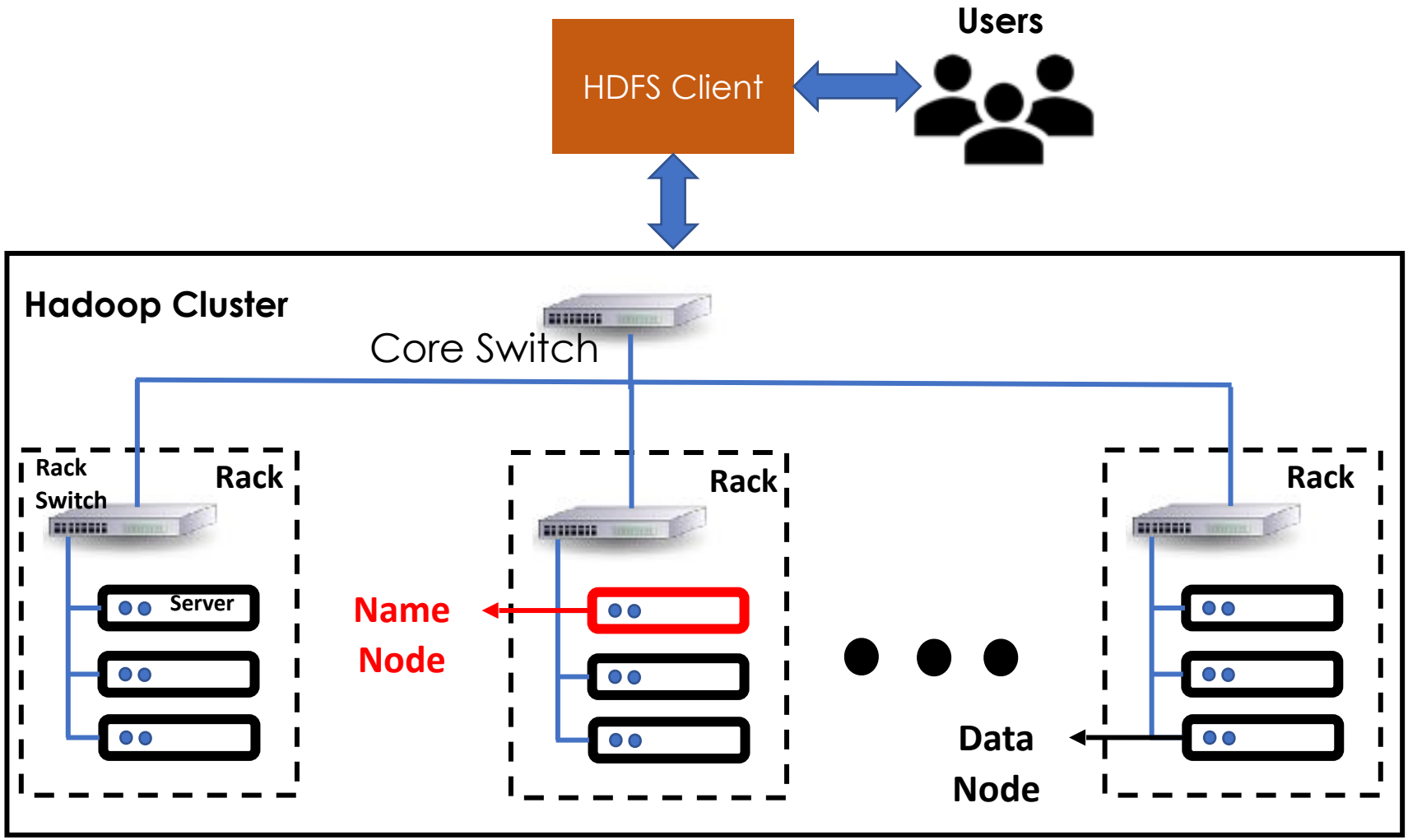
DataNode

- Stores the actual data
- Each block contains 2 files:
  - First: Contains the actual data
  - Second: Contains block's metadata (e.g. checksum)
- The size of the data file equals the block size
  - No need for round up like traditional file systems
- On start up DataNode register to the NameNode and gets a storage ID

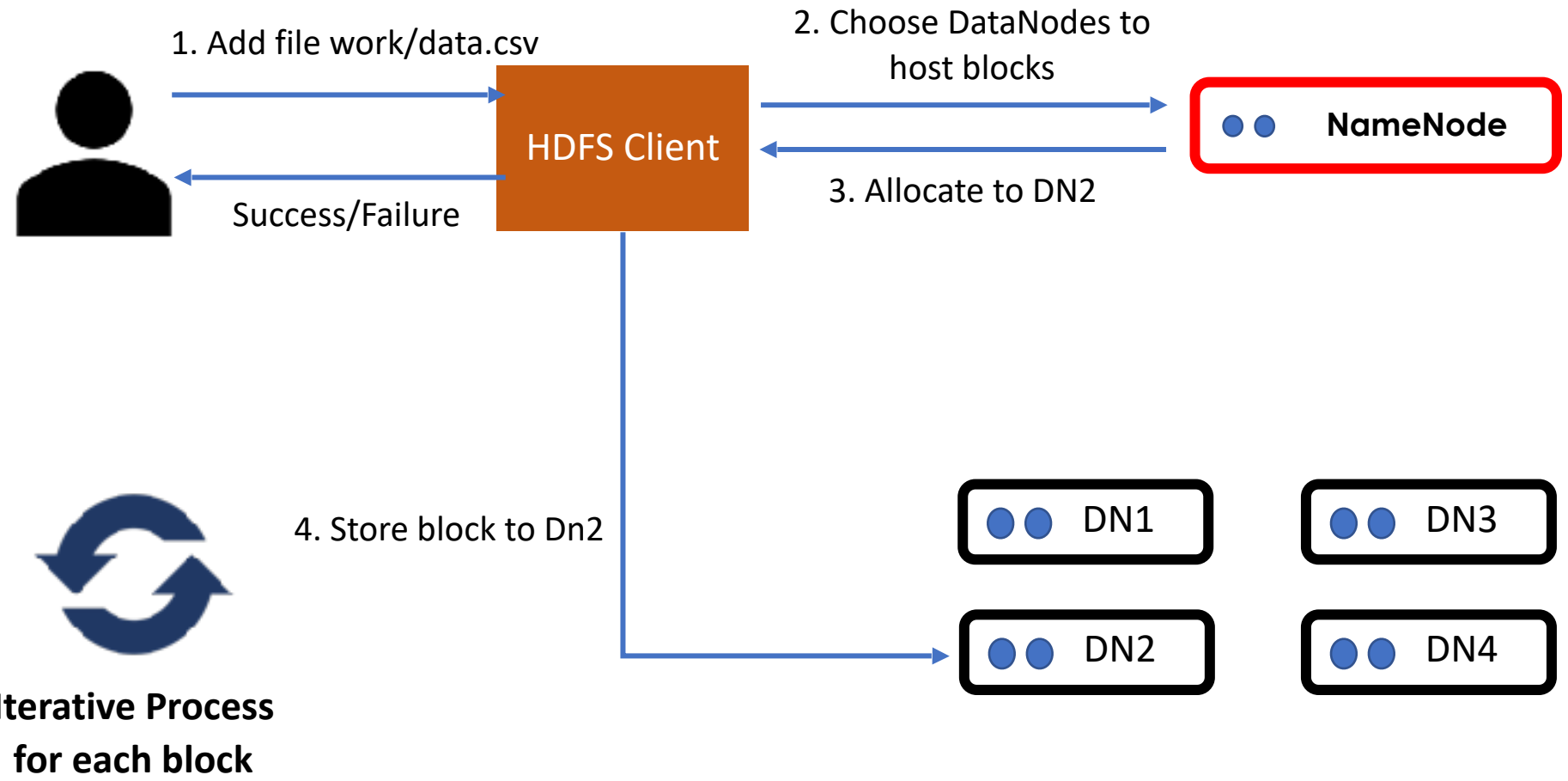
DataNode

- **Heartbeat**: Each DataNode sends periodically a block report to the NameNode
- **Block Report**: < BlockID, Generation Stamp, Length of each block>
- **Heartbeat Interval**: 3 seconds
- If a NameNode does not receive a heartbeat for more than 10s then considers the specific DataNode as out of service.
- Additional Information:
  - Total storage capacity,
  - Fraction of storage in use,
  - Number of data transfers

HDFS Client


Users
HDFS Client
Hadoop Cluster
Core Switch
Rack
Rack Switch
Server
Name Node
Rack
Rack
Data Node

HDFS Write File Example



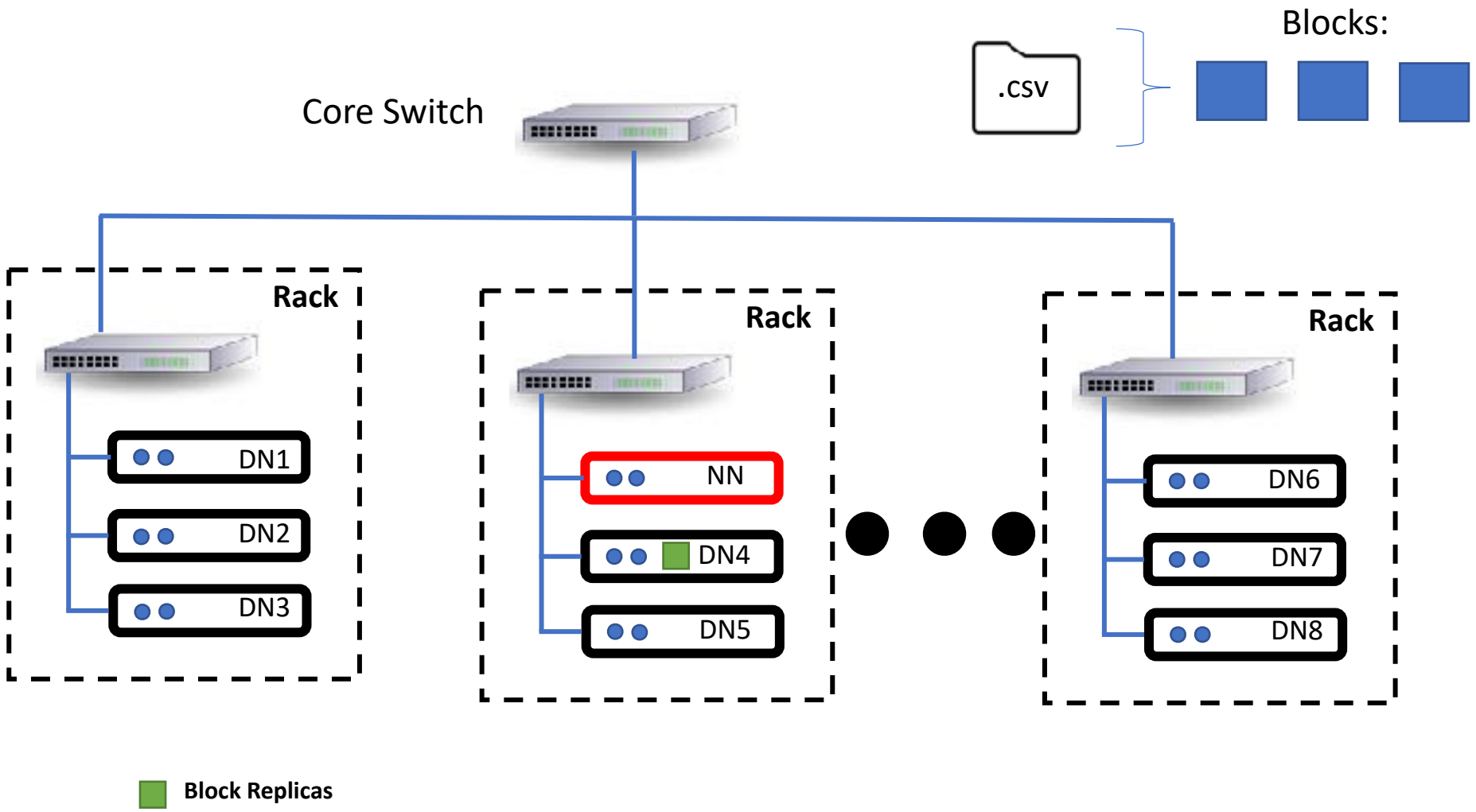
HDFS File Read/Write

- HDFS implements a single-writer, multiple-reader model
- A clients granted a lease in order to write a file
- Lease Duration:
- Soft Limit: If expires another client can ask for lease
- Hard Limit: Close file and deallocate lease
- After data are written to an HDFS file, HDFS does not provide any guarantee that data are visible to a new reader until the file is closed
- Client can call hflush operation to guarantee visible changes

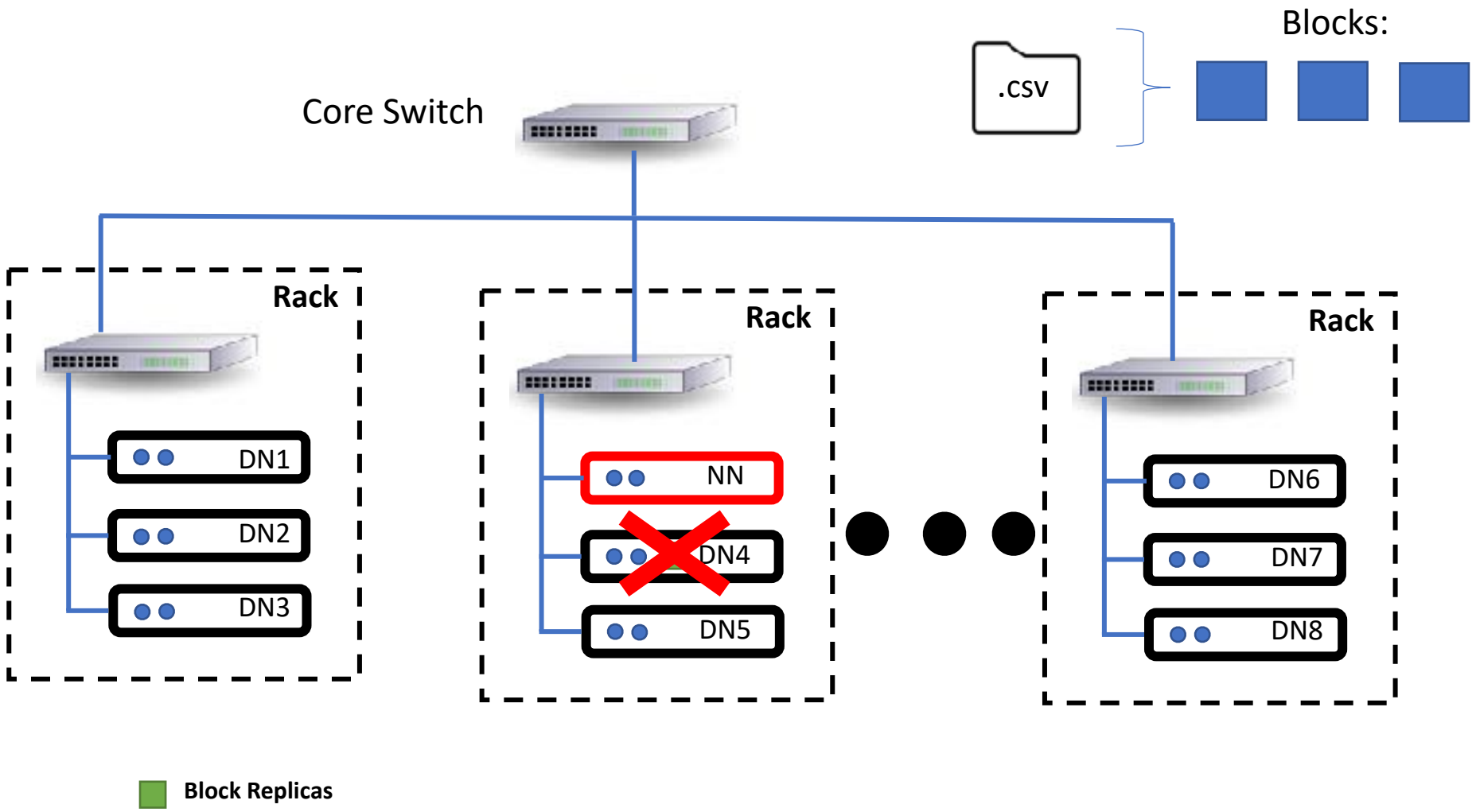
HDFS File Read/Write (ctd.)

- The location of each replica block is ordered by the distance from the client.
- When reading the content of a block, the client tries to closest replica first.
- HDFS I/O is particularly optimized for batch processing systems (e.g. MapReduce)

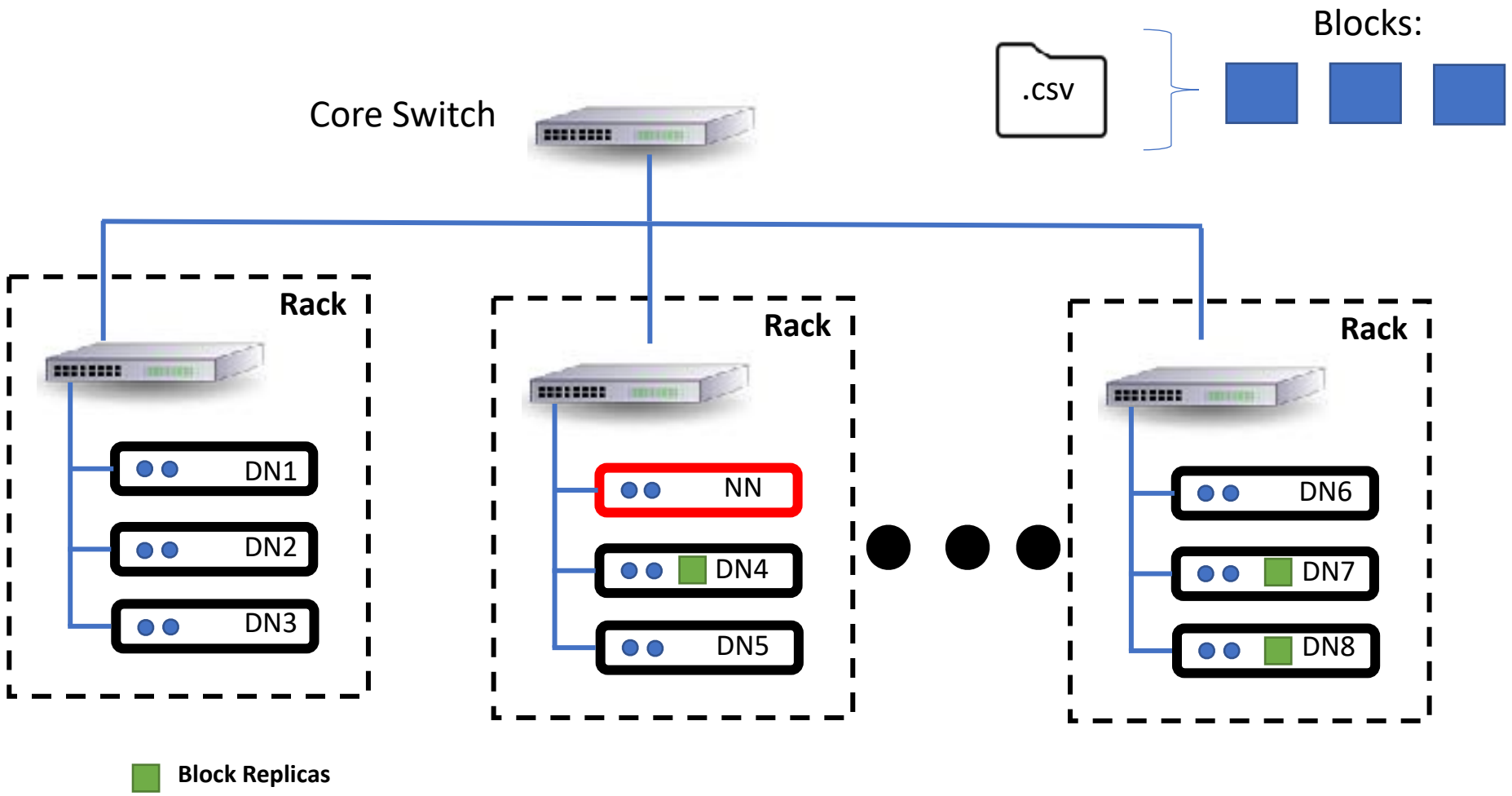
HDFS Architecture Overview



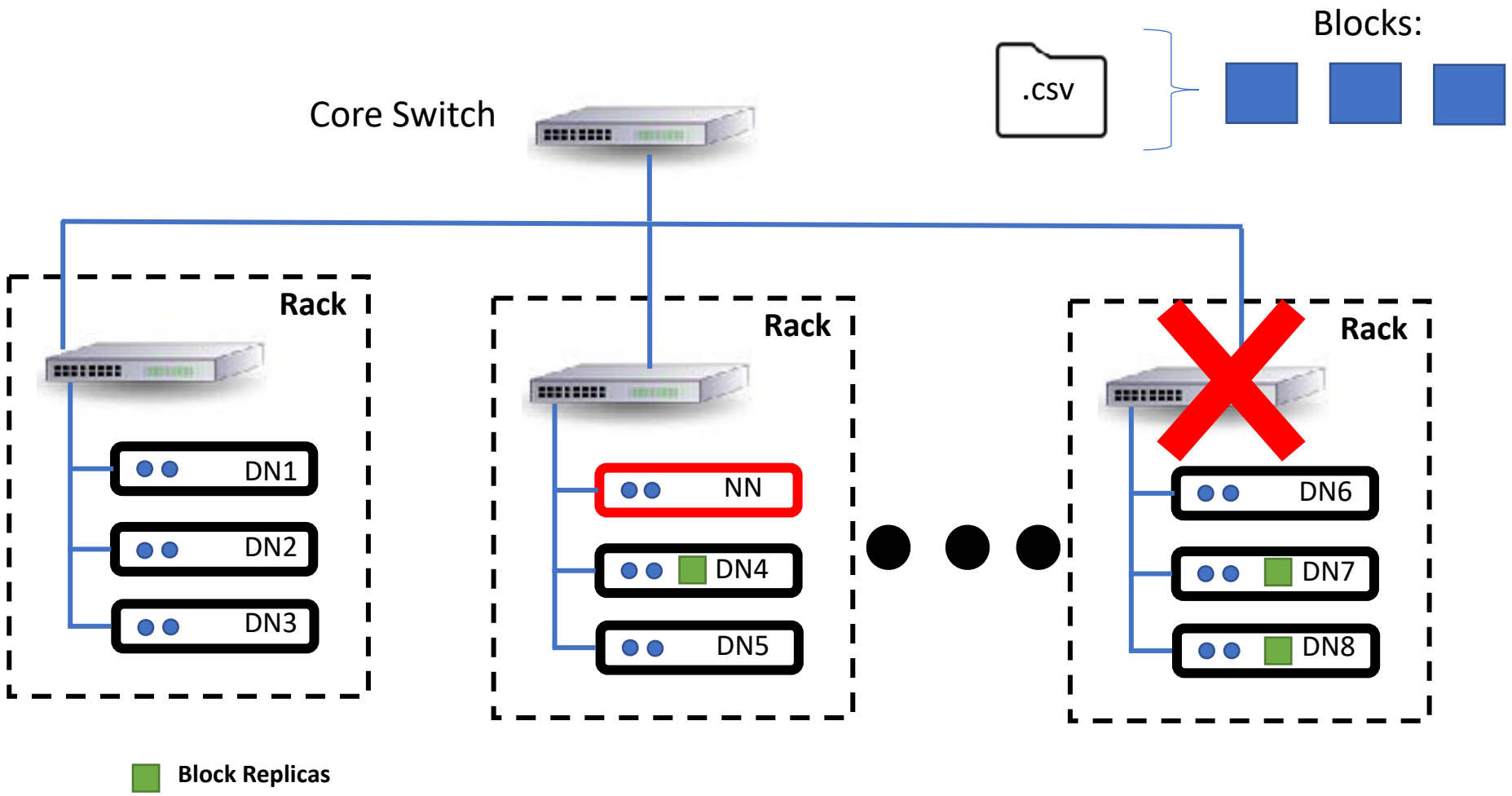
HDFS Architecture Overview



HDFS Architecture Overview



HDFS Architecture Overview



HDFS Architecture Overview

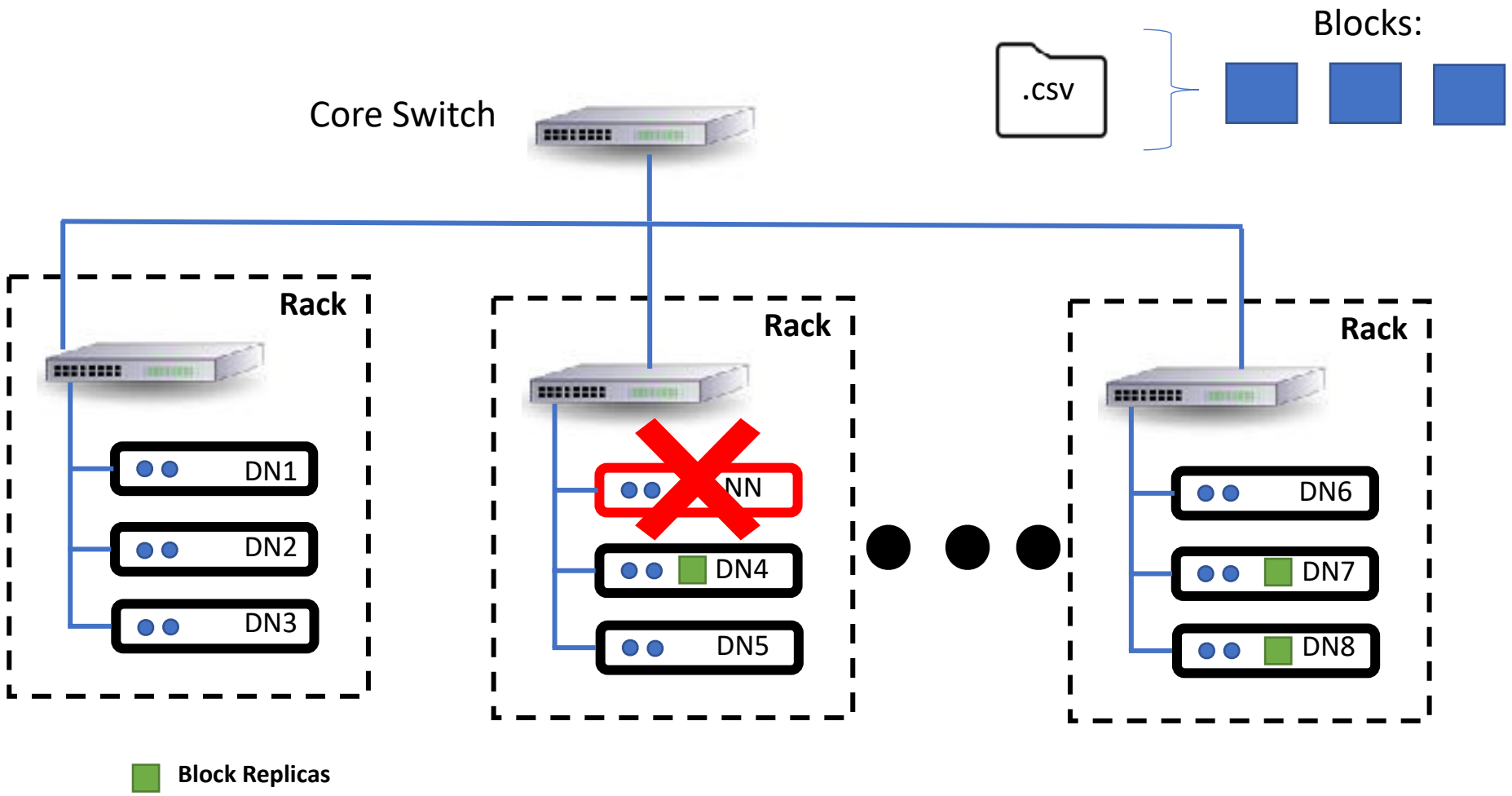


Image and Journal

- **Image**: An image of the whole namespace
- **Journal**: A log file which stores every transaction
- A persistent record of the image on the disk called checkpoint!
- **The storage of the checkpoint and the journal is crucial!**
  - Possible failure of the NameNode and the checkpoint or the journal may lead to partly or entirely lost of data
- A common practice is the storage the files to different volumes and one of them should be in a remote NFS server

Checkpoint Node

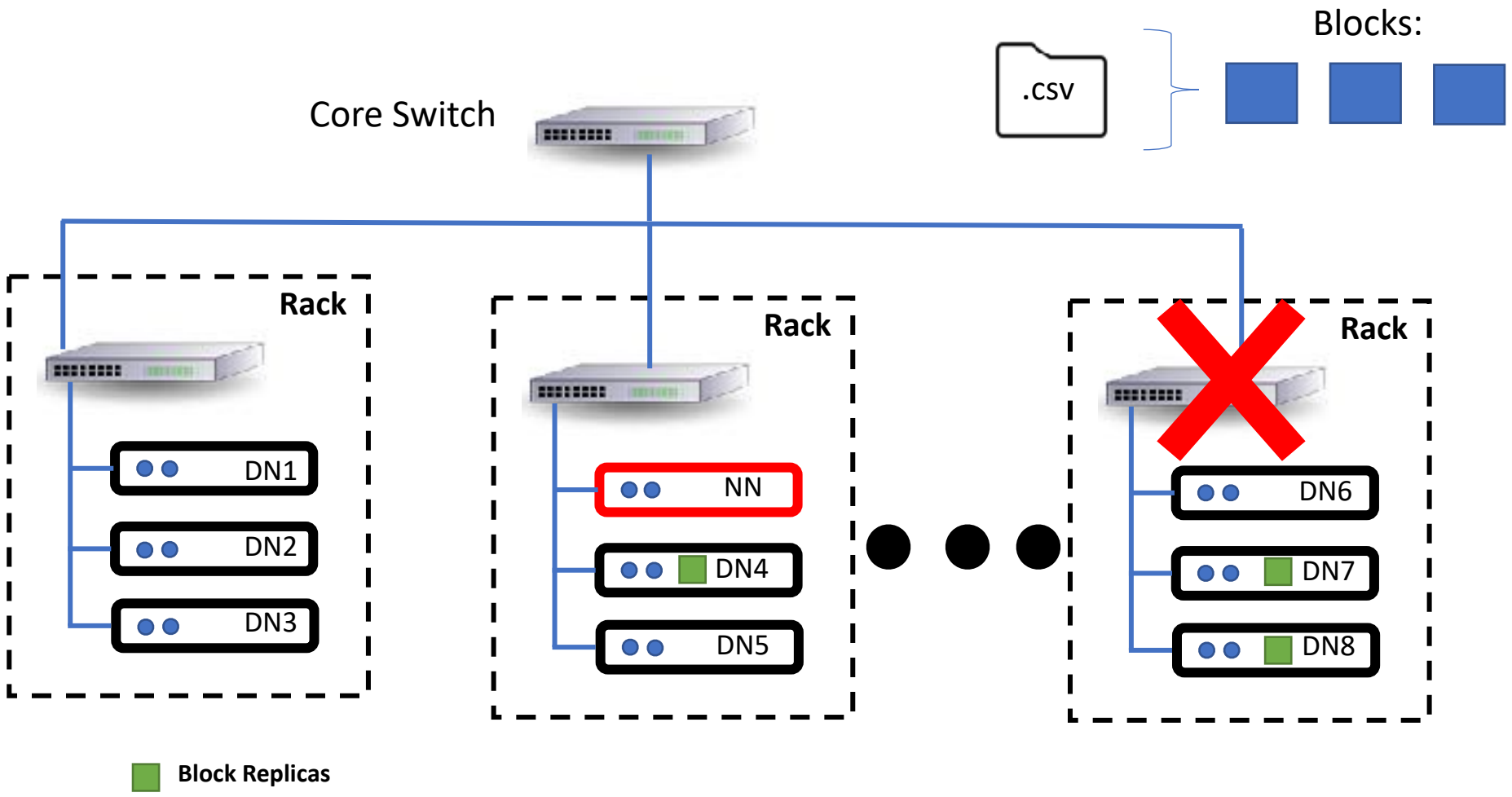
- The CheckpointNode periodically combines the existing checkpoint and journal to create a new checkpoint and an empty journal.
- Image + Journal = new checkpoint, then empty Journal
- Create a new checkpoint every 1 hour to keep the journal file small enough.

Backup Node

- Same as Checkpoint Node, with an in-memory image of the file system namespace
- Synchronized with NameNode
- Includes all the information of the NameNode except block location
- Read-only NameNode
- Help us run the NameNode without persistent storage

Other Functionalities

- Replication Manager: Detects whether a block becomes under-replicated or over-replicated.
  - NameNode is responsible for the operation of Replication Manager.
  - When a block is under-replicated, Replication Manager put it in the replication priority queue for placement.
  - When a block is over-replicated, Replication Manager prefers to remove the block from host with the list amount of storage, while maintain the replication policy.


Blocks:
.csv
Core Switch
Rack
Rack
Rack
DN1
DN2
DN3
NN
DN4
DN5
DN6
DN7
DN8
Block Replicas

Other Functionalities

- Replication Manager: Detects whether a block becomes under-replicated or over-replicated.
  - NameNode is responsible for the operation of Replication Manager.
  - When a block is under-replicated, Replication Manager put it in the replication priority queue for placement.
  - When a block is over-replicated, Replication Manager prefers to remove the block from host with the list amount of storage, while maintain the replication policy.
- **Balancer**: Balance the disk space usage on the HDFS cluster
  - HDFS block placement does not take into account disk space utilization.
  - Iteratively moves data from DataNodes with high utilization to DataNodes with lower utilization.

Other Functionalities (cont.)

- **Block Scanner**: Periodically Scans block replicas to ensure that stored checksums match block data.
  - When the checksum verification succeeds, Block Scanner informs DataNode to consider the replica as verified.
  - When the block is corrupted, Block Scanner informs NameNode. Then, NameNode schedules a replacement.
  - NameNode does not delete the corrupted block immediately. (Over-replication)
- **Decommissioning**: System's Administrator specifies which node can join the cluster.
  - When a DataNode marked as decommissioning, it will not be selected for replication, but it will still serve read requests.
  - NameNode starts to schedule replication of the blocks to another DataNode.
- **Inter-Cluster Data Copy**: HDFS provides a tool called DistCp for large inter/intra-cluster parallel copying.
  - Implemented by a Map/Reduce Job.

Real-World Experiences

- Yahoo:
  - 3500 Nodes
  - 70% of the total disk space is allocated to HDFS
  - 9.8 PB of available storage
  - ~ 3.3 PB for application usage because of the replication factor=3

Durability of Data

- For a large cluster, the probability of losing a block for one year is less than .005. -> Quite small but not zero!
- What can cause the block loss?
  - Rack Failures and Core Switch failures are a common insistent in a large cluster
  - Accidental or deliberate loss of electrical power to the cluster.
    - 0.5 – 1 % of the nodes will not survive a full power-on restart
  - 1-2 Node loss every day (Difficult to lead in a block loss due to replication)

Evaluation

- What is bandwidth observed from a contrived benchmark?
- What bandwidth is observed in a production cluster with a mix of user jobs?
- What bandwidth can be obtained by the most carefully constructed large-scale user application?

Bandwidth from a contrived benchmark

- DFSIO benchmark measures average throughput for read, write, and append operations. (Map/Reduce Application)
- Measures performance only during data transfer.

DFSIO Read: 66 MB /s per node

DFSIO Write: 40 MB /s per node

Bandwidth from a contrived benchmark (NN)

- NNThroughput benchmark measures the performance of the NameNode.
- Each client performs the same NameNode operation repeatedly

Operation	Throughput (ops/s)
Open file for read	126 100
Create file	5600
Rename file	8300
Delete file	20 700
DataNode Heartbeat	300 000
Blocks report (blocks/s)	639 700

Table 3. NNThroughput benchmark

Bandwidth with a mix of jobs

- Collected data from a production cluster. (Real Example)

Busy Cluster Read: 1.02 MB/s per node

Busy Cluster Write: 1.09 MB/s per node

Bandwidth on most-careful large-scale user applications

- A common sort algorithm
- Table are about the best a user application can achieve with the current design and hardware. (Came from Gray Sort competition)

Bytes (TB)	Nodes	Maps	Reduces	Time	HDFS I/0 Bytes/s	
					Aggregate (GB)	Per Node (MB)
1	1460	8000	2700	62 s	32	22.1
1000	3658	80 000	20 000	58 500 s	34.2	9.35

Future Work

- The Hadoop cluster is effectively unavailable when its NameNode is down.
  - Solution: Use of Zookeeper for automated failover solution √
- Scalability of the NameNode has been a key struggle. Problems when memory is close to the maximum
  - Solution: Allow multiple namespaces to share physical storage within a cluster. √

Cloud Applications Paradigms

Microservices

Microservices

- A paradigm to architect large, complex and long-lived applications as a set of (small) cohesive services that evolve over time.
  - How small? 10-100 LoC? (Not necessarily)
- Not a new concept!
  - Essentially a distributed systems architecture.
  - Resembles Service Oriented Architecture (SOA) - called sometimes lightweight or fine-grained SOA.

Micro-service Architecture

- An approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API.
- These services are built around business capabilities and independently deployable by fully automated deployment machinery.
- There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

https://martinfowler.com/articles/microservices.html

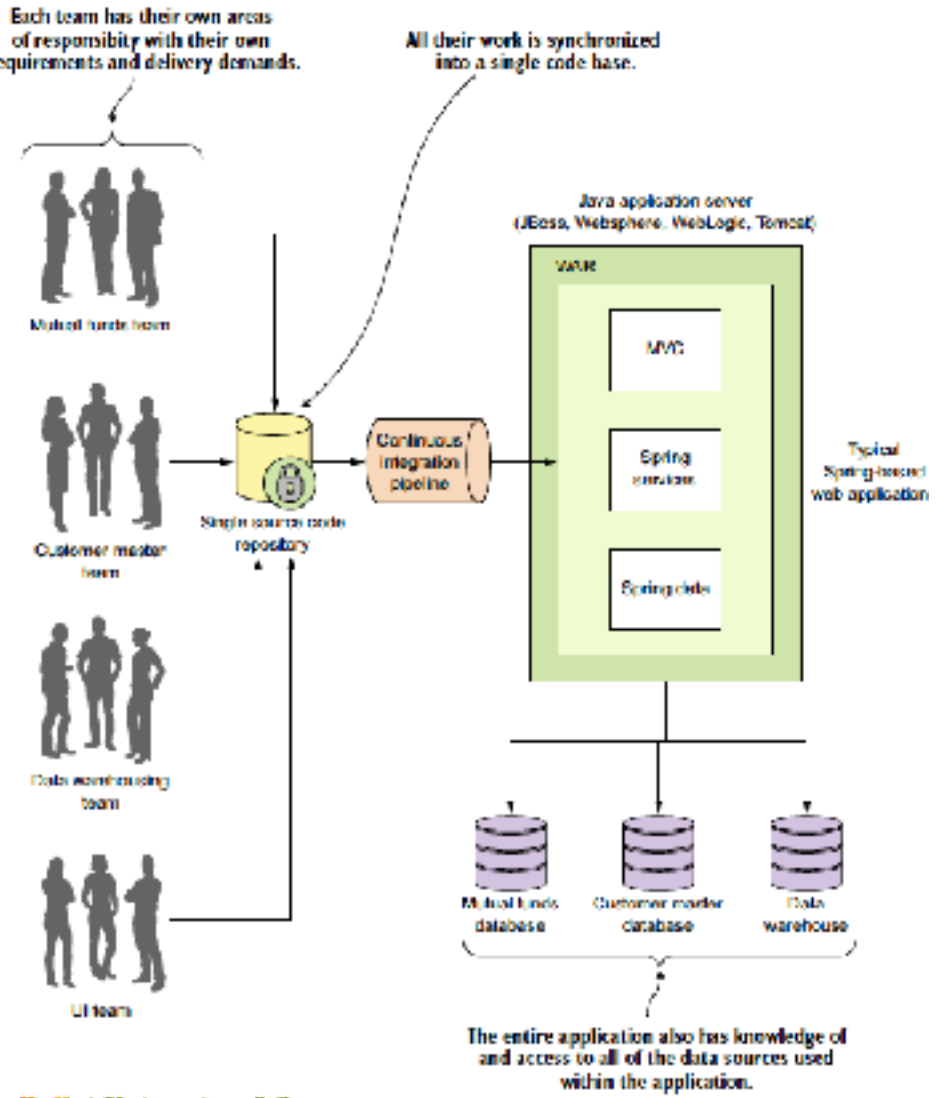
Monolithic Architectures

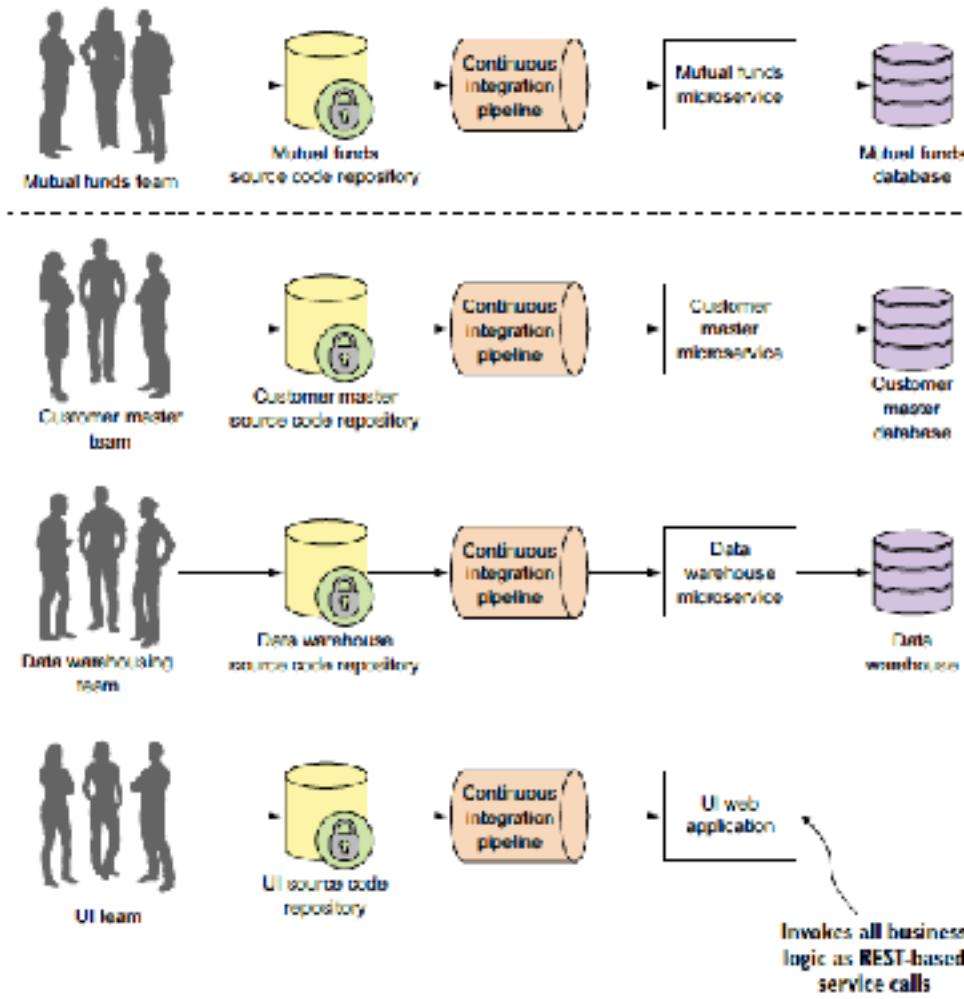
- Packages all server-side components of an application into a single unit.
- Simple to:
  - Develop: IDEs are oriented around developing a single application.
  - Test: launch and test one application.
  - Deploy: copy the deployment unit (a file or directory) to a machine running the appropriate type of server
- Complex applications:
  - Can be difficult for developers to understand and maintain.
  - Cumbersome to deploy changes to application components: you have to build and deploy the entire monolith - risky, time consuming, coordination-heavy, long test-cycles.
  - Difficult to try and adopt new technologies, without rewriting the entire application.

Key Characteristics

- Application logic is broken down into small-grained components with well-defined boundaries of responsibility that coordinate to deliver a solution.
- Each component has a small domain of responsibility and is deployed completely independently of one another. Microservices should have responsibility for a single part of a business domain. Also, a microservice should be reusable across multiple applications.
- Microservices communicate based on a few basic principles (notice I said principles, not standards) and employ lightweight communication protocols such as HTTP and JSON (JavaScript Object Notation) for exchanging data between the service consumer and service provider.
- The underlying technical implementation of the service is irrelevant because
- the applications always communicate with a technology-neutral protocol (JSON
- is the most common). This means an application built using a microservice
- application could be built with multiple languages and technologies.
- Microservices—by their small, independent, and distributed nature—allow
- organizations to have small development teams with well-defined areas of
- responsibility. These teams might work toward a single goal such as delivering
- an application, but each team is responsible only for the services on which
- they're working.

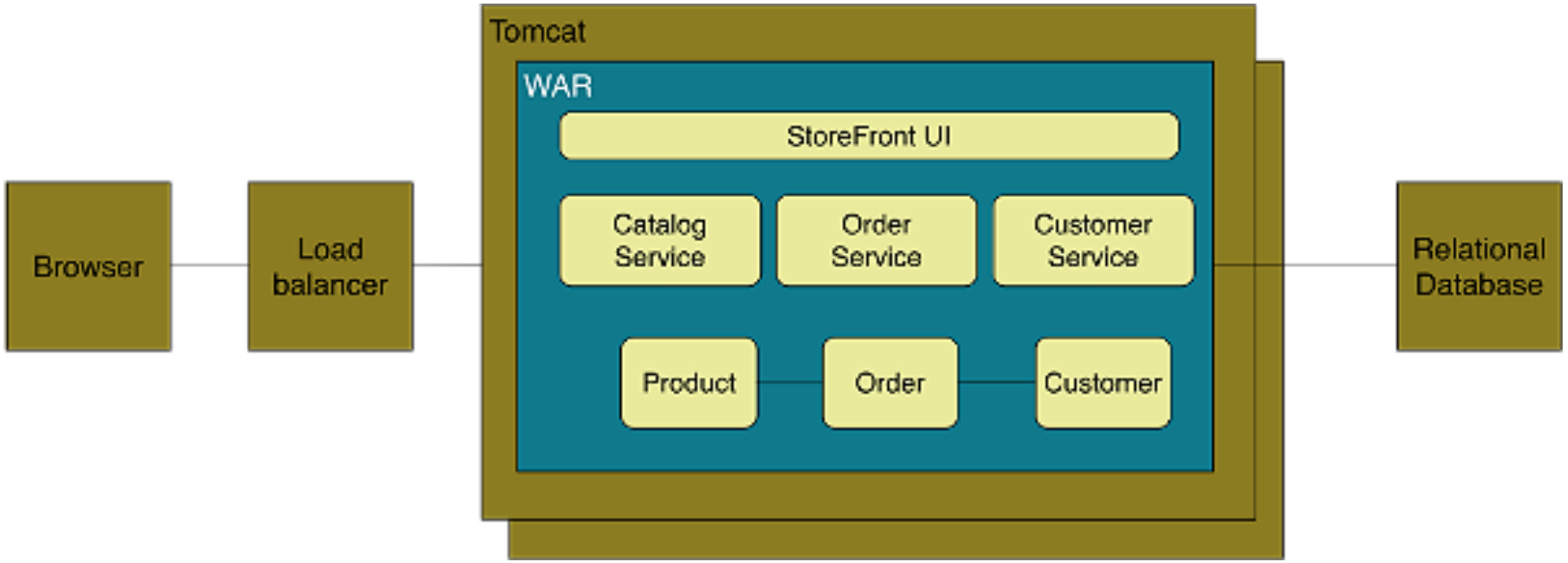
Monolithic vs Micro-service Architecture





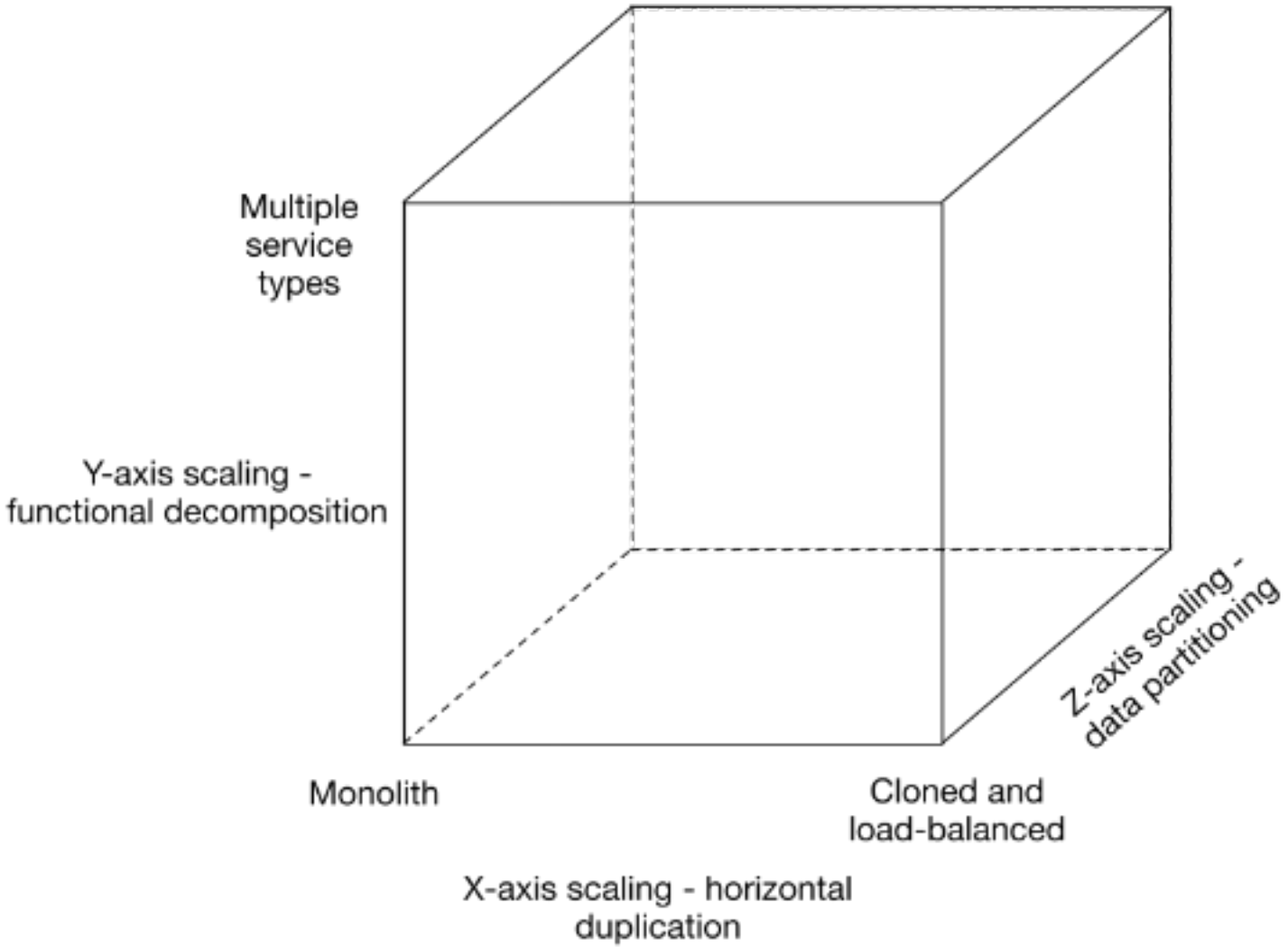
Source: Spring Microservices in Action, J. Carnell, 2017

Monolithic Architecture



https://www.infoq.com/articles/microservices-intro/

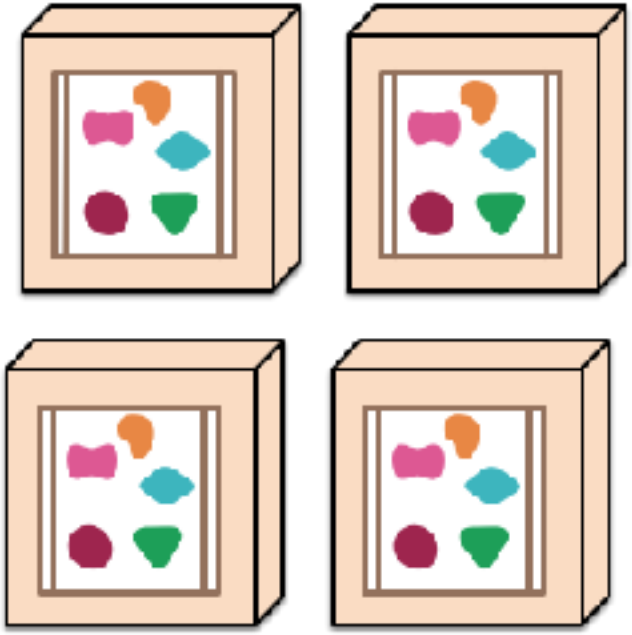
Decomposition Axes: The Scale Cube



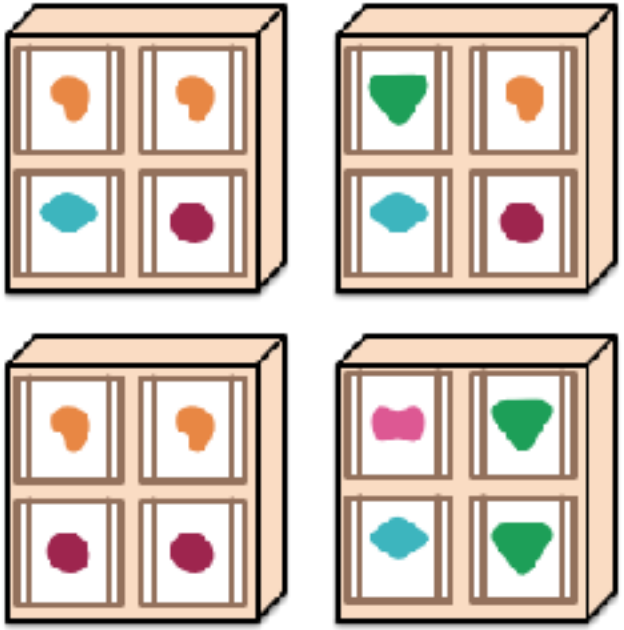
A monolithic application puts all its functionality into a single process...

A microservices architecture puts each element of functionality into a separate service...

... and scales by replicating the monolith on multiple servers



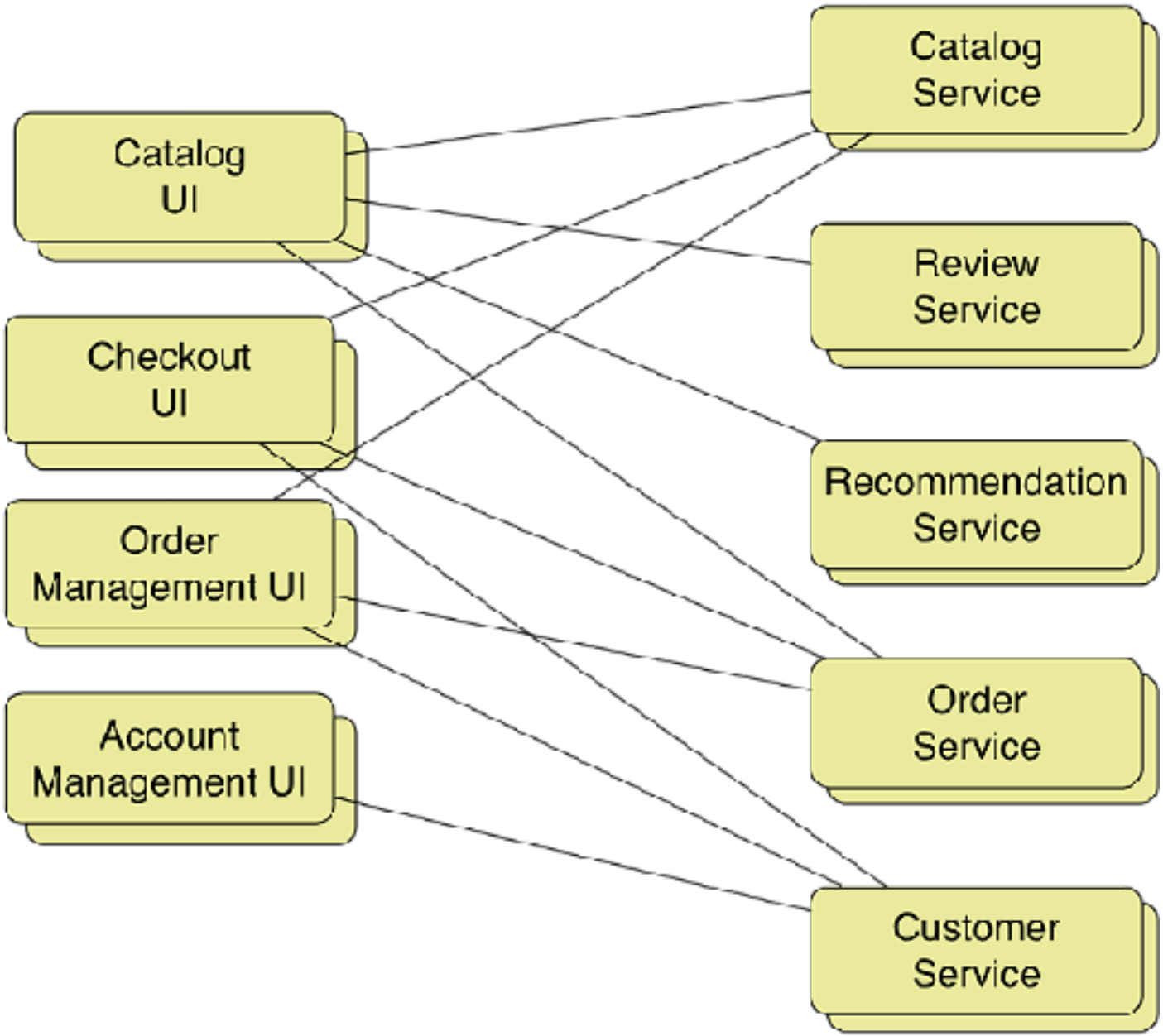
... and scales by distributing these services across servers, replicating as needed.



https://martinfowler.com/articles/microservices.html

How to partition along y-Axis?

- How to partition a system into services: art rather than science.
- Partitioning strategies:
  - By use-case or *verb*: e.g. CheckoutUI implements the UI for the **checkout** use case.
  - By resource or *noun*: a service is responsible for all operations that operate on entities/resources of a given type - e.g. a Catalog service, manages a **catalog of products**.
- Ideally, each service should have only a small set of responsibilities.
  - Review the Single Responsibility Principle (SRP).


Catalog Service
Catalog UI
Review Service
Checkout UI
Recommendation Service
Order Management UI
Order Service
Account Management UI
Customer Service

Benefits

- Each micro service can be:
  - Relatively small:
    - Code is easier for a developer to understand.
    - Small code base doesn't slow down the IDE making developers more productive.
    - Each service typically starts a lot faster than a large monolith: developers more productive, deployment speeds up.
  - Deployed independently:
    - No need for coordination with other developers when deploying local changes: continuous deployment feasible.
  - Scaled independently of other services using X-axis cloning and Z-axis partitioning.
  - Deployed on hardware that is best suited to its resource requirements.

Microservice Architecture Benefits

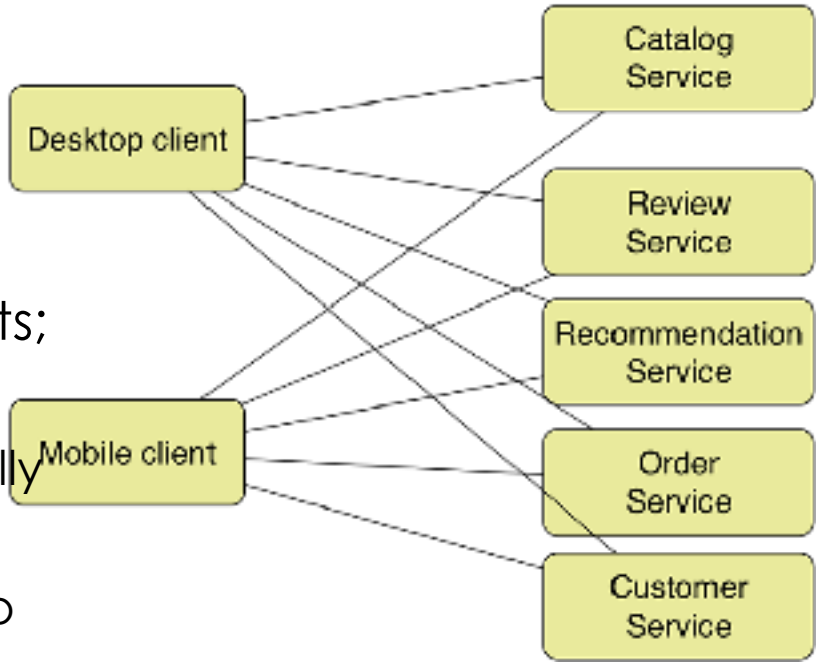
- Easier to scale development.
  - You can organize the development effort around multiple, small (e.g. two pizza) teams.
  - Each team is solely responsible for the development and deployment of a single service or a collection of related services.
  - Each team can develop, deploy and scale their service independently of all of the other teams.
- Better fault isolation. For example, a memory leak in one service only affects that service. Other services will continue to handle requests normally. In comparison, one misbehaving component of a monolithic architecture will bring down the entire system.
- Eliminates any long-term commitment to a technology stack.
  - In principle, when developing a new service the developers are free to pick whatever language and frameworks are best suited for that service.
  - Because the services are small, it becomes practical to rewrite them using better languages and technologies. If the trial of a new technology fails you can throw away that work without risking the entire project.

Drawbacks

- Additional complexity of creating a distributed system:
  - Need an inter-process communication mechanism.
  - Implementing use cases that span multiple services without using distributed transactions is difficult.
  - IDEs and other development tools focused on building monolithic applications.
  - Writing automated tests that involve multiple services is challenging.
- Significant operational complexity: many more moving parts – multiple instances of different types of service – that must be managed in production through some automated continuous delivery tool.
- Deploying features that span multiple services requires careful coordination between the various development teams: Rollout plan that orders service deployments based on the dependencies between services.
- At what point during the lifecycle of the application you should use this architecture?
  - When developing the first version of an application, you often do not have the problems that this architecture solves.
  - Moreover, using an elaborate, distributed architecture will slow down development. Using Y-axis splits might make it much more difficult to iterate rapidly.
  - Later on, however, when the challenge is how to scale and you need to use functional decomposition, then tangled dependencies might make it difficult to decompose your monolithic application into a set of services.
- Adopting a microservice architecture should not be undertaken lightly. However, for applications that need to scale, such as a consumer-facing web application or SaaS application, it is usually the right choice.

Communication Mechanisms

- In a monolithic architecture, clients of the application make HTTP requests via a load balancer to one of N identical instances of the application.
- On a microservice architecture:
- Significant mismatch in granularity between the APIs of the individual services and data required by the clients; e.g.:
  - Displaying one web page could potentially require calls to large numbers of services.
  - amazon.com: some pages require calls to 100+ services
  - Making that many requests, would be very inefficient and result in a poor user experience.



API Gateway

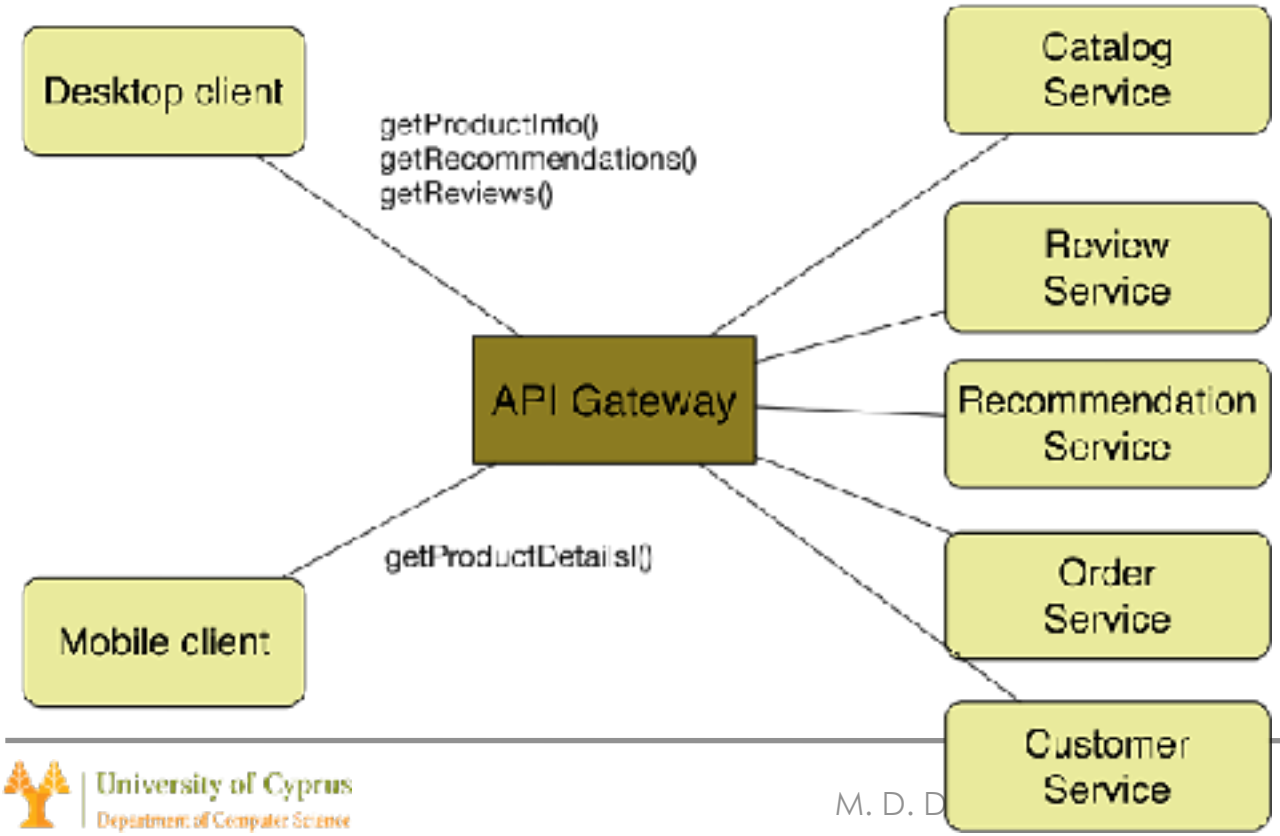
- Clients make a small number of requests per-page, perhaps as few as one, over the Internet to a front-end server known as an API gateway.
- The **API gateway** sits between the application's clients and the microservices.
- It provides APIs that are tailored to the client
- The API gateway:
  - Handles incoming requests by making requests to some number of microservices over the high-performance LAN
  - Optimizes communication between clients and the application and encapsulates the details of the microservices.
  - This enables the microservices to evolve without impacting the clients.
- For example, two microservices might be merged. Another microservice might be partitioned into two or more services. Only the API gateway needs to be updated to reflect these changes. The clients are unaffected.

API Gateway

In this example, the desktop clients makes multiple requests to retrieve information about a product, where as a mobile client makes a single request.

Fine-grained requests from a desktop client are simply proxied to the corresponding service,

Each coarse-grained request from a mobile client is handled by aggregating the results of calling multiple services.

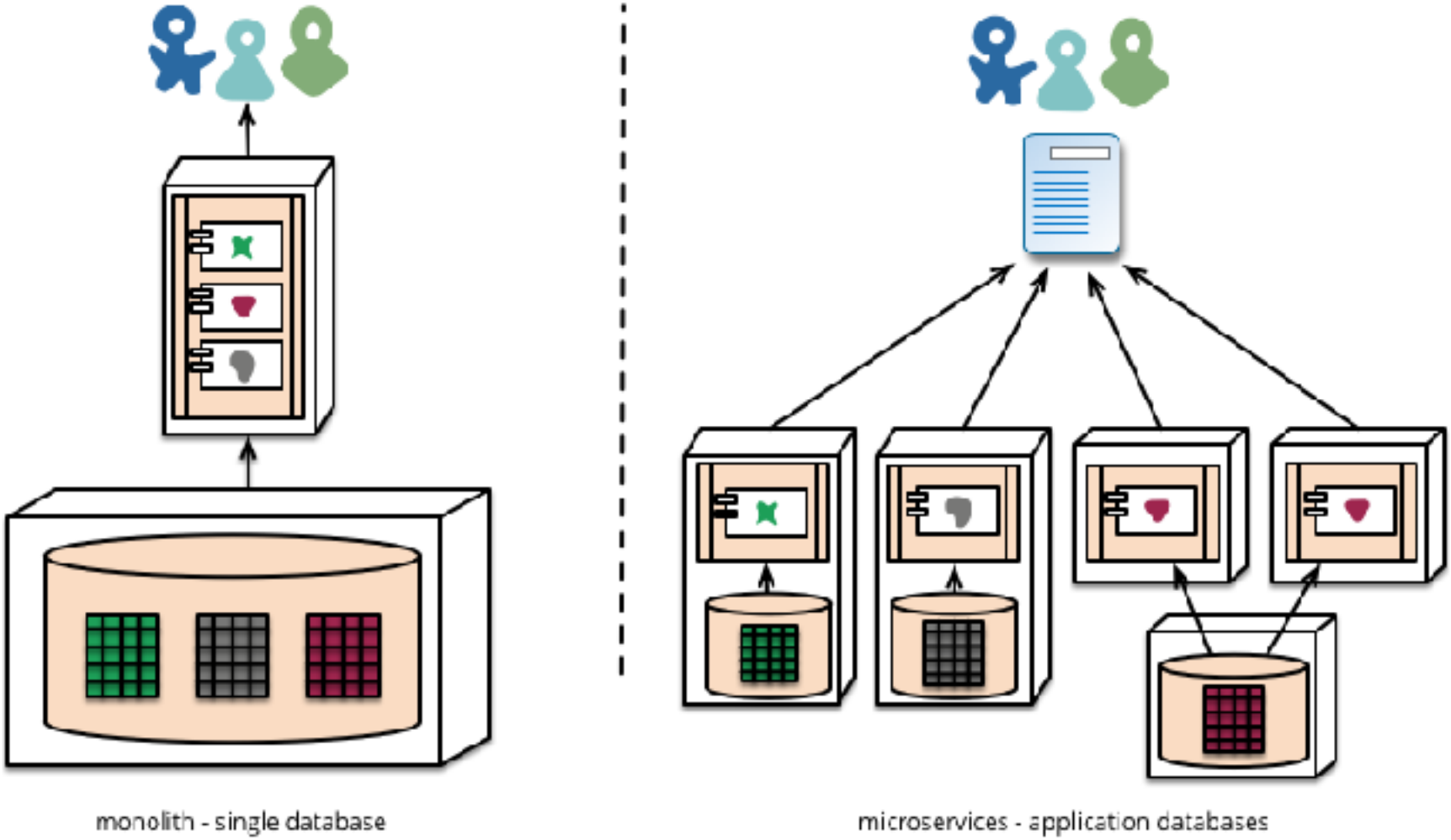


Inter-service communication mechanisms

- In a monolithic application, components call one another via regular method calls.
- In a microservice architecture, different services run in different processes: services must use an inter-process communication (IPC) to communicate.
- There are two main approaches to inter-process communication in a microservice architecture:
- Synchronous HTTP-based mechanism such as REST or SOAP.
  - Simple, firewall friendly, easy to implement the request/reply style of communication
  - Doesn't support other patterns of communication such as publish-subscribe.
  - Both the client and the server must be simultaneously available, which is not always the case since distributed systems are prone to partial failures.
  - Also, an HTTP client needs to know the host and the port of the server => applications need to use a service registry/discovery mechanism (Zookeeper, Consul, Eureka)
- Asynchronous message-based mechanism such as an AMQP-based message broker.
  - Decouples message producers from consumers. Producers are completely unaware of the consumers.
  - Message broker buffers messages until the consumer is able to process them. Producer simply talks to the message broker and does not need to use a service discovery mechanism.
  - Supports a variety of communication patterns: one-way requests and publish-subscribe, but request/reply-style is **not** a natural fit.

Data Management in Microservices

- A consequence of decomposing the application into services is that the database is also partitioned.
- To ensure loose coupling:
  - Each service has its own database (schema).
  - Different services might use different types of database – (polyglot persistence architecture).
- Partitioning the database is essential, but we now have a new problem to solve: how to handle those requests that access data owned by multiple services.



monolith - single database

microservices - application databases

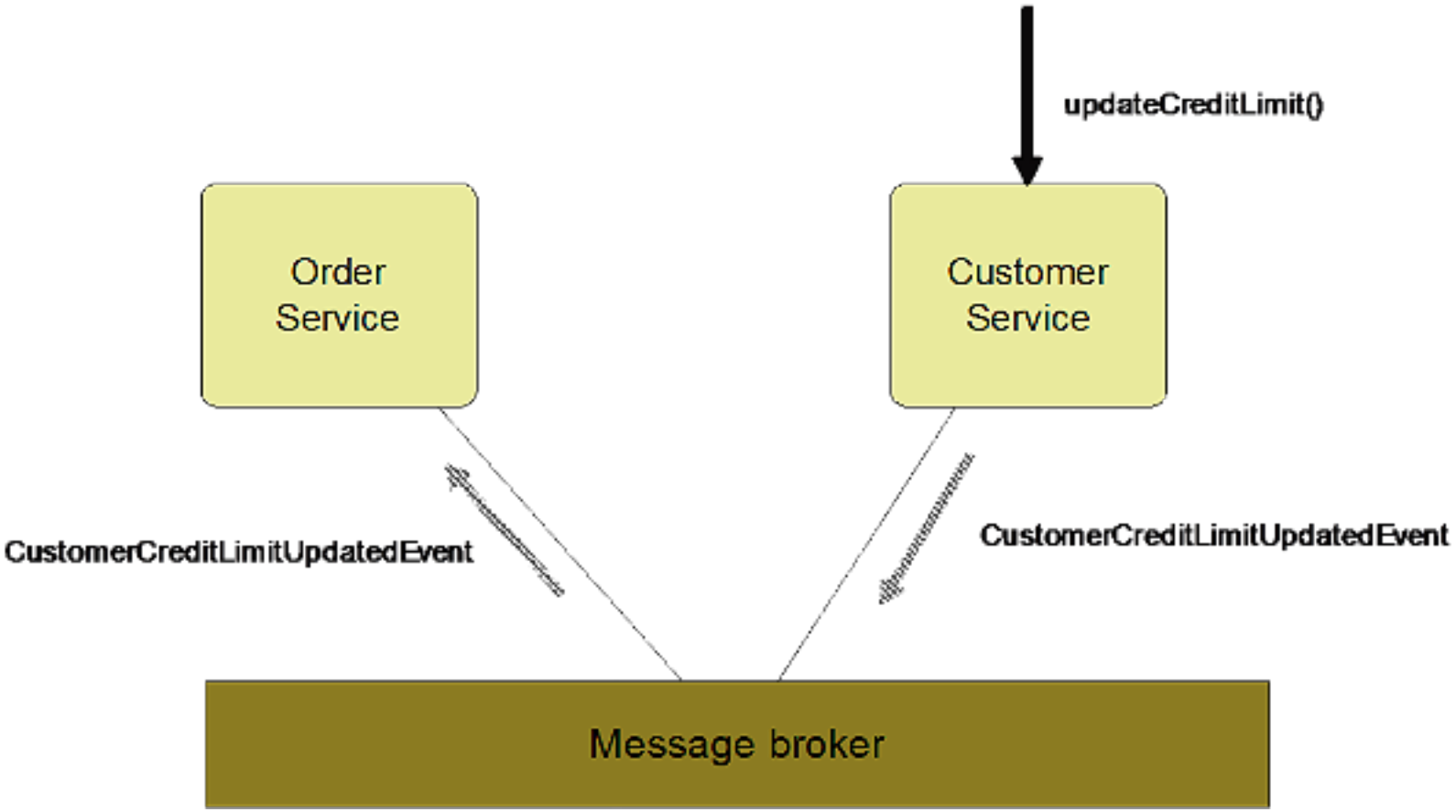
https://martinfowler.com/articles/microservices.html

Handling Reads

- If data not available within a service, make RPC call to owner:
  - Reduces availability, if owner is down.
  - Reduces performance, due to extra RPC.
- Keep copy local to the service:
  - Eliminates need for RPC, improving availability and response time.
  - Need to have an update mechanism in place.

Handling updates

- How to handle requests that update data owned by multiple services?
- Distributed transactions:
  - Would ensure that the data is always consistent.
  - Downside: reduces system availability since all participants must be available in order for the transaction to commit.
  - Distributed transactions really have fallen out of favor and are generally not supported by modern software stacks, e.g. REST, NoSQL databases, etc.
- Event-driven asynchronous updates:
  - Services publish events announcing that some data has changed.
  - Other services subscribe to those events and update their data.
  - Producers and consumers of the events decoupled: simplifies development and improves availability. If a consumer isn't available to process an event then the message broker will queue the event until it can.
  - Trades consistency for availability: the application has to be written in a way that can tolerate eventually consistent data. Developers might also need to implement compensating transactions to perform logical rollbacks.


updateCreditLimit()
Order Service
Customer Service
CustomerCreditLimitUpdatedEvent
CustomerCreditLimitUpdatedEvent
Message broker

Cloud Applications Paradigms

Serverless

Cloud Deploying issues

1. Redundancy for availability, so that a single machine failure doesn't take down a service or application.
2. Geographic distribution of redundant copies to preserve the service in case of disaster.
3. Load balancing and request routing to efficiently utilize resources.
4. Autoscaling in response to changes in load to scale up or down the system.
5. Monitoring to make sure the service is still running well.
6. Logging to record messages needed for debugging or performance tuning.
7. System upgrades, including security patching.
8. Migration to new instances as they become available.

Serveless: The Origins

- Seeking cost savings and scalability without needing to have a high level of cloud computing expertise that is time-consuming to acquire (scarcity of DevOps expertise).
- A substantial gap between the resources that cloud customers allocate and pay for (leasing VMs), and actual resource utilization (CPU, memory, and so on).
- Need to improve resource utilization in Cloud Computing infrastructures.
- The recent shift of enterprise application architectures to containers and microservices.

Serveless: The Origins

- Serverless platforms can be considered an evolution of Platform-as-a-Service (PaaS) as provided by platforms such as Cloud Foundry, Heroku, and Google App Engine (GAE):
  - PaaS was defined by NIST as "*the capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages and tools supported by the provider. The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations.*"
- In this definition, users are expected to manage deployments of applications and have control over hosting environment configurations.
- Implementing auto-scaling in PaaS is not easy and it is very difficult to scale to zero.

Serverless: Key Concepts

- Gives pay-as-you-go without additional work to start and stop server and is closer to original expectations for cloud computing to be treated like as a utility.
- Is about removing user control over hosting to provide simpler scaling and more attractive billing model
  - **Cloud provider** controls the hosting environment's configuration, runs user-provided code only when it is invoked, and only bills for actual usage while hiding the complexity of scaling.
  - The **user** just writes a cloud function in a high-level language, picks the event that should trigger the running of the function and lets the serverless system handle everything else: instance selection, scaling, deployment, fault tolerance, monitoring, logging, security patches, etc.

Serverless: Popular Uses

- Most prominent cloud providers including Amazon (AWS Lambda), IBM, Microsoft, Google, and others have already released serverless computing capabilities with several additional open source efforts driven by both industry and academic institutions (for example, see CNCF Serverless Cloud

Percent	*Use Case*
32%	Web and API serving
21%	Data Processing, e.g., batch ETL (database Extract, Transform, and Load)
17%	Integrating 3rd Party Services
16%	Internal tooling
8%	Chat bots e.g., Alexa Skills (SDK for Alexa AI Assistant)
6%	Internet of Things

Table 4: Popularity of serverless computing use cases according to a 2018 survey [22].

Serverless Definitions

"Serverless computing is a platform that hides server usage from developers and runs code on-demand automatically scaled and billed only for the time the code is running." [CACM, 12/19]

"The concept of building and running applications that do not require server management. It describes a finer-grained deployment model where applications, bundled as one or more functions, are uploaded to a platform and then executed, scaled, and billed in response to the exact demand needed at the moment." [CNCF]

"A serverless solution is one that costs you nothing to run if nobody is using it (excluding data storage)." [Paul Johnston]

The name 'serverless computing' does not mean servers are not used, but merely that developers can leave most operational concerns of managing servers and other resources, including provisioning, monitoring, maintenance, scalability, and fault-tolerance to the cloud provider.

Key Features

- Cost billed only for what is running (pay-as-you-go).
  - As execution time may be short, then it should be charged in fine-grained time units (like hundreds of milliseconds) and developers do not need to pay for overhead of servers creation or destructions.
  - Cost model supports "scaling to zero" = avoid need to pay for idle servers.
- Elasticity scaling from zero to "infinity."
  - Decisions about scaling are left to cloud providers.
  - Developers do not need to write auto-scaling policies or define how machine-level usage (CPU, memory, and so on) translates to application usage: depend on the cloud provider.
  - Developers can assume cloud provider will take care of maintenance, security updates, availability and reliability monitoring of servers.
- Typically favors small, self-contained units of computation to make it easier to manage and scale in the cloud.
- However, a computation, which can be interrupted or restarted, cannot depend on the cloud platform to maintain its state. This inherently influences the serverless computing programming models.
  - No equivalent notion of scaling to zero when it comes to state, since a persistent storage layer is needed. which supports powerful auto-scaling capabilities but requires minimum memory and CPU allocations and hence does not scale to zero and has ongoing costs.

Critical Distinctions

- Decoupled computation and storage.
  - The storage and computation scale separately and are provisioned and priced independently.
  - Storage is provided by a separate cloud service and the computation is stateless.
- Executing code without managing resource allocation.
  - Instead of requesting resources, the user provides a piece of code and the cloud automatically provisions resources to execute that code.
- Paying in proportion to resources used instead of for resources allocated.
  - Billing done by some dimension of the execution (exec. time) rather than by a dimension of the base cloud platform, such as size and number of VMs allocated.

Opportunities & Risks: App Developer

- Opportunities:
  - A simplified programming model for creating cloud applications that abstracts away most, if not all, operational concerns.
  - No longer have to worry about availability, scalability, fault tolerance, over/ underprovisioning of VM resources, managing servers, and other infrastructure issues.
  - Can focus on the business aspects of applications.
  - Lower cost of deploying cloud code by charging for execution time rather than resource allocation.
- Risks:
  - Deploying applications requires relinquishing design decisions to the platform provider: quality-of-service (QoS) monitoring, scaling, and fault-tolerance properties.
  - Application's requirements may evolve to conflict with the capabilities of the platform.

Opportunities & Risks: Cloud Operator

- Opportunities:
  - Control the entire development stack
  - Reduce operational costs by efficient optimization and management of cloud resources
  - Offer a platform that encourages the use of additional services in their ecosystem
  - Lower the effort required to author and manage cloud-scale applications
- Risks:
  - Slow adoption because of lack of proper tools and frameworks.

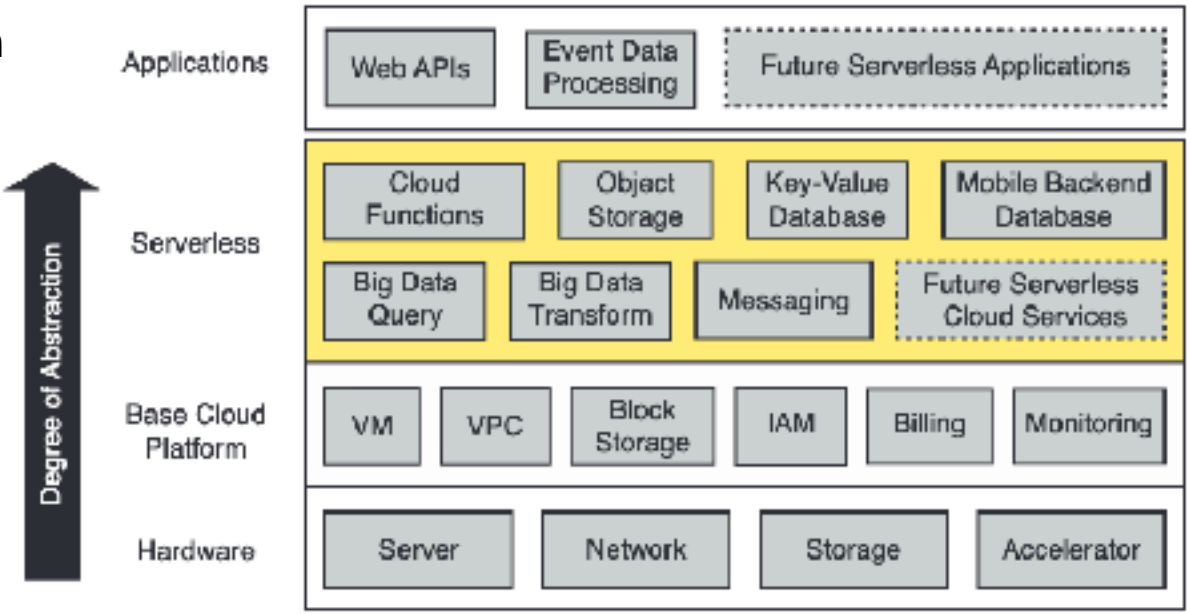
Serverless vs Serverful

	Characteristic	*AWS Serverless Cloud*	*AWS Serverful Cloud*
PROGRAMMER	When the program is run	On event selected by Cloud user	Continuously until explicitly stopped
	Programming Language	JavaScript, Python, Java, Go, C#, etc[4]	Any
	Program State	Kept in storage (stateless)	Anywhere (stateful or stateless)
	Maximum Memory Size	0.125 - 3 GiB (Cloud user selects)	0.5 - 1952 GiB (Cloud user selects)
	Maximum Local Storage	0.5 GiB	0 - 3600 GiB (Cloud user selects)
	Maximum Run Time	900 seconds	None
	Minimum Accounting Unit	0.1 seconds	60 seconds
	Price per Accounting Unit	$0.0000002 (assuming 0.125 GiB)	$0.0000867 - $0.4080000
	Operating System & Libraries	Cloud provider selects[5]	Cloud user selects
SYSADMIN	Server Instance	Cloud provider selects	Cloud user selects
	Scaling[6]	Cloud provider responsible	Cloud user responsible
	Deployment	Cloud provider responsible	Cloud user responsible
	Fault Tolerance	Cloud provider responsible	Cloud user responsible
	Monitoring	Cloud provider responsible	Cloud user responsible
	Logging	Cloud provider responsible	Cloud user responsible

Specifications and prices correspond to AWS Lambda and to on-demand AWS EC2 instances (2019)

Serverless Architecture

- Serverless layer sits between applications and the base cloud platform.
- Cloud functions (i.e., FaaS) provide general compute services and are complemented by an ecosystem of specialized Backend as a Service (BaaS) offerings such as object storage, databases, or messaging:
  - On AWS: Lambda
  - DynamoDB (key-
  - On Google's clou
    - Firestore (mobile
    - Pub/Sub (messag
- Serverless also comprises certain big data services such as AWS Athena and Google BigQuery (big data query) and Google Cloud Dataflow and AWS Glue (bigdata transform).



Function as a Service

- What is the most natural way to use serverless computing?
  - Provide a piece of code (function) to be executed by the serverless platform.
- Leads to the rise of Function-as-a-service (FaaS) platforms focused on:
  - allowing small pieces of code represented as functions to run for limited amount of time (at most minutes),
  - with executions triggered by events or HTTP requests (or other triggers),
  - and not allowed to keep persistent state (function may be restarted any time).
- By limiting time of execution and not allowing functions to keep persistent state FaaS platforms can be easily maintained and scaled by service providers.
  - Cloud providers can allocate servers to run code as needed and can stop servers after functions finish as they run for limited amount of time.
  - For functions to maintain state, they can use external services to persist their state.

Function as a Service

- FaaS is an embodiment of serverless computing principles, which we define as follows:

Function-as-a-Service is a serverless computing platform where the unit of computation is a function that is executed in response to triggers such as events or HTTP requests.

FaaS vs SaaS vs PaaS vs IaaS

	IaaS	1st Gen PaaS	FaaS	BaaS/SaaS
Expertise required	High	Medium	Low	Low

FaaS vs SaaS vs PaaS vs IaaS

	IaaS	1st Gen PaaS	FaaS	BaaS/SaaS
Expertise required	High	Medium	Low	Low
Developer Control/Customization allowed	High	Medium	Low	Very low

FaaS vs SaaS vs PaaS vs IaaS

	IaaS	1st Gen PaaS	FaaS	BaaS/SaaS
Expertise required	High	Medium	Low	Low
Developer Control/Customization allowed	High	Medium	Low	Very low
Scaling/Cost	Requires high-level of expertise to build auto-scaling rules and tune them	Requires high-level of expertise to build auto-scaling rules and tune them	Auto-scaling to work load requested (function calls), and only paying for when running (scale to zero)	Hidden from users, limits set based on pricing and QoS

FaaS vs SaaS vs PaaS vs IaaS

	IaaS	1st Gen PaaS	FaaS	BaaS/SaaS
Expertise required	High	Medium	Low	Low
Developer Control/Customization allowed	High	Medium	Low	Very low
Scaling/Cost	Requires high-level of expertise to build auto-scaling rules and tune them	Requires high-level of expertise to build auto-scaling rules and tune them	Auto-scaling to workload requested (function calls), and only paying for when running (scale to zero)	Hidden from users, limits set based on pricing and QoS
Unit of work deployed	Low-level infrastructure building blocks (VMs, network, storage)	Packaged code that is deployed and running as a service	One function execution	App-specific extensions

FaaS vs SaaS vs PaaS vs IaaS

	IaaS	1st Gen PaaS	FaaS	BaaS/SaaS
Expertise required	High	Medium	Low	Low
Developer Control/Customization allowed	High	Medium	Low	Very low
Scaling/Cost	Requires high-level of expertise to build auto-scaling rules and tune them	Requires high-level of expertise to build auto-scaling rules and tune them	Auto-scaling to workload requested (function calls), and only paying for when running (scale to zero)	Hidden from users, limits set based on pricing and QoS
Unit of work deployed	Low-level infrastructure building blocks (VMs, network, storage)	Packaged code that is deployed and running as a service	One function execution	App-specific extensions
Granularity of billing	Medium to large granularity: minutes to hours per resource to years for discount pricing	Medium to large granularity: minutes to hours per resource to years for discount pricing	Very low granularity: hundreds of milliseconds of function execution time	Large: typically, subscription available based on maximum number of users and billed in months

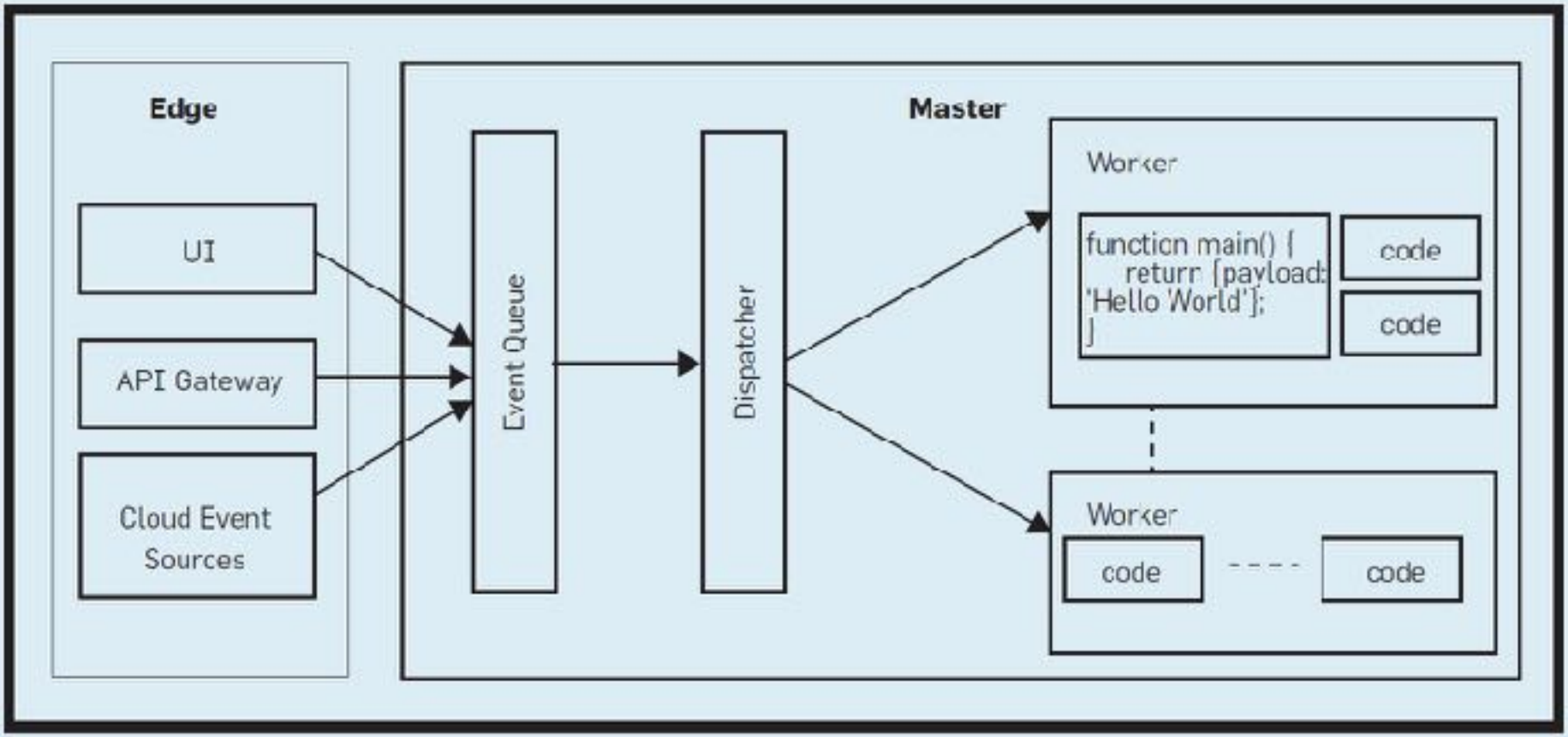
Serverless Implementation

- Core functionality?
  - Event processing system
- Main activity?
  - Once a request is received over HTTP from an event data source (a.k.a. triggers), the system determines which action(s) should handle the event, create a new container instance, send the event to the function instance, wait for a response, gather execution logs, make the response available to the user, and stop the function when it is no longer needed.
- Key abstraction level?
  - A short-running stateless function.
  - Expressive enough to build useful applications, but simple enough to allow the platform to autoscale in an application agnostic manner.

Request Lifecylce

1. Upon the arrival of an event, the platform proceeds to validate the event ensuring it has the appropriate authentication and authorization to execute.
2. Platform also checks the resource limits for that particular event.
3. Once the event passes validation, the platform the event is queued to be processed.
4. A worker fetches the request, allocates the appropriate container, copies over the function - use code from storage - into the container and executes the event.
5. The platform also manages stopping and deallocating resources for idle function instances.

High-level Serverless FaaS platform Architecture



Implementation Challenges

- Relies on strong performance & security isolation to make multi-tenant hardware sharing possible.
- How?
  - VM-like isolation: current standard for multi-tenant hardware sharing for cloud functions
- But: VM provisioning can take many seconds
- Need elaborate techniques to speed up the creation of function execution environments:
  - [AWS Lambda]: maintain a "warm pool" of VM instances that need only be assigned to a tenant, and an "active pool" of instances that have been used to run a function before and are maintained to serve future invocations.
  - Reduce the overhead of providing multi-tenant isolation by leveraging containers (Docker), unikernels, library OSes, or language VMs.

Implementation Challenges

- Cold-start problem: latency of creating, instantiating, and destroying a new container for each function invocation.
  - Warm containers are containers that were already instantiated and executed a function
  - Stem-cell containers: warm-container reuse
- Library installation delay

In previous lecture

- Explored the underpinnings of Serverless Cloud Computing.
- Covered the definitions, reference architecture of server less and key differences with SaaS, PaaS, IaaS
- Discussed the programming model of serverless computing

Today

- Finish the discussion on serverless computing and see presentations of recent articles on serverless.

Examples of Serverless Services

Service	*Programming Interface*	*Cost Model*
Cloud Functions	Arbitrary code	Function execution time
BigQuery/Athena	SQL-like query	The amount of data scanned by the query
DynamoDB	puts() and gets()	Per put() or get() request + storage
SQS	enqueue/dequeue events	per-API call

Serverless = FaaS + BaaS

- For BigQuery, Athena,and cloud functions, the user pays separately for storage (e.g., in Google Cloud Storage, AWS S3,or Azure Blob Storage)

FaaS Programming Model

- A typical FaaS programming model consists of two major primitives: **Action** (ενέργεια) and **Trigger** (έναυσ□α).
- An **Action** is a stateless function that executes arbitrary code.
  - Actions can be invoked directly via REST API or executed based on a trigger.
  - Invocation can be:
    - Asynchronous: invoker caller request does not expect a response, or
    - Synchronous: invoker expects a response as a result of the action execution.
- A **Trigger** is a class of events from a variety of sources:
  - Events can trigger multiple functions (parallel invocations), or
  - The result of an action can trigger another function (sequential invocations).
- Some serverless frameworks provide higher level programming abstractions such as function packaging, sequencing, and composition.

FaaS Programming Model

- Serverless frameworks execute a single main function that takes a dictionary (such as a JSON object) as input and produces a dictionary as output (circa 2020).
- Serverless functions have limited expressiveness as they are built to scale.
- To maximize scaling, functions do not maintain state between executions.
  - Instead, the developer can write code in the function to retrieve and update any needed state.
  - The function is also able to access a context object that represents the environment in which the function is running (such as a security context).
- The stateless nature of serverless functions leads to application structure similar to those found in functional reactive programming
  - Applications that exhibit event-driven and flow-like processing patterns

```
function main(params, context) {
    return {payload: 'Hello, ' + params.name
                    + ' from ' + params.place};
}
```

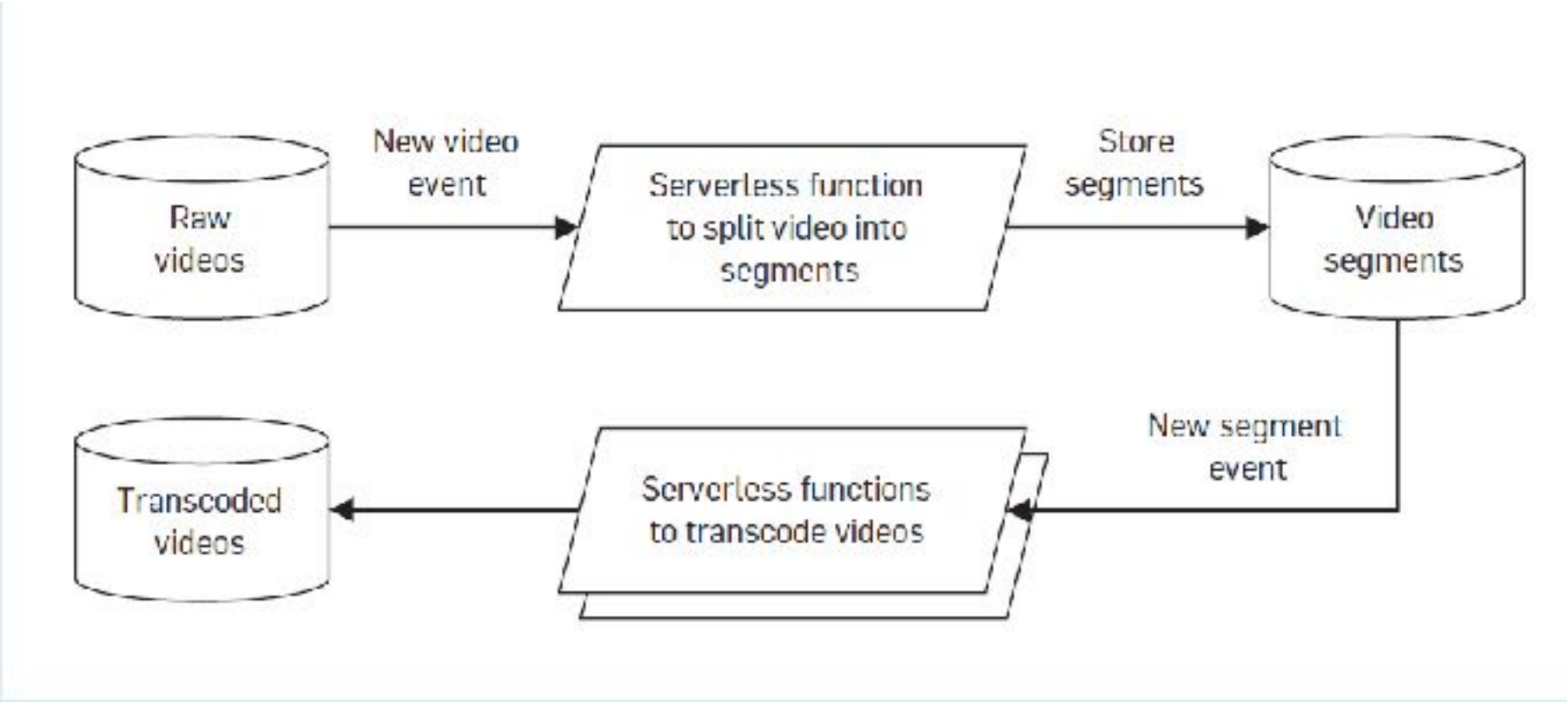
Current Cloud Provider Serverless offerings support Java, Python, Swift, C#, and Node.js etc.

Some of the platforms also support extensibility mechanisms for code written in any language as long as it is packaged in a Docker image that supports a well-defined API.

Use case: Event Processing

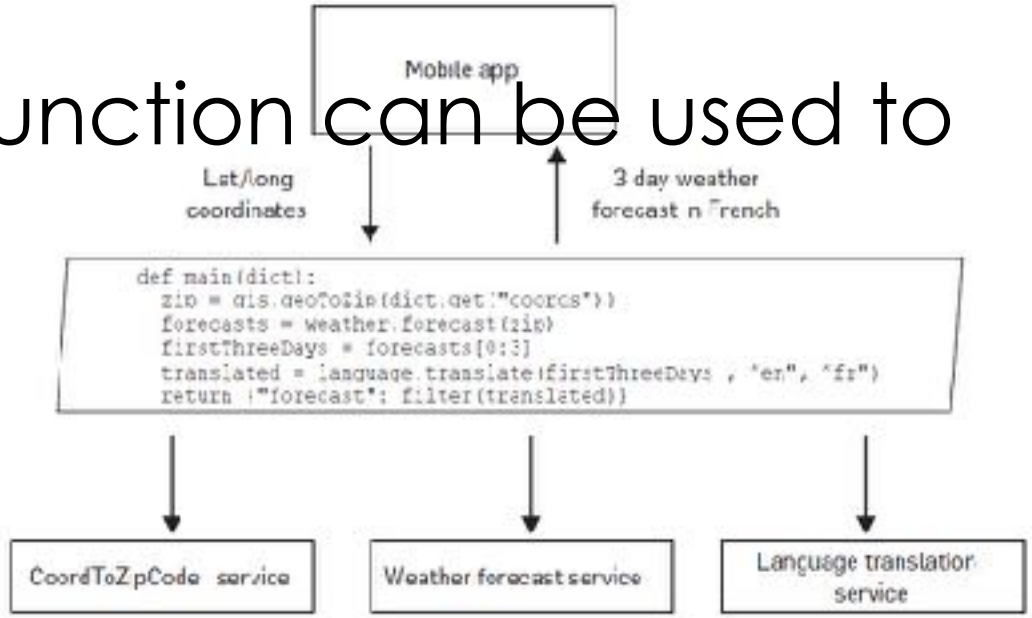
- [Netflix] Uses serverless functions to process video files:
  - The videos are uploaded to Amazon S3 which emits events that
  - trigger Lambda functions that split the videos and transcode them in parallel to different formats.
- The function is completely stateless and idempotent:
  - In the case of failure, the function can be executed again with *no side effects*.
- Combine serverless functions with other services from the cloud provider, to develop more complex applications, e.g.:
  - stream processing, filtering and transforming data on the fly, chatbots, and Web applications.

Use case: Event Processing



Use Case: API Composition

- Mobile app that sequentially invokes a geolocation, weather, and language translation APIs to render the weather forecast for a user's current location.
- A short serverless function can be used to invoke these APIs:
  - The mobile app avoids
    - invoking multiple APIs over a potentially resource constrained mobile network connection, and
    - offloads the filtering and aggregation logic to the backend.

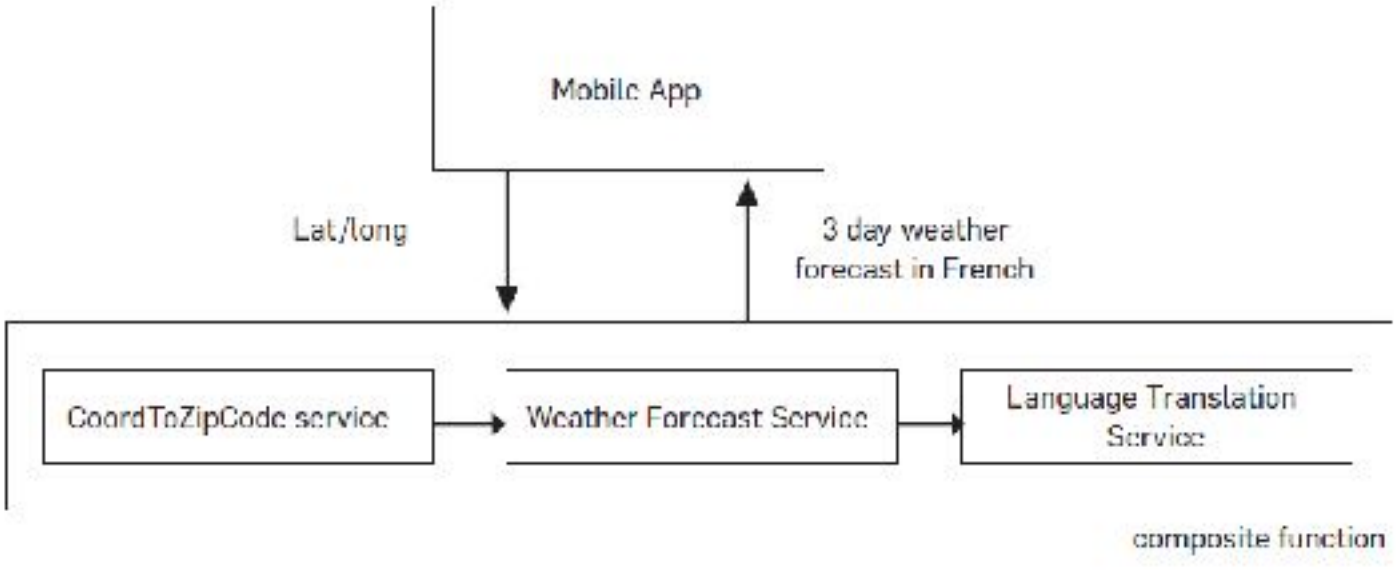


Use Case: API Composition

- Problem:
  - Main function is acting as an orchestrator that is waiting for a response from a function before invoking another, thus incurring a cost of execution while the function is basically waiting for I/O.
- Such a pattern of programming is referred to as a serverless anti-pattern.

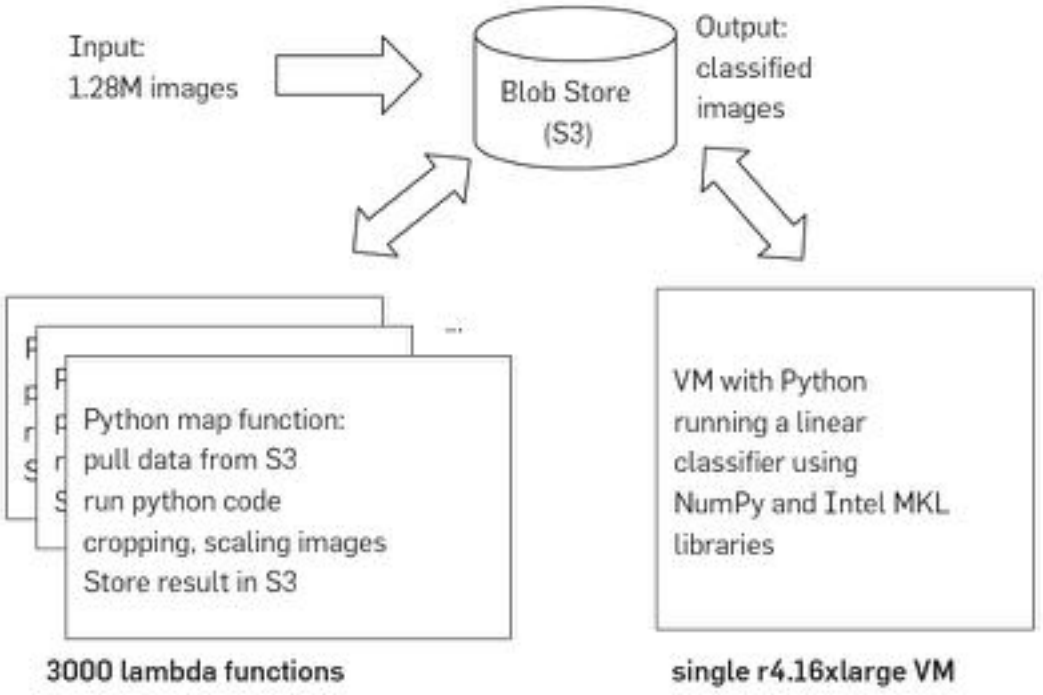
Use Case: API Composition

- The serverless programming approach would be to encapsulate each API call as serverless function, and chain the invocation of these functions in a se
- Th
cc



Use Case: Map-Reduce Style Analytics

- Python-based system that utilizes the serverless framework to help users avoid the significant development and management overhead of running



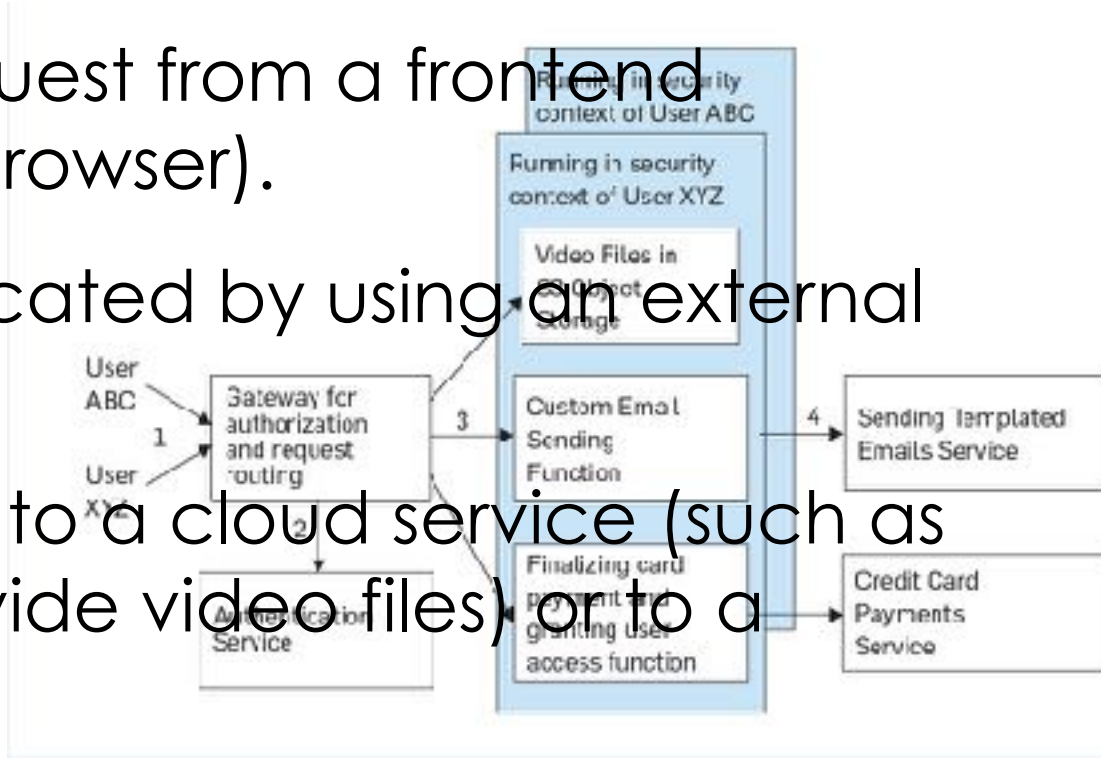
ak
da, using
ıg.

Use Case: Multi-Tenant Cloud Services

- Cloud Guru provides users with cloud training that includes videos:
  - on-demand
  - optimizing delivery cost
- Has unpredictable usage patterns; subject to change depending on holidays or promotions.
- Must scale and isolate users for security reasons while providing for each user backend functionality such as payment processing or sending email messages.

Lifecycle

1. A user makes a request from a frontend application (Web browser).
2. Request is authenticated by using an external service.
3. Request sent either to a cloud service (such as object store to provide video files) or to a serverless function.
4. Function makes necessary customizations and typically invokes other functions or cloud services.



Readings

- P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," Commun. ACM, vol. 62, no. 12, pp. 44–54, Nov. 2019.
- E. Jonas et al., "Cloud Programming Simplified: A Berkeley View on Serverless Computing," Feb. 2019.

Serveless Applications Study

Application	*Description*	*Challenges*	*Workarounds*	*Cost-performance*
Real-time video compression (ExCamera)	On-the-fly video encoding	Object store too slow to support fine-grained communication; functions too coarse grained for tasks.	Function-to-function communication to avoid object store; a function executes more than one task.	60x faster, 6x cheaper versus VM instances.
MapReduce	Big data processing (Sort 100TB)	Shuffle doesn't scale due to object stores latency and IOPS limits	Small storage with low-latency, high IOPS to speed-up shuffle.	Sorted 100 TB 1% faster than VM instances, costs 15% more.
Linear algebra (Numpy-wren)	Large scale linear algebra	Need large problem size to overcome storage (S3) latency, hard to implement efficient broadcast.	Storage with low-latency high-throughput to handle smaller problem sizes.	Up to 3x slower completion time. 1.26x to 2.5x lower in CPU resource consumption.

Serveless Applications Study

Application	*Description*	*Challenges*	*Workarounds*	*Cost-performance*
ML pipelines (Cirrus)	ML training at scale	Lack of fast storage to implement parameter server; hard to implement efficient broadcast, aggregation.	Storage with low-latency, high IOPS to implement parameter server.	3x-5x faster than VM instances, up to 7x higher total cost.
Databases (Serverless SQLite)	Primary state for applications (OLTP)	Lack of shared memory, object store has high latency, lack of support for inbound connectivity.	Shared file system can work if write needs are low.	3x higher cost per transaction than published TPC-C benchmarks. Reads scale to match but writes do not.

Limits of Serverless

- Inadequate storage for fine-grained operations
- Lack of fine-grained coordination
- Poor performance for standard communication patterns
- Achieving predictable performance

Inadequate storage

- Difficult to support applications that have fine-grained state sharing needs:
  - mostly due to the limitations of existing storage services offered by cloud providers.
- Object storage services such as AWS S3, Azure Blob Storage, and Google Cloud Storage are highly scalable and provide inexpensive long-term object storage, but exhibit high access costs and high access latencies:
  - At least 10 milliseconds to read or write small objects.
  - You can get high IOPS throughput, but with a high cost: 100K IOPS sustained, costs $30/min.
  - Key-value databases, such as AWS DynamoDB, Google Cloud Datastore, or Azure CosmosDB provide high IOPS, but are expensive and can take a long time to scale up.
- In-memory storage instances based on popular open source projects such as Memcached or Redis, are offered by Cloud providers but are not fault tolerant and do not autoscale.

Characteristics of storage services

- Costs are monthly values for storing **1 GB** (capacity), transferring **1 MB/s** (throughput), and issuing **1 IOPS**(or 2.4 million requests in 30 days).
- All values reflect a 50/50 read/write balance and a minimum transfer size of 4 KB.
- Color codings of entries are green for good, orange for medium, and red for poor.
- Persistence and availability guarantees describe how well the system tolerates failures:
  - local provides reliable storage at one site
  - distributed ensures the ability to survive site failures, and
  - ephemeral describes data that resides in memory and is subject to loss, e.g., in the event of software crashes.

Characteristics of storage services

	Block Storage (e.g., AWS EBS, IBM Block Storage)	Object Storage (e.g., AWS S3, Azure Blob Store, Google Cloud Storage)	File System (e.g., AWS EFS, Google Filestore)	Elastic Database (e.g., Google Cloud Datastore, Azure Cosmos DB)	Memory Store (e.g., AWS ElastiCache, Google Cloud Memorystore)	**"Ideal" storage service for serverless computing**
Cloud functions access	No	Yes	Yes[13]	Yes	Yes	Yes
Transparent Provisioning	No	Yes	Capacity only[14]	Yes[15]	No	Yes
Availability and persistence guarantees	Local & Persistent	Distributed & Persistent	Distributed & Persistent	Distributed & Persistent	Local & Ephemeral	Various
Latency (mean)	< 1ms	10 – 20ms	4 – 10ms	8 – 15ms	< 1ms	< 1ms
Cost[16] Storage capacity (1 GB/month)	$0.10	$0.023	$0.30	$0.18–$0.25	$1.87	~$0.10
Cost[16] Throughput (1 MB/s for 1 month)	$0.03	$0.0071	$6.00	$3.15-$255.1	$0.96	~$0.03
Cost[16] IOPS (1/s for 1 month)	$0.03	$7.1	$0.23	$1-$3.15	$0.037	~$0.03

Storage Services: what is needed?

- The serverless ideal would provide cost and performance comparable to block storage, while adding:
- Access for cloud functions.
- Transparent provisioning (the storage equivalent of compute autoscaling)
- Different guarantees of persistence and availability.
- Guarantees for latency or other performance measures.

Coordination Mechanisms

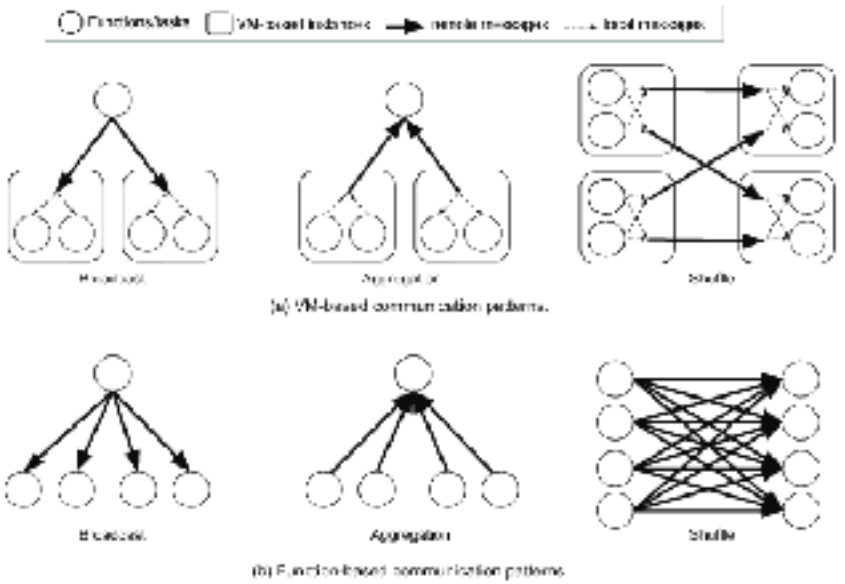
- To expand support to **stateful applications**, serverless frameworks need to provide a way for tasks to coordinate:
  - If task A uses task B's output there must be a way for A to know when its input is available, even if A and B reside on different nodes.
  - Many protocols aiming to ensure data consistency also require similar coordination.
- Cloud providers offer stand-alone notification services (SNS, SQS), but:
  - these services add significant latency, sometimes hundreds of milliseconds
  - can be costly when used for fine grained coordination.

Coordination: What is needed?

- Stateful applications are left with no choice but to:
  - Manage a VM-based system that provides notifications (ElastiCache and SAND) or
  - Implement own notification mechanism, that enables cloud functions to communicate with each other via a long-running VM-based rendezvous server.
- Explore new variants of serverless computing:
  - Naming function instances
  - Allowing direct addressability of functions, for access to their internal state (e.g., Actors as a Service)

Poor communication performance

- Most common communication patterns: Broadcast, aggregation, and shuffle.
  - Commonly employed by applications such as machine learning training and big data analytics.



- Each VM instance runs two functions/tasks.
- Note the significantly lower number of remote messages for the VM-based solutions:
  - VM instances offer ample opportunities to share, aggregate, or combine data locally across tasks before sending it or after receiving it.

Predictable Performance

- Cloud functions have a much lower startup latency than traditional VM-based instances.
- However, the delays incurred when starting new instances can be high for some applications, because of:
  - the time it takes to start a cloud function (can take less than 1 sec);
  - the time it takes to initialize the software environment of the function, e.g., load Python libraries (it might take tens of seconds to load all application libraries); and
  - application-specific initialization in user code.
- The latter two can dwarf the former.
- Variability in the hardware resources can result from giving the cloud provider flexibility to choose the underlying server.
  - A fundamental tradeoff between the cloud provider's desire to maximize their resources' use and predictability

Research Challenges

- System-level research opportunities:
  - Minimizing cold-starts while still scaling to zero
  - Containers vs Unikernels
- Legacy code in serverless
- Stateful serverless:
  - Will there be inherently stateful serverless applications in the future with different degrees of QoS without sacrificing the scalability and fault-tolerance properties?
- Service-level agreements (SLA): Serverless computing is poised to make developing services easier, but providing QoS guarantees remains difficult.
- Serverless at the edge: There is a natural connection between serverless functions and edge computing as events are typically generated at the edge with the increased adoption of IoT and other mobile devices.

Research Challenges

- **Abstraction challenges**
  - Resource requirements
  - Data dependencies
- System challenges
  - High-performance, affordable, transparently provisioned storage
  - Coordination/signalling service
  - Minimize startup time
- Networking challenges
- Security challenges
  - Scheduling randomization and physical isolation
  - Fine-grained security contexts
  - Oblivious serverless computing
- Computer architecture challenges
  - Hardware Heterogeneity, Pricing, and Ease of Management

Resource Requirements

- Developers can specify: cloud function's memory size and execution time limit.
- Often more control needed: specify number of CPUs, GPUs, accelerators.
- **Approach**: enable specification of extra resource requirements. However:
  - This adds more constraints on function scheduling: harder to achieve high utilization through statistical multiplexing
  - Increases the management overhead for cloud application developers.
- Other **alternative**: raise the level of abstraction, having the cloud provider infer resource requirements.
- **How?**
  - Static code analysis
  - Profiling previous runs
  - Dynamic(re)compilation to retarget the code to other architectures.
  - Provisioning just the right amount of memory automatically (hard in the presence of automatic garbage collection).

Data dependencies

- Cloud function platforms have no knowledge of:
  - data dependencies between the cloud functions
  - the amount of data exchange between these functions
- This can lead to suboptimal placement that could result in inefficient communication patterns.
- Approach: the cloud provider to expose an API that allows an application to specify its computation graph, enabling better placement decisions that minimize communication and improve performance.

Research Challenges

- Abstraction challenges
  - Resource requirements
  - Data dependencies
- System challenges
  - **High-performance, affordable, transparently provisioned storage**
  - Coordination/signalling service
  - Minimize startup time
- Networking challenges
- Security challenges
  - Scheduling randomization and physical isolation
  - Fine-grained security contexts
  - Oblivious serverless computing
- Computer architecture challenges
  - Hardware Heterogeneity, Pricing, and Ease of Management

Serverless Ephemeral Storage

- Why/where is it needed?
  - To transfer state between cloud functions and maintain application state during the application lifetime.
  - Once the application finishes, the state can be discarded. Such ephemeral storage might also be configured as a cache in other applications
- Possible approach: distributed in-memory service
- Expectations:
  - Microsecond-level latency.
  - Automatic scaling the storage capacity and the IOPS with application's demands.
  - Transparent allocation and release of memory.
  - Access protection and performance isolation across applications.
  - Lleverage statistical multiplexing, to provide improved memory-efficiency over today's serverful computing: any memory not used by one serverless application can be allocated to another.

Durable Storage

- Why/where is it needed?
  - Serverless databases
  - Serverless applications that require longer retention and greater durability than ephemeral storage, with low-latency, high IOPS and mutable-state semantics of a file system.
- Possible approach: leverage an SSD-based distributed store paired with a distributed in-memory cache, and novel non-volatile memory (NVM) - microsecond-level access times.
- Expectations / challenges:
  - Offer low tail latency in the presence of heavy tail access distributions, although in-memory cache capacity is likely to be much lower than SSD capacity.
  - Transparent provisioning.
  - Isolation across applications and tenants for security and predictable performance.
  - Explicit release of memory resources.
  - Durability, so that acknowledged writes survive failures.

Minimize Startup Time

- Three factors of startup time:
  1. scheduling and starting resources to run the cloud function
  2. downloading application software environment(e.g., operating system, libs) to run the function code,
  3. performing application-specific startup tasks such as loading and initializing data structures and libraries.
- Resource scheduling and initialization can incur significant delays and overheads from creating an isolated execution environment, and from configuring customer's VPC and IAM policies.
- Possible approaches for reducing factor 2:
  - Unikernels:
    - Preconfigured OS kernel based on hardware they are running on and statically allocating the data structures.
    - Include only the drivers and system libraries strictly required by the application => much lower footprint.
  - Load the libraries dynamically and incrementally as they are invoked by the application.
- Possible approaches for reducing factor 3:
  - Include a readiness signal in Cloud providers' API to avoid sending work to function instances before they can start processing it.
  - Seek to perform startup tasks ahead of time (maintain "warm pool" of VMs to be shared between tenants).

Research Challenges

- Abstraction challenges
  - Resource requirements
  - Data dependencies
- System challenges
  - High-performance, affordable, transparently provisioned storage
  - Coordination/signalling service
  - Minimize startup time
- **Networking challenges**
- Security challenges
  - Scheduling randomization and physical isolation
  - Fine-grained security contexts
  - Oblivious serverless computing
- Computer architecture challenges
  - Hardware Heterogeneity, Pricing, and Ease of Management

Networking Challenges

- What?
  - Cloud functions can impose significant overhead on popular communication primitives such as broadcast, aggregation, and shuffle.
- Possible approaches:
  - Provide cloud functions with a larger number of cores, so multiple tasks can combine and share data among them before sending over the network or after receiving it.
  - Allow the developer to explicitly place the cloud functions on the same VM instance. Offer distributed communication primitives that applications can use out-of-the-box so that cloud providers can allocate cloud functions to the same VM instance.
  - Let applications provide a computation graph, enabling the cloud provider to co-locate the cloud functions to minimize communication overhead.
- Cons?

Security Challenges

- Scheduling randomization and physical isolation, to avoid hardware-level side-channel attacks inside the cloud.
- Fine-grained security contexts required by Cloud functions:
  - Access to private keys, storage objects, and even local temporary resources.
  - Highly-expressive security APIs available for dynamic use: for example, a cloud function may have to delegate security privileges to another cloud function or cloud service.
  - More fine-grained security isolation for each function, as an option.
  - Share state between repeated function invocations to maintain a short startup time while providing function-level sandboxing.

Research Challenges

- Abstraction challenges
  - Resource requirements
  - Data dependencies
- System challenges
  - High-performance, affordable, transparently provisioned storage
  - Coordination/signalling service
  - Minimize startup time
- **Networking challenges**
- Security challenges
  - Scheduling randomization and physical isolation
  - Fine-grained security contexts
  - Oblivious serverless computing
- **Computer architecture challenges**
  - Hardware Heterogeneity, Pricing, and Ease of Management

Computer Architecture Challenges

- The end of Moore's law:
  - The x86 microprocessors that dominate the cloud are barely improving in performance.
  - In 2017, single program performance improvement only 3%. Assuming the trends continue, performance won't double for 20 years.
  - Similarly, DRAM capacity per chip is approaching its limits; 16 Gbit DRAMs are for sale today, but it appears infeasible to build a 32 Gbit DRAM chip.
- Performance problems for general purpose microprocessors **do not reduce the demand for faster computation**. There are two paths forward:
  - For functions written in high-level scriptingl anguages like JavaScript or Python, hardware-software co-design could lead to language-specific custom processors that run one to three orders of magnitude faster.
  - Domain Specific Architectures: tailored to a specific problem domain and offering significant performance and efficiency gains for that domain, but perform poorly for applications outside that domain: Tensor Processing Units (TPUs). TPUs can outperform CPUs by a factor of 30x.

Implications for Serveless

- Serverless could embrace multiple instance types, with a different price per accounting unit depending on the hardware used.
- The cloud provider could select language-based accelerators and DSAs automatically:
  - Implicitly based on the software libraries or languages used in a cloud function, e.g. GPU hardware for CUDA (Compute Unified Device Architecture) code and TPU hardware for TensorFlow code.
  - The cloud provider could monitor the performance of the cloud functions and migrate them to the most appropriate hardware the next time they are run.