THE UNIVERSITY OF CHICAGO

9. Distributed Data

MPCS: 52040 Distributed Systems

Kyle Chard

chard@uchicago.edu

Course schedule (* things might change a little)

Date	Lecture discussion topic	Class	Assessment
January 10	1. Introduction to Distributed Systems	Intro/Logistics	
January 17	2 Distributed architectures 3 Processes and virtualization	1 Docker	Homework 1 due
January 24	4 Networks and Communication	2 RPC/ZMQ/MPI	Homework 2 due
January 31	5 Naming	3 DNS/LDAP	Homework 3 due
February 7	6 Coordination and Synchronization		**Mid term exam**
February 14	7 Fault tolerance and consensus	Raft Project description	Homework 4 due (Project released)
February 21	8 Consistency and replication		
February 28	9 Distributed data	4 Distributed Data	
March 7	10 Data-intensive computing	5 FaaS	**Project due (March 9)** **Final exam (March 14)**

Agenda

- Part 1: RDBMS and NoSQL
- Part 2: File systems and shared memory

Part 2: Distributed File Systems and Shared Memory

File systems

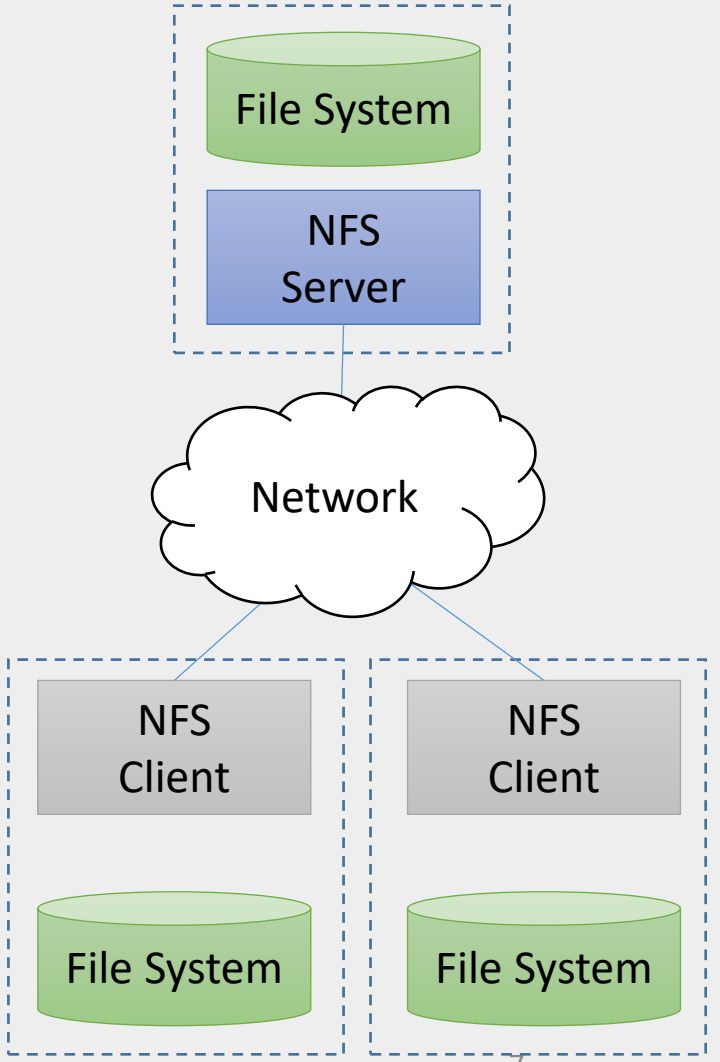
- Traditional file system
  - Implemented by a single system (e.g., ext3/4, FAT)
- Remote file systems
  - File system is served by a remote resource
  - E.g., NFS, AFS
- Parallel file system
  - File system is distributed across many nodes
  - Lustre, GPFS, PVFS (ANL)
- Distributed file system
  - Files are divided and replicated across nodes
  - GFS, HDFS

Remote file system

- Essentially files are stored on a remote server
- Clients can perform file system calls using RPC to the server
- Benefits
  - Data sharing between users
  - Location transparency (files are accessed as if they are local)
  - Backups for fault-tolerance

Network File System (NFS)

- Originally developed by Sun in 1984
- Client-server model
  - Server implements the shared file system and storage
  - Clients use RPC to perform common operations (list, read, write, etc.)
  - Caching at client and server for performance
- In Linux, users mount the remote file system so it looks like another path in their directory
- **Weak consistency**, client flushes updates on close().
- Clients periodically check if data has changed ➔ periods of inconsistency.
- No promises when multiple writes



Andrew File System (AFS)

- Named after founders of CMU
- Designed for institutional deployments – provide a file space to all client workstations
- Assumptions: most files are small, accessed by a single user, and reads happen much more often than writes
- More aggressive caching than NFS (lower server load than NFS but likely slower)
- **Weak consistency**
  - Files are cached locally and permanently on disk (to survive failures)
  - Read/writes directed to the local cache. When file is closed, changed portions are copied to the file server
  - Clients register with server that they have a copy of a file, server then sends cache invalidation messages if file changes (tradeoff server state vs better consistency)

File system in Userspace (FUSE)

- Software interface for Linux/Unix allowing non-privileged users to create file systems without editing the kernel

- Users simply write a handler program (in many languages) that implements the common file system operations

FUSE + Python

- Pip install fusepy

- Implement standard interfaces
  - def open(self, path, flags):
  - def read(self, path, size, offset, fh)
  - def write(self, path, data, offset, fh):

- Examples: https://github.com/fusepy/fusepy/blob/master/examples/memory.py

```
class Memory(LoggingMixIn, Operations):
    'Example memory filesystem. Supports only one level of files.'

    def __init__(self):
        self.files = {}
        self.data = defaultdict(bytes)
        self.fd = 0
        now = time()
        self.files['/'] = dict(
            st_mode=(S_IFDIR | 0o755),
            st_ctime=now,
            st_mtime=now,
            st_atime=now,
            st_nlink=2)
```

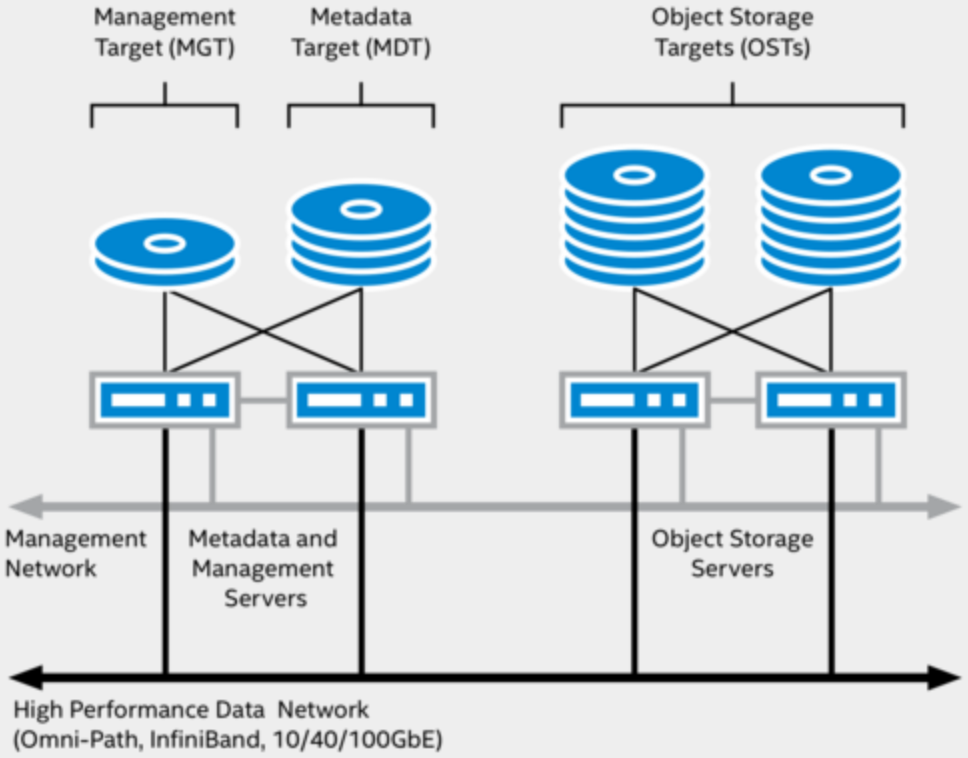
```
def mkdir(self, path, mode):
    self.files[path] = dict(
        st_mode=(S_IFDIR | mode),
        st_nlink=2,
        st_size=0,
        st_ctime=time(),
        st_mtime=time(),
        st_atime=time())

    self.files['/']['st_nlink'] += 1

def open(self, path, flags):
    self.fd += 1
    return self.fd

def read(self, path, size, offset, fh):
    return self.data[path][offset:offset + size]
```

Parallel file systems

- Parallel file systems store data across multiple networked servers
  - All servers are given a portion of the namespace
- Typically used by HPC environments that require access to large files, many files, or simultaneous access by many nodes/users
  - Ensure consistency across file system
- Unlike general distributed file systems, they generally require specialized client drivers to access at high speed (e.g., on infiniband)
- Benefits
  - Scale, performance, reliability

Parallel File System Examples

- IBM GPFS (renamed as Spectrum Scale)
  - Block-based file system, with blocks of tunable width and dynamic metadata
- Lustre
  - Open source, single namespace
  - POSIX interface
  - Object-based storage
  - Separated metadata stores
  - ANL 100PB stores (8,480 drives)
    - > TB/s throughput
  - OLCF – 679 PB file system



Google File System (GFS)

- Google has unique usage requirements
  - Applications: search, gmail, youtube, etc.
  - Enormous amounts of data must be stored and queried
  - Data creation and access rates are extreme
  - Lead to unique read/write patterns

- Google also has huge data centers (with commodity hardware)

- GFS is a specialized distributed fault-tolerant file system

https://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.pdf

GFS: Infrastructure

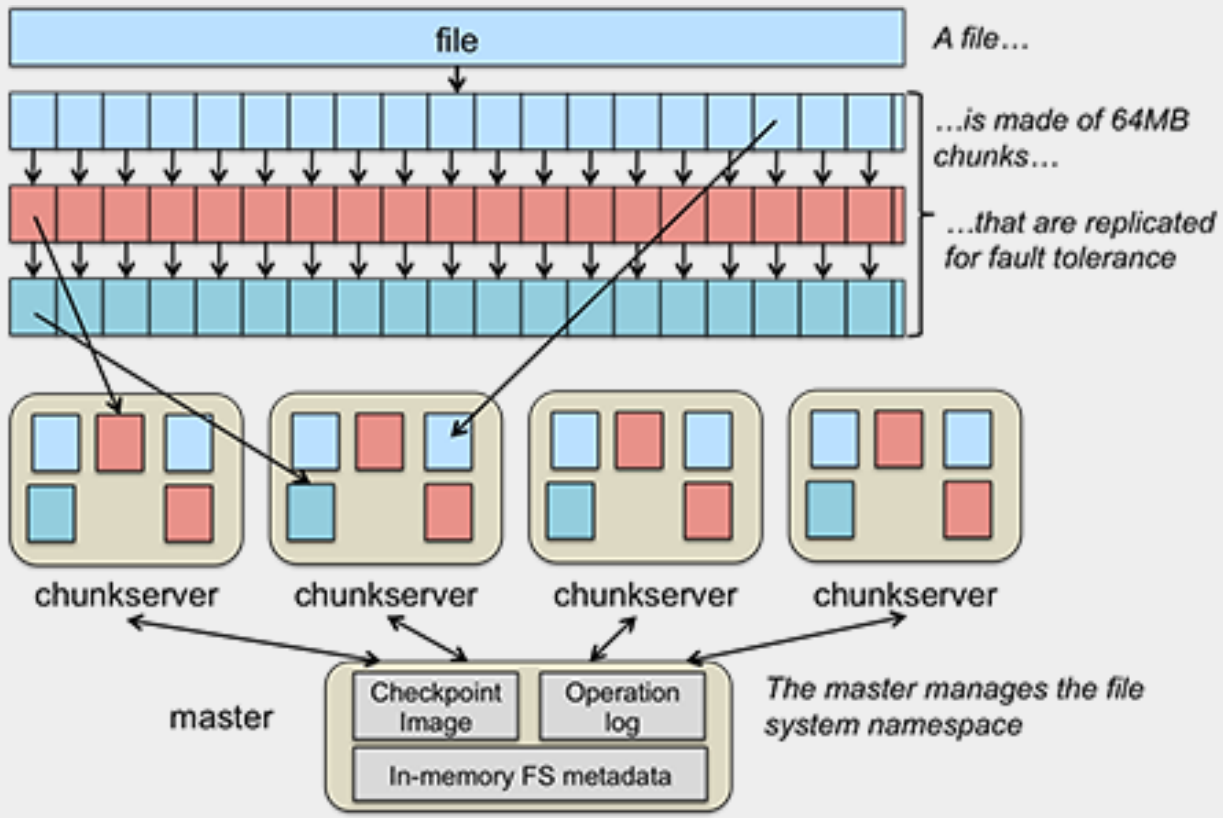
- Use of commodity computers (thousands or more)
- Each computer has many commodity disks
- Failures are to be expected
  - Need to design accordingly

GFS: Assumptions

- Modest number of large files
  - A few million files (>100MB)
  - Multi-GB files are common
  - Small files should be supported, but we do not need to optimize for them
- Workloads
  - Large streaming reads (read hundreds of KBs, or MBs)
    - Read from the same client normally contiguous region of a file
  - Small random reads (few KBs at arbitrary offset)
  - Large sequential writes that append data to files (KBs-MBs)
    - Once written, files are seldom modified
- Must support multiple clients that concurrently append to the same file
- High sustained bandwidth is more important than low latency
  - Applications want to process bulk data quickly (don’t have response requirements)

GFS: Basic architecture

- Files divided into fixed-size chunks (64MB)
  - Each assigned a unique 64-bit label
- Nodes are divided into two types
  - Master node
  - Many chunkservers (storing the chunks)
- Chunks are replicated several times (default 3, but configurable)
  - Popular files may have higher replication



GFS Master

- Manages metadata:
  - Namespace
  - Mapping from files to chunks
  - Current chunk locations
  - Access control information
- Stores metadata in RAM to provide fast operations
- Garbage collects chunks (those that are no longer associated with files) and manages chunk migration across chunkservers (for balancing)

GFS Chunkserver

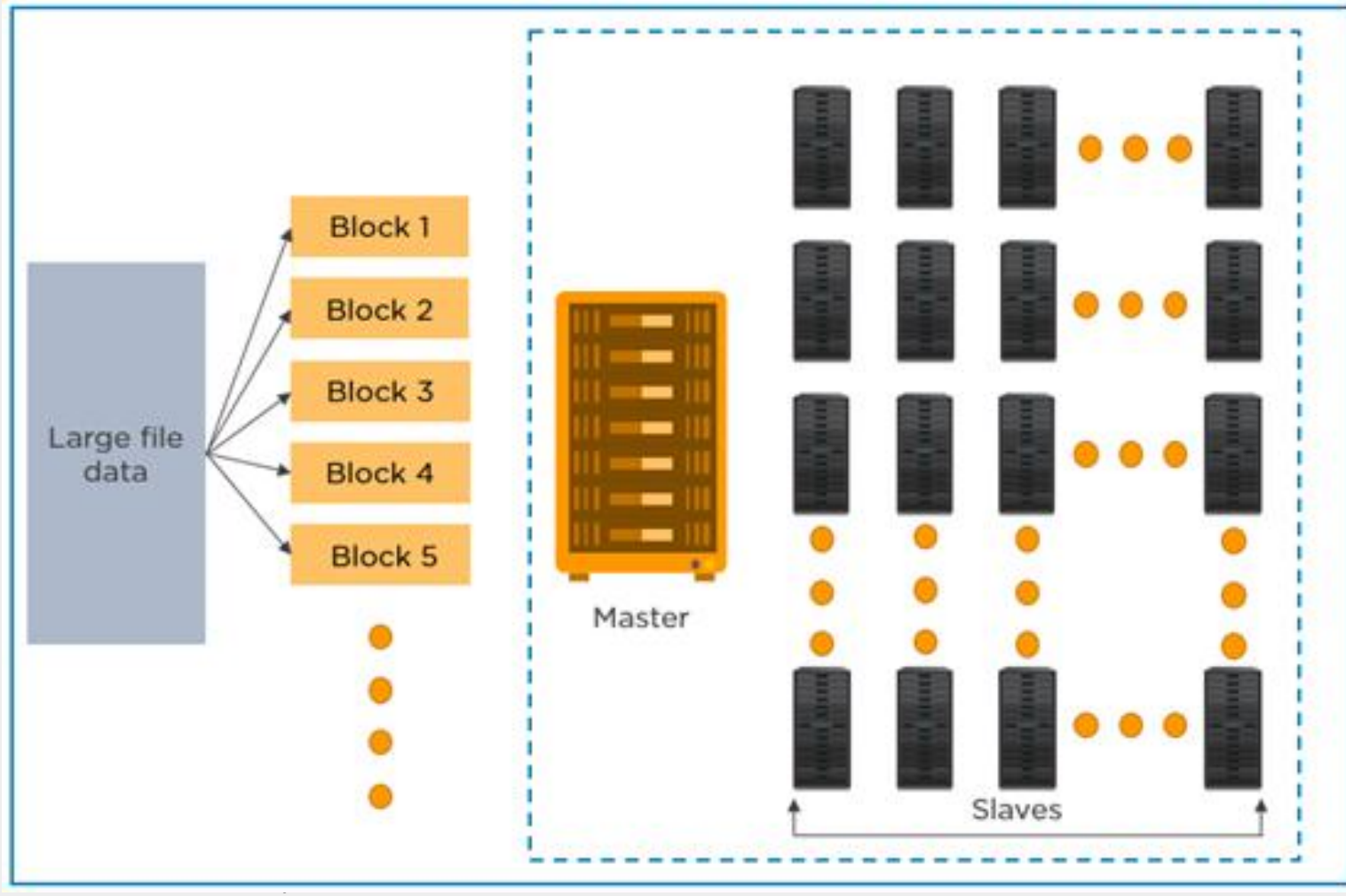
- Simple server that stores 64MB chunks on commodity disks using standard file system
  - 64MB? (much larger than typical file system block)
    - Reduces client need to interact with the master when operating within the same chunk
    - Reduce network overhead by keeping persistent connection with chunkserver
- Read/write requests operate on a specific chunk ID and byte range

GFS Consistency

- Relaxed consistency model to support highly distributed applications
- Metadata changes are atomic (e.g., file creation)
  - Managed by single master (using a log to establish order)
- Data changes on replicas are ordered following the order of the primary replica
  - Avoid the use of stale replicas in an operation (master wont point to them)
- Client caching may result in some stale data within a defined window period

Hadoop File System (HDFS)

- Designed for running MapReduce applications
  - Implemented in Java
  - Data store API (not POSIX)

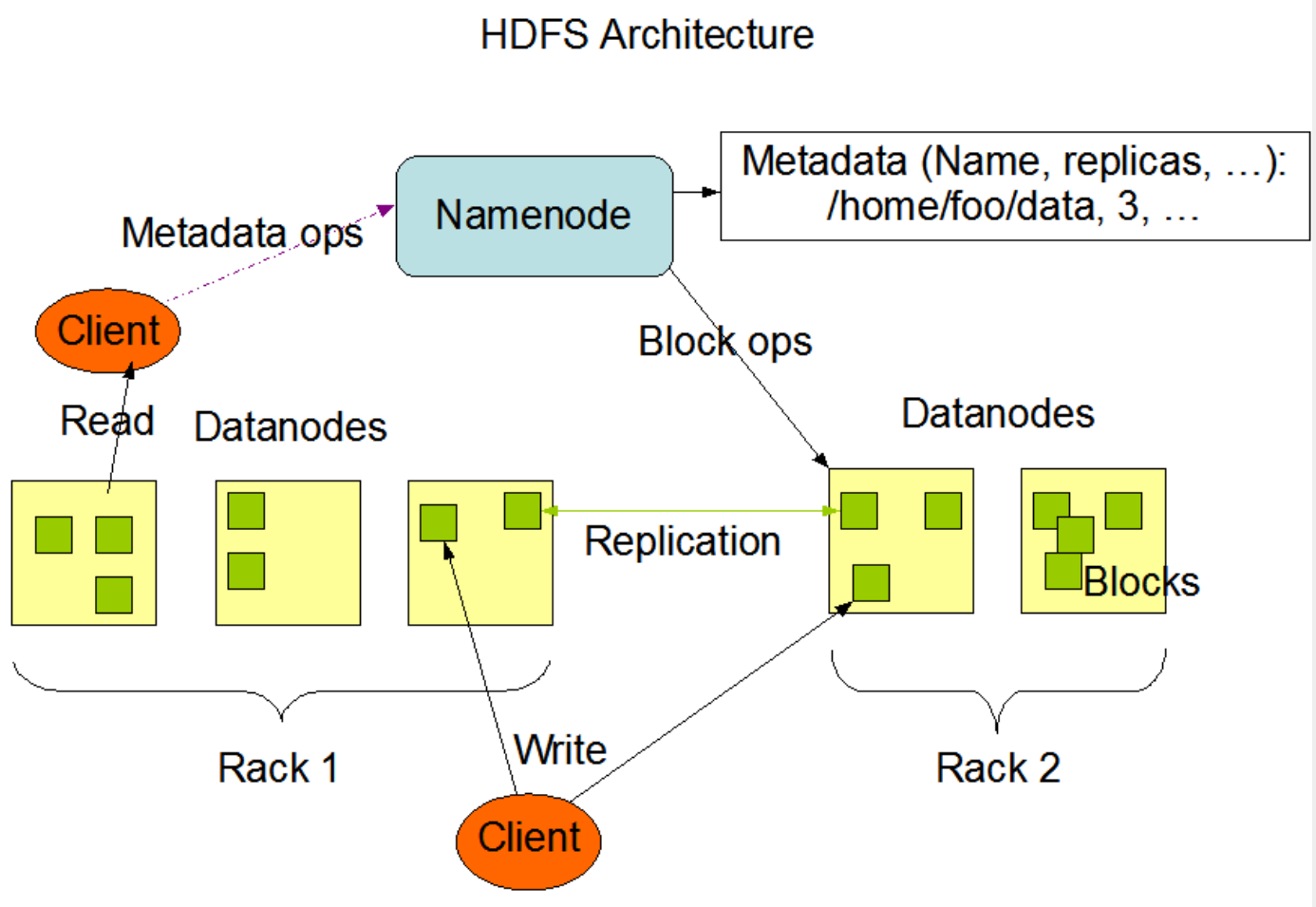


HDFS Overview

- Stores large files (GB-TB) across multiple machines
- Files divided into 128MB blocks
- Blocks are replicated across multiple hosts
- Designed for commodity hardware
- Failures are expected
- Designed for primarily immutable files (write once, read many times)
- Portable across hardware platforms and compatible with many OSs

HDFS Primary services

1. Master
    1. Name Node
    2. Job Tracker
    3. Task Tracker
2. Worker
    1. Data Node
    2. Task Tracker



HDFS name node

- Single master of the system

- Manages metadata and tracks file blocks
  - Number of blocks
  - Data node on which a block is stored
  - Management of replicas

HDFS Data node

- Designed to run on commodity machines
- Usually one data node per node in the cluster
- Manages local storage on that node
- Stores blocks (like chunks) on the data node file system

GFS vs HDFS

- Very similar (designed for slightly different purposes)
  - GFS: Google product suite
  - HDFS: Hadoop analyses
- Block sizes are twice as big in HDFS (although that can be configured)
- Some differences in protocols around chunk allocation/migration, integrity, deletion, snapshots, etc.

Google Colossus

- Google's storage requirements are changing (2010)
  - "You know you have a large storage system when you get paged at 1AM because you only have a few PB of storage left"

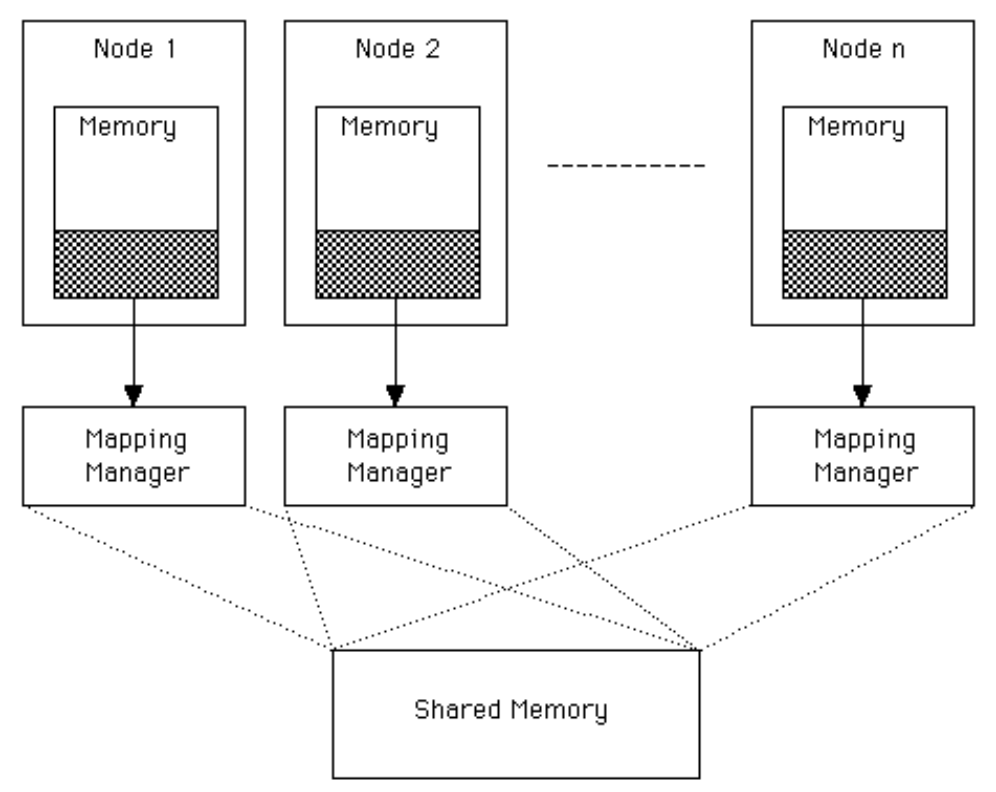
https://cloud.google.com/files/storage_architecture_and_challenges.pdf

Distributed Shared Memory

- Memory architecture in which physically separated memories can be addressed as one logically shared address space

- Simple primitives:
  - Read(address)
  - Write(address, data)

- Completely virtual (like virtual memory)



Summary

- When we scale across nodes we need to partition and distribute the file system
  - Parallel file system (e.g., Lustre) expand file system size with single global shared namespace and POSIX mounts to systems, usually uses
  - Distributed file system (e.g., GFS, HDFS) split up files across nodes for processing

- Shared memory provides a single system view of memory across nodes
  - Simple programming model when low latency networks are available

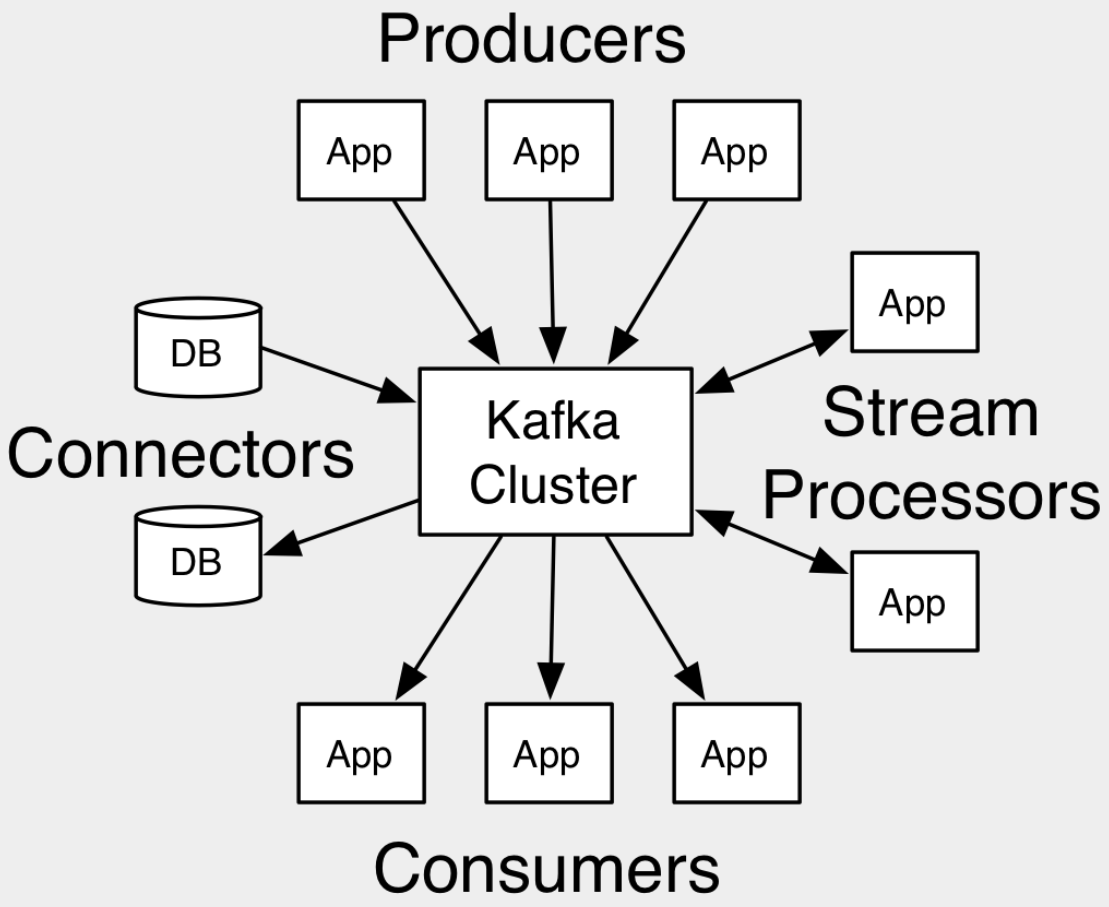
Streaming

- Continuous transfer of data at high speed (rather than batching with files)
- Real time data processing
- Sometimes memory to memory
- Sensors, market data, social media, app logs

Apache Kafka

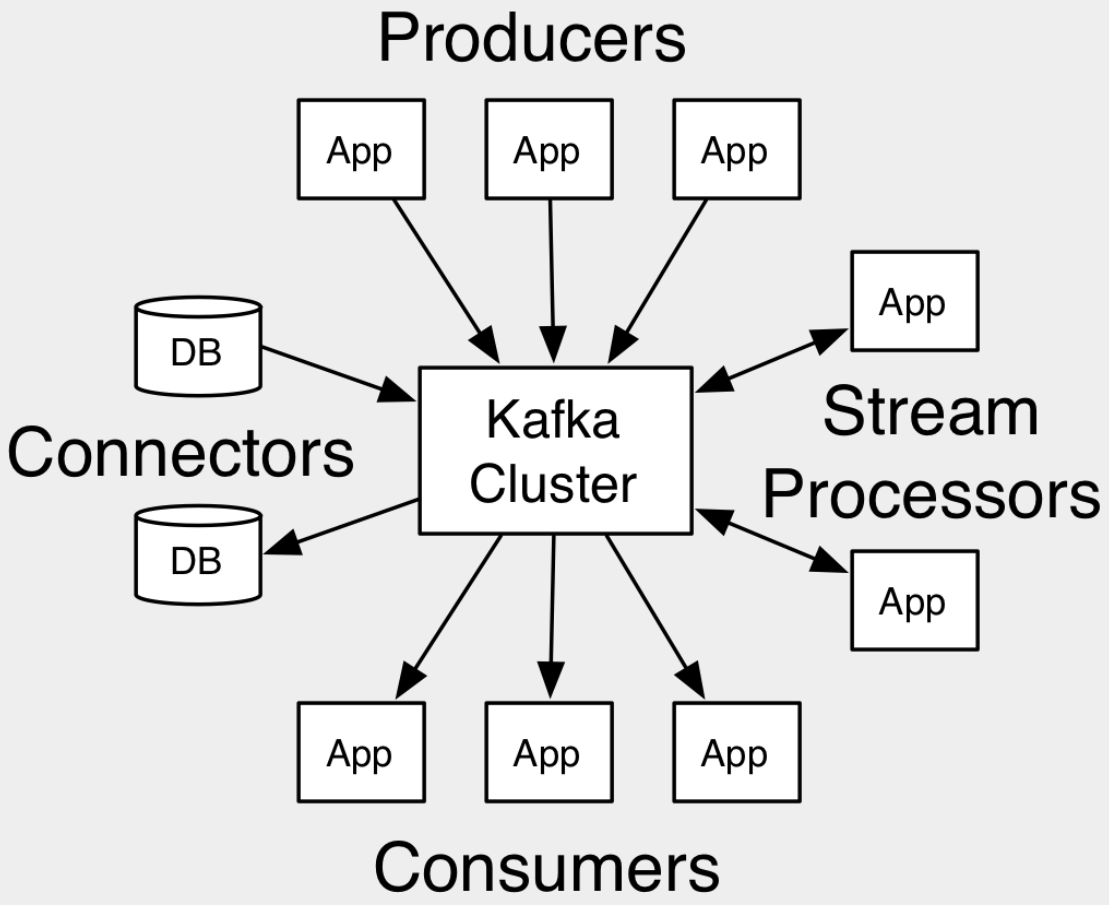
Kafka: distributed streaming platform

1. To **publish** (write) and **subscribe to** (read) streams of events, including continuous import/export of your data from other systems
2. To **store** streams of events durably and reliably for as long as you want
3. To **process** streams of events as they occur or retrospectively



Apache Kafka APIs

- Producer: allows app to publish a stream of records to *topics*
- Consumer: allows app to subscribe to topics and process records
- Streams: allows app to act as a stream processor, consuming input stream from one topic, applying transformation, and producing output stream
- Connector: connect Kafka topics to external apps or data systems



Events

- An **event** records the fact that "something happened" in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here's an example event:
  - Event key: "Alice"
  - Event value: "Made a payment of $200 to Bob"
  - Event timestamp: "Jun. 25, 2020 at 2:06 p.m."

Producers and Consumers

Producers are client applications that publish (write) events to Kafka, and **consumers** are applications that subscribe to (read and process) these events.

Producers and consumers are fully decoupled

- producers never need to wait for consumers.

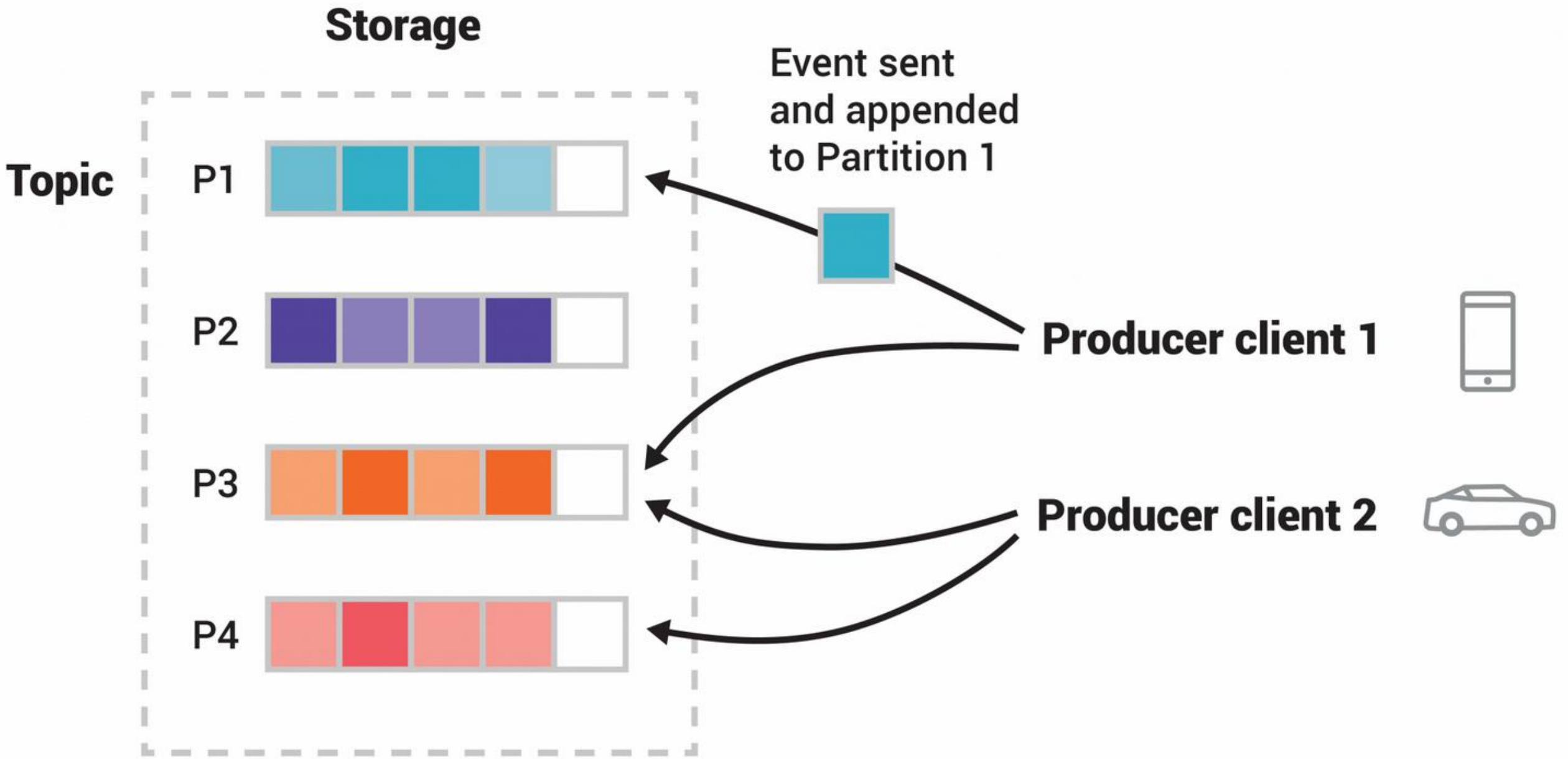
Kafka provides various guarantees e.g., process events exactly-once.

Topics

- Events are organized and durably stored in **topics**
  - Named feeds
- Topics are multi-producer and multi-subscriber
- Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption
- Users can define how long Kafka should retain events through a per-topic configuration, after which old events will be discarded.
  - Consumers can specify an offset from which to retrieve messages

Partitions

- Topics are **partitioned**, meaning a topic is spread over a number of "buckets" located on different Kafka brokers.
  - Scalability and performance (many client apps can read/write at the same time)
- When a new event is published to a topic, it is appended to one of the topic's partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka gurantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written.
- Topics can be **replicated** so that multiple brokers that have a copy of the data


Storage
Topic
P1
P2
P3
P4
Event sent
and appended
to Partition 1
Producer client 1
Producer client 2

Apache Kafka

- docker run -d -p 9092:9092 --name broker apache/kafka:latest
- Pip install kafka-python