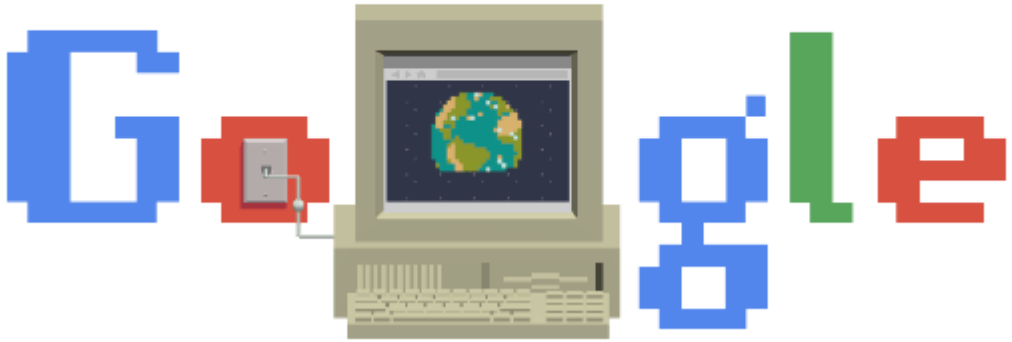
Z6110X0035:
Introduction to Cloud Computing
-Distributed File System

Lecturer: Prof. Zichen Xu

Recap from last Lecture

- Answer questions from Emails
- We have our first homework out
- The deadline is Mar 18th, 11:59PM, send your homework to ncuhomework@outlook.com
  - Fail to meet deadline or send your work to incorrect email will be discarded

30 years anniversary for WWW



The Web Foundation (@webfoundation)

30 years on, what's next for the web? Read @timberners_lee's letter on where we stand as a web community. It is up to us to fight for the web we want #ForTheWeb #Web30 webfoundation.org/2019/...

Ruben Verborgh (@RubenVerborgh)

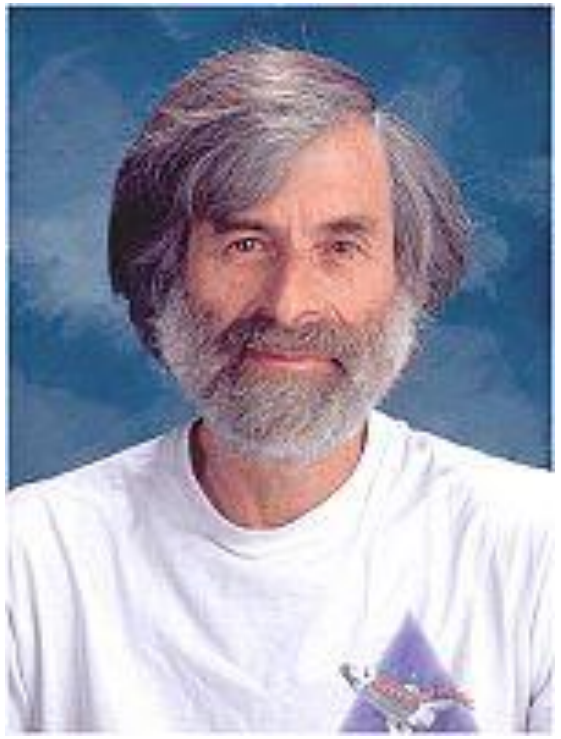
"The Web is for everyone and collectively we hold the power to change it. It won't be easy. But if we dream a little and work a lot, we can get the Web we want." webfoundation.org/2019/... Happy 30th birthday, #WorldWideWeb. Thanks, @timberners_lee and @W3C! #WebWeWant

John Allsopp (@johnallsopp)

Happy 30th Birthday World Wide Web. In honor of which I've made it super easy to contribute a milestone to my Web History project See project: www.webdirections.org/h... Contribute Milestone webdirections.typeform....

Warning from Leslie Lamport

- Who is he?
  - Leslie Lamport (born February 7, 1941 in New York City) is an American computer scientist. Lamport is best known for his seminal work in distributed systems and as the initial developer of the document preparation system LaTeX.
- What did he say?
  - A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.



Outline

- What is a distributed system?
- Build our Google File System
- So, how to build a distributed (file) system
- Course syllabus

What is a distributed system?

- Any system that has multiple processing units and connected by networks should be called as a distributed system.
  - Cluster, Internet, P2P system, Data Centers
  - Email, Distributed File System, MPI, MapReduce
- Even in a single machine, the multicore architecture can be considered as a distributed system.
- Normally the distributed systems are considered as machines (called nodes) connected by various kinds of network (wireless, low wired, LAN, high speed network).
- Connected computers do cooperative work and provide some services.

Why distributed?

- One single machine can not handle the problem: the scale is beyond the capability of a single machine even a very powerful machine.
- The reliability is an issue. Availability is problematic if only one machine is used. You will not get your information while the single machine is crashed.
- We want to get the information from all of the world thus we want the machines in the world to be connected.

Some issues that should be considered in distributed systems

- Scalability: Can the system balance the workload allover the provided computing resources?
- Availability: Can the system provide services while some components might crash.
- Consistency: Is the system able to provide the service correctly with some pre-defined criteria?
- Security: Can the system provide the privilege access control? Will the system leak information for unintended users?

Let’s see how Google builds its own distributed file system

- Let’s build a distributed file system following Google File System design.
- After this, you will see weather building a distributed system easy or not.

What should a file system provide?

- Directory operations
  - readdir
  - createdir
  - deletedir
  - deletefile
- File operations
  - createfile
  - openfile
  - readfile
  - writefile
  - closefile

What kind of special problems that Google faces?

- Google is building the largest web search engine
- Google needed a good distributed file system
  - Redundant storage of massive amounts of data on cheap and unreliable computers.

- Why not use an existing file system?
  - Google's problems are different from anyone else's.
    - Different workload and design priorities
  - GFS is designed for Google apps and workloads.
  - Google apps are designed for GFS

Assumptions

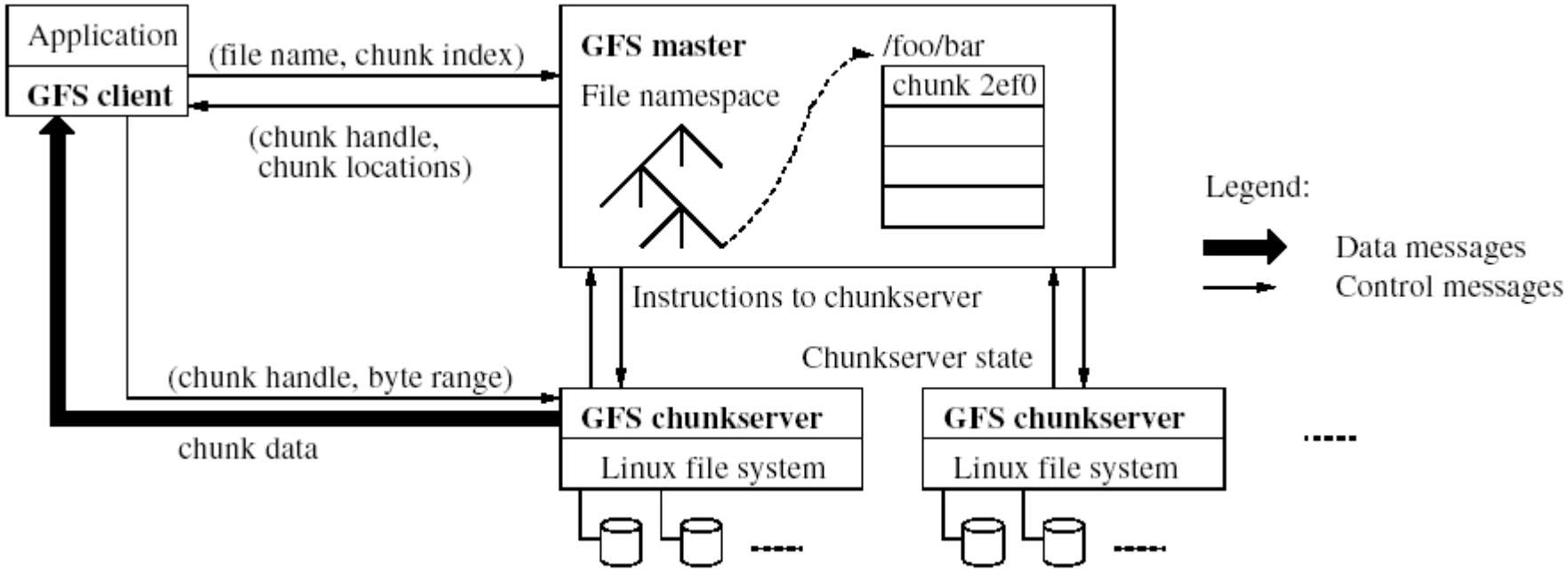
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of HUGE files
  - Just a few million
  - Each is 100MB or larger; multi-GB files typical
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads
- High sustained throughput favored over low latency

GFS Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ *chunkservers*
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large data sets, streaming reads
- Familiar interface, but customize the API
  - Simplify the problem; focus on Google apps
  - Add *snapshot* and *record append* operations

GFS Architecture

- Single master
- Mutiple chunkservers



Can any one see a potential weakness in this design?

Discussion

- How Google file system harness the full computing resources of thousands of machines?
- How Google file system can handle the failures of nodes or networks?
- Each chunk has more than three copies, how can they keep the same?

- Is a distributed system easy to build?

Steps before doing system building

- Perform the requirement analysis (what we will build, an email system or a file system)
- What kind of machines we have to use (mobile devices, desktops, servers, high performance MPPs)
- What kind of networks we have to build our system (wireless network, low speed network, High speed network(1Gbps Ethernet), InfiniBand

Take the distributed file system as an example

- Build the functions that must be provided to the users (File system interfaces)
- Build a system that can harness the full power of underlying building blocks(Scalability, thousands of machines)
- Build a system that never fail (Availability, despite the failure of components, the system can still provide the service)
- Build a system that never give wrong answer to users (Consistency, we have multiple copies and what should be the final answer?)
- Build a system that can tolerate unfriendly behavior (Security)

To build a workable distributed system is hard

- Complex: you have to consider a lot of components (Hardware, OS, users, interface, protocols)
- A lot of failures, (concurrent) node failure, network failure, byzantine failure, and even hacker penetration

Who should take this course?

- If you are interested in building a real distributed system
- If you are curious about the principles of building a distributed system
- If you want to do the similar things during your career

- Welcome to this course, you will get the ideas behind the distributed systems

Pre-requisite

- Undergraduate operating systems
- Programming experiences in C or C++
- Basic knowledge of computer networks

Course Organization

- Lectures
  - Papers and documents will be assigned using the network classroom system before each lecture
  - Each class will propose several questions, you are required to hand-in answers to those questions
  - This course follows the distributed system engineering course of MIT 6.824. You can find some information on their website.
- Labs
  - Build a distributed file system (yfs) that uses multiple processes as the distributed environment

Textbooks?

- No official textbook
- Papers that will be uploaded to the website before each class. You are required to read at least the introduction part of each paper or you well feel boring during the lectures.
- Reference books, not required but might be helpful
  - 1 Computer System: A Programmer’s Perspective. Randal E. Bryant and David R. O'Hallaron.
  - STL reference
  - Pthread programming
  - Socket programming

What you will learn from this course?

- Basic concepts and principles of distributed systems
  - System abstractions (abstraction might be the most important part while you think about build something useful)
  - some basic distributed algorithms that can be applied in practical situation (distributed algorithms might give you the sad part of the truth that you can not build a true system. However, we will use the bright part)
  - Fundamental techniques to build the distributed systems
- Analysis of true and (in)famous distributed systems
- Try to build a experimental distributed system through a serial of labs
  - This is quite important. You wont truly understand the principles until you build something that can work.

Contents of this course

- The concepts of distributed systems
- Programming for distributed systems
- Consistency
- Fault Tolerant
- Large Scale Data Processing Paradigm
- Case Studies

- Labs

Concepts of Distributed Systems

- Organization of Distributed Systems (cluster, peer-to-peer, cloud)
- Availability
- Scalability
- Consistency
- Security
- Safety and Liveness of Distributed Algorithms

Consistency

- Distributed Shared Memory System Introduction
- Sequential consistency, Release Consistency, Lazy Release Consistency
- Eventual Consistency
- Transactions and All-or-Nothing Atomicity
- Current Practical Consistencies analysis
- Concurrency Control

Fault Tolerant

- Crash Recovery and Logging
- Two Phase Commit
- Consensus, Replicated State Machine
- Paxos

Large Scale Data Processing Paradigm

- MPI (concept introduction)
- MapReduce Programming
- Dryad
- Spark and Shark
  - Memory matters

Case Studies

- Distributed File System and Google File System
- Consistency Systems: Linearability, Sequential Consistency, Parallel Snapshot Consistency, Causal+ Consistency, Eventual Consistency. Systems related all to those consistencies.
- Distributed Database and No-SQL database systems
- ……

Single Master

- From distributed systems we know this is a:
  - Single point of failure
  - Scalability bottleneck
- GFS solutions:
  - Shadow masters
  - Minimize master involvement
    - never move data through it, use only for metadata
      - and cache metadata at clients
    - large chunk size
    - master delegates authority to primary replicas in data mutations (chunk leases)
- Simple, and good enough!

Metadata (1/2)

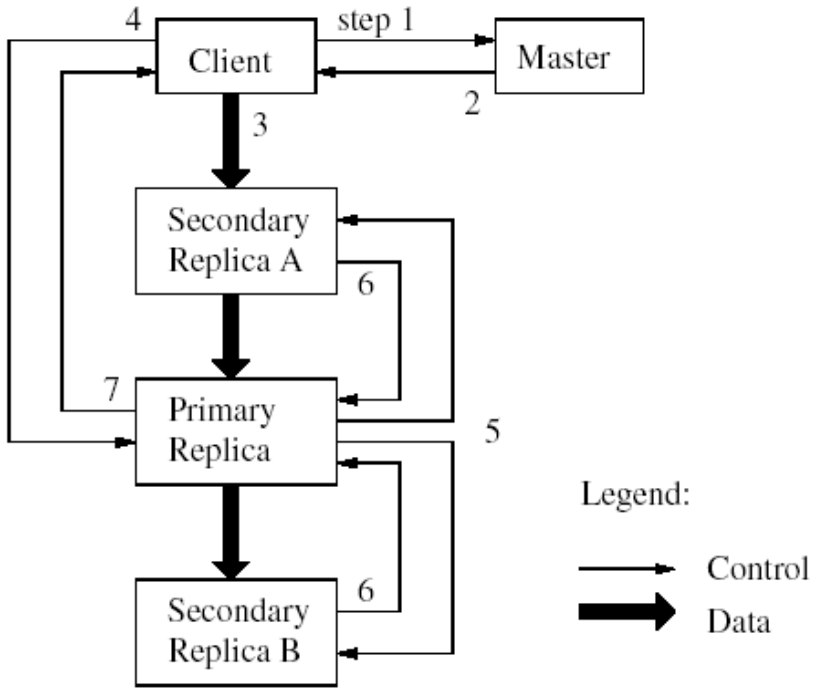
- Global metadata is stored on the master
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk’s replicas
- All in memory (64 bytes / chunk)
  - Fast
  - Easily accessible

Metadata (2/2)

- Master has an *operation log* for persistent logging of critical metadata updates
  - persistent on local disk
  - replicated
  - checkpoints for faster recovery

Mutations

- Mutation = write or append
  - must be done for all replicas
- Goal: minimize master involvement
- Lease mechanism:
  - master picks one replica as primary; gives it a “lease” for mutations
  - primary defines a serial order of mutations
  - all replicas follow this order
- Data flow decoupled from control flow



Atomic record append

- Client specifies data
- GFS appends it to the file atomically at least once
  - GFS picks the offset
  - works for concurrent writers
- Used heavily by Google apps
  - e.g., for files that serve as multiple-producer/single-consumer queues

Relaxed consistency model (1/2)

- “Consistent” = all replicas have the same value
- “Defined” = replica reflects the mutation, consistent
- Some properties:
  - concurrent writes leave region consistent, but possibly undefined
  - failed writes leave the region inconsistent
- Some work has moved into the applications:
  - e.g., self-validating, self-identifying records

Relaxed consistency model (2/2)

- Simple, efficient
  - Google apps can live with it
  - what about other apps?
- Namespace updates atomic and serializable

Master’s responsibilities (1/2)

- Metadata storage
- Namespace management/locking
- Periodic communication with chunkservers
  - give instructions, collect state, track cluster health
- Chunk creation, re-replication, rebalancing
  - balance space utilization and access speed
  - spread replicas across racks to reduce correlated failures
  - re-replicate data if redundancy falls below threshold
  - rebalance data to smooth out storage and request load

Master’s responsibilities (2/2)

- Garbage Collection
  - simpler, more reliable than traditional file delete
  - master logs the deletion, renames the file to a hidden name
  - lazily garbage collects hidden files
- Stale replica deletion
  - detect “stale” replicas using chunk version numbers

Fault Tolerance

- High availability
  - fast recovery
    - master and chunkservers restartable in a few seconds
  - chunk replication
    - default: 3 replicas.
  - shadow masters
- Data integrity
  - checksum every 64KB block in each chunk

Distributed File System

- Don’t move data to workers… move workers to the data!
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- Why?
  - (Perhaps) not enough RAM to hold all the data in memory
  - Data center network is slow.
  - Disk access is slow, but disk throughput is reasonable
- A distributed file system is the answer
  - GFS (Google File System) for Google’s MapReduce
  - HDFS (Hadoop Distributed File System) for Hadoop

GFS: Assumptions

- Commodity hardware over “exotic” hardware
  - Scale “out”, not “up”
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of huge files
  - Multi-gigabyte files are common, if not encouraged
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads over random access
  - High sustained throughput over low latency

GFS slides adapted from material by (Ghemawat et al., SOSP 2003)

GFS: Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit due to large datasets, streaming reads
- Simplify the API
  - Push some of the issues onto the client (e.g., data layout)

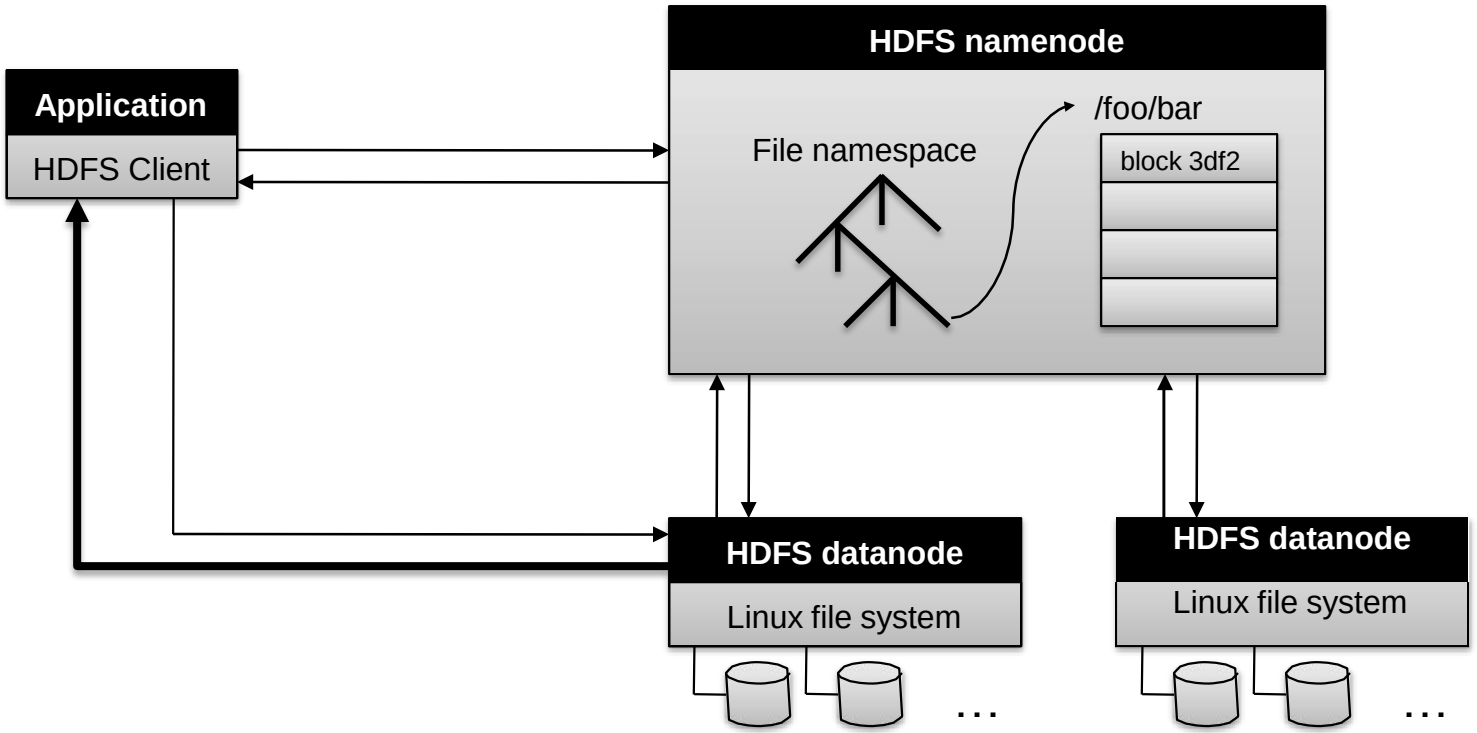
HDFS = GFS clone (same basic ideas)

From GFS to HDFS

- Terminology differences:
  - GFS master = Hadoop namenode
  - GFS chunkservers = Hadoop datanodes
- Differences:
  - Implementation
  - Performance

For the most part, we'll use Hadoop terminology…

HDFS Architecture

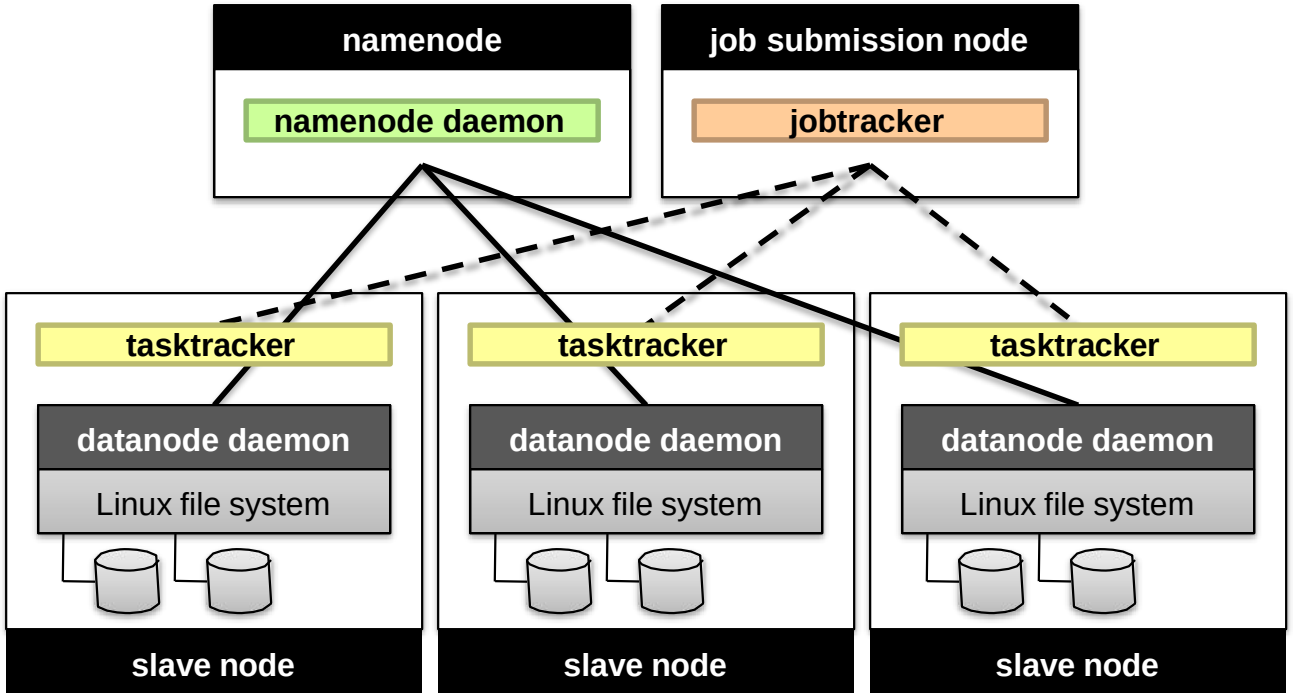


Adapted from (Ghemawat et al., SOSP 2003)

Namenode Responsibilities

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection

Putting everything together…



(Not Quite… We'll come back to YARN later)

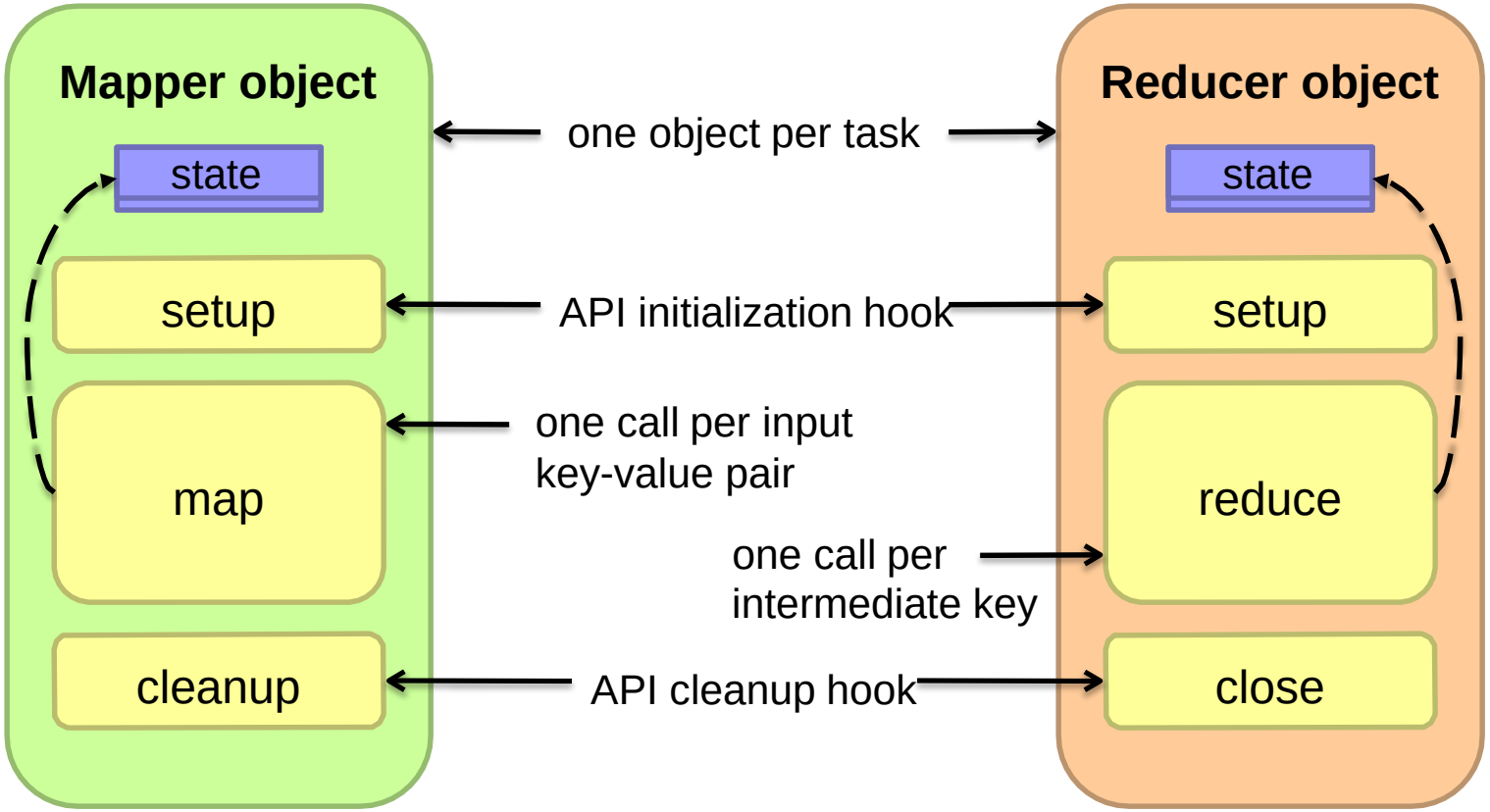
Basic Algorithm Design with MapReduce

- Limited control over data and execution flow
  - All algorithms must expressed in *mapper()*, *reducer()*, *combiner()*, *partitioner ().*
- You don’t know:
  - Where mappers and reducers run
  - When a mapper or reducer begins or finishes
  - Which input a particular mapper is processing
  - Which intermediate key a particular reducer is processing

Tools for Synchronization

- Cleverly-constructed data structures
  - Bring partial results together
- Sort order of intermediate keys
  - Control order in which reducers process keys
- Partitioner
  - Control which reducer processes which keys
- Preserving state in mappers and reducers
  - Capture dependencies across multiple keys and values

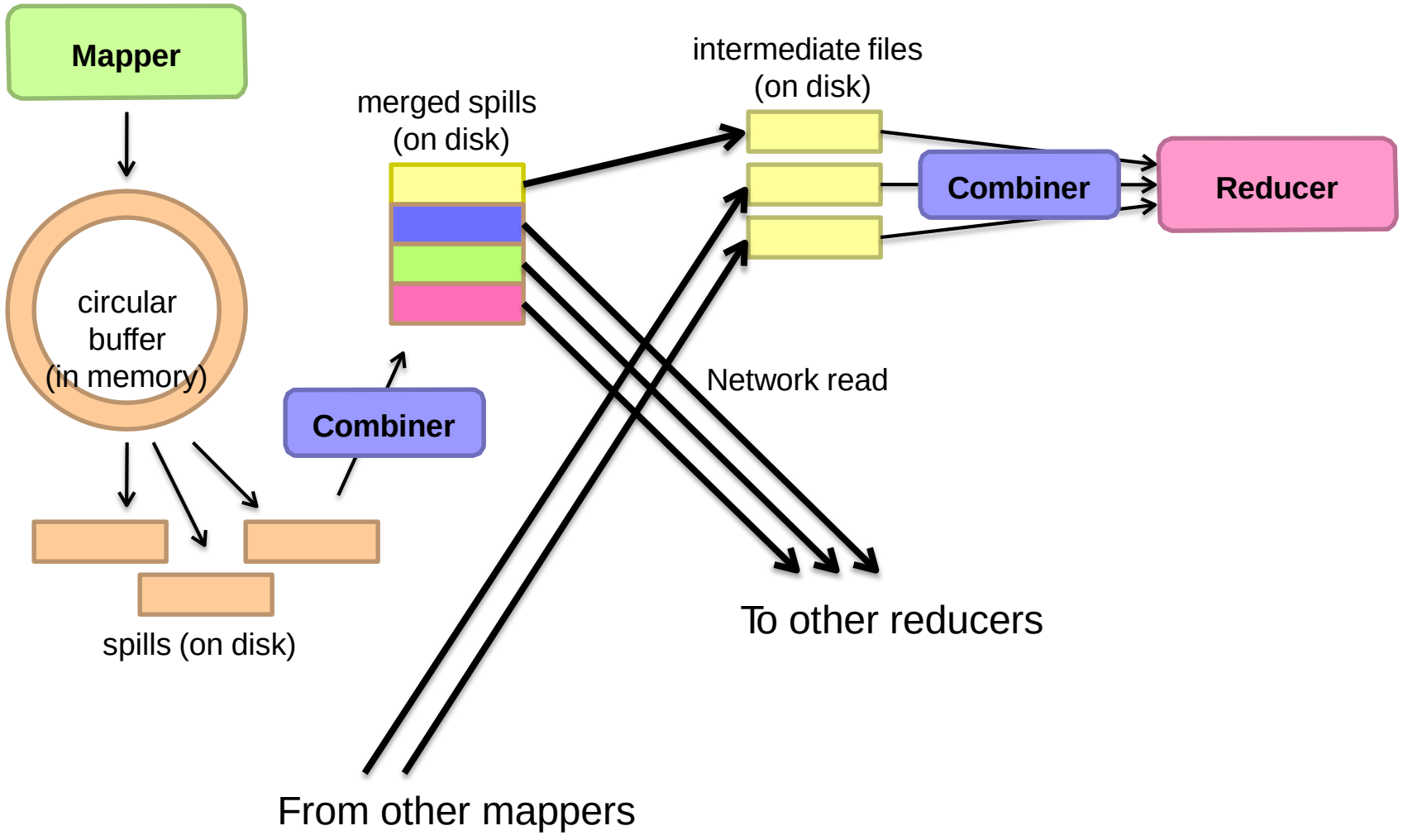
Preserving State



Importance of Local Aggregation

- Ideal scaling characteristics:
  - Twice the data, twice the running time
  - Twice the resources, half the running time
- Why can’t we achieve this?
  - Synchronization requires communication
  - Communication kills performance
- Thus… avoid communication!
  - Reduce intermediate data via local aggregation
  - Combiners can help

Shuffle and Sort



Word Count: Baseline

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         for all term t ∈ doc d do
4:             Emit(term t, count 1)

1: class Reducer
2:     method Reduce(term t, counts [c1, c2, ...])
3:         sum ← 0
4:         for all count c ∈ counts [c1, c2, ...] do
5:             sum ← sum + c
6:         Emit(term t, count s)
```

What's the impact of combiners?

Word Count: Version 1

```
1: class Mapper
2:     method Map(docid a, doc d)
3:         H ← new AssociativeArray
4:         for all term t ∈ doc d do
5:             H{t} ← H{t} + 1                ▷ Tally counts for entire document
6:         for all term t ∈ H do
7:             Emit(term t, count H{t})
```

Are combiners still needed?

Word Count: Version 2

```
1: class Mapper
2:     method Initialize
3:         H ← new AssociativeArray
4:     method Map(docid a, doc d)
5:         for all term t ∈ doc d do
6:             H{t} ← H{t} + 1                ▷ Tally counts across documents
7:     method Close
8:         for all term t ∈ H do
9:             Emit(term t, count H{t})
```

Key idea: preserve state across input key-value pairs!

Are combiners still needed?

Design Pattern for Local Aggregation

- Version 2 uses “In-mapper combining”
  - Fold the functionality of the combiner into the mapper by preserving state across multiple map calls
- Advantages
  - Speed
  - Why is this faster than actual combiners?
- Disadvantages
  - Explicit memory management required
  - Potential for order-dependent bugs

Combiner Design

- Combiners and reducers can share same method signature
  - Sometimes, reducers can serve as combiners
  - Often, not…
- Remember: combiner are optional optimizations
  - Should not affect algorithm correctness
  - May be run 0, 1, or multiple times
- Example: find average of integers associated with the same key (*see tutorial*)

Further Reading

- "Operating Systems: A Concept-based Approach,2E" Chapter 19
  - https://books.google.com/books?id=kbBn4X9x2mcC&pg=PA725&dq=distributed+file+system&hl=en&sa=X&ved=0ahUKEwjQ0saK5fvgAhUNv54KHTXzBkUQ6AEIUzAJ
- Shvachko, Konstantin, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The hadoop distributed file system." In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pp. 1-10. Ieee, 2010. http://www.alexanderpokluda.ca/coursework/cs848/CS848%20Paper%20Presentation%20-%20Alexander%20Pokluda.pdf
- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung: The Google File System
  - http://labs.google.com/papers/gfs.html

Take-home Message

- Distributed File System supports large business and entities for modern cloud services.
- Hadoop and other tools upon help to build large streaming data analytic jobs.
- We learnt the basic programming model and underground work for Cloud Computing upon.
  - Next time, virtualization and computational virtualization

Questions?


Celebrating
30 YEARS
Embracing the Past and
Building the Future