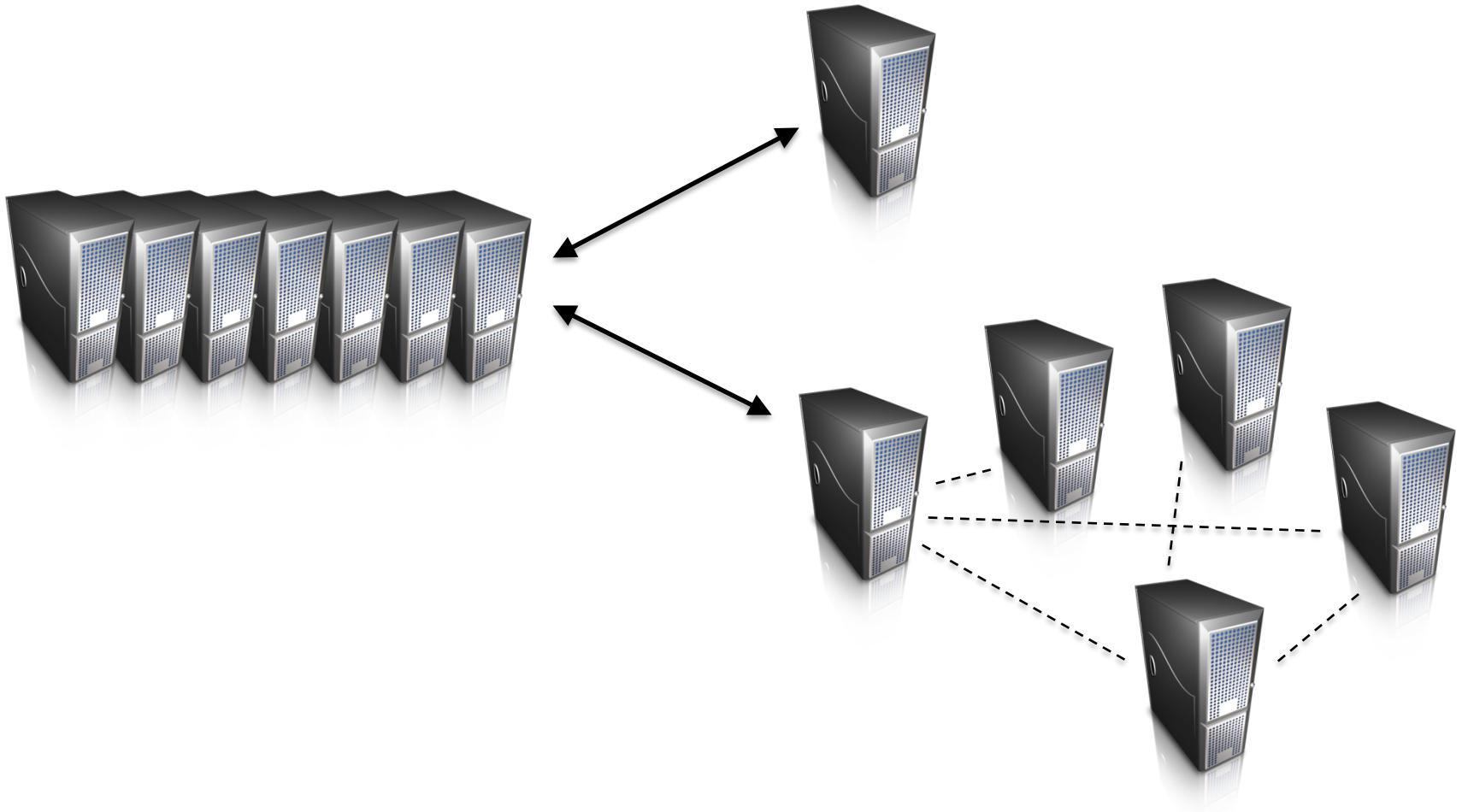
Distributed File System

How do we get data to the workers?



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?
  - Not enough RAM to hold all the data in memory
  - 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

Features

- Highly fault-tolerant
  - Failure is the norm rather than exception
- High throughput
  - May consist of thousands of server machines, each storing part of the file system's data.
- Suitable for applications with large data sets
  - Time to read the whole file is more important than the reading the first record
  - Not fit for
    - Low latency data access
    - Lost of small files
    - Multiple writers, arbitrary file modifications
- Streaming access to file system data
- Can be built out of commodity hardware

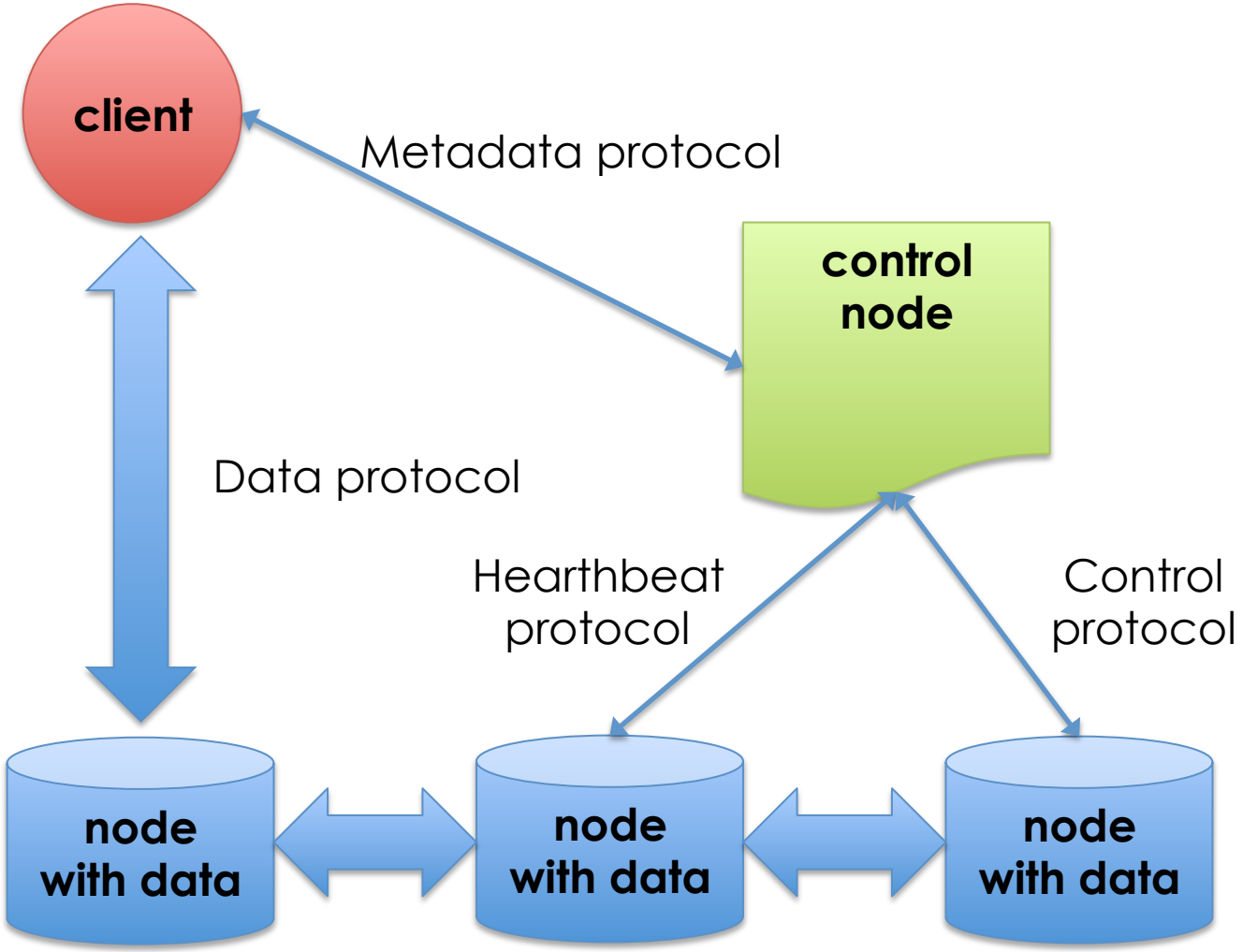
Blocks

- Minimum amount of data that it can read or write
- File System Blocks are typically few KB
- Disk blocks are normally 512 bytes
- HDFS Block is much larger – 64MB by default
  - Unlike file system the smaller file does not occupy the full 64MB block size
  - Large to minimize the cost of seeks
  - Time to transfer blocks happens at disk transfer rate
- Block abstractions allows
  - Files can be larges than block
  - Need not be stored on the same disk
  - Simplifies the storage subsystem
  - Fit well for replications
  - Copies can be read transparent to the client

Name Nodes and Data Nodes

- Master/slave architecture
- DFS cluster consists of a single **name node**, a master server that manages the file system namespace and regulates access to files by clients.
- There are a number of **data nodes** usually one per node in a cluster.
- The data nodes manage storage attached to the nodes that they run on.
- DFS exposes a file system namespace and allows user data to be stored in files.
- A file is split into one or more blocks and set of blocks are stored in data nodes.
- Data nodes serve read, write requests, perform block creation, deletion, and replication upon instruction from name node.

DFS Architecture



Google File System

We will review the Google DFS implementation, highlighting the differences with Hadoop DFS on the way.

Slides based on

S. Ghemawat, H. Gobioff, and S.-T. Leung, Google Inc.
The Google File System
SOSP'03, October 19–22, 2003, Bolton Landing, New York, USA.

http://labs.google.com/papers/gfs-sosp2003.pdf

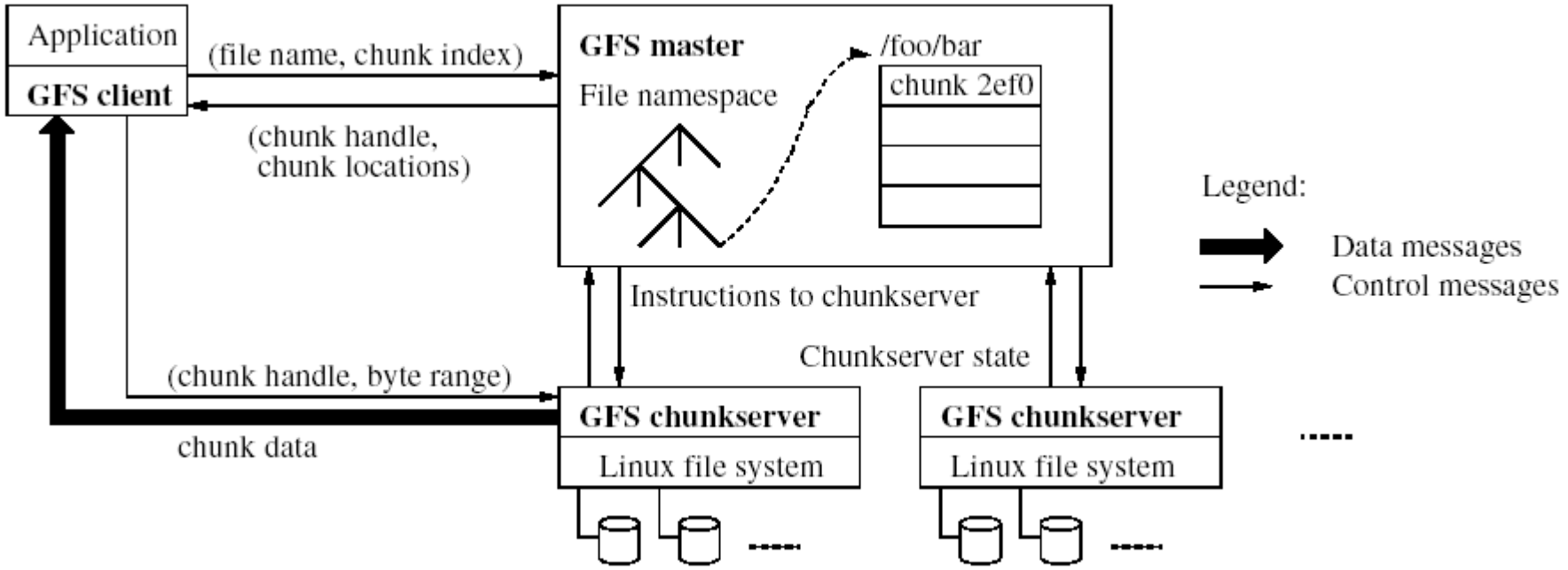
Operational Scenario

- High component failure rates
  - 1000s low cost commodity parts
  - Inevitable software bugs
- Huge files
  - Few files
  - Size greater than 100 MB (typically many GB)
- Read/write semantics
  - Many sequential reads, few random reads
  - Write once then append
  - Caching is not so appealing
  - Multiple writers
- High throughput better than low latency

Design Decisions

- Files are stored in chunks (blocks)
  - typical size: 64 MB
- Simple centralized management
  - master server (namenode)
  - metadata
- Reliability through replication
  - Each chunk replicated across 3+ chunkservers (datanodes)
- No data caching
  - metada caching in the master server
- Custom API
  - Easy to use, but no POSIX-compliant
  - create, delete, open, close, read, write
  - snapshot, record append

Architecture



Picture taken from the referenced paper (see slide #2)

Master server

- Scalability bottleneck
  - Clients never read and write file data through the master
  - Large chunk size reduces:
    - clients' need to interact with the master
    - network overhead by keeping a persistent TCP connection
    - the size of the metadata stored on the master
    - but hot spots with executables
- Single point of failure
  - Persistent, replicated operation log (secondary namenode)

Master responsabilities

- Store and manage metadata
- Manage and lock namespace
- Periodic chunkservers communication
  - Issue commands
  - Collect status
  - Track health
- Replica management
  - Create/delete/monitor/replicate/balance chunks
  - Garbege collection (deleted & stale replicas)

Metadata

- Three major types of metadata
  - the file and chunk namespaces
  - the mapping from files to chunks
  - the locations of each chunk's replicas

operation log

- All metadata is kept in the master's memory
  - Fast periodic scanning
- Memory limit
  - 64 bits per chunk (64MB)
  - Filenames compressed

Anatomy of a read

1. Client sends to master
   - read(filename)
2. Master replies to client
   - (chunk ID, chunk version, replicas locations)
3. Client (namenode) selects closest replica
   - IP-based inspection (rack-aware topology)
4. Client sends to chunkserver
   - read(chunk ID, byte range)
5. Chunkserver replies with data
6. In case of errors, client proceeds with next replica, remembering failed replicas

Anatomy of a write (GFS)

1. The client asks the master
   - which chunkserver holds the current lease for the chunk
   - the locations of the other replicas
   - If no one has a lease, the master chooses a replica
2. The client pushes the data to all the replicas
3. Replicas acknowledge receiving the data
4. The client sends a write request to the primary replica
5. The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients
6. The primary applies the write to its own local state in serial number order
7. The primary forwards the write request and order to all secondary replicas
8. The secondary replicas reply to the primary indicating that they have completed the operation
9. The primary replies to the client.
10. In case of error the write is failed and the client must handle the inconsistencies (retry or fallback)

Anatomy of a write (HDFS)

1. In HDFS, the data push is performed by data nodes in pipeline
2. In case of error, if at least a replica is correct, the system is going to build asynchronously the missing replicas
3. The current release of HDFS supports writing new files only.

Fault Tolerance

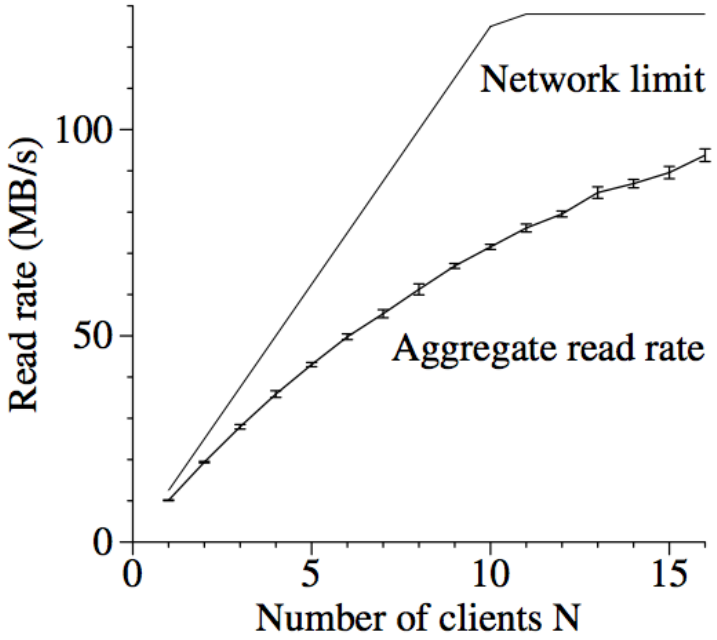
- High availability
  - Fast (re)start(up) of every component
  - Replication
  - Operation log
  - Shadow masters
- Data Integrity
  - Chunkservers checksumming
  - Optimized for append operations

Benchmarks

- Machine configuration
  - 1 Master, 2 Master replicas, 16 chunk servers, 16 clients
  - Dual 1.4 Ghz, P3, 2GB, 2 80Gb 5400 rpm disks, 100 Mbps full-duplex to Hp switch
  - 19 GFS machines are connected to one switch
  - 16 clients machines to other switch
  - Two switches are connected with a 1 Gbps link

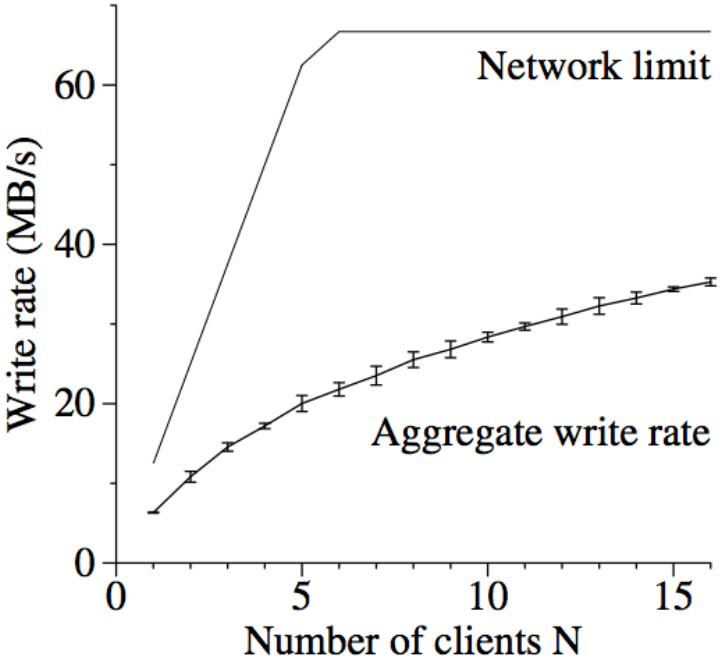
Performance - Read

- 4MB region from a 320GB file set
- Repeat random 256 times to read 1 Gig of data
- Limit
  - 125 Mbps when 1Gps link is saturated or
  - 12.5 Mbps per client on 100Mbps network
- Measured
  - 10Mbps or 80% of the limit
  - Aggregate reaches 94Mbps
  - Efficiency drops from 80% to 75% as the number of readers increases



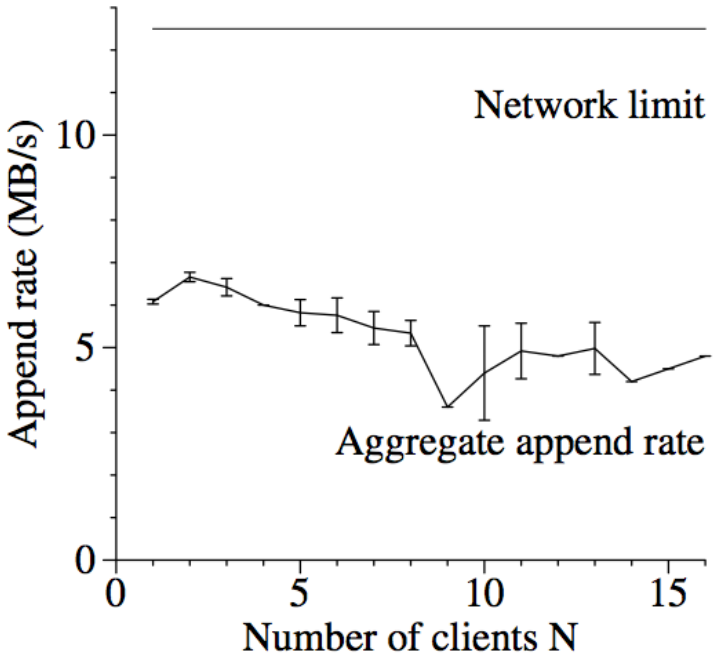
Performance - Write

- N clients write simultaneously to N distinct files.
- Each client writes 1 GB of data to a new file in a series of 1 MB writes.
- Write rate is 6.3Mbps about half of the limit
- Aggregate rate reaches to 35Mbps for 16 clients or 2.2 Mbps



Performance - Append

- N clients append simultaneously to a single file
- Performance is limited by network bandwidth of chunkservers independent of no of chunk servers
- Starts at 6.0 MBps drops to 4.8Mbps due to congestion



Lab Requirements

- Linux-compliant machine
- Java 6 SDK installed
- SSH and SSHD installed and running
- Editor installed

- Hadoop 0.20.2 distribution downloaded from http://hadoop.apache.org/mapreduce/releases.html