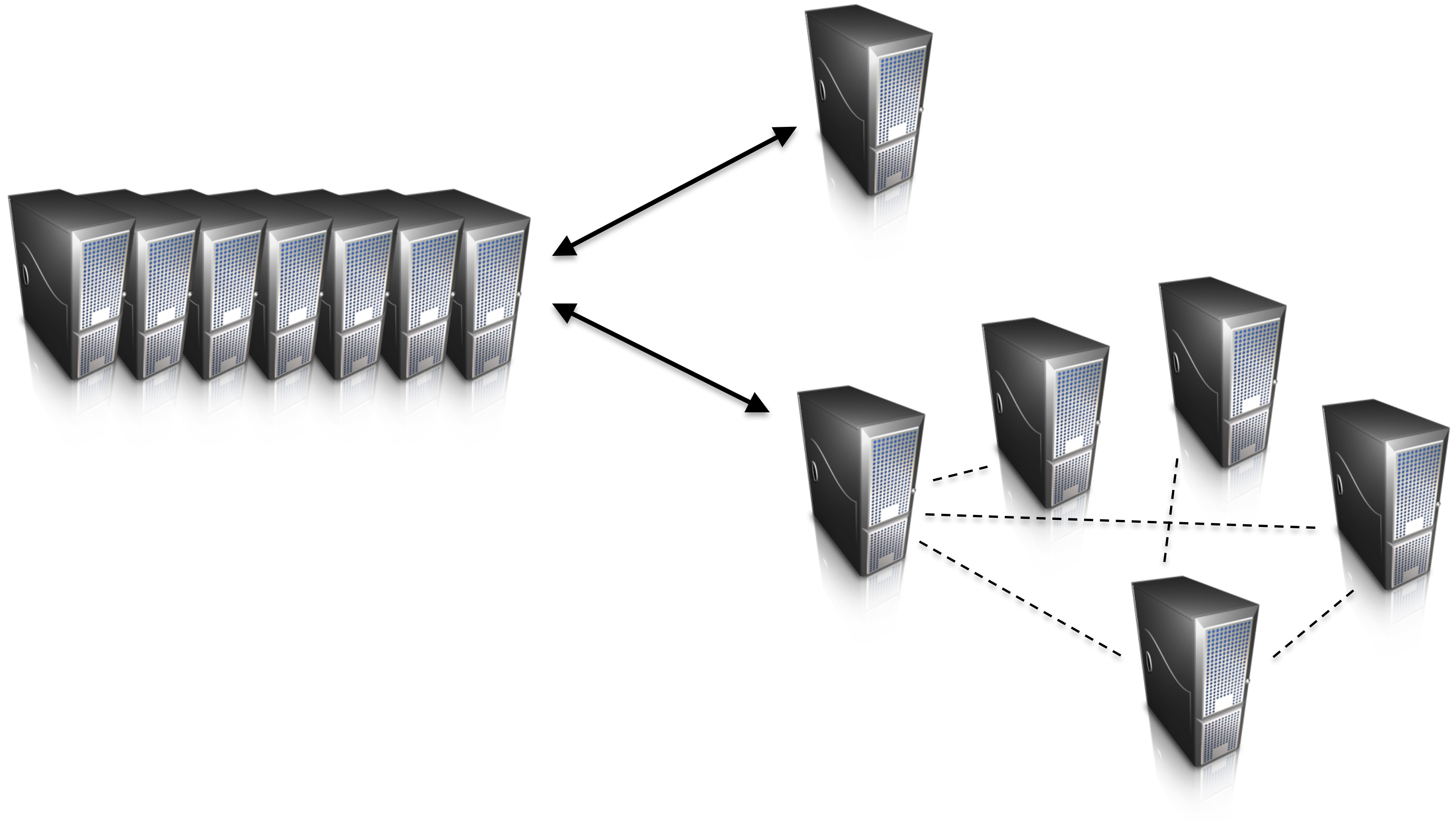
Distributed Filesystem

How do we get data to the workers?



Distributing Code!

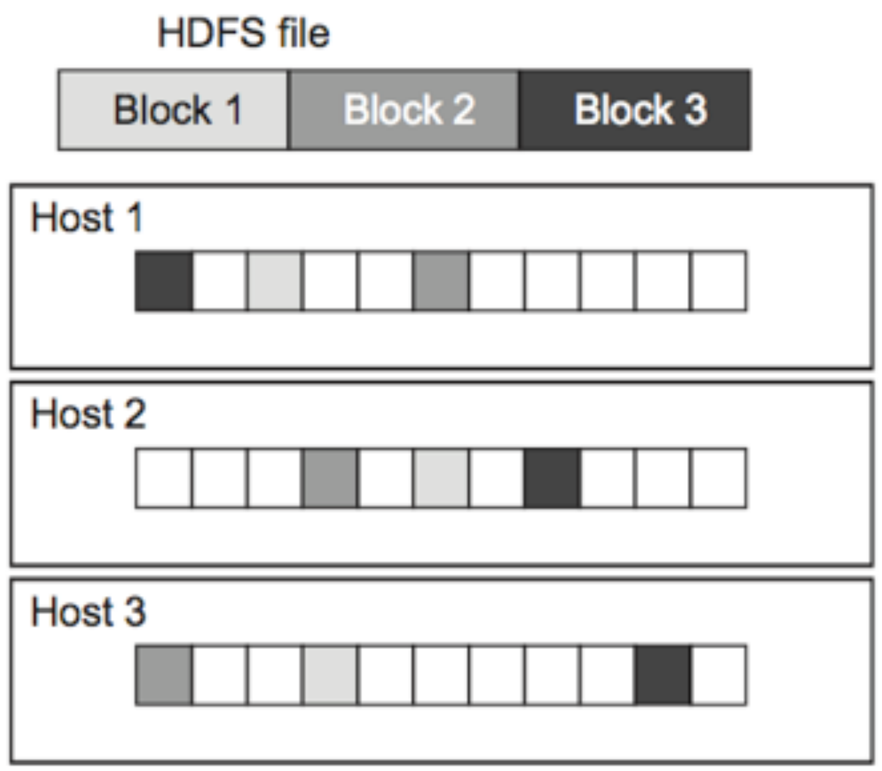
- **Don't move data to workers… move workers to the data!**
  - Store data on the local disks of nodes in the cluster
  - Start up the workers on the node that has the data local
- **Why?**
  - Not enough RAM to hold all the data in memory
  - Disk access is slow, but disk throughput is reasonable
-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

Requirements/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

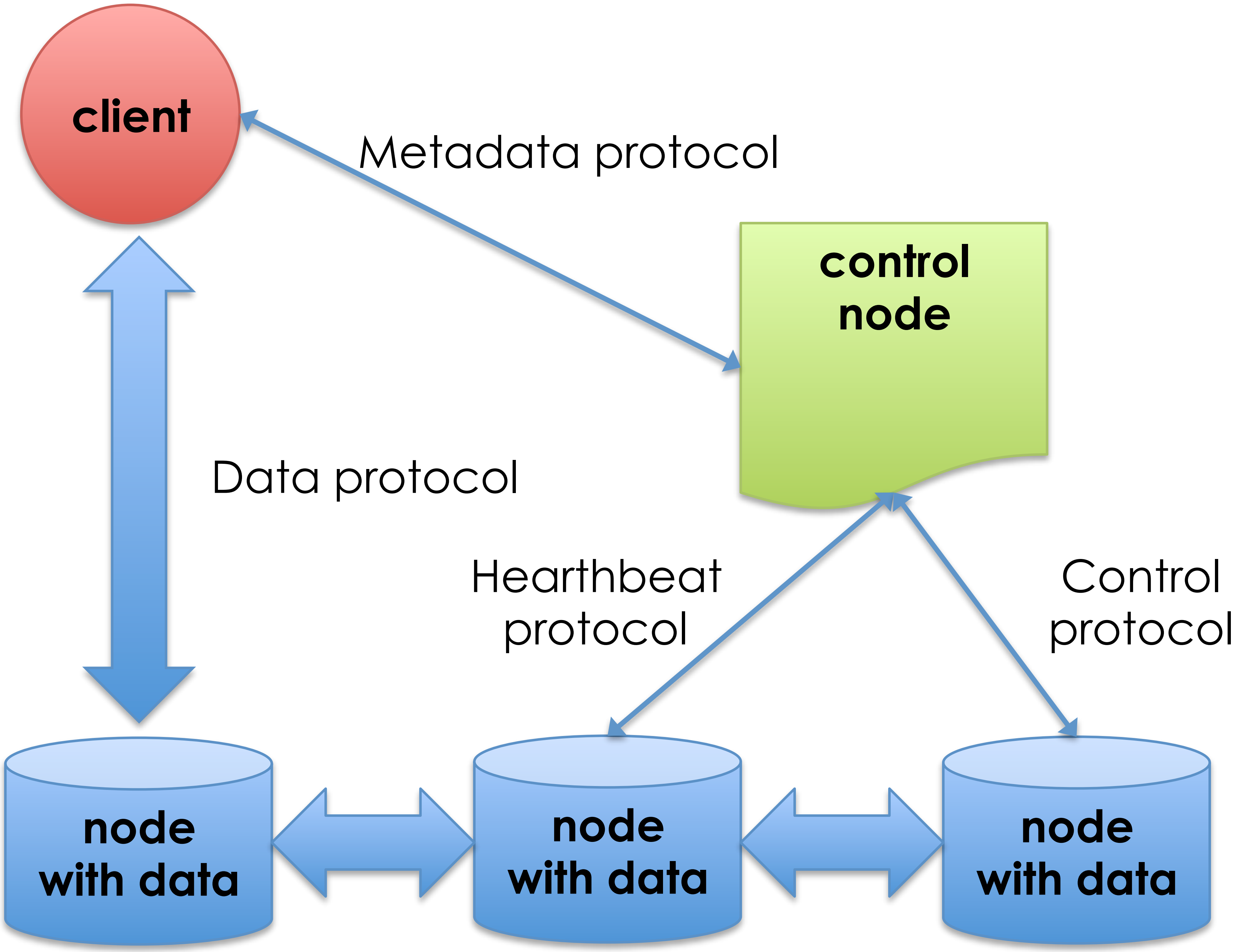
- **Minimum amount of data that it can read or write**
- **File System Blocks are typically few KB**
- **Disk blocks are normally 512 bytes**
- **HDFS Block is much larger – 64 MB by default**
    - Unlike file system the smaller file does not occupy the full 64MB block size
    - Large to minimize the 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



Namenodes and Datanodes

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

DFS Architecture



Disclaimer

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

Operational Scenario

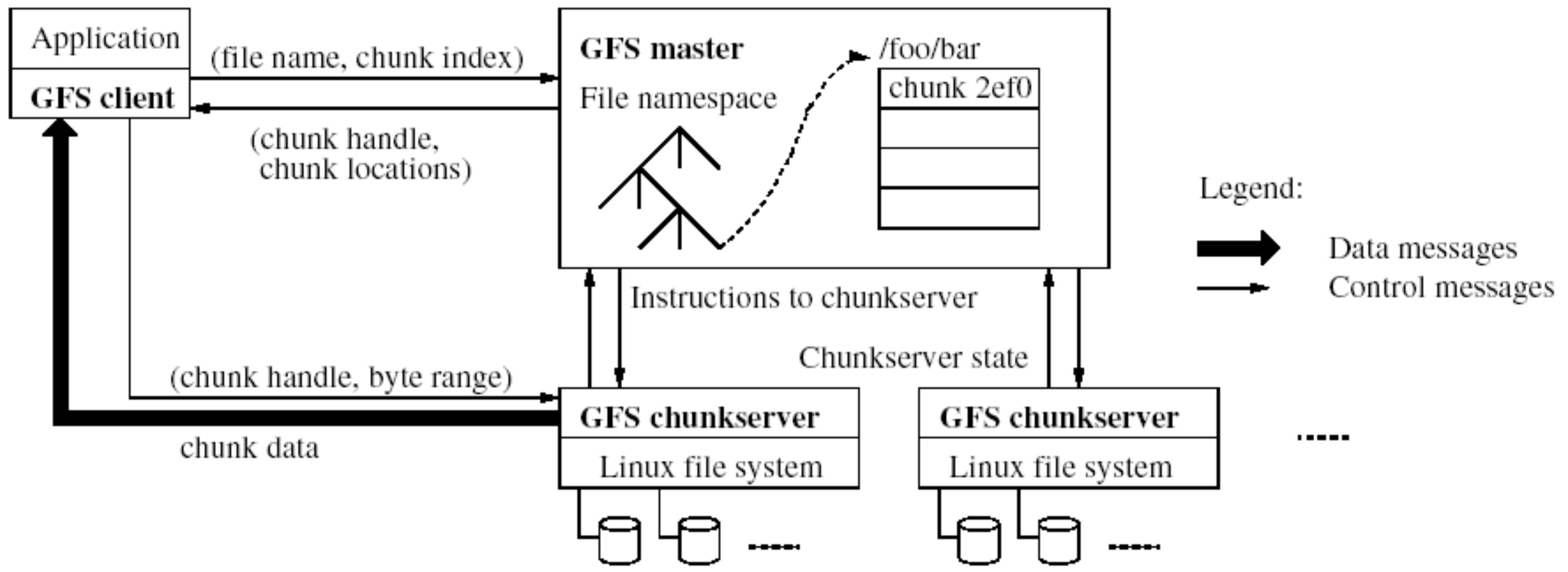
- High component failure rates
  - 1000s low cost commodity parts
  - Inevitable software bugs
- Huge files
  - Few files
  - Size greater than 100 MB (typically many GB)
- Read/write semantics
  - Many sequential reads, few random reads
  - Write once then append
  - Caching is not so appealing
  -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

ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

Design Choices

- 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+ chunk servers (**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

GFS Architecture



ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

mercoledì 21 novembre 12

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
  - Garbage collection (deleted & stale replicas)

Master Issues

- 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**)

Metadata

- Three major types of metadata

$$\left.\begin{array}{l}\text{- the file and chunk namespaces}\\ \text{- the mapping from files to chunks}\end{array}\right\}\ \text{Operation Log}$$

  - the locations of each chunk's replicas
- All metadata is kept in the master's memory
  - Fast periodic scanning
- Memory limit
  - 64 bits per chunk (64 MB)
  - 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

Rack-aware Topology

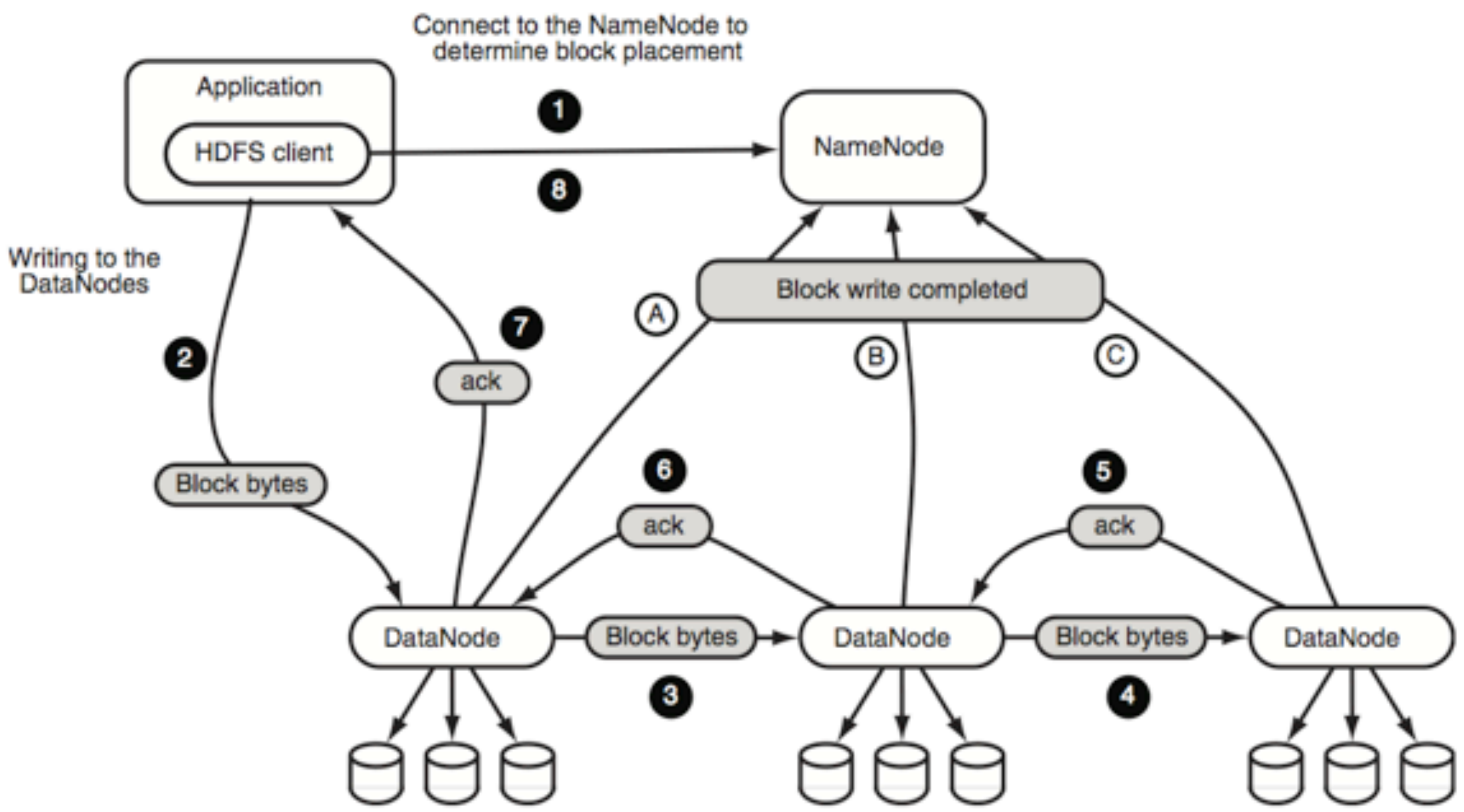
- DFS stores files across one or more blocks, so for each block the namenode is consulted to determine which datanodes will hold the data for the block.
- Nodes are prioritized in the following order:
  1. **Local disk** — Network I/O is expensive, so reading from local disk is always preferred over any network I/O.
  2. **Rack-local** — Typically network speeds between nodes in a rack are faster than across racks.
  3. **Other** — The data is resident on a separate rack, which makes it the slowest I/O of the three nodes (generally due to the additional network hops between the client and data).
- In a rack-aware environment, the namenode will ensure that at least one copy of the replica is on a rack separate from the other replicas.

Anatomy of a write

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)

Writing in HDFS...

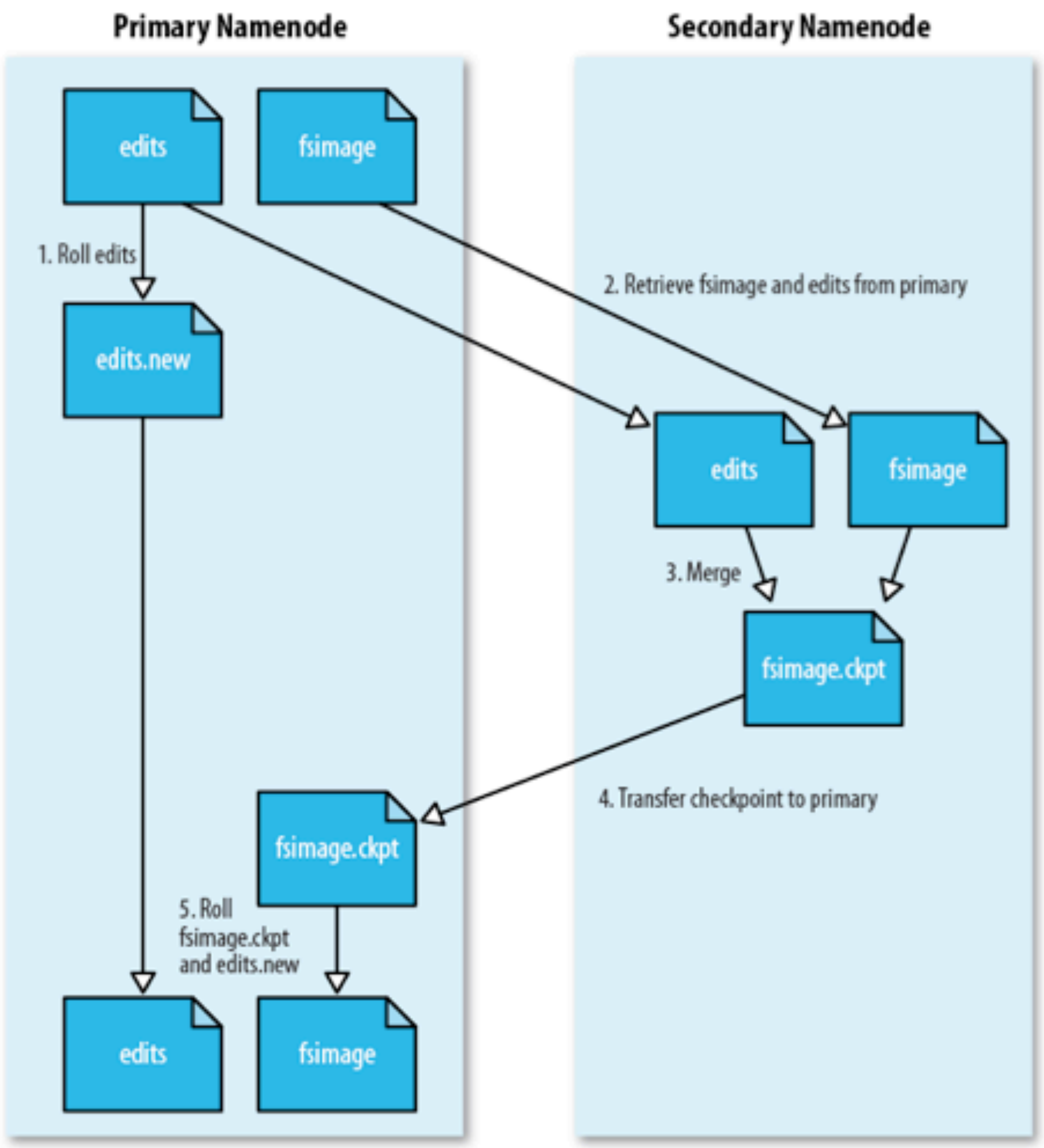
- In HDFS, the data push is performed by data nodes in pipeline
- In case of error, if at least a replica is correct, the system is going to build asynchronously the missing replicas
- The current release of HDFS supports writing new files only.



Secondary Namenode (HDFS)

- The namenode stores its filesystem metadata on local filesystem disks in a few different files, the two most important of which are **fsimage** and **edits**.
  - **fsimage** contains a complete snapshot of the filesystem metadata
  - **edits** contains only incremental modifications made to the metadata.
- When a client performs a write operation (such as creating or moving a file), it is first recorded in the edits.
- The fsimage is not updated for every filesystem write operation, because writing out the fsimage file, which can grow to be gigabytes in size, would be very slow.
  - This does not compromise resilience, however, because if the namenode fails, then the latest state of its metadata can be reconstructed by loading the fsimage from disk into memory, and then applying each of the operations in the edit log.
- The edits would grow without bound. Though this state of affairs would have no impact on the system while the namenode is running, if the namenode were restarted, it would take a long time to apply each of the operations in its (very long) edit log. During this time, the filesystem would be offline, which is generally undesirable.
  - The solution is to run the secondary namenode, whose purpose is to produce checkpoints of the primary's in-memory filesystem metadata.

HDFS Checkpointing

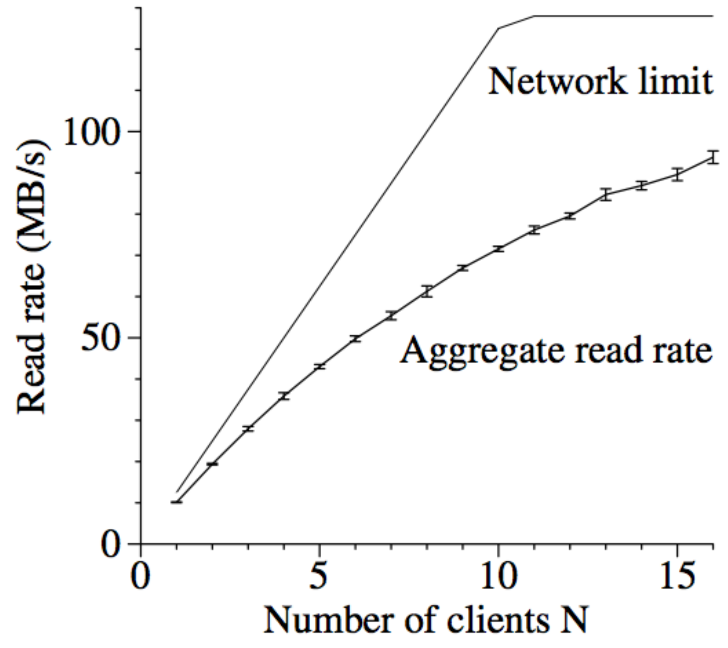


Benchmarks

- 1 Master, 2 Master replicas, 16 chunk servers, **16 clients**
- Dual 1.4 Ghz, P3, 2 GB, 2 x 80 Gb 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

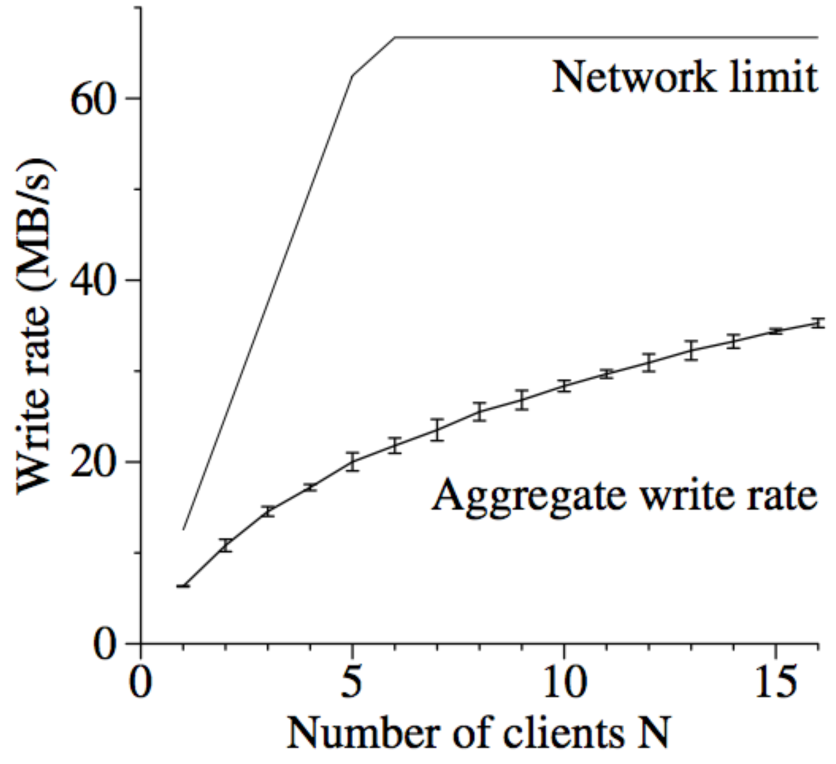
Read Performance

- Each client reads a 4 MB region from a 320 GB file set
- Repeat random 256 times to read 1 GB of data
- Theoretical Limits
  - 12.5 MB/s per client on local 100 Mbps network
  - 125 MB/s maximum on global 1 Gbps network
- Measured
  - 10 MB/s per client, or 80% of the limit
  - 94 MB/s aggregated, or 75% of the limit
  - Efficiency drops from 80% to 75% as the number of readers increases, so does the probability that multiple readers simultaneously read from the same chunkserver



Write Performance

- 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.
- Theoretical Limits
  - 67 MB/s maximum because we need to write each byte to 3 of the 16 chunk servers, each with a 12.5 MB/s input connection
  - 1 replica is input from client, 2 replicas are output to servers
- Measured
  - 6.3 MB/s per client, or 50% of the limit
  - 35 MB/s aggregated, or 50% of the limit


Network limit
Aggregate write rate
Write rate (MB/s)
Number of clients N