Lecture: The Google File System

http://research.google.com/archive/gfs-sosp2003.pdf

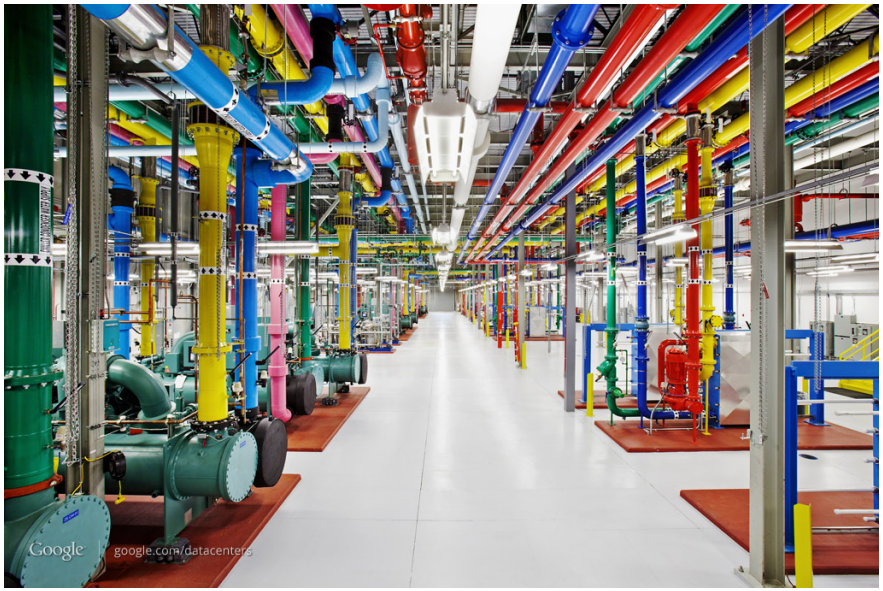
10/01/2014

Romain Jacotin

romain.jacotin@orange.fr

Agenda

- **Introduction**
- Design overview
- Systems interactions
- Master operation
- Fault tolerance and diagnosis
- Measurements
- Experiences
- Conclusions



http://research.google.com/archive/gfs-sosp2003.pdf

Introduction

Distributed file system goals :

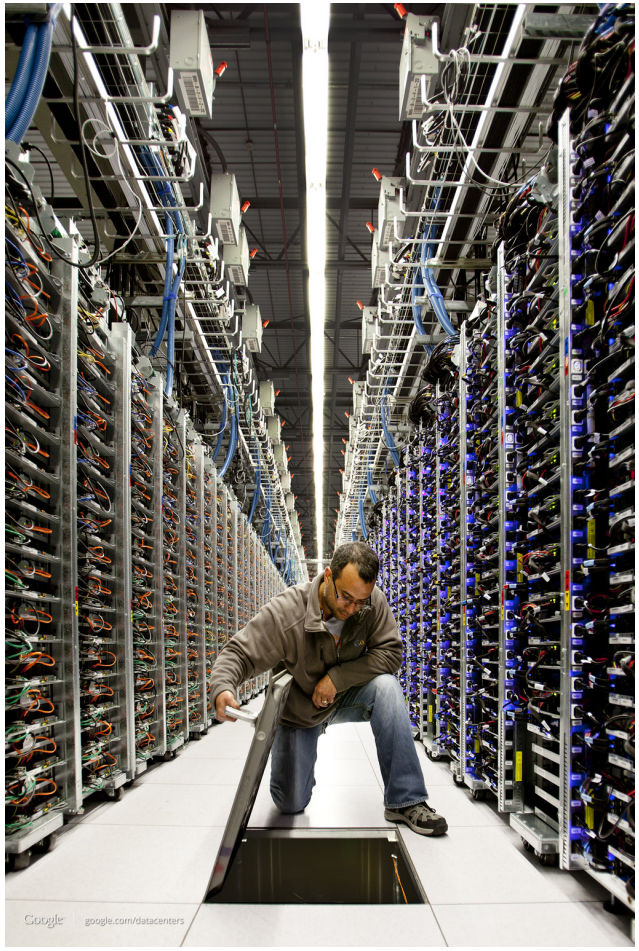
- Performance
- Scalability
- Reliability
- Availability

Design choices :

1. Component failures are the norm (COTS x86 servers)
2. Files are huge, multi-GB files are common (multi TB data sets)
3. Most files are mutated by appending new data. Random writes practically non-existent. Once written, files are only read, often sequentially.
4. File system and applications are co-designed

Agenda

- Introduction
- **Design overview**
  - Assumptions
  - Interface
  - Architecture
  - Single master
  - Chunk size
  - Metadata
  - Consistency model
- Systems interactions
- Master operation
- Fault tolerance and diagnosis
- Measurements
- Experiences
- Conclusions



http://research.google.com/archive/gfs-sosp2003.pdf

Design overview

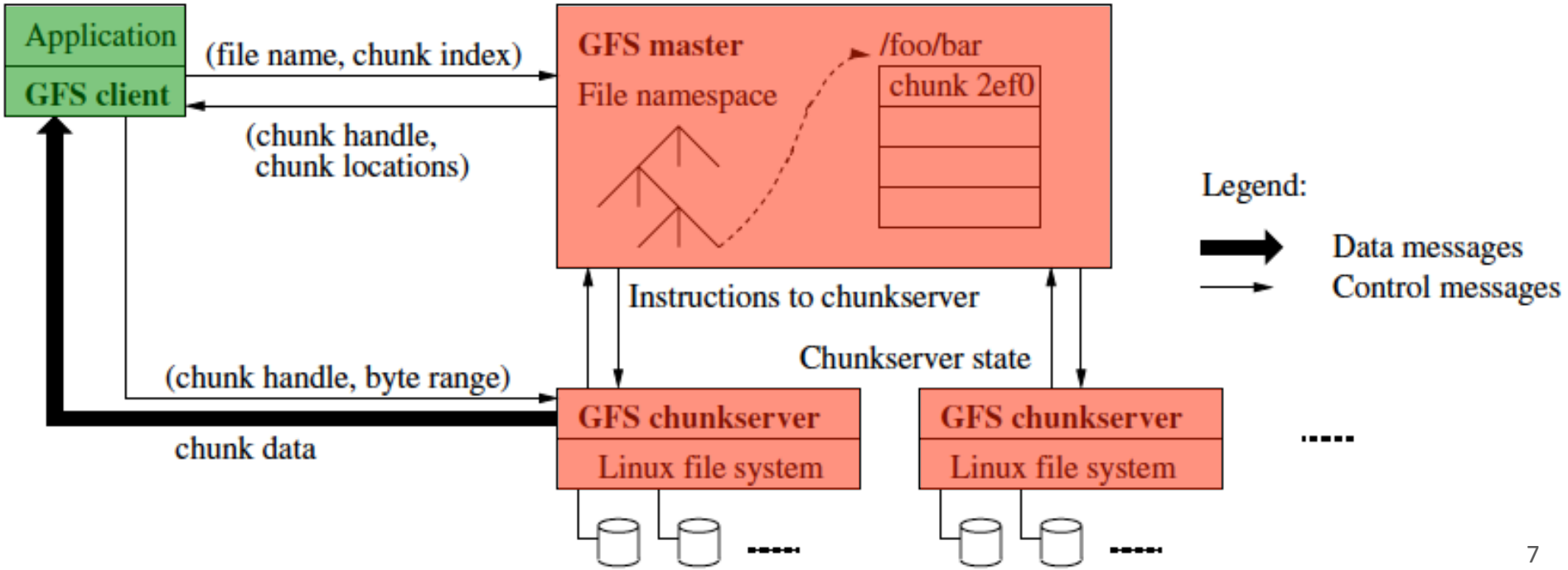
- **Assumptions**
  - COTS x86 servers
  - Few million files, typically 100MB or larger
  - Large streaming reads, small random reads
  - Sequential writes, append data
  - Multiple concurrent clients that append to the same file
  - High sustained bandwidth more important than low latency

Design overview

- **Interface**
  - Familiar file system interface API but not POSIX compatible
  - Usual files operations:
    - create
    - delete
    - open
    - close
    - read
    - write
  - Enhanced files operations:
    - snapshot (copy-on-write)
    - record append (concurrency atomic append support)

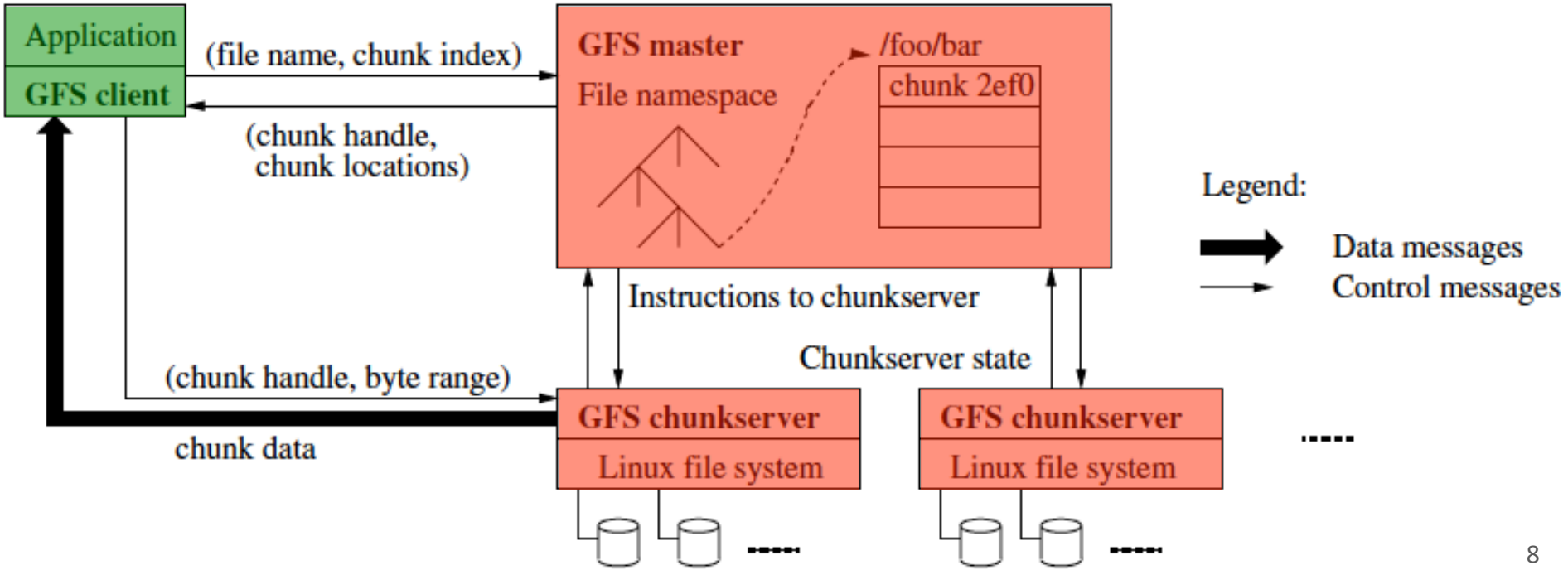
Design overview

- **Architecture**
  - Single ***GFS master*** ☹ and multiple ***GFS chunkservers*** accessed by multiple ***GFS clients***
  - GFS Files are divided into fixed-size chunks (64MB)
  - Each chunk is identified by a globally unique "chunk handle" (64 bits)
  - Chunkservers store chunks on local disks as Linux file
  - For reliability each chunk is replicated on multiple chunkservers (default = 3)



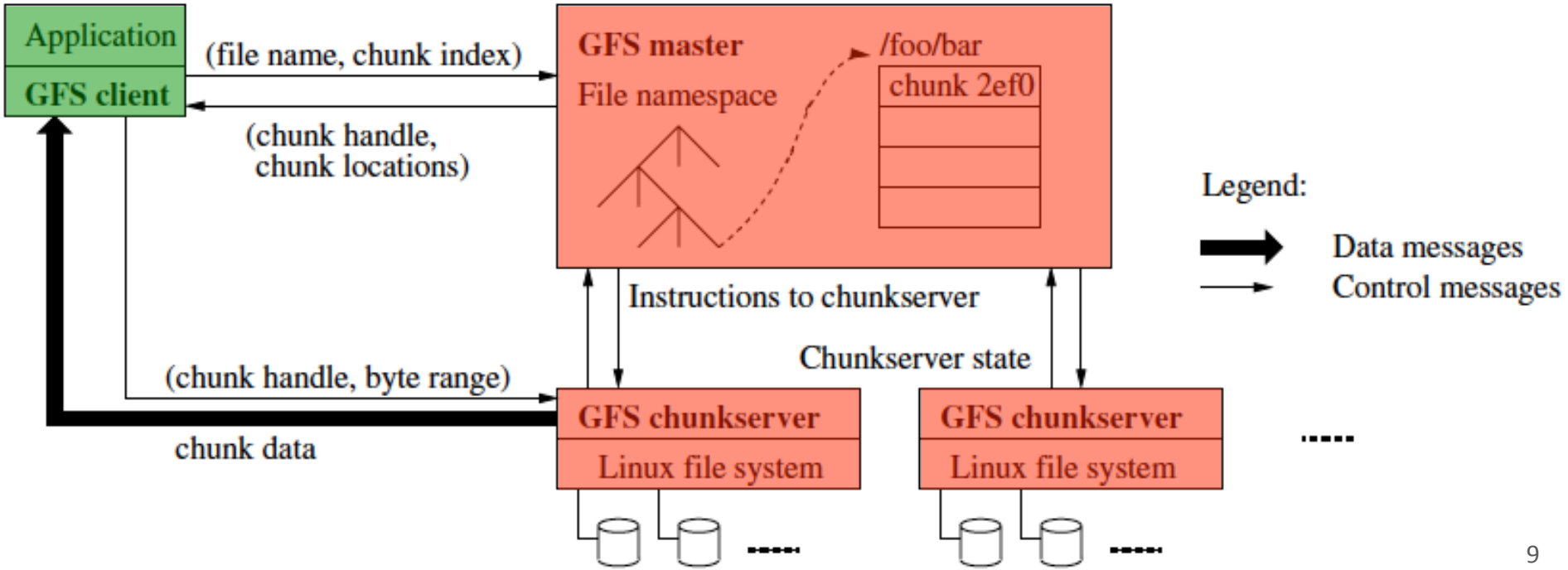
Design overview

- **Architecture**
  - GFS master
    - maintains all file system metadata
      - namespace, access control, chunk mapping & locations (files→chunks→replicas)
    - send periodically heartbeat messages with chunkservers
      - instructions + state monitoring



Design overview

- **Architecture**
  - GFS client
    - Library code linked into each applications
    - Communicates with GFS master for metadata operations (control plane)
    - Communicates with chunkservers for read/write operations (data plane)



Design overview

- **Single master** ☹
  - Design simplification *(but bad for availability and MTTR …)*
  - Global knowledge permit sophisticated chunk placement and replication decisions
  - No data plane bottleneck
    - GFS clients communicate with master only for metadata (cached with TTL expiration)
    - GFS clients communicate directly with chunkservers for read/write

Design overview

- **Chunk size**
  - 64MB
  - Stored as a plain Linux file on chunkserver
  - Extended only as needed → *Lazy space allocation avoids wasting space (no fragmentation)*
  - Why large chunk size ?
    - Advantages
      - Reduces client interaction with GFS master for chunk location information
      - Reduces size of metadata stored on master (full in-memory)
      - Reduces network overhead by keeping persistent TCP connections to the chunkserver over an extended period of time
    - Disadvantages
      - Small files can create hot spots on chunkservers if many clients accessing the same file

Design overview

- **Metadata**
  - **3 types of metadata**
    - File and chunk namespaces *(in-memory + operation log)*
    - Mapping from files to chunks *(in-memory + operation log)*
    - Locations of each chunks'replicas *(in-memory only)*
  - **All metadata is kept in GFS master's memory (RAM)**
    - Periodic scanning to implement chunk garbage collection, re-replication (when chunkserver failure) and chunk migration to balance load and disk space usage across chunkservers
  - **Operation log file**
    - The logical time line that defines the order of concurrent operations
    - Contains only metadata Namespaces + Chunk mapping
    - kept on GFS master's local disk and replicated on remote machines
    - No persistent record for chunk locations (master polls chunkservers at startup)
    - GFS master checkpoint its state (B-tree) whenever the log grows beyond a certain size

Design overview

- **Consistency model**
  - File namespace mutation are atomic (handle by one master in global total order)
  - Data mutation:

	Write	Record Append
Serial success	*defined*	*defined* interspersed with
Concurrent successes	*consistent* but *undefined*	*inconsistent*
Failure	*inconsistent*	

  - **Write** = data written at an application-specified file offset
  - **Record append** = data appended atomically at least once even in the presence of concurrent mutations, but at an offset of GFS's choosing. **(GFS may insert padding or records duplicates in between)**
  - A file region is ***consistent*** if all clients will always see the same data, regardless of which replicas
  - A region is ***defined*** after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety
  - Since a failure at any replica makes the client try the write again, there might be some duplicate data. Therefore GFS does not guarantee against duplicates but in anyway the write will be carried out at least once

Agenda

- Introduction
- Design overview
- **Systems interactions**
  - Leases and mutation order
  - Data flow
  - Atomic record appends
  - Snapshot
- Master operation
- Fault tolerance and diagnosis
- Measurements
- Experiences
- Conclusions

http://research.google.com/archive/gfs-sosp2003.pdf

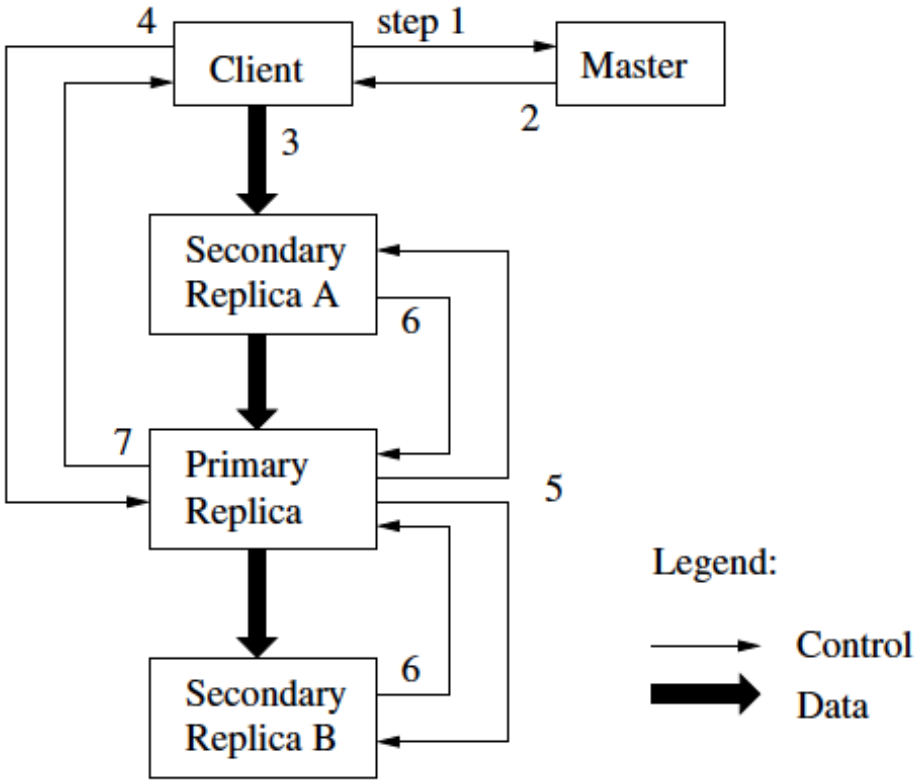
Systems interactions

- **Leases and mutation order**
  - Each mutation *(write or append)* is performed at all the chunk's replicas
  - Leases used to maintain a consistent mutation order across replicas
    - Master grants a chunk lease for 60 seconds to one of the replicas: the primary replica *(primary can request lease extension: piggyback on heartbeat message)*
    - The primary replica picks a serial order for all mutations to the chunk
    - All replicas follow this order when applying mutations (global mutation order defined first by the lease grant order chosen by the master, and within a lease by the serial numbers assigned by the primary replica)
  - Master may sometimes try to revoke a lease before it expires (when master wants to disable mutations on a file that is being renamed)

Systems interactions

Write control and Data flow

1. Client ask master for primary and secondaries replicas info for a chunk
2. Master replies with replicas identity (client put info in cache with timeout)
3. Client pushes Data to all replicas (pipelined fashion) Each chunkserver store the Data in an internal LRU buffer cache until Data is used or aged out
4. One all replicas have acknowledged receiving the data, client send a write request to the primary replica
5. Primary replica forward the write request to all secondary replicas that applies mutations in the same serial number order assigned by the primary
6. Secondary replicas acknowledge the primary that they have completed the operation
7. The primary replica 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 a arbitrary subset of the secondary replicas …

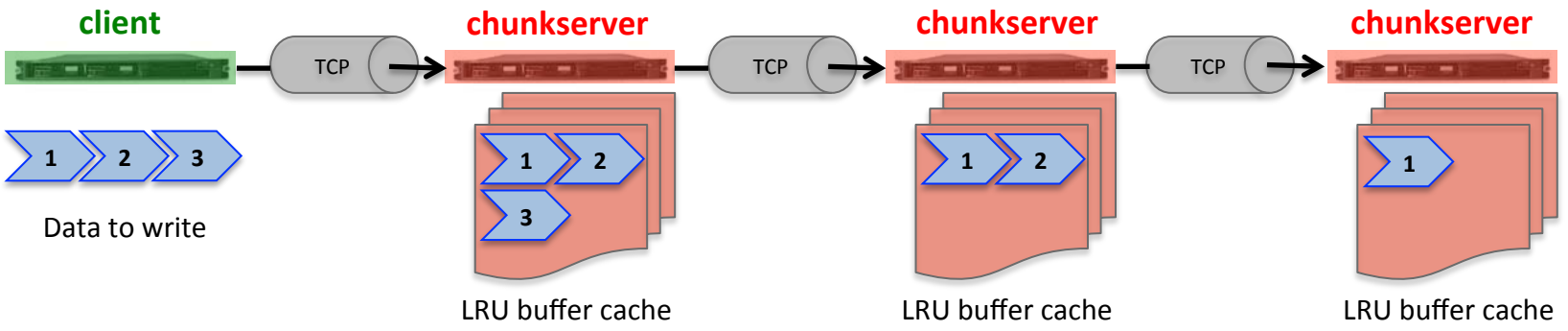
The client request is considered to have failed, and **the modified region is left in an inconsistent state**. GFS client code handle such errors by retrying the failed mutation **(→duplicates append record)**

Systems interactions

- **Data flow**
  - Data flow decoupled from Control flow to use the network efficiently

    looks like today ONF definition about SDN but used by Google in 2003 **;-)**

  - Data is pushed linearly along a carefully picked chain of chunkservers in a TCP pipelined fashion
    - Once a chunkserver receives some data, it starts forwarding immediately to the next chunkserver
  - Each machine forwards the data to the closest machine in the network topology that has not received it



Systems interactions

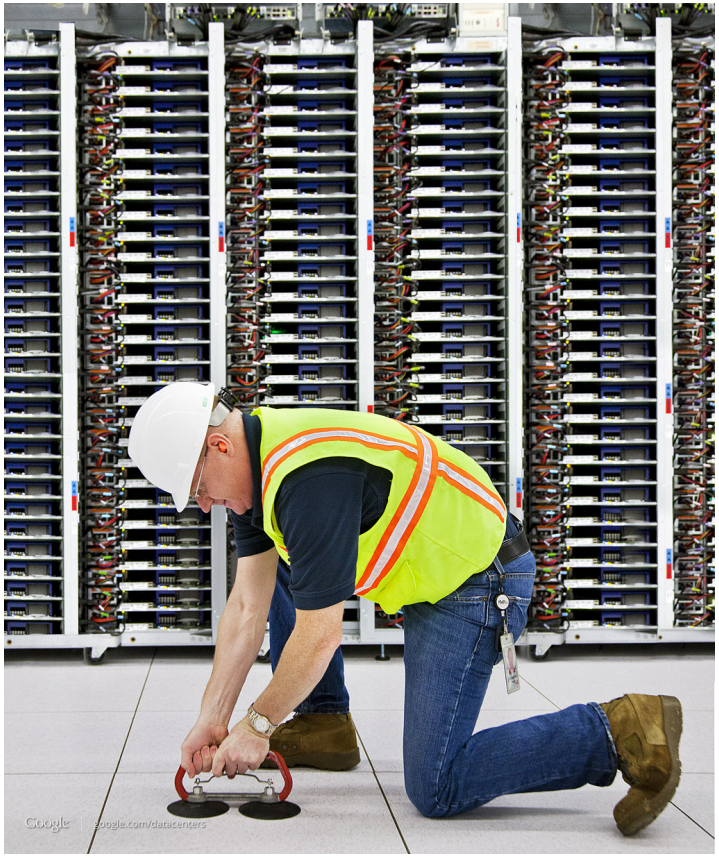
- **Atomic record appends**
  - In a record append, client specify only the data, and GFS choose the offset when appending the data to the file (concurrent atomic record appends are serializable)
    - Maximum size of an append record is 16 MB
    - Client push the data to all replicas of the last chunk of the file (as for write)
    - Client sends its request to the primary replica
    - Primary replica check if the data size would cause the chunk to exceed the chunk maximum size (64MB)
      - If so, it pads the chunk to the max size, tells secondary to do the same, and replies to the client to retry on the next chunk
      - If the record size can fit in the chunk, primary append the data to its replica and tells secondary replicas to do the same. Any futur record will be assigned a higher offset or a different chunk
  - In Google workloads, such file serve as multiple-producer/single-consumer queues or contain merged results from many different clients (**MapReduce**)

Systems interactions

- **Snapshot**
  - Make a instantaneously copy of a file or a directory tree by using standard copy-on-write techniques
    - Master receives a snapshot request
    - Master revokes any outstanding leases on the chunks in the files to snapshot
    - Master wait leases to be revoked or expired
    - Master logs the snapshot operation to disk
    - Master duplicate the metadata for the source file or directory tree: newly created snapshot files point to the same chunk as source files
    - First time a client wants to write to a chunk C, it sends a request to the master to find current lease holder. The master notices that the reference count for chunk C is greater than one. It defers replying to the client request and instead picks a new chunk handle C' and ask each chunkserver that has a current replica of C to create a new chunk called C' (local disk cloning without using the network)

Agenda

- Introduction
- Design overview
- Systems interactions
- **Master operation**
  - Namespace management and locking
  - Replica placement
  - Creation, re-replication, rebalancing
  - Garbage collection
  - Stale replication detection
- Fault tolerance and diagnosis
- Measurements
- Experiences
- Conclusions



http://research.google.com/archive/gfs-sosp2003.pdf

Master operation

- **Namespace management and Locking**
  - Locks over region of the namespace to ensure proper serialization AND allow multiple operations at the master
  - Each absolute file name or absolute directory name has an associated read-write lock
  - Each master operation acquires a set of locks before it runs
    - To make operation on /dir1/dir2/dir3/leaf it first needs the following locks
      - Read-lock on /dir1
      - Read-lock on /dir1/dir2
      - Read-lock on /dir1/dir2/dir3
      - Read-lock or Write-lock on /dir1/dir2/dir3/leaf
  - File creation doesn't require write-lock on parent directory: read-lock on the name sufficient to protect the parent directory from deletion, rename, or snapshotted
  - Write-locks on file names serialize attempts to create a file with the same name twice
  - Locks are acquired in a consistent total order to prevent deadlock:
    - First ordered by level in the namespace tree
    - Lexicographically ordered within the same level

Master operation

- **Replica placement**
  - Serves two purposes:
    - Maximize data reliability and availability
    - Maximize network bandwidth utilization
  - Spread chunk replicas across racks
    - ☺ To ensure chunk survivability
    - ☺ To exploit aggregate read bandwidth of multiple racks
    - ☹ write traffic has to flow through multiple racks

Master operation

- **Creation, Re-replication, Balancing**
  - Chunk replicas created for 3 reasons: chunk creation, chunk re-replication, chunk rebalancing
  - **Creation**: Master considers several factors
    - Place new replicas on chunkservers with below-average disk space utilization
    - Limit the number of "recent" creations on each chunkserver
    - Spread replicas of a chunk across racks

Master operation

- **Creation, Re-replication, Balancing**
  - **Re-replication**: Master re-replicate a chunk as soon as the number of available replicas falls below a user-specified goal
    - When a chunkserver becomes unavailable
    - When a chunkserver reports a corrupted chunk
    - When the replication goal is increased
  - Re-replicated chunk is prioritized based on several factors
    - Higher priority to chunk that has lost 2 replicas than chunk that lost 1 replica
    - Chunk for live files preferred over chunks that belong to recently deleted files
    - Boost the priority of any chunk that is blocking client progress
  - Re-replication placement is similar as for “creation”
  - Master limits the numbers of active clone operations both for the cluster and for each chunkservers
  - Each chunkserver limits the amount of bandwidth it spends on each clone operation

Master operation

- **Creation, Re-replication, Balancing**
  - **Balancing**: Master re-balances replicas periodically for better disk space and load balancing
  - Master gradually fills up a new chunkserver rather than instantly swaps it with new chunks (and the heavy write traffic that come with them !)
    - Re-balancing placement is similar as for “creation”
    - Master must also choose wich existing replica to remove: better to remove from chunkservers with below-average free space so as to equalize disk space usage

Master operation

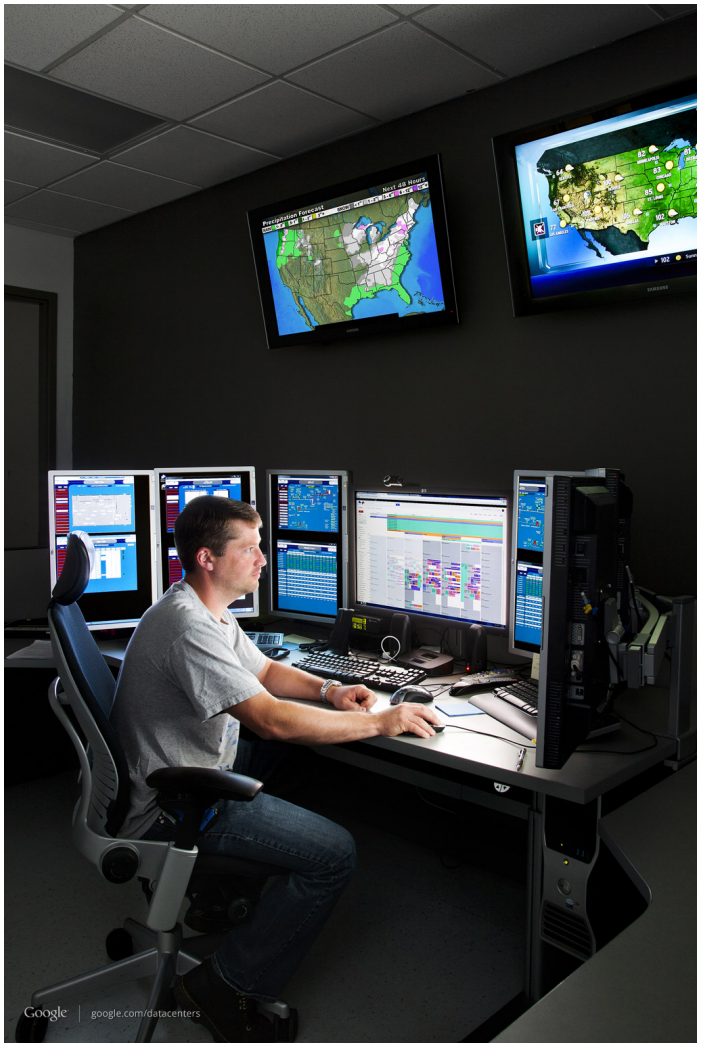
- **Garbage Collection**
  - GFS doesn't immediately reclaim the available physical storage after a file is deleted: does it lazily during regular garbage collection at both the file and chunks levels *(much simpler and more reliable)*
  - Mechanism:
    - Master logs the deletion like others changes
    - File renamed to a hidden name that include deletion timestamp
    - During regular scan of file system namespace master removes any such hidden files if they have existed for more than 3 days (configurable). When the hidden file is removed from the namespace, its in-memory metadata is erased
    - Master does a similar scan of the chunk namespace to identifies orphaned chunks and erases the metadata for those chunks.
    - In heartbeat message regularly exchange with the master, chunkserver reports a subset of the chunks it has, master replies with the identity of chunks not present in its metadata →chunkserver is free to delete those chunks

Master operation

- **Stale replica detection**
  - **Problem**: chunk replica may become stale if a chunkserver fails and misses mutations
  - **Solution**: for each chunk, master maintains a chunk version number
    - Whenever master grants a new lease on a chunk, master increases the chunk version number and informs up-to-date replicas (chunk version number stored persistently on master and associated chunkservers)
    - Master detect that chunkserver has a stale replica when the chunkserver restarts and reports its set of chunks and associated version numbers
    - Master removes stale replica in its regular garbage collection
    - Master includes chunk version number when it informs clients which chunkserver holds a lease on a chunk, or when it instructs a chunkserver to read the chunk from another chunkserver in cloning operation

Agenda

- Introduction
- Design overview
- Systems interactions
- Master operation
- **Fault tolerance and diagnosis**
  - High Availability
  - Data Integrity
  - Diagnostic tools
- Measurements
- Experiences
- Conclusions



http://research.google.com/archive/gfs-sosp2003.pdf

Fault tolerance and diagnosis

- **High Availability**
  - Two simple effective strategies: **fast recovery and replication**
  - **Fast Recovery**
    - Master and chunkservers designed to restore their state in seconds
    - No matter how they terminated, no distinction between normal and abnormal termination (servers routinely shutdown just by killing process)
    - Clients and other servers experience minor hiccup (timeout) on outstanding requests, reconnect to the restarted server, and retry

Fault tolerance and diagnosis

- **High Availability**
  - Two simple effective strategies: **fast recovery and replication**
  - **Chunk replication**
    - Chunk replicated on multiple chunkservers on different racks *(different parts of the file namespace can have different replication level)*
    - Master clones existing replicas as chunkservers go offline or detect corrupted replicas *(checksum verification)*
    - *Google is exploring others cross-server redundancy (**parity or erasure coding**) for increasing read-only storage requirements (challenging but manageable because of appends and reads traffic domination)*
  - **Master replication**
    - Master operation log and checkpoints replicated on multiple machine for reliability
    - If the machine or disk failed, monitoring process outside of GFS starts a new master process elsewhere with the replicated log
    - Clients use only the canonical name of the master (DNS alias that can be changed)
    - Shadow masters provides read-only access to file system even when master is down

Fault tolerance and diagnosis

- **Data Integrity**
  - Each chunkserver uses checksumming to detect corruption of stored chunk
  - Chunk broken into 64 KB blocks with associated 32 bits checksum
  - Checksums are metadata kept in memory and stored persistently with logging, separate from user data

  - For **READS**, chunkserver verifies the checksum of data blocks that overlap the range before returning any data
    - If a block is corrupted the chunkserver returns an error to the requestor and reports the mismatch to the master.
      - The requestor read from others replicas
      - The master clone the chunk from another replica and after instruct the chunkserver to delete its corrupted replica

Fault tolerance and diagnosis

- **Data Integrity**
  - For **WRITES**, chunkserver verifies the checksum of first and last data blocks that overlap the write range before perform the write, and finally compute and record the new checksums
    - If a block is corrupted the chunkserver returns an error to the requestor and reports the mismatch to the master.
      - The requestor writes from others replicas
      - The master clone the chunk from another replica and after instruct the chunkserver to delete its corrupted replica
  - For **APPENDS**, checksum computation is optimized:
    - No checksum verification on the last block, just incrementally update the checksum for the last partial checksum block
    - Compute new checksums for any brand new checksum blocks filled by the append
  - During idle periods, chunkservers can scan and verify the contents of inactive chunks *(prevents an inactive but corrupted chunk replica from fooling the master into thinking that it has enough valid replicas of a chunk…)*

Fault tolerance and diagnosis

- **Diagnostic tool**
  - GFS servers generate diagnostic logs (sequentially + asynchronously) about significant events (chunkservers going up/down) and all RPC requests/replies
  - RPC logs include exact requests and responses sent on the wire without data
  - Interaction history can be reconstruct by matching requests with replies and collating RPC records on different machines
  - Logs used for load testing and performance analysis
  - Most recent events kept in memory and available for continuous online monitoring

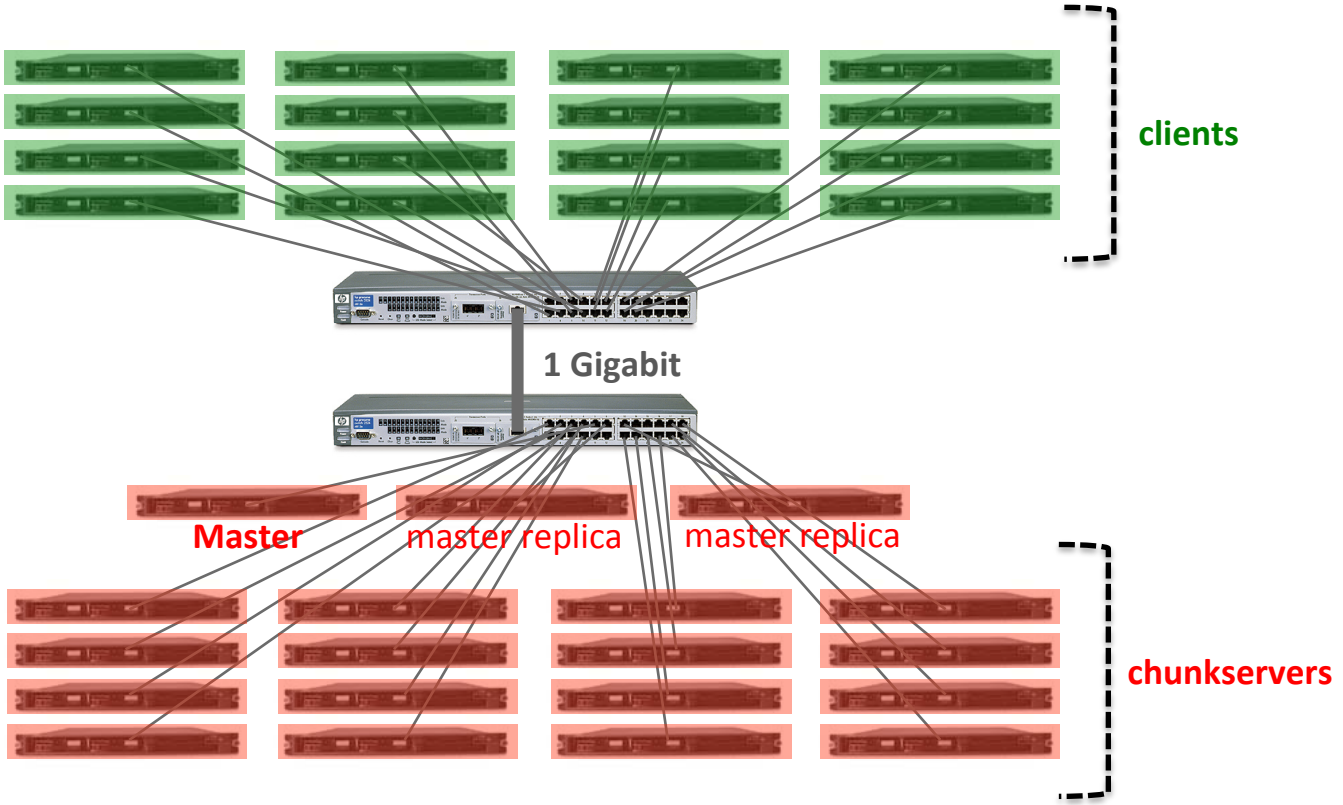
Agenda

- Introduction
- Design overview
- Systems interactions
- Master operation
- Fault tolerance and diagnosis
- **Measurements**
  - Micro-benchmarks
  - Real world clusters
  - Workload breakdown
- Experiences
- Conclusions

http://research.google.com/archive/gfs-sosp2003.pdf

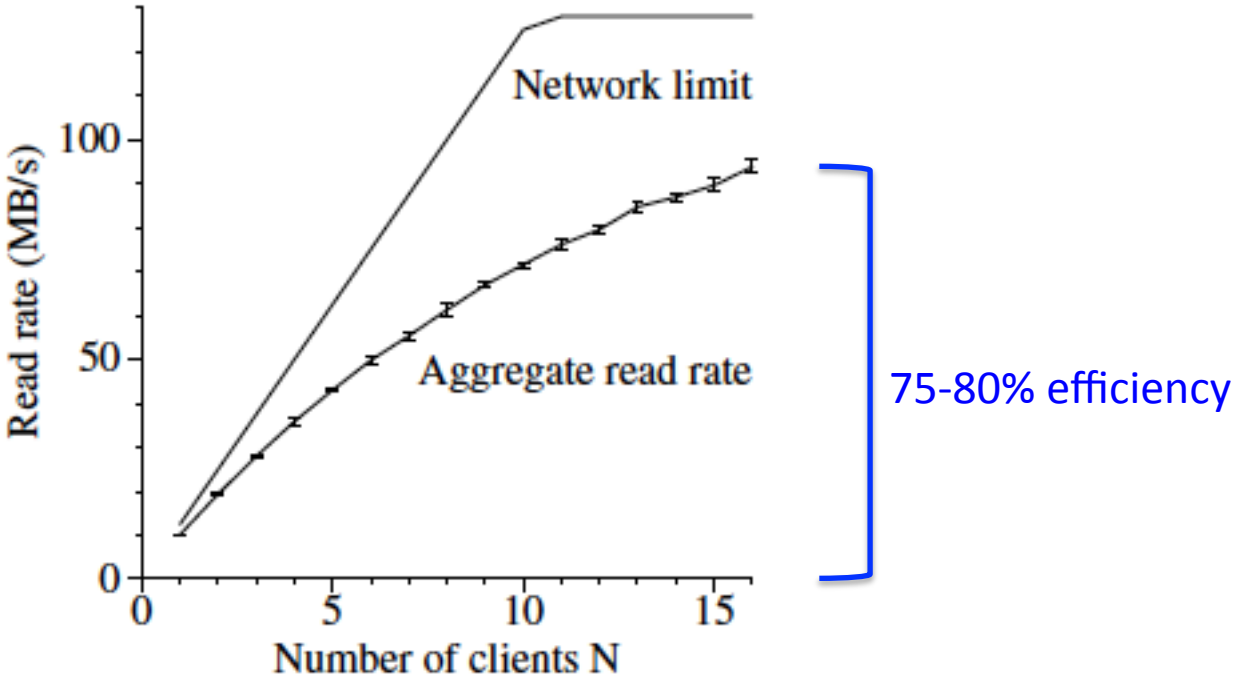
Measurements (2003)

- **Micro-benchmarks: GFS CLUSTER**
  - 1 **master**, 2 **master replicas**, 16 **chunkservers** with 16 **clients**
  - **Dual 1.4 GHz PIII** processors, **2GB RAM**, **2x80GB** 5400 rpm disks, **FastEthernet NIC** connected to one HP 2524 Ethernet switch 24 ports 10/100 + **Gigabit uplink**



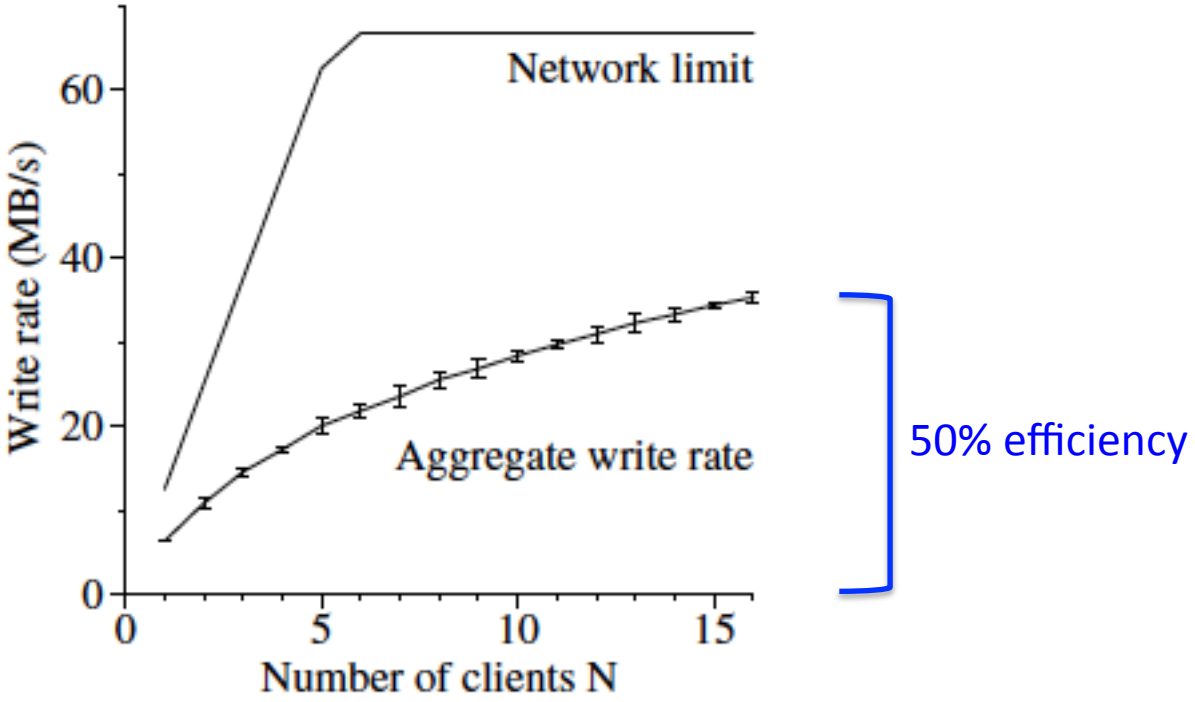
Measurements (2003)

- **Micro-benchmarks: READS**
  - Each client read a randomly selected 4MB region 256 times (= 1 GB of data) from a 320MB file
  - Aggregate chunkservers memory is 32GB, so 10% hit rate in Linux buffer cache expected



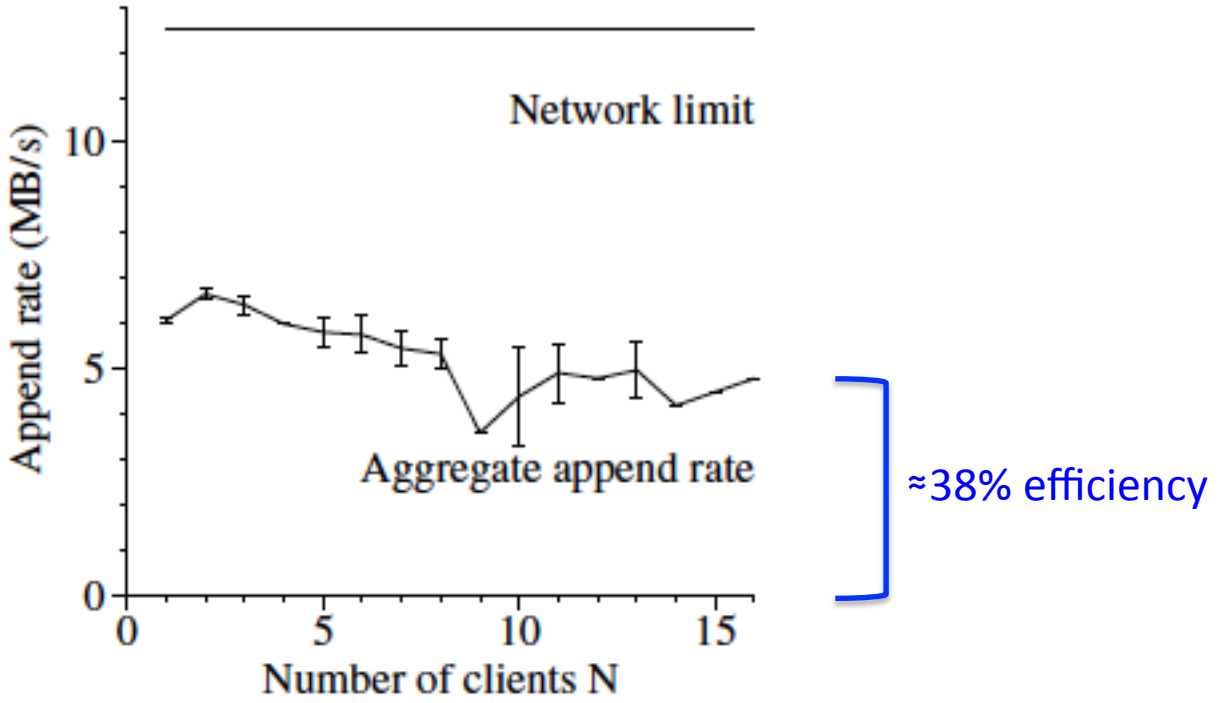
Measurements (2003)

- **Micro-benchmarks: WRITES**
  - Each client writes 1 GB of data to a new file in a series of 1 MB writes
  - Network stack does not interact very well with the pipelining scheme used for pushing data to the chunk replicas: network congestion is more likely for 16 writers than for 16 readers because each write involves 3 different replicas (→ see: write data flow)



Measurements (2003)

- **Micro-benchmarks: RECORDS APPENDS**
  - Each client appends simultaneously to a single file
  - Performance is limited by the network bandwidth of the 3 chunkservers that store the last chunk of the file, independent of the number of clients



Measurements (2003)

- **Real world clusters: STORAGE & METADATA**
  - Cluster A used regularly for R&D by +100 engineers
  - Cluster B used for production data processing

	R&D	Production
Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Cluster A store 55/3= **18 TB** of data
Cluster B store 155/3= **52 TB** of data

Chunkservers metadata = checksums for each 64 KB data block + chunk version number

Master metadata = file names + file ownership and permissions + file→chunk mapping + chunk→replicas mapping + chunk current version number + reference count (for copy-on-write)

Measurements (2003)

- **Real world clusters: READ/WRITE RATES & MASTER LOAD**
  - Cluster A used regularly for R&D by +100 engineers
  - Cluster B used for production data processing

	R&D	Production
Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Cluster A network configuration can support read rate of 750 MB/s

Cluster B network configuration can support read rate of 1300 MB/s

Measurements (2003)

- **Real world clusters: RECOVERY TIME**
  - **Kill 1 chunkserver in cluster B** (Production)
    - 15.000 chunks on it (= 600 GB of data)
    - Cluster limited to 91 concurrent chunk cloning (= 40% of 227 chunkservers)
    - Each clone operation consume at most 50 Mbps
    - **15.000 chunks restored in 23.2 minutes effective replication rate of 440 MB/s**
  - **Kill 2 chunkservers in cluster B** (Production)
    - 16.000 chunks on each (= 660 GB of data)
    - This double failure reduced 266 chunks to having a single replica… ☹
    - **These 266 chunks cloned at higher priority and were all restored to a least 2xreplication within 2 minutes**

Measurements (2003)

- **Workload breakdown: CHUNKSERVER WORKLOAD**
  - Cluster X used regularly for R&D
  - Cluster Y used for production data processing
  - Results reflects only client originated requests (no inter-server or internal background cluster activity)

Operations breakdown by size (%)

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Measurements (2003)

- **Workload breakdown: CHUNKSERVER WORKLOAD (READS)**
  - **Small reads (under 64 KB)** come from seek-intensive clients that look up small pieces of data within huge files
  - **Large reads (over 512 KB)** come from long sequential reads through entire files
  - Producers append concurrently to a file while a consumer reads the end of file. **Occasionally, no data is returned when the consumer outpaces the producers**

Operations breakdown by size (%)

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Measurements (2003)

- **Workload breakdown: CHUNKSERVER WORKLOAD (WRITES)**
  - **Large operations (over 256 KB)** result from significant buffering within writers
  - **Small writes (under 64 KB)** result from writers than buffer less data, checkpoint or synchronize more often or generate less data

Operations breakdown by size (%)

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Measurements (2003)

- **Workload breakdown: CHUNKSERVER WORKLOAD**
  - **Larger operations (over 256 KB)** generally account for most of the bytes transferred
  - **Small reads (under 64 KB)** do transfer a small but significant portion of the read data because of the random seek workload

Bytes transferred breakdown by operation size (%)

Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Measurements (2003)

- **Workload breakdown: CHUNKSERVER WORKLOAD (APPENDS)**
  - **Records appends heavily used especially in production systems**
    - Writes to record appends ratio in cluster X
      - 108:1 by bytes transferred
      - 8:1 by operation counts
    - Writes to record appends ratio in cluster Y
      - 3.7:1 by bytes transferred
      - 2.5:1 by operation counts
  - **Data mutation workload dominated by appending**
    - Overwriting account in cluster X
      - 0.0001% of bytes mutated
      - 0.0003% of mutation operations
    - Overwriting account in cluster Y
      - 0.05% of bytes mutated
      - 0.05% of mutation operations

Measurements (2003)

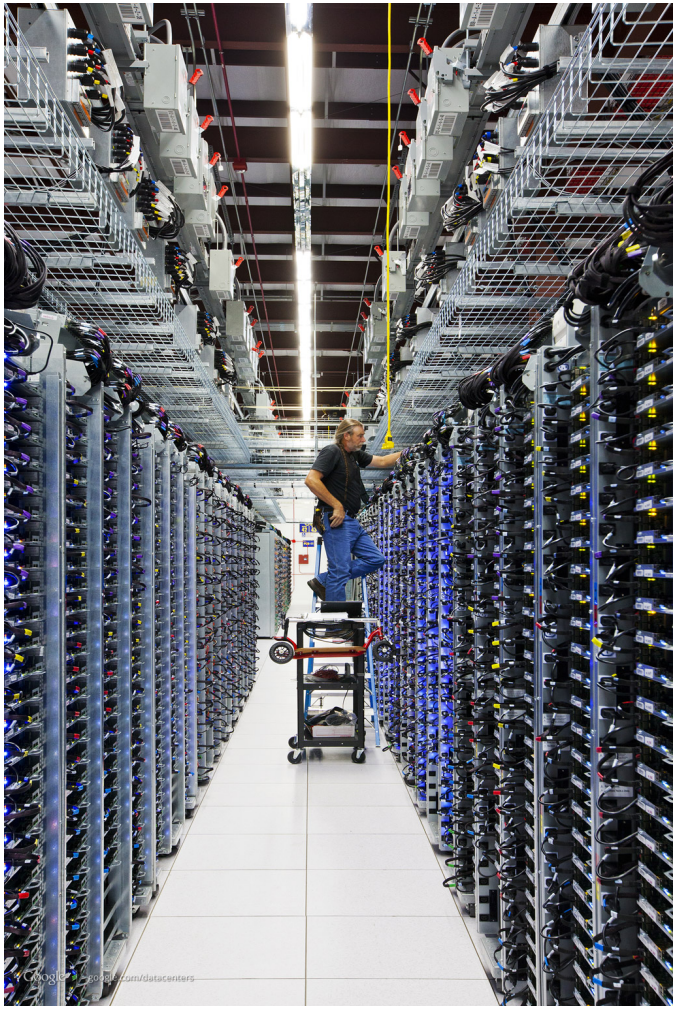
- **Workload breakdown: MASTER WORKLOAD**
  - Most request ask for chunk locations ***(findLocation)*** for reads and lease holder information ***(FindLeaseHolder)***
  - ***FindMatchingFiles*** is a pattern matching request that support "ls" and similar file system operations. Unlike other requests to the master, it may process a large part of the namespace and so may be expensive

Master requests breakdown by type (%)

Cluster	X	Y
Open	26.1	16.3
Delete	0.7	1.5
FindLocation	64.3	65.8
FindLeaseHolder	7.8	13.4
FindMatchingFiles	0.6	2.2
All other combined	0.5	0.8

Agenda

- Introduction
- Design overview
- Systems interactions
- Master operation
- Fault tolerance and diagnosis
- Measurements
- **Experiences**
- Conclusions



http://research.google.com/archive/gfs-sosp2003.pdf

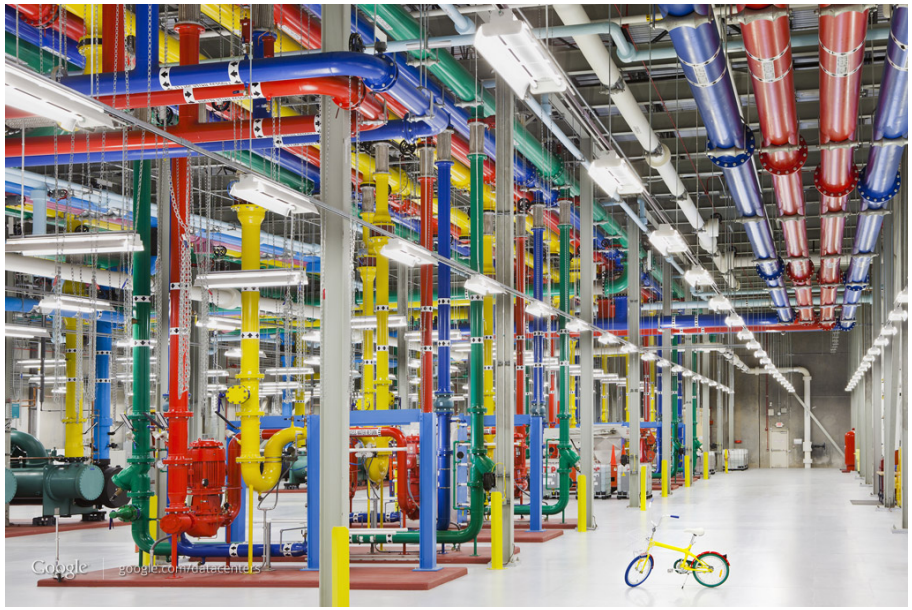
Experiences (2001-2003)

- **Operational issues**
  - Original design for backend file system on production systems, not with R&D tasks in mind (lack of support for permissions and quotas…)
    "Production systems are well disciplined and controlled, users sometimes are not." ☺

- **Technical issues**
  - Biggest problems were disk and Linux related
    - IDE protocol versions support that corrupt data silently : this problem motivated checksums usage
    - fsync() cost in kernel 2.2 towards large operational logs files
    - mmap() bug make change to pread()

Agenda

- Introduction
- Design overview
- Systems interactions
- Master operation
- Fault tolerance and diagnosis
- Measurements
- Experiences
- **Conclusions**

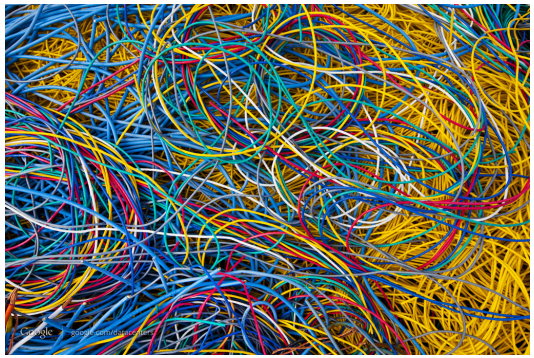


http://research.google.com/archive/gfs-sosp2003.pdf

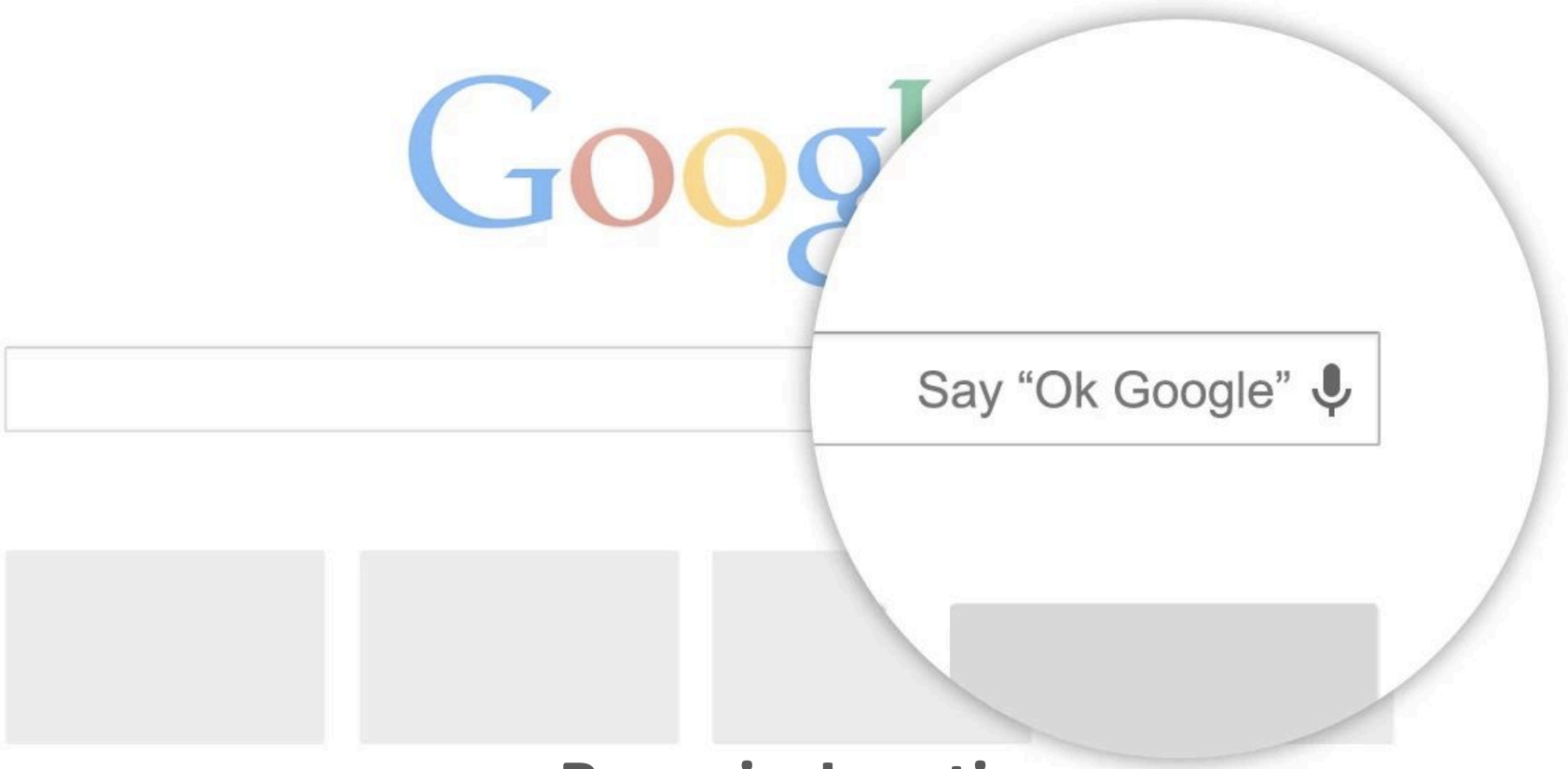
Conclusions (2003)

- **GFS**
  - Support large-scale data processing workloads on COTS x86 servers
  - 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 and automatic recovery (+ checksum to detect data corruption)
  - Delivers high aggregate throughput to many concurrent readers and writers

- **Future improvements**
  - **Networking stack**: current limitation on the write throughput



Any questions ?



Romain Jacotin

romain.jacotin@orange.fr