UNIVERSITÀ DEGLI STUDI DI TRENTO
Dipartimento di Ingegneria
e Scienza dell'Informazione

(Big) Distributed Systems

Gian Pietro Picco
Dipartimento di Ingegneria e Scienza dell'Informazione
University of Trento, Italy
gianpietro.picco@unitn.it
http://disi.unitn.it/~picco

One Server … or 10,000?

- We all use a number of common Internet services, provided by world-wide companies
  - E.g., Google, Amazon, FaceBook, Yahoo!, …
- We access their services by contacting "the server"
  - Is it really one machine that does all the work?
- Did you ever think about the amount of "work" that these systems must do?
  - Data processing
  - Data storage and retrieval
  - Number of user requests to be served simultaneously

Example: FaceBook

- More than 1.55 ***billion*** users
  - 1.01 billion users login every day
  - The average user has 130 friends
- More than 900 million objects (users and group pages) stored
  - Average user creates 90 pieces of content per month
- More than 550,000 active applications
- Facebook Photos:
  - Stores 260 billion images = 20 petabytes of data
  - Each week, 1 billion new photos (= 60 terabytes)

Example: Google

- In July 2008, Google served 9.9 billion searches
  - 7.5 billion through google.com, 2.5 billion through YouTube
- This is only one of the many Google services
- Trivia about scalability:
  - Search:
    - In 1996: 25 million pages indexed
    - In 2010: 40 billion pages indexed (a 1600x increase)
  - 1.5 billion images
  - Gmail: the current storage across all users is equivalent to 1.74 billions of full audio CDs

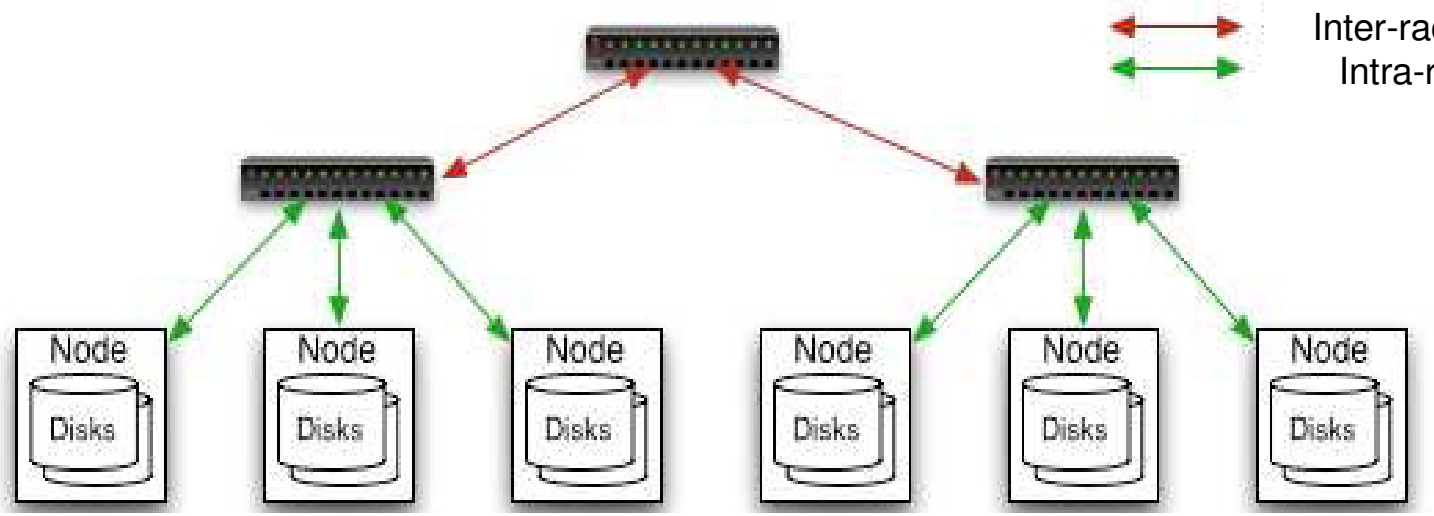
Data Centers

- Clearly, these loads and performances cannot be sustained by a single machine
- Companies rely on data centers containing thousands of machines, organized in clusters, connected by high-speed networks
  - Usually commodity machines (not high-end servers)
  - Power and heating issues
  - Google has multiple data centers, some of which allegedly stored in containers moved around the US
- Reliability and fault tolerance is fundamental
  - When you deal with several thousands of machines, there are always some of them that failed
    - Or their disk, their network, their services, …
  - In the systems surveyed, reliability is provided ***in software***, instead of relying on expensive, more reliable hardware

Example: Yahoo!

courtesy of Srigurunath Chakravarthi, Yahoo!India

- Yahoo! serves 600 million users

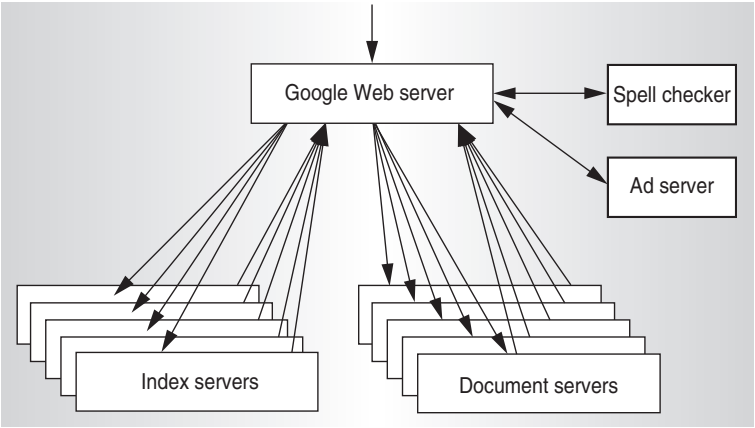


- Nodes are commodity servers (Intel multi-core)
- 16 GB RAM, 4-6 SATA Disks.
- 40 nodes/rack
- Intra-rack bandwidth is 1 Gbps
- Inter-rack: 200-500 Mbps/node
- 2000-4000 nodes per Hadoop cluster!
- 30,000+ nodes across Yahoo!

How Google Works

- *"On average, a single query on Google reads hundreds of MB of data and consumes tens of billions of CPU cycles."*
- When a user "googles" a query, the DNS maps **`www.google.com`** to a specific IP address, determined using round-robin
  - First-level of load balancing
  - The web server further redirects to a cluster
  - Each cluster contains a few thousands machines
- Why PC-based clusters are more convenient:
  - A rack contains overall 176 2GHz Xeon CPUs, 176GB of RAM, 7TB of storage for $278,000 (in 2003)
  - A high-end x86-based server: 8 2GHz Xeon CPUs, 64GB of RAM, 8TB of storage for $758,000

How Google Works



- Query execution goes first to (a set of) index servers
  - consult an inverted index, mapping each query word to a matching list of documents
  - determine a relevance score for each document (PageRank)
  - Large amount of data: several tens of TB of documents
  - Search is highly parallelizable by dividing the index into pieces (*shards*) based on a random set of documents, and processed by a different set of machines
- The output is an ordered list of document identifiers, processed by document servers
  - These are the ones that build the actual page by merging URL, text snippets, and so on
  - *"The docserver cluster must have access to an online, low-latency copy of the entire Web. In fact, because of the replication required for performance and availability, Google stores dozens of copies of the Web across its clusters."*
- Other services are also consulted in parallel

MapReduce

- Lots of computations at Google are conceptually simple, but the data input is huge
  - Examples: inverted indexes, distributed grep, count of URL access frequencies, reverse web link graph, term-vector per host, distributed sort
  - Must be distributed to hundreds or thousands of machines to finish in reasonable time
  - Dealing with parallelization, data distribution, and fault tolerance is complex and repetitive
- MapReduce is an abstraction that simplifies development by enabling programmers to focus on the application logic
- Based on two functions:
  - Map: is applied to the input set, specified as key-value pairs, and outputs a different, intermediate set of key-value pairs
  - Reduce: merges the intermediate set, based on the key, into the final output

Example: Counting Word Occurrences in Documents

- For instance, given the phrase “To be or not to be”

```
map(String key, String value):
  // key: document name
  // value: document contents
     for each word w in value:
  EmitIntermediate(w, "1");
```

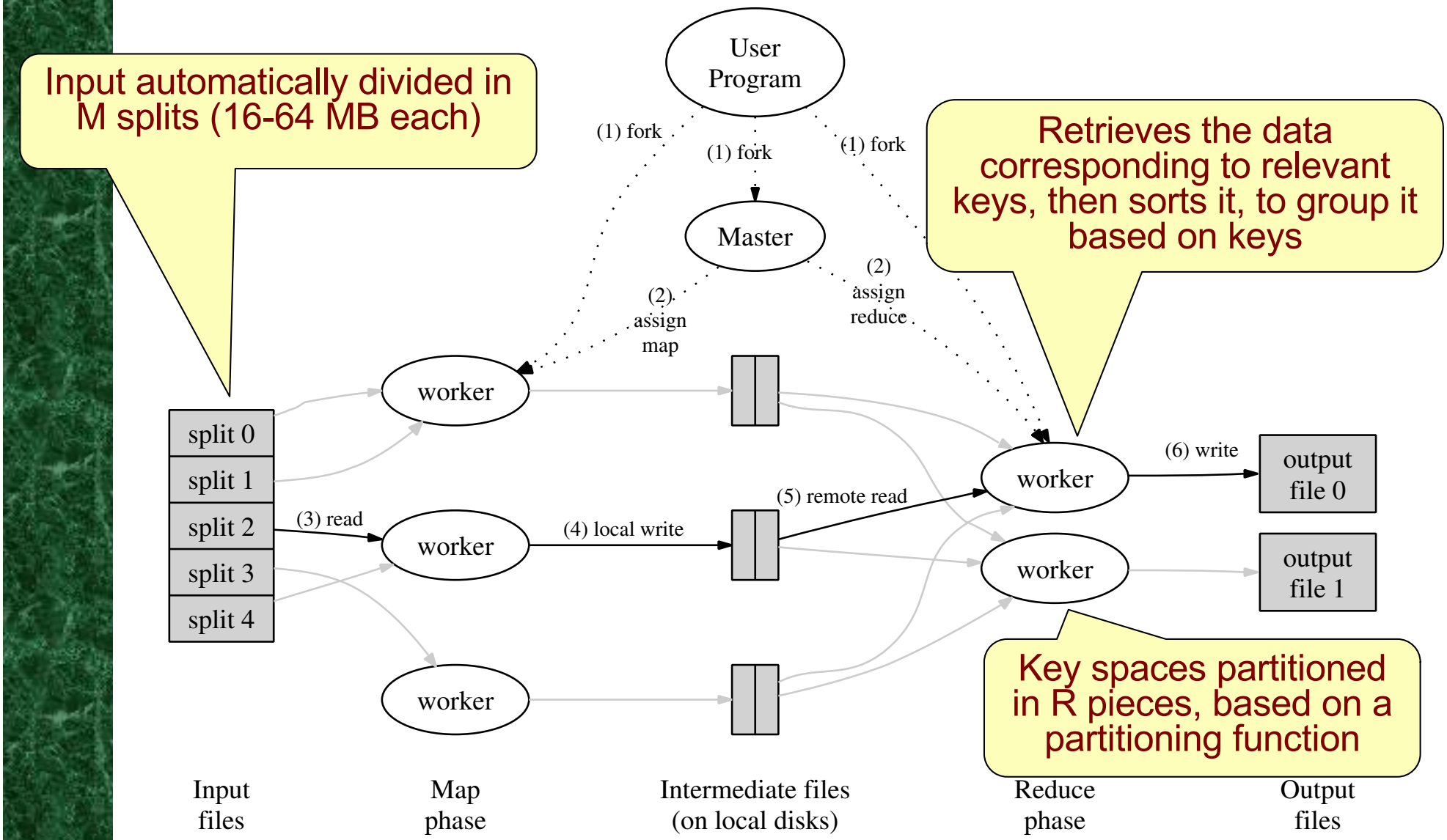
key	TO	BE	OR	NOT	TO	BE
value	1	1	1	1	1	1

```
reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

key	TO	BE	OR	NOT
value	2	2	1	1

- In addition, the user provides a “mapreduce specification” with input and output files, and configuration parameters

MapReduce in Action



Fault Tolerance

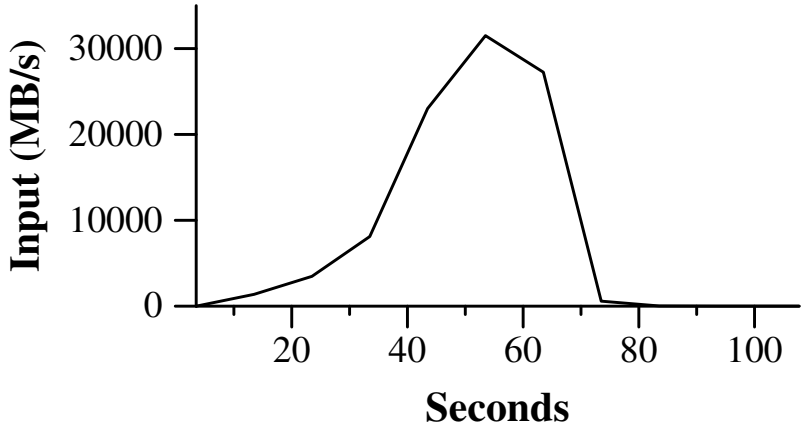
- Worker failure:
  - The master pings every worker periodically
  - If no response is received within a certain time, the worker is marked as failed, and its tasks re-scheduled to other workers
  - completed map tasks must be re-executed because their output is on an inaccessible host
    - Rescheduling is communicated to all reduce workers, to be able to fetch the data from the proper worker
  - completed reduce tasks are instead on GFS and do not need to be re-executed
- Master failure:
  - a new copy is started from checkpointed state
  - Failure is unlikely (w.r.t. the other hosts): if this happens, computation is simply aborted
- The semantics in presence of failures is well-defined and relies on atomic commits of the output of the map and reduce tasks

Allocating Tasks

- Locality:
  - The input data is available to the cluster through GFS, which replicates each file
  - The master tries to allocate map tasks on a machine hosting a replica of the input, or a nearby one
  - Most of the input is read locally, without going through the network
- The map phase is divided into M pieces, and the reduce phase into R
  - M and R should be much greater than the number of hosts in the cluster
  - R is often constrained by users, as it determines the number of output files
  - Typical values are M=200,000 and R=5,000 on a cluster of 2,000 machines
- As usual, performance is dominated by the slowest components (“stragglers”)
  - E.g., due to a bad disk that forces repeated writes
  - when a MapReduce is close to an end, the master schedules “backup” executions of in-progress tasks in parallel with the original: the first that completes terminates the computation
  - Significant improvements, e.g., computations completing 40% faster

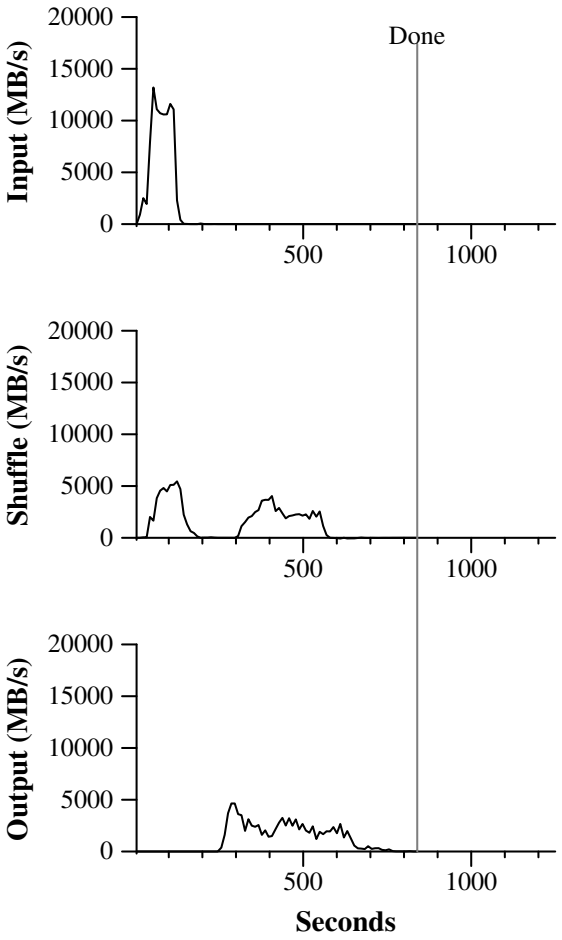
Performance

- Experiments on a cluster of 1800 machines
  - 2GHz Intel Xeon, 4GB RAM, 2 x 160GB IDE HD, gigabit Ethernet , 2-level tree-shaped switched network with about 100-200 Gbps and <1ms RTT
- Computations:
  - **`grep`** scanning through 10^{10} 100B records, searching for a “rare” 3-char pattern (in 92,337 records)
    - M=15,000, R=1
    - About 1min overhead for sending around the code and storing the input on GFS

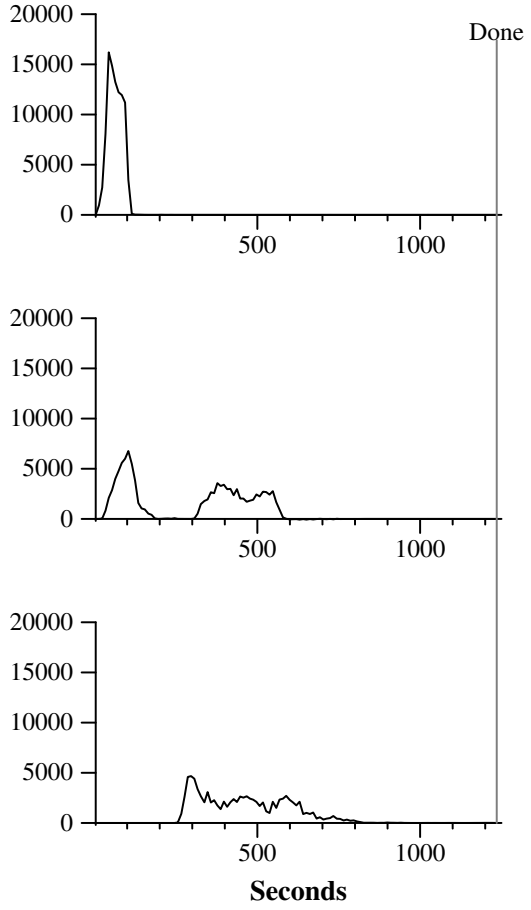


  - Sorting through 10^{10} 100B records (about 1TB)
    - <50 lines of code, M=15,000, R=4,000

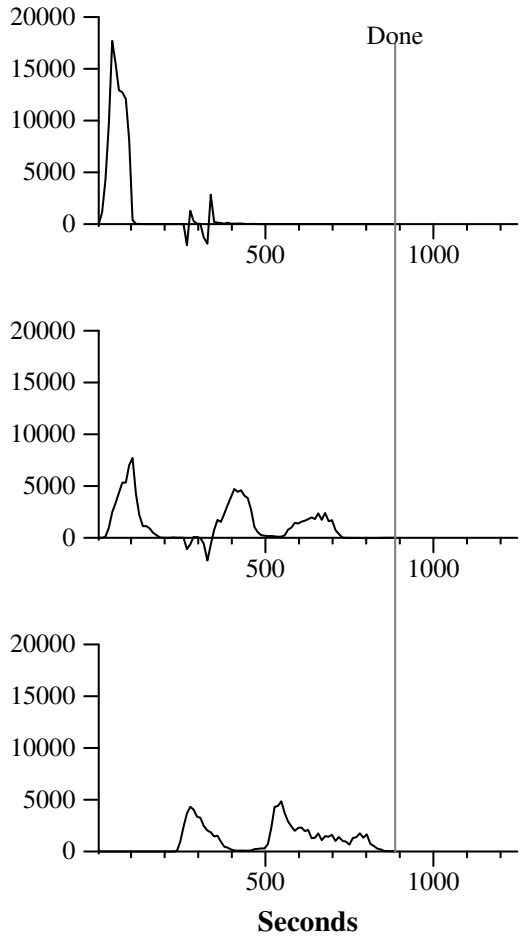
Performance



(a) Normal execution



(b) No backup tasks



(c) 200 tasks killed

Use at Google

- MapReduce is used for a variety of tasks
- Main impact: it led to a complete rewrite of the production indexing system for web search
  - Input: 20TB of data crawled from the Web, stored as GFS files
  - 5 to 10 MapReduce performed on it
  - Code is simpler, smaller, easier to understand: the code dealing with fault tolerance, parallelization, etc. is hidden
  - Easier to operate the service

Number of jobs	29,423
Average job completion time	634 secs
Machine days used	79,186 days
Input data read	3,288 TB
Intermediate data produced	758 TB
Output data written	193 TB
Average worker machines per job	157
Average worker deaths per job	1.2
Average map tasks per job	3,351
Average reduce tasks per job	55
Unique *map* implementations	395
Unique *reduce* implementations	269
Unique *map/reduce* combinations	426

MapReduce jobs in August 2004, on all Google services

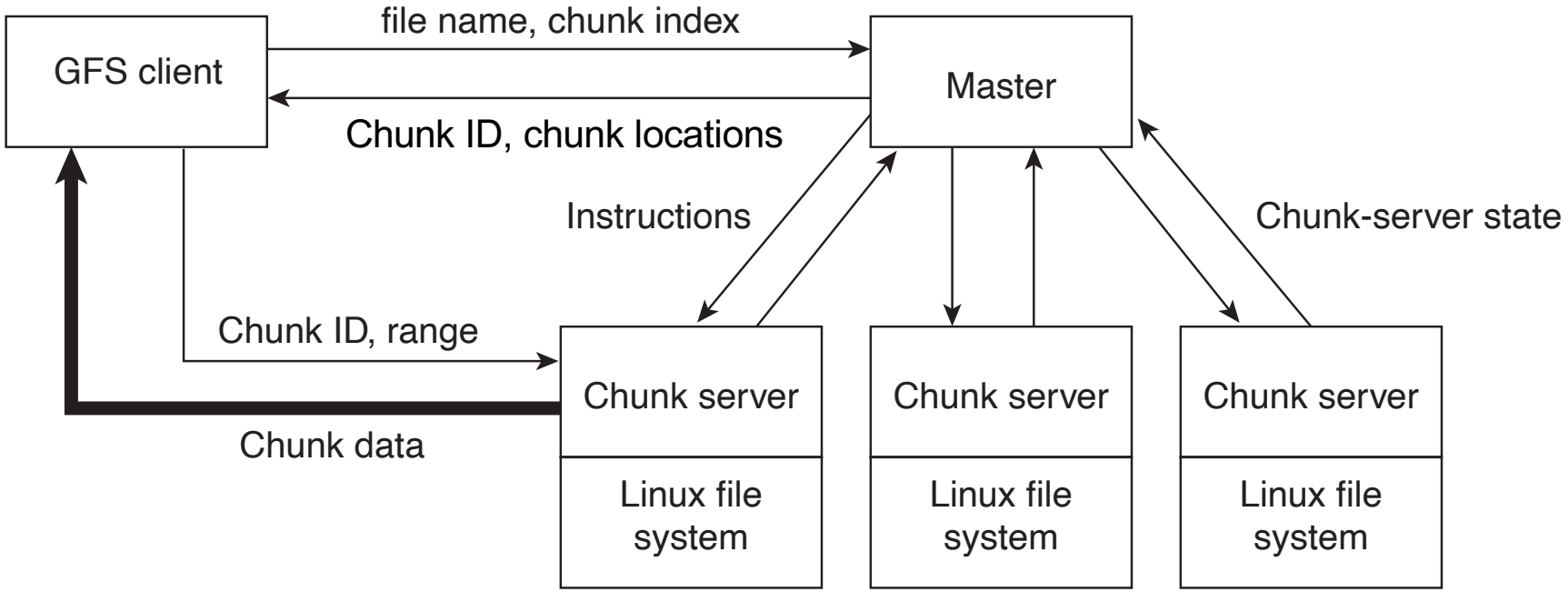
The Google File System (GFS)

- Goal: a ***distributed*** file system customized to the specific needs of Google applications
- Applications in Google:
  - Generate very large files, up to several GB
    - E.g., results of Web crawls
    - Inefficient to parse them using file blocks of KB size, as in traditional file systems
  - Typically write these files as append-only, and read them sequentially
    - E.g., data streams, parsing through big repositories
    - Caching loses most of its appeal
  - Execute in clusters:
    - Hundreds or even thousands of commodity PCs
    - Component failures are the norm, not the exception
    - Big clusters: 1000+ storage nodes, 300+ TB of storage, heavily and continuously accessed by hundreds of distinct clients

Design Assumptions/Goals

- Built out of inexpensive components that often fail
  - Detecting, tolerating, recovering from faults is key
- Must store a “modest” number of large files
  - A few million files! Each 100 MB or larger, multi-GB files are common
  - Access to small files supported but not optimized
- Workload is made of:
  - large streaming reads: read 100 KB – 1 MB or more in our shot, often one after the other
  - small random reads: read a few KB
  - large (see above) sequential writes to append data
- Multiple clients access simultaneously
  - For instance, hundreds of machines concurrently appending to the same file
  - Atomicity, but with minimal synchronization overhead
- High sustained bandwidth more important than low latency

GFS Architecture



- The above serves a single Google cluster
- Files are divided into ***chunks***, 64 MB each
  - stored as files on the chunk server local file system
  - Identified by an immutable, globally-unique chunk handle
  - replicated across multiple servers, to handle host failures
- The client caches the location information for some time
  - Also, clients typically access the chunk through multiple reads: the master is involved only in the first
- Communication relies on RPC

The Master and Scalability

- The architecture is not fully decentralized
  - Having a single master greatly simplifies design
  - Master has global knowledge, can take best decisions in allocating chunks
- The master is critical, however it is kept out of the loop as much as possible
  - It allocates chunks to servers: the allocation view is not actively kept consistent (periodic polling is used instead)
  - It uses this allocation information along with metadata to answer requests by clients: after a request is served, the bulk of the actual work is done by the chunk server

Metadata at the Master

- The master maintains three kinds of metadata:
  - The file and chunk name spaces (directory tree)
    - Entirely contained in memory: less than 64 B of metadata for each 64 MB chunk
  - a simple ⟨filename, chunk server⟩ table, along with the location of replicas
    - Rebuilt upon bootstrap and periodically through polling
  - An ***operation log***, stored on the master and replicated on other remote machines
    - Record of critical changes to metadata (e.g., file/directory creation/deletion)
    - A client operation is replied to only after the corresponding record log is saved on both the master and the log replicas
    - Used to restore the state of the master upon failure
    - Periodically checkpointed

Consistency

- Relaxed consistency, suited for Google apps
- File namespace mutations (e.g., file creation) are atomic, and handled only at the master
  - Consistency guaranteed by namespace locking and the operation log
- Guarantees on data mutations are not as strict
- A data mutation can be a:
  - ***write*** operation: the client decides the offset at which data is placed
  - ***record append*** operation: atomic, at-least-once properties are guaranteed
    - the client provides only the data, the system decides the offset

Consistency

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

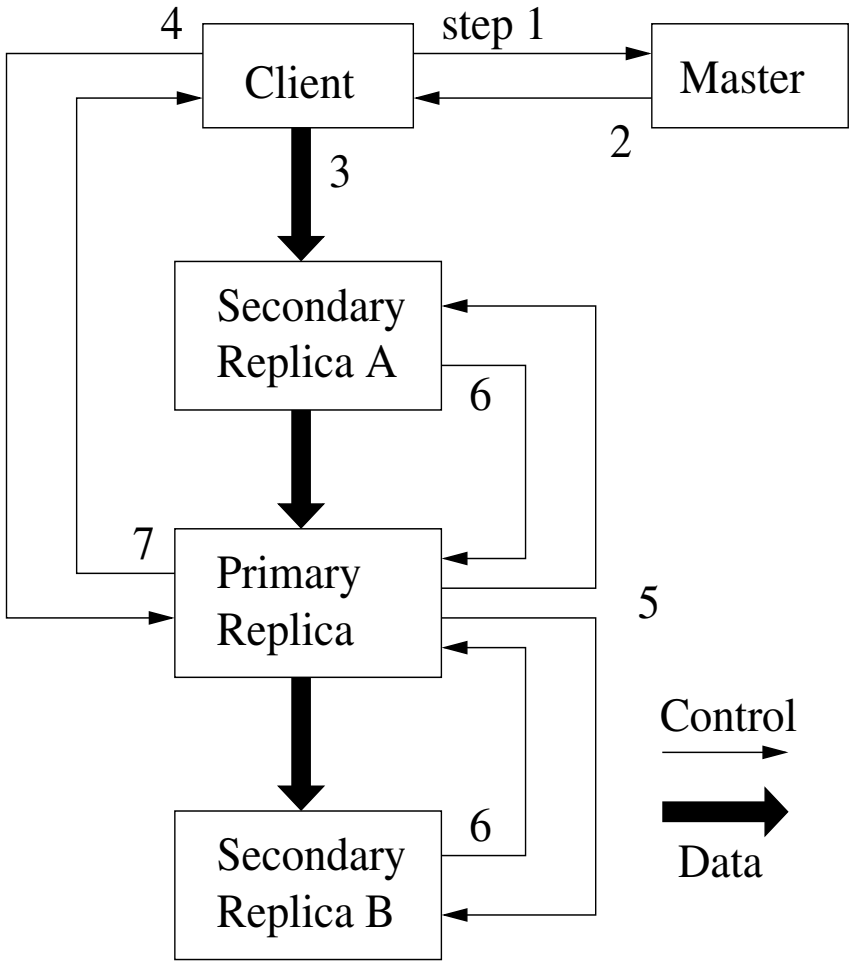
- A file region can be:
  - ***consistent***: all clients see the same data, regardless of the replica accessed;
  - ***defined***: it is consistent and all clients see what the mutation has written
- Consistency guarantees:
  - A sequence of successful mutations leaves the region defined
    - Mutations are applied in the same order to all replicas
    - Chunk version numbers are used to detect stale replicas
  - Concurrent mutations may leave the region consistent but made of bits and pieces from different clients
    - Record appends are always guaranteed to occur atomically, with at-least-once semantics, therefore they are always individually defined
  - A failed mutation may leave the region inconsistent
- It is up to the application to resolve inconsistencies
  - E.g., privilege append to writes, removing duplicates based on IDs

Data Mutation and Replication

- Each data mutation is performed at all replicas
- The master grants a ***chunk lease*** to one of the replicas, the ***primary***
- The rest of the operations are performed by the replicas themselves, without the master
  - The primary picks a serial order for all the incoming mutations to the chunk
  - All replicas follow this order
  - Global mutation order is defined by the lease order (master) and the access order (primary)
  - Leases can be renewed indefinitely, but also revoked by the master
    - E.g., during file renaming

Performing Data Mutation

1. client asks the master for info about the primary
2. the master replies with the address of the primary and the location of secondary replicas
3. client pushes data to the secondary replicas, in a ***chain***
4. after all replicas ack, the client sends request to primary, who picks the order among concurrent requests
5. primary forwards the request to all replicas, that apply writes in the same order
6. The secondary replicas ack the write to the primary
7. The primary replies to the client, reporting errors if any



Pushing files in a chain fully utilizes the bandwidth available; pipelining is used, and the chain is designed to minimize the "distance" of each hop

Namespace Locking

- Some master operations take a long time: cannot simply be executed one at a time
  - E.g., snapshots, essentially copies of files or entire directory trees
- Solution: allow multiple, concurrent operations and use locking over the namespace to ensure serialization
  - Master operations acquire locks before running
  - For instance, an operation on **`/d1/d2/…/dn/leaf`**, will acquire read-locks on **`/d1`**, **`/d1/d2`**, … and either a read- or write-lock on the full path **`/d1/d2/…/dn/leaf`**
  - Still allows multiple mutations (e.g., file creations) in the same directory
    - each acquires a read-lock on the directory, preventing it from being deleted or renamed
    - The write-lock on the file name serializes separate attempts to create a new file with the same name
  - Locks are acquired in order to avoid deadlock
    - based on the namespace tree and then on lexicographic order

Fault Tolerance

- High availability through fast recovery and replication
- Both master and chunkservers designed to restore their state and restart in seconds
  - No distinction between normal and abnormal termination
  - Clients or servers with outstanding requests simply retry
- Each chunk replicated on multiple chunkservers
  - Default: 3 replicas, sitting on different racks
- The master state (operation log and checkpoints) is replicated as well
  - If it fails, it restarts almost instantly
  - If failure is permanent (e.g., disk) an external monitoring client starts a new master on a different machine
  - Clients do not see the change: the DNS name, used for access, remains the same
  - “Shadow” masters (not mirrors!) provide read-only access to the file system structure, by using the same state of the primary master, while the latter is down for a long(er) time
    - only the filesystem structure, actual changes done by the chunkservers

Some Real Data… (2003)

- Cluster A, used for research & development by 100+ engineers
  - Tasks last several hours, read through few MB-TB of data, transform them, and write them back
- Cluster B, used for production data processing
  - Tasks last much longer, continuously generate and process multi-TB data
- See performance in table

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	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

- A chunkserver (15,000 chunks, about 600 GB) was killed in B
  - Content replicated to 40% of the chunkservers (91) in 23.2 mins, at the replication rate of 440 MB/s
- 2 chunkservers killed (each with 16,000 chunks and 660 GB of data)
  - This caused 266 chunks to have a single replica, potentially dangerous
  - Higher priority cloning, bringing them all to at least 2 replicas in 2 mins

MapReduce+GFS vs Hadoop

- Hadoop is an open-source project run by Apache, providing an alternative to the Google proprietary solution
  - Completely written in Java
  - Hadoop Distributed File System (HDFS) provides features analogous to GFS
  - Other file systems are supported
    - e.g., cloud-based ones like the Amazon S3
  - Big community of developers
    - Yahoo! is the main contributor
  - Lots of users
    - Facebook, AOL, eBay, IBM, Joost, LinkedIn, Meebo, The New York Times, Twitter, Fox Interactive Media

Some Considerations

- Many other Google-specific services exist, e.g.:
  - BigTable: distributed storage system for managing *structured* data of very large size (petabytes)
    - 60+ services use it, including Web indexing, Google Analytics, Orkut, Google Earth, …
    - A sort of huge distributed data base
    - Uses the Google File System
  - The Chubby lock service: coarse-grained locking and reliable (low-volume) storage
    - Distributed consensus with asynchronous communication
    - Used by GFS and BigTable
- All of MapReduce, GFS, BigTable, Chubby, use a master to coordinate activities
  - and use replicated server slaves to do the bulk of activities
  - This fits well the majority of their computations, but does not necessarily generalize

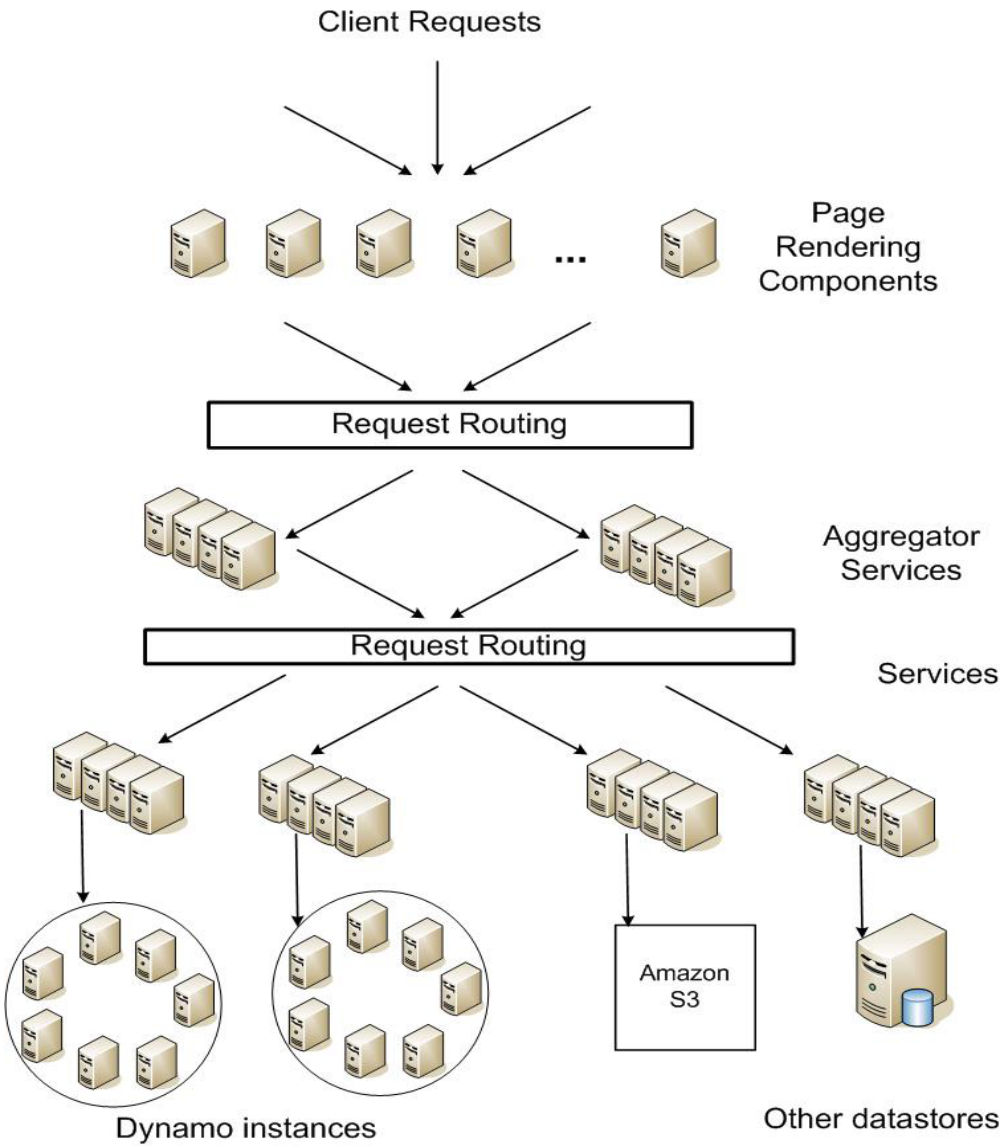
Dynamo

- Developed and operational at Amazon:
  - peaks of tens of millions customers, using tens of thousands of servers all around the world
    - There is always some component failing at any given time
    - In the 2003 holiday season, Amazon processed 1 million shipments, and 20 million inventory updates in one day
    - On 2012 CyberMonday, sold 27 million items, 306 item/s
  - Performance and reliability are key: outages have financial and “image” consequences
  - Several services, very flexible architecture
- Dynamo focuses on services that need only a “primary-key” access
  - E.g., find a product given the identifier, opposed to “searching” for products based on their features
  - Emphasis on high reliability, and the ability to tune the system based on the requirements

Assumptions & Requirements

- Simple read and write operations on a single data item, identified by a key
  - No operation spans multiple objects, no need for a relational schema (as in data bases)
  - Stored objects are relatively small (< 1 MB)
- Weak consistency guarantees
  - Difficult to guarantee ACID properties with high availability: operations are often prevented/delayed to avoid inconsistencies
    - Unacceptable for the user experience
  - The “C” of ACID is sacrificed for higher availability
- Stringent latency requirements
  - A typical service level agreement (SLA) requires that 99.9% of requests are served within 300 ms at a peak of 500 requests/s
  - However, applications must be able to configure the system according to their needs
- System must run on commodity hardware

Amazon's Service-oriented Architecture



- A page request to one of the e-commerce sites requires sending requests to over 150+ services
  - Often rerouted to other services, etc.
- SLA are fundamental to guarantee responsiveness
- Requirements specified as percentiles instead of average/median
  - To ensure all customers have a good experience
  - Typically 99.9%: beyond this is not cost effective

Design Goals

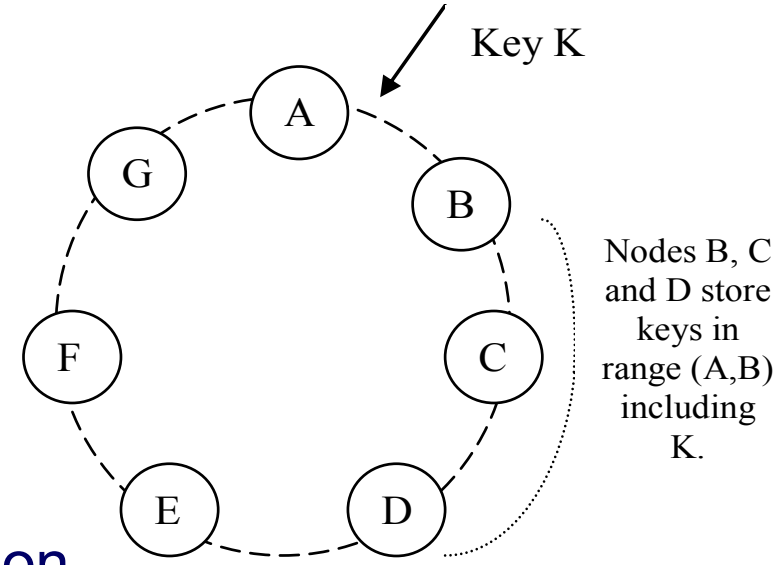
- Eventually-consistent data store
  - Optimistic replication, conflict resolution is key
- "Always writeable data store": high availability when writing
  - In contrast, many traditional data stores perform conflict resolution upon writes, and keep reads simple
  - This would result in a poor user experience for several services (e.g., changing items in the cart)
- Conflict resolution can be done by the data store or by the application
  - Data store can only apply simple policies such as "last write wins"
  - The application has more information: for instance, a typical strategy is to "merge" the contents of two versions of the cart
- Incremental scalability, symmetry and decentralization, heterogeneity (servers with different capabilities)

Overview

- Objects stored are associated with a key
- The programming interface is simple:
  - `get(key)`: locates and returns the object (or a list of objects with conflicting versions) along with a *context*
  - `put(key,context, obj)` determines where the replicas of `obj` should be placed, and writes them to disk
  - context is metadata about the object, opaque to the application (e.g., version info)

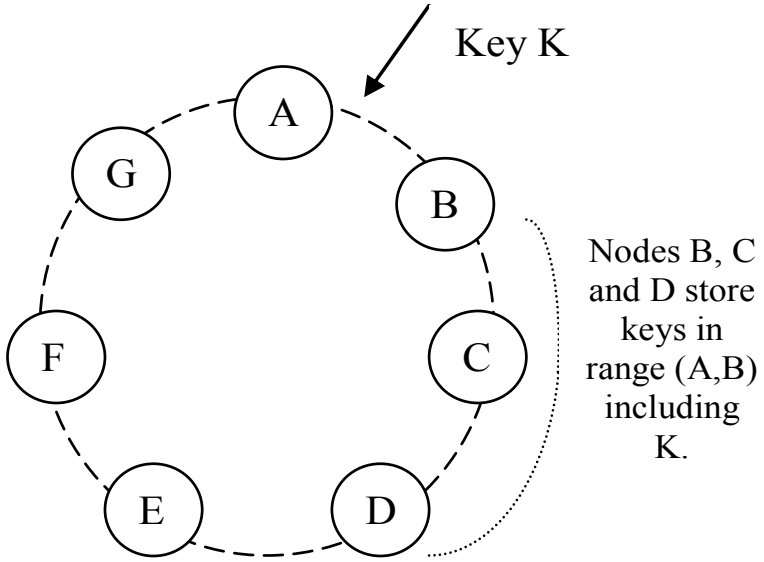
Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Partitioning



- The system must be able to scale incrementally
- Consistent hashing:
  - the output range of a hash function is treated as a fixed circular space (ring)
  - Each node in the system is assigned a random value within the key space, representing its ring “position”
  - Each data item (identified by the key) is assigned to a node by hashing the key to get a position in the ring
    - And then walking clockwise to find the first node
  - Each node becomes responsible for all keys between it and its predecessor
    - Advantage: departure/arrival of a node affects only neighbors
- Similar to distributed hash tables in P2P systems
  - E.g., Chord

Replication



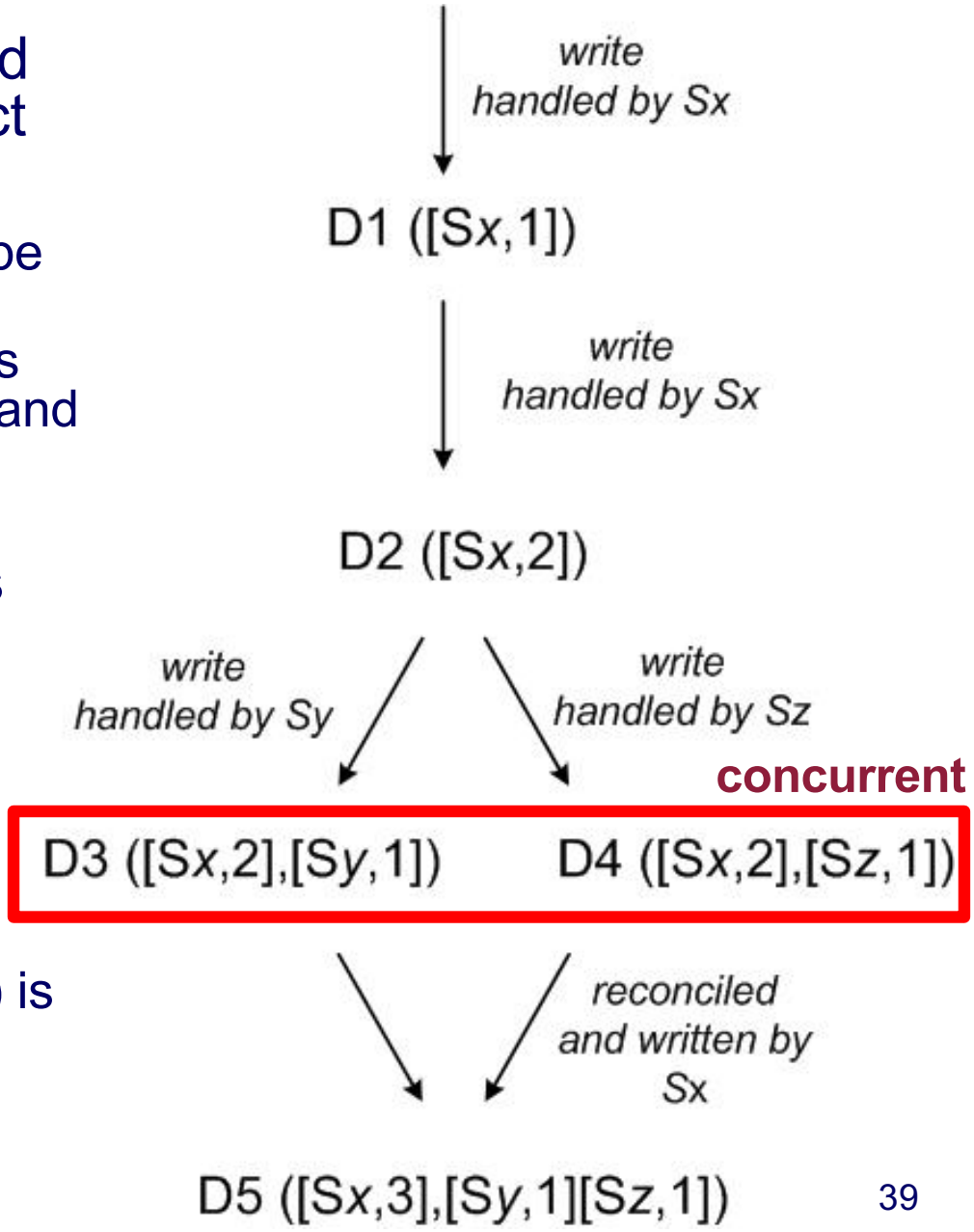
- Each data item is replicated at N hosts, to increase availability and durability
  - N depends on the application
  - Typically, N=3
- Each key is assigned a “coordinator”, in charge of making the N-1 replicas
  - These are written on the clockwise successors of the coordinator in the ring
  - Es., D will store keys in range (A,B], (B,C], (C,D]
- Preference list for a key k: the list of nodes responsible for storing k
  - Contains more than N nodes, to account for failures

Data Versioning

- Eventual consistency: asynchronous propagation of replica updates
  - put() returns immediately without waiting for update propagation at all replicas
  - get() may return an object without the latest updates
- Example: shopping cart
  - "Add to/delete from cart" never forgotten/rejected
    - Add and delete both translate into put() operations
  - if the newer state is unavailable and the user makes a change on an old version, the change must be preserved but cannot supersede what is in the unavailable state
  - Dynamo treats each modification as a new, immutable version of the object
  - Multiple versions can be present at the same time
    - Most of the times, the new subsume the old ones, detected automatically **using vector clocks**
    - In some cases version branch: all versions are brought to client for reconciliation based on the application-level logic
    - For a cart, versions are merged: no item is lost, but deleted items may resurface

Vector Clocks in Dynamo

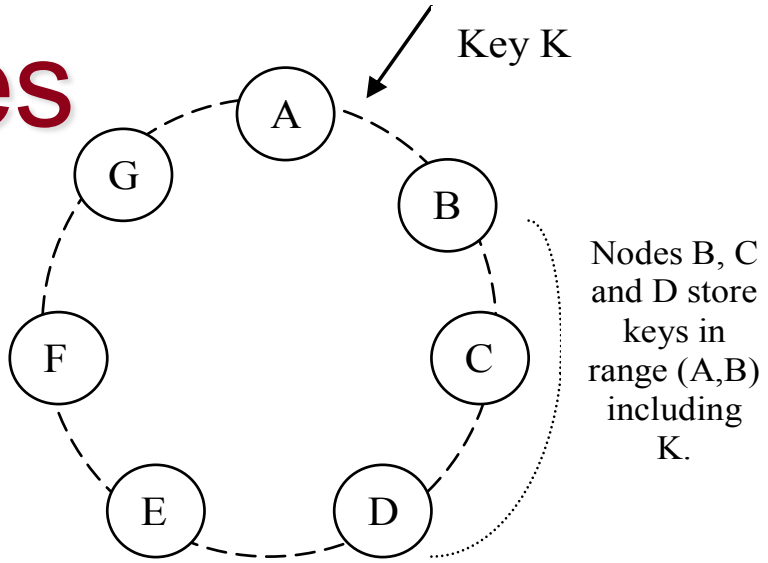
- A vector clock is associated to each version of an object
- Conflict resolution:
  - If $VC_1 < VC_2$, version 1 can be forgotten
  - Otherwise, the two changes are in conflict (concurrent) and must be reconciled
- When a client wishes to update an object, specifies the version
  - By passing the context, contains VCs
- In principle, the VCs can grow indefinitely
  - In practice, setting a high-enough threshold (e.g., 10) is enough to truncate without consistency problems



Executing Reads and Writes

- Any storage node can receive get() and put() operations for any key k
  - Called the coordinator: in principle the first node in the preference list for k
- Consistency is maintained using a quorum
  - Upon receiving a put(k, context, obj):
    - the coordinator generates the VC for the new version and writes the new object locally
    - The new version is sent (along with VC) to the N highest-ranked reachable nodes in preference list for k
    - if at least W-1 reply, the write is successful
  - Upon receiving a get(k):
    - The coordinator requests all existing versions of the object for k from the N highest-ranked reachable nodes in the preference list for k
    - When at least R responses are received, it evaluates whether reconciliation is needed or not
  - A common configuration is N=3, R=W=2

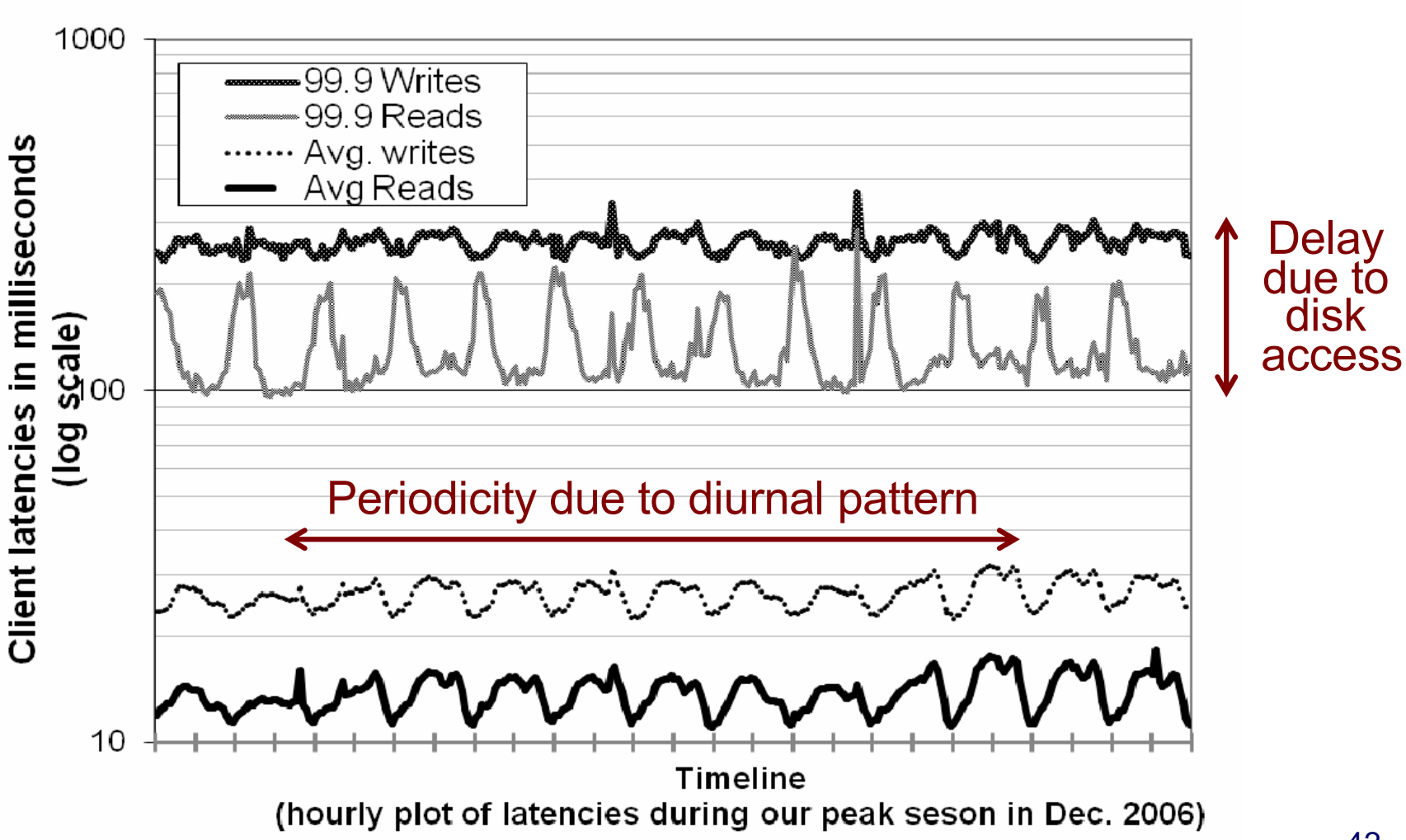
Dealing with Failures



- Traditional quorum causes problems to availability/durability in presence of server failures and network partitions
- Dynamo uses a "sloppy quorum" where read/write operations are performed on the first N healthy nodes in the preference list
  - Not necessarily the first N encountered when walking on the ring
  - Example:
    - if A is down during a write for an object it manages, the new object will be sent to D (even though it is not responsible for it) to maintain availability and durability
    - D has a "hint" in the metadata, saying that the replica was intended for A: the replica is kept in a separate store at D
    - When D detects that A is back, the replica is delivered to A and removed from D
- An anti-entropy protocol is used to resolve inconsistencies in the keys assigned to a node
- A gossip-based protocol is used to propagate membership changes due to nodes joining/leaving the ring
- Data must survive the failure of entire data centers
  - Replicas are performed on nodes belonging to different data centers, connected through high-speed links

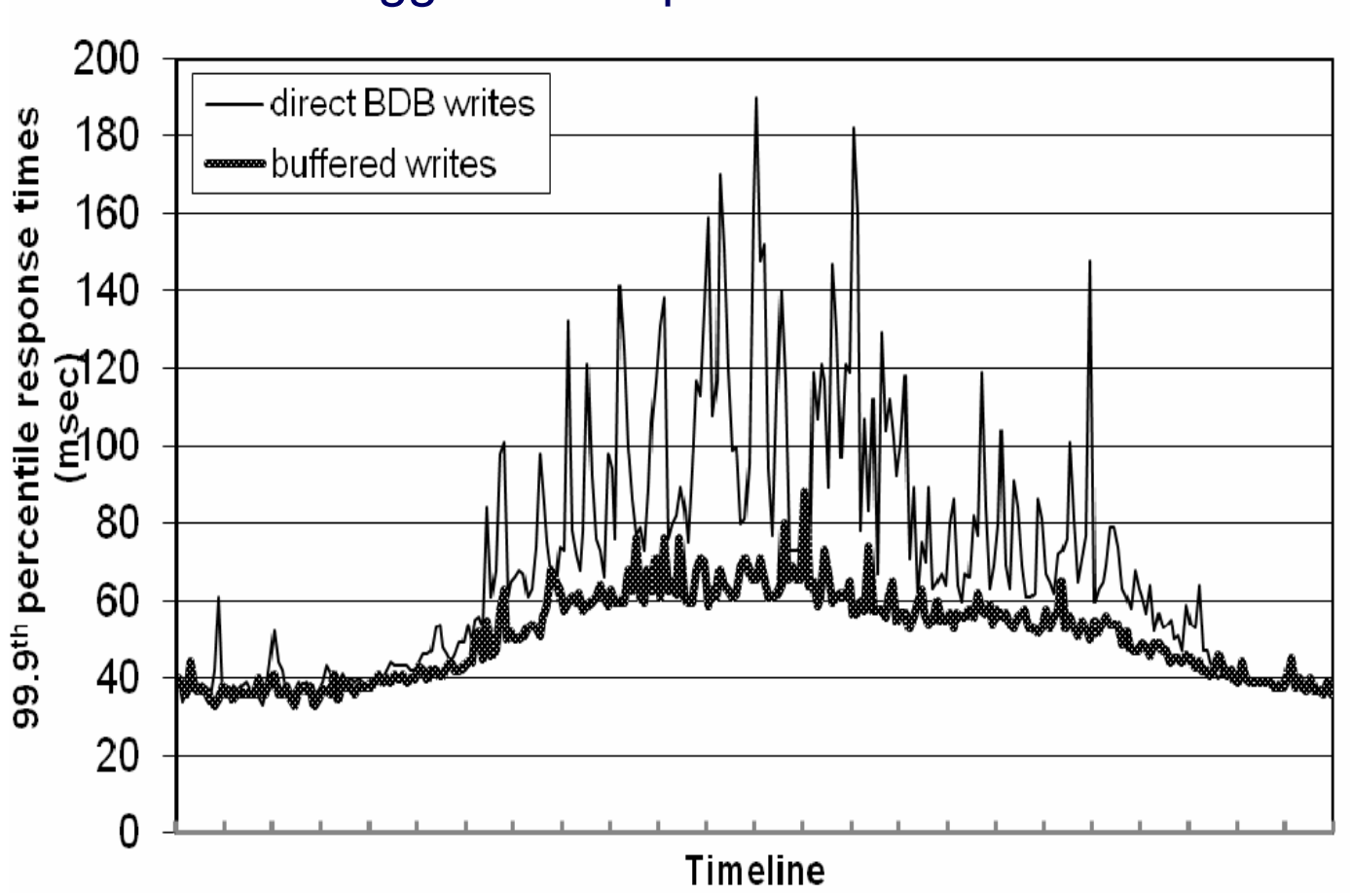
Performance

- Typical SLA agreement: 99.9% of reads execute within 300ms



Performance

- Impact of disk access can be reduced by trading off durability for performance
  - Can be chosen by the application
- Use of a buffer, periodically written to disk
  - Performance improves by a 5x factor
  - A crash has bigger consequences



Divergent Versions in Practice

- Measurements on the number of divergent versions seen by the application in production
- Divergent versions arise due to:
  - Failures of nodes or data centers, network partitions
  - Large numbers of concurrent writes, coordinated by different nodes
  - If the system cannot reconcile automatically, the burden goes to the client
- Observation during a 24h-period for the shopping cart service:
  - 1 version: 99.94%; 2 versions: 0.00057%; 3 versions: 0.00047%; 4 versions: 0.00009%
- In practice, the cause is almost always concurrent writes caused by automated client services, rarely by humans
  - *"This issue is not discussed in detail due to the sensitive nature of the story"*

References

- L.A. Barroso, J. Dean, and U. Holzle, “Web Search for a Planet: The Google Cluster Architecture,” *IEEE Micro*, 23(2), 2003, p. 22–28.
- G. DeCandia et al. “Dynamo: Amazon’s highly available key-value store,” *21st Symp. on Operating Systems Principles (SOSP)*, ACM, 2007, pp. 205-220.
- J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *5th Symp. on Operating Systems Design and Implementation (OSDI)*, 2004, pp. 137-150.
- S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google file system,” *17th Symp. on Operating Systems Principles (SOSP)*, 2003, pp. 29-43.
- M. Burrows, “The Chubby lock service for loosely-coupled distributed systems,” *7th Symp. on Operating Systems Design and Implementation (OSDI)*, 2006.
- D. Beaver, S. Kumar, H.C. Li, J. Sobel, and P. Vajgel, “Finding a needle in Haystack: Facebook’s photo storage,” *9th Symp. on Operating Systems Design and Implementation (OSDI)*, 2010
- F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R.E. Gruber, “Bigtable: A distributed storage system for structured data,” *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, ACM, 2006.