Introduction to Distributed Data Systems

Serge Abiteboul Ioana Manolescu Philippe Rigaux
Marie-Christine Rousset Pierre Senellart

Web Data Management and Distribution
http://webdam.inria.fr/textbook

January 13, 2014

Outline

1 Introduction

2 Overview of distributed data management principles

3 Properties of a distributed system

4 Failure management

5 Consistent hashing

6 Case study: large scale distributed file systems (GFS)

Outline

Guidelines and principles for distributed data management

- Basics of distributed systems (networks, performance, principles)
- Failure management and distributed transactions
- Properties of a distributed system
- Specifics of peer-to-peer networks
- A powerful and resilient distribution method: consistent hashing
- Case study: GFS, a distributed file system

Outline

1 Introduction

2 Overview of distributed data management principles
- Data replication and consistency

3 Properties of a distributed system

4 Failure management

5 Consistent hashing

6 Case study: large scale distributed file systems (GFS)

Distributed systems

A distributed system is an application that coordinates the actions of several computers to achieve a specific task.

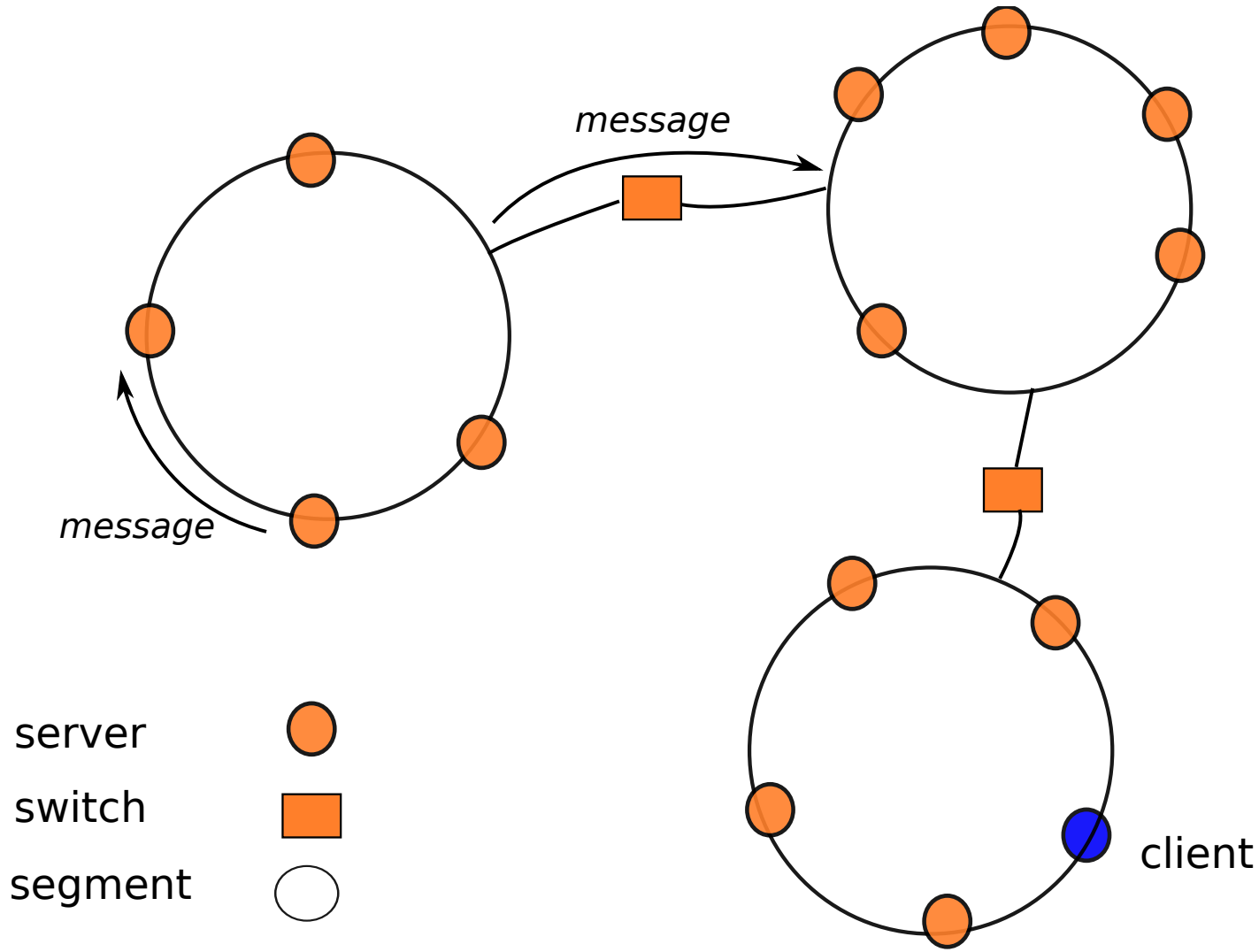
This coordination is achieved by exchanging messages which are pieces of data that convey some information.
$\Rightarrow$ "shared-nothing" architecture -> no shared memory, no shared disk.

The system relies on a network that connects the computers and handles the routing of messages.
$\Rightarrow$ Local area networks (LAN), Peer to peer (P2P) networks...

Client (nodes) and Server (nodes) are communicating software components: we assimilate them with the machines they run on.

LAN-based infrastructure: clusters of machines

Three communication levels: “racks”, clusters, and groups of clusters.



Example: data centers

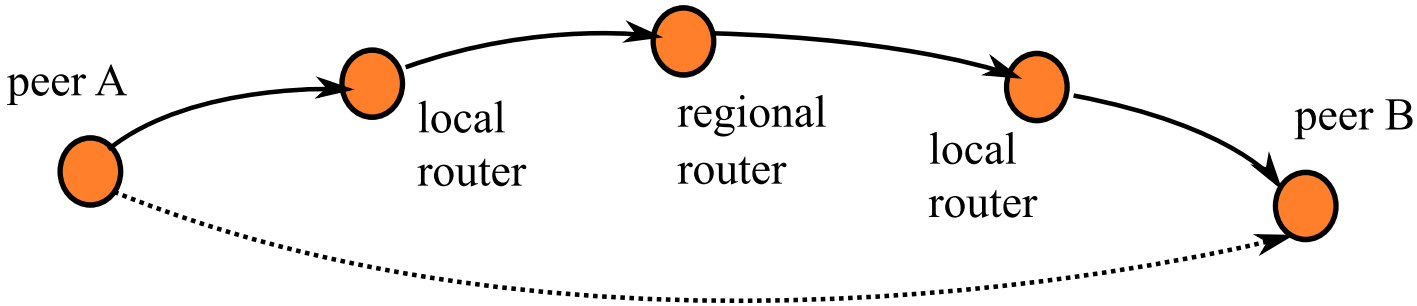
Typical setting of a Google data center.

1. $\approx$ 40 servers per rack;
2. $\approx$ 150 racks per data center (cluster);
3. $\approx$ 6,000 servers per data center;
4. how many clusters? Google's secret, and constantly evolving . . .

Rough estimate: 150-200 data centers? 1,000,000 servers?

P2P infrastructure: Internet-based communication

Nodes, or “peers” communicate with messages sent over the Internet network.



The physical route may consist of 10 or more forwarding messages, or “hops”.

Suggestion: use the `traceroute` utility to check the route between your laptop and a Web site of your choice.

Performance

Type	Latency	Bandwidth
Disk	$\approx 5 \times 10^{-3}$s (5 millisec.);	At best 100 MB/s
LAN	$\approx 1 - 2 \times 10^{-3}$s (1-2 millisec.);	$\approx$ 1Gb/s (single rack); $\approx$ 100Mb/s (switched);
Internet	Highly variable. Typ. 10-100 ms.;	Highly variable. Typ. a few MB/s.;

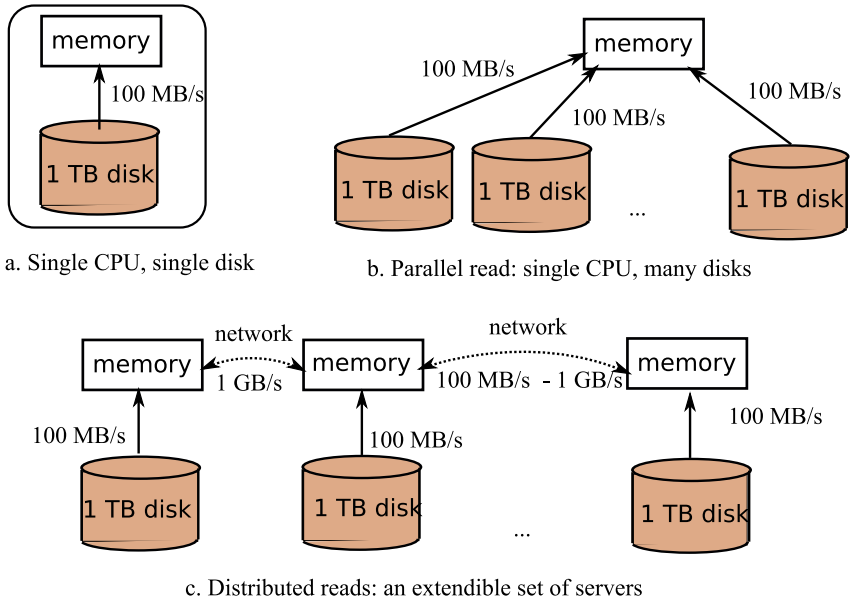
Bottom line (1): it is approx. one order of magnitude faster to exchange main memory data between 2 machines in the same rack of a data center, that to read on the disk.
Bottom line (2): exchanging through the Internet is slow and unreliable with respect to LANs.

Distribution, why?

Sequential access. It takes 166 minutes (more than 2 hours and a half) to read a 1 TB disk.

Parallel access. With 100 disks, assuming that the disks work in parallel and sequentially: about 1mn 30s.

Distributed access. With 100 computers, each disposing of its own local disk: each CPU processes its own dataset.



Scalability

The latter solution is *scalable*, by adding new computing resources.

What you should remember: performance of data-centric distr. systems

1. disk transfer rate is a bottleneck for large scale data management; parallelization and distribution of the data on many machines is a means to eliminate this bottleneck;
2. *write once, read many*: a distributed storage system is appropriate for large files that are written once and then repeatedly scanned;
3. *data locality*: bandwidth is a scarce resource, and program should be "pushed" near the data they must access to.

A distr. system also gives an opportunity to reinforce the security of data with replication.

What is it about?

Replication: a mechanism that copies data item located on a machine *A* to a remote machine *B*

$\Rightarrow$ one obtains replica

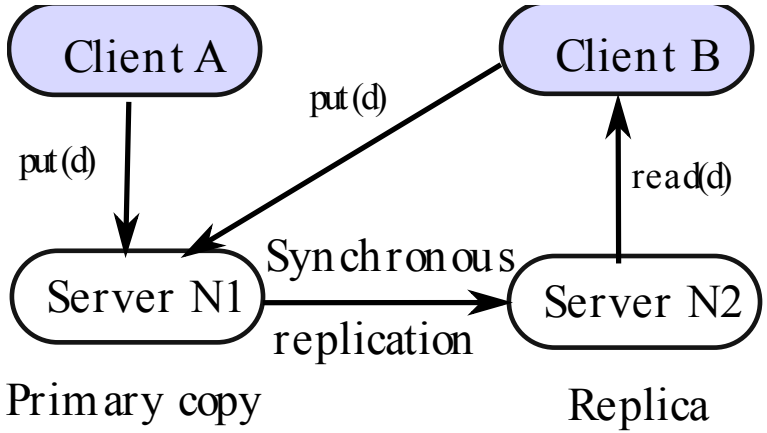
Consistency: ability of a system to behave as if the transaction of each user always run in isolation from other transactions, and never fails.

Example: shopping basket in an e-commerce application.

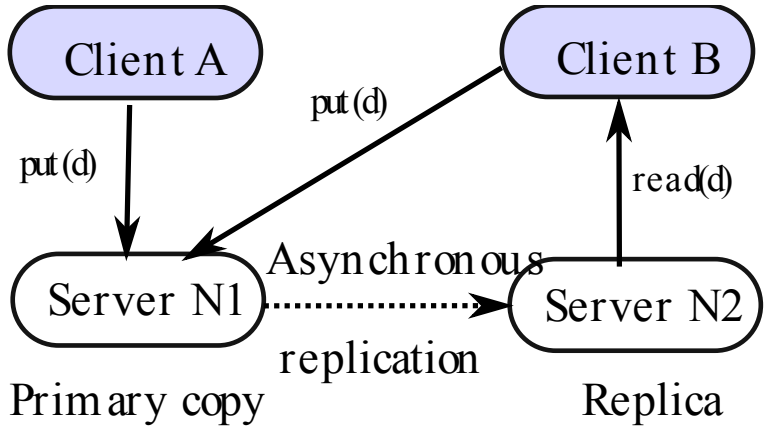
$\Rightarrow$ difficult in centralized systems because of multi-users and concurrency.

$\Rightarrow$ even more difficult in distributed systems because of replica.

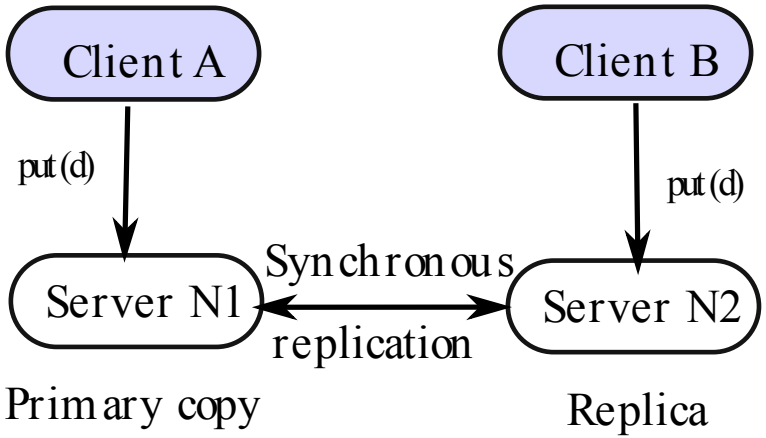
Some illustrative scenarios



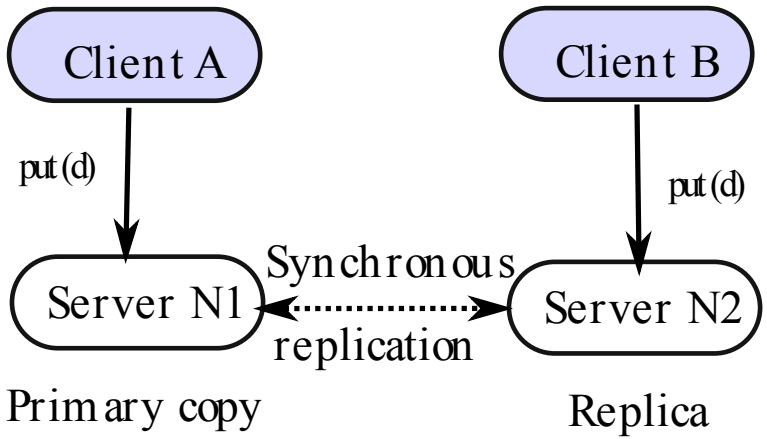
a) Eager replication with primary copy



b) Lazy replication with primary copy



c) Eager replication, distributed



d) Lazy replication, distributed

Consistency management in distr. systems

Consistency: essentially, ensures that the system faithfully reflects the actions of a user.

- Strong consistency (ACID properties) – requires a (slow) synchronous replication, and possibly heavy locking mechanisms.
- Weak consistency – accept to serve some requests with outdated data.
- Eventual consistency – same as before, but the system is guaranteed to converge towards a consistent state based on the last version.

In a system that is not eventually consistent, conflicts occur and the application must take care of data reconciliation: given the two conflicting copies, determine the new current one.

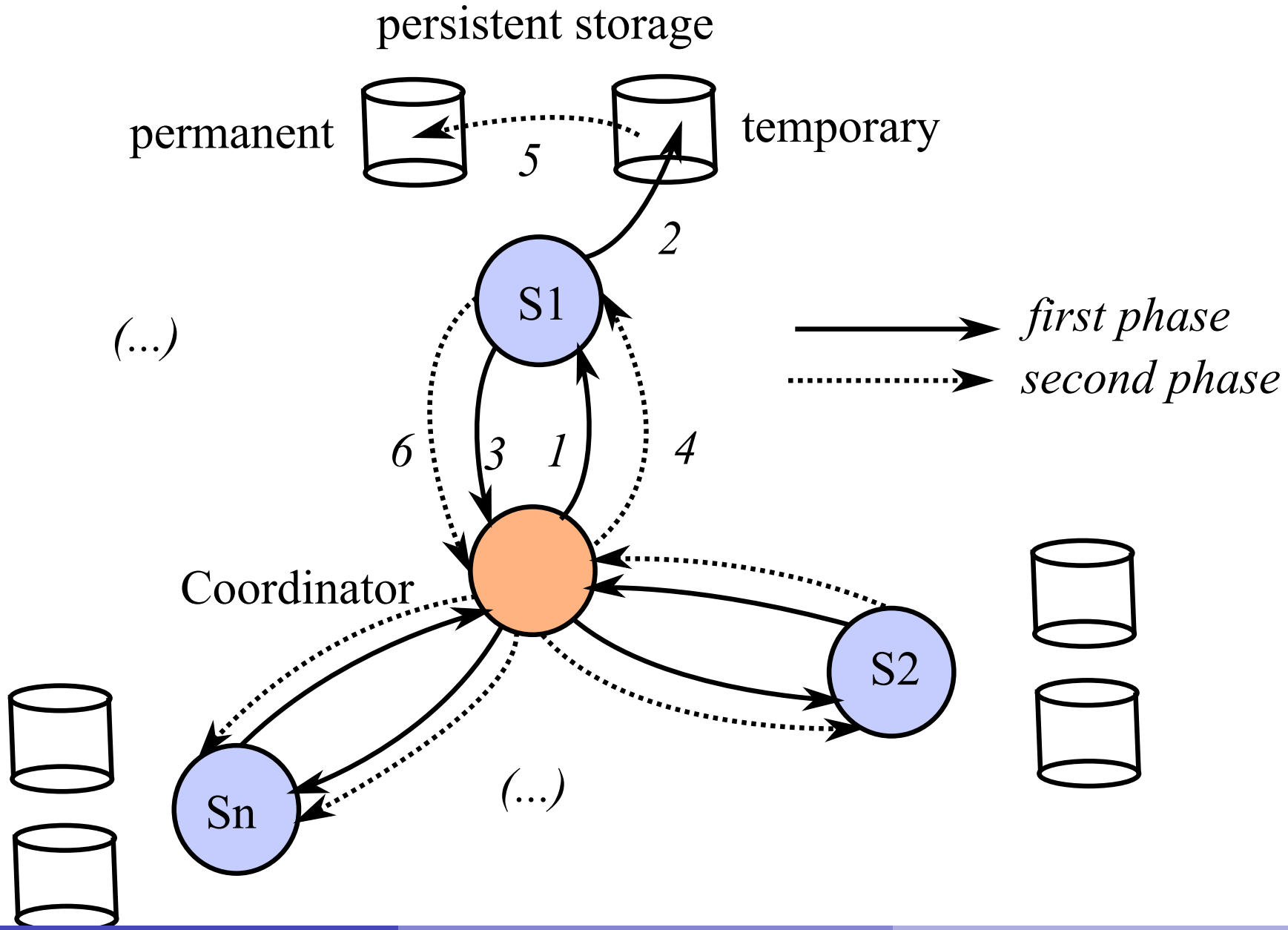
Standard RDBMS favor consistency over availability – one of the reasons (?) of the 'NoSQL' trend.

Achieving strong consistency – The 2PC protocol

2PC = Two Phase Commit. The algorithm of choice to ensure ACID properties in a distributed setting.

Problem: the update operations may occur on distinct servers $\{S_1, \ldots, S_n\}$, called *participants*.

Two Phase Commit: Overview



Exercises and questions

You are a customer using an e-commerce application which is known to be eventually consistent (e.g., Amazon . . .):

1. Show a scenario where you buy an item, but this item does not appear in your basket.
2. You reload the page: the item appears. What happened?
3. You delete an item from your command, and add another one: the basket shows both items. What happened?
4. Will the situation change if you reload the page?
5. Would you expect both items to disapear in an e-commerce application?

Outline

1 Introduction

2 Overview of distributed data management principles

3 Properties of a distributed system

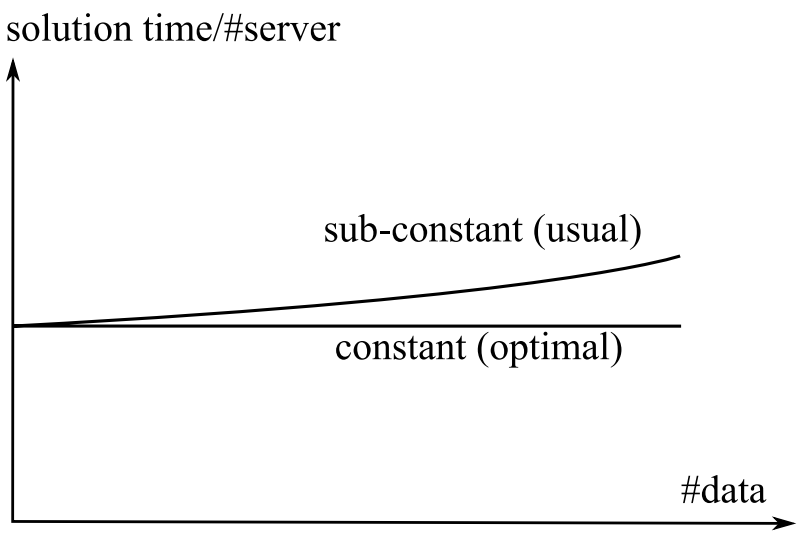
4 Failure management

5 Consistent hashing

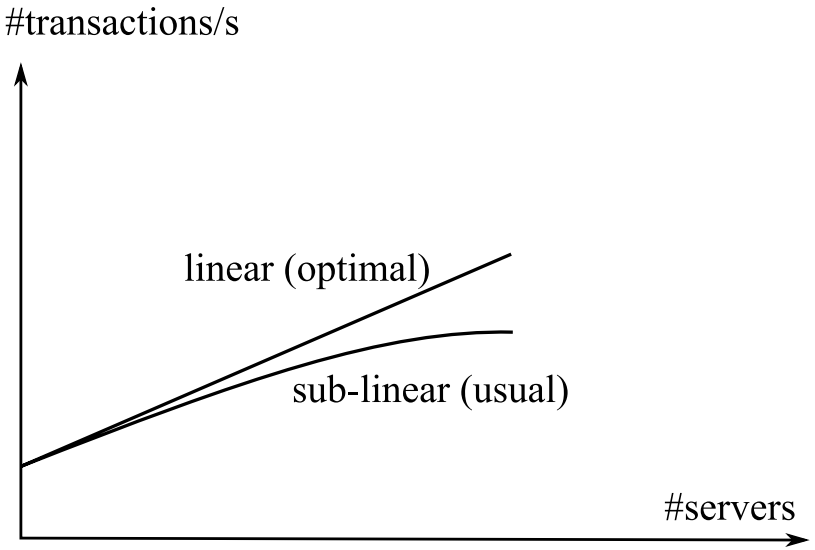
6 Case study: large scale distributed file systems (GFS)

Properties of a distributed system: (i) scalability

Scalability refers to the ability of a system to continuously evolve in order to support an evergrowing amount of tasks.



Weak scaling: the solution time per server remains constant as the dataset grows



Strong scaling: the global throughput raises linearly with the number of servers (fixed problem size)

A scalable system should (i) distribute evenly the task load to all participants, and (ii) ensure a negligible distribution management cost.

Properties of a distributed system: (ii) efficiency

Two usual measures of its efficiency are the response time (or latency) which denotes the delay to obtain the first item, and the throughput (or bandwidth) which denotes the number of items delivered in a given period unit (e.g., a second).

Unit costs:

1. *number of messages* globally sent by the nodes of the system, regardless of the message size;
2. *size of messages* representing the volume of data exchanges.

Properties of a distributed system: (iii) availability

Availability is the capacity of of a system to limit as much as possible its latency.
Involves several aspects:

- Failure detection.
  - monitor the participating nodes to detect failures as early as possible (usually via “heartbeat” messages);
  - design quick restart protocols.
- Replication on several nodes.
  Replication may be synchronous or asynchronous. In the later case, a Client *write()* returns before the operation has been completed on each site.

Outline

1 Introduction

2 Overview of distributed data management principles

3 Properties of a distributed system

4 Failure management

5 Consistent hashing

6 Case study: large scale distributed file systems (GFS)

Failure recovery in centralized DBMSs

Common principles:

1. The state of a (data) system is the set of item committed by the application.
2. Updating "in place" is considered as inefficient because of disk seeks.
3. Instead, update are written in main memory *and* in a sequential log file.
4. Failure? The main memory is lost, but all the committed transactions are in the log: a REDO operations is carried out when the system restarts.

$\Rightarrow$ implemented in all DBMSs.

Failure and distribution

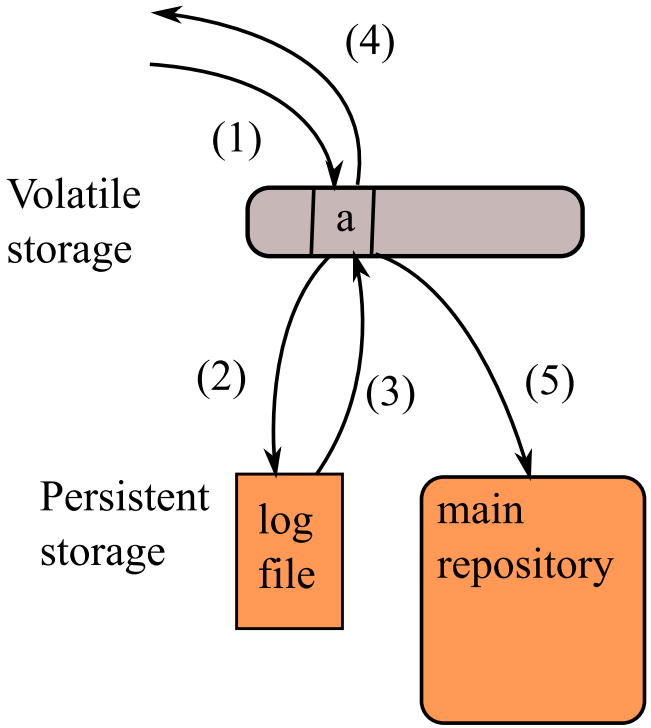
First: when do we kown that a component is in failed state?

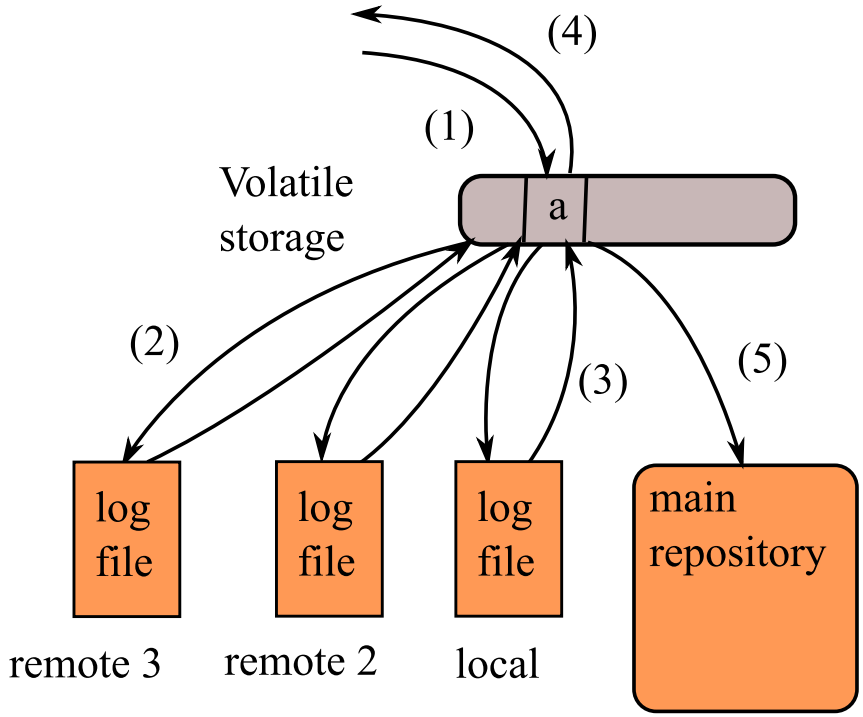
$\Rightarrow$ periodically send message to each participant.

Second: does the centralized recovery still hold?

Yes, providing the log file is accessible . . .

Distributed logging





Note: distributed logging can be asynchronous (efficient, risky) or asynchronous (just the opposite).

Outline

1 Introduction

2 Overview of distributed data management principles

3 Properties of a distributed system

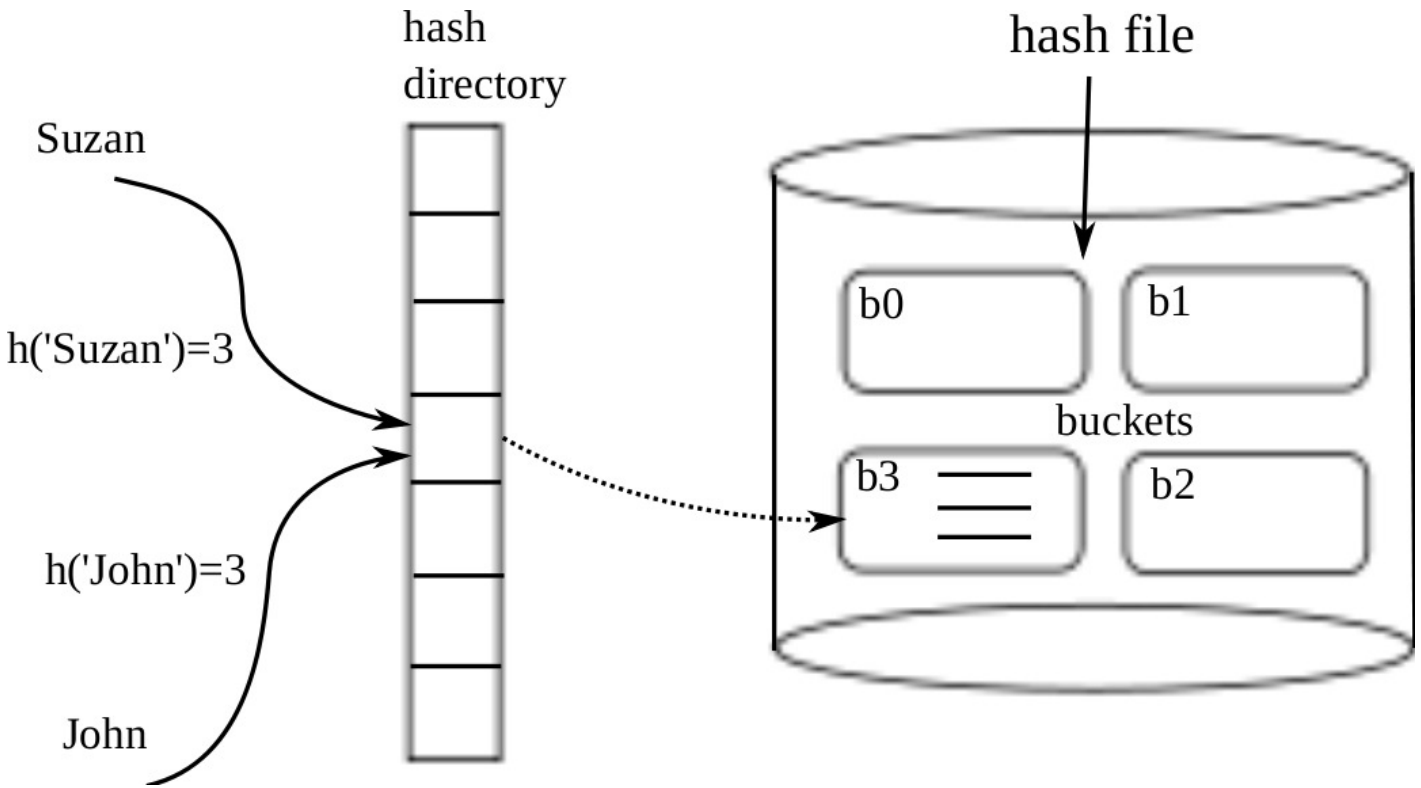
4 Failure management

5 Consistent hashing

6 Case study: large scale distributed file systems (GFS)

Basics: Centralized Hash files

The collection consists of $(key, value)$ pairs. A hash function evenly distributes the values in buckets w.r.t. the key.



This is the basic, static, scheme: the number of buckets is fixed.
Dynamic hashing extends the number of buckets as the collection grows – the most popular method is linear hashing.

Issues with hash structures distribution

Straighforward idea: everybody uses the same hash function, and buckets are replaced by servers.

Two issues:

- Dynamicity. At Web scale, we must be able to add or remove servers at any moment.
- Inconsistencies. It is very hard to ensure that all participants share an accurante view of the system (e.g., the hash function).

Some solutions:

- Distributed linear hashing: sophisticated scheme that allows Client nodes to use an outdated image of the has file; guarantees eventual convergence.
- Consistent hashing: to be presented next.
  NB: consistent hashing is used in several systems, including Dynamo (Amazon)/Voldemort (Open Source), and P2P structures, e.g. Chord.

Consistent hashing

Let N be the number of servers. The following functions

$$hash(key) \rightarrow modulo(key, N) = i$$

maps a pair $(key, value)$ to server S_i.

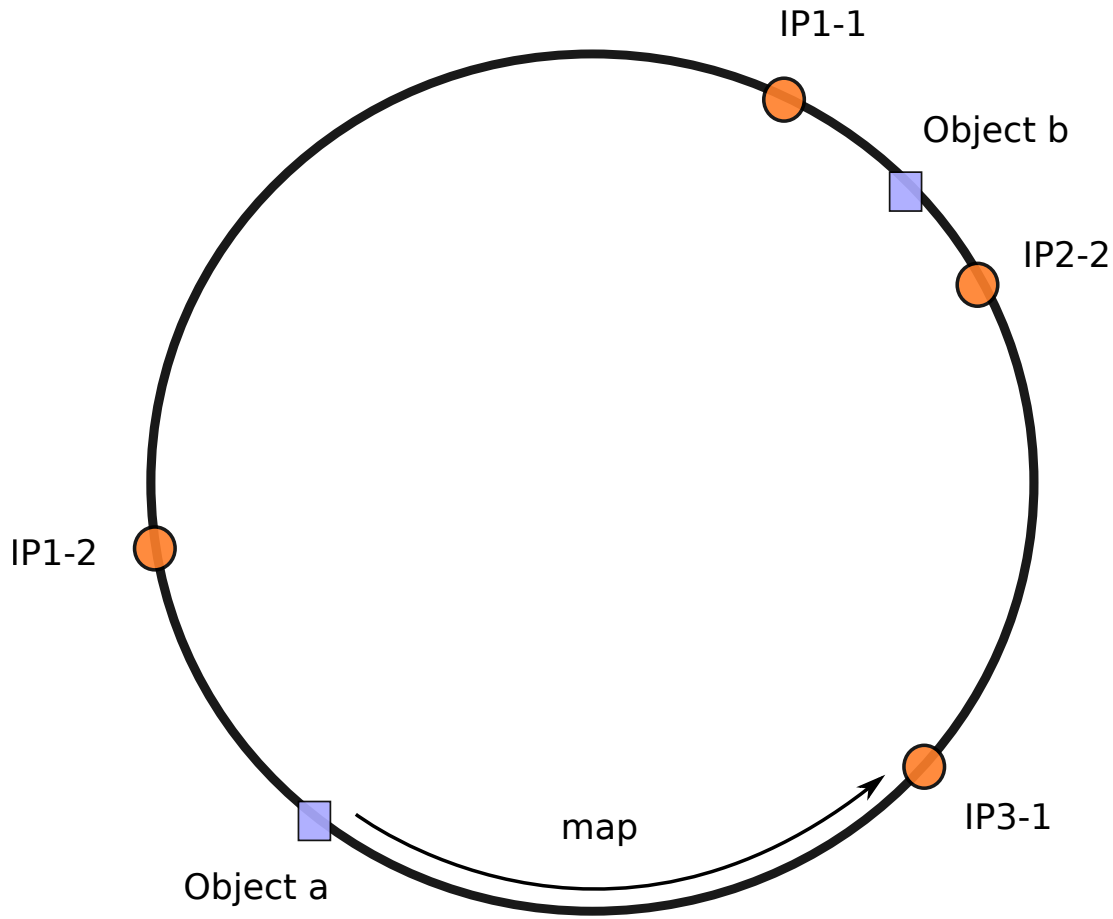
Fact: if N changes, or if a client uses an invalid value for N, the mapping becomes inconsistent.

With Consistent hashing, addition or removal of an instance does not significantly change the mapping of keys to servers.

- a simple, non-mutable hash function h maps both the keys ad the servers IPs to a large address space A (e.g., $[0, 2^{64} - 1]$);
- A is organized as a ring, scanned in clockwise order;
- if S and S' are two adjacent servers on the ring: all the keys in range $]h(S), h(S')]$ are mapped to S'.

Illustration

Example: item *A* is mapped to server IP1-2; item B to server . . .



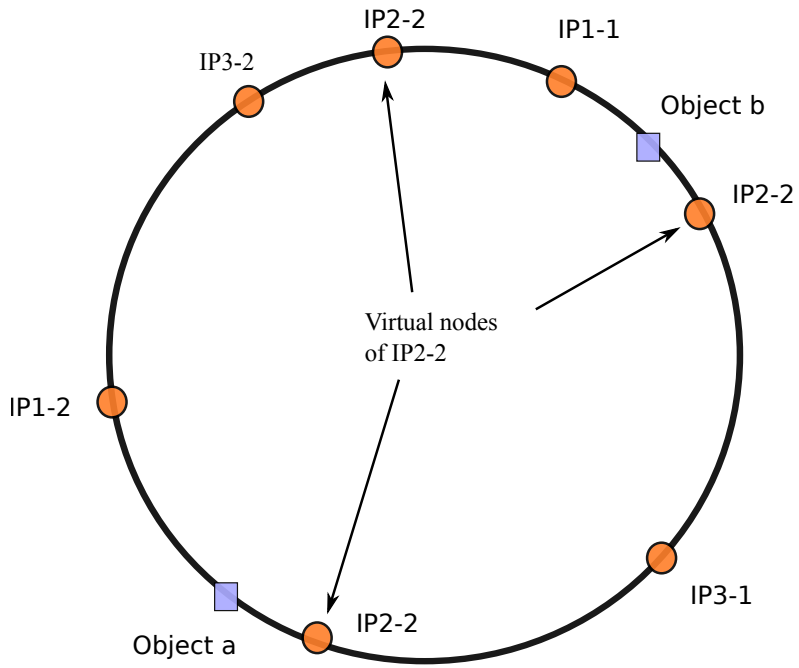
A server is added or removed? A local re-hashing is sufficient.

Some (really useful) refinements

What if a server fails? How can we balance the load?

Failure $\Rightarrow$ use replication; put a copy on the next machine (on the ring), then on the next after the next, and so on.
Load balancing $\Rightarrow$ map a server to several points on the ring (virtual nodes)

- the more points, the more load received by a server;
- also useful if the server fails: data rellocation is more evenly distributed.
- also useful in case of heterogeneity (the rule in large-scale systems).



Distributed indexing based on consistent hashing

Main question: where is the hash directory (servers locations)? Several possible answers:

- On a specific (“Master") node, acting as a load balancer. Example: caching systems.
  $\Rightarrow$ raises scalability issues.
- Each node records its successor on the ring.
  $\Rightarrow$ may require $O(N)$ messages for routing queries – not resilient to failures.
- Each node records $\log N$ carefully chosen other nodes.
  $\Rightarrow$ ensures $O(\log N)$ messages for routing queries – convenient trade-off for highly dynamic networks (e.g., P2P)
- Full duplication of the hash directory at each node.
  $\Rightarrow$ ensures 1 message for routing – heavy maintenance protocol which can be achieved through gossiping (broadcast of any event affecting the network topology).

Case study: Dynamo (Amazon)

A distributed system that targets high availability (your shopping cart is stored there!).

- Duplicates and maintains the hash directory at each node via gossiping – queries can be routed to the correct server with 1 message.
- The hosting server replicates N (application parameter) copies of its objects on the N distinct nodes that follow S on the ring.
- Propagates updates asynchronously $\rightarrow$ may result in update conflicts, solved by the application at read-time.
- Use a fully distributed failure detection mechanism (failure are detected by individual nodes when then fail to communicate with others)

An Open-source version is available at *http://project-voldemort.com/*

Outline

1 Introduction

2 Overview of distributed data management principles

3 Properties of a distributed system

4 Failure management

5 Consistent hashing

6 Case study: large scale distributed file systems (GFS)

History and development of GFS

Google File System, a paper published in 2003 by Google Labs at OSDI.

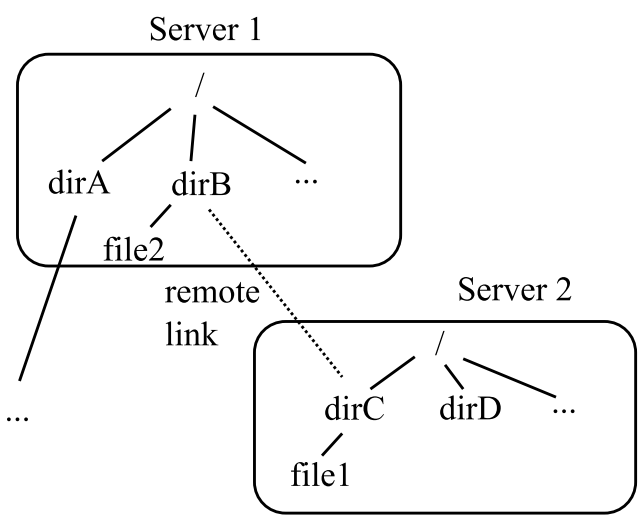
Explains the design and architecture of a distributed system apt at serving very large data files; internally used by Google for storing documents collected from the Web.

Open Source versions have been developed at once: Hadoop File System (HDFS), and Kosmos File System (KFS).

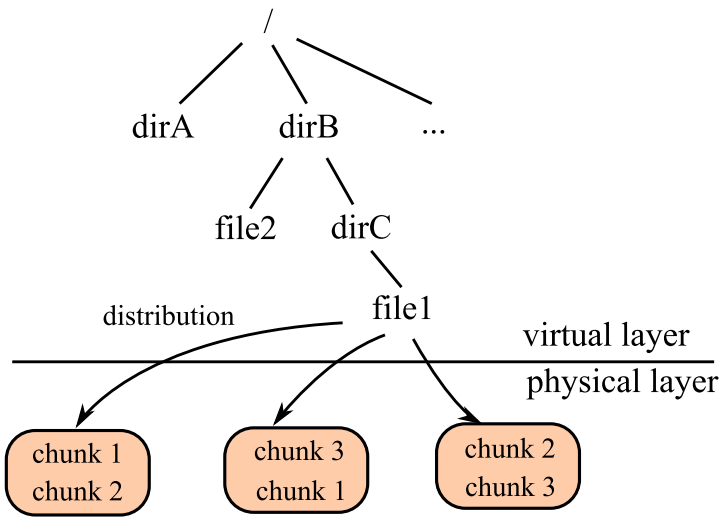
The problem

Why do we need a distributed file system in the first place?

Fact: standard NFS (left part) does not meet scalability requirements (what if *file1* gets really big?).



A traditional network file system

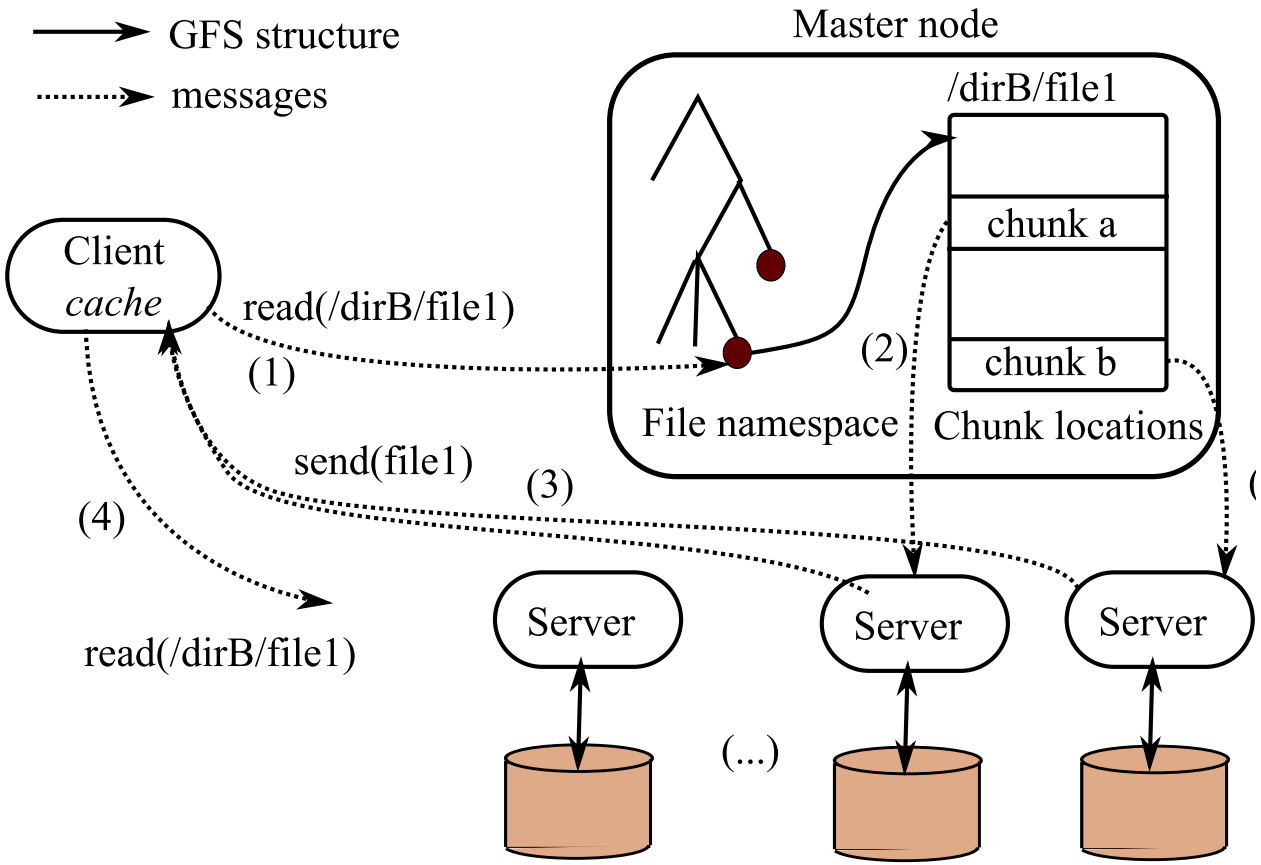


A large scale distributed file system

Right part: GFS/HDFS storage, based on (i) a virtual file namespace, and (ii) partitioning of files in “chunks”.

Architecture

A Master node performs administrative tasks, while servers store “chunks” and send them to Client nodes.



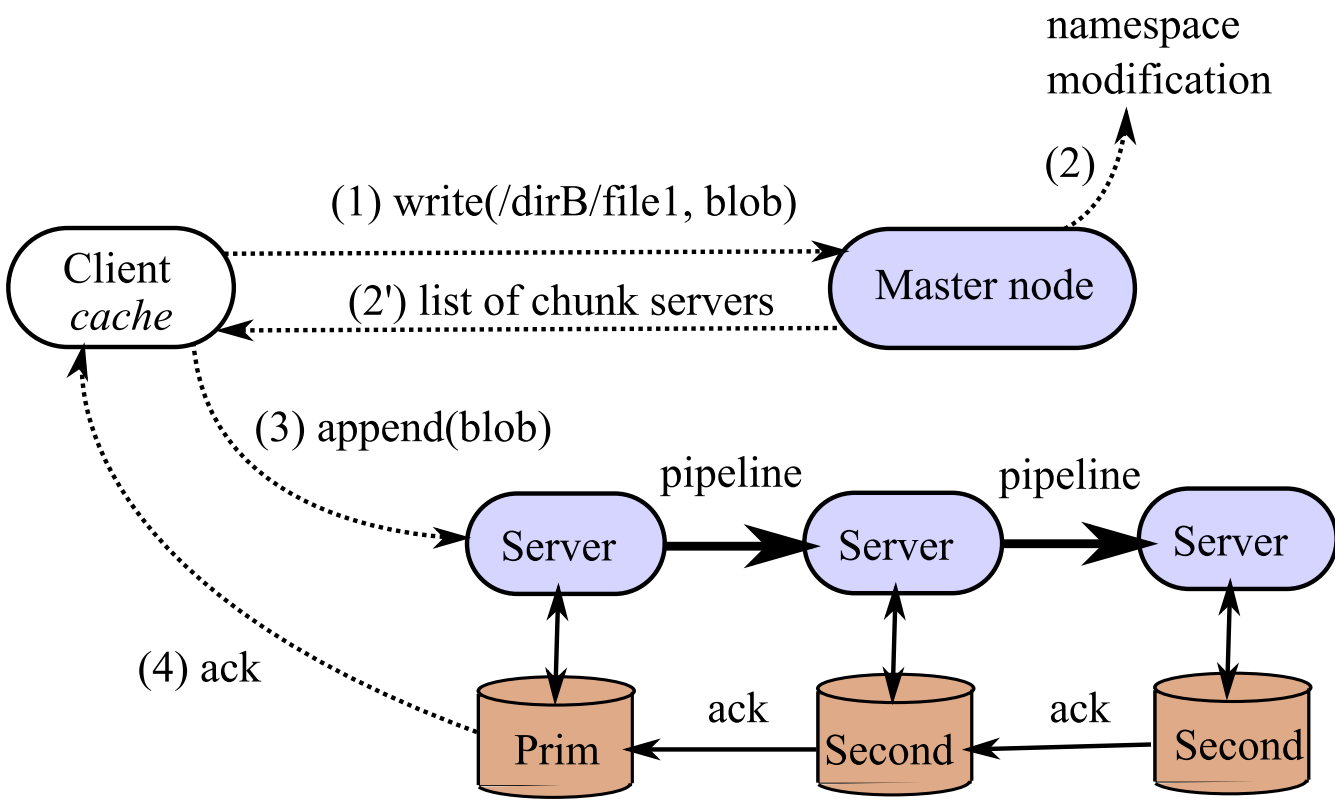
The Client maintains a cache with chunks locations, and directly communicates with servers.

Technical details

- The architecture works best for very large files (e.g., several Gigabytes), divided in large (64-128 MBs) chunks.
  $\Rightarrow$ this limits the metadata information served by the Master.
- Each server implements recovery and replication techniques (default: 3 replicas).
- (Availability) The Master sends heartbeat messages to servers, and initiates a replacement when a failure occurs.
- (Scalability) The Master is a potential single point of failure; its protection relies on distributed recovery techniques for all changes that affect the file namespace.

Workflow of a *write()* operation (simplified)

The following figure shows a non-concurrent *append()* operation.

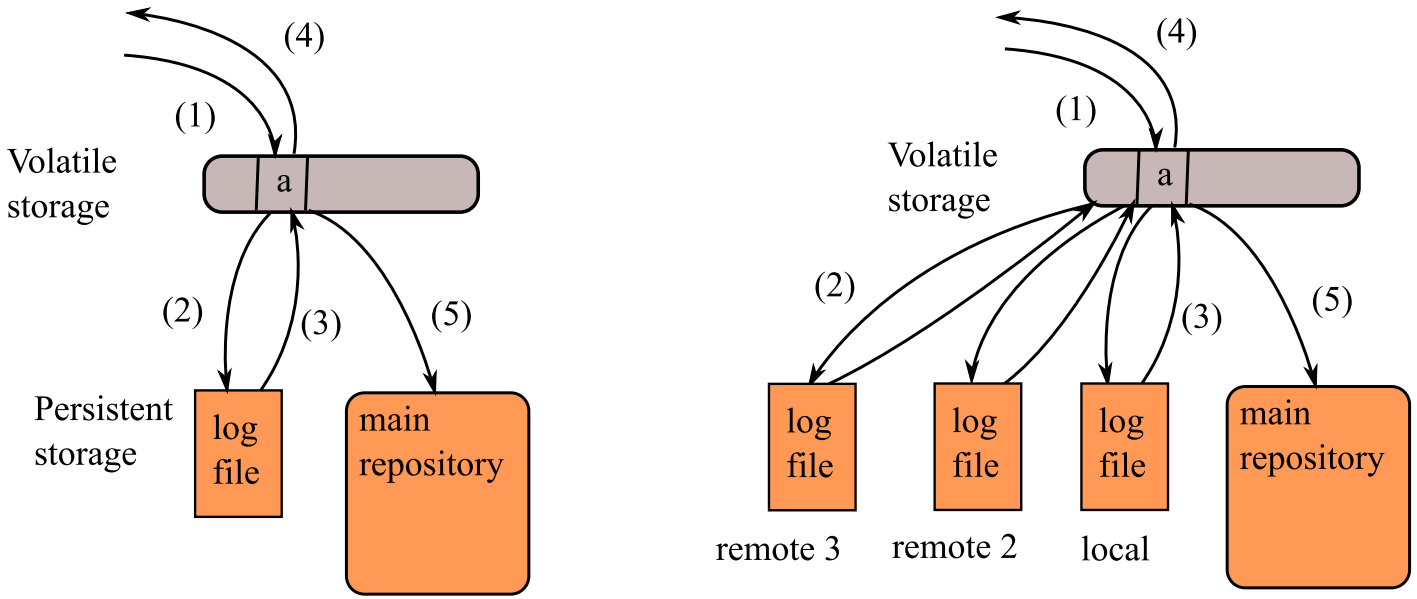


Write (append) in GFS (simplified to non-concurrent operations)

In case of concurrent appends to a chunk, the primary replica assigns serial numbers to the mutation, and coordinates the secondary replicas.

Namespace updates: distributed recovery protocol

Extension of standard techniques for recovery (left: centralized; right: distributed).



If a node fails, the replicated log file can be used to recover the last transactions on one of its mirrors.