Web search

Introduction to Distributed Computing

13 July 2011

Pierre Senellart

TELECOM ParisTech

Outline

Distributed systems

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

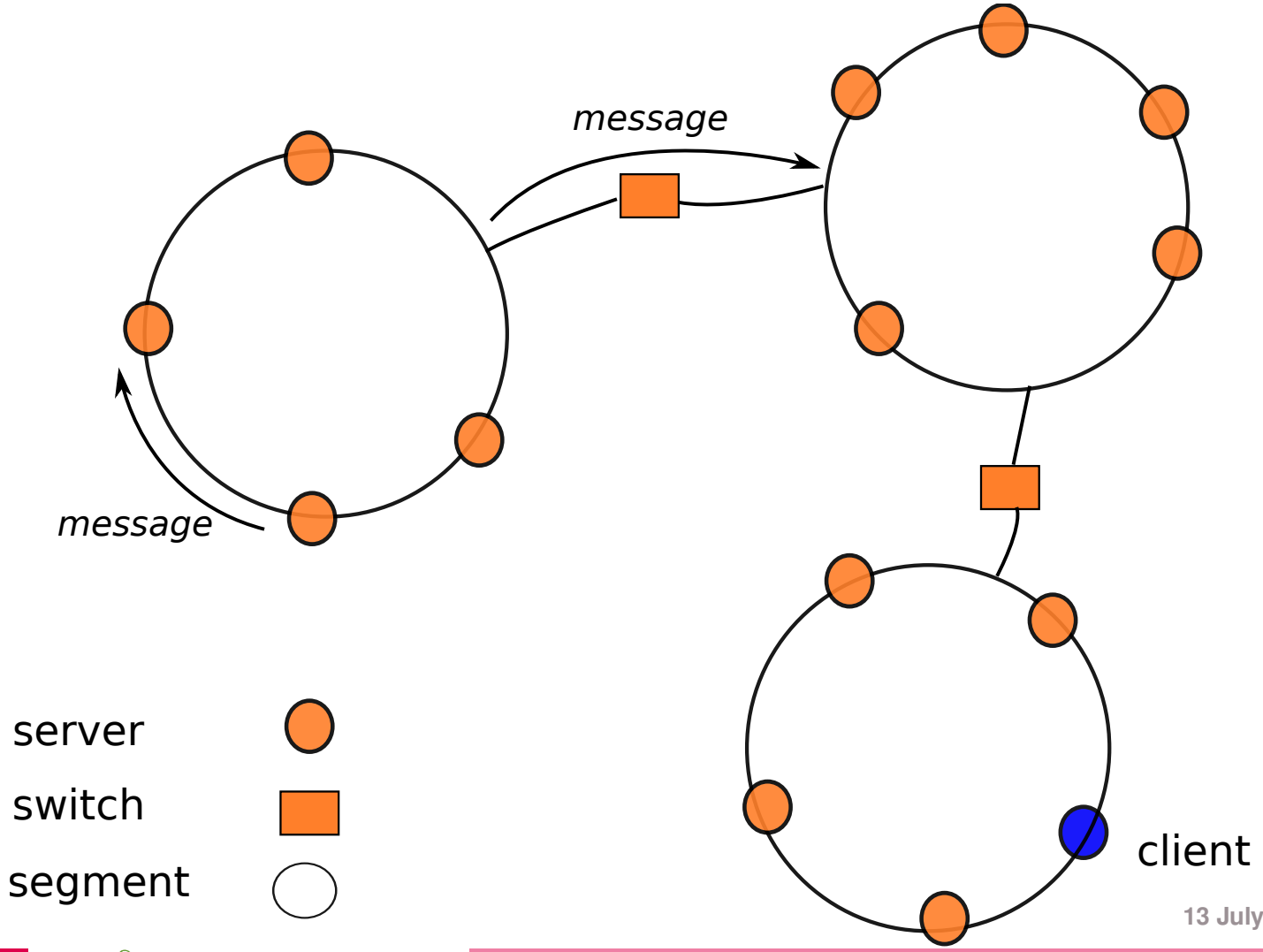
This coordination is achieved by exchanging messages which are pieces of data that convey some information.
⇒ "shared-nothing" architecture: no shared memory, no shared disk.

The system relies on a network that connects the computers and handles the routing of messages.
⇒ 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

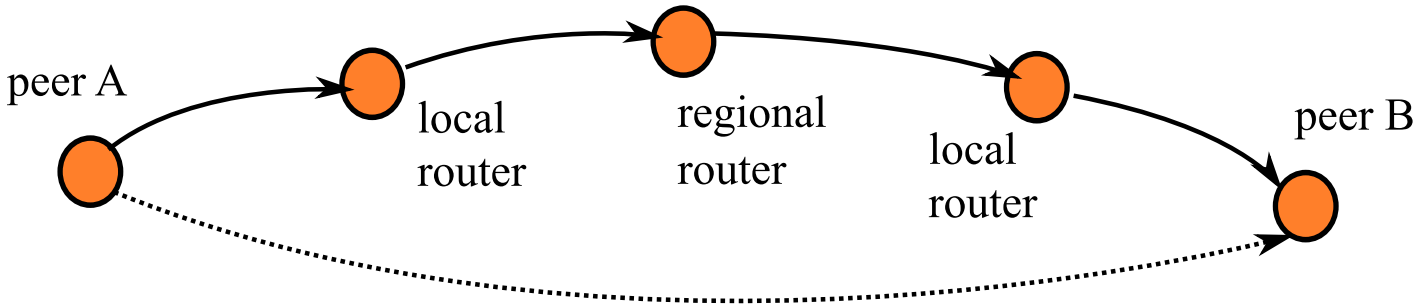
Typical setting of a Google data center.

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

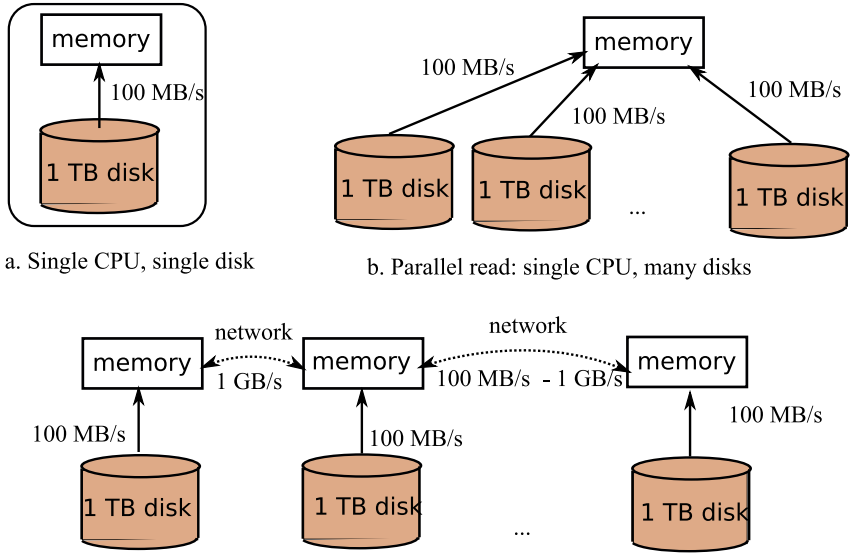
Bottom line (1): it is approx. one order of magnitude faster to exchange main memory data between 2 machines in a data center, that to read on the disk.
Bottom line (2): exchanging through the Internet is slow and unreliable with respect to LANs.

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.



a. Single CPU, single disk

b. Parallel read: single CPU, many disks

c. Distributed reads: an extendible set of servers

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.

Outline

History and development of GFS

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

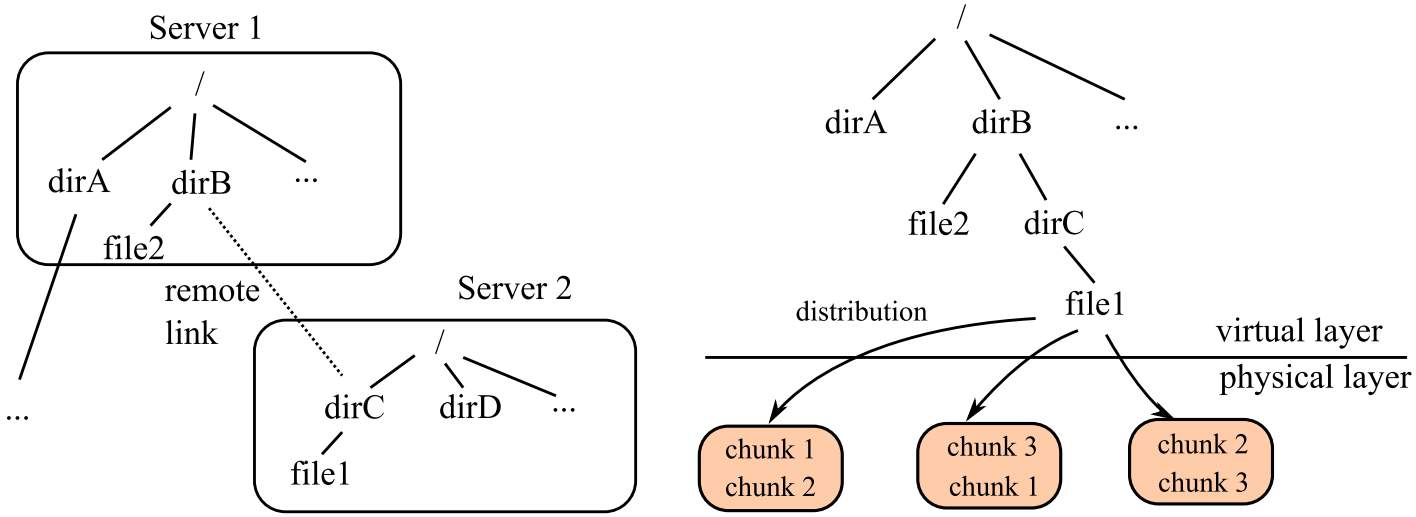
Explains the design and architecture of a distributed system apt at serving very large data files; internally used by Google for storing documents collected from the Web.

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?).



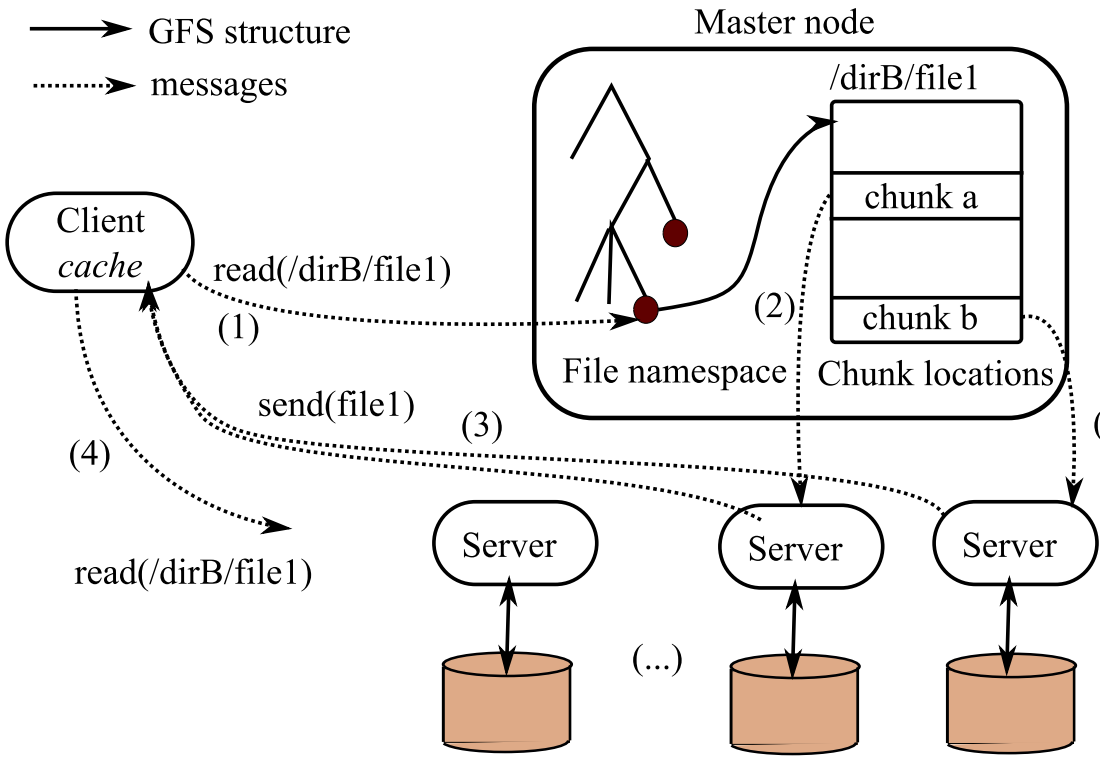
A traditional network file system

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.

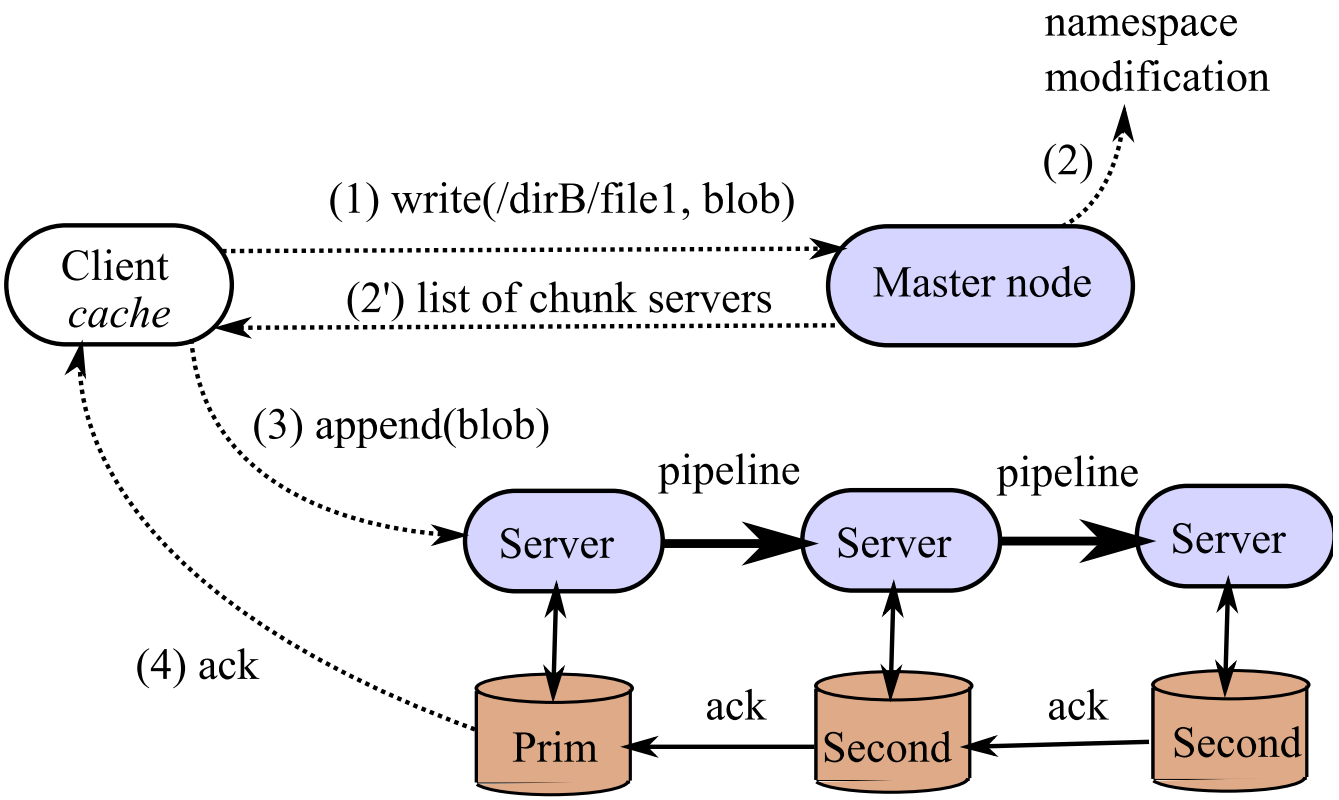


The Client maintains a cache with locations of chunks, and directly communicates with servers.

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)

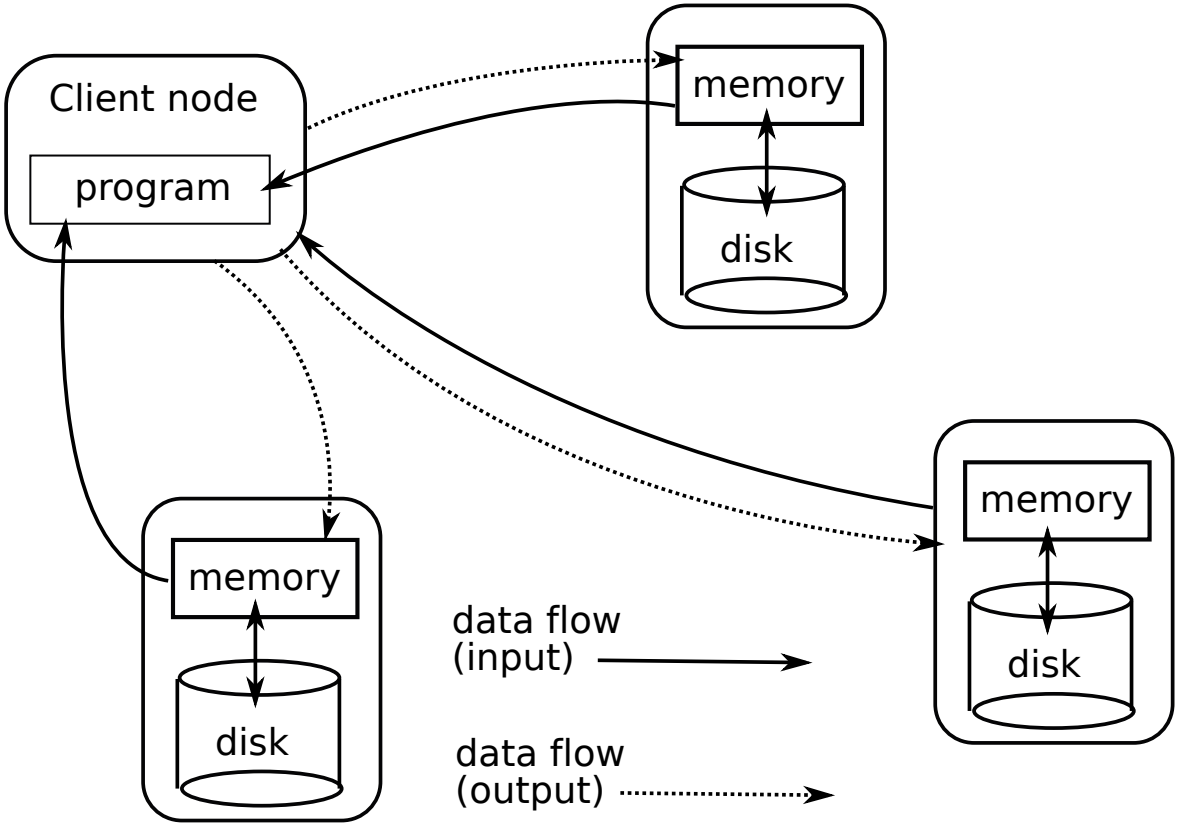


Write (append) in GFS (simplified to non-concurrent operations)

Outline

Centralized computing with distributed data storage

Run the program at the Client, get data from the distributed system.

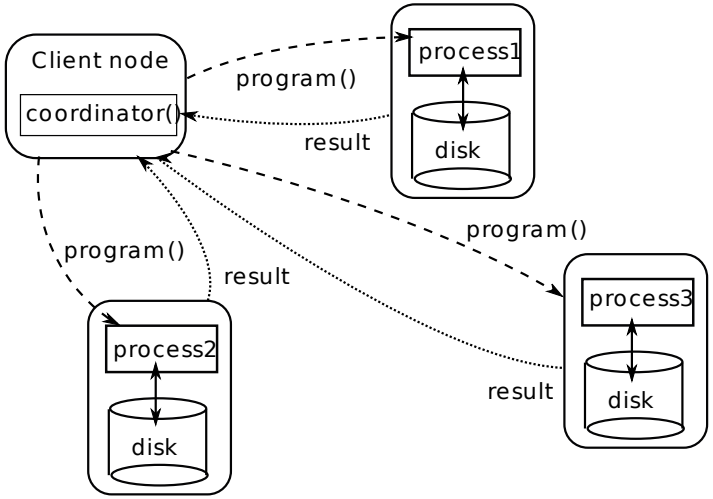


Downsides: important data flows, no use of the cluster computing resources.

History and development of MapReduce

Published by Google Labs in 2004 at OSDI. Implemented in Hadoop, widely used (Yahoo!, Amazon EC2).

A programming model (inspired by standard functional programming operators) to facilitate the development and execution of distributed tasks.



Main idea: “push the program near the data”. The programmer defines two functions; MapReduce takes in charge distribution aspects.

The programming model of MapReduce

Used to process data flows of $(key, value)$ pairs.

1. *map()* takes as input a list of pairs $(k, v) \in K_1 \times V_1$ and produces (for each pair), another *list* of pairs $(k', v') \in K_2 \times V_2$, called *intermediate pairs*.

   Example: take a pair $(uri, document)$, produce list of pairs $(term, count)$

2. (shuffle) the MapReduce execution environment groups intermediate pairs on the key, and produces grouped instances of type $(K_2, list(V_2))$

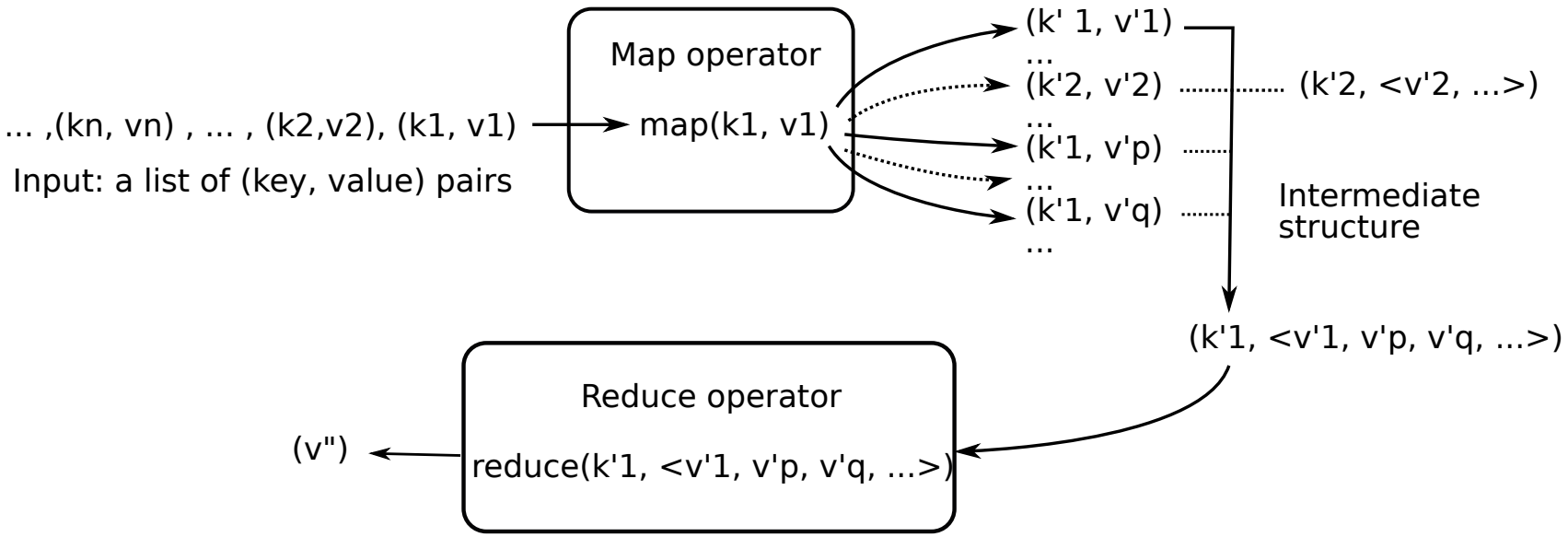
   Example: intermediate pairs for term 'job' are grouped as $('job', < 1, 4, 2, 8 >)$

3. *reduce()* operates on grouped instances of intermediate pairs $(k'_1, < v'_1, \cdots, v'_p, \cdots, v'_q, \cdots >)$; Each instance processed by the procedure outputs a result, usually a single value v''.

   Example: take a grouped instance $('job', < 1, 4, 2, 8 >)$ and output the sum: 15

Job workflow in MapReduce

Important: each pair, at each phase, is processed independently from the other pairs.



Network and distribution are transparently managed by the MapReduce environment.

Example: Counting terms occurrences in documents

The *map()* function:

```
mapCW(String key, String value):
  // key: document name
  // value: document contents
  for each term t in value:
    return (t, 1);
```

Note: we do not even need to count the occurrences of *t* in *value*.

Counting terms occurrences in documents (cont')

The *reduce()* function. It takes as input a grouped instance on a *key*. The nested list can be scanned with an iterator.

```
reduceCW(String key, Iterator values):
  // key: a term
  // values: a list of counts
  int result = 0;

  // Loop on the values list; cumulate in result
  for each v in values:
    result += v;

  // Send the result
  return result;
```

And, finally, the Job Driver program which submits both functions.

```
// A specification object for MapReduce execution
MapReduceSpecification spec;

// Define input files
  MapReduceInput* input = spec.add_input();
  input->set_filepattern("/movies/*.xml");
  input->set_mapper_class("MapWC");

// Specify the output files:
MapReduceOutput* out = spec.output();
out->set_filebase("/gfs/freq");
out->set_num_tasks(100);
out->set_reducer_class("ReduceWC");

// Now run it
MapReduceResult result;
if (!MapReduce(spec, &result)) abort();
  // Done: 'result' structure contains result info
   return 0;
```

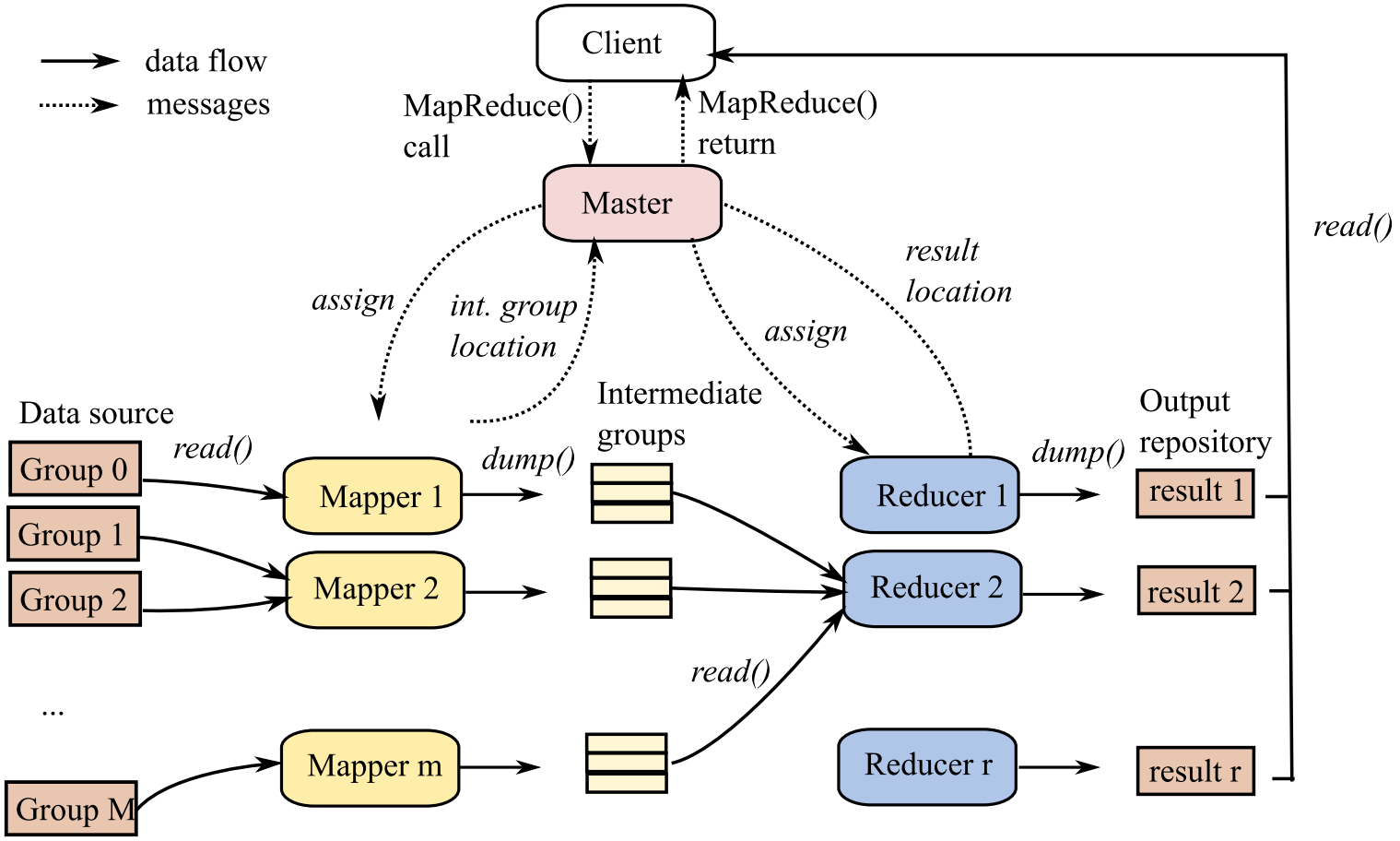
Processing *map()* and *reduce()* as a MapReduce job.

A MapReduce *job* takes care of the distribution, synchronization and failure handling. Specifically:

- the input is split in *M* groups; each group is assigned to a *mapper* (assignment is based on the data locality principle);
- each mapper processes a group and stores the intermediate pairs locally;
- grouped instances are assigned to *reducers* thanks to a hash function.
- (Shuffle) intermediate pairs are sorted on their key by the reducer;
- one obtains grouped instances, submitted to the *reduce()* function;

NB: the data locality does no longer hold for the Reduce phase, since it reads from the mappers.

Distributed execution of a MapReduce job.



Processing the *WordCount()* example (1)

Let the input consists of documents, say, one million 100-terms documents of approximately 1 KB each.

The split operation distributes these documents in groups of 64 MBs: each group consist of 64,000 documents. Therefore $M : \lceil 1,000,000/64,000 \rceil \approx 16,000$ groups.

We assume a global count of 1,000 distinct terms in the chunk; the (local) Map phase produces 6,400,000 pairs (t, c).

Let $hash(t) = t \mod 1,000$. Each intermediate group $i, 0 \leq i < 1000$ contains 6,400 pairs, each with 6-7 distinct terms t such that $hash(t) = i$.

Processing the *WordCount()* example (2)

Assume that *hash('call')* = *hash('mine')* = *hash('blog')* = i = 100. We focus on three Mappers M^p, M^q and M^r:

1. G_i^p =(<. . . , ('mine', 1), . . . , ('call',1), . . . , ('mine',1), . . . , ('blog', 1) . . . >
2. G_i^q =(< . . . , ('call',1), . . . , ('blog',1), . . . >
3. G_i^r =(<. . . , ('blog', 1), . . . , ('mine',1), . . . , ('blog',1), . . . >

R_i reads G_i^p, G_i^p and G_i^p from the three Mappers, sorts their unioned content, and groups the pairs with a common key:

. . . , ('blog', <1, 1, 1, 1>), . . . , ('call', <1, 1>), . . . , ('mine', <1, 1, 1>)

Our $reduce_{WC}()$ function is then applied by R_i to each element of this list. The output is *('blog', 4), ('call', 2) and ('mine', 3)*.

Failure management

In a distributed setting, the specific job handled by a machine is only a minor part of the overall computing task.

Moreover, because the task is distributed on hundreds or thousands of machines, the chances that a problems occurs somewhere are much larger; starting the job from the beginning is not a valid option.

The Master periodically checks the availability and reachability of the "Workers"

1. if a reducer fails, its task may be reassigned;
2. if a mapper fails, its task must be started from scratch, even in the Reduce phase (it holds intermediate groups).
3. if the Master fails? Then the whole job should be re-initiated.

Outline

Overview of existing systems

- Google File System (GFS). Distr. storage for very large, unstructured files. Open-source: HDFS (Hadoop)
- Dynamo. Internal data store of Amazon. Open source: Voldemort project.
- Bigtable, sorted distributed storage. Open source: HTable (Hadoop).
- Amazon proposes a Cloud environment, EC2, with: S3 and Hadoop/MapReduce.
- and Yahoo! with PNuts, Microsoft, with Scope, etc.

Conclusion

What you should remember

- Distributed systems: storing very large collections, exploit high network bandwidth vs. disk bandwidth
- MapReduce is a simple model for batch processing of very large collections.
  $\Rightarrow$ good for data analytics, index construction, etc.; not good for point queries (high latency).
- MapReduce also brings robustness against failure of a component and transparent distribution.

To go further

- More information in the *Web Data Management* book
- A Hadoop programming textbook, *Hadoop: The Definitive Guide*
- Hadoop freely downloadable from `http://hadoop.apache.org/`

Licence de droits d'usage

Contexte public } avec modifications

Par le téléchargement ou la consultation de ce document, l'utilisateur accepte la licence d'utilisation qui y est attachée, telle que détaillée dans les dispositions suivantes, et s'engage à la respecter intégralement.

La licence confère à l'utilisateur un droit d'usage sur le document consulté ou téléchargé, totalement ou en partie, dans les conditions définies ci-après et à l'exclusion expresse de toute utilisation commerciale.
Le droit d'usage défini par la licence autorise un usage à destination de tout public qui comprend :
– le droit de reproduire tout ou partie du document sur support informatique ou papier,
– le droit de diffuser tout ou partie du document au public sur support papier ou informatique, y compris par la mise à la disposition du public sur un réseau numérique,
– le droit de modifier la forme ou la présentation du document,
– le droit d'intégrer tout ou partie du document dans un document composite et de le diffuser dans ce nouveau document, à condition que :
  – L'auteur soit informé.

Les mentions relatives à la source du document et/ou à son auteur doivent être conservées dans leur intégralité.

Le droit d'usage défini par la licence est personnel et non exclusif.
Tout autre usage que ceux prévus par la licence est soumis à autorisation préalable et expresse de l'auteur : `sitepedago@telecom-paristech.fr`