14 An Introduction to Distributed Systems

This chapter is an introduction to very large data management in distributed systems. Here, "very large" means a context where Gigabytes (1,000 MB = 10^9 bytes) constitute the unit size for measuring data volumes. Terabytes (10^{12} bytes) are commonly encountered, and many Web companies, scientific or financial institutions must deal with Petabytes (10^{15} bytes). In a near future, we can expect Exabytes (10^{18} bytes) data sets, with the world-wide digital universe roughly estimated (in 2010) as about 1 Zetabytes (10^{21} bytes).

Distribution is the key for handling very large data sets. Distribution is necessary (but not sufficient) to bring *scalability*, i.e., the means of maintaining stable performance for steadily growing data collections by adding new resources to the system. However, distribution brings a number of technical problems that make the design and implementation of distributed storage, indexing and computing a delicate issue. A prominent concern is the risk of *failure*. In an environment that consists of hundreds or thousands of computers (a common setting for large Web companies), it becomes very common to face the failure of components (hardware, network, local systems, disks), and the system must be ready to cope with it at any moment.

Our presentation covers principles and techniques that recently emerged to handle Web-scale data sets. We examine the extension of traditional storage and indexing methods to large-scale distributed settings. We describe techniques to efficiently process *point queries* that aim at retrieving a particular object. Here there typically is a human being waiting for an answer in front of a screen. So, efficient means a response time in the order of a few milliseconds, a difficult challenge in the presence of Terabytes of data. We also consider the *batch analysis* of large collections of documents to extract statistical or descriptive information. The problem is very different. Possibly Terabytes of data are streamed into a program. Efficient computation now means hours or even days and a most critical issue is the reliable execution of processes that may run so long, in spite of the many glitches that are likely to affect the infrastructure in such a time frame. We should keep in mind these specificities in the presentation that follows, as it motivates many design choices.

The present chapter introduces the essentials of distributed systems devoted to large scale data sets. Its material represents by no means an in-depth or accurate coverage of the topic, but merely aims at supplying the neophyte reader with the minimal background. As usual, the Further Reading section points to complementary references.

14.1 Basics of distributed systems

A *distributed system* is piece of software that serves to coordinate the actions of several computers. This coordination is achieved by exchanging *messages*, i.e., pieces of data conveying information. The system relies on a network that connects the computers and handles the routing of messages.

14.1.1 Networking infrastructures

We limit the discussion in this chapter to the following two classes of networks: Local Area Networks and P2P Networks.

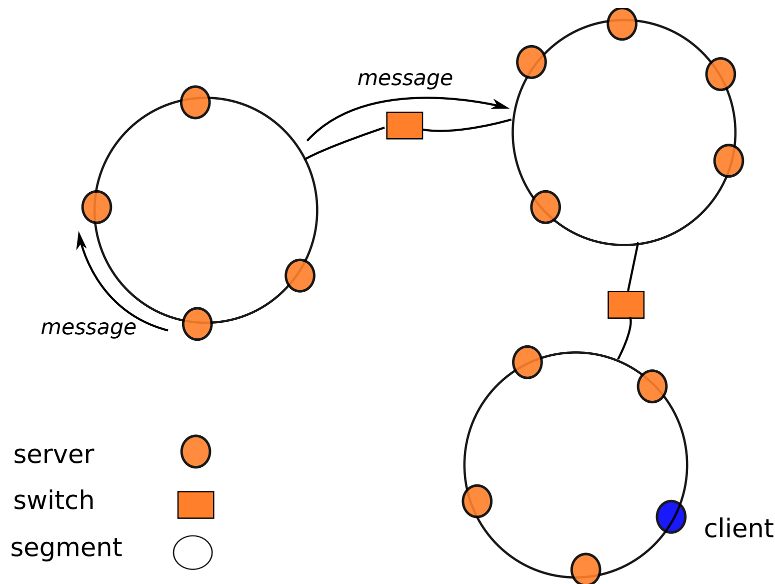

Figure 14.1: A simplified view of a local network

Local Area Network (LAN). LANs are for instance used in data centers to connect hundreds or even thousands of servers. Figure 14.1 shows the main features of a typical Local Area Network (LAN) in this context. We roughly distinguish three communication levels:

- First, servers are grouped on "racks", linked by a high-speed cable. A typical rack contains a few dozens of servers.
- Second, a data center consists of one to a large number of racks connected by *routers* (or *switches*) that transfer non-local messages.
- A third (slower) communication level, between distinct clusters, may also be considered. It may for instance allow some independent data centers to cooperate, e.g., to consolidate global statistics.

In all cases, servers only communicate via message passing. They do not share storage or computing resources. The architecture is said "shared-nothing".

Example 14.1.1 *At the beginning of 2010, a typical Google data center consists of 100-200 racks, each hosting about 40 servers. The number of servers in such a center is roughly estimated around 5,000. The number of data centers is constantly evolving, and the total number of servers is probably already above one million.*

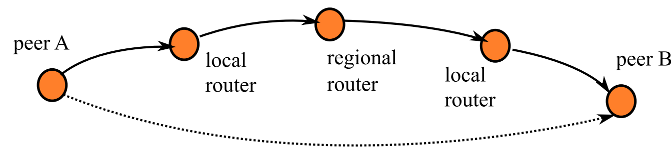

Figure 14.2: Internet networking

Peer-to-Peer Network (P2P). A P2P network is a particular kind of *overlay network*, a graph structure build over a native physical network. The physical network we consider here is the Internet. Nodes, or "peers" communicate with messages sent over the Internet. The route that connects two peers on the Internet is typically intricate. Typically (Figure 14.2), a message sent by peer A first reaches a local router, that forwards the message to other routers (local, regional, or world-wide) until it is delivered to peer B. By abstracting this complexity, a P2P network imagines a direct link between A and B, as if they were directly connected, as soon as they know the IP addresses of each other. This pseudo-direct connection that may (physically) consist of 10 or more forwarding messages, or "hops", is called an *overlay link*, therefore the term *overlay network*.

Example 14.1.2 *If you are connected to the Internet, you can use the* `traceroute` *utility program to inspect the routers involved in the connection between your computer and a site of your choice. For instance:* `traceroute Webdam.inria.fr` *gives the list of routers on the forwarding Internet path to the Webdam INRIA Web site. Several sites propose a traceroute interface if you do not have access to a console. One can find some, e.g., at* `traceroute.org`*.*

For our purposes, we will assimilate nodes to computers running programs of interest to the distributed system. A computer often runs several programs involved in different kinds of services. A process on computer A may for instance be in charge of file accesses, while another, running on A as well, handles HTTP requests. If we focus on a specific task of the distributed system, there is generally one and only one process that fulfills this task on each computer. This allows blurring the distinction, and we will simply denote as *node* a process running on a computer at a specific location of the network, and in charge of the particular task.

Next, it is often convenient to distinguish *server nodes* from *client nodes*. A server node provides, through cooperation with other server nodes, a service of the distributed system. A client node consumes this service. Nothing prevents a client node to run on the same computer than a server node (this is typically the case in P2P networks), but the point is most often irrelevant to the discussion. In practice, a client node is often a library incorporated in a larger application, that implements the communication protocol with the server nodes. When no ambiguity arises, we will simple use "Client" and "Server" to denote respectively a client node and a server node it communicates with.

14.1.2 Performance of a distributed storage system

Nodes exchange *messages* following a particular protocol. The Ethernet protocol is the most widely used. It splits messages into small packets of, typically, 1,500 bytes each. At the time of writing, the data transfer rate of a local Ethernet network can (theoretically) reach 1

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

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 MBs.;

Table 14.1: Disk vs. network latency and bandwidth

Gigabytes/s. The bandwidth is higher that the maximal disk rate which is at most 100 MB/s. Roughly speaking, it is one order of magnitude faster to exchange in-memory data between two computers connected by a high-speed LAN, than for a single computer to read the same data written on the disk. However, bandwidth is a resource that many participants compete for, and this invites to use it with care in data intensive applications. The latency (time to initiate an operation) is also cheaper with networks, although the gap is less impressive.

Internet figures for latency and bandwidth are highly varying, as they depend both on the distance between the communicating nodes, and on the network devices involved, particularly at local ends. (For instance, a Wifi connection in an Internet cafe is a nightmare for data intensive manipulations!) As an illustration, the latency of a connection between INRIA Paris and Stanford University is less than 200 ms., and the bandwidth is 7 MB/s (download) and 3 MB/s (upload). You are encouraged to test these values on your own infrastructure, with the `ping` command or some of the numerous Web sites. For instance, see

http://www.pcpitstop.com/internet/Bandwidth.asp.

The latency of the average Internet path is estimated at 10 ms. The performance of Internet is definitely at least one order of magnitude worse than LANs. Table 14.1 summarizes the values that should be kept in mind.

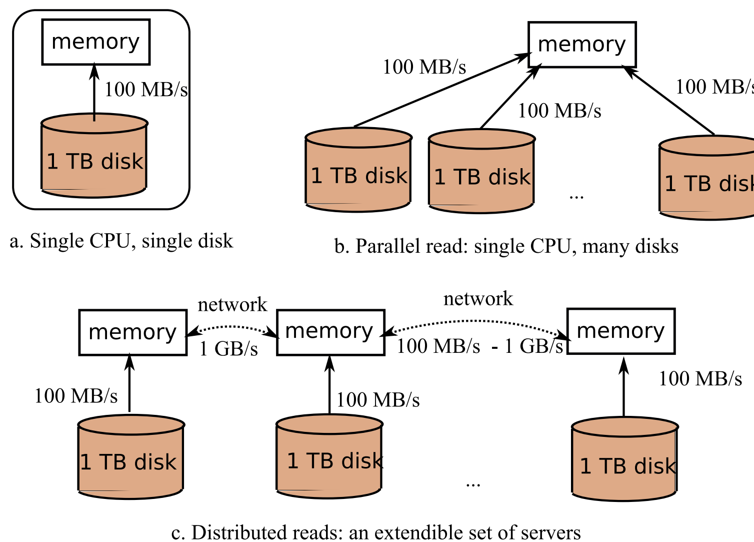

Figure 14.3: Distributed data management: why?

The following reasoning helps understand the advantage of distributed storage (see Figure 14.3 for an illustration):

Sequential access. Consider a typical 1 Terabytes disk with 100 MB/s maximal transfer rate. It takes 166 mns (more than 2 hours and a half!) to read the whole content of the disk.

Parallel access. Now imagine the 1 TB data set spread over 100 disks on a same machine. In order to read this data set (i.e., to bring it in the computer's main memory), we must retrieve 10 GBs from each disk. This is done, assuming that the disks work in parallel, in a little more that 1mn 30s. But, when the size of the data set increases, the CPU of the computer is typically overwhelmed at some point by the data flow and it is slowed down.

Distributed access. The same disk-memory transfer time can be achieved with 100 computers, each disposing of its own local disk. The advantage now is that the CPU will not be overwhelmed as the number of disks increases.

This is a good basis to discuss some important aspects of data distribution. Note first that we assume that the maximal transfer rate is achieved for each disk. This is only true for *sequential* reads, and can only be obtained for operations that fully scan a data set. As a result, the seek time (time to position the head on appropriate disk track) is negligible regarding the transfer time. Therefore the previous analysis mostly holds for *batch* operations that access the whole collection, and is particularly relevant for applications where most files are written once (by *appending* new content), then read many times. This scenario differs from the classical behavior of a centralized database.

Now consider in contrast a workload consisting of lots of operations, each one randomly accessing a small piece of data in a large collection. (Such an operation is more in the spirit of a database operation where a row in a large table is accessed.) The access may be a read or a write operation. In both cases, we have to perform a random access to a large file and seek time cannot be ignored. Distribution is here of little help to speed up a single operation. However, if we can afford to replicate the data on many servers, this is an opportunity to balance the query load by distributing evenly read and/or write requests. Architectures for such transactional scenarios can actually be classified by their read/write distribution policy: distributing writes raises concurrency issues; distributing reads raises consistency issues. We further develop this important point in the following.

Finally, look again at Figure 14.3. The distribution mechanism shows two possible data flows. The first one comes from the disk to the *local* CPU, the second one (with dotted arrows) represents exchanges between computers. The performance of network exchanges depends both on the latency and on the network bandwidth. As said above, the typical transfer rate is 100 MB/s and can reach 1 GB/s, one order of magnitude higher than disks, but bandwidth is a shared resource that must be exploited with care.

A general principle, known as the *data locality principle*, states that a data set stored on a disk should be processed by a task of the local CPU. The data locality principle is valid for data intensive applications. The architecture adopted in such cases is different from that of High Performance Computing or Grid Computing that distribute a task across a set of CPU that share a common file system. This works as long as the task is CPU intensive, but becomes unsuited if large data exchanges are involved.

To summarize:

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

1. disk transfer rate is a bottleneck for batch processing of large scale data sets; parallelization and distribution of the data on many machines is a means to eliminate this bottleneck;
2. disk seek time is a bottleneck for transactional applications that submit a high rate of random accesses; replication, distribution of writes and distribution of reads are the technical means to make such applications scalable;
3. data locality: when possible, program should be "pushed" near the data they need to access to avoid costly data exchange over the network.

14.1.3 Data replication and consistency

Most of the properties required from a distributed system depend on the replication of data. Without replication, the loss of a server hosting a unique copy of some data item results in unrecoverable damages. As already said, replication also brings other advantages, including the ability to distribute read/write operations for improved scalability. However, it raises the following intricate issues:

Performance. Writing several copies of an item takes more time, which may affect the throughput of the system.

Consistency. Consistency is the ability of a system to behave as if the transaction of each user always run in isolation from other transactions, and never fails. Consider for instance a transaction on an e-commerce site. There is a "basket" which is progressively filled with bought items. At the end the user is directed to a secure payment interface. Such a transaction involves many HTTP accesses, and may last an extended period of time (typically, a few minutes). Consistency in this context means that if the user added an item to her basket at some point, it should remain there until the end of the transaction. Furthermore, the item should still be available when time comes to pay and deliver the product.

Data replication complicates the management of consistency in a distributed setting. We illustrate next four typical replication protocols that show the interactions between performance considerations and consistency issues (Figure 14.4). The scenario assumes two concurrent Client applications, A and B, that put/read a data idem d which is replicated on two servers S_1 and S_2. The four cases depicted in Figure 14.4 correspond to the possible combinations of two technical choices: eager (synchronous) or lazy (asynchronous) replication, and primary or distributed versioning:

Eager, primary. Consider the first case (a). Here, the replication policy is "eager": a *put(d)* request sent by Client A to Server 1 is replicated at once on Server 2. The request is completed only when both S_1 and S_2 have sent an acknowledgment; meanwhile, A is frozen, as well as any other Client that would access d. Moreover, the second design choice in case (a) is that each data item has a primary copy and several (at least one) secondary copies. Each update is first sent to the primary copy.

From an application point of view, such a design offers some nice properties. Because the replication is managed synchronously, a read request sent by Client 2 always access

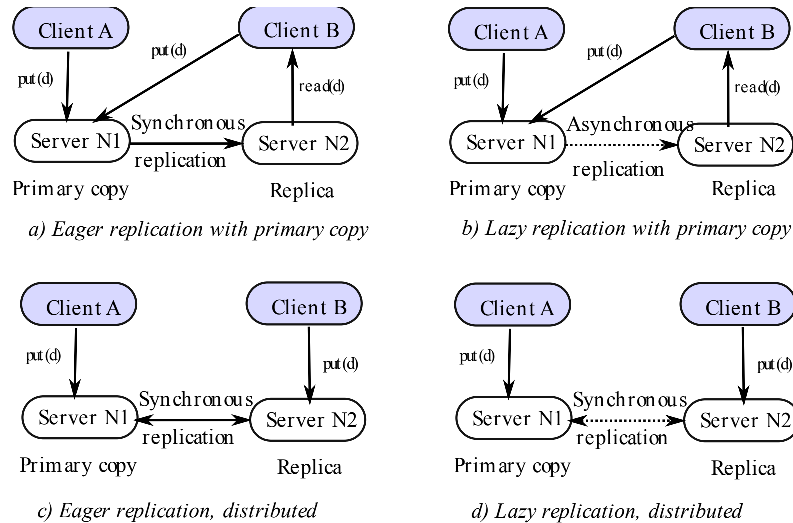

Figure 14.4: Four replication policies in a distributed system

a consistent state of d, whether it reads from S_1 or S_2. And because there is a primary copy, requests sent by several clients relating to a same item can be queued, which ensures that updates are applied sequentially and not in parallel. The obvious downside is that these applications have to wait for the completion of other clients' requests, both for writing and reading.

Async, primary. Case (b) (often referred to as "Master-slave" replication) shows a slightly different design choice. There is still a primary copy, but the replication is asynchronous. Thus, some of the replicas may be out of date with respect to Client's requests. Client B for instance may read on S_2 an old version of item d because the synchronization is not yet completed. Note that, because of the primary copy, we can still be sure that the replicas will be *eventually* consistent because there cannot be independent updates of distinct replicas. This situation is considered acceptable in many modern, "NoSQL" data management systems that accept to trade strong consistency for a higher read throughput.

Eager, no primary. Case (c), where there is no primary copy anymore (but eager replication), yields a complex situation where two Clients can simultaneously write on distinct replicas, whereas the eager replication implies that these replications must be synchronized right away. This is likely to lead to some kind of interlocking, where both Clients wait for some resource locked by another one.

Async, no-primary. The most flexible case is (d) (often referred to as "Master-Master" replication), in which both primary copies and synchronous replication are given up. There is an advantage (often viewed as decisive for Web-scale data intensive applications): Client operations are *never* stalled by concurrent operations, at the price of possibly inconsistent states

Inconsistencies sometimes entailed by asynchronous protocols never occur in centralized database systems whose transactional model guarantees ACID properties. This may however

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

be the preferred choice of distributed systems that favor efficiency and adopt a more permissive consistency model. A pending issue in that case is the management of inconsistent versions, a process often called *data reconciliation*. What happens when the system detects that two replicas have been independently modified? It turns out that the answer is, in most cases, quite practical: data reconciliation is seen as application-dependent. The system is brought back to a consistent state, possibly by promoting one of the versions as the "current" one, and notifying the Client applications that a conflict occurred. Readers familiar with Concurrent Versioning Systems like CVS or Subversion will recognize the optimistic, lock-free mechanism adopted by these tools.

It has been argued that a distributed system cannot simultaneously satisfy consistency and availability while being tolerant to failures ("the CAP theorem", discussed further). Therefore a system designer has to choose which property should be (at least partly) sacrificed. This often leads to giving up strong consistency requirements, in favor of availability and fault tolerance.

In summary, data replication leads to distinguishing several consistency levels, namely:

- *Strong consistency* (ACID properties), requires a (slow) synchronous replication, and possibly heavy locking mechanisms. This is the traditional choice of database systems.
- *Eventual consistency* trades eager replication for performance. The system is guaranteed to converge toward a consistent state (possible relying on a primary copy).
- *Weak consistency* chooses to fully favor efficiency, and never wait for write and read operations. As a consequence, some requests may serve outdated data. Also, inconsistencies typically arise and the system relies on reconciliation based on the application logic.

Existing database systems are often seen as too heavy and rigid in distributed systems that give up strong consistency to achieve better performance. This idea that the strong consistency requirements imposed by RDBMS are incompatible with distributed data management, is one of the founding principles of the "NoSQL" trend.

14.2 Failure management

In a centralized system, if a program fails for any reason, the simple (and, actually, standard) solution is to abort then restart its transactions. On the other hand, chances to see a single machine fail are low. Things are quite different in the case of a distributed system with thousands of computers. Failure becomes a possibly frequent situation, due to program bugs, human errors, hardware or network problems, etc. For small tasks, it is just simpler to restart them. But for long lasting distributed tasks, restarting them is often not an acceptable option in such settings, since errors typically occur too often. Moreover, in most cases, a failure affects a minor part of the task, which can be quickly completed providing that the system knows how to cope with faulty components.

Some common principles are met in all distributed systems that try to make them resilient to failures. One of the most important is *independence*. The task handled by an individual node should be independent from the other components. This allows recovering the failure by only considering its initial state, without having to take into account complex relationships or synchronization with other tasks. Independence is best achieved in *shared-nothing*

architectures, when both the CPU and the local disk of a server run in isolation of the other components of the servers.

Thanks to replication methods examined earlier, a failure can usually be recovered by replacing the faulty node by a mirror. The critical question in this context is to detect that a system met a failure. Why for instance is a Client unable to communicate with a server? This may be because of a failure of the server itself, of because the communication network suffers from a transient problem. The Client can wait for the failed node to come back, but this runs against availability, since the application becomes idle for an unpredictable period of time.

14.2.1 Failure recovery

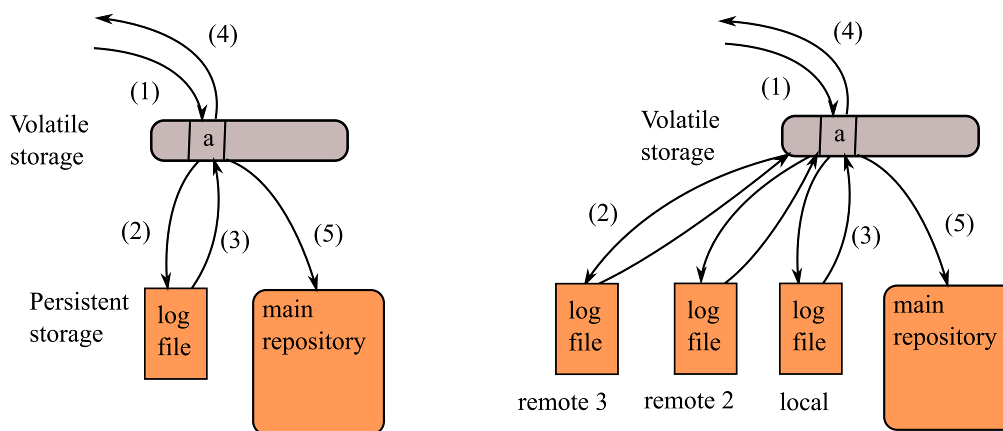

Figure 14.5: Recovery techniques for centralized (left) and replicated architectures (right)

Figure 14.5 recalls the main aspects of data recovery in a centralized data management system, and its extension to distributed settings.

Consider first a client-server application with a single server node (left part). (1) The Client issues a *write(a)*. The server does not write immediately *a* in its repository. Because this involves a random access, it would be very inefficient to do so. Instead, it puts *a* in its volatile memory. Now, if the system crashes or if the memory is corrupted in any way, the write is lost. Therefore, the server writes in a *log file* (2). A log is a sequential file which supports very fast append operations. When the log manager confirms that the data is indeed on persistent storage (3), the server can send back an acknowledgment to the Client (4). Eventually, the main memory data will be flushed in the repository (5).

This is standard recovery protocol, implemented in centralized DBMSs. In a distributed setting, the server must log a write operation not only to the local log file, but also to 1, 2 or more remote logs. The issue is close to replication methods, the main choice being to adopt either a *synchronous* or *asynchronous* protocol.

Synchronous protocol. The server acknowledges the Client only when all the remote nodes have sent a confirmation of the successful completion of their *write()* operation. In practice, the Client waits until the slower of all the writers sends its acknowledgment. This may severely hinder the efficiency of updates, but the obvious advantage is that all the replicas are consistent.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

Asynchronous protocol. The Client application waits only until one of the copies (the fastest) has been effectively written. Clearly, this puts a risk on data consistency, as a subsequent read operation may access an older version that does not yet reflect the update.

The multi-log recovery process, synchronous or asynchronous, has a cost, but it brings availability (and reliability). If the server dies, its volatile memory vanishes and its local log cannot be used for a while. However, the closest mirror can be chosen. It reads from it own log a state equivalent to that of the dead server, and can begin to answer Client's requests. This is standard REDO protocol, described in detail in any classical textbook on centralized database. We do not elaborate further here.

14.2.2 Distributed transactions

A *transaction* is a sequence of data update operations, that is required to be an "all-or-nothing" unit of work. That is, when a *commit* is requested, the system has to perform *all* the updates in the transaction. We say the transaction has been *validated*. In case of problem, the system has also the option to perform *nothing* of it. We say the transaction has been *aborted*. On the other hand, the system is not allowed to perform some of the updates and not others, i.e., partial validation is forbidden.

In a distributed setting, the update operations may occur on distinct servers $\{S_1, \ldots, S_n\}$, called *participants*. A typical case is the eager replication explained earlier. The problem is to find a protocol, implemented and controlled by a distinct node called *coordinator*, that communicates with the participants so that the all-or-nothing semantics is ensured. The main algorithm that is used to achieve this goal is the *two-phase commit* (2PC) protocol:

1. first, the coordinator asks each participant whether it is able to perform the required operation with a *Prepare* message;

2. second, if all participants answered with a confirmation, the coordinator sends a *Decision* message: the transaction is then committed at each site.

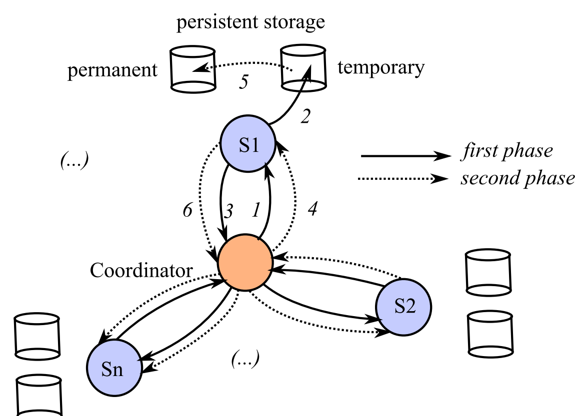

Figure 14.6: The two-phase commit protocol (details are given for the Coordinator-Server 1 communications only)

Assume for the time being that everything proceeds gracefully, without node failure or network communication problem. In this ideal scenario, a distributed data update transaction conforms to the following workflow (Figure 14.6, with focus on S_1). Initially (1) the Coordinator sends a Prepare message. Each participant then takes appropriate measures to guarantee that it will be able to fulfill its task in the second phase. Typically, updated data stored in volatile memory is written on a safe temporary persistent storage to prevent a loss due to a system crash (2). The participant can then send a confirmation message to the Coordinator (3), either `confirm` if the participant is ready to commit, or `refuse`.

The second phase begins when the Coordinator got all the answers from the participating nodes. It sends then a *Decision* message (4) which can either be `commit` or `abort`. The rule is that if *at least one* participant refused its part of the transaction, the whole operation must be aborted. If all confirm their readiness, the Coordinator can send a `commit`. (Although it is not compelled to do so: a `refuse` is also acceptable).

In case of `commit`, each participant copies the data from the temporary area to the main repository (5), else it can simply remove the temporary storage associated to the ongoing transaction. An acknowledgment of success is required for this second round, so that the Coordinator closes the transaction.

Now, the question is: what if a failure occurs somewhere? We can distinguish between network communication problems and node failures. In addition, we have to examine separately the roles of the Coordinator from that of the participants. We start with the latter and examine the appropriate recovery action, depending on the instant of the failure occurrence.

Initial failure Such a failure occurs when the Participant p_i is unable to receive the *prepare* message; in that case it cannot answer, and the Coordinator aborts the transaction.

Failure in *prepared* state. p_i received the *prepare* message and took the appropriate measures to ensure that it is indeed ready to commit if required to. Note that, at this point, p_i probably allocates resources to the transaction and holds some locks that possibly hinder the overall system throughput. The protocol must ensure that it will eventually (and as soon as possible) receive a decision from the Coordinator, even if it fails and restart.

Failure in `commit` or `abort` state. p_i learned the decision of the Coordinator, and is compelled to carry out the operations that do implement this decision, even if it undergoes one or several failures.

Technically speaking, such a distributed protocol must preserve some vital information regardless of the failures that affect a node. For instance, a Participant that fails in the *prepared* state must be able to figure out, when it recovers, whether the Coordinator sent its decision. For instance, it could contact the Coordinator to learn the current state. In `commit` state, a failure may occur while p_i is proceeding with the validation of the transaction. After restart, the validation needs to be re-executed, which implies that it is implemented as an *idempotent* operation (a property common to all recovery mechanisms).

We now turn our attention to the Coordinator. A first remark is that it must implement the necessary actions to preserve its ability to monitor the distributed transaction, even if it fails. For instance, before sending the *Decision* message, the `commit` or `abort` choice must be logged in a safe (persistent) area. Indeed, if the Coordinator fails after sending its decision, it would restart in an undefined status if this information could not be recovered.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

In general, if the Coordinator fails, the distributed process may be in an intermediate state which can only be solved if the Coordinator restarts and is able to resume properly the transaction. If, for instance, failure occurs when the Coordinator is sending *prepare* messages, some Participants may be informed of the transaction request, while others may not. On restart, the Coordinator should look for pending transactions in its log and re-send the message.

The same approach holds for dealing with Coordinator failure in the various steps of the protocol. The main problem is that the Coordinator may fail permanently, or suffer from network communication problems that leave the process pending for an unbounded time period. Meanwhile, the Participants are *blocked* and maintain their locks on the resources allocated to the transaction.

Note that a Participant *cannot* decide independently to commit or abort. (We could imagine for instance a timeout mechanism that triggers an `abort` if the Participant is left in a *prepared* state without receiving the decision.) Indeed, it may be the case that the Coordinator sent a `commit` decision that reached all the participants save one. Aborting this part of the transaction would break the all-or-nothing requirements. Several techniques have been proposed to overcome the blocking nature of the 2PL protocol, including communication among the Participants themselves. We invite the reader to consult the last section of the chapter for references.

The 2PL protocol is a good illustration of the difficulty to coordinate the execution of several related processes, in particular in case of failures. Applications that need to execute distributed transactions enter in a mechanism where nodes become *dependent* from one another, and this makes the whole data management much more intricate. Moreover, the mechanism tends to block the operations of other applications and therefore restricts the global throughput of the system. In general, solutions implemented by organizations dealing with Web scale data tend to adopt a non-transactional approach, or at least consistency rules less strict than the standard semantics.

14.3 Required properties of a distributed system

There is a long list of "**-ity" that characterize the good properties of distributed systems: reliability, scalability, availability, etc. We briefly review some of particular interest to the book's scope. The end of the section proposes a discussion on the ability of distributed systems to simultaneously maintain these good properties.

14.3.1 Reliability

Reliability denotes the ability of a distributed system to deliver its services even when one or several of its software of hardware components fail. It definitely constitutes one of the main expected advantages of a distributed solution, based on the assumption that a participating machine affected by a failure can always be replaced by another one, and not prevent the completion of a requested task. For instance, a common requirements of large electronic Web sites is that a user transaction should never be canceled because of a failure of the particular machine that is running that transaction. An immediate and obvious consequence is that reliability relies on *redundancy* of both the software components and data. At the limit, should the entire data center be destroyed by an earthquake, it should be replaced by another one that

has a replica of the shopping carts of the user. Clearly, this has a cost and depending of the application, one may more or less fully achieve such a resilience for services, by eliminating every *single point of failure*.

14.3.2 Scalability

The concept of scalability refers to the ability of a system to continuously evolve in order to support a growing amount of tasks. In our setting, a system may have to scale because of an increase of data volume, or because of an increase of work, e.g., number of transactions. We would like to achieve this scaling without performance loss. We will favor here *horizontal scalability* achieved by adding new servers. But, one can also consider *vertical scalability* obtained by adding more resources to a single server.

To illustrate these options, suppose we have distributed the workload of an application between 100 servers, in a somehow perfect and abstract manner, with each holding 1/100 of the data and serving 1/100 of the queries. Now suppose we get 20% more data, or 20% more queries, we can simply get 20 new servers. This is horizontal scalability that is virtually limitless for very parallelizable applications. Now we could also add extra disk/memory to the 100 servers (to handle the increase in data), and add extra memory or change the processors to faster ones (to handle the increase in queries). This is vertical scalability that typically reaches rather fast the limits of the machine.

In parallel computing, one further distinguishes *weak scalability* from *strong scalability* (see Figure 14.7). The former analyzes how the time to obtain a solution varies with respect to the processor count with a fixed data set size per processor. In the perfect case, this time remains constant (per processor), indicating the ability of the system to maintain a perfect balance. Strong scalability refers to the global throughput of a system, for a fixed data set size. If the throughput raises linearly as new servers are added, the system does not suffer from an overhead due to the management tasks associated to a distributed job. (Note that the above discussion assumes a linear complexity of the system behavior, which is true at least for basic read/write/search operations.)

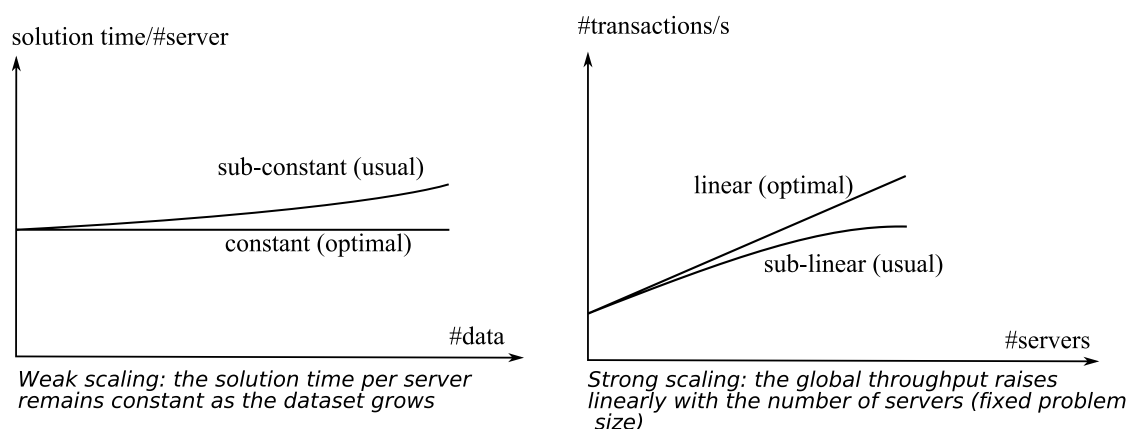

Figure 14.7: Dimensions of scalability

It is actually a common situation that the performance of a system, although designed (or claimed) to be scalable, declines with the system size, due to the management or environment

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

cost. For instance network exchanges may become slower because machines tend to be far apart from one another. More generally, it may happen that some tasks are not distributed, either because of their inherent atomic nature or because of some flaw in the system design. At some point, these tasks (if any) limit the speed-up obtained by distribution (a phenomenon known as Amdahl's law in the related context of parallel computing).

A scalable architecture avoids this situation and attempts to balance evenly the load on all the participating nodes. Let us consider the simple case of a server that would carry out 10% more work that the others, due to some special role. This is a source of non-scalability. For small workloads, such a difference is unnoticeable, but eventually it will reach an importance that will make the "stressed" node a bottleneck. However, a node dedicated to some administrative tasks that is really negligible or that does not increase proportionally to the global workload is acceptable.

Many architectures presented in the rest of this chapter are of type "one Master – many Servers". The Master is a node that handles a few specific tasks (e.g., adding a new server to the cluster or connecting a client) but does not participate to the core functionalities of the application. The servers hold the data set, either via a full replication (each item is present on each each server) or, more commonly, via "sharding": the data set is partitioned and each subset is stored on one server and replicated on a few others. This Master-Server approach is easier to manage than a cluster where all nodes play an equivalent role, and often remains valid on the long run.

14.3.3 Availability

A task that is partially allocated to a server may become idle if the server crashes or turns out to be unavailable for any reason. In the worst case, it can be delayed until the problem is fixed or the faulty server replaced by a replica. *Availability* is the capacity of a system to limit as much as possible this latency (note that this implicitly assumes that the system is already reliable: failures can be detected and repair actions initiated). This involves two different mechanisms: the failure (crash, unavailability, etc.) must be detected as soon as possible, and a quick recovery procedure must be initiated. The process of setting up a protection system to face and fix quickly node failures is usually termed *failover*.

The first mechanism is handled by periodically monitoring the status of each server ("heart-beat"). It is typically assigned to the node dedicated to administrative tasks (the "master"). Implementing this mechanism in a fully distributed way is more difficult due to the absence of a well-identified manager. Structured P2P networks promote one of the nodes as "Super-peer" in order to take in charge this kind of background monitoring surveillance. Note that some P2P approaches assume that a node will kindly inform its companions when it needs to leave the network, an assumption (sometimes called "fail-stop") that facilitates the design. This may be possible for some kinds of failures, but is unrealistic in many cases, e.g., for hardware errors.

The second mechanism is achieved through replication (each piece of data is stored on several servers) and redundancy (there should be more than one connection between servers for instance). Providing failure management at the infrastructure level is not sufficient. As seen above, a service that runs in such an environment must also take care of adopting adapted recovery techniques for preserving the content of its volatile storage.

14.3.4 Efficiency

How do we estimate the efficiency of a distributed system? Assume an operation that runs in a distributed manner, and delivers a set of items as result. Two usual measures of its efficiency are the *response time* (or latency) that 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). These measures are useful to qualify the practical behavior of a system at an analytical level, expressed as a function of the network traffic. The two measures correspond to the following 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.

The complexity of operations supported by distributed data structures (e.g., searching for a specific key in a distributed index) can be characterized as a function of one of these cost units.

Generally speaking, the analysis of a distributed structure in terms of number of messages is over-simplistic. It ignores the impact of many aspects, including the network topology, the network load and its variation, the possible heterogeneity of the software and hardware components involved in data processing and routing, etc. However, developing a precise cost model that would accurately take into account all these performance factors is a difficult task, and we have to live with rough but robust estimates of the system behavior.

14.3.5 Putting everything together: the CAP theorem

We now come to the question of building systems that simultaneously satisfy all the properties expected from a large-scale distributed system. It should scale to an unbounded number of transactions on unlimited data repositories, always be available with high efficiency (say, a few milliseconds to serve each user's request) and provide strong consistency guarantees.

In a keynote speech given in 2000 at the *Symposium on Principles of Distributed Computing*, Eric Brewer proposed the following conjecture: no distributed system can simultaneously provide all three of the following properties: **C**onsistency (all nodes see the same data at the same time), **A**vailability (node failures do not prevent survivors from continuing to operate), and **P**artition tolerance (the system continues to operate despite arbitrary message loss). This conjecture, formalized and proved two years later, is now known as the CAP theorem, and strongly influences the design of Web-scale distributed systems.

The problem can be simply explained with a figure (Fig. 14.8). Assume two applications A and B running on two distinct servers S_1 and S_2. A executes writes to a repository, whereas B reads from a replicated version of the repository. The synchronization is obtained by replication messages sent from S_1 to S_2.

When the Client application sends a *put(d)* to update a piece of data d, A receives the request and writes in its local repository; S_1 then sends the replication message that replaces d', the older replica, with d, and a subsequent *read(d)* sent by the Client retrieves from S_2 the updated version. So, the system seems consistent.

Now, assume a failure in the system that entails a loss of messages. If we want the system to be fault-tolerant, it continues to run, and the replica is out of date: the Client receives an old

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

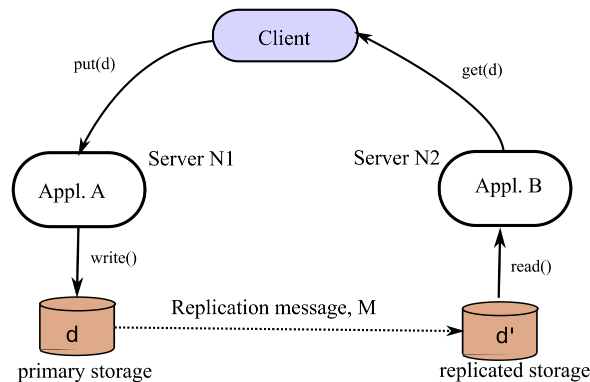

Figure 14.8: The CAP theorem illustrated

version of its data (inconsistency). If S_1 synchronizes the write operation and the replication message M as an *atomic* transaction, this goes against availability, because waiting for the acknowledgment of S_2 may take an unpredictable amount of time.

The CAP theorem essentially says that there is a trade-off between availability and consistency (partition tolerance is something we have to deal with anyway) in large-scale distributed systems. In an "eventual consistency" model, the replication message is asynchronous, but S_1 resends the messages if it does not receive an acknowledgment until, *eventually*, the replica on S_2 is known to be consistent with S_1. Meanwhile, the Client may have to deal with an inconsistent state. In concrete terms, if you remove an item from your basket, it possibly re-appears later in the transaction! Obviously, this is a better choice for the e-commerce site than a user who gives up her transaction due to high system latency.

The CAP theorem gave rise to debates regarding its exact definition and consequences. We already noted that the partition tolerance property is not symmetric to the other ones, since we do not really have the choice to give it up. This leaves two possible combinations: CP (consistent and partition tolerant) and AP (available and partition tolerant). Moreover, the concept of availability (a transaction always terminates) ignores the efficiency aspect (how long does it take?) which is an important factor. Still, the theorem points out that consistency and availability are central, and somehow incompatible, issues in the design of distributed systems, and that a clear trade-off should be made explicit.

14.4 Particularities of P2P networks

A *peer-to-peer* network is a large network of nodes, called *peers*, that agree to cooperate in order to achieve a particular task. A P2P system is a distributed system, and as such it shares a lot of features with the settings previously presented.

What makes P2P systems particular with respect to the cluster systems examined so far is their very loose and flexible (not to say unstable) organization. Peers often consist of personal computers connected to the network (e.g., the Internet) participating in a specific task. The rationale behind P2P emergence is the huge amount of available CPU, memory,

disk, network resources available on the Web. One would like to use these existing resources to support heavy applications as close to zero hardware cost. Furthermore, this approach allows achieving high scalability using massively distribution and parallel computation.

A second particularity is that a peer plays simultaneously the role a client (of other peers) and a server (to other peers). This is in fact not such a strong specificity, if we recall that "Client" and "Server" actually denote processes hosted on possibly the same computer. Nothing in a distributed architecture prevents the same machine from running several processes, possibly client/server from one another. In P2P systems, however, this situation becomes the rule. A canonical application is file-sharing: a Client (node) gets a file from another (Server) node, and the file, once stored on the Client disk, becomes available to other peers (so, the former Client becomes indeed a Server). In theory, this leads to high availability, reliability (due to large replication) and adequate load balancing.

P2P systems raise many problems, though, even if we set aside the somewhat illegal nature of their most popular applications. First, the behavior of each peer is fully autonomous. A peer owns its computing power and storage resource and can independently choose to allocate these resources to a particular task. A peer can also join or leave the system at will (as mentioned above, the fail-stop hypothesis hardly holds in practice). Second, P2P networks connect nodes via a possibly slow communication channel (usually, the Internet) and this may bring a quite high communication overhead compared to a cluster of machine on a very high-speed local network (See Table 14.1, page 284). Finally, the lack of control on the infrastructure makes P2P networks not adapted to very rapidly changing data and high quality of services, and in particular not adapted to transactional tasks.

Peers in a P2P network refer to each other by their IP addresses, forming a structure over the Internet called an *overlay network* (e.g., a graph laid over a physical infrastructure). A peer p in this structure is connected to a few other peers (often called its "friends") which are its primary (and possibly) unique way to communicate with the rest of the system. P2P systems mostly differ by the topology of their overlay network, which dictates how dictionary operations (insert, search, update) can be implemented. We should be aware nevertheless that even if two peers p_1 and p_2 seem friends in the overlay, a message sent from p_1 to p_2 must actually follow a physical route in the underlying Internet graph, with possibly many hops. So, things may not work as nicely as expected when considering the overlay topology.

A general (although not very efficient) search technique is *flooding*: a peer p disseminates its request to all its friends, which flood in turn their own friends distinct from p, and so on until the target of the request (e.g., a peer holding the requested music file) is reached. A P2P system that only supports flooding is called an *unstructured P2P network*. The approach is simple and works as follows. A peer only needs to know some friends to join a network. From them, it can discover new friends. Queries are then supported using flooding typically limited by a "Time to live" bound (abbreviated TTL). The TTL limits the number of times a particular query is forwarded before it should be discarded to avoid using too much resource on a single query. Unstructured P2P networks are not very efficient. They are in particular inherently unstable. Because the peers in the community are autonomous and selfish, one can often observe a very high rate of peers going in and out of the system (one speaks of high *churn*). As a consequence, it is difficult to guarantee that a node stays connected to the system, or that the overall topology remains consistent.

More structured ways of looking up the network ("Structured P2P networks") have been designed to avoid the blind and uncontrolled nature of the flooding mechanism among which *Distributed Hash Tables* (DHTs) are probably the most popular. Joining the network becomes

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

For personal use only, not for distribution.

more involved, but the performance and stability are improved. We will consider DHTs in Chapter 15.