1. What is a Distributed System?

Various definitions of distributed systems have been given. For our purposes it is sufficient to give a loose characterization:

"A distributed system is a collection of autonomous computing elements that appears to its users as a single coherent system."

This definition refers to two characteristic features of distributed systems:

1. a distributed system is a collection of computing elements each being able to behave independently of each other.
2. users (be they people or applications) believe they are dealing with a single system. This means that one way or another the autonomous nodes need to collaborate.

2. Goals of a Distributed system

A distributed system should make resources easily accessible; it should hide the fact that resources are distributed across a network; it should be open; and it should be scalable.

Supporting resource sharing - An important goal of a distributed system is to make it easy for users (and applications) to access and share remote resources. Peer-to-peer networks are often associated with distribution of media files such as audio and video. In other cases, the technology is used for distributing large amounts of data, as in the case of software updates, backup services, and data synchronization across multiple servers.

Making distribution transparent - An important goal of a distributed system is to hide the fact that its processes and resources are physically distributed across multiple computers, possibly separated by large distances. In other words, it tries to make the distribution of processes and resources transparent, that is, invisible, to end users and applications.

Being open - Another important goal of distributed systems is openness. An open distributed system is essentially a system that offers components that can easily be used by, or integrated into other systems.

Being scalable - scalability has become one of the most important design goals for developers of distributed systems. There are 3 kinds of scalability:

1. **Size scalability** - A system can be scalable with respect to its size, meaning that we can easily add more users and resources to the system without any noticeable loss of performance.

2. **Geographical scalability** - A geographically scalable system is one in which the users and resources may lie far apart, but the fact that communication delays may be significant is hardly noticed.

3. **Administrative scalability** - An administratively scalable system is one that can still be easily managed even if it spans many independent administrative organizations.

3. Scaling Techniques

In most cases, scalability problems in distributed systems appear as performance problems caused by limited capacity of servers and network. Simply improving their capacity is often a solution, referred to as scaling up.
When it comes to scaling out, that is, expanding the distributed system by essentially deploying more machines, there are basically only three techniques we can apply:

1. Hiding communication latencies
2. Distribution of work
3. Replication

3.1. Hiding communication latencies

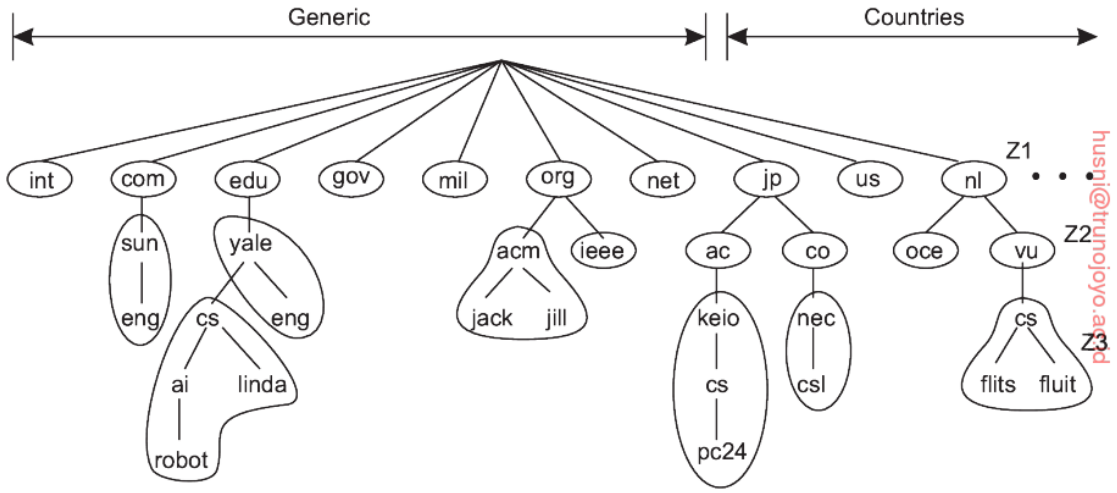
The basic idea is simple: try to avoid waiting for responses to remote-service requests as much as possible. Essentially, this means constructing the requesting application in such a way that it uses only **asynchronous communication**.
When a reply comes in, the application is interrupted and a special handler is called to complete the previously issued request. Alternatively, a new thread of control can be started to perform the request. Although it blocks waiting for the reply, other threads in the process can continue.

However, there are many applications that cannot make effective use of asynchronous communication. In such cases, a much better solution is to reduce the overall communication, for example, by moving part of the computation that is normally done at the server to the client process requesting the service.

3.2. Partitioning and distribution

Another important scaling technique is partitioning and distribution, which involves taking a component, splitting it into smaller parts, and subsequently spreading those parts across the system. A good example of partitioning and distribution is the Internet Domain Name System

(DNS). The DNS name space is hierarchically organized into a tree of domains, which are divided into non overlapping zones, as shown for the original DNS.



3.3. Replication

Considering that scalability problems often appear in the form of performance degradation, it is generally a good idea to actually replicate components across a distributed system.

Replication not only increases availability, but also helps to balance the load between components leading to better performance.

Caching is a special form of replication, although the distinction between the two is often hard to make or even artificial. As in the case of replication, caching results means making a copy of a resource, generally in the proximity of the client accessing that resource.
However, in contrast to replication, caching is a decision made by the client of a resource and not by the owner of a resource.

There is one serious drawback to caching and replication that may adversely affect scalability. Because we now have multiple copies of a resource, modifying one copy makes that copy different from the others. Consequently, caching and replication leads to **consistency problems**.

4. Pitfalls

Distributed systems differ from traditional software because components are dispersed across a network. Peter Deutsch, at the time working at Sun Microsystems, formulated these flaws as the following false assumptions that everyone makes when developing a distributed application for the first time:

- The network is reliable
- The network is secure
- The network is homogeneous
- The topology does not change
- Latency is zero
- Bandwidth is infinite
- Transport cost is zero
- There is one administrator

Note how these assumptions relate to properties that are unique to distributed systems: reliability, security, heterogeneity, and topology of the network; latency and bandwidth; transport costs; and finally administrative domains. When developing non distributed applications, most of these issues will most likely not show up.

5. Message-based Communication

Message-based communication is a type of IPC (Inter-Process Communication) mechanism in which processes or threads communicate with each other by sending messages. In this approach, a process sends a message to another process using a message queue, and the receiving process reads the message from the queue.

A message is an atomic bit string whose format and meaning are specified by a communications protocol. The transport of a message from its source to its destination is performed by a computer network.

Advantages:

1. Simplicity
2. Asynchronous communication - sender and receiver do not have to be active at the same time
3. Reliability - messages can be re-sent until they are received by the receiver
4. Scalability - messages can be routed between multiple computers or networks

Disadvantages:

1. Overhead
2. Complexity
3. Latency
4. Limited data size

6. Properties of a Communication Channel

- Connection-based vs. connectionless
- Reliable vs. unreliable (may loose messages)
- Maintains order of messages vs does not
- Message-based vs. stream-based
- With vs without flow control
- Number of ends of the channel

6.1. Connection-based vs. Connectionless

Connection-oriented communication is a communication protocol where a communication session or a semi-permanent connection is established before any useful data can be transferred – analogous to the telephone network.

Connection-oriented protocol services are often, but not always, reliable network services that provide acknowledgment after successful delivery and automatic repeat request functions in case of missing or corrupted data.

Connectionless communication is a data transmission method used in packet switching networks in which each data unit is individually addressed and routed based on information carried in each unit, rather than in the setup information of a prearranged, fixed data channel – analogous to mail.

6.2. Reliable vs. Unreliable

Reliable connections ensure that the data sent is delivered to the respective destination, under some assumptions. If not, the communicating processes are notified.

In unreliable connections it is up to the communicating processes to detect the loss of messages and proceed as required by the application.

Sometimes the packets can be sent in duplicates to up the chances of its receivement. Channels that send more than one copy of the same packet “generate duplicates”, as opposed to “no duplicates” channels.

6.3. Order

An ordered channel ensures that the data is delivered to its recipients in the order in which it was sent.

In an unordered channel, if it is important to preserve the order, it is up to the application to detect that the data is out of order and if necessary to reorder it.

Order and reliability are orthogonal, meaning they are not directly correlated.
(obrigado por esta, tiago)

6.4. Message-based vs. stream-based

Message (datagram) channels support the transport of messages – sequences of bits processed atomically. A datagram socket is like passing a note in class. Consider the case where you are not directly next to the person you are passing the note to; the note will travel from person to person. It may not reach its destination, and it may be modified by the time it gets there.

Stream-based channels do not support messages. Essentially, they work as a pipe for a sequence of bytes – analogous to Unix pipes. A stream socket is like a phone call -- one side places the call, the other answers, you say hello to each other (SYN/ACK in TCP), and then you exchange information. Once you are done, you say goodbye (FIN/ACK in TCP).

6.5. Flow control

In data communications, flow control is the process of managing the rate of data transmission between two nodes to prevent a fast sender from overwhelming a slow receiver. It does not necessarily mean that the sender has more computing power than the receiver. Some communications channels can choose to implement Flow Control techniques (Stop-And-Wait, Go-Back-N, Selective repeat).

6.6. Number of ends

Number of ends of the communication channel.

1. **Unicast** or **point-to-point** = only two ends
2. **Broadcast** = all nodes in the "network"

3. **Multicast** = subset of nodes in the network

7. How to identify a process on the Internet?

Communication endpoints (e.g., sockets) are characterized by a pair of identifiers:

1. IP Address: Identifies the computer on the Internet; it is a 32-bit number, usually represented in dotted decimal, e.g.: 193.136.28.31.

2. Port Number: Identifies the endpoint of a communication channel inside a computer (transport layer identifier); it is a 16-bit integer (between 0 and 65535); some ports are reserved for some services.

This convention is similar to a residence phone number: the person that answers a call may change (process may differ), but the number (endpoint) stays the same.

8. UDP protocol

UDP channels transport messages called *UDP datagrams*.
The UDP API supports two operations: send() e receive(); Each message is transmitted by invoking send() once; If delivered, the message will be delivered atomically, in a single invocation of receive().

Datagrams have a maximum size of 65535 bytes. Applications may have to split the data to send in datagrams before transmitting, and reassemble the data from the fragments after receiving.

UDP allows a process to start transmitting data immediately, but it also requires the specification of the other channel endpoint on every invocation of send(). UDP supports multicast.

UDP has no flow control, a receiver may be flooded with requests and run out of resources (e.g. buffers) to receive other messages.

9. TCP protocol

TCP does not ensure the "separation" between bytes sent by invoking two send() calls. write() and read() match TCP semantics better.

TCP is connection-oriented. Communication with TCP has 3 phases:

1. Connection set up
2. Data exchange

3. Connection tear down

TCP ensures reliability (both loss and duplication of data). In the event of communication problems, the connection may be aborted and the processes on its ends notified.

TCP ensures flow control. Prevents data loss because of insufficient resources. Nevertheless, TCP may be vulnerable to denial-of-service attacks, i.e. SYN attacks. TCP channels have only two endpoints, supporting the communication between only two processes.

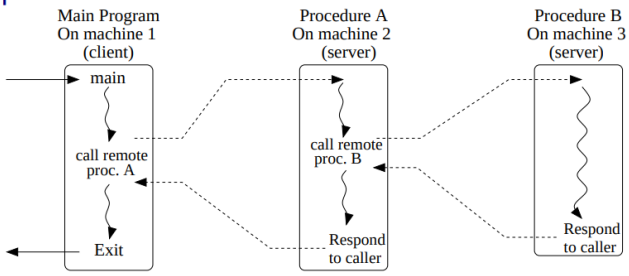
Unlike what happens with UDP, TCP channels on the same computer may have the same port number.

10. Remote Procedure Call

RPCs allow programs to call procedures located on other machines. When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing at all is visible to the programmer.

We want RPC to be transparent—the calling procedure should not be aware that the called procedure is executing on a different machine or vice versa.

Remote procedure call:



A remote procedure, called a **client stub**, is offered to the calling client. It is called using a normal calling sequence. However, instead of a local procedure call, it packs the parameters into a message and requests that message to be sent to the server.

When the message arrives at the server, the server's operating system passes it to a server stub. A **server stub** is the server-side equivalent of a client stub: it is a piece of code that transforms requests coming in over the network into local procedure calls.
From the server's point of view, it is as though it is being called directly by the client - the parameters and return address are all on the stack where they belong and nothing seems unusual. The server performs its work and then returns the result to the caller (in this case the server stub) in the usual way.

When the result message arrives at the client's machine, the operating system passes it through the receive() operation. The client stub inspects the message, unpacks the result, copies it to its caller, and returns in the usual way. It has no idea that the work was done remotely at another machine.

11. Parameter passing

When exchanging parameters, which travel as bytes, we need to handle the case that the placement of bytes in memory may differ between machine architectures. The Intel format is **little endian** and the (older) ARM format is **big endian**, for example. However, big endian is what is normally used for transferring bytes across a network.

The solution to this problem is to transform data that is to be sent to a machine, making sure that both communicating parties expect the same message data type to be transmitted.

We now come to a difficult problem: How are pointers, or in general, references passed? A pointer is meaningful only within the address space of the process in which it is being used.

One solution is just to forbid pointers and reference parameters in general. However, these are so important that this solution is highly undesirable. In fact, it is often not necessary either.

First, reference parameters are often used with fixed-sized data types, such as static arrays, or with dynamic data types for which it is easy to compute their size at runtime, such as strings or dynamic arrays. In such cases, we can simply copy the entire data structure to which the parameter is referring, effectively replacing the copy-by-reference mechanism by copy-by-value.

Pointer problems can be alleviated by using global references: references that are meaningful to the calling and the called process. For example, if the client and the server have access to the same file system, passing a file handle instead of a pointer may do the trick. There is one important observation: both processes need to know exactly what to do when a global reference is passed.

12. RPC semantics in the presence of failures

The goal of RPC is to hide communication by making remote procedure calls look just like local ones. A problem comes about when errors occur. It is then that the differences between local and remote calls are not always easy to mask.

There are five different classes of failures that can occur in RPC systems:

1. The client is unable to locate the server
2. The request message from the client to the server is lost
3. The server crashes after receiving a request
4. The reply message from the server to the client is lost
5. The client crashes after sending a request

11.1. Client cannot locate the server

When the client is run, the binder will be unable to match it up with a server and will report failure. One possible solution is to have the error raise an exception.

This approach has drawbacks. To start with, not every language has exceptions or signals. Another point is that having to write an exception or signal handler destroys the transparency we have been trying to achieve. Writing an exception handler for “Cannot locate server” would be a rather unusual request in a non distributed system.

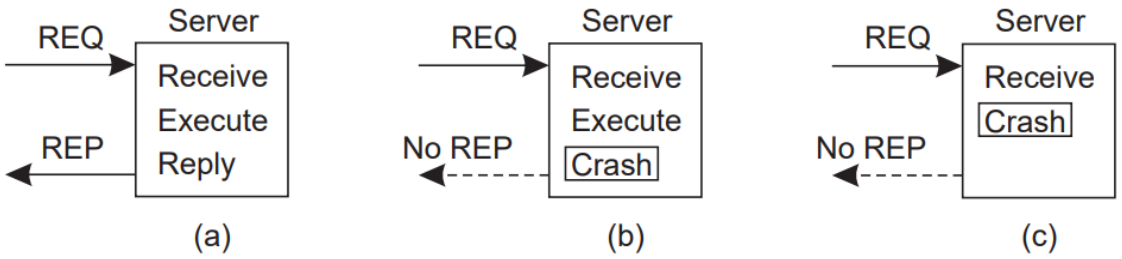


Figure 8.18: A server in client-server communication. (a) The normal case. (b) Crash after execution. (c) Crash before execution.

11.2. Lost request messages

This is the easiest one to deal with: just have the operating system or client stub start a timer when sending the request. If the timer expires before a reply or acknowledgment comes back, the message is sent again. If the message was truly lost, the server will not be able to tell the difference between the retransmission and the original, and everything will work fine.

11.3. Server crashes

A request arrives and is carried out, just as before, but the server crashes before it can send the reply. And, of course, no reply is sent back.

The annoying part of Figure 8.18 is that the correct treatment differs for (b) and (c). In (b) the system has to report failure back to the client (e.g., raise an exception), whereas in (c) it can just retransmit the request. The problem is that the client's operating system cannot tell which is which. All it knows is that its timer has expired.

Three schools of thought exist on what to do here.

One philosophy is to wait until the server reboots and try the operation again. This technique is called **at-least-once** semantics and guarantees that the RPC has been carried out at least one time, but possibly more.
The second philosophy gives up immediately and reports back failure. This approach is called **at-most-once** semantics and guarantees that the RPC has been carried out at most one time, but possibly not at all.
The third philosophy is to guarantee nothing. When a server crashes, the client gets no help and no promises about what happened. The RPC may have been carried out anywhere from zero to a large number of times. The main virtue of this scheme is that it is easy to implement.

None of these are terribly attractive. What one would like is **exactly-once** semantics, but in general, there is no way to arrange this.

There are two strategies the server can follow. It can either send a completion message just before it actually tells the document processor to do its work, or after the document has been processed.

Assume that the server crashes and subsequently recovers. It announces to all clients that it has just crashed but is now up and running again. The problem is that the client does not know whether its request to process a document will actually have been carried out.

There are four strategies the client can follow.

First, the client can decide to never reissue a request, at the risk that the document will not be processed.
Second, it can decide to always reissue a request, but this may lead to the document being processed twice (which may easily incur a significant amount of work when dealing with intricate documents).
Third, it can decide to reissue a request only if it did not yet receive an acknowledgment that its request had been delivered to the server. In that case, the client is counting on the fact that the server crashed before the request could be delivered.
The fourth and last strategy is to reissue a request only if it has received an acknowledgment for the request.

With two strategies for the server, and four for the client, there are a total of eight combinations to consider. Unfortunately, as it turns out, no combination is satisfactory: it can be shown that for any combination either the request is lost forever, or carried out twice.

** detalhes sobre as combinações todas na página 467, Note 8.9

11.3. Lost reply messages

Lost replies can also be difficult to deal with. The obvious solution is just to rely on a timer again that has been set by the client's operating system. If no reply is forthcoming within a reasonable period, just send the request once more.
The trouble with this solution is that the client is not really sure why there was no answer. Did the request or reply get lost, or is the server merely slow? It may make a difference.

One way of solving this problem is to try to structure all the requests in an idempotent way. In practice, however, many requests are inherently non-idempotent, so something else is needed. Another method is to have the client assign each request a sequence number. By having the server keep track of the most recently received sequence number from each

client that is using it, the server can tell the difference between an original request and a retransmission and can refuse to carry out any request a second time. However, the server will still have to send a response to the client.

11.4. Lost reply messages

What happens if a client sends a request to a server to do some work and crashes before the server replies? At this point a computation is active and no parent is waiting for the result. Such an unwanted computation is called an **orphan (computation)**.

Orphan computations can cause a variety of problems that can interfere with normal operation of the system. As a bare minimum, they waste processing power. They can also lock files or otherwise tie up valuable resources.

** podia falar de soluçoes para isto mas não esta na matéria

Simple services/applications

At-least-once

- Suits if requests are idempotent

At-most-once

- Appropriate when requests are not idempotent

Issue Some requests may be idempotent whereas others are not

- Need to pick one semantics

More sophisticated services/applications

- No clear advantage: the service/application itself may have to take special measures

12. Threads

The granularity of processes as provided by the operating systems on which distributed systems are built is not sufficient. Having a finer granularity in the form of multiple threads of control per process makes it much easier to build distributed applications and to get better performance.

12.1. Introduction

A **process** is often defined as a program in execution, that is, a program that is currently being executed on one of the operating system's virtual processors.

The operating system takes great care to ensure that independent processes cannot maliciously or inadvertently affect the correctness of each other's behavior, in other words, the sharing of the same CPU and other hardware resources is made transparent.

This concurrency transparency comes at a price. Each time a process is created, the operating system must create a complete independent address space.

Similarly to a process, a thread executes its own piece of code, independently from other threads. However, in contrast to processes, no attempt is made to achieve a high degree of concurrency transparency if this would result in performance degradation. Therefore, a thread system generally maintains only the minimum information to allow a CPU to be shared by several threads.

There are two important implications of deploying threads.

First of all, the performance of a multithreaded application need hardly ever be worse than that of its single-threaded counterpart. In fact, in many cases, multithreading even leads to a performance gain.
Second, because threads are not automatically protected against each other the way processes are, development of multithreaded applications requires additional intellectual effort.

12.2. Thread usage in non distributed systems

Instead of using processes, an application can be constructed such that different parts are executed by separate threads. Communication between those parts is entirely dealt with by using shared data. Thread switching can sometimes be done entirely in user space, although in other implementations, the kernel is aware of threads and schedules them. The effect can be a dramatic improvement in performance.

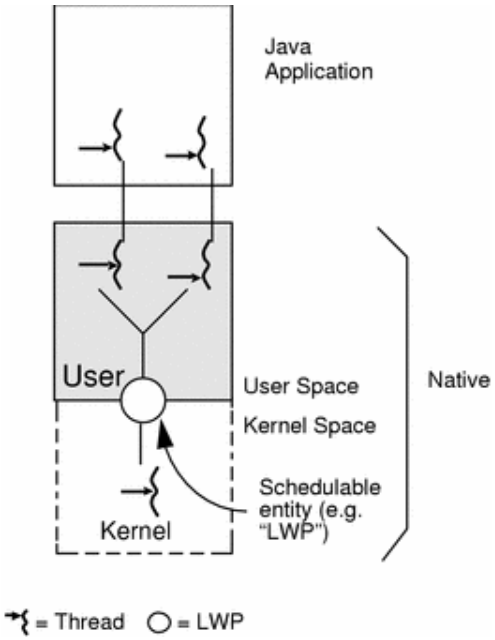
12.3. Thread Implementation

There are basically two approaches to implement a thread package. The first approach is to construct a thread library that is executed entirely in user space. The second approach is to have the kernel be aware of threads and schedule them.

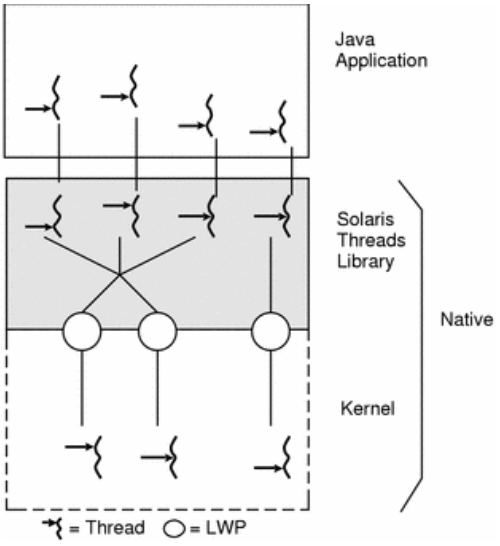
A **user-level thread** library has a number of advantages. First, it is cheap to create and destroy threads. Because all thread administration is kept in the user's address space, the price of creating a thread is primarily determined by the cost for allocating memory to set up a thread stack. Analogously, destroying a thread mainly involves freeing memory for the stack, which is no longer used. Both operations are cheap.

A second advantage of user-level threads is that switching thread context can often be done in just a few instructions. Only the values of the CPU registers need to be stored and subsequently reloaded with the previously stored values of the thread to which it is being switched.

A major drawback of user-level threads comes from deploying the **many-to-one** threading model: multiple threads are mapped to a single schedulable entity. As a consequence, the invocation of a blocking system call will immediately block the entire process to which the thread belongs, and thus also all the other threads in that process.



These problems can be mostly circumvented by implementing **kernel-level threads** in the operating system's kernel, leading to what is known as the **one-to-one** threading model in which every thread is a schedulable entity. The price to pay is that every thread operation will have to be carried out by the kernel, requiring a system call.



OBJ

12.3. Multithreaded clients

So far we have seen two possible server designs: a multithreaded file server and a single-threaded file server.
A third alternative is to run the server as a big single-threaded finite-state machine. When a request comes in, the one and only thread examines it. If it can be satisfied from the in-memory cache, fine, but if not, the thread must access the disk.

However, instead of issuing a blocking disk operation, the thread schedules an asynchronous disk operation for which it will be later interrupted by the operating system. To make this work, the thread will record the status of the request and continues to see if there were any other incoming requests that require its attention. Once a pending disk operation has been completed, the operating system will notify the thread.

In this design, the “sequential process” model that we had in the first two cases is lost. Every time the thread needs to do a blocking operation, it needs to record exactly where it was in processing the request, possibly also storing additional state.

Model	Characteristics
Multithreading	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls

Figure 3.5: Three ways to construct a server.

It should now be clear what threads have to offer. They make it possible to retain the idea of sequential processes that make blocking system calls and still achieve parallelism.

13. Client-side software for distribution transparency

In many cases, parts of the processing and data level in a client-server application are executed on the client side as well.
Ideally, a client should not be aware that it is communicating with remote processes. In contrast, distribution is often less transparent to servers for reasons of performance and correctness.

There are different ways to handle location, migration, and relocation transparency. Using a convenient naming system is crucial. In many cases, cooperation with client-side software is also important.

Many distributed systems implement replication transparency by means of client-side solutions. For example, imagine a distributed system with replicated servers, such replication can be achieved by forwarding a request to each replica. Client-side software can transparently collect all responses and pass a single response to the client application.

Regarding failure transparency, masking communication failures with a server is typically done through client middleware. For example, client middleware can be configured to repeatedly attempt to connect to a server, or perhaps try another server after several attempts.

Finally, concurrency transparency can be handled through special intermediate servers, notably transaction monitors, and requires less support from client software.

14. Servers

14.1. Concurrent vs. Iterative servers

In an **iterative server**, the server itself handles the request and, if necessary, returns a response to the requesting client.

A **concurrent server** does not handle the request itself, but passes it to a separate thread or another process, after which it immediately waits for the next incoming request.
A multithreaded server is an example of a concurrent server. An alternative implementation of a concurrent server is to fork a new process for each new incoming request.

14.2. Stateless versus stateful servers

A **stateless server** does not keep information on the state of its clients, and can change its own state without having to inform any client. A Web server, for example, is stateless. It merely responds to incoming HTTP requests, which can be either for uploading a file to the server or (most often) for fetching a file.

Note that in many stateless designs, the server actually does maintain information on its clients, but crucial is the fact that if this information is lost, it will not lead to a disruption of the service offered by the server. For example, a Web server generally logs all client requests.

A particular form of a stateless design is where the server maintains what is known as **soft state**. In this case, the server promises to maintain state on behalf of the client, but only for a limited time. After that time has expired, the server falls back to default behavior, thereby discarding any information it kept on account of the associated client.
In contrast, a **stateful server** generally maintains persistent information on its clients. This means that the information needs to be explicitly deleted by the server. A typical example is

a file server that allows a client to keep a local copy of a file, even for performing update operations. Such a server would maintain a table containing (client, file) entries.

This approach can improve the performance of read and write operations as perceived by the client. Performance improvement over stateless servers is often an important benefit of stateful designs. However, if the server crashes, it has to recover its table of (client, file) entries, or otherwise it cannot guarantee that it has processed the most recent updates on a file. In general, a stateful server needs to recover its entire state as it was just before the crash.

Nonetheless, one should actually make a distinction between (temporary) **session state** and **permanent state**. The example above is typical for session state: it is associated with a series of operations by a single user and should be maintained for some time, but not indefinitely.
Usually, session state is maintained in three-tiered client-server architectures, where the application server actually needs to access a database server through a series of queries before being able to respond to the requesting client. Hence, no real harm is done if the session state is lost, provided that the client can simply re-issue the original request.

When designing a server, the choice for a stateless or stateful design should not affect the services provided by the server. For example, if files have to be opened before they can be read from, or written to, then a stateless server should one way or the other mimic this behavior.

What if the service keeps state in main memory and the service uses load balancing (distributing network traffic across resources)?

We need to ensure that the state is accessible to all users. For example, by saving it in a data-base.

14.3. Security

Challenge: servers execute with priviledges that their clients usually do not have

Solution: servers must

- authenticate clients: i.e. "ensure" that a client is who it claims to be;
- control access to resources: i.e. check whether the client has the necessary permissions to execute the operation it requests.

- A related requirement is data **confidentiality**
  - need to encrypt data transmitted over the network
- Code migration (i.e. downloaded from the network) raises even more issues.

15. Fault Tolerance

A characteristic feature of distributed systems that distinguishes them from single-machine systems is the notion of partial failure: part of the system is failing while the remaining part continues to operate, and seemingly correctly.
Whenever a failure occurs, the system should continue to operate in an acceptable way while repairs are being made. In other words, a distributed system is expected to be fault tolerant.

Achieving fault tolerance and reliable communication are strongly related.

15.1. Basic Concepts

Being fault tolerant is strongly related to what are called **dependable systems**. Dependability is a term that covers a number of useful requirements for distributed systems including the following.

- Availability
- Reliability
- Safety
- Maintainability

Availability is defined as the property that a system is ready to be used immediately. In general, it refers to the probability that the system is operating correctly at any given moment and is available to perform its functions on behalf of its users.

Reliability refers to the property that a system can run continuously without failure. In contrast to availability, reliability is defined in terms of a time interval instead of an instant in time. A highly reliable system is one that will most likely continue to work without interruption during a relatively long period of time.

If a system goes down on average for one, seemingly random millisecond every hour, it has an availability of more than 99.9999 percent, but is still unreliable. Similarly, a system that never crashes but is shut down for two specific weeks every August has high reliability but only 96 percent availability. The two are not the same!

Traditionally, fault-tolerance has been related to the following three metrics:

- **Mean Time To Failure (MTTF)**: The average time until a component fails.
- **Mean Time To Repair (MTTR)**: The average time needed to repair a component.
- **Mean Time Between Failures (MTBF)**: Simply MTTF + MTTR.

$$A = \frac{MTTF}{MTBF} = \frac{MTTF}{MTTF + MTTR}$$

A system is said to **fail** when it cannot meet its promises. In particular, if a distributed system is designed to provide its users with a number of services, the system has failed when one or more of those services cannot be (completely) provided.

An **error** is a part of a system's state that may lead to a failure. The cause of an error is called a **fault**. Clearly, finding out what caused an error is important.

As an example, a crashed program is clearly a failure. In this case, the programmer is the fault of the error (programming bug), in turn leading to a failure (a crashed program).

15.2. Failure Models

Type of failure	Description of server's behavior
Crash failure	Halts, but is working correctly until it halts
Omission failure	Fails to respond to incoming requests
Receive omission	Fails to receive incoming messages
Send omission	Fails to send messages
Timing failure	Response lies outside a specified time interval
Response failure	Response is incorrect
Value failure	The value of the response is wrong
State-transition failure	Deviates from the correct flow of control
Arbitrary failure	May produce arbitrary responses at arbitrary times

Figure 8.2: Different types of failures.

A **crash failure** occurs when a server prematurely halts, but was working correctly until it stopped.

An **omission failure** occurs when a server fails to respond to a request. In the case of a *receive-omission* failure, possibly the server never got the request in the first place. Likewise, a *send-omission* failure happens when the server has done its work, but somehow fails in sending a response.

Timing failures occur when the response lies outside a specified real-time interval. For example, a server responds too late.

The most serious are **arbitrary failures**, also known as **Byzantine failures**. In effect, when arbitrary failures occur, clients should be prepared for the worst. In particular, it may happen that a server is producing output it should never have produced, but which cannot be detected as being incorrect.

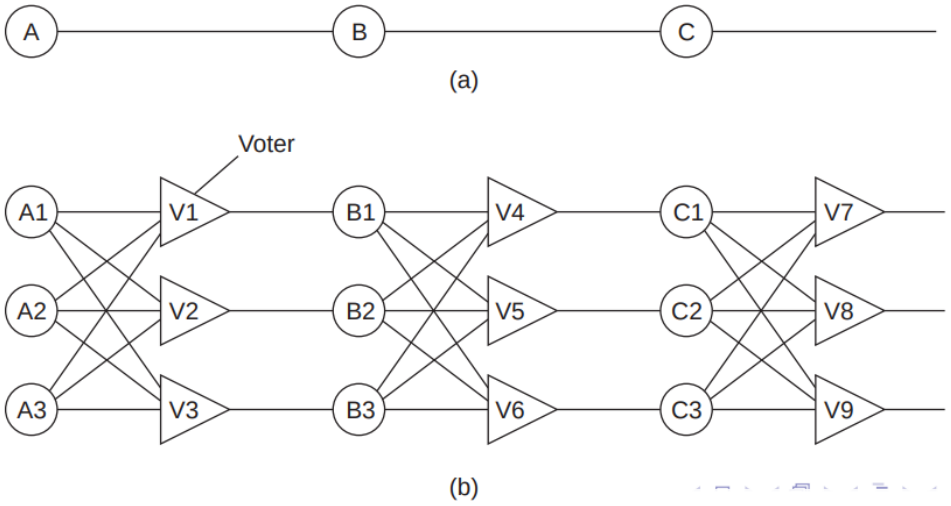
15.3. Synchronous vs Asynchronous System

In an **asynchronous system**, no assumptions about process execution speeds or message delivery times are made. The consequence is that when process P no longer perceives any actions from Q, it cannot conclude that Q crashed. Instead, it may just be slow or its messages may have been lost.

In a **synchronous system**, process execution speeds and message-delivery times are bounded. This also means that when Q shows no more activity when it is expected to do so, process P can rightfully conclude that Q has crashed.

15.4. Triple modular redundancy

Consider, for example, the circuit of figure (a). Here signals pass through devices A, B, and C, in sequence. If one of them is faulty, the final result will probably be incorrect.



In figure (b), each device is replicated three times. Following each stage in the circuit is a triplicated voter. Each voter is a circuit that has three inputs and one output. If two or three of the inputs are the same, the output is equal to that input. If all three inputs are different, the output is undefined. This kind of design is known as Triple Modular Redundancy (TMR).

At first it may not be obvious why three voters are needed at each stage. After all, one voter could also detect and pass though the majority view. However, a voter is also a component and can also be faulty.

16. Atomic Commitment

What is often needed in a distributed system is the guarantee that a message is delivered to either all group members or to none at all. This is also known as the **atomic multicast problem**.
The atomic multicasting problem is an example of a more general problem, known as **distributed commit**. The distributed commit problem involves having an operation being performed by each member of a process group, or none at all. In the case of reliable multicasting, the operation is the delivery of a message.

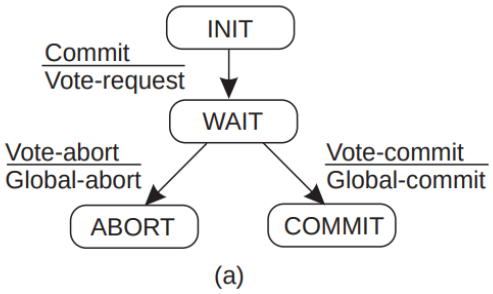
ACID Properties:

- **A**tomicity: either all operations of a transaction are executed or none of them
- **C**onsistency: a transaction transforms a consistent state into another consistent state
- **I**solation: the effects of a transaction are as if no other transactions executed concurrently
- **D**urability: the effects of a transaction that commits are permanent

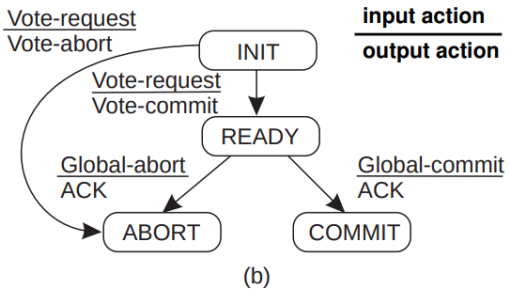
16.1. Two-phase commit protocol

The original two-phase commit protocol (2PC) consists of the following two phases, each consisting of two steps.

1. The coordinator sends a *vote-request* message to all participants.
2. When a participant receives a *vote-request* message, it returns either a *vote-commit* message to the coordinator telling the coordinator that it is prepared to locally commit its part of the transaction, or otherwise a *vote-abort* message.
3. The coordinator collects all votes from the participants. If all participants have voted to commit the transaction, then so will the coordinator. In that case, it sends a *global-commit* message to all participants. However, if one participant had voted to abort the transaction, the coordinator will also decide to abort the transaction and multicasts a *global-abort* message.
4. Each participant that voted for a commit waits for the final reaction by the coordinator. If a participant receives a *global-commit* message, it locally commits the transaction. Otherwise, when receiving a *global-abort* message, the transaction is locally aborted as well.



Coordinator



Participant

The first phase is the voting phase, and consists of steps 1 and 2. The second phase is the decision phase, and consists of steps 3 and 4.

There are a total of three states in which either a coordinator or participant is blocked waiting for an incoming message. First, a participant may be waiting in its *INIT* state for a *vote-request* message from the coordinator. If that message is not received after some time, the participant will simply decide to locally abort the transaction, and thus send a *vote-abort* message to the coordinator.

Likewise, the coordinator can be blocked in state *WAIT*, waiting for the votes of each participant. If not all votes have been collected after a certain period of time, the coordinator should vote for an abort as well, and subsequently send *global-abort* to all participants.

Finally, a participant can be blocked in state *READY*, waiting for the global vote as sent by the coordinator. If that message is not received within a given time, the participant cannot simply decide to abort the transaction.

A better solution is to let a participant P contact another participant Q to see if it can decide from Q's current state what it should do. For example, suppose that Q had reached state *COMMIT*. This is possible only if the coordinator had sent a *global-commit* message to Q just before crashing. Apparently, this message had not yet been sent to P. Consequently, P may now also decide to locally commit.
Likewise, if Q is in state ABORT, P can safely abort as well.

Now suppose that Q is still in state *INIT*. This situation can occur when the coordinator has sent *vote-request* to all participants, but this message has reached P but has not reached Q. In other words, the coordinator had crashed while multicasting *vote-request*. In this case, it is safe to abort the transaction; both P and Q can make a transition to state *ABORT*.

The most difficult situation occurs when Q is also in state *READY*, waiting for a response from the coordinator. In particular, if it turns out that all participants are in state *READY*, no decision can be taken. Consequently, the protocol blocks until the coordinator recovers. For this reason, 2PC is also referred to as a **blocking commit protocol**.

State of Q	**Action by** P
COMMIT	Make transition to COMMIT
ABORT	Make transition to ABORT
INIT	Make transition to ABORT
READY	Contact another participant

Figure 8.31: Actions taken by a participant P when residing in state READY and having contacted another participant Q.

16.2. Stable Storage

Many protocols like 2PC assume that the state of processes, or at least some part of it, survives the failure of the process. Usually, this means storing the state on disk. How do you ensure that the data survives disk failures?

The solution lies in Stable Storage. It uses two identical disks. Writing a block requires writing first in disk 1 and then in disk 2. Upon reading a block, three things can happen:

- If the checksum of disk 1 is valid, and the two blocks are different, copy block from disk 1 to disk 2;
- If the checksum of disk 1 is not valid, use the block on disk 2, if its checksum is valid;
- If the checksums of both disks are not valid, then data has been lost.

17. Election Algorithms

Many distributed algorithms require one process to act as coordinator, initiator, or otherwise perform some special role. We will look at algorithms for electing a coordinator (using this as a generic name for the special process).

If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process P has a unique identifier id(P). Furthermore, we also assume that every process knows the identifier of every other process.

What the processes do not know is which ones are currently up and which ones are currently down.

The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be.

17.1. Bully Algorithm

A well-known solution for electing a coordinator is the bully algorithm devised by Garcia-Molina.

When any process notices that the coordinator is no longer responding to requests, it initiates an election. A process, Pk , holds an election as follows:

1. Pk sends an ELECTION message to all processes with higher identifiers: Pk+1 , Pk+2 , . . . , PN−1.
2. If no one responds, Pk wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over and Pk 's job is done.

At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues.
When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one.
Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.

If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

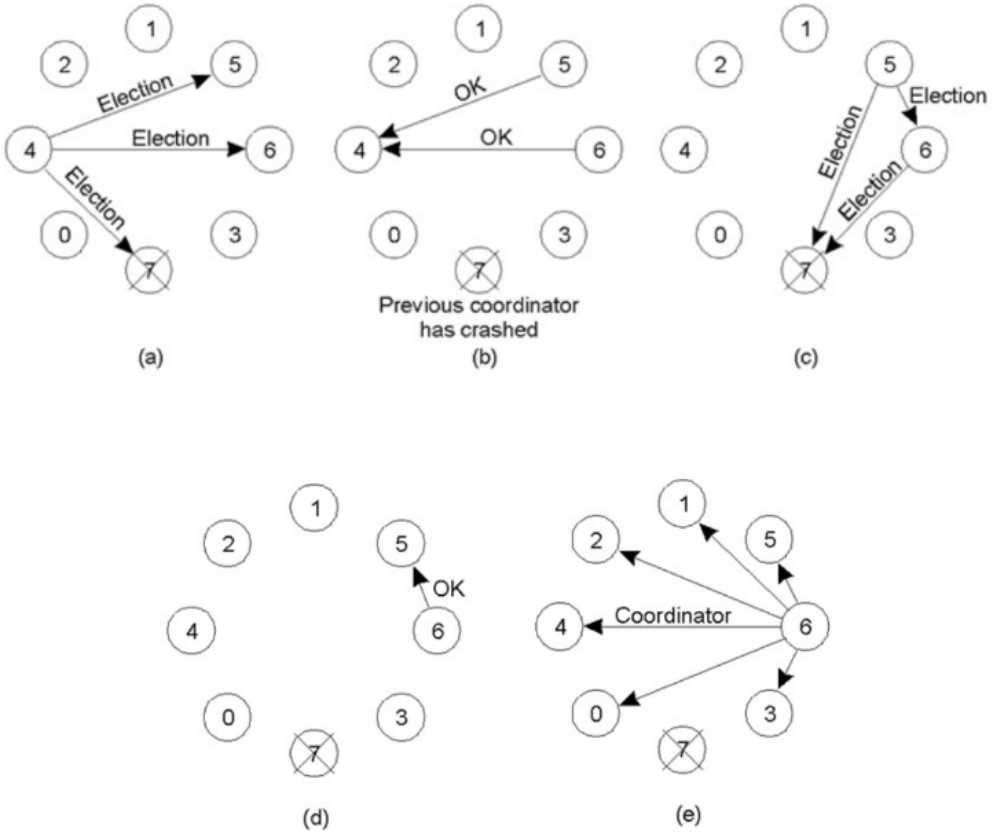


Figure 6.20: The bully election algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

17.2. Invitation Election Algorithm

This algorithm is based on two principles:

- Recovery: the processes, which detect that the coordinator of their group has failed, create a new group of one member (themselves), of which they are coordinators. It also happens when a process is incorporated or after failing, it is activated again.

- Large groups: each group coordinator asks the rest of the processes for coordinators to invite them to join his group and make it bigger.

We also assume that the algorithm meets a series of conditions to be able to apply it.

Compared to the bully algorithm, the invitation algorithm is thought to work in an asynchronous system in which communication failures and processes stops can happen.

Each process is responsible for performing the tasks for their group, making it more appropriate when service failures or partitions in the network may occur, causing communication failures between processes.
Therefore, we avoid leaving tasks unfinished and in each partition we will have different subgroups doing them until the network or service is restored.

Initially, each process has its own group in which it is itself the coordinator. These processes have an 'id' and a group identifier.

Periodically each coordinator will look for other coordinators to ask if they want to join their group through the invitation message.

To avoid multiple invitation requests and subsequent crashes the following mechanism is used:

> The higher the 'id', the higher the priority. The time to send his invitation request will be inversely proportional to that priority. That is, processes with more priority will send their invitation requests before those with lower priority.
> Doesn’t guarantee conflict erasure, but it does reduce their probability.

Once a coordinator receives an invitation message, the coordinator will accept it if it has a lower priority than the transmitter and will notify with a coordinator message to all of his group members, the change of group and they will accept with a confirmation message.

The coordinator of this new group will be the process that sent the invitation request. When all the processes are confirmed to be in the group, the coordinator sends a message with the 'id' of the new group to all members.

Every change in the group structure means that a new group 'id' is assigned each time.

This procedure will be repeated until the largest possible group is achieved or until another group is required for some other function.

In the event that there is some type of failure in any process, the procedure will be restarted, each process involved being the coordinator of itself.

The invitation algorithm analysis:

- Liveness: at the end of the algorithm the processes know their coordinator or they have aborted.

- Safety: there will always be a coordinator with a higher priority, either as a group or himself in each group.

- Performance: depending on the situation, in the worst case the lowest priority process that has been sent an invitation message will have to wait a long time until get a response if the rest that are above do not confirm or fail.