**Chapter 3: Processes**

Table of Contents

[Process Control Block 4](#_Toc83033076)

[Process Scheduling 6](#_Toc83033077)

[Queuing Diagram 6](#_Toc83033078)

[Schedulers 7](#_Toc83033079)

[Process Creation 9](#_Toc83033080)

[Resource Sharing 9](#_Toc83033081)

[Process Execution 10](#_Toc83033082)

[Address Spaces 10](#_Toc83033083)

[Process Creation Code 11](#_Toc83033084)

[Process Termination 12](#_Toc83033085)

[Cascading Termination 12](#_Toc83033086)

[Zombie Processes 13](#_Toc83033087)

[Orphan Processes 13](#_Toc83033088)

[Inter-Process Communication 14](#_Toc83033089)

[Shared Memory 15](#_Toc83033090)

[Message Passing 15](#_Toc83033091)

[Communication in Client-Server Systems 18](#_Toc83033092)

[Sockets 18](#_Toc83033093)

[Remote Procedure Calls 19](#_Toc83033094)

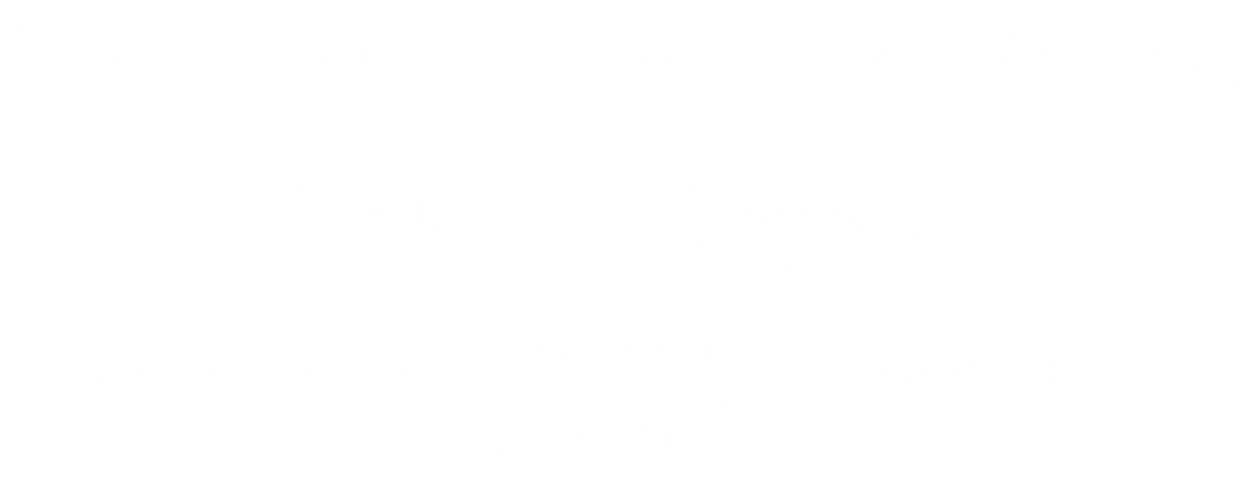
[Remote Method Invocation 22](#_Toc83033095)

A **program** is a **passive** entity, while a **process** is an **active** entity. Basically, a program that is currently being **executed** is a process.

A program may have **multiple processes**, for example if a browser has multiple tabs open, each tab could be considered a process.

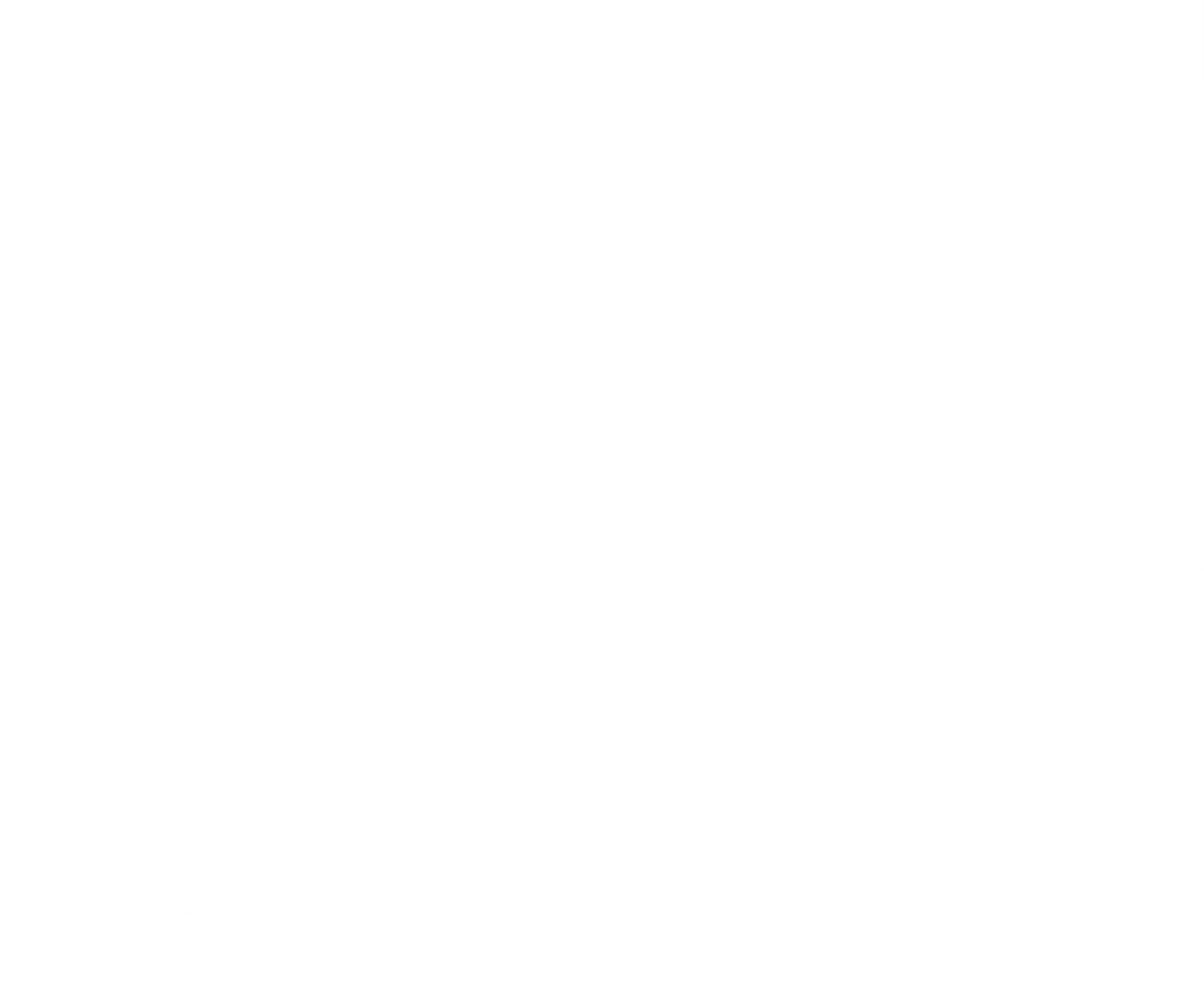
A process can have **five states**:

1. **New** State – A new process has just been **created**. The new process will immediately be sent into the **ready** state.
2. **Running** State – The process is currently **executing**. While executing, if an interrupt occurs, the process will be sent back to the **ready** state.
3. **Waiting** State – The process is waiting for some event to occur, such as some I/O operation. Once this event is complete, the process goes back to the **ready** state.
4. **Ready** State – The process is waiting to be assigned to the processor.
5. **Terminated** State – The process has finished execution.



## Process Control Block

When the CPU switches from one program to another, some information about the program being switched out of (also called the **state**, confusingly) must be stored. This information is stored in a **Process Control Block (PCB)**. When we return to the program, the information is restored from the PCB. Each program has its own PCB.



The process of changing from one program to another is called a **context switch**. The time during which the CPU is saving the PCB for one program and loading the PCB of another, it is not doing any ‘useful’ work, so this time is purely an overhead. We must try to minimize this time.

The PCB has the following parts:

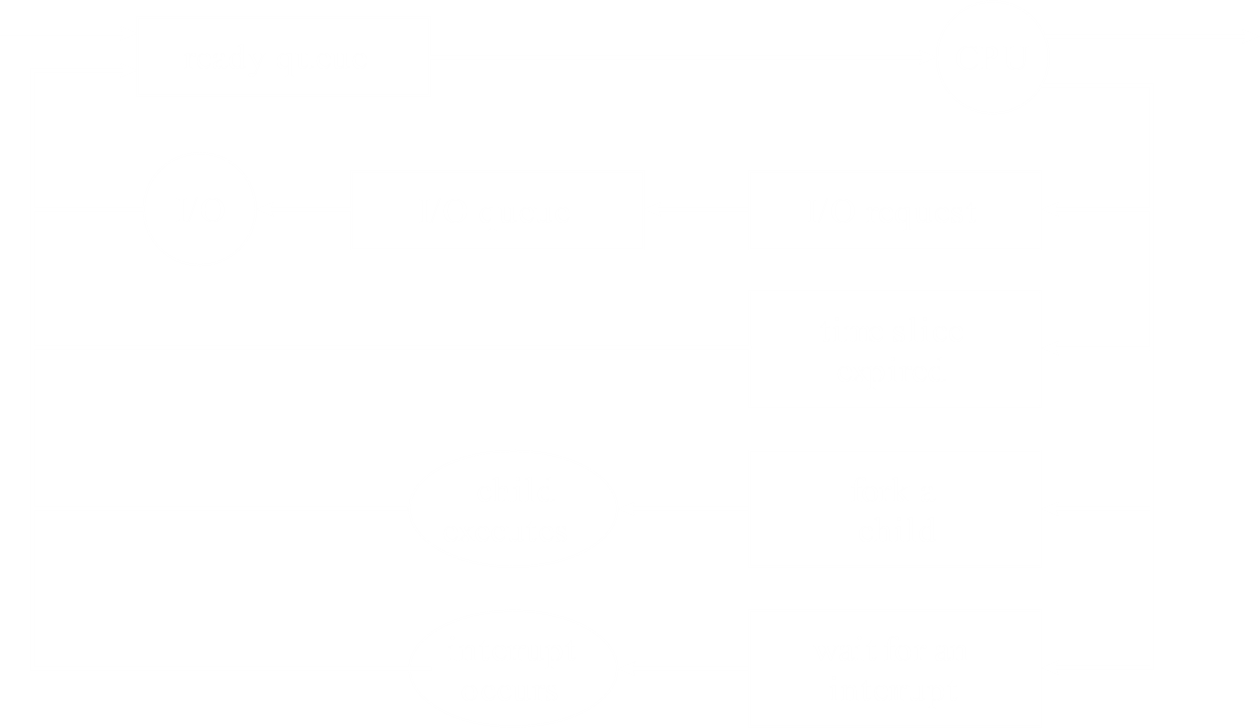
1. **Process State** – The state the process was in when it was stopped.
2. **Program Counter** – The address of the instruction that would have been executed next.
3. **CPU Registers** – Memory components from inside the CPU.
4. **CPU Scheduling Information** – The priority and position of the process in different scheduling queues.
5. **Memory Management Information** – These are memory locations associated with the process.
6. **Accounting Information** – This is data about the process, such as how long it has been running.
7. **I/O Status Information** – This is information related to the I/O devices the process is using.

## Process Scheduling

Processes are scheduled using different **queues**. For example, all processes that are in the **ready** state are kept in the **ready queue** while processes that are waiting for I/O operations to complete are kept in the **I/O Queue**.

Each of these queues have a common shape. There is a **head** pointer and a **tail** pointer. The **head** pointer points to the **first PCB** in the queue. The **tail** pointer points to the **last PCB** in the queue. Each PCB in turn points to the **next PCB** in the queue.

### Queuing Diagram



The **queuing diagram** above explains the way a process goes through the different queues.

1. The process enters the **ready queue** first, from where it is taken to the **CPU**.
2. If the process makes an **I/O Request**, it is put in the **I/O Queue** while the I/O request is dealt with. Once the I/O request is dealt with, the process goes back to the **ready** queue.
3. Each process is given a limited number of **processor cycles** while it is being executed. Once the provided number of cycles **expires**, it is said that the **time slice** has expired. The process goes back to the **ready queue**. This is done so that a single process does not take up all our resources.
4. If an **interrupt** occurs, the process goes back to the **ready queue**.
5. It is possible that a process creates a **child process** and waits for that child process to finish executing. The process is said to have **forked a child**. In this scenario, the main process goes back to the **ready queue** until the child process has finished executing.

### Schedulers

When a process is put into the **ready queue**, it is said to be **swapped in**. When it is brought to the **processor**, it is said to be **swapped out**. Which processes need to be swapped in and which need to be swapped out is managed by **schedulers**.

There are three types of schedulers:

1. Long-Term Schedulers
2. Short-Term Schedulers
3. Mid-Term Schedulers

The **Long-Term Scheduler** deals with the **job pool** and the **I/O Queue**, since I/O operations tend to take relatively more time. It decides which processes need to be brought in from the job-pool or the I/O queue to the **ready queue**. Since the job pool is in the **ROM** and I/O operations are slow, the long-term scheduler can afford to have **long-decision times**. Additionally, it is also executed much **less frequently** compared to the other schedulers.

The **Short-Term Scheduler** deals with the processes in the **ready queue**, which is in the **RAM**. As such, it needs to have **short-decision times**, since if it takes too long, it will make it pointless for the RAM to be so fast. It is executed **more-frequently**.

The **Mid-Term Scheduler** is a bit special and requires that we discuss **CPU-bound** and **I/O-bound** processes first.

A process which spends most of its time doing **calculations** is said to be **CPU-bound**. A process which spends most of its time doing **I/O Operations** is said to be **I/O-bound**. If we have a lot of **I/O-bound** processes, the **ready-queue** will be mostly empty, which means the **Short-Term Scheduler** has no work while the **Long-Term Scheduler** has a lot of work. On the flip side, if most of the processes are **CPU-bound**, the **Short-Term Scheduler** will have a lot of work while the **I/O Queue** remains mostly empty, meaning the **Long-Term Scheduler** has no work.

Both of these scenarios are undesirable. To deal with it, the **Mid-Term Scheduler** was introduced. The Mid-Term Scheduler is a mixture of the Short-Term Scheduler and the Long-Term Scheduler. It ensures that the Long-Term Scheduler and the Short-Term Scheduler are executed in a way such that the system remains **balanced**. It is responsible for dealing with **batch processes**, which are processes that are executed in **bulk**.

## Process Creation

A process may create several processes, say . The creating process, , is called the **parent** process, while the created processes are called **child** processes. Each of those processes could go on to create other processes.

Every process is given an identifier in the OS, called the **Process ID** (PID). This is used for various purposes, such as to access attributes related to the process.

### Resource Sharing

There are two ways in which **resources** are allocated by the OS when discussing parent-child processes:

1. Directly from the **CPU**. This means that the child process is just treated like a **normal process** and provided resources in the same manner than the parent process was provided.
2. From the **parent process**. This means that the resources that were provided to the parent process must be divided amongst the child processes by the parent process. This method prevents the parent process from creating too many child processes and **overloading** the system.

The second scenario, where resources from the parent process are divided amongst the child processes, could be handled in two ways:

1. The parent process might **share** resources. This means while one child process is using some resource, any other child process that requires that resource needs to wait for it to become free.
2. The parent process might **partition** resources. This means the available resources are simply divided amongst the child processes.

### Process Execution

In terms of parent-child processes, **process execution** could occur in two ways:

1. Parent and child processes execute **concurrently**.
2. The parent process **waits** for some or all of the child processes to terminate to continue executing.

### Address Spaces

**Address space** refers to where a particular process is executing. For example, a child process could be part of the **same program** as its parent process and use the same data. It is said to have a **duplicate** address space. Alternatively, it could have a **new program** to work with. In that case, the address space for the child process will be different.

### Process Creation Code

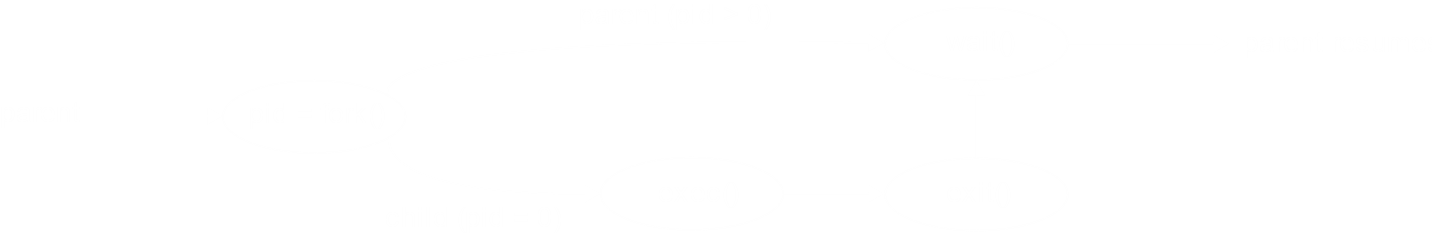
#include<sys/types.h>  
#include<stdio.h>  
#include<unistd.h>  
  
int main()  
{  
 pid\_t pid;  
 pid = fork(); *// creating a child process* if (pid < 0) *// could not create child process* {  
 fprintf(stderr, "Fork failed");  
 return 1;  
 }  
 *// parent and child processes continue execution* else if (pid == 0) execlp();  
 else wait(NULL); *// child process is executing; wait* return 0;  
}

C++

The first thing that was done was a **new process** was created using the fork() command. This command can return one of several values. If the child process could not be created for any reason, it returns a negative value. If the child process was created successfully, the child process is returned a value , while the parent process is returned the actual PID of the child process.

After the fork() command, both the parent and child processes **continue execution**. The rest of the code deals with that. If the child process looks at the code, it will find that its PID is and it will execute the else if block. If the parent process looks at the code, it will end up waiting for the child process to terminate.

The execlp() command that the child process executes actually starts a **new program** which takes away control from the current program. Control is only returned once the new program has terminated.



### Process Termination

Say a child process **terminates** via the exit() system call. Once this happens, the child process returns a **status value** to its parent process via the wait() system call. All resources that were allocated to the child process must now be **deallocated**.

Alternatively, the child process could be **forcefully terminated** via a **system call**. This system call is only invoked by the parent. A child process can never terminate a parent process. A child process could be terminated for a few reasons.

* 1. The child process has exceeded its allocated **resource usage**.
  2. The task assigned to the child process is **no longer required**.
  3. The parent process is **exiting**, and the OS will not allow the child process to continue without any parent process.

### Cascading Termination

If a **parent process** is exiting, all its **child processes** must be terminated. This process is called **cascading termination**.

### Zombie Processes

When a child process has finished execution, its resources are immediately **deallocated**. However, its entry from the **process table** is not removed until the parent process invokes the wait() system call. This is because the process table contains the **exit status** of the child process, which the parent process needs. In between the time when a child process has been terminated but has not been removed from the process table, it is called a **zombie process**.

### Orphan Processes

If, for some reason, the **parent process** itself **terminates** before invoking the wait() system call, the **child process** is called an **orphan process**. In some operating systems, like Linux or UNIX, a **new process** is assigned as the parent of the orphan process. In case of Linux or UNIX, this is the init process. If this is not done, the child process will remain in the process table forever and its exit status and PID will not be cleared.

All orphan processes were zombie processes recently but not all zombie processes will become orphan processes.

## Inter-Process Communication

Processes can be of two types, **independent processes** and **cooperating processes**. An independent process does not share data with any other processes. They do not affect other processes and cannot be affected by other processes. A cooperating process shares data with other processes. They can affect other processes and be affected by other processes. These processes could be on the same system or on different systems, communicating over a network.

There are several advantages to a process being a cooperating process:

1. **Information Sharing** – Cooperating processes are able to share information.
2. **Computational Speedup** – Processes that cooperate with each other are able to perform tasks faster than their independent counterparts.
3. **Modularity** – Allowing cooperation between processes allows us to break down tasks into different processes. For example, the main calculator program’s process could use other processes to perform specific mathematical operations.
4. **Convenience** – Cooperating processes make operations more convenient.

Inter-process communication (IPC) is handled in one of two ways: **Shared Memory** and **Message Passing**.

### Shared Memory

If **shared memory** is used, then one process creates a shared memory segment within its **address space**. Other processes can attach themselves to this address space in order to use the shared memory. The OS would normally prevent this, so all the processes involved must agree to it first. If they do, the processes will be responsible for determining the **form** and **location** of data and for ensuring they do not write to the same location simultaneously.

The region of memory in the address space that is being shared is a **buffer**. The buffer can be of two types, an **unbound buffer**, with no size limitations, or a **bound buffer**, with limited size.

The advantage of shared memory is its **speed**, which makes it convenient for **large amounts of data**. Sadly, for **distributed systems**, i.e. processes on different devices communicating over a network, using shared memory is not possible.

### Message Passing

**Message Passing** is ideal for distributed systems. Additionally, the OS provides the means for processes to communicate, which removes the burdens placed on the programmer by the shared memory method.

If two processes want to communicate, they must **send and receive messages** via a **communication link**. The messages themselves could be of a **fixed** or a **variable size**. Each feature of the communication link could be implemented in two ways:

* Direct or indirect communication
* Synchronous or asynchronous communication
* Automatic or explicit buffering

#### Direct and Indirect Communication

In **direct** communication, a process must provide the **name** of the process with which it wishes to communicate. This name must be provided by the sender with ever message being sent and by the receiver for every message they wish to receive.

This scheme exhibits **symmetry**, since both parties must identify the other. If only the sender where required to provide the name of the receiving process, the scheme would be **asymmetrical**. Regardless, the issue with this scheme is that if the process identifier changes, it has to be found again manually and all previous data must be updated with the new identifier.

In **indirect** communication, a **mailbox** is used. Any processes that wish to communicate must use a shared mailbox to which their messages will go. This allows **multiple processes** to be connected at the same time.

An issue arises with multiple processes connected to the same mailbox. Who will **receive** a message that has been sent if more than one process tries to receive it? This could be dealt with in one of three ways. Either allow only two processes to use the mailbox, or allow at most one process to execute the receive operation, or allow the OS to select a receiver randomly or based on some algorithm.

A mailbox could also be owned by a **process** itself, which means the mailbox resides in the process’s **address space**. In this case, there is no question of who the messages should be received by, since other processes would only be able to send messages. This is opposed to the situation where the mailbox is owned by the **OS**, which means the OS must allow processes to request a new mailbox, use it and delete it.

#### Synchronization

The sending and receiving processes could be **synchronous** or **asynchronous**.

For a **synchronous** scheme, the sending process is blocked until the message sent previously has been received. Similarly, the receiving process is blocked until some message has been sent.

For an **asynchronous** scheme, both processes can execute freely, even though the latter may not retrieve anything.

#### Buffering

The **temporary queue** in which messages will be received can be implemented in one of three ways:

1. **Zero-Capacity** – There is no queue. The sender must be blocked until the receiver receives the message.
2. **Bounded Capacity** – A limited number of messages can be sent at once. If the receiver does not receive messages and the limit has been exceeded, the sending operation is blocked.
3. **Unbounded Capacity** – The queue length is potentially infinite. The sender is never blocked.

The zero-capacity case is often called **no buffering** while the other two are called **automatic buffering**.

## Communication in Client-Server Systems

### Sockets

A **socket** is an end-point of communication. For processes communicating over a **network**, there needs to be 1 socket for every process.

Say we have a pair of processes that are communicating over a network. To be able to do this, we need the **IP addresses** and the **port numbers** on which the processes are running. A socket is the IP address concatenated with the port number in the format .

#### Working Procedure

1. The **client process** requests a connection. The host PC assigns a **port number** greater than 1024. Port numbers below that are reserved.
2. A **web socket** is created using the **IP address** and the **port number**. For example, if the host has an IP address and the port number assigned is , then the web socket will be .
3. The **web server** the client process wants to connect to also has an IP address and a port number. The connection is made between these two sockets. Packets are delivered from one IP address to the other, and the machines on either end decide which process the packets need to go to based on the port numbers.
4. If another client process wishes to create a connection with the same web server, it must be assigned a different port number. All connections must be **unique**.

#### Advantages and Disadvantages

Although sockets are **common and efficient**, it is still considered a **low-level** form of communication. This is because it does not have any data structure. It is said to transfer an **unstructured stream** of data. This means the client and server need to manually define a data structure in order to communicate.

### Remote Procedure Calls

A **Remote Procedure Call** (RPC) is used when one machine (the client) wishes to **execute a function** on another machine (the server). This requires a **daemon** to be running on a specific port that supports the required RPC. For example, if the server wished to allow client devices to view a list of all its current users, then the server would have a daemon that supports such an RPC attached to a specific port. Clients could then connect to that port and run the function that retrieves the list for them. This does not require any special coding. The actual function is written as though it were running locally.

On both the client and the server, there is a piece of code called a **stub**. When the client calls an RPC, the stub on the client side is responsible for packaging some information required to make the call into a transmittable form. This information includes things like the unique identifier for the function, the parameters for the function and the relevant IP addresses and port numbers. The packaging process is called **marshalling**. On the server’s end, the server’s stub **unmarshalls** the package to retrieve the information.

#### Issues

The first and foremost issue is **data representation**. It is entirely possible that the client and the server use different representations for the data they are transmitting. One could use the big-endian notation while the other uses little-endian notation.

To deal with this, the **stub** converts the data on the client side to a machine-independent data representation format called the **External Data Representation** format (XDR). On the server’s end, the stub converts the data from XDR to the relevant data representation format for the machine.

The second issue is the **semantics** of the call. Local procedure calls are very unlikely to fail, but RPC calls can fail, be duplicated or be executed repeatedly due to common network errors (think about dropped ACKs).

This issue can be dealt with by ensuring that each message is acted upon **exactly once** instead of **at most once**.

For the **at most once** semantic, each message has a **timestamp** attached to it. In this way, if the exact same message is received again by the server, it will see that the timestamp is repeated and will ignore the message.

With the at most once semantic, we still run the risk that the RPC call was received by the server but that it failed to execute the call. In order to deal with this issue, we use the **exactly once** semantic, where RPC calls are given all the functionality of at most once and on top of that, an **ACK** is sent to the client once the RPC is executed successfully. Until the ACK is received, the client will repeatedly send the RPC message.

A third issue is **communication** between the server and the client. We mentioned that the client needs to connect to the RPC daemon on the server, but how would it know what the port number for the daemon is? The two systems do not share memory, so they cannot know information about each other beforehand.

There are two approaches to solving this. The first approach is that the information is **predetermined**, i.e. a fixed port address is used. The port number is assigned during program compilation and cannot be changed by the server. The second approach is to use a **rendezvous** daemon, provided by the OS. The client connects to this rendezvous daemon, which itself has a fixed port address, and requests the port address of the RPC daemon it needs by mentioning the name of the RPC. The second approach has more overhead, but it is more flexible.

### Remote Method Invocation

**Remote Method Invocation** (RMI) is similar to RPC, but with one major difference. It is a **Java** feature that allows us to invoke remote **objects**. The object might be on the same machine, or on a different host. Java uses the Java Virtual Machine (JVM), so it is possible that RMI is being used for an object that is on the same physical machine, but on a different JVM.

There are two major differences between RPCs and RMIs:

1. RPC allows only **function calls**, following the **procedural programming** approach. RMI is **object-oriented**, meaning we can invoke **methods** of **remote objects**.
2. For RPC, the **parameters** are ordinary **data structures**. For RMI, the parameters can be **objects** as well.

#### Stubs and Skeletons

RMI implements remote objects using two things, **stubs** and **skeletons**.

The remote server provides an **interface**, which is what tells the client what methods of the remote object it can use. The actual implementation for these methods is on the remote server. The interface is part of the **skeleton**.

When a particular method is called, the **stub** marshalsa **parcel**, which includes the name of the **method** to be called and the **parameters** to pass. This parcel is sent to the remote server.

On the remote server, the **skeleton** is what receives the parcel. It unmarshalls it and **invokes the method** required using the parameters provided. The method being called may also **return** some data. This data is marshalled by the skeleton into another **parcel**, which is sent back to the client.

Back on the client end, the **stub** unmarshalls the returned packet, extracts the data and passes it on to the client.

Say a client invokes the following remote method:

bool returnValue = someObject.someMethod(firstArgument, secondArgument);

C++

On the server, we have this remote object that has the method defined:

bool someMethod(firstParameter, secondParameter)  
{  
 *// some code* return someBooleanVariable;  
}

C++

On the client side, the stub creates a parcel for the skeleton on the server end. The skeleton invokes the method. The method returns some value to the skeleton. The skeleton sends the Boolean value to the stub on the client side.

#### Behaviour of Parameter Passing

If the object we are passing as an argument is a **local variable**, then we need to send a **copy** of the object, since the remote server does not have that object. If the object is also on the remote server, then we can **pass by reference**.

If we are passing an object by copying it, we need to **serialize** it. This process is called **object serialization**. This uses the java.io.serializable interface. The interface will contain the object’s **declarations**. The serialization process encodes the objects into a **byte stream** which is then sent over to the remote server by the stub. The remote object uses the same interface to decode the byte stream.