What is RPC (Remote Procedure Call)?

Imagine you have a function in your computer program that you want to run. Normally, this function would be in the same program, and you can easily call it. However, in some cases, you might want to call a function that is not on your computer—it might be on another computer or in a different program. **RPC** lets you do this.

So, with RPC, you can call a function on another machine or process as if it were on your own machine.

How Does It Work?

- The key idea is that **RPC allows a program to request a service from a different program** located on another computer in a network without needing to know the details of the network.
- It's similar to a normal function call, but instead of the function running on your computer, it runs somewhere else.

Steps in RPC (Simplified):

1. **Caller (Client) makes a request**: You ask the function to run and give it some input (called arguments).
2. **Request goes to the server**: This request gets sent to the server (which could be on another machine).
3. **Server processes the request**: The server runs the function using the inputs you gave it.
4. **Server sends back the result**: Once the function is done, the result gets sent back to the client (you).
5. **Caller receives the result**: You get the result of the function as if it had been run locally.

Why is RPC Useful?

- **Simplicity**: RPCs make calling a function on another machine feel just like calling a local function.
- **Efficient**: It's more efficient than manually handling all the network communication yourself.
- **General**: RPC works both for calling functions on the same machine or across different machines.

Differences Between RPC and Local Function Calls:

- In a **local function call**, the function runs in the same memory space as the program calling it.
- In **RPC**, the function runs in a **different memory space**, possibly on another computer, which means:
    - There is **no shared memory** between the caller and the called function.
    - You can't use pointers or references because the two programs have different memory spaces.
    - Instead of directly sharing data, the caller and the function exchange data through **messages**.

Transparency in RPC

- **Syntactic Transparency**: The syntax (how you write the call) of an RPC should look the same as a local function call. This makes it easy for programmers to use.
- **Semantic Transparency**: The behavior of the function should also be the same, but this is more difficult to achieve because network delays or failures can affect remote calls.

Asynchronous RPC

In **normal RPC**, the caller waits until the server finishes the function and sends back the result. This can block the caller from doing anything else.

However, in **asynchronous RPC**, the caller can continue doing other work while waiting for the server's reply, which is useful for improving performance.

Summary:

- **RPC** allows a program to call a function on another machine or in another process, making it feel like a local function call.
- It's efficient and simplifies communication between different machines or processes.
- However, since there's no shared memory, data has to be sent back and forth in messages.
- Asynchronous RPC allows the caller to continue working while waiting for the server's response.

Vulnerability of RPC

1. **RPCs are more vulnerable to failures** compared to local function calls because:
    - There can be **processor crashes** (the computer running the function might crash).
    - There could be **network communication problems** (the connection between the client and server may fail).
2. **RPCs take more time** than local function calls because they involve sending messages over a network. This makes **total semantic transparency** (making RPCs work exactly like local calls) impossible.

Implementing RPC (How RPC Works Behind the Scenes)

To make RPC work as smoothly as possible (trying to make it look like a local function call), the **RPC mechanism** uses something called **stubs**.

What are Stubs?

- **Stubs** are small programs that hide all the complicated details of making remote calls.
- They make it **look like you are calling a local function**, even though it's happening on another machine.

- Each side (the client and server) has its own **stub** to handle the process.

Five Key Parts of RPC:

1. **Client**: This is the program or process that is making the request (the one calling the function).
2. **Client Stub**: A small program on the client's side that helps package and send the request to the server.
3. **RPC Runtime**: A communication system that handles sending and receiving messages between the client and server across the network.
4. **Server Stub**: A small program on the server's side that receives the request, unpacks it, and passes it to the actual function.
5. **Server**: This is where the function or procedure being called is actually run.

How It All Works Together:

- The **client**, **client stub**, and an instance of the **RPC Runtime** work on the client's machine.
- The **server**, **server stub**, and another instance of the **RPC Runtime** work on the server's machine.
- The **client** doesn't know all these details; it just thinks it's calling a normal local function.

Client Stub (What It Does):

- When the **client** wants to make a call, the **client stub** does the following:
    1. **Packs** the function name and input arguments into a message (like putting all the information into a package).
    2. **Sends** this package to the **server** using the **RPC Runtime**.
- After the server processes the request, the **client stub**:
    1. **Unpacks** the result from the server.
    2. **Passes** the result back to the client.

RPC Runtime:

- The **RPC Runtime** manages the **communication between the client and server**. It handles:
    - **Retransmission**: If a message doesn't go through, it tries to send it again.
    - **Acknowledgment**: Making sure both sides know the message was received.
    - **Routing**: Figuring out the best path for sending the message through the network.
    - **Encryption**: Ensuring that messages are secure.

Server Stub (What It Does):

- On the **server side**, the **server stub** does the following:
    1. **Unpacks** the request from the client.
    2. **Calls the function** on the server (like a normal function call).
    3. **Packs** the result of the function back into a message and sends it back to the client via the **RPC Runtime**.

Stub Generation (Creating Stubs):

Stubs can be created in two ways:

1. **Manually**: Programmers write the stub code themselves.
2. **Automatically**: Tools can generate stubs automatically, making it easier for the programmer.

Summary:

- **RPCs are more vulnerable to failure** due to network and processor issues.
- **RPCs take more time** than local calls because they involve network communication.
- **Stubs** help make RPCs look like local function calls by hiding the network details.
- **Client Stub** prepares and sends the request, while the **Server Stub** receives and processes it.
- The **RPC Runtime** handles the network communication between the client and server.

By using stubs and RPC Runtime, the whole process of calling functions on remote servers becomes simpler and more transparent to the programmer.

Automatic Stub Generation

1. **Interface Definition Language (IDL)**:
    - IDL is a language used to define the communication between the **client** and the **server**.
    - It helps describe **what functions the server offers** and what kind of **input (arguments)** and **output (results)** these functions need.
2. **Interface Definition**:
    - This is simply a **list of function names** that the server supports, along with the **types of arguments** and **results** for each function.
    - It helps in **reducing data storage** and **controlling the amount of data transferred** over the network.
    - It includes things like **type definitions**, **enumerated types** (lists of possible values), and **constants** (fixed values).
3. **Exporting the Interface**:

- The **server program** implements the functions defined in the interface.

4. **Importing the Interface**:
    - The **client program** uses the functions from the interface to make requests to the server.
5. **IDL Compiler**:
    - The **IDL compiler** takes the interface definition and generates:
        - Components that can be combined with the client and server programs.
        - The **client stub** and **server stub**.
        - The necessary **marshaling (packing) and unmarshaling (unpacking)** operations.
        - A **header file** that defines the data types used.

RPC Messages

1. **Transport Protocol Independence**:
    - The **RPC system** does not care how messages travel across the network. It just ensures they get from the client to the server and back.
2. **Types of Messages in RPC**:
    - **Call Messages**: These are sent from the client to the server to request a function to be executed.
    - **Reply Messages**: These are sent from the server to the client with the result of the executed function.

Call Messages (Sent by the Client)

A **call message** contains:

1. **Procedure Information**: Identifies the function the client wants to run on the server.
2. **Arguments**: The input values needed to run the procedure.
3. **Message Identification**: A unique sequence number to keep track of messages and avoid duplicates or lost messages.
4. **Message Type**: Specifies whether the message is a **call** or a **reply**.
5. **Client Identification**: Helps the server recognize which client sent the message and also helps in **authenticating** the client.

Reply Messages (Sent by the Server)

The **reply message** sends the result back to the client. It can be:

- **Successful**: The server completed the procedure and sends back the result.
- **Unsuccessful**: The server couldn't process the request due to:
    1. The message wasn't understandable.
    2. The client isn't authorized to use the service.
    3. The procedure identifier is missing.
    4. The arguments are incorrect.
    5. An error (exception) occurred.

The reply message includes:

- **Message identifier**: To match the reply with the original request.
- **Message type**: Indicates that it's a reply.
- **Reply status**: Whether the request was successful or unsuccessful.
- **Result** (if successful) or **Reason for failure** (if unsuccessful).

Marshalling Arguments and Results

Marshalling is the process of **packing data** (arguments or results) into a format that can be sent over the network.

- **Unmarshalling** is the process of **unpacking the data** when it arrives on the other side (client or server).
- Marshalling is necessary because computers use different ways to store data, so **encoding and decoding** ensure both sides understand the data.

Two Types of Marshalling Procedures:

1. **Built-in RPC Marshalling**: These are provided by the RPC system itself to handle common data types.
2. **User-defined Marshalling**: In some cases, users define their own methods for marshaling complex or custom data types.

Summary:

- **IDL** defines the functions and data types used by the client and server, making communication clear.

- **Messages** are exchanged between the client and server to make requests and send results.
- **Marshalling** ensures that the data sent over the network is properly encoded and decoded.

Server Management

Issues in Server Management:

1. **Server Implementation**: How to set up the server to handle client requests.
2. **Server Creation**: How and when to create the server.

Server Implementation

Types of Servers:

1. **Stateful Servers**:
    - These servers remember information about the client across multiple requests (RPCs).
    - They make programming easier and are generally more efficient.

Example: In a stateful file server, when a client opens a file, the server remembers the file's identifier (fid) and the read/write position for future operations.

2. **Stateless Servers**:
    - These servers don't store any information between client requests.
    - Every request must include all necessary information.
    - Advantage: If a failure occurs, the stateless server can recover easily.

Example: In a stateless file server, each request contains the file name and the read/write position.

Server Creation Semantics

- **Servers operate independently from clients**, meaning they don't rely on a specific client process to exist.

Types of Server Lifespan:

1. **Instance-per-call Servers**:
    - These servers exist only for the duration of a single client request.
    - They are created when a request arrives and deleted after handling the request.
    - They are stateless but can be expensive if many similar requests occur.

2. **Instance-per-session Servers**:
    - These servers exist for an entire session between the client and server, allowing them to store information (state) between multiple requests during that session.
3. **Persistent Servers**:
    - These servers stay active indefinitely.
    - They are the most commonly used and improve performance since many clients can use them at once.

Parameter Passing Semantics

Two Common Approaches:

1. **Call-by-Value**:
    - The parameters are copied and sent in a message from the client to the server.
    - It can be slow when dealing with large data (like arrays or trees).
2. **Call-by-Reference**:
    - Instead of copying data, the client sends a reference (or address) of the data.
    - This is useful when client and server share a memory space.

Special Case: Call-by-Object-Reference

- A type of call-by-reference where the parameter is an object, and only its reference is passed.

Call-by-Move

- Similar to call-by-reference, but the parameter object can be moved from the client to the server, reducing network traffic.

Call Semantics

Call Failures:

- Failures can occur because of:
    - Call message not arriving
    - Reply not received
    - Caller or server crash
    - Network issues

Types of Call Semantics:

1. **Possibly or May-Be Call Semantics**:
    - The server may or may not reply to the request.
    - Used for tasks that don't require confirmation, like periodic updates.
2. **Last-One Call Semantics**:
    - The client keeps resending the request until it gets a reply.
    - If the server crashes during a nested call, it might leave orphan calls.
3. **Last-of-Many Call Semantics**:
    - Similar to Last-One, but orphan calls are ignored.
    - The client uses a unique identifier to make sure it processes the latest response.
4. **At-Least-Once Call Semantics**:
    - Guarantees that the call is executed at least once.
    - If multiple responses arrive, the client uses the first one and ignores the rest.
5. **Exactly-Once Call Semantics**:
    - Ensures that a procedure is executed only once, no matter how many times the client retries.
    - Requires the client to acknowledge the server's reply to confirm that it has received the result.

Communication Protocols for RPCs (Remote Procedure Calls)

1. **Request Protocol (R Protocol):**
    - The client sends a request message to the server.
    - The server processes the request and executes the procedure.
    - The client moves to the next request without waiting for a reply.
    - This is a simple one-way communication where the server does not send back any response.
2. **Request/Reply Protocol (RR Protocol):**
    - The client sends a request to the server.
    - The server executes the requested procedure and sends back a reply message.
    - The client uses this reply as an acknowledgment that the request was processed successfully.

    - Every request from the client gets a response from the server.
3. **Request/Reply/Acknowledge-Reply Protocol (RRA Protocol):**
    - The client sends a request message to the server.
    - The server processes the request and sends back a reply message.
    - The client then sends an acknowledgment to confirm that it received the reply.
    - This protocol ensures more reliable communication by adding a final confirmation from the client.

Complicated RPCs

RPCs can get more complex in certain scenarios, especially in cases involving:

- **Long-duration calls** (when the procedure takes a long time to execute).
- **Large messages** (when the data being sent is too big for one message).

Handling Long-Duration RPC Calls:

There are two ways to handle long-duration calls or large gaps between RPCs:

1. **Periodic Probing by the Client:**
    - After the client sends a request, it regularly sends "probe" messages to check if the server is still working.
    - The server acknowledges these probes to let the client know it's still active.
2. **Periodic Acknowledgment by the Server:**
    - If the server knows it will take a long time to respond, it sends periodic acknowledgment messages to inform the client that it's still working on the request.

Handling Long Messages in RPC:

- **Multidatagram Messages:**
    - If the message is too large for one packet, it is split into smaller parts (datagrams).
    - After sending all parts, a single acknowledgment is sent for the whole message.
- **Breaking Large RPCs into Smaller Calls:**
    - In some systems, like SUN Microsystems, a large RPC can be split into smaller chunks of data (e.g., limited to 8 kilobytes).

Client-Server Binding:

- **Binding:**
    - Before an RPC can take place, the client must know where the server is located. This process of linking the client with the server is called **binding**.
- **How Binding Works:**

1. **Server Registration:** Servers register themselves, announcing that they are available to handle client requests.
2. **Client Importing:** The client looks for the server's location information and binds to it.

Server Naming:

- **Naming:**
    - Clients specify the server they want to connect with using a unique **interface name**.
    - The name is made up of two parts:
        1. **Type:** The version of the server interface.
        2. **Instance:** The specific server offering the service.

Locating a Server:

1. **Broadcasting:**
    - The client sends a message to all nodes (computers) in the network.
    - The server responds if it's available. This is good for small networks.
2. **Binding Agent:**
    - A **binding agent** works like a name server. It helps the client locate the server.
    - The binding agent maintains a table that maps server names to their locations.

Binding Agent Process:

1. The server registers itself with the binding agent.
2. The client asks the binding agent for the server's location.
3. The binding agent sends the server's location to the client.

4. The client connects to the server and makes the RPC call.

Advantages and Disadvantages of Binding Agents:

- **Advantages:**
    - Multiple servers can handle the same type of request, allowing for load balancing (spreading out the work).
- **Disadvantages:**
    - If there are many short-lived client processes, the binding overhead can become significant.
    - The binding agent itself must be reliable and not a bottleneck (slowing down the system).

Solution:

- Use multiple binding agents and replicate (copy) the information across them to ensure smooth operation and avoid failures.

Binding Time

Binding time refers to when a client gets connected to a server in the context of RPC (Remote Procedure Call). There are different binding times:

1. **Binding at Compile Time**:
   The server's network address is hard-coded into the client program during the compilation stage. This is static binding and cannot change after the client is compiled.
2. **Binding at Link Time**:
   In this method, the server's location (called a "handle") is cached by the client. The client makes an import request to a binding agent to retrieve the server's information before making an RPC. This method is efficient when the client makes repeated calls to the same server because the server's handle is stored, avoiding frequent binding agent contacts.
3. **Binding at Call Time (Indirect Call)**:
   In this method, the client contacts the binding agent each time it makes an RPC. The binding agent locates the server, sends the call to the server, and returns the result to the client. This is more dynamic and flexible but slightly slower due to additional steps.

Security Issues

When performing RPC, security becomes critical. Some of the issues include:

1. **Server Authentication by the Client**:
   Should the client verify that it is communicating with the correct server to prevent fraud?

2. **Client Authentication by the Server**:
   Should the server authenticate the client to ensure that it only provides results to an authorized user?

3. **Data Accessibility**:
   Should the RPC arguments and results be protected to prevent unauthorized access during transmission?

Special Types of RPCs

There are various special types of RPCs to handle specific scenarios:

1. **Callback RPC**:
   A client can also act as a server in this model. The server can "call back" the client with additional information. This is useful for peer-to-peer interactions but involves some complexity in managing client-server roles and deadlocks.

2. **Broadcast RPC**:
   In broadcast RPC, a client sends a request to all servers on the network that can process the request. It's helpful in discovering services but can increase network traffic.

3. **Batch-mode RPC**:
   This method queues multiple requests on the client side and sends them in one batch to the server. It’s useful when there are multiple requests but no need for immediate replies. It reduces network overhead and is efficient for low-frequency calls.

Lightweight RPC (LRPC)

LRPC is designed for communication within the same machine (cross-domain) instead of between machines (cross-machine). It offers optimizations for better performance when communicating across different domains on the same machine.

1. **Simple Control Transfer**:
   LRPC uses "handoff scheduling" to directly switch the thread from the client domain to the server domain, saving time on context switches.

2. **Simple Data Transfer**:
   Traditional RPC involves four data copy operations, but LRPC reduces it to one by using a **shared-argument stack**, which both client and server can access. This reduces overhead and increases performance.

3. **Simple Stub**:
   LRPC stubs are minimal in complexity. Instead of involving multiple protocol layers, LRPC directly invokes the necessary procedures, reducing delays.

Optimizations for Better Performance in RPCs

Some ways to improve the performance of RPCs:

1. **Concurrent Access to Multiple Servers**:
   Allowing the client to communicate with multiple servers simultaneously.
2. **Serving Multiple Requests Simultaneously**:
   A server should be capable of handling several RPC requests at the same time to improve efficiency.
3. **Reducing Per-call Workload**:
   Minimizing the amount of work the server needs to do for each individual call.
4. **Reply Caching**:
   For idempotent operations (ones that can be repeated without side effects), caching the reply and reusing it can reduce processing time.
5. **Proper Timeout Values**:
   Setting appropriate timeout values for RPCs can reduce unnecessary delays and retries.
6. **Efficient RPC Protocol Design**:
   RPC protocols should be simple and only include essential information. For example, only 3 out of 13 fields in the IP header are typically needed for RPC, which reduces overhead.