Distributed Systems

Christian J. Rudder

January 2025

Contents

This page is left intentionally blank.

Big thanks to **Professor Ioannis Liagouris**
for teaching CS351: Distributed Systems
at Boston University [1].

All illustration contain original assets.

Disclaimer: These notes are my personal understanding and interpretation of the course material. They are not officially endorsed by the instructor or the university. Please use them as a supplementary resource and refer to the official course materials for accurate information.

Prerequisites

— 1 —

Introduction

1.1 Remote Procedure Call (RPC)

This section will cover the concept of Remote Procedure Calls (RPCs) and how they are used in distributed systems. We will discuss the basic idea of RPCs, how they work, and the challenges they face. We'll be using the GO RPC library to demonstrate their use with examples.

Definition 1.1: client-server model

The client-server model is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, **called servers**, and service requesters, **called clients**.

Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system.

Definition 1.2: Remote Procedure Call (RPC)

A Remote Procedure Call (RPC) is a protocol that allows a **client** computer request the execution of functions on a separate **server** computer.

RPC's abstract the network communication between the client and server enabling developers to write programs that may run on different machines, but appear to run locally.

Now to define the components of an RPC call:

Definition 1.3: RPC Call Stack

The RPC call stack facilitates communication between two systems via four layers:

1. **Application Layer:** The highest layer where the client application initiates a function call. On the server side, this layer corresponds to the service handling the request.
2. **Stub:** A client-side stub acts as a proxy for the remote function, **marshaling arguments** (converting them into a transmittable format) and forwarding them to the RPC library. On the server side, a corresponding stub, **the dispatcher**, receives the request, unmarshals the data, and passes it to the actual function.
3. **RPC Library:** The RPC runtime that manages communication between the client and server, ensuring request formatting, serialization, and deserialization.
4. **OS & Networking Layer:** The lowest layer, responsible for transmitting RPC request and response messages over the network using underlying transport protocols.

The request message travels from the client's application layer down through the stack and across the network to the server. The server processes the request in reverse, executing the function and returning the result to the client.

To illustrate the RPC call stack, observe the following diagram:

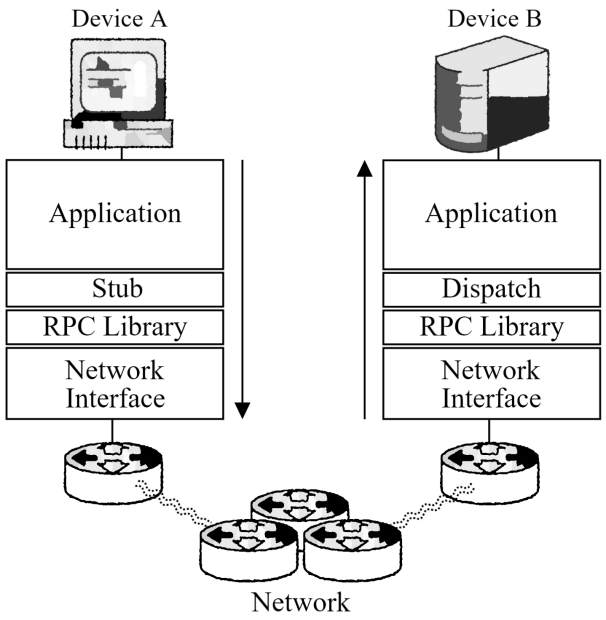

Figure 1.1: Client system A making a request to Server system B over RPC.

In terms of time it might look like:

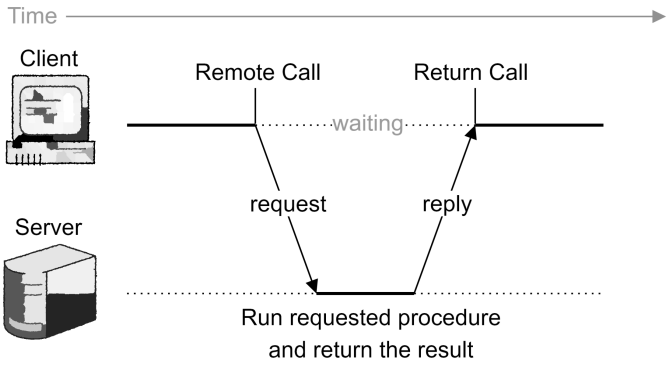

Figure 1.2: RPC call stack over time.

Once the client makes the call it waits for the server to process the request and return the result. The programmer need not worry beyond sending the request and receiving the response. The RPC deals with all the heavy work of facilitating the communication.

Now to discuss what marshaling and unmarshaling are:

Definition 1.4: Marshaling and Unmarshaling

Marshaling handles data format conversions, converting the object into a byte stream (binary data). This conversion is known as **serialization**. This is done as the network can only transmit bytes

Unmarshaling is the process of converting the byte stream into the original object called **deserialization**. This allows the server to process the request.

To illustrate serialization and deserialization, consider the following diagram:

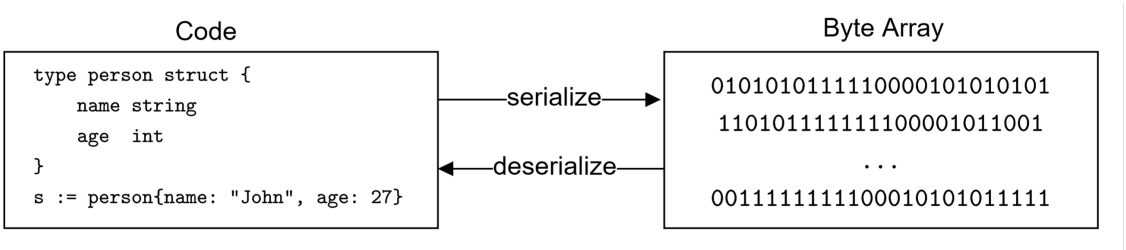

Figure 1.3: Serialization and Deserialization of data.

There is one cardinal rule to remember when dealing with RPCs:

Theorem 1.1: Network Reliability

The network is always unreliable.

That is to say, the network can drop packets, delay messages, or deliver them out of order. Anything that can go wrong will go wrong.

To handle network unreliability, we'll first consider two failure models:

Definition 1.5: At-least-once & At-most-once

- **At-least-once:** Regardless of failures, make the RPC call until the server responds. Works for read-only operations, otherwise, a strategy to handle duplicate requests is needed.
- **At-most-once:** Ensure the RPC call is made only once, even if the server fails to respond. This is done by having a unique identifier for each request. Each subsequent request tells the server which calls have already been processed.

For our communication to work *reliably* we need At-least-once and At-most-once with unlimited tries coupled by a fault-tolerant implementation. This brings us to the **GO RGC library**.

Definition 1.6: Go RPC Library

The Go RPC library provides a simple way to implement RPCs in the programming language Go. This gives us:

- At-most-once model with respect to a single client-server
- Built on top of single **TCP connection** (Transport Layer Protocol). This protocol ensures reliable communication between client and server.
- Returns error if reply is not received, e.g., connection broken (TCP timeout)

Now to discuss briefly how a basic TCP connection is made:

Definition 1.7: Establishing a TCP Connection (SYN ACK)

First a three-way handshake is a method used in a TCP/IP network to create a connection between a local host/client and server. It is a three-step method that requires both the client and server to exchange **SYN (synchronize)** and **ACK (acknowledgment)**.

1. The client sends a SYN packet to the server requesting to synchronize sequence numbers.
2. The server responds with a SYN-ACK packet, acknowledging the request and sending its own SYN request.
3. The client responds with an ACK packet, acknowledging the server's SYN request.

After the three-way handshake, the connection is established and the client and server can communicate exchanging SYN and ACK data-packets. To end the connection another three-way handshake takes placed, where instead of SYN, **FIN (finish)** is used.

Given this implementation, we approach somewhere in the realm of an **Exactly-Once model**:

Definition 1.8: Exactly-Once Model

The Exactly-Once model guarantees that a message is delivered exactly once to the recipient. Meaning, messages aren't duplicated, lost, or delivered out of order. However, In practice, data packets might do all of the above. Though with the right protocols in place, we can ensure order of logic is preserved.

Below we illustrate a simple TCP connection:

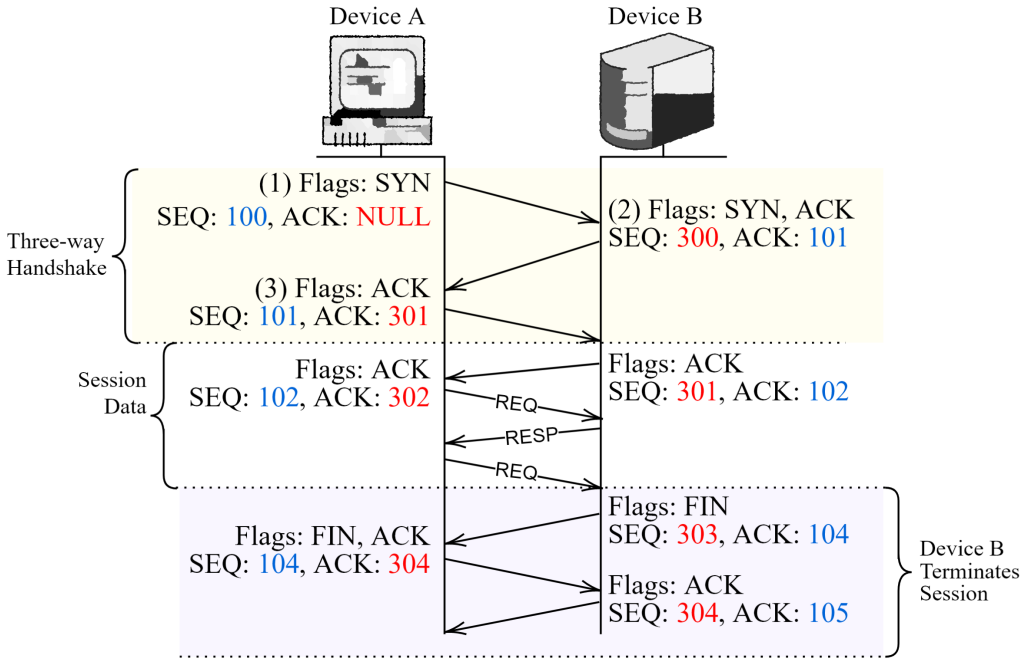

Figure 1.4: TCP Handshake, data transfer, and session termination.

Here the client (Device A) begins a three-way handshake with the server (Device B) to establish a connection. Both start with arbitrary sequence numbers for security purposes. With each packet received the two devices increment their sequence numbers accordingly.

Tip: If there still resides curiosity for the networking aspect of RPCs, consider reading our other notes: https://github.com/Concise-Works/Cyber-Security/blob/main/main.pdf

Bibliography

[1] Ioannis Liagouris. Cs351: Distributed systems. Lecture notes, Boston University, Spring Semester, 2025. Boston University, CS Department.