DSC204A: Scaleable Data Systems Winter 2025

19: Networking Fundamentals Continued

Instructor: Umesh Bellur *Scribe: Nikhil C. Paleti, Shankara N. V. Raju, Raghav Kachroo*

The Network Stack

From a networking perspective, we also have a layered stack analogous to a single-machine software stack:

- **Networking Hardware**: NICs, routers, switches—moves raw bits.
- **Network OS (NOS)**: Configures devices (firewalls, routing policies); programmers rarely see it directly.
- **Protocols**: Define how to form "messages" from bits, enforce ordering or not, detect/recover errors, and manage connection setup/teardown (e.g., IP, TCP, UDP, Ethernet).
- **OS Networking APIs**: Sockets/RPC libraries—provide a function-call view over the underlying protocols.
- **Distributed-System Layer**: Higher-level coordination (e.g., Ray, distributed file systems) hides complexity of data placement and communication.
- **Applications**: Domain logic (web servers, databases, analytics) running atop the entire stack.

Protocol Responsibilities:

- Encapsulate bits into self-contained messages.
- Decide on packet ordering (e.g., TCP enforces order; UDP does not).
- Handle lost packets (timeouts, retransmissions).
- Manage connection handshakes (e.g., TCP's three-way handshake).

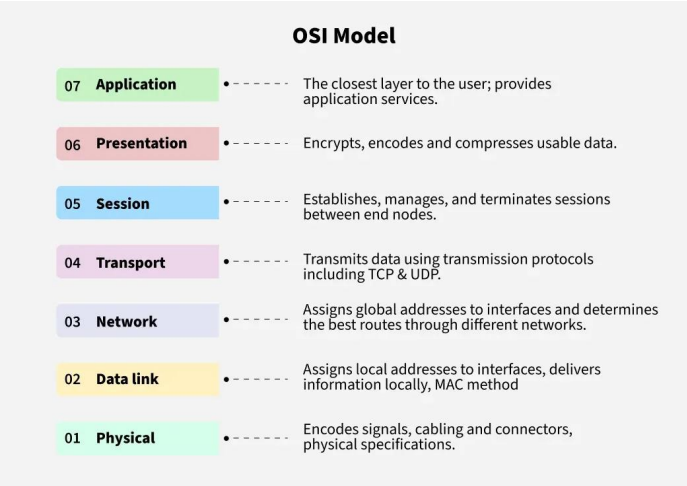
Networking has its own stack, similar to a software stack on a computer. The network stack is layered to manage complexity and modularize communication tasks. Each layer adds metadata (in the form of headers) to the data as it passes through:

1. **Physical Layer**:
    - Responsible for the transmission of raw bits over a physical medium (e.g., copper wires, fiber optics, radio waves).
    - Defines electrical/optical signaling, bit rate, voltage levels, modulation, and connector types.
    - *Headers Added:* None — only physical encoding and transmission occur here.
2. **Link Layer**:
    - Ensures reliable frame transfer between directly connected devices on the same network segment.
    - Handles framing, error detection (e.g., CRC), and MAC (Media Access Control) addressing.

- Common protocols: Ethernet, Wi-Fi, NVLink.
- *Headers Added:* Frame headers with MAC source/destination addresses and CRC.

3. **Network Layer**:
   - Manages logical addressing and routes packets across interconnected networks.
   - Introduces IP addressing to identify source and destination devices across network boundaries.
   - Supports both IPv4 (32-bit) and IPv6 (128-bit) addresses.
   - *Headers Added:* IP headers including source and destination IP addresses, TTL, and fragmentation info.
4. **Transport Layer**:
   - Provides end-to-end communication between processes on different hosts.
   - Responsible for segmentation, reassembly, reliability, and flow control.
   - Two main protocols:
     - **TCP** — reliable, ordered, connection-oriented (used in HTTP, SSH, FTP).
     - **UDP** — unreliable, unordered, connectionless (used in DNS, streaming, VoIP).
   - *Headers Added:* TCP/UDP headers with source/destination ports, sequence numbers, checksums, flags.
5. **Application Layer**:
   - Closest to the end user; supports application-specific network functions.
   - Examples include HTTP (web), FTP (file transfer), SMTP (email), DNS (name resolution).
   - Interprets and generates data for specific applications.
   - *Headers Added:* Protocol-specific headers like HTTP GET lines, SMTP commands, etc.

Each layer adds headers to the message. At the receiver, these are stripped in reverse. This modular structure is useful but adds overhead. Some systems bypass layers (e.g., DPDK) for performance.

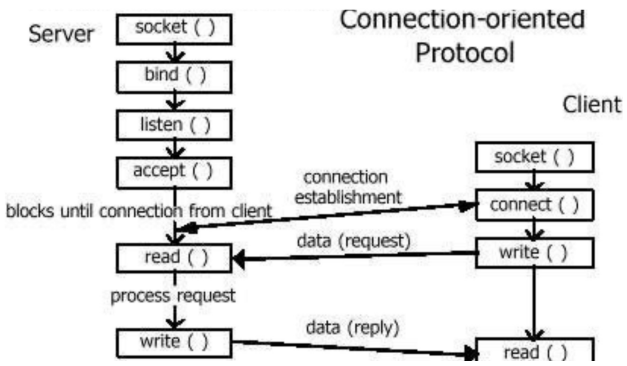


Local vs. Remote Communication

On a single machine, processes can communicate using mechanisms like:

- **Shared memory** — Efficient since everything is in RAM.
- **Pipes** — A basic way to pass data between processes.

However, when processes run on different machines, these mechanisms no longer work. There is no shared memory or pipe between machines. In such cases, we use **network-based communication**, typically via **sockets**.



Sockets

Sockets allow communication between processes over a network. They're often used in a client-server model.

Server (Python)

```
import socket

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('127.0.0.1', 65432))
server_socket.listen()
print("Server is listening...")

conn, addr = server_socket.accept()
print(f"Connected by {addr}")

data = conn.recv(1024)
if data:
    print(f"Received: {data.decode('utf-8')}")
    conn.sendall("Hello, client!".encode('utf-8'))

conn.close()
server_socket.close()
```

Client (Python)

```
import socket
```

Code	Explanation
`socket.socket(socket.AF_INET, socket.SOCK_STREAM)`	Creates a TCP socket using IPv4. `AF_INET` specifies IPv4, and `SOCK_STREAM` specifies TCP, which provides reliable, connection-oriented communication.
`bind(('127.0.0.1', 65432))`	Binds the socket to localhost (loopback IP) and port 65432, allowing the server to listen on that address.
`listen()`	Puts the server socket into listening mode, enabling it to accept incoming connection requests.
`conn, addr = accept()`	Blocks until a client connects. Returns a new socket object `conn` to communicate with the client, and `addr` is the client's address.
`data = conn.recv(1024)`	Receives up to 1024 bytes from the client. This is a blocking call and waits for data.
`conn.sendall(...)`	Sends a UTF-8 encoded message back to the client. `sendall` ensures the complete message is sent.
`conn.close(), server_socket.close()`	Closes the individual client connection and then the main server socket.

Table 1: Line-by-line Explanation of Server Socket Code

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(('127.0.0.1', 65432))
client_socket.sendall("Hello, server!".encode('utf-8'))

data = client_socket.recv(1024)
print(f"Received from server: {data.decode('utf-8')}")

client_socket.close()
```

Code	Explanation
`socket.socket(socket.AF_INET, socket.SOCK_STREAM)`	Creates a TCP socket with IPv4 addressing. This matches the server's configuration.
`connect(('127.0.0.1', 65432))`	Connects to the server at the specified IP and port. Triggers a TCP 3-way handshake.
`sendall(...)`	Sends a UTF-8 encoded string message to the server.
`recv(1024)`	Waits and reads up to 1024 bytes from the server's response.
`close()`	Gracefully closes the socket, ending the connection.

Table 2: Line-by-line Explanation of Client Socket Code

Data sent through sockets is just raw bytes. Both ends must agree on how to interpret it.

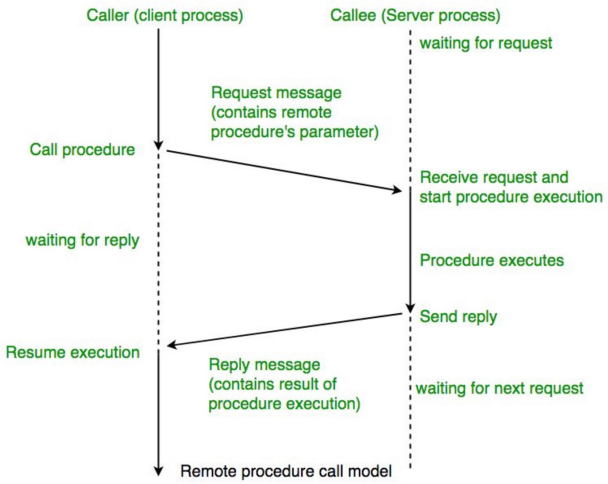
Remote Function Calls and RPC

Invoking a function on another machine—called a **remote function call**—requires several steps:

- Establishing a network connection
- Serializing and sending the function arguments
- Executing the function on the remote machine

- Receiving and deserializing the result

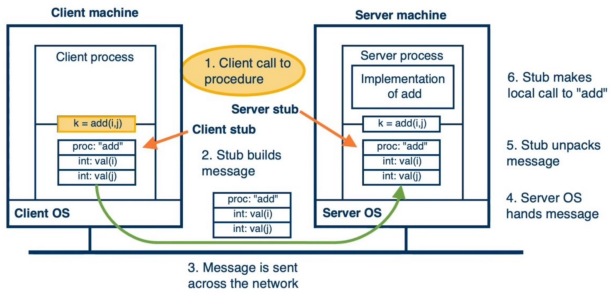
Doing this manually is tedious and error-prone. To simplify this, we use the abstraction of a **Remote Procedure Call (RPC)**. RPC allows a program to invoke procedures located on a remote system as if



they were local functions. It hides the underlying complexity of network communication, making distributed programming easier and more intuitive.

Key Concepts

- **Transparency**: RPC provides *call transparency*, meaning remote calls appear identical to local function calls in code.
- **Service**: A collection of related remote procedures exposed by a server.
- **IDL (Interface Definition Language)**: Specifies the procedures available and their argument/return types in a language-neutral way.
- **Stub and Skeleton**:
  - **Client Stub**: Acts as a proxy for the remote function. It serializes the arguments, sends the request, waits for a reply, and deserializes the result.
  - **Server Skeleton**: Receives and deserializes the request, invokes the actual local function, and sends the result back.

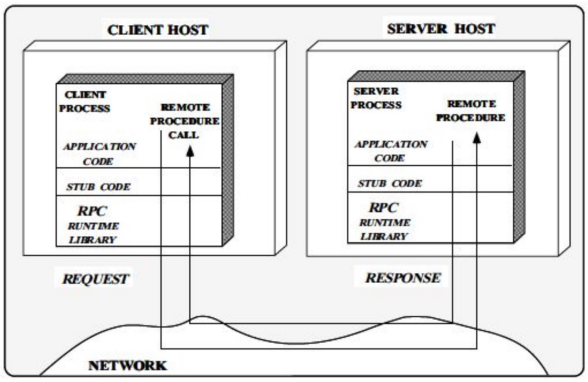


What RPC Handles Automatically

- Argument/result serialization and deserialization (also called *marshalling* and *unmarshalling*)
- Establishing and managing network communication
- Handling platform differences (e.g., endianness, data types)
- Optional support for timeouts, retries, and exception handling

RPC systems are typically built on the following components:

- **Service Definitions**: A specification of the remote functions the server provides.
- **IDL Compiler**: Processes the service definition to generate:
  - Client-side stubs
  - Server-side skeletons
  - Serialization and deserialization code



Example: Python RPC with XML-RPC

Python provides a built-in XML-RPC module for simple remote procedure calls.

Server

```
from xmlrpc.server import SimpleXMLRPCServer

def add_numbers(x, y):
    return x + y

server = SimpleXMLRPCServer(('127.0.0.1', 8000))
server.register_function(add_numbers, 'add')
server.serve_forever()
```

Client

```
import xmlrpc.client

server = xmlrpc.client.ServerProxy('http://127.0.0.1:8000')
result = server.add(5, 7)
print(f"Result: {result}")
```

From the client's perspective, the call `server.add(5, 7)` behaves just like a regular function call, even though it is executed remotely.

Limitations and Challenges of RPC

- **No pass-by-reference**: All arguments and return values must be serialized; object references can't be shared across machines.
- **Network faults**: Unlike local calls, remote calls can fail due to latency, timeouts, or disconnections.
- **Service discovery**: In large systems, additional infrastructure is needed to locate and manage services dynamically.
- **Over-abstraction**: Treating remote calls like local ones can mask latency or failure behavior, leading to inefficient or brittle designs.

Note: RPC is usually provided as a library (e.g., gRPC, Thrift, XML-RPC), not as a built-in language feature—though some languages (like Java with RMI) offer native support.