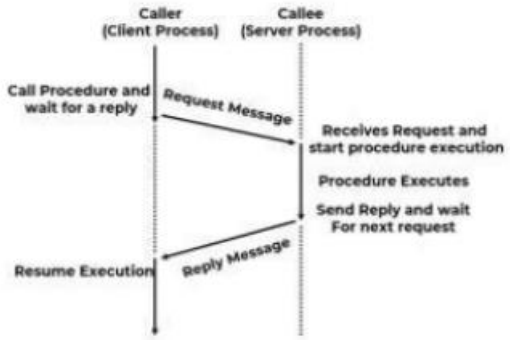
Q. Explain the concept of Remote Procedure Call

RPC stands for **Remote Procedure Call**, which is also called **Remote Function Call** or **Remote Subroutine Call**. It's a way for a program (called the **client**) to call a function or procedure on another computer (called the **server**) over a network, as if it were a local function call.

Key Points:

1. **Interprocess Communication**: RPC helps different programs (on different machines) communicate with each other in a client-server setup.
2. **Transparency**: The client doesn't need to know that the procedure is actually running on another machine. It behaves just like a local function call.
3. **Simple and Efficient**: It makes distributed applications simpler to write and more efficient, without the developer worrying about low-level network details.



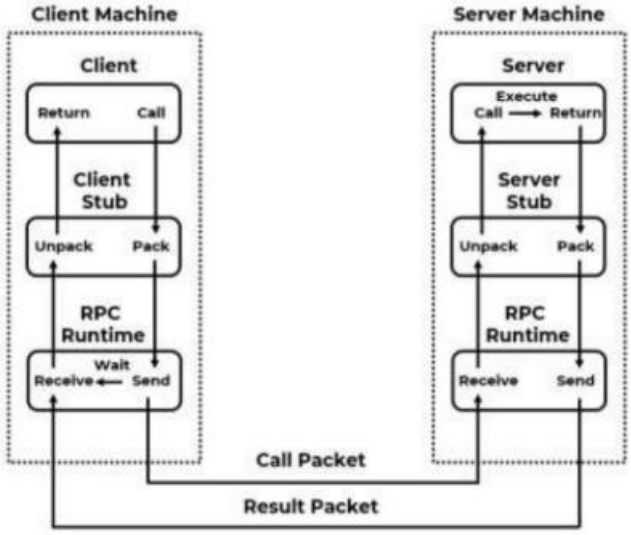
How RPC Works:

1. The **client** wants to call a procedure on the **server**.
2. The client sends a message to the server with the procedure name and parameters (inputs).
3. The **server** receives the request, executes the procedure, and sends the result back to the client.
4. The client receives the result, and continues its work, as if the procedure was called locally.

RPC Model:

- **Client**: Starts the remote procedure call.
- **Client Stub**: Packs the call request and sends it to the server. Once the server responds, it unpacks the result for the client.
- **RPC Runtime**: Handles network communication (sending messages, routing, etc.).

- **Server Stub**: Receives the call from the client, forwards it to the correct procedure, and sends back the result.
- **Server**: Executes the procedure and returns the result to the client.



Advantages of RPC:

1. **Easy to Use**: It abstracts the complexity of network communication, making remote calls look like local calls.
2. **Reusability**: Once implemented, RPC allows code to be reused across different environments.
3. **Supports Different Models**: Works with both process-oriented and thread-oriented models.

Disadvantages of RPC:

1. **No Standard**: There are many ways to implement RPC, but no single standard.
2. **Limited Flexibility**: It works well for software communication but has limitations with hardware differences.
3. **Cost**: There may be additional overhead or costs due to the network communication.

Example in Simple Terms:

Imagine you're using a website (client) to get weather information from a server:

- You ask for the weather.
- The website (client) sends a request to the server to get the weather data (RPC call).
- The server processes the request and sends the data back.
- The website (client) displays the weather information to you.

In this case, the website (client) doesn't need to know how the server processes the weather data. It just makes the request and gets the result.

Q. Explain Call Semantics of RPC

In an RPC system, the **call semantics** defines how an RPC behaves when things go wrong, like if messages are lost or nodes crash. It tells us how often a remote procedure will run when there are faults (like a crash or message loss).

Types of RPC Call Semantics:

1. Possibly (or May-Be) Call Semantics:

- **Weakest type** of RPC semantics.
- It works in an **asynchronous** way, meaning the caller doesn't wait forever for a response.
- If a response doesn't come back in time (due to a timeout), the caller just moves on and doesn't worry about the result.
- This is useful when **getting a response isn't crucial** (like checking if a server is alive).

2. Last-Once Call Semantics:

- **Request/Reply protocol** with a **timeout** mechanism.
- If the response isn't received within the timeout period, the caller retries the request.
- After a retry, the caller **uses the result of the last execution**.
- This is useful for simple RPC systems where **the caller can deal with not getting a response right away**.

3. Last-of-Many Call Semantics:

- Similar to **Last-Once**, but it uses **unique call identifiers** to track calls.
- If a **caller crashes** (called an **orphan call**), the system ignores it.
- Each call gets a unique **call identifier** to avoid repeating calls.
- The response is accepted only if the **call ID** matches the most recent one, ensuring that old responses from orphaned calls are ignored.

4. At-Least-Once Call Semantics:

- **Guarantees** that the procedure will be executed at least once.

- Even if the message is lost and the system retries, the call will be made **at least once**.
- The system doesn't care if **multiple responses** are returned—it just ensures **one or more executions** of the procedure.

5. Exactly-Once Call Semantics:

- The **strongest** and **most desirable** type of semantics.
- No matter how many times the call is retried or how many messages are sent, the procedure will only be **executed once**.
- This eliminates any risk of running the procedure more than once, even if there are retries.
- It uses **timeouts, retransmissions, unique call identifiers**, and **caching** to make sure the call is executed only once.

Summary:

- **Possibly Call Semantics**: Might not execute, no guarantees.
- **Last-Once Call Semantics**: Executes at least once, uses last result after a timeout.
- **Last-of-Many Call Semantics**: Similar to last-one, but avoids orphan calls using call IDs.
- **At-Least-Once Call Semantics**: Guarantees at least one execution, but may execute multiple times.
- **Exactly-Once Call Semantics**: Guarantees only one execution, no matter how many retries.

Q. Explain parameters passing in RPC

When the client wants to call a function on a remote server, it needs to pass data (parameters) to the function. The process of sending this data from the client to the server and back is called **parameter passing**.

Steps Involved:

1. **Client Stub**: The client stub (a piece of code that acts like a local function call) takes the parameters and **packs them into a message**.
2. This message is then sent over the network to the **Server Stub** (on the server side).
3. The **Server Stub** unpacks the message, extracts the parameters, and calls the actual function on the server.

4. Once the server finishes the function, it sends the result back in another message.
5. The **Client Stub** receives this result, unpacks it, and passes it to the client's function.

Types of Parameters Passing in RPC:

I) Call by Value:

- **Packing parameters** into a message is called **marshaling**.
- All parameters (like numbers, strings) are copied into the message and sent from the client to the server.
- Example: Imagine the client wants to call a function sum(i, j) to add two numbers. The client sends i and j in the message.
- When the server receives the message, it knows which function to call and gets the parameters (values of i and j).
- After the function is done, the server sends the result (like the sum) back to the client.

Example of an issue:

- Imagine one computer uses **ASCII** for characters and another uses **EBCDIC**. Passing characters directly might cause problems because the two systems interpret characters differently.
- **Drawback**: This method isn't good for **large data** because copying and sending large amounts of data in a message can be slow and inefficient.

II) Call by Reference:

- Instead of sending the actual data, the **reference (address)** to the data is sent.
- This means the client sends a **pointer** (like an address) that points to where the data is stored, instead of sending the data itself.
- This method is often used in **distributed systems** with **shared memory** where processes can access the same data from different machines.

Example:

- Imagine the client wants to pass a large buffer of data to the server. Instead of copying the entire buffer, the client just sends the **memory address** where the buffer is stored.
- However, this can create problems: if the server is on a **different machine**, the memory address won't work because the server's memory is different from the client's memory.

Drawback:

- **Pointers** are only meaningful within a single machine's memory. When the client and server are on different machines, pointers don't work properly.

- This method is usually used in systems where the client and server are closely connected (like in a **closed system**).

Q. Explain message-oriented communication with suitable example.

Message-oriented communication is a way for processes (or programs) to communicate by sending and receiving messages. These messages are the **basic units of data** that are exchanged between processes. Think of it like sending a letter where the letter is the "message" and the post office handles the delivery.

There are two main types of message-oriented communication:

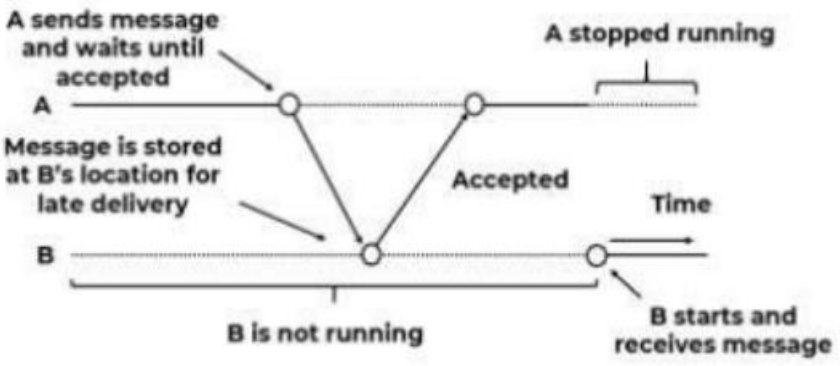
1. **Persistent Communication**: The message is stored until it's delivered.
2. **Transient Communication**: The message is lost if it can't be delivered immediately.

1. Persistent Communication:

In this type, the message is **stored** at a server until it can be delivered to the receiver. The message is not lost if the receiver is not available at the moment.

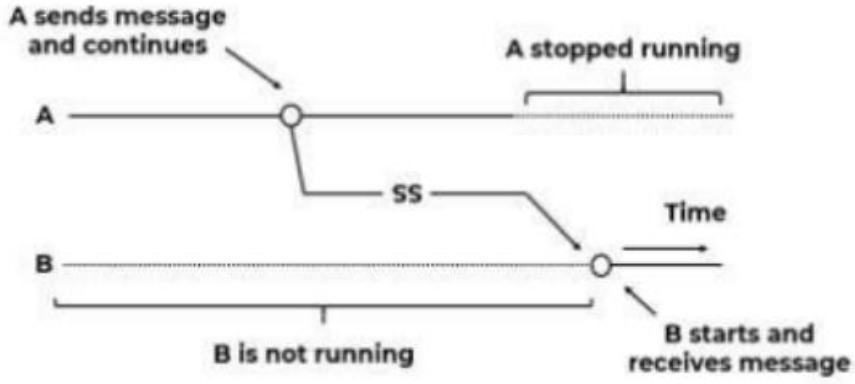
Types of Persistent Communication:

- **Persistent Synchronous Communication**:
    - The message is stored at the receiving host (the server).
    - The sender **waits (is blocked)** until the message is safely stored at the receiver's end.
    - Example: **Email system**—the message stays in the server's inbox until it's delivered to the recipient, even if they're not online.



- **Persistent Asynchronous Communication**:

- The message is stored temporarily in a buffer or server until the receiver can handle it.
- The sender doesn't have to wait for a response and can continue working immediately after sending the message.
- Example: **Email system** again—just like the previous type, but the sender doesn't block while waiting for the message to be received.

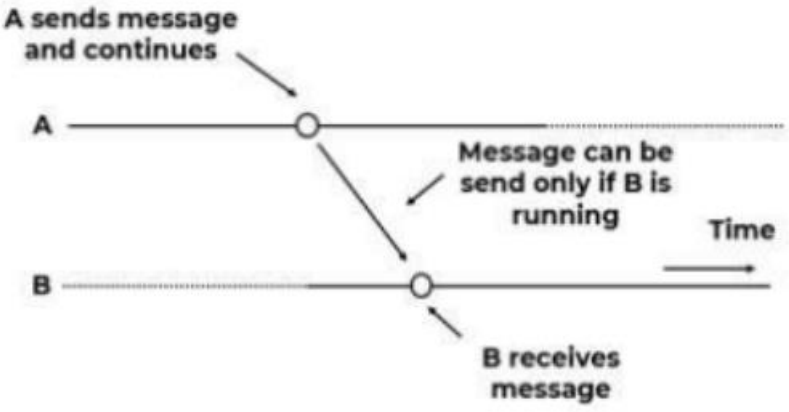


2. Transient Communication:

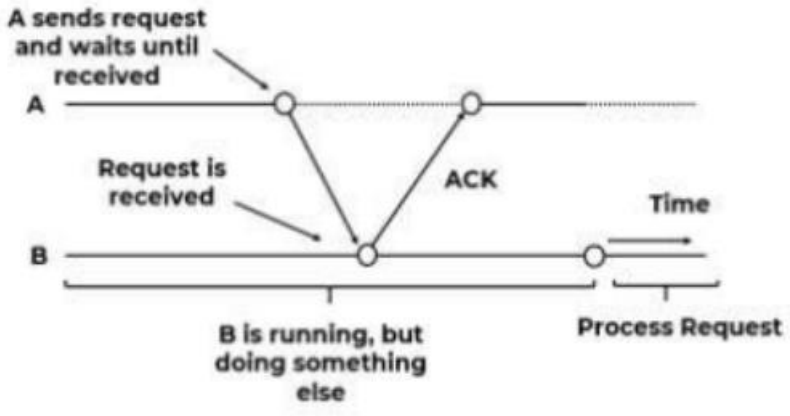
In this type, the message is **not stored**. If the receiver is not available, the message is lost. This happens when both sender and receiver need to be active at the same time for the message to be delivered.

Types of Transient Communication:

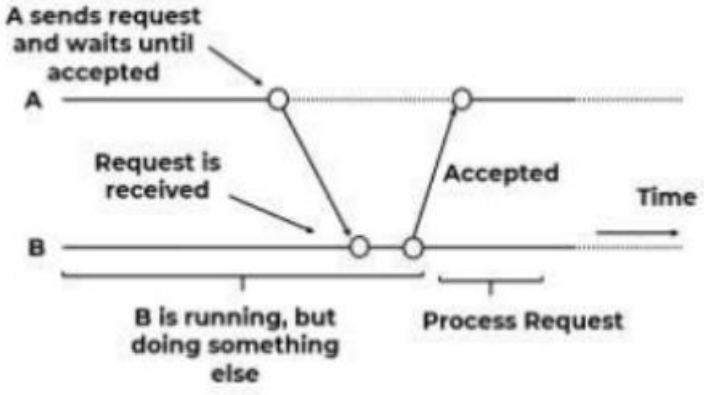
- **Transient Asynchronous Communication**:
  - The sender sends the message and **continues immediately** without waiting.
  - If the receiver is not available or some router along the way is down, the message is **discarded**.
  - Example: **UDP (User Datagram Protocol)**—this is a fast communication method where if the receiver is not available, the message is simply lost.



- **Receipt-based Transient Synchronous Communication**:
    - The sender waits (blocks) until it **receives an acknowledgment (ack)** that the message was **received** by the receiver.
    - This doesn't mean the receiver processed the message, just that it was delivered.
    - Example: A **confirmation** message that the receiver received the request, but not necessarily that it acted on it.

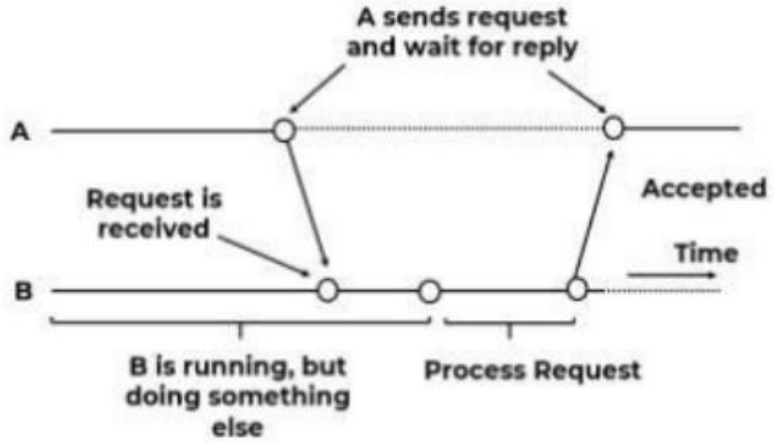


- **Delivery-based Transient Synchronous Communication**:
    - Similar to the receipt-based type, but here the sender waits until the message is actually **delivered** and processed by the receiver.
    - The sender waits for a more detailed **acknowledgment** that the message was both **received and acted upon**.
    - Example: **Remote Procedure Call (RPC)** where the sender is blocked until the server completes its action and sends a reply.



- **Response-based Transient Synchronous Communication**:
    - This is similar to traditional RPC where the sender is blocked until it **receives a response** from the receiver.

- The message is delivered, processed, and then the sender gets a reply (response).
- Example: A typical **RPC** where the client waits for the server to process the request and send back a result.



Q. Explain stream-oriented communication with suitable example.

Stream-oriented communication is different from **message-oriented communication**. Instead of sending discrete messages, like an email or a request, stream-oriented communication sends a **continuous flow of data**.

In this type of communication, timing is important, and data must be delivered at a specific rate, as well as correctly. This is crucial for things like **music** or **video**, where the data needs to flow smoothly and without interruptions.

Examples of Stream-Oriented Communication:

- **Music or Video streaming**: When you're listening to music or watching a video online, the data is sent as a continuous stream. The audio and video must be delivered at the right speed to avoid buffering or interruptions.

Transmission Modes in Stream-Oriented Communication:

Stream-oriented communication can happen in three different modes based on how the data is delivered:

1. **Asynchronous Mode**:
    - There's no strict requirement on **when** data should be delivered.
    - The system just sends the data as it can, without worrying about timing.

- Example: Online video platforms (like YouTube) that can adjust streaming quality based on available bandwidth.

2. **Synchronous Mode**:
    - The system sets a **maximum delay** for how long it can take for data to travel from one place to another.
    - Each packet of data (like a part of a video or song) must arrive within a certain time.
    - Example: Video calls where each frame needs to arrive quickly so the video is smooth.
3. **Isochronous Mode**:
    - This is a stricter version of synchronous mode, where there are two rules:
        - A **maximum delay** for data.
        - A **maximum variance** in delay (i.e., the delay shouldn't change drastically).
    - This is important for high-quality streaming, where consistent timing is critical.
    - Example: Live streaming a concert or live sports events, where the timing must be extremely precise.

Stream Characteristics:

- **Stream**: A stream is a continuous flow of data between two points, and it's usually **one-way (unidirectional)**.
- **Single Source**: The stream typically has a **single source** of data.

Types of Streams:

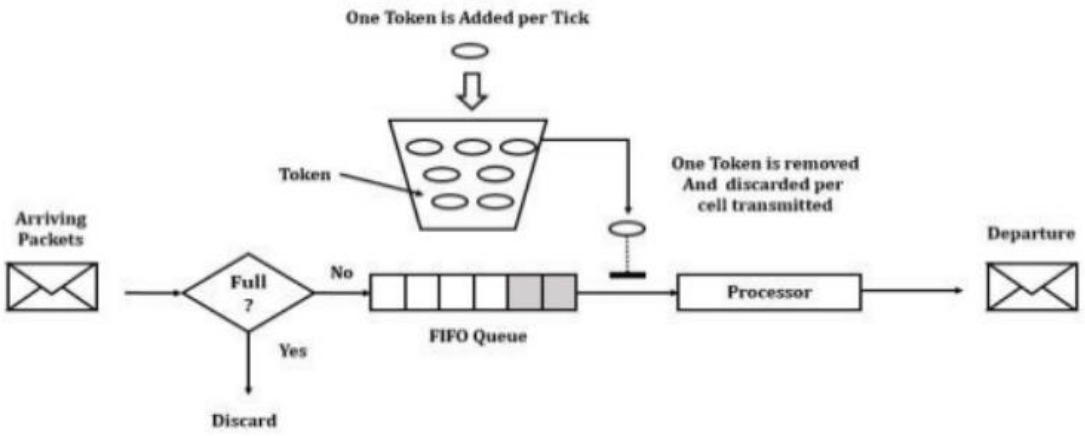
1. **Simple Stream**:
    - A stream with a single flow of data.
    - Example: **Audio** or **video** being streamed in a single direction (e.g., a song or a movie).
2. **Complex Stream**:
    - A stream with multiple types of data flowing together.
    - Example: **Stereo audio** (two audio channels) or **audio and video** combined in a video call.

Example of Stream-Oriented Communication: Token Bucket Algorithm

One way to manage the flow of data in stream-oriented communication is using something called the **Token Bucket Algorithm**. Here's how it works:

1. **Token Bucket**: Imagine a bucket where each "token" represents permission to send a unit of data.
2. Each time a token is added to the bucket, it increments by 1.
3. When data is sent, a token is **removed** from the bucket.
4. If the bucket is empty (no tokens), the system cannot send any more data.
5. This controls the flow of data, making sure it's sent at a certain rate without overloading the system.

Example: If you're streaming video, the **token bucket** ensures that the data (video packets) are sent at a steady, manageable rate.



Q. Differentiate between Message oriented & Stream oriented communications

Feature	Stream-Oriented Communication	Message-Oriented Communication
Type of Communication	Connection-oriented (requires a stable connection)	Connectionless (does not require a connection)
Communication Style	1-to-1 (one sender, one receiver)	Many-to-many (multiple senders and receivers)
Data Transfer	Transfers a continuous flow of bytes (like a stream)	Transfers discrete, separate messages (like a letter)

Feature	Stream-Oriented Communication	Message-Oriented Communication
Data Sending	Sends data **byte-by-byte**	Sends data as **complete messages**
Message Boundaries	Does **not** keep track of message boundaries	**Keeps track** of message boundaries
Data Size	Can transfer **arbitrary length** of data	Each message is usually **limited to 64 KB**
Usage	Common for many applications (like file transfers)	Used for **multimedia** and similar applications
Protocol	Typically runs over **TCP** (reliable)	Typically runs over **UDP** (faster, but less reliable)
Timing Impact	**Timing matters**; delays affect performance	**Timing doesn't matter** as much
Example	Downloading a song or watching a movie	Sending a sequence of web pages (HTTP)

Q. Explain Group Communication

Group Communication refers to sending messages to a **group** of processes or users who share a common interest. It involves **one-to-many**, **many-to-one**, or **many-to-many** communication patterns.

Types of Group Communication

1. One-to-Many Communication (Multicast Communication)

- **One sender, many receivers**.
- **Example**: Sending an email to multiple recipients at once.

Key Concepts:

- **Group Management**:
    - **Closed Group**: Only group members can send messages to the group.
    - **Open Group**: Anyone can send a message to the group.
- **Group Addressing**:

    - Special addresses (like **multicast** or **broadcast**) are used to send messages to multiple recipients.

- **Buffered vs. Unbuffered Multicast**:
    - **Unbuffered**: If a receiver is not ready, the message is lost.
    - **Buffered**: The message is stored until the receiver is ready.
- **Semantics**:
    - **Send-to-All**: Every group member gets the message, and it is stored until accepted.
    - **Bulletin-board**: The message is sent to a channel instead of each individual group member.
- **Reliability**:
    - **0 Reliable**: No acknowledgment needed.
    - **1 Reliable**: At least one acknowledgment expected.
    - **M out of N Reliable**: Sender decides how many acknowledgments are needed from N members.

2. Many-to-One Communication

- **Multiple senders, one receiver**.
- **Example**: Several people trying to send their reports to a manager.

Key Concepts:

- **Selective Receiver**: The receiver only accepts messages from specific senders.
- **Nonselective Receiver**: The receiver accepts messages from any sender.

3. Many-to-Many Communication

- **Multiple senders, multiple receivers**.
- **Example**: A group chat where everyone can send and receive messages.

Key Concepts:

- **Message Ordering**: In many-to-many communication, messages may arrive in different orders for different receivers.
    - **No Ordering**: Messages may arrive out of order.
    - **Absolute Ordering**: Messages are delivered in the exact order they were sent.
    - **Consistent Ordering**: All receivers get messages in the same order, but the order might differ from the sender's order.

- **Casual Ordering**: If messages are related, they must be delivered in the correct order.

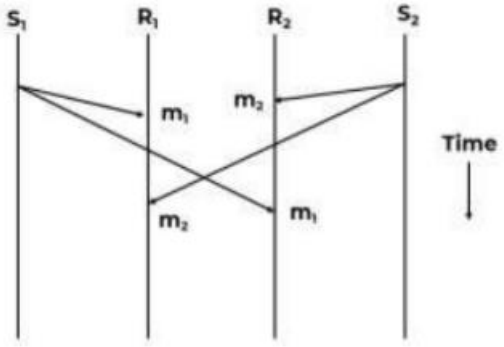


Figure 2.10: No Ordering

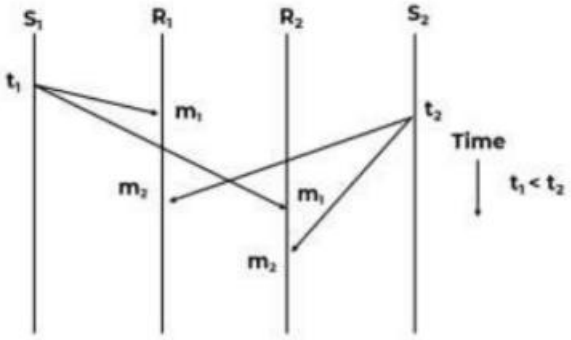


Figure 2.11: Absolute Ordering

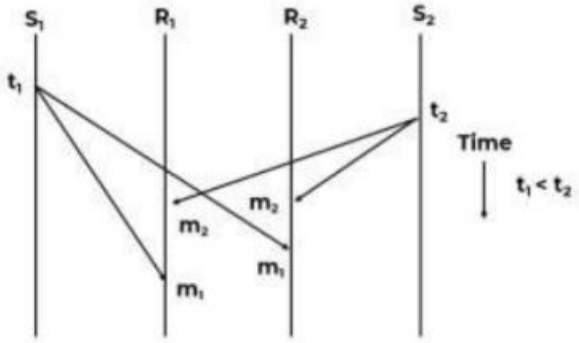


Figure 2.12: Consistent Ordering

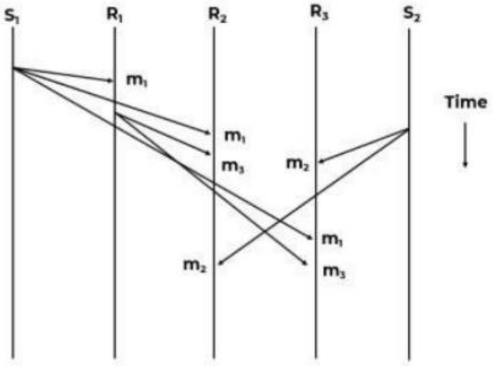


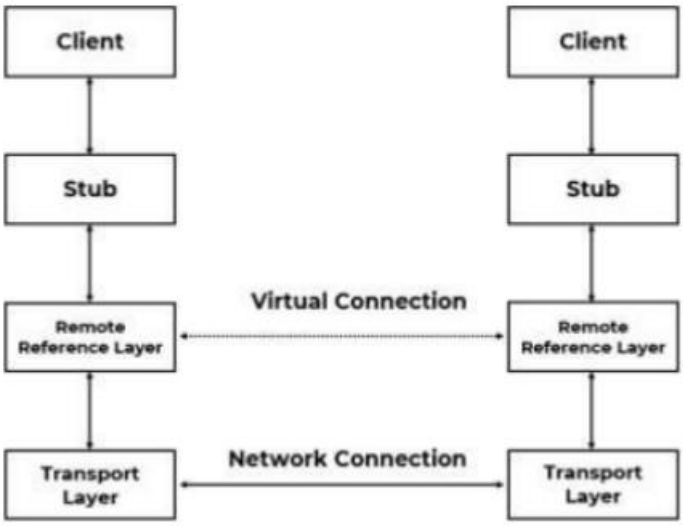
Figure 2.13: Casual Ordering

Q. Explain Remote Method Invocation (RMI)

RMI is a Java API that allows a program (client) to call methods on an object located on a different machine (server) over a network. It enables communication between a **client** (on one machine) and a **server** (on another machine) through **method calls**. The client invokes a method on a remote object that resides on the server.

RMI Architecture

- RMI uses **two programs**:
    1. **Server Program** (on the server)
    2. **Client Program** (on the client)
- The **server** creates a remote object and makes it available to the client via a **registry**.
- The **client** then uses the object on the server by invoking its methods.



Key Components in RMI:

1. **Transport Layer**: Connects the client and server, managing the connections.
2. **Stub**: A "proxy" for the remote object on the client side. It's like a gateway through which the client makes calls.
3. **Skeleton**: The server-side object that communicates with the stub to handle the request and pass it to the actual remote object.
4. **Remote Reference Layer (RRL)**: Manages references to the remote object and ensures the correct communication between the client and server.

How Does RMI Work?

1. The **client** calls a method on a **remote object** via the **stub**.
2. The **stub** sends the request to the **Remote Reference Layer (RRL)** on the client side.
3. The **RRL** sends the request to the **RRL** on the **server side**.
4. The **server-side RRL** passes the request to the **skeleton**, which then invokes the required method on the **remote object**.
5. The **server** sends the result back through the same steps, eventually reaching the client.

Marshalling and Unmarshalling:

- **Marshalling**: When the client sends data (parameters) to the server, the data is packed into a message.
    - If the data is a **primitive type**, it is bundled and sent.
    - If it's an **object**, it is converted (serialized) into a format that can be sent over the network.

- **Unmarshalling**: At the server, the data is unpacked, and the required method is called using the received parameters.

Goals of RMI:

1. **Simplify Application Development**: RMI makes it easier to create distributed applications by abstracting network complexities.
2. **Type Safety**: Ensures the correct types of parameters are passed in method calls.
3. **Distributed Garbage Collection**: Automatically cleans up objects that are no longer in use.
4. **Local and Remote Objects Are Treated Similarly**: RMI minimizes the difference between interacting with local (on the same machine) and remote (on another machine) objects.

Q. Explain ISO OSI reference model with diagram?

The OSI Model is a standard created by **ISO (International Organization for Standardization)** to help different computer systems communicate with each other, no matter where they are in the world.
It has **7 layers**, and each layer has its own job in the communication process.

7 Layers of the OSI Model (from bottom to top):

1) Physical Layer

- This is the lowest layer.
- It handles the actual physical connection between devices (like cables and switches).
- It sends raw bits (0s and 1s) over the network.
- It decides how devices are connected and how fast data is sent.

Main jobs:

- Send bits as signals
- Set data transfer rate
- Control how devices connect
- Sync data using a clock

2) Data Link Layer

- It makes sure data is passed safely from one device to another directly connected device.

- It puts data into **frames** and adds the **physical address (MAC address)**.
- It also checks for errors.

Main jobs:

- Error detection
- Flow control
- Divides data into frames
- Manages access to the physical medium

3) Network Layer

- It handles sending data between **different networks**.
- Adds **logical addressing (IP address)** to packets.
- Uses **routers** to send data to the right destination.

Main jobs:

- Routing (choosing the best path)
- Logical addressing (like IP)
- Connects different networks
- Ensures data goes from source to destination

4) Transport Layer

- Sends data from one **application (program)** to another.
- Breaks big messages into smaller **packets** and reassembles them at the other end.
- Makes sure all packets arrive safely and in order.

Main jobs:

- Port addressing
- Error and flow control
- Divides and reassembles data
- Can use **TCP (reliable)** or **UDP (fast)**

5) Session Layer

- Manages the **sessions (conversations)** between two computers.
- Starts, manages, and ends communication between apps.

Main jobs:

- Session management (start/stop)
- Dialog control (who talks when)
- Keeps data in the correct order
- Helps with login/logoff

6) Presentation Layer

- Prepares data so that it's readable by the receiving system.
- Converts data formats, and handles **encryption/decryption** and **compression**.

Main jobs:

- Data format translation (e.g., ASCII to EBCDIC)
- Data encryption/decryption
- Data compression
- Makes sure data looks right to the receiving device

7) Application Layer

- This is the topmost layer.
- It allows users to interact with the network through applications like email, browsers, and file transfer tools.

Main jobs:

- Provides services like Email, FTP, and Web Browsing
- Remote login
- File transfers
- User interface for network services