**UNIT-2  
  
Primitives for Distributed Communication**

**1. Message Send and Receive Primitives**

* **Send(Send())** → Sends data from one process to another.
  + Parameters:
    1. **Destination** (where to send the message)
    2. **Buffer** (data to be sent)
* **Receive(Receive())** → Receives data from another process.
  + Parameters:
    1. **Source** (from where the message is coming; can be a wildcard)
    2. **User Buffer** (where to store received data)

📌 **Example:**

* Process A sends "Hello" to Process B using Send(B, "Hello").
* Process B receives it using Receive(A, buffer).

**2. Ways of Sending Data**

**A. Buffered Option (Standard Method)**

* Data is first copied to a **kernel buffer**, then sent over the network.
* Prevents **data loss** if the receiver is not ready.

📌 **Example:**  
Saving a message as a draft before sending it.

**B. Unbuffered Option (Direct Method)**

* Data is **immediately sent** from the user buffer to the network.
* Faster but **risky** if the receiver is not ready.

📌 **Example:**  
Sending a text message without saving it first.

**C. Buffered Receiving (Mostly Used)**

* Data is **stored in a kernel buffer** before being placed in the user’s buffer.

📌 **Example:**  
A receptionist receives packages and holds them until the recipient collects them.

**3. Synchronous and Asynchronous Primitives**

**A. Synchronous Primitives**

* **Send() and Receive() must handshake** (both must be called).
* Send completes **only after** the receiver confirms receipt.
* Ensures **message delivery** but may slow down communication.

📌 **Example:**  
A phone call → Both people must be available for the conversation to happen.

**B. Asynchronous Primitives**

* **Sender does not wait** for confirmation; it continues processing.
* Receiver may not be ready, so message gets stored.

📌 **Example:**  
Sending an email → The sender doesn’t wait for the receiver to open it.

**4. Blocking vs. Non-Blocking Primitives**

**A. Blocking Primitives**

* The process **waits** until the operation completes.
* Used when **message delivery is crucial**.

📌 **Example:**  
Calling a restaurant and waiting for them to confirm your order.

**B. Non-Blocking Primitives**

* The process **continues working immediately** after sending or receiving.
* A **handle** is used to check status later.

📌 **Example:**  
Ordering food online and getting a notification when it’s ready.

**How to Check Message Status?**

1. **Polling:** Keep checking the status in a loop.
2. **Wait with Handles:** Wait until a notification arrives.

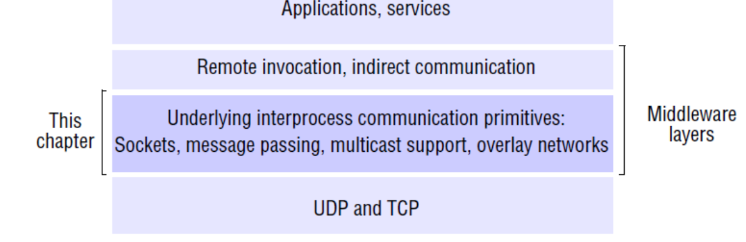
**5. Inter-Process Communication (IPC)**

* Characteristics of protocols for communication between processes in a distributed system
* Processes in a distributed system communicate using **Send** and **Receive**.
* The Message Passing Interface (MPI) is a standard developed to provide an API for a set of message-passing operations

with synchronous and asynchronous variants.

* IPC in the Internet provides both :
  1. **Datagram Communication** → Independent message transmission (e.g., UDP).
  2. **Stream Communication** → Continuous data flow (e.g., TCP).

📌 **Example:**

* **Datagram:** Sending a text message (each message is separate).
* **Stream:** Watching a live stream (continuous data transfer).  
  

**Characteristics/api of Inter-Process Communication**

**1. Message Passing**

* Communication between processes occurs through **send** and **receive** operations.
* A **message** (sequence of bytes) is sent to a **destination**, where another process **receives** it.
* This involves:
  + **Data transfer** (from sender to receiver).
  + **Process synchronization** (ensuring correct message exchange).
* **Example:**
  + Process A: Send(destination, message)
  + Process B: Receive(source, buffer)

**2. Synchronous and Asynchronous Communication**A queue is associated with each message destination. Sending processes cause messages to be added to remote queues and receiving processes remove messages from local queues.

**A. Synchronous Communication**

* **Requires synchronization** between sender and receiver. Whenever a send is issued the sending process (or thread) is blocked until the corresponding receive is issued. Whenever a receive is issued by a process (or thread), it blocks until a message arrives.
* **Blocking behavior**:
  + Send() blocks until Receive() is called.
  + Receive() blocks until a message arrives.
* **Example:**
  + A **phone call** (both parties must be available).

**B. Asynchronous Communication**

* **No synchronization needed** between sender and receiver.
* **Non-blocking Send()**: The sender continues processing after copying the message to a local buffer.
* **Receive() can be:**
  1. **Blocking:** Waits until a message arrives.
  2. **Non-blocking:** Proceeds immediately and gets notified later (via polling or interrupt).
* **Example:**
  1. **Email** (sent immediately, read later).

**3. Message Destinations**

* In **Internet protocols**, messages are sent to (Internet Address, Local Port).
* **Local Port:** A unique number within a computer (specified as an integer).
* **Rules for message destinations:**
  + A **port** has **one receiver** but **many senders**.
  + Processes can use **multiple ports** for different message types.
  + Servers **publicize their port numbers** for client communication.
* **Example:**
  + Web servers use **port 80** for HTTP and **port 443** for HTTPS.

**4. Reliability in Communication**

**A. Reliable Communication**

* Ensures **messages are always delivered**, even if some packets are lost.
* **Validity Property:** Messages must be delivered despite a ‘reasonable’ number of packet losses.
* **Integrity Property:**
  + Messages **must not be corrupted**.
  + Messages **must not be duplicated**.
* **Example:**
  + **TCP (Transmission Control Protocol)** guarantees reliable communication.

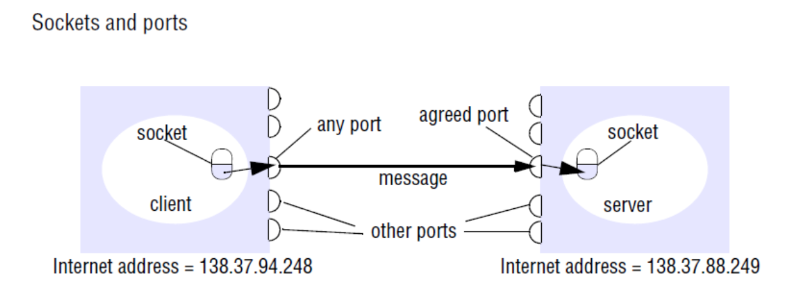
**B. Unreliable Communication**

* Messages **may be lost** if even a single packet is dropped.
* Used in applications where speed is prioritized over guaranteed delivery.
* **Example:**
  + **UDP (User Datagram Protocol)** used in online gaming and live streaming.

**5. Message Ordering**

* Some applications require messages to be **delivered in the same order they were sent**.
* If messages are received **out of order**, it is considered a failure for such applications.
* **Example:**
  + **Bank transactions** (if a withdrawal is processed before a deposit, it can cause errors).

**Sockets and Inter-Process Communication**



**1. Sockets Overview**

* **Sockets** provide an **endpoint** for communication between processes using **UDP or TCP**.provide a single long lived connection bw server and client -> chatapp where mssg and notification are received without reloading
* Communication happens by **transmitting a message** between a socket in one process and a socket in another process.
* A **socket must be bound** to a **local port** and an **Internet address** to receive messages.
* Only a process whose socket is **associated with the correct Internet address and port number** can receive messages sent to that address.
* **Java API:**
  + Java provides the InetAddress class to represent **Internet addresses**, allowing interaction with IP-based networks.

**2. UDP Datagram Communication**

**A. Characteristics of UDP Communication**

A datagram sent by UDP is transmitted from a sending process to a receiving process without acknowledgement or retries. One sendzs it other receives it

* **Connectionless:** No handshake between sender and receiver.
* **No Acknowledgments or Retries:** If a message is lost, it is not resent.
* **Unreliable:** Messages may be lost or arrive out of order.
* **Faster than TCP:** No overhead of establishing and maintaining a connection.

**B. Sending and Receiving Messages**

1. **creates a socket** and binds it to an **Internet address of localhost** and local **port**.
2. **Server binds its socket** to a **known port** so clients can send messages to it.
3. **Receive Method:**
   * Returns the **Internet address** and **port** of the sender + the message.
   * Allows the recipient to **send a reply** back.

**3. Issues in Datagram Communication**

**A. Message Size**

* The receiving process **must allocate a byte array** of a specific size for incoming messages.
* If a message is **larger than the allocated size**, it must be fragmented into smaller chunks.

**B. Blocking Behavior**

* **Send()** → Usually **non-blocking** (process continues after sending the message).
* **Receive()** → Usually **blocking** (process waits until a message arrives).
* Some implementations support **non-blocking receive** as an option.

**C. Timeouts**

* If a message does not arrive within a **specified timeout interval**, the receive operation fails.
* Setting an appropriate **timeout is challenging**; it should be significantly larger than the expected message transmission time.

**D. Receiving from Any Source**

* The **receive() method** does not specify a sender.
* It **receives messages from any sender** that sends data to the bound socket.

**4. UDP Failure Model**

**A. Omission Failures**

* **Messages may be dropped** due to:
  + **Checksum errors** (corrupted data).
  + **Lack of buffer space** at source or destination.

**B. Ordering Issues**

* Messages **can arrive out of order** due to network delays or dropped packets.
* Unlike TCP, UDP **does not guarantee in-order delivery**.

**5. Use Cases of UDP**

UDP is used when **speed** is prioritized over **reliability** and occasional message loss is acceptable.

| **Application** | **Why Use UDP?** |
| --- | --- |
| **DNS (Domain Name System)** | Fast lookups; occasional packet loss is acceptable. |
| **Voice over IP (VoIP)** | Low latency needed; minor data loss does not affect conversation. |
| **Online Gaming** | Fast real-time communication; minor packet loss is tolerable. |
| **Video Streaming** | Speed is more important than reliability; minor quality loss is okay. |

1. **There are three main sources of overhead:**

the need to store state information at the source and destination

the transmission of extra messages;

Latency for Sender: Sender waits for acknowledgments, increasing response time.

Since UDP lacks these mechanisms, it is **faster** but also **less reliable**.

* **TCP (Transmission Control Protocol)** provides **reliable, ordered, and error-checked** communication between processes.
* Data is sent in a **continuous stream**, unlike UDP, which sends discrete messages (datagrams).
* **Connection-oriented:** Before communication begins, a **connection** must be established between the sender and receiver.

**2. Message Sizes**

* Applications can **choose** how much data they **write** to or **read** from a stream.
* Data may be **small (few bytes) or large (megabytes)**.
* TCP **decides** how much data to collect before transmitting it as **one or more IP packets**.
* On arrival, the data is delivered **as requested by the application**.
* Applications can **force immediate sending** if needed (e.g., using flush() in Java).

**3. Reliability Features**

**A. Lost Messages Handling**

* TCP uses an **acknowledgment scheme** to ensure **reliable delivery**.
* Example of a simple (not TCP) acknowledgment scheme:
  + **Sender keeps a record** of sent packets.
  + **Receiver acknowledges** received packets.
  + If no acknowledgment is received **within a timeout**, the sender **retransmits** the message.

**B. Flow Control**

* TCP **matches the speed** of processes reading from and writing to a stream.
* If the **sender is faster** than the receiver, the sender is **blocked** until the receiver consumes enough data.

**C. Message Duplication & Ordering**

* **Message identifiers** are assigned to packets:
  + **Duplicates are detected &rejected**.
  + **Messages are reordered** if they arrive out of order.

**4. Connection Establishment**

* **Before communication, a connection must be established.**
* A connection follows this sequence:
  1. **Client sends a connect request** to the server.
  2. **Server accepts the request** and responds.
  3. A **two-way connection** is established.
* **Overhead Concern:**
  1. For **single client-server interactions**, the connection process may introduce **extra latency** compared to UDP.

**5. Issues in Stream Communication**

**A. Matching of Data Items**

* **Both processes must agree** on data formats.
* Example:
  + If **Sender writes an int followed by a double**,
  + The **Receiver must read an int followed by a double** to correctly interpret the data.

**B. Blocking Behavior**

* Data written to a **TCP stream** is stored in a **queue at the destination socket**.
* If a process tries to **read** from an **empty queue**, it **blocks** (waits) until data is available.

**C. Multithreading in Servers**

* When a **server accepts a client connection**, it **creates a new thread** for the client.
* **Advantages of multithreading:**
  + The server can **wait for client input** **without blocking** other clients.
  + Allows multiple clients to communicate **simultaneously**.

**6. TCP Failure Model**

TCP ensures **data integrity and reliability** through:

**A. Integrity Property (Detecting Corrupt & Duplicate Packets)**

* Uses **checksums** to detect and reject **corrupt packets**.
* Uses **sequence numbers** to detect and discard **duplicate packets**.

**B. Validity Property (Ensuring Delivery)**

* Uses **timeouts & retransmissions** for **lost packets**.
* Guarantees **eventual delivery** of all messages **even if some packets are lost**.

**7. Common Applications Using TCP**

| **Service** | **Description** |
| --- | --- |
| **HTTP (HyperText Transfer Protocol)** | Used for web browsing (e.g., loading websites). |
| **FTP (File Transfer Protocol)** | Used for file transfers. |
| **Telnet** | Used for remote command-line access. |
| **SMTP (Simple Mail Transfer Protocol)** | Used for email transmission. |

**Why Do We Need External Data Representation?**

* When sending data between computers, it must be converted into a format that both sender and receiver understand.
* Different computers store data (e.g., integers, floating-point numbers, characters) in different ways.
* Data must be **flattened (serialized)** before transmission and **rebuilt (deserialized)** on arrival.

**Key Issues in Data Representation**

1. **Byte Order (Endianness)**
   * **Big-Endian:** Most significant byte comes first (e.g., used by Motorola processors).
   * **Little-Endian:** Least significant byte comes first (e.g., used by Intel processors).
   * Example: The number **0x1234** is stored as:
     + **Big-Endian:** 12 34
     + **Little-Endian:** 34 12
   * Computers must convert values to a standard format before sending.
2. **Character Encoding**
   * **ASCII:** Uses 1 byte per character (common on UNIX systems).
   * **Unicode:** Uses 2 bytes per character (supports multiple languages).
   * If two systems use different character encodings, they must agree on one format.

**How to Enable Data Exchange?**

There are two common methods:

1. **Agreed External Format:**
   * Convert values to a standard format before transmission.
   * The receiver converts them back to its local format.
   * If both computers are the same type, conversion may be skipped.
2. **Sender's Format with Metadata:**
   * Send values in the sender’s format along with metadata describing the format.
   * The receiver converts the values only if necessary.

An **external data representation (EDR)** is a standardized way to represent data for transmission.

**Marshalling and Unmarshalling**

* **Marshalling:**
  + Converts structured data into a format suitable for transmission.
  + Involves serializing primitive values and complex objects.
* **Unmarshalling:**
  + Converts received data back into its original form.
  + Involves deserializing primitive values and rebuilding data structures.

**Three Approaches to External Data Representation**

1. **CORBA Common Data Representation (CDR)**
   * Used in CORBA (Common Object Request Broker Architecture).
   * Defines an external format for structured and primitive types.
   * Supports multiple programming languages.
2. **Java Object Serialization**
   * Used only in Java.
   * Converts Java objects into a byte stream for transmission or storage.
   * Can serialize a single object or an entire tree of objects.
3. **XML (Extensible Markup Language)**
   * A text-based format for structured data.
   * Originally for web documents but now used for client-server communication.
   * Self-descriptive and human-readable but has higher processing overhead.

**Multicast Communication – Simplified Notes**

**What is Multicast Communication?**

* **Unicast:** Sends a message to a single receiver.
* **Multicast:** Sends a message **simultaneously** to multiple receivers (a group of processes).
* **Broadcast:** Sends a message to **all** processes in a network.

Multicast is useful when a process needs to send the same message to multiple recipients without knowing their individual addresses.

**Why Use Multicast?**

Multicast is useful for distributed systems with the following needs:

1. **Fault Tolerance with Replicated Services**
   * A service is replicated across multiple servers.
   * A client **multicasts** a request to all replicas.
   * Even if some replicas fail, others can still respond.
2. **Service Discovery in Dynamic Networks**
   * Used in **spontaneous networking**, where devices need to discover services.
   * Example: A printer **multicasts** its availability, so computers can find it.
3. **Faster Performance with Replicated Data**
   * Data is replicated across different computers for faster access.
   * When data changes, updates are **multicasted** to all replicas.
4. **Event Notifications (Publish-Subscribe Model)**
   * **Multicast** allows event updates to be sent to multiple subscribers.
   * Example: Facebook status updates are sent to all friends.

**IP Multicast – How It Works**

* **IP multicast** allows a process to send a single **IP packet** to multiple computers.
* The sender does not need to know the number or identities of receivers.
* Uses a **Class D IP address** (range **224.0.0.0 to 239.255.255.255**) to identify a multicast group.

**Multicast Group**

* A **set of computers** that receive multicast messages.
* Membership is **dynamic** (computers can **join** or **leave** at any time).
* A computer can **belong to multiple groups** at once.

**Multicast Routers**

* **Local multicast:** Uses Ethernet's multicast feature.
* **Internet multicast:** Uses **multicast routers** to forward packets to different networks.

**Multicast Address Ranges (Class D)**

| **Address Range** | **Usage** |
| --- | --- |
| **224.0.0.0 - 224.0.0.225** | Local network multicast |
| **224.0.1.0 - 224.0.1.225** | Internet-wide multicast |
| **224.0.2.0 - 224.0.255.0** | Ad hoc multicast |
| **239.0.0.0 - 239.255.255.255** | Restricted multicast |

**Remote Invocation (RPC & RMI)**

**What is Remote Invocation?**

* In distributed systems, processes **communicate remotely** to request services.
* **Request-Reply Protocols** provide structured communication.

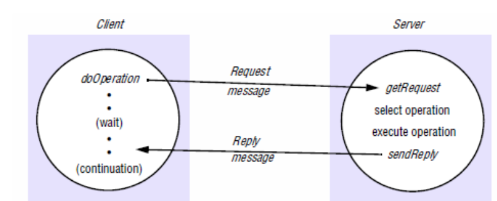
**Types of Remote Invocation**

1. **Remote Procedure Call (RPC)**
   * Allows a client to call a function on a remote server **as if it were local**.
   * The client **sends a request**, the server **processes it**, and the client **waits for a reply**.
2. **Remote Method Invocation (RMI)**
   * Similar to RPC but specifically for **object-oriented programming**.
   * Supports calling methods on remote objects.

**How Request-Reply Works?**

1. The **client** sends a **request** to the server.
2. The **server** processes the request.
3. The **server** sends a **reply** back to the client.
4. The **client blocks (waits)** until the reply is received.

Since the communication is **synchronous**, the client must wait for the server to respond before proceeding.



**Introduction to RPC**

1. **What is RPC?**
   * RPC (Remote Procedure Call) allows a program to call a procedure (function) on another machine as if it were a local function.
   * It hides the complexities of network communication from the programmer.
2. **How does RPC work?**
   * Process A (on machine 1) calls a function on Process B (on machine 2).
   * Process A **waits** (gets suspended) while the function executes on B.
   * Data is sent back to A as a return value.
3. **Example of RPC**

c

CopyEdit

result = add(10, 20); // Called on remote machine

**Conventional Procedure Call vs RPC**

1. **Conventional (Local) Procedure Call:**
   * Function executes **on the same machine**.
   * Example:

c

CopyEdit

count = read(fd, buf, bytes);

* + Uses **stack** for parameter passing.

1. **RPC Procedure Call:**
   * Function executes **on a remote machine**.
   * Parameters are sent over the **network**.

**RPC Semantics (Rules)**

1. **Transparency in RPC**
   * The caller should not need to know whether the function is local or remote.
2. **How RPC Works in Background**
   * The function call is **converted into a message**.
   * This message is sent over the network.
   * A **server stub** receives the message and calls the actual function.
   * The result is **sent back** as a response message.
3. **Example Flow of read() in RPC:**
   * Client calls read().
   * Instead of directly reading, a **client stub** creates a network message.
   * Server stub **receives the message** and calls actual read().
   * The result is packed into a message and sent back to the client.

**Client and Server Stubs in RPC**

1. **Client Stub:**
   * Acts as a **proxy** for the actual function on the client side.
   * Packs the function name and parameters into a message.
   * Sends it to the server.
2. **Server Stub:**
   * Waits for incoming requests.
   * Extracts the function name and parameters.
   * Calls the actual function and sends back the result.
3. **Example Steps in RPC:**
   * Client calls a function.
   * Client stub **packs** the data into a message.
   * Message is sent to the server.
   * Server stub **extracts** data and calls the function.
   * The function executes, result is packed and sent back.
   * Client stub **receives** the result and returns it to the caller.

**Parameter Passing in RPC**

1. **Marshalling:**
   * The process of **packing function parameters** into a message.
2. **Call-by-Value vs Call-by-Reference:**
   * **Call-by-Value**: Directly sends the value (e.g., x = 10).
   * **Call-by-Reference**: **Cannot send pointers** because memory addresses are different on client and server.
3. **Problems in Parameter Passing:**
   * Different computers use different **data formats** (e.g., ASCII vs EBCDIC).
   * Different systems store numbers in different ways (**big-endian vs little-endian**).
4. **Example of Parameter Passing Issue:**
   * A PC (little-endian) sends 100 as 01100100.
   * A SPARC system (big-endian) reads it as 10000000, causing errors.
   * **Solution**: Convert everything into a standard format.

**Stub Generation in RPC**

1. **Manual Stub Generation:**
   * Programmer **writes** conversion code manually.
2. **Automatic Stub Generation:**
   * Uses **Interface Definition Language (IDL)** to generate stubs.
   * Compiler creates **client and server stubs** automatically.
3. **Steps in Stub Generation:**
   * Define an **interface** in IDL.
   * Compile it using an **IDL compiler**.
   * The compiler generates:
     + **Client stub** (packs data).
     + **Server stub** (unpacks data and calls function).

**Server Management in RPC**

1. **Types of Servers:**
   * **Stateful Server:**
     + Remembers client data between requests.
     + Example: A banking system that remembers your session.
   * **Stateless Server:**
     + Does **not** store client data.
     + Example: A simple file server that reads and writes files.
2. **Server Creation Methods:**
   * **Instance-per-Call:** Server exists only **for one request**.
   * **Instance-per-Session:** Server stays **until the client session ends**.
   * **Persistent Server:** Server **always runs** and handles multiple clients.

**Dynamic Binding in RPC**

1. **What is Dynamic Binding?**
   * Instead of **hardcoding** the server address, the client finds the server dynamically.
2. **How Dynamic Binding Works?**
   * A special **binder** keeps track of all available servers.
   * The client **requests** a server from the binder.
   * The binder provides the **server address** dynamically.
3. **Advantages of Dynamic Binding:**
   * Load balancing (distributes clients evenly).
   * Automatic failure detection (removes crashed servers).

**RPC Failure Handling**

1. **What Can Go Wrong?**
   * Server **not found**.
   * Request message **lost**.
   * Server **crashes** before sending a response.
2. **Solutions:**
   * Use **timeouts** and **retries**.
   * Store client requests for **re-execution** in case of failure.
   * Use **checkpointing** to save progress.

**Conclusion**

* RPC allows **calling remote functions** as if they were local.
* Uses **client and server stubs** for communication.
* Handles **parameter passing issues** using marshalling.
* **Dynamic binding** makes server discovery flexible.
* Different **server models** (stateful, stateless) suit different applications.
* **Failure handling** ensures reliability.

**Real-Life Example of RPC**

* **Google Drive Sync**: When you open a file, your PC calls a function on Google’s server (RPC).
* **Online Banking**: When you transfer money, your request is sent to the bank’s server using RPC.

**SERVER MANAGEMENT (1) - Issues & Server Implementation**

**Issues in RPC-based Applications for Server Management**

1. **Handling Client Requests Efficiently:**
   * Servers should manage multiple clients without delays.
2. **Maintaining State Information:**
   * Some applications need to **remember past client interactions** (stateful).
3. **Scalability:**
   * How to manage **many clients** without performance issues.
4. **Server Creation Strategy:**
   * Whether to create a new server **for every request** or keep one running.

**SERVER MANAGEMENT (2) - Stateful Servers**

**What is a Stateful Server?**

* A **stateful server** remembers client information between calls.
* Used when **a client’s previous request affects future requests**.

**How it Works:**

1. **Client sends a request.**
2. **Server processes the request and stores the state (data).**
3. **For the next request, the server uses stored data.**

**Example:**

* **Online Banking:**
  + You log in, check balance, and transfer money.
  + The server **remembers** your login session and bank balance.

**SERVER MANAGEMENT (3) - Operations in Stateful Servers**

* The server **remembers** which files are open and their positions.

**SERVER MANAGEMENT (4) - Stateless Servers**

**What is a Stateless Server?**

* A **stateless server** does **not** remember client requests.
* Each request **must contain all necessary data**.
* The client is responsible for storing state information.

**Example:**

* **HTTP Web Servers:**
  + Every request (e.g., loading a webpage) is **independent**.
* The client must **specify the file position** every time.

**SERVER MANAGEMENT (5) - Server Creation Semantics**

**How are Servers Created?**

* Servers can be created **before clients** or **on-demand**.
* Based on survival duration, servers are classified into:
  1. **Instance-per-call servers**
  2. **Instance-per-session servers**
  3. **Persistent servers**

**SERVER MANAGEMENT (6) - Instance-per-Call Servers**

**What is an Instance-per-Call Server?**

* Created **only for a single RPC request**.
* **Deleted immediately** after handling the request.

**Example:**

* **A print server that prints one document and exits.**

**Disadvantages:**

* **Stateless** (cannot remember client requests).
* **Expensive** (creating a new server every time).

**SERVER MANAGEMENT (7) - Problems with Instance-per-Call Servers**

1. **High Overhead:**
   * Server must be created and destroyed for every request.
2. **No State Maintenance:**
   * The server cannot **remember previous client requests**.
3. **Alternative Solution:**
   * Let **clients** store state information or use **persistent servers**.

**SERVER MANAGEMENT (8) - Instance-per-Session Servers**

**What is an Instance-per-Session Server?**

* A server stays active **for the entire session** of a client.
* It can **remember state information**.

**Example:**

* **Video Streaming Service:**
  + When you play a video, the server stays active **until you close the player**.

**Advantages:**

✅ Less overhead than instance-per-call servers.  
✅ Maintains state information.

**SERVER MANAGEMENT (9) - Implementation of Instance-per-Session Servers**

1. **Server Manager** is created for each type of service.
2. **All server managers are registered with a binding agent.**
3. **Client requests a service** → Binding agent provides the **server address**.
4. **Client communicates with the server manager.**

**SERVER MANAGEMENT (10) - How Instance-per-Session Servers Work**

1. **Server Manager creates a new server for the client.**
2. **The client interacts with the server for the entire session.**
3. **When the session ends, the server is deleted.**

**Example:**

* A **database session** remains active while a user is logged in.

**SERVER MANAGEMENT (11) - Persistent Servers**

**What is a Persistent Server?**

* A **long-running server** that serves multiple clients.
* **Does not terminate** after a single request or session.

**Example:**

* **Google Drive**
  + The file server stays active and handles multiple user uploads.

**Advantages:**

✅ Shared by many clients.  
✅ Can store multiple clients' state information.  
✅ Efficient for services with frequent requests.

**SERVER MANAGEMENT (12) - Persistent Server Management**

1. Similar to **session-based servers**, but serves multiple clients.
2. Registers itself with the **binding agent**.
3. **Manages multiple client states simultaneously.**
4. Uses **load balancing** to distribute requests.

**Example:**

* **Cloud Services (AWS, Google Cloud, Microsoft Azure)**
  + A **single server** handles multiple client requests efficiently

**DYNAMIC BINDING (1) - How Client Locates a Server?**

**Hardwired Address Approach (Static Binding)**

* The client **stores the server's network address** permanently.
* **Problem:** If the server **changes its address**, the client won’t find it.

**Dynamic Binding Approach**

* Instead of **hardcoding**, the client **dynamically** finds a server.
* This allows **flexibility** if the server changes or multiple servers exist.

**DYNAMIC BINDING (2) - File Server Specification Example**

**Explanation:**

* A file server offers **read, write, create, and delete operations** dynamically.
* The client **does not hardcode a server address**; instead, it **searches** for a suitable server.

**DYNAMIC BINDING (3) - Binder Interface**

A **binder** is a program that keeps track of available servers.

**Binder Functions:**

| **Call** | **Input** | **Output** |
| --- | --- | --- |
| **Register** | Name, version, handle, unique ID | Confirms registration |
| **Deregister** | Name, version, unique ID | Confirms removal |
| **Lookup** | Name, version | Handle, Unique ID (if found) |

**Example:**

* A **server registers itself** with the binder.
* The **client asks** the binder for an available server.
* The binder **returns the server’s address** to the client.

**DYNAMIC BINDING (4) - Implementation for Stateless Servers (Part 1)**

**How It Works:**

1. A **formal specification** of the server is created.
2. A **stub generator** creates client and server stubs.
3. **Stubs are stored** in the client and server libraries.
4. The **client program calls a remote procedure**, and the stub links it to the correct server.
5. The **server stub** is linked when the server is compiled.

**Example:**

* If the client wants to **read a file**, it calls read(), and the stub **automatically finds a server** using dynamic binding.

**DYNAMIC BINDING (5) - Registering a Server with the Binder**

1. When the **server starts**, it **registers itself** with the **binder**.
2. The server sends the binder:
   * **Name** (e.g., file\_server)
   * **Version** (e.g., 3.1)
   * **Unique ID** (e.g., 32-bit identifier)
   * **Handle** (server address - IP, Ethernet, etc.)
3. If the server **wants to stop**, it can **deregister** from the binder.

**DYNAMIC BINDING (6) - Client Requests a Server**

1. When the client calls a function (e.g., read()), the **client stub checks** if it's bound to a server.
2. If **not bound**, the client **asks the binder** for a server with version 3.1.
3. The binder **searches** for a registered server.
4. If a matching server exists, the **binder returns the server’s handle** and unique ID.
5. If no server is found, the client gets an **error**.

**Example:**

* A user tries to **open a file** stored in a cloud.
* The client **contacts the binder** to find an available file server.

**DYNAMIC BINDING (7) - Client Uses the Server**

1. The **client stub uses the handle** from the binder to send requests.
2. The **message contains:**
   * Function parameters (e.g., file name, size).
   * The **unique ID** (to match the correct server).
3. If **multiple servers** exist, the request is sent to the correct one.

**Example:**

* If a **persistent server** (always running) is used, the client can send **multiple requests** without rebinding.

**DYNAMIC BINDING (8) - Advantages**

✅ **Handles Multiple Servers Efficiently:** If **many servers** exist, the binder **distributes clients** evenly.

✅ **Load Balancing:** Clients are **spread across servers** to prevent overload.

✅ **Fault Tolerance:** The binder **automatically removes servers** that stop responding.

✅ **Better Security & Authentication:** The binder **verifies servers** before assigning them to clients.

**DYNAMIC BINDING (9) - Disadvantages**

❌ **Extra Overhead:** The **client must request a server** before each operation.

❌ **Binder Bottleneck:**If the binder **gets overloaded**, performance may drop.

❌ **Complex Registration:** Servers must **register & deregister**, requiring extra management.

❌ **Binder Synchronization Needed:** All binders must be updated **if a server joins or leaves**.

**1. What is RMI?**

Remote Method Invocation (RMI) is a Java feature that allows one Java program to invoke methods on an object located on another machine (remotely). It makes remote method calls look like local method calls.

**2. How RMI Works?**

1. **Client requests a remote object** from the server.
2. **Server processes the request** and executes the method.
3. **Result is sent back** to the client.

**Example:**  
A client application calls a method getBalance() on a remote bank server. The server calculates the balance and sends the result back.

**3. Goals of RMI**

* Allow communication between objects in different Java Virtual Machines (JVMs).
* Support **callbacks** (server calling client methods).
* Make distributed programming easier than using **sockets**.
* Provide **secure execution** within Java's runtime environment.
* Support **multiple transport mechanisms** like TCP, UDP.

**4. Key Components of RMI**

| **Component** | **Description** |
| --- | --- |
| **Client** | The application that requests a remote method execution. |
| **Server** | The system that has the remote object and executes the requested method. |
| **Stub** | A proxy object on the client side that communicates with the remote object. |
| **Skeleton** | A proxy on the server side (not needed in JDK 1.2+). |
| **Registry** | A directory where remote objects are registered. Clients use this to look up objects. |

**5. RMI Architecture**

| **Layer** | **Purpose** |
| --- | --- |
| **Stub/Skeleton Layer** | Handles method invocation from client to server. |
| **Remote Reference Layer** | Supports different remote invocation techniques (unicast, multicast). |
| **Transport Layer** | Manages network communication between JVMs. |

**Example Workflow:**

1. Client calls a method on a **Stub**.
2. Stub sends request to server **Skeleton**.
3. Skeleton calls actual method on the server object.
4. The result is sent back to the client.

**6. Using the RMI Registry**

* The **server registers the remote object** with a name.
* The **client looks up the remote object** by name to use it.
* **Example:**

java

CopyEdit

Registry registry = LocateRegistry.getRegistry("localhost", 1099);

RemoteObject obj = (RemoteObject) registry.lookup("BankService");

**7. Parameter Marshalling & Unmarshalling**

When sending data over the network:

* **Marshalling:** Convert method parameters into a stream of bytes for transmission.
* **Unmarshalling:** Convert received bytes back into method parameters.

**Example:**

* Client calls transferMoney(500, "A123").
* RMI converts 500 and "A123" into bytes and sends them to the server.
* The server reads the bytes, reconstructs the values, and processes the request.

**8. RMI in Action - Step by Step**

**Client Side:**

1. Calls the stub method.
2. Stub establishes a connection with the server.
3. Sends marshalled parameters.
4. Waits for the result.
5. Receives and unmarshals the result.

**Server Side:**

1. Receives marshalled parameters.
2. Locates the remote object.
3. Calls the actual method.
4. Marshals and returns the result.