Remote Procedure Call (RPC) Summary

Concept of RPC

- Enables calling procedures on remote machines as if they were local.
- Abstracts complexities of distributed systems.

Design Issues for RPC

- **Programming with Interfaces**
  - Modules use explicit interfaces to define accessible procedures.
  - Service interfaces allow communication without knowing implementation details.
  - **Benefits:**
    - Abstraction from implementation.
    - Independence from language/platform.
    - Supports software evolution.
- **Interface Definition Languages (IDLs)**
  - Define interfaces for different programming languages.
  - Ensure remote invocation by specifying input/output parameters.
  - Examples: Sun XDR, CORBA IDL, WSDL, Protocol Buffers.

RPC Call Semantics

- **Maybe Semantics**: Procedure may execute once or not at all (no fault tolerance).
- **At-least-once Semantics**: Procedure executes at least once, may cause multiple executions.
- **At-most-once Semantics**: Procedure executes exactly once or fails safely.

Transparency in RPC

- **Aims to resemble local procedure calls:**
  - **Location Transparency**: Hides remote procedure's location.
  - **Access Transparency**: Local and remote procedures appear the same.
- **Challenges:** Remote calls are more failure-prone due to network dependencies.

Latency and Parameter Passing

- Remote calls have higher latency than local calls.
- **No call-by-reference support** → Requires explicit parameter passing.

Implementation of RPC

Software Components

- **Client Process:**
  - Contains client stubs for service procedures.
  - Stubs marshal/unmarshal data and communicate with the server.
- **Server Process:**
  - Includes dispatcher, server stubs, and service procedures.
  - Dispatcher selects the right procedure, unmarshals data, and processes requests.
- **Interface Compiler:**
  - Automatically generates stubs and dispatcher from service interface definition.

Communication and Protocols

- Uses a **request-reply protocol** for structured message exchange.

- Ensures message adherence to request-reply formats.

Invocation Semantics

- Supports **at-least-once** or **at-most-once** invocation models.
- The communication module handles:
  - Request retransmissions.
  - Duplicate message detection.
  - Result retransmission.

UDP & TCP Communication Summary

UDP Datagram Communication

- **Connectionless protocol**: No acknowledgments or retries, messages may be lost.
- **Message Size**: Messages exceeding buffer size are truncated.
- **Blocking**: Non-blocking sends, blocking receives (messages are queued).
- **Timeouts**: Prevent indefinite blocking with socket timeouts.
- **Receive from Any**: Messages can be received from any sender.

Failure Model for UDP

- Messages may be lost due to checksum errors or buffer limitations.
- Messages may arrive out of order.
- Applications must handle reliability using acknowledgments.

Use Cases of UDP

- Suitable for **DNS, VoIP**, and real-time applications where occasional message loss is acceptable.
- Lower overhead than TCP, as it does not maintain state.

Java API for UDP

- **DatagramPacket**: Represents a datagram (message, address, and port).
- **DatagramSocket**: Supports sending, receiving, and setting timeouts.

TCP Stream Communication

- Provides **byte stream abstraction**, handling message size, loss, and flow control.
- Data can be sent and received in any size without concern for packet boundaries.

Key Characteristics of TCP

- **Message Sizes**:
  - TCP decides when and how much data to send.
  - Applications can force immediate transmission (e.g., `flush()`).
- **Lost Messages**:
  - Uses acknowledgments and retransmissions for reliable delivery.
  - Sliding window mechanism improves efficiency.
- **Flow Control**:
  - Prevents sender from overwhelming receiver.
- **Message Ordering**:
  - Sequence numbers ensure in-order delivery and detect duplicates.

- **Connection-Oriented**:
  - Requires connection establishment before communication.
  - Overhead makes it less efficient for short-lived interactions.

TCP Sockets

- **ServerSocket**: Listens for incoming connections.
- **Socket**: Represents a connection between client and server.
- **Each socket has input and output streams** for reading and writing data.

Closing TCP Connections

- Closing a socket sends remaining data before termination.
- If a process crashes, sockets close automatically.

Issues with TCP Streams

- **Data Matching**: Sender and receiver must follow the same data format.
- **Blocking**:
  - Reading an empty stream blocks the process.
  - Flow control may block the sender if the receiver is slow.
- **Threads**: Servers typically use multiple threads to handle clients.
  - Alternatives: `select()` (UNIX) for handling multiple connections without threading.

TCP Failure Model

- Uses **checksums, sequence numbers, and retransmissions** for reliability.

- Cannot guarantee delivery in severe network failures.
- Cannot distinguish between network failures and process failures.

Common Uses of TCP

- **HTTP** (Web browsing)
- **FTP** (File transfers)
- **Telnet** (Remote terminal access)
- **SMTP** (Email transmission)

Java API for TCP

- **ServerSocket**: Accepts incoming connections.
- **Socket**: Establishes a connection to a server.
- **Streams**: `getInputStream()`, `getOutputStream()` for communication.
- **DataInputStream & DataOutputStream**: Read/write binary data.

Example Pattern

- **Client** connects to server → Sends message → Receives response.
- **Server** listens → Accepts connection → Processes requests.
- Abstracts network complexities, making TCP communication seamless.

Characteristics of Interprocess Communication (IPC)

Message Exchange

- Involves **send** and **receive** operations.
- Messages include **data** and **destination** (Internet address & port).

- May require **synchronization** between processes.

Synchronous vs. Asynchronous Communication

- **Synchronous:**
  - Sender and receiver **block** until message exchange completes.
- **Asynchronous:**
  - Sender continues execution after sending.
  - Receiver can use **blocking or non-blocking** receive.
  - **Multi-threading** simplifies synchronization in blocking mode.

Message Destinations

- Sent to **(Internet address, local port)** pairs.
- Each **port has one receiver** but **many senders**.
- **Servers** advertise port numbers for client communication.
- **Location transparency** via **name servers or binders** (no runtime migration).

Reliability

- **Validity**: Ensures message delivery despite packet loss.
- **Integrity**: Ensures messages **arrive uncorrupted** without duplication.

Ordering

- Some applications require **messages to be delivered in order**.
- **Out-of-order messages** are considered a failure in such cases.

Sockets

- **Endpoints for communication** in both UDP and TCP.
- Each socket has an **Internet address & port number**.
- **Same socket** can be used for both sending and receiving.
- **IP multicast** allows port sharing among multiple processes.

Peer-to-Peer (P2P) Architecture Overview

Key Characteristics

- **Decentralized Structure**: No distinction between clients and servers; all nodes act as both.
- **Uniform Software & Interfaces**: Every node runs the same program and offers the same functionality.

Advantages

- **Scalability**: System grows stronger as more users join.
- **Resource Utilization**: Uses users' computing power, storage, and bandwidth.
- **Load Distribution**: Data and processing are spread across multiple computers, preventing bottlenecks.

Technological Evolution

- **Modern PCs**: High-performance machines with always-on broadband connections make P2P feasible.
- **Resource Sharing**: Large-scale pooling of computing resources for data access and processing.

Examples

- **Napster**: Early P2P system for music sharing, though it relied on a central index server.

- **BitTorrent**: Decentralized file-sharing system that improves efficiency by distributing files in smaller chunks among peers.

Challenges

- **Complexity**: Managing data distribution, placement, and replication is more complicated than in client-server models.
- **Data Replication**: Ensures reliability but increases management difficulty.
- **Security Risks**: Nodes must safeguard data while allowing access to distributed peers.

Service & Data Placement

- **Performance Optimization**: Proper placement enhances speed, reliability, and security.
- **Dynamic Considerations**: Placement strategies depend on network conditions, load balancing, and failure tolerance.

Conclusion

P2P architectures provide a **scalable, distributed** solution for resource-sharing but require careful design for **data replication, load balancing, and security** to function effectively.