MODULE 2

Characteristics of Interprocess Communication (IPC)

Interprocess communication involves message exchange between processes using the operations send and receive. These operations involve two key components: the message and the destination. The message is transmitted from the sending process to the receiving process, and this may include synchronization between the processes.

Synchronous and Asynchronous Communication

In **synchronous communication**, both sending and receiving processes synchronize at each message exchange. The send and receive operations are **blocking**; the sender is blocked until the receiver issues a corresponding receive, and the receiver is blocked until a message arrives.

In **asynchronous communication**, the send operation is **non-blocking**, meaning the sender can continue execution after the message is placed in a buffer. The receive operation has both blocking and non-blocking variants. Non-blocking receive allows the process to continue while the system fills the buffer, but the process must check separately if the buffer is full. In systems like Java, with multi-threading, blocking receive can be used without issues since other threads remain active, simplifying synchronization.

Message Destinations

Messages in a distributed system are sent to a specific (Internet address, local port) pair. A **local port** is a message destination within a computer, specified by an integer. Each port can have one receiver and many senders. Servers typically publicize their port numbers to allow clients to send messages. **Location transparency** can be achieved using name servers or binders, translating service names to server locations at runtime, though this does not allow migration while the system is running.

Reliability

Communication reliability is defined by two properties: **validity** and **integrity**. A service is reliable if messages are guaranteed to be delivered despite some packet loss. For integrity, messages must arrive uncorrupted and without duplication.

Ordering

Some applications require that messages be delivered in the order they were sent. If messages arrive out of order, it is considered a failure for such applications.

Sockets

Sockets provide an endpoint for communication between processes. They are a key abstraction in both **UDP** and **TCP** communication, originating from BSD UNIX and used in most operating systems. Each socket is associated with an Internet address and port number, and messages are exchanged between sockets. Processes can use the same socket for both

sending and receiving messages, but a port cannot be shared by multiple processes, except in cases of IP multicast.

Java offers the InetAddress class to represent Internet addresses, allowing programmers to work with DNS hostnames to retrieve addresses. This class encapsulates the representation of Internet addresses, supporting both IPv4 (4 bytes) and IPv6 (16 bytes).

UDP Datagram Communication

UDP (User Datagram Protocol) is a connectionless communication protocol. It sends datagrams between processes without requiring acknowledgment or retries, meaning messages may be lost if errors occur. A server binds its socket to a known port, and clients bind to any free local port. The receive method returns both the message and the sender's Internet address and port.

Key issues with UDP communication include:

- **Message Size**: The receiving process must provide an array of bytes to receive the message. If the message exceeds the array size, it will be truncated.
- **Blocking**: UDP allows non-blocking sends and blocking receives. Messages are placed in a queue at the destination port and can be retrieved by receive.
- **Timeouts**: To prevent indefinite blocking in cases of failure, sockets can have a timeout, after which the receive method will throw an exception.
- **Receive from Any**: The receive method retrieves a message from any origin, providing the sender's address and port.

Failure Model for UDP

UDP datagrams can suffer from omission failures, where messages are dropped due to checksum errors or buffer space limitations. Additionally, messages may be delivered out of order. Applications must handle such failures to ensure reliable communication, often using acknowledgments.

Use of UDP

UDP is suitable for applications like **DNS** and **VoIP**, where occasional message loss is acceptable. The advantage of UDP lies in its lower overhead compared to TCP, as it does not store state information or send extra messages for reliability.

Java API for UDP Datagram Communication

The Java API provides two key classes for UDP communication:

- **DatagramPacket**: This class represents a datagram, containing an array of bytes for the message, its length, the Internet address, and the destination port. It provides methods to retrieve the message data, sender's port, and Internet address.
- **DatagramSocket**: This class supports the sending and receiving of UDP datagrams. It provides methods such as send, receive, and setSoTimeout for communication. Sockets can also be connected to a specific remote address and port for more controlled communication.

TCP Stream Communication

TCP (Transmission Control Protocol) provides a **stream of bytes** abstraction, hiding various complexities of network communication. This abstraction allows data to be written to and read from the stream without worrying about underlying network conditions, such as message size, packet loss, or flow control.

Key Characteristics of TCP Stream Communication

1. **Message Sizes**:
    - The application can write or read any amount of data to or from the stream, whether large or small.
    - The TCP implementation decides how much data to collect before transmitting it in one or more IP packets. It may collect multiple messages into one packet.
    - Applications can force data to be sent immediately if needed (using mechanisms like flush() in some APIs).
2. **Lost Messages**:
    - TCP uses an **acknowledgement scheme** to ensure message delivery.
    - If a message is lost, it is retransmitted after a timeout if no acknowledgement is received.
    - A more sophisticated **sliding window scheme** reduces the number of acknowledgements needed, improving efficiency.
3. **Flow Control**:
    - TCP adjusts the rate of data transmission between processes.
    - If the receiving process is slower, TCP temporarily blocks the sender from writing more data until the receiver catches up.
4. **Message Duplication and Ordering**:
    - TCP assigns **sequence numbers** to each packet, enabling the receiver to detect duplicates or reorder packets that arrive out of order.
    - TCP ensures that messages are delivered in the order they were sent.
5. **Message Destinations**:
    - Before communication can begin, two processes must establish a **TCP connection**.
    - The client sends a connection request to the server, which the server must accept.
    - Once the connection is established, the processes can read and write data to the stream without worrying about specific Internet addresses or ports.

- Connection setup introduces overhead, making TCP less efficient for short-lived interactions compared to connectionless protocols like UDP.

6. **Sockets in TCP**:
    - In TCP communication, a **socket** represents one end of the communication link.
    - The server creates a **listening socket** to accept incoming connection requests from clients.
    - Once a connection is accepted, a new socket is created for communication between the client and the server.
    - Each socket has two streams: one for reading input and another for writing output.

Closing TCP Connections

When an application finishes sending data, it closes its socket. This signals the end of data transmission to the other end, and any remaining data in the output buffer is sent before the connection is fully terminated. After the socket is closed, any further attempts to read from the stream will result in an end-of-stream indication.

If a process exits or crashes, its sockets are closed automatically, and any attempt to communicate with it will fail. TCP provides notifications of broken connections, but it cannot distinguish between network failures and process failures at the other end.

Issues with TCP Streams

1. **Data Matching**:
    - Communicating processes must agree on the data format. For example, if one process sends an integer followed by a string, the receiver must read the data in the same order (int first, then string).
    - Mismatched data formats can cause errors or deadlocks due to insufficient data in the stream.
2. **Blocking**:
    - If there is no data in the socket's queue when a process attempts to read from the stream, the process is blocked until data becomes available.
    - Similarly, if the sender writes data faster than the receiver can process it, TCP's flow control will block the sender to prevent overwhelming the receiver.
3. **Threads**:
    - In server applications, a separate thread is usually created for each client connection. This allows the server to handle multiple clients simultaneously without blocking.

- If threading is unavailable, the server can use mechanisms like **select()** (in UNIX) to check for available input before attempting to read.

TCP Failure Model

TCP provides reliability through several mechanisms:

- **Checksums** detect and reject corrupted packets.
- **Sequence numbers** detect and reject duplicate packets.
- **Timeouts** and **retransmissions** ensure that messages are resent if they are lost.

However, TCP cannot guarantee delivery under all circumstances. If the network is severely congested or a connection is severed, TCP will eventually declare the connection broken after failing to receive acknowledgements. This limits TCP's reliability in extreme failure scenarios.

Additionally, processes cannot distinguish between network failures and failures of the process at the other end. They also cannot tell whether the messages they recently sent were received.

Common Uses of TCP

TCP is widely used in many internet services, including:

- **HTTP**: Web browsers and servers use HTTP over TCP to communicate.
- **FTP**: Allows file transfers between computers.
- **Telnet**: Provides terminal access to remote computers.
- **SMTP**: Used for sending emails between computers.

Java API for TCP Streams

Java provides classes to support TCP communication:

1. **ServerSocket**:
    - Used by the server to create a listening socket at a specified port.
    - Its accept() method retrieves a connection request from a queue, blocking if no request is available. Upon success, it returns a new **Socket** for communicating with the client.
2. **Socket**:
    - A client uses the Socket constructor to create a socket and connect to a server. This constructor initiates the connection and can throw exceptions like UnknownHostException (if the hostname is incorrect) or IOException (if an input/output error occurs).
    - A socket provides access to **input and output streams** via getInputStream() and getOutputStream(), which return abstract classes for reading and writing data.

- In Java, **DataInputStream** and **DataOutputStream** can be used for reading and writing binary data over the streams.

Example of TCP Communication

A TCP client connects to a server at a specific port, sends a message, and receives a response. This simple pattern is widely used in client-server communication, with the client and server exchanging messages over their respective streams.

This encapsulates TCP communication in a user-friendly way, abstracting many of the complexities of the underlying network operations.

External Data Representation and Marshalling

In distributed systems, data structures in programs need to be converted into sequences of bytes for transmission. This process involves **marshalling** (flattening data for sending) and **unmarshalling** (reconstructing data upon receipt).

Key Challenges in Data Representation

1. **Byte Ordering (Endianness)**: Different systems use either **big-endian** (most significant byte first) or **little-endian** (least significant byte first).
2. **Character Encoding**: Systems might use **ASCII** (1 byte per character) or **Unicode** (2 bytes per character) for text data.
3. **Floating-Point Representation**: Different architectures may store floating-point numbers in varying formats.

Methods of Exchanging Binary Data

1. **External Format Conversion**: Convert data into a standard format before sending and convert it back on receipt.
2. **Sender's Format with Indicator**: Send data in the sender's format along with an indicator, allowing the receiver to adjust if needed.

Approaches to Data Representation and Marshalling

1. **CORBA CDR**: Defines a standard for marshalling structured and primitive data types for Remote Method Invocations (RMI), supporting both byte orders.
2. **Java Object Serialization**: Java-specific, flattens objects for transmission and includes type information.
3. **XML**: A textual format used for representing structured data, especially in web services. It is more readable but less compact than binary formats.

Alternatives

1. **Protocol Buffers**: Google's lightweight, efficient format for transmitting structured data.
2. **JSON**: A simple, lightweight format for transmitting structured data, commonly used in web services.

Java Object Serialization

In Java RMI (Remote Method Invocation), objects and primitive data values can be passed as arguments and results of method invocations. An object is an instance of a Java class. To enable an object to be serialized (converted into a byte stream for storage or transmission), it must implement the Serializable interface, which is part of the java.io package.

Serialization is the process of flattening an object, or a connected set of objects, into a serial form. This format is suitable for saving on disk or transmitting through a network. Deserialization restores the object's state from its serialized form. The process assumes that the deserializing system does not have prior knowledge of the object's type, so information about the object's class, including the class name and version number, is included in the serialized form.

The version number helps ensure compatibility when objects undergo changes. The version can be set manually or automatically generated based on a hash of the class name, instance variables, methods, and interfaces.

When an object is serialized, any referenced objects are also serialized to maintain the object's structure when reconstructed. References to objects are serialized as handles, ensuring that each object is serialized once, and any subsequent references to that object use the same handle.

For example, to serialize an object like:

java

Copy code

Person p = new Person("Smith", "London", 1984);

You would use ObjectOutputStream to serialize and ObjectInputStream to deserialize the object. Serialization of arguments and results in remote invocations is generally automatic, though developers can customize serialization if needed.

Reflection in Java enables generic serialization and deserialization, without needing to create special marshalling functions for each object type. This is done by inspecting an object’s class and its instance variables dynamically.

Extensible Markup Language (XML)

XML is a markup language defined by the World Wide Web Consortium (W3C) and is commonly used for structured data exchange on the web. Unlike HTML, which is designed for displaying web pages, XML describes the logical structure of data through tags and attribute-value pairs.

Tags define the structure and meaning of data, while attributes provide additional information about elements. XML allows users to define their own tags and is extensible, meaning different applications can use agreed-upon tags for communication. XML documents can be read by humans and machines, which makes XML a popular format for web services and data interchange, despite its verbosity compared to binary formats.

An XML document must be well-formed, with matching tags and proper nesting, and contain a single root element. The prolog at the beginning of an XML document specifies the version and encoding (e.g., UTF-8).

Namespaces in XML prevent naming conflicts by associating tags with a specific URL, allowing multiple XML documents to work together without interference. XML schemas further define the structure and data types that can appear in an XML document and are used to validate the document's conformance.

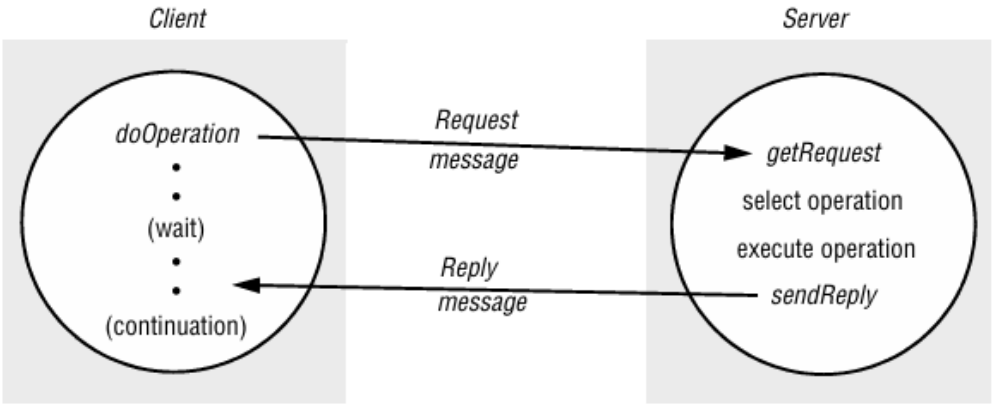
Remote Object References

In distributed systems like Java RMI and CORBA, remote object references are unique identifiers for remote objects. These references allow a client to specify which remote object to invoke. To ensure uniqueness across a distributed system, remote object references include information such as the Internet address of the host machine, the port number of the process, the creation time, and a local object number.

These references remain valid even if the object has been deleted, ensuring that invocations on obsolete references produce errors rather than accessing incorrect objects.

Request-Reply Protocol

Figure 5.2 Request-reply communication



This protocol is based on three key communication primitives: doOperation, getRequest, and sendReply. It matches requests with replies and can be designed to provide delivery guarantees. When UDP datagrams are used, the protocol provides these guarantees, using server replies as acknowledgments for client requests.

Communication Primitives

1. **doOperation**:
    - Used by clients to invoke remote operations by sending a request message to a remote server. It includes arguments specifying the server, operation ID, and necessary parameters.
    - The client marshals arguments into a byte array, sends the request, waits for the reply, and unmarshals the result.
    - The client remains blocked until the reply is received.
2. **getRequest**:
    - Used by the server to acquire requests from clients. Once the request is received, the server performs the operation.
3. **sendReply**:
    - The server sends the reply back to the client after completing the requested operation.

Message Structure

Messages contain the following fields:

- **MessageType**: Specifies if it is a Request or Reply.
- **RequestId**: Identifies the message.

- **RemoteReference**: Refers to the remote server.
- **OperationId**: Identifies the operation to be invoked.
- **Arguments**: Contains operation parameters.

Message Identifiers

Each message has a unique identifier consisting of:

1. **RequestId**: Generated by the sender from an increasing sequence of integers.
2. **Sender Identifier**: Combines the sender's port and Internet address.

Failure Model

The protocol can experience various failures when implemented over UDP:

- **Omission Failures**: Messages may be lost.
- **Out-of-Order Delivery**: Messages may not be received in the order they were sent.
- **Process Failures**: Processes may crash and not recover.

To address potential failures, the client employs timeouts when waiting for replies.

Timeouts and Retransmission

When a timeout occurs, doOperation can:

- **Return Failure**: Indicate to the client that the operation failed.
- **Retransmit Requests**: The request message is resent until a reply is received or the delay is identified as a server issue.

Duplicate Request Handling

When requests are retransmitted, the server may receive duplicates. The protocol is designed to filter out duplicate requests based on request identifiers, ensuring the operation is executed only once.

Lost Reply Messages

If a reply message is lost and the server receives a duplicate request, the server can:

- **Recompute the Result**: If the operation is idempotent (produces the same result if repeated).
- **Resend the Reply**: If the result has already been stored, the server can resend the reply without re-executing the operation.

Remote Procedure Call (RPC)

The concept of a Remote Procedure Call (RPC) represents a significant advancement in distributed computing, aiming to make the programming of distributed systems resemble conventional programming. RPC allows procedures on remote machines to be called as if they were local, with the underlying system managing the complexities of distribution.

Design Issues for RPC

Programming with Interfaces

- Modern programming languages use modules for organization, where explicit interfaces define accessible procedures and variables.
- In distributed systems, modules run in separate processes, with a service interface specifying procedures available to clients, enabling communication without knowledge of implementation details.
- Benefits of programming with interfaces include:
    - Abstraction from implementation details.
    - Independence from the programming language or platform of the service.
    - Support for software evolution through interface compatibility.

Interface Definition Languages (IDLs)

- IDLs provide a notation for defining interfaces across different programming languages.
- An IDL allows remote invocation by specifying input and output parameters, ensuring that procedures can be called irrespective of the programming language used for implementation.
- Examples of IDLs include Sun XDR, CORBA IDL, WSDL, and Google's protocol buffers.

RPC Call Semantics

RPC call semantics address the reliability and guarantees of remote invocations. Different semantics are defined as follows:

- **Maybe Semantics**: The remote procedure may execute once or not at all. This arises without fault tolerance, leading to uncertain execution due to message loss or server crashes.
- **At-least-once Semantics**: The invoker receives a result indicating the procedure was executed at least once or an exception if no result is received. This can lead to multiple executions if retries occur unless operations are designed to be idempotent.
- **At-most-once Semantics**: The caller receives a result confirming the procedure executed exactly once or an exception indicating failure. This guarantees no multiple executions through robust fault-tolerance measures.

Transparency in RPC

The goal of RPC is to provide transparency akin to local procedure calls, hiding complexities such as marshalling and message passing. RPC systems strive to achieve:

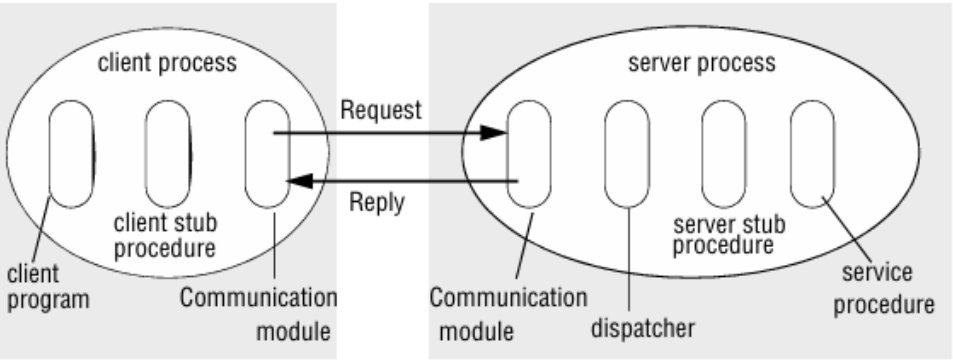
- **Location Transparency**: Hiding the physical location of the remote procedure.
- **Access Transparency**: Allowing access to local and remote procedures in the same way.

However, remote calls are more susceptible to failure due to network dependencies, necessitating clients to manage recovery from failures.

Latency and Parameter Passing

- Remote procedure calls introduce significant latency compared to local calls, which may require developers to minimize remote interactions.
- RPC does not support call by reference, requiring different parameter-passing mechanisms.

Figure 5.10 Role of client and server stub procedures in RPC



Implementation of RPC

Software Components

The implementation of RPC involves specific software components that facilitate communication between the client and the server:

- Client Process:
    - Contains a client stub for each procedure in the service interface, which acts like a local procedure.
    - Instead of executing the procedure, the client stub marshals the procedure identifier and arguments into a request message and sends it to the server through a communication module.

    - Upon receiving a reply message, the client stub unmarshals the results.
- Server Process:
    - Comprises a dispatcher, a server stub, and a service procedure for each procedure in the service interface.
    - The dispatcher selects the appropriate server stub based on the procedure identifier in the request message.
    - The server stub unmarshals the request arguments, calls the corresponding service procedure, and marshals the return values into a reply message.
- Interface Compiler:
    - The client and server stubs, along with the dispatcher, can be automatically generated by an interface compiler from the service's interface definition.

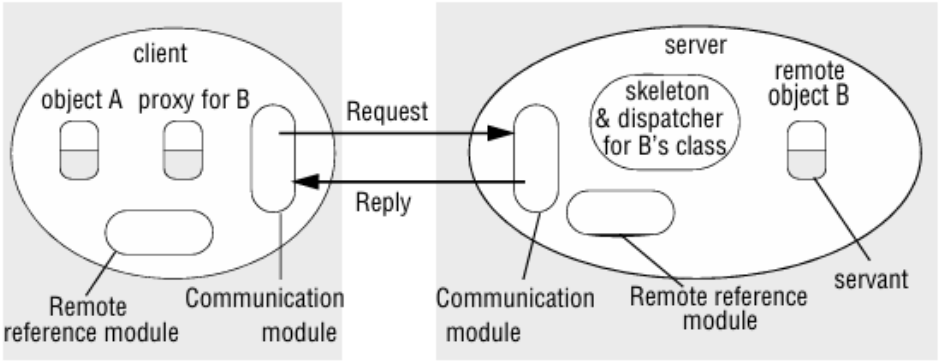
Communication and Protocols

- RPC is typically implemented over a request-reply protocol, which governs the structure of request and reply messages.
- The messages' contents adhere to the established request-reply protocols.

Invocation Semantics

- RPC can be designed to support various invocation semantics, primarily at-least-once or at-most-once.
- The communication module is responsible for implementing these semantics by managing:
    - Retransmission of requests.
    - Handling of duplicate messages.
    - Retransmission of results.

Figure 5.15 The role of proxy and skeleton in remote method invocation



5.4.2 Implementation of RMI

- **Components**: RMI involves various components including communication modules, remote reference modules, and middleware for managing remote objects.
- **Communication Module**: It carries out the request-reply protocol, transmitting messages between the client and server. The communication module handles invocation semantics like "at-most-once."
- **Remote Reference Module**: This module translates between local and remote object references and maintains a remote object table, which keeps track of the correspondence between local proxies and remote objects.
- **Servants**: A servant is an instance of a class that implements the methods of a remote object. Servants are created within the server process and are responsible for handling remote requests.
- **RMI Middleware**:
    - **Proxy**: The proxy represents the remote object on the client side, forwarding method invocations to the remote object and handling marshalling and unmarshalling.
    - **Dispatcher**: The dispatcher selects the appropriate method in the skeleton based on the operation identifier from the request.
    - **Skeleton**: The skeleton unmarshals the request, invokes the corresponding servant method, and marshals the result for sending back to the client.
- **Dynamic Invocation**: For cases where the remote interface is not known at compile time, dynamic invocation allows clients to make calls using generic representations, although it’s less convenient than static proxies.
- **Server and Client Programs**: The server program hosts the classes for dispatchers, skeletons, and servants, while the client program contains proxy classes and can use a binder to look up remote object references.
- **Binder**: A binder maintains mappings between textual names and remote object references, allowing clients to look up remote objects by name.
- **Activation of Remote Objects**: Some remote objects are not active all the time. They can be activated on demand by activators, which manage the activation process and keep track of server locations.
- **Persistent Object Stores**: Persistent objects live beyond process activations and are managed by persistent object stores, which allow them to be activated as needed.
- **Object Location Services**: Location services help clients find remote objects, maintaining a database that maps remote object references to their current locations.