4. DISTRIBUTED HETEROGENEOUS APPLICATION AND CORBA

1. Heterogeneity in Distributed System

Heterogeneity in distributed systems refers to the diversity and variation in the components that make up the system.

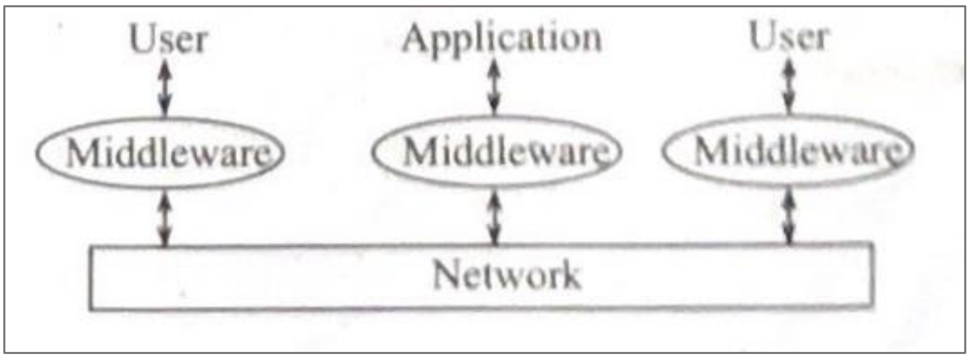
These components can differ in terms of hardware, operating systems, programming languages, network protocols, middleware, and data formats.

Reasons for Heterogeneity in Distributed Systems

1. **Variety of Hardware**: Different nodes in a distributed system might use different types of hardware, such as servers, desktops, mobile devices, and IoT devices, each with unique specifications and capabilities.
2. **Multiple Operating Systems**: Nodes in a distributed system may run different operating systems, such as Windows, Linux, macOS, or Android, necessitating compatibility across these platforms.
3. **Diverse Programming Languages**: Applications within the system may be developed using various programming languages, such as Java, C++, Python, and JavaScript, each requiring appropriate interfaces and interoperability mechanisms.
4. **Different Network Protocols**: Communication within the system may rely on different network protocols, including TCP/IP, HTTP, FTP, and proprietary protocols, demanding support for multiple communication standards.
5. **Middleware Variations**: Middleware solutions, which facilitate communication and management of data between components, can vary widely, requiring the system to handle multiple middleware platforms and versions.
6. **Data Format Differences**: Data exchanged within the system may be in different formats (e.g., JSON, XML, CSV, binary), necessitating translation and compatibility mechanisms.

2. Middleware

Middleware is a layer of software that sits between the operating system and the applications in a distributed system. It provides a set of common services and capabilities designed to facilitate communication, data management, and input/output among disparate applications across a network. Middleware abstracts the complexities of the underlying hardware and network protocols, enabling developers to focus on higher-level application logic without worrying about the intricacies of the distributed environment.

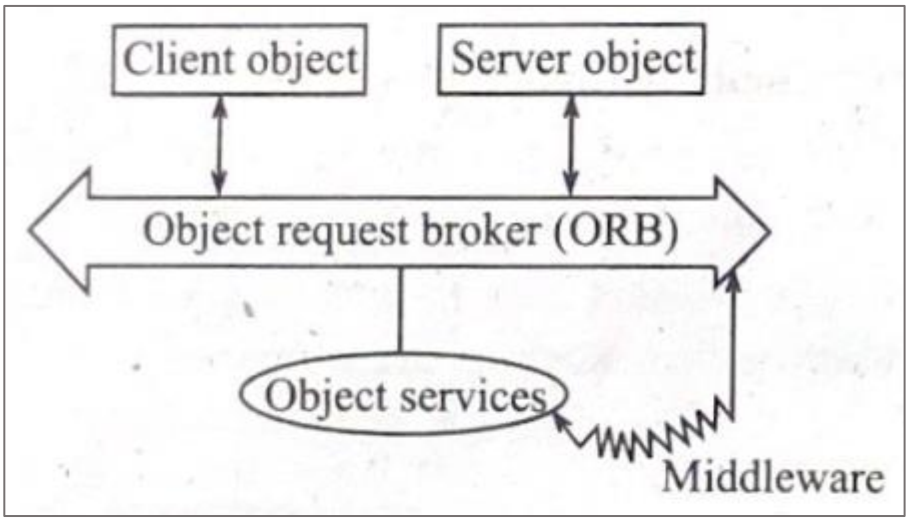


Examples of Middleware:

- **Database Middleware**: Facilitates interaction with database systems (e.g., ODBC, JDBC).
- **Message-Oriented Middleware**: Supports communication between distributed applications through message queues (e.g., IBM MQ, RabbitMQ).
- **Object Middleware**: Enables communication between distributed objects (e.g., CORBA).
- **Web Middleware**: Supports web-based applications and services (e.g., Web servers, application servers).

3. Objects in Distributed System

In distributed systems, objects are self-contained entities that encapsulate both data and the methods (functions) that operate on that data. These objects can interact with one another across different network nodes. Objects can be clients, server or both.



Object Request Broker (ORB)

An Object Request Broker (ORB) is a middleware component that facilitates communication between objects in a distributed environment. The ORB acts as an intermediary, handling the complexities of locating objects, managing requests, and delivering responses. It allows objects to communicate seamlessly regardless of where they reside within the network.

Object Services

It allows to create, name, move, copy, store, delete, restore and manage objects.

4. Interface Definition Language (IDL)

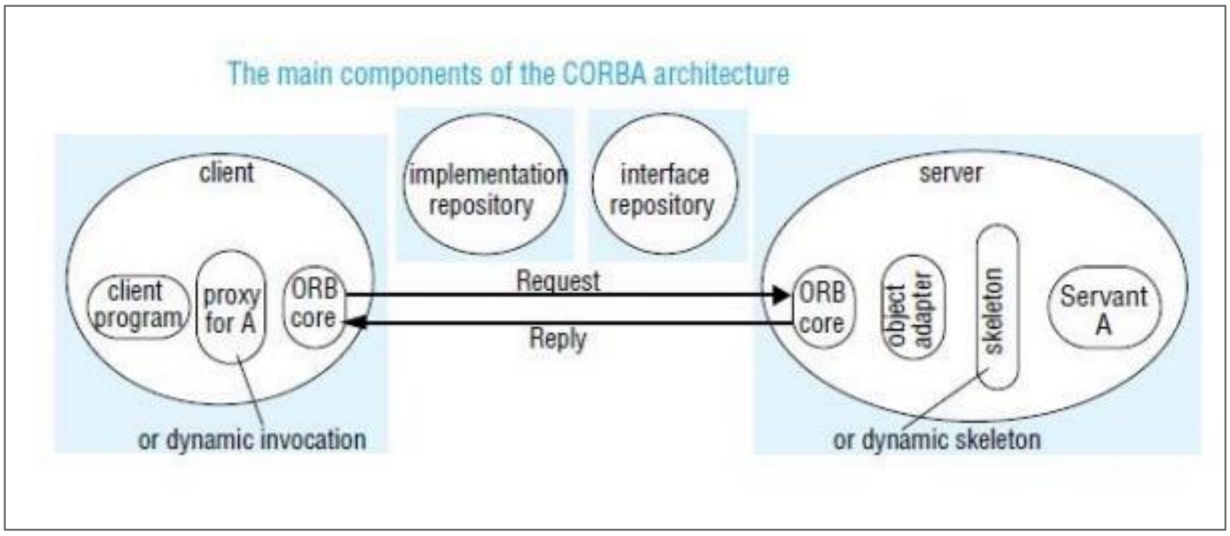
Interface Definition Language (IDL) is a specification language used to define the interface between software components. It allows for the description of data types and interfaces in a language-agnostic way, facilitating communication between programs written in different programming languages. IDL is commonly used in remote procedure call (RPC) systems and component-based software engineering. Key features include:

1. **Language Independence**: IDL enables components written in different programming languages to communicate with each other by defining a common interface.
2. **Data Type Definitions**: It defines complex data types that can be shared across different systems.
3. **Method Signatures**: It specifies the methods, their parameters, and return types that a component exposes.
4. **Platform Independence**: IDL helps in creating a consistent interface that is independent of the underlying hardware or operating system.
5. **Serialization**: It assists in the serialization and deserialization of data to ensure proper transmission over a network or between processes.

IDLs are used in technologies like CORBA (Common Object Request Broker Architecture), Microsoft COM (Component Object Model), and Thrift.

5. CORBA

CORBA (Common Object Request Broker Architecture) is a standard defined by the Object Management Group (OMG) that allows pieces of programs, which may be located on different computers and written in different programming languages, to communicate with each other. CORBA achieves this through an Object Request Broker (ORB) that handles the communication between objects.



Architecture

CORBA's architecture includes several key components:

1. **ORB Core**:

   The central component that handles the communication between clients and server objects. The ORB core is the backbone of CORBA, responsible for all communication and data exchange. It abstracts the details of the network protocols and provides a high-level API for invoking remote methods.

2. **Object Adapter**:

   Connects the ORB to the server objects, helping the ORB manage object references and invoke methods. The object adapter is a critical component that helps manage server-side objects. It performs the following functions:

   - Activates and deactivates objects.
   - Associates object references with the actual implementations.
   - Manages request dispatching to the appropriate server objects.

3. **Client Stubs and Skeletons**:

   These are generated from the IDL definitions. Stubs act as proxies on the client side, while skeletons handle method calls on the server side.

   - **Client Stubs:** These are generated from IDL definitions and reside on the client side. They act as local proxies, forwarding method calls to the ORB.
   - **Skeletons:** Also generated from IDL, these reside on the server side. They receive method calls from the ORB and forward them to the appropriate server object implementations.

4. **Implementation Repository**:

   The implementation repository holds information about the server implementations. It helps the ORB locate and activate the server objects as needed. This repository maintains:

   - The location of object implementations.
   - Activation policies.
   - Additional metadata required for object activation.

5. **Interface Repository**:

   The interface repository stores the metadata about CORBA objects' interfaces, such as their methods and data types. This repository enables dynamic invocation of methods and discovery of object interfaces at runtime.

CORBA Services:

- **Naming Service**: This service allows clients to bind and look up objects by name. It is similar to a directory service, where objects are registered with unique names, and clients can resolve these names to obtain object references.
- **Trading Service**: The trading service allows clients to find services based on service attributes. Instead of searching by name, clients can query the service based on what it offers (e.g., finding a printer service based on location and capabilities).
- **Transaction Management Service**: This service supports distributed transactions, ensuring that operations across multiple objects and systems are completed reliably and consistently, following the ACID properties (Atomicity, Consistency, Isolation, Durability).
- **Concurrency Control Service**: This service manages concurrent access to objects, ensuring data integrity when multiple clients try to read and write simultaneously. It provides mechanisms for locking and synchronization.
- **Security Service**: The security service provides various security mechanisms such as authentication (verifying identity), authorization (access control), data encryption (confidentiality), and integrity checks (ensuring data has not been tampered with).
- **Time Service**: The time service synchronizes clocks across different systems, providing a consistent notion of time in a distributed system. It can be used for time-stamping events and coordinating actions based on time.
- **Life Cycle Service**: This service manages the life cycle of objects, including their creation, deletion, copying, and movement between servers. It provides a standardized way to handle these operations across different implementations.