Distributed Computing Notes 🔥

By Garvit Singh, IT Undergraduate

What are Distributed Systems?

A distributed system is a collection of independent computers that appears to its users as a single coherent system.

It is a computing environment where multiple computers or nodes work together to achieve a common goal.

These systems can be organized in various ways, including mainframes, clusters, and grids, each with its unique characteristics and use cases.

Distributed Computing Techniques include :

1. **Mainframe Computing**
    - Mainframes are large, powerful, and centralized computing systems that can process a vast amount of data and handle many concurrent users.
    - They have traditionally been used in enterprises, government organizations, and industries where reliability, scalability, and security are of utmost importance.
    - Mainframes are known for their robustness, high availability, and support for large-scale transaction processing.

- In a mainframe-based distributed system, the mainframe serves as the central hub or control unit, while other devices or terminals connect to it for processing tasks. This model is highly reliable and well-suited for critical applications like banking and airline reservation systems.

2. **Cluster Computing**
    - Clusters are collections of interconnected computers (nodes) that work together as a single system. These nodes are often commodity hardware or servers.
    - Clusters can be categorized into high-availability clusters, load-balancing clusters, and high-performance clusters, depending on their purpose.
    - High-availability clusters are designed to provide fault tolerance and ensure system availability. If one node fails, another node takes over.
    - Load-balancing clusters distribute workloads across multiple nodes to improve performance and ensure efficient resource utilization.
    - High-performance clusters are used for parallel computing and scientific simulations, where the processing power of multiple nodes is harnessed to solve complex problems faster.
3. **Grid Computing**
    - Grid computing involves connecting distributed and often heterogeneous resources, such as computers, storage, and data, to form a seamless and virtualized computing environment.
    - Grids are typically used for scientific and research applications that require massive computational power, data storage, and collaboration across different organizations.
    - Grids allow resources to be shared and accessed remotely, enabling researchers to leverage computing power and data storage facilities that may be distributed across the globe.

- An example of grid computing is the Large Hadron Collider (LHC) experiments, where scientists from various countries collaborate by sharing and analyzing massive datasets.

4. **High Performance Computing(HPC)**
    - Uses distributed computing facilities for solving problems that need large computing power.
    - Supercomputers and clusters are specifically designed to support HPC applications to solve challenging scientific and engineering problems.
5. **High Throughput Computing(HTC)**
    - Uses distributed computing facilities for applications requiring large computing power over a long period of time.
    - HTC systems need to be robust and reliably operate over a long time scale.
6. **Many Tasks Computing(MTC)**
    - Bridges the gap between HPC & HTC.
    - MTC is similar to High Throughput Computing, but it concentrates on the use of many computing resources over a short period of time to accomplish many computational tasks.

Remote Procedure Call(RPC)

A Remote Procedure Call (RPC) is a protocol that enables a program to execute code or procedures on a remote server as if they were local, without the programmer explicitly coding the details for remote communication.

RPC allows programs to request services or functions from a server or another application running on a different machine or in a different process.

RPC is used in distributed computing and remote service invocation scenarios, such as client-server applications, microservices, and distributed systems.

It abstracts the complexities of network communication, making it easier for developers to build applications that span multiple machines or processes while maintaining a seamless developer experience, much like invoking local functions.

Common examples of RPC frameworks include gRPC, Apache Thrift, Java RMI etc.

1. **Client-Server Interaction**

- RPC involves a client (the requester) and a server (the provider of services).
- The client sends a request to the server to execute a specific function or procedure.

2. **Procedure Call Semantics**

- RPC abstracts the invocation of remote procedures to make it resemble a local function call.
- The client calls a function on the server as if it were a local function.

3. **Stubs**

- To make remote calls look like local calls, RPC systems often use stubs or proxies.
- The client-side stub marshals the parameters, sends the request to the server, and unmarshals the results.
- The server-side stub receives the request, unpacks the parameters, calls the actual function, and sends the results back.

4. **Marshalling and Unmarshalling**

- Marshalling is the process of converting function parameters and return values into a format suitable for transmission, often in a binary or text format.
- Unmarshalling is the reverse process on the server side, converting the transmitted data back into usable parameters and results.

5. **Transport Protocol**

- RPC systems typically use a transport protocol (e.g., TCP/IP, HTTP, or custom protocols) to transmit the request and receive the response between the client and server.

6. **Binding**

- Binding is the process of associating a specific remote procedure with its corresponding server address and communication details.
- It can be done statically (at compile time) or dynamically (at runtime).

7. **IDL (Interface Definition Language)**

- Many RPC systems use IDL to define the interface between the client and server.
- The IDL provides a platform-independent way to describe data types, functions, and procedures, making it easier for client and server code to interact.

Distributed Object Frameworks

Also known as Distributed Object Computing (DOC) frameworks or Distributed Object Middleware, are software frameworks and technologies that facilitate the development of distributed applications by extending the concept of object-oriented programming to distributed systems.

Distributed Object Frameworks simplify the development of distributed systems by providing a higher-level, object-oriented abstraction for network communication. They have been used in various domains, including enterprise applications, telecommunications, and distributed systems where flexibility, scalability, and interoperability are critical.

These frameworks enable objects (software components) to interact with one another seamlessly, even when they are located on different machines within a network or distributed environment.

Key features and concepts of Distributed Object Frameworks include:

1. **Object-Oriented Paradigm**

- Distributed Object Frameworks are built upon the principles of object-oriented programming.
- They enable objects to communicate and collaborate in a distributed environment, preserving the object-oriented model's encapsulation, inheritance, and polymorphism.

2. **Location Transparency**

- Distributed objects are designed to be location-transparent, meaning that clients interact with objects using the same syntax and method calls regardless of the object's physical location.

- This abstracts the complexities of network communication.

3. **Remote Method Invocation (RMI)**

- Distributed Object Frameworks typically use remote method invocation mechanisms to allow objects to invoke methods on remote objects as if they were local.
- This involves serializing method parameters and sending them over the network to the remote object for execution.

4. **Object Serialization**

- Objects must be serializable, meaning they can be converted into a format that can be transmitted over the network and reconstructed as objects on the remote side.

5. **Object Activation**

- Some distributed object frameworks support object activation, where objects can be created on-demand on remote servers.
- This allows resources to be allocated dynamically based on demand.

6. **Security and Access Control**

- Distributed Object Frameworks often include mechanisms for secure communication and access control, ensuring that only authorized clients can access and invoke methods on distributed objects.

7. **Middleware Services**

- These frameworks may provide additional middleware services such as naming and directory services, transaction management, and event notification, which simplify the development of distributed applications.

8. **Interoperability**

- Distributed Object Frameworks aim to be platform-agnostic and support interoperability across various programming languages and platforms.

Popular Distributed Object Frameworks

1. **Common Object Request Broker Architecture(CORBA)**

- A platform-agnostic middleware that enables interoperability between objects in different languages and on different platforms.

2. **Java Remote Method Invocation(RMI)**

- A Java-based framework for remote communication between objects, allowing Java objects to interact over a network.

3. **Distributed Component Object Model(DCOM)**

- A Microsoft technology that extends the Component Object Model(COM) to support distributed computing in Windows environments.

4. **Enterprise JavaBeans(EJB)**

- A component architecture for building distributed business applications using Java.
- It provides a framework for building and running distributed, transactional, and secure enterprise applications.

Service-Oriented Architecture (SOA)

It is a design approach and architectural style for developing software systems that promote the use of services as fundamental building blocks.

In SOA, a service is a self-contained, modular unit of functionality that is designed to be independent, reusable, and interoperable.

The goal of SOA is to create a flexible and scalable architecture that supports the efficient integration of disparate systems and applications.

1. **Standardized Service Contract**

- A standardized service contract defines the interface and interaction patterns for a service.
- This contract includes information about how to access the service, what it does, and the data formats it uses.
- Standardization ensures that services can be easily discovered, understood, and integrated into applications.

2. **Loose Coupling**

- Loose coupling refers to the degree of dependency between services.
- In SOA, services are designed to be loosely coupled, which means they are relatively independent and can function without intimate knowledge of each other.
- Loose coupling enhances flexibility, as changes to one service have minimal impact on other services.

3. **Abstraction**

- Abstraction involves hiding the complex details of a service's implementation and exposing only the necessary information through the service contract.
- This simplifies the interaction with the service and allows for changes in the underlying implementation without affecting service consumers.

4. **Reusability**

- Reusability is a key principle in SOA. Services are designed to be reusable components that can be utilized in various applications and contexts.
- This reduces development effort and enhances consistency across the organization.

5. **Autonomy**

- Services in SOA are autonomous, meaning they have control over their own functionality and data.
- Autonomy allows services to evolve independently and make decisions about their operations.

6. **Lack of State**

- Services in SOA are typically designed to be stateless, which means they don't retain information about previous interactions with clients.
- This simplifies the management and scalability of services.

7. **Discoverability**

- Discoverability is the ability for clients to find and access services easily.
- Services should be discoverable through directories, registries, or service metadata.

- Discoverability is crucial for enabling service integration.

8. **Composability**

- Composability refers to the ability to combine and orchestrate services to create more complex, higher-level applications.
- SOA promotes the creation of composite applications through the assembly of services.

Thanks For Reading! 💙

By GARVIT SINGH

Information Technology