# Client Server Application Programming

## Week 13: Java IDL and CORBA( Overview, Architectural View of CORBA, Interface Definition Language (IDL))

By Elubu Joseph - MSc.IS

Lecturer

Department of Information Technology

Kumi University

Email: josebulinda@gmail.com

jose@kumiuniversity.ac.ug

# Summary of Previous Lecture

1. Remote Method Invocation Deployment Issues
2. Using RMI to Implement Callbacks
3. Remote Object Activation

# Agenda

1. Overview of IDL and CORBA
2. Interface Definition Language (IDL)
3. Architectural View of CORBA

# Overview of IDL and CORBA

This section of the course introduces us to Common Object Request Broker Architecture (CORBA) and Interface Definition language-IDL, where we shall learn a number of concepts.

We covered other concepts relating to Remote Method Invocation (RMI) in our previous lectures, however like with many things in life, there is more than one way to achieve the same goal.

**Common Object Request Broker Architecture** is a standard for making object method invocation requests over a network.

The Object Management Group (OMG) , a large consortium of companies devoted to improving **remote object method invocation**, developed the CORBA specification. Visit http://www.omg.org/ for more details bout OMG.

# Interface Definition Language (IDL)

The IDL defines many language constructs, allowing a rich schema of objects to be described. We'll cover some few concepts before we finally end this lecture.

# Key Features of IDL

1. **Interface Definition Language:** IDL is a language-independent specification language used to describe the interfaces of CORBA objects. It provides a standardized way to define the operations, attributes, and exceptions that objects expose.
2. **Platform and Language Independence:** IDL allows developers to define interfaces in a language-independent manner. It enables objects implemented in different programming languages and running on different platforms to communicate with each other seamlessly.

# Key Features of IDL

3. **Interface Inheritance:** IDL supports inheritance, allowing interfaces to inherit from other interfaces. This promotes code reuse and supports polymorphism in distributed systems.
4. **Data Types and Marshaling:** IDL provides a set of primitive data types and data structures for defining the parameters and return types of operations. It also handles the marshaling and unmarshaling of data between different programming languages and platforms. Etc.

**Note: Marshaling**, also known as serialization or encoding, is the process of converting complex data structures or objects into a format that can be transmitted or stored, and later reconstructed back into their original form (**unmasrshaling**). It is commonly used in distributed systems, network communication, and persistence mechanisms.

## Why is it important to learn IDL?

In addition to above, we will be able to cover some more concepts of IDL in this lecture. However, allow me build your confidence to learn IDL beyond what we'll cover here.

# Interface Definition Language (IDL)+ Why you may need to depend your IDL Knowledge.

Learning IDL (Interface Definition Language) can be valuable in certain scenarios, particularly if you work with technologies that rely on CORBA or other middleware systems that use IDL as a communication interface. Here are some considerations to help you decide if learning IDL is worth it for you:

1. **Working with Legacy Systems:** If you are involved in maintaining or integrating with legacy systems that use CORBA or other middleware technologies based on IDL, learning IDL becomes essential. It allows you to understand and modify existing IDL interfaces, generate code stubs and skeletons, and interact with the legacy systems effectively.
2. **Interfacing with CORBA or Middleware:** IDL serves as a language-independent interface definition that facilitates communication between different programming languages and platforms. If you need to interface with CORBA or other middleware systems that use IDL, understanding IDL is crucial for defining interfaces and enabling interoperability.

# Interface Definition Language (IDL)+ Why you may need to depend your IDL Knowledge.

3. **Distributed Systems Development:** If you are interested in distributed systems development and want to explore middleware technologies **beyond RESTful APIs** and **web services**, learning IDL can provide you with insights into the principles and concepts of distributed computing. It broadens your understanding of communication protocols, object-oriented architectures, and remote procedure calls.
4. **Expanding Technical Skills:** Learning IDL expands your skill set and knowledge base. It demonstrates your ability to work with different technologies and adapt to varying communication protocols. This can be advantageous in certain job roles or projects that require integration with diverse systems and architectures.

Ultimately, the decision to obtain dtailed IDL knowledge depends on who you are or what you want to be. **Good enough!! Lets see how IDL works.**

# How IDL Works +

The IDL defines many language constructs, allowing a rich schema of objects to be described.

Below are IDL language constructs;-

1. IDL Datatypes
2. IDL Interfaces
3. IDL Modules
4. IDL Artributes
5. IDL Operations
6. IDL Exception Handling

# How IDL Works +

## IDL Datatypes

IDL defines a set of primitive datatypes, such as numbers and text, as well as the ability to create more powerful constructs like arrays, sequences, and data structures.

IDL supports various datatypes for defining interfaces. The specific set of datatypes available in IDL may vary depending on the version of IDL being used and the IDL mapping used for a particular programming language. However, below is a list of commonly used datatypes in IDL:

**1. Primitive Types:**

1. Boolean: Represents a boolean value (true or false).
2. Integer: Represents integer values (e.g., short, long, unsigned, etc.).
3. Floating-Point: Represents floating-point values (e.g., float, double).
4. Character: Represents a single character.

**2. String:** Represents a sequence of characters.

**3. Enumeration:** Represents a set of named values.

# How IDL Works+

## IDL Datatypes+

below are a list of commonly used datatypes in IDL:

4. **Struct:** Represents a composite type that groups together related fields or members.
5. **Union:** Represents a type that can hold values of different types, but only one at a time.
6. **Array**: Represents a collection of elements of the same type.
7. **Sequence:** Represents a variable-length array.
8. **Interface:** Represents a reference to another interface.
9. **Object:** Represents an object reference.
10. **Any:** Represents a variant or dynamically typed value that can hold values of any type.

These are some of the commonly used datatypes in IDL. It's important to note that the actual set of datatypes may vary depending on the specific IDL implementation or the programming language used for implementing the IDL interface.

# How IDL Works +
# IDL Interfaces

Interface definitions describe a remote object, the methods that the object exposes, as well as member variables, such as constants. The *interface* keyword is used to describe an interface. The following is an example of an interface:

```
interface UserAccount {
    float getBalance();
    void setBalance(in float amount);
}
```

# How IDL Works +
# IDL Modules

Modules are a convenient way of grouping logically related interfaces together for convenience.

Consider modules similar to a package in Java; related classes are grouped together into categories.

**Java Datatypes and Their Mapping to IDL Datatypes**

| Java Datatype | IDL Datatype |
|---|---|
| Void | `void` |
| Boolean | `boolean` |
| Char | `wchar` |
| Byte | `octet` |
| Short | `short` |
| Int | `long` |
| Long | `long long` |
| Float | `float` |
| Double | `double` |
| java.lang.String | `string / wstring` |

# How IDL Works +

## IDL Attributes

In IDL (Interface Definition Language), attributes are used to define properties or characteristics of an interface member. They provide additional information about the member's behavior, visibility, or other aspects. Here are some common attributes used in IDL:

1. **Read-only Attribute:** indicates that the corresponding member can only be read and not modified. It is used for defining constant or read-only properties.
2. **Read-write Attribute:** is used to define a read-write property. It allows both reading and writing values to the member.
3. **Static Attribute:** is used to specify that the member belongs to the interface itself and not to instances of the interface. Static members can be accessed directly using the interface name without creating an instance.

# How IDL Works +

## IDL Attributes+

4. **ID Attribute:** is used to assign a unique identifier to the member. It is particularly useful when mapping IDL interfaces to other languages or systems.
5. **Context Attribute:** is used to specify that the member requires a specific context or environment information for its operation. It is often used in distributed systems to pass contextual information between components.
6. **Version Attribute:** is used to specify the version number or identifier associated with the member. It helps in managing backward compatibility and versioning of interfaces.

# How IDL Works +

## IDL Attributes++

7. **Deprecated Attribute:** is used to indicate that the member is no longer recommended for use. It is typically used when a new version or alternative approach is available, and the member is considered outdated or problematic.

These attributes, along with other IDL constructs, provide a way to describe the characteristics and behavior of interface members. They enhance the expressiveness and flexibility of IDL and assist in generating code stubs, documenting interfaces, and enforcing certain rules or constraints during the implementation and usage of interfaces.

# How IDL Works +
## IDL Attributes+++

The following is an example of an attribute, in contrast to our first interface example, which defined accessor functions:

```
interface UserAccount {
        // balance is not protected, and can be easily changed
        attribute float balance;

        // account id may be read but not remotely modified      readonly

        attribute long long accountid;
};
```

# How IDL Works +

## IDL Operations

Operations are functions that can return a value upon completion and can accept parameters. They can be thought of as the public interface of a component, exposing the functionality it provides to other components or clients.

**IDL operations typically specify the following information:**

1. **Operation Name:** The name of the method or function.
2. **Parameters:** The input values that the operation expects, including their names and data types.
3. **Return Type:** The data type of the value that the operation returns.
4. **Exceptions:** Any exceptions or errors that the operation may throw.

# How IDL Works +

## IDL Operations+

While attributes have their uses, the most important part of an IDL schema is the operations that are defined. These operations may be invoked remotely, and can perform both simple and complex tasks, limited only by the imagination.

Below are three types of parameters that may be specified for an IDL operation:

1. in— parameter used for input only, and is immutable

2. out— parameter whose contents may be modified, and is not immutable

3. inout— parameter that combines the properties of in and out. Such parameters may be used as input and may also be modified.

# How IDL Works +
## IDL Operations+++

The following interface shows three simple operations, two of which accept an input parameter. Each operation also specified a return value (in this case, of type float).

```
interface UserBalance {
   float getBalance();
   float addBalance(in float amount);
   float subtractBalance(in float amount);
};
```

Note that if a method does not have a return value, just as in Java you should specify this by using the *void* keyword.

# How IDL Works +
## IDL Exception Handling

IDL provides a mechanism for defining exception handling in distributed systems. Exception handling in IDL allows you to specify and handle exceptional conditions that may occur during remote method invocations. Here's how IDL handles exceptions:

1. **Defining Exceptions:** In IDL, you can define custom exception types using the exception keyword. Exception definitions include a name and a list of member variables that provide additional information about the exception. For example:

   The following is an **example of two IDL exceptions** thrown by an operation. The ***raises*** keyword is used to identify a method that throws an exception.

# How IDL Works +
## IDL Exception Handling+

```
module BankingSystem{
        // Account inactive exception
        exception AccountInactive{
        string reason;
        };
        // Account overdrawn exception
        exception AccountOverdrawn{
        string reason;
        };
        interface BankAccount{
        double withdrawMoney(in double amount)raises(AccountInactive, AccountOverdrawn);
        // Further bank account methods might go here ...
        };
};
```

# How IDL Works +
## IDL Exception Handling++

2. **Throwing Exceptions:** In IDL, exceptions can be thrown explicitly by the server or implicitly by the middleware system. When an exceptional condition occurs during a remote method invocation, the server can create an instance of the exception and throw it back to the client. The exception is then propagated through the middleware to the client application.
3. **Handling Exceptions:** On the client side, IDL provides mechanisms to handle exceptions thrown by the server. When invoking a remote method, the client can use try-catch blocks to catch and handle specific exceptions. The catch blocks can perform appropriate error handling or take necessary recovery actions based on the type of the caught exception.

# How IDL Works +

## IDL Exception Handling+

### try and Catch blocks

```
try {
// Remote method invocation
}
catch (AccountInactive ob){
// Exception handling for AccountInactive
}
catch (AccountOverdrawn ob){
// Exception handling for AccountOverdrawn
}
catch (Exception ex){
// Generic exception handling
}
```

# How IDL Works +

## IDL Exception Handling+

### Propagating Exceptions

4. **Propagating Exceptions:** Exceptions can propagate through the middleware and across different programming languages. IDL specifies the mapping rules for exceptions between the IDL definition and the programming language bindings used by the client and server. The middleware system handles the serialization and deserialization of exceptions to ensure proper transmission and handling across distributed components.

IDL's exception handling mechanism provides a structured approach to deal with exceptional situations in distributed systems. It allows servers to communicate specific error conditions to clients, enables clients to handle exceptions gracefully, and ensures consistency in error reporting across different programming languages and platforms.

# Common Object Request Broker Architecture (CORBA)

CORBA (Common Object Request Broker Architecture) is a middleware technology and architecture that enables interoperability and communication between distributed objects across different platforms and programming languages.

CORBA provides a standardized and language-independent approach for building distributed systems. Within it there are a number of features as shown below.

# Key Features of CORBA

CORBA (Common Object Request Broker Architecture) is a middleware technology and architecture that enables interoperability and communication between distributed objects across different platforms and programming languages. It provides a standardized and language-independent approach for building distributed systems. Within it there are a number of features as shown below.

1. **Object Request Broker (ORB):** CORBA uses an ORB as a middleware component that enables communication and interaction between distributed objects. The ORB handles the request and response messaging between the client and server objects.
2. **Interface Definition Language (IDL):** CORBA uses IDL to define the interfaces of distributed objects. IDL provides a language-independent way to describe the operations, attributes, and exceptions of objects.

# Key Features of CORBA+

3. **Object Adapter:** is responsible for mapping the CORBA object interfaces to the actual implementation objects. It acts as an intermediary between the ORB and the implementation objects.
4. **Interface Repository:** is a centralized repository that stores the IDL definitions of all CORBA interfaces. It allows clients and servers to dynamically discover and access interface definitions at runtime.

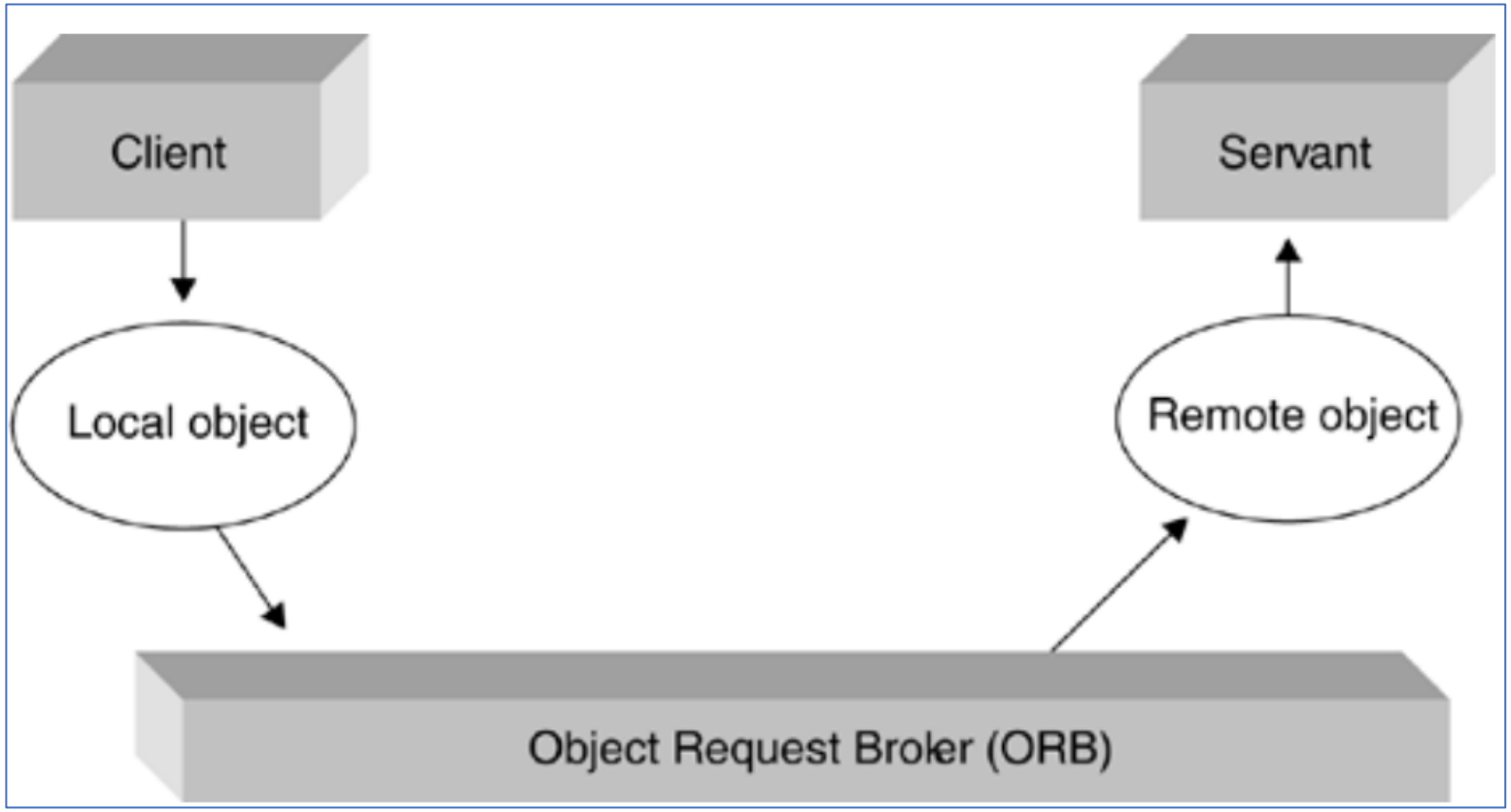
# Architectural View of CORBA

CORBA is made up of a collection of objects and software applications that work together cooperatively. A schema specification, written in IDL, describes the objects that are exported for remote usage. An object request broker (ORB) implements the interface between remote object and software client.

The ORB handles requests for an object's services, as well as sending a response. The Figure below provides an overview of how clients and servants (the end implementation of the service) interact using an ORB.

# Architectural View of CORBA+

Objects accessed locally act as a proxy for remote objects

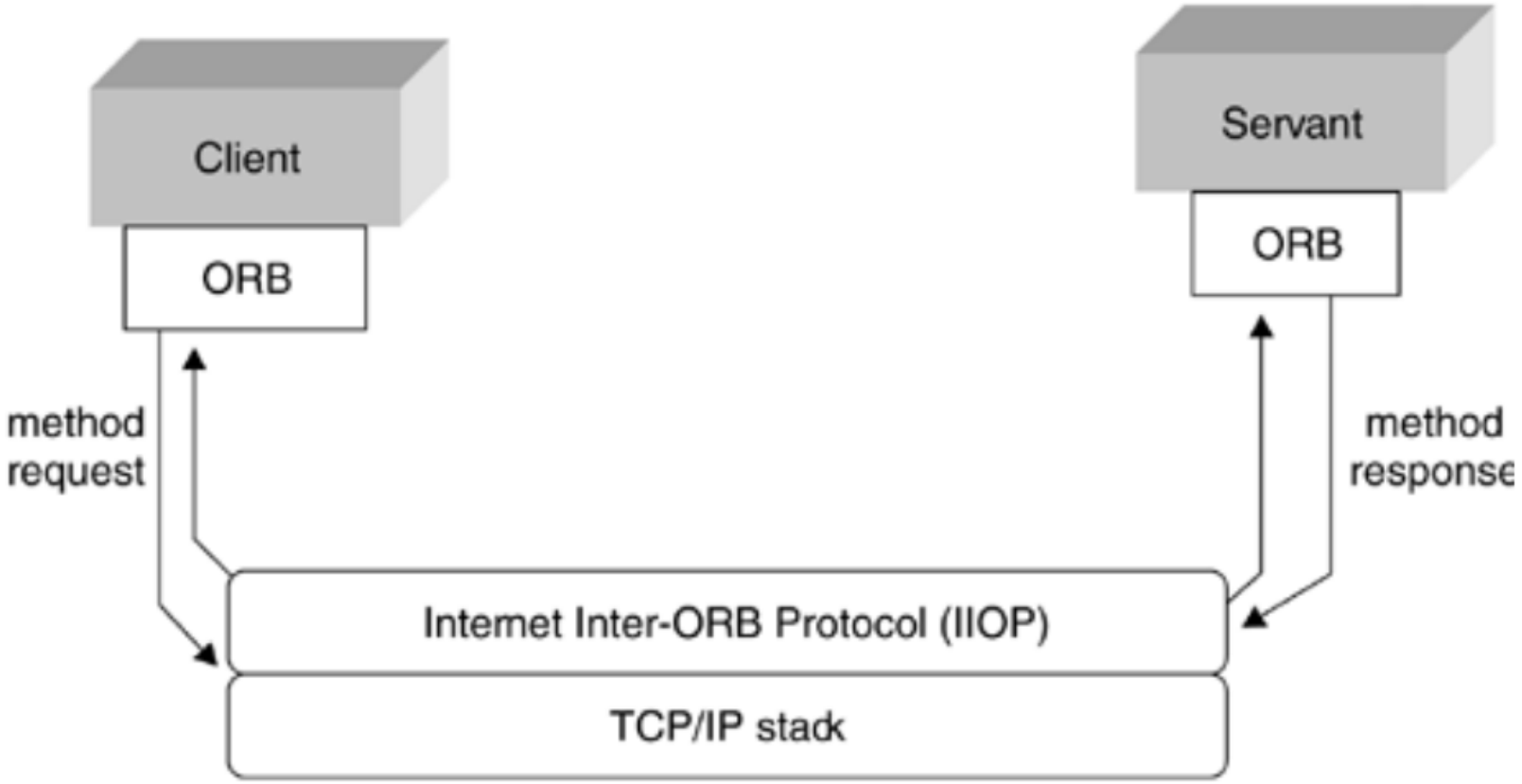



ORB is Acting as an intermediary between client and servant .

Communication between client and servant occurs over the network via the Internet Inter-ORB Protocol (IIOP), shown in the figure on the next slide

# Architectural View of CORBA+

**Communication takes places using IIOP**



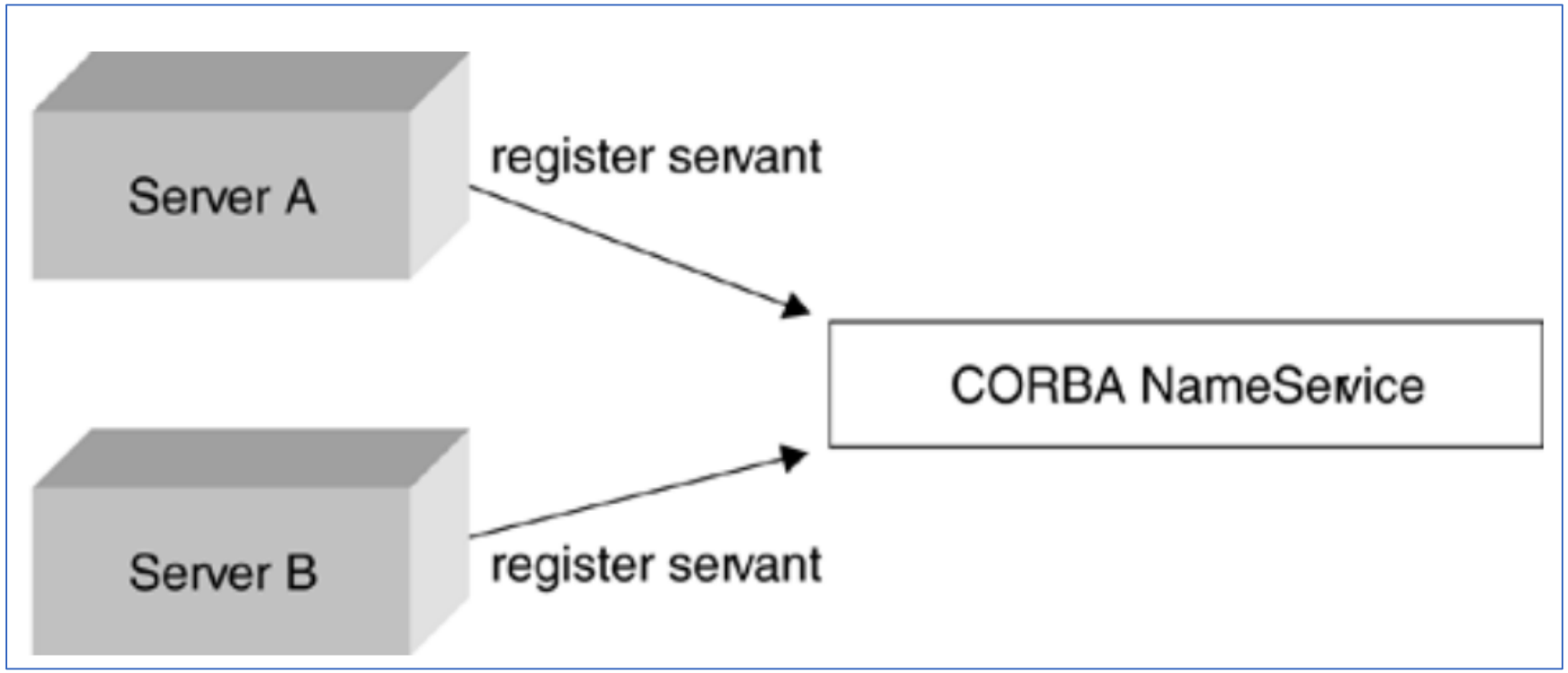
# CORBA Services

Within the CORBA architecture, software services are **described by a schema** and **implemented by a servant**. The **servant** is a special piece of software that registers itself with a lookup service, so that other CORBA software can locate and access its services.

Typically a **CORBA server will create a CORBA servant**, and is then responsible for creating an ORB for the servant and registering the service for access by other clients as shown below.

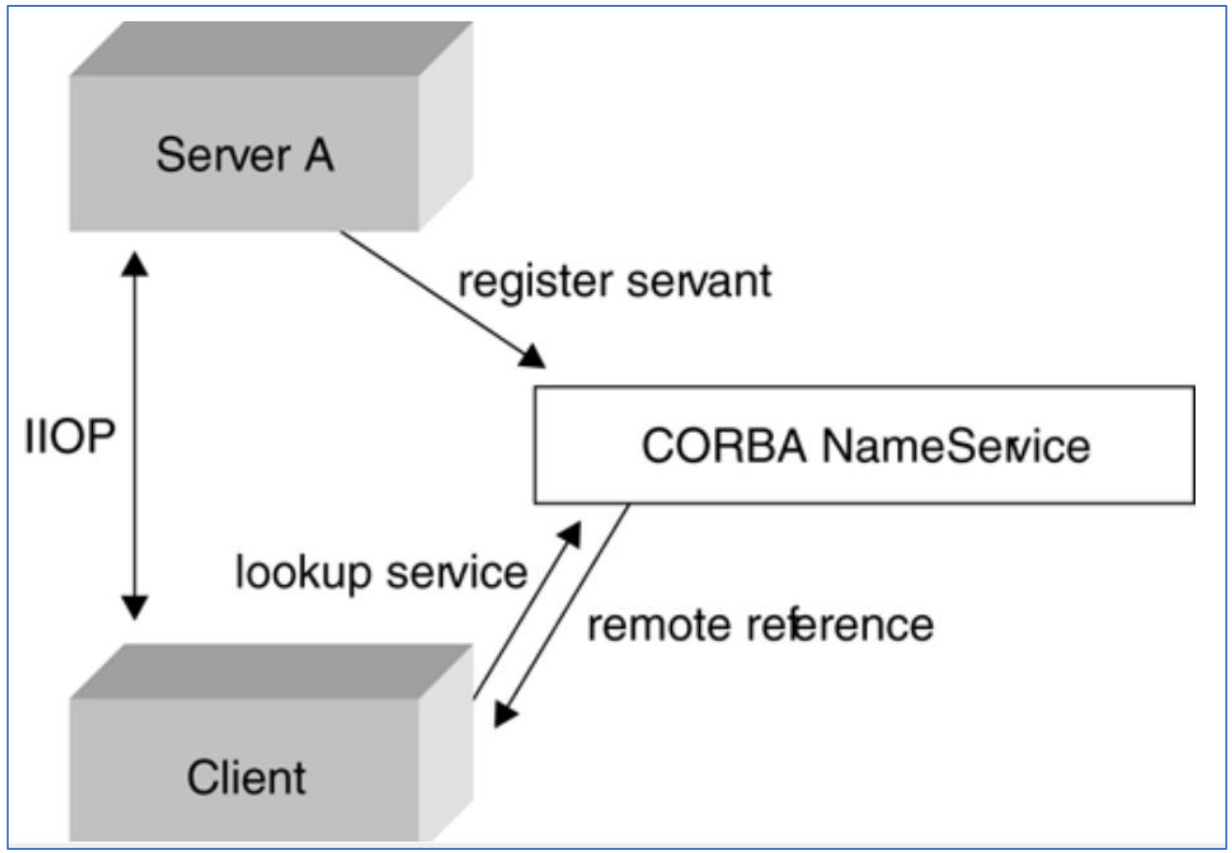
# CORBA Services+

**Servers register with a name service.**



# CORBA Clients

Clients, on the other hand, don't need to register with a name service. They do, however, want to use the name service to look up services as shown below.



# The State of CORBA popularity

While CORBA (Common Object Request Broker Architecture) was widely used in the past for building distributed systems, its popularity has significantly decreased in recent years.

However, it is important to note that there are still some legacy systems and applications that continue to use CORBA. And for purposes of history, we still go a head to learn some things about CORBA.

# Reasons for the decline in the use of CORBA+

The decline in CORBA's usage can be attributed to several factors:

1. **Complexity:** CORBA is known for its complexity, requiring significant effort and expertise to develop and maintain CORBA-based systems. This complexity has made it less attractive compared to simpler and more lightweight alternatives.
2. **Evolving Technologies:** Over time, new technologies and protocols have emerged that offer more streamlined and efficient solutions for building distributed systems. These include web services, RESTful APIs, and messaging frameworks such as MQTT and Apache Kafka.

# Reasons for the decline in the use of CORBA

The decline in CORBA's usage can be attributed to several factors:

4. **Web Services and REST:** The rise of web services and the REST architectural style has provided a simpler and more widely adopted approach to building distributed systems. Web services based on **SOAP** (Simple Object Access Protocol) or RESTful APIs have become the de facto standard for inter-system communication.
5. **Language-Specific Frameworks:** Many programming languages now offer their own language-specific frameworks and libraries for building distributed systems. These frameworks provide a more integrated and native approach to distributed computing, making CORBA less appealing.

# Should One Continue to Learn CORBA?

# Should One Continue to Learn CORBA?

I sincerely do not want to be a prophet of doom, but the decision to learn CORBA depends on your specific needs and circumstances. Here are some considerations to help you decide:

1. **Relevance:** CORBA is a mature technology that was widely used in the past for building distributed systems. However, its popularity has significantly decreased in recent years. If you are primarily interested in current and emerging technologies, such as **web services**, **RESTful APIs**, **microservices**, or **message-based** architectures, investing time in **learning CORBA may not be the most practical choice**.
2. **Legacy Systems:** If you are working with or maintaining legacy systems that still rely on CORBA, learning CORBA can be valuable to understand and support those systems. In such cases, learning CORBA will enable you to work with existing code, troubleshoot issues, and make necessary modifications or updates.

# Should One Continue to Learn CORBA?

3. **Industry Demand:** The demand for CORBA skills in the job market has diminished compared to newer technologies. Most organizations have transitioned to modern architectures and protocols, and the demand for CORBA expertise has decreased accordingly. If employability and market demand are important factors for you, focusing on more contemporary technologies may be a better strategy.
4. **Learning Curve and Complexity:** CORBA has a reputation for being complex and requiring a significant learning curve. It involves understanding concepts like IDL, ORB, object adapters, and distributed object models. If you are new to distributed systems or have limited experience, you may find other technologies that offer simpler learning paths more accessible.

# Should One Continue to Learn CORBA?

Considering factors above, it is generally advisable to prioritize learning technologies that are more widely adopted, have active communities, and align with current industry trends.

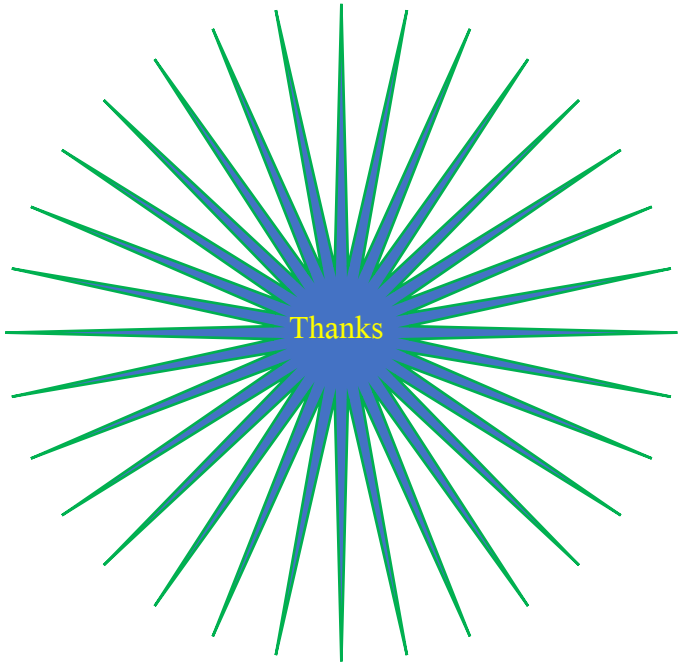
However, if you are working with legacy systems that heavily rely on CORBA or have a specific requirement to interface with CORBA-based systems, learning CORBA can be beneficial.

# Summary

# Summary

1. Overview of IDL and CORBA
2. Interface Definition Language (IDL)-(IDL interfaces, Attributes, Modules, Operations, Exception Handling etc. )
3. Architectural View of CORBA-Key features, Services and Clients and finally looked at CORBA's state of Popularity.

Thank you for Listening



# References

*Java™ Network Programming and Distributed Computing,* (David R.,Michael R. 2002), Publisher : Addison Wesley; ISBN: 0201710374