Chapter 2
Distributed Information Systems

Presented by: Dr. R. BENTRCIA

Department of Computer Science, M'sila University

Outline

- Distributed Architecture:
  - Client-Server Architecture
  - Multi-Tier Architecture (n-tier Architecture)
  - Broker Architectural Style
  - Broker implementation in CORBA
  - Remote Method Invocation (RMI)

Distributed Architecture

- In distributed architecture, components are presented on different platforms and several components can cooperate with one another over a communication network in order to achieve a specific objective or goal.
- In this architecture, information processing is not confined to a single machine rather it is distributed over several independent computers.
- A distributed system can be demonstrated by the *client-server architecture* which forms the base for *multi-tier architectures*; alternatives are the *broker architecture such as CORBA*, and the Service-Oriented Architecture.

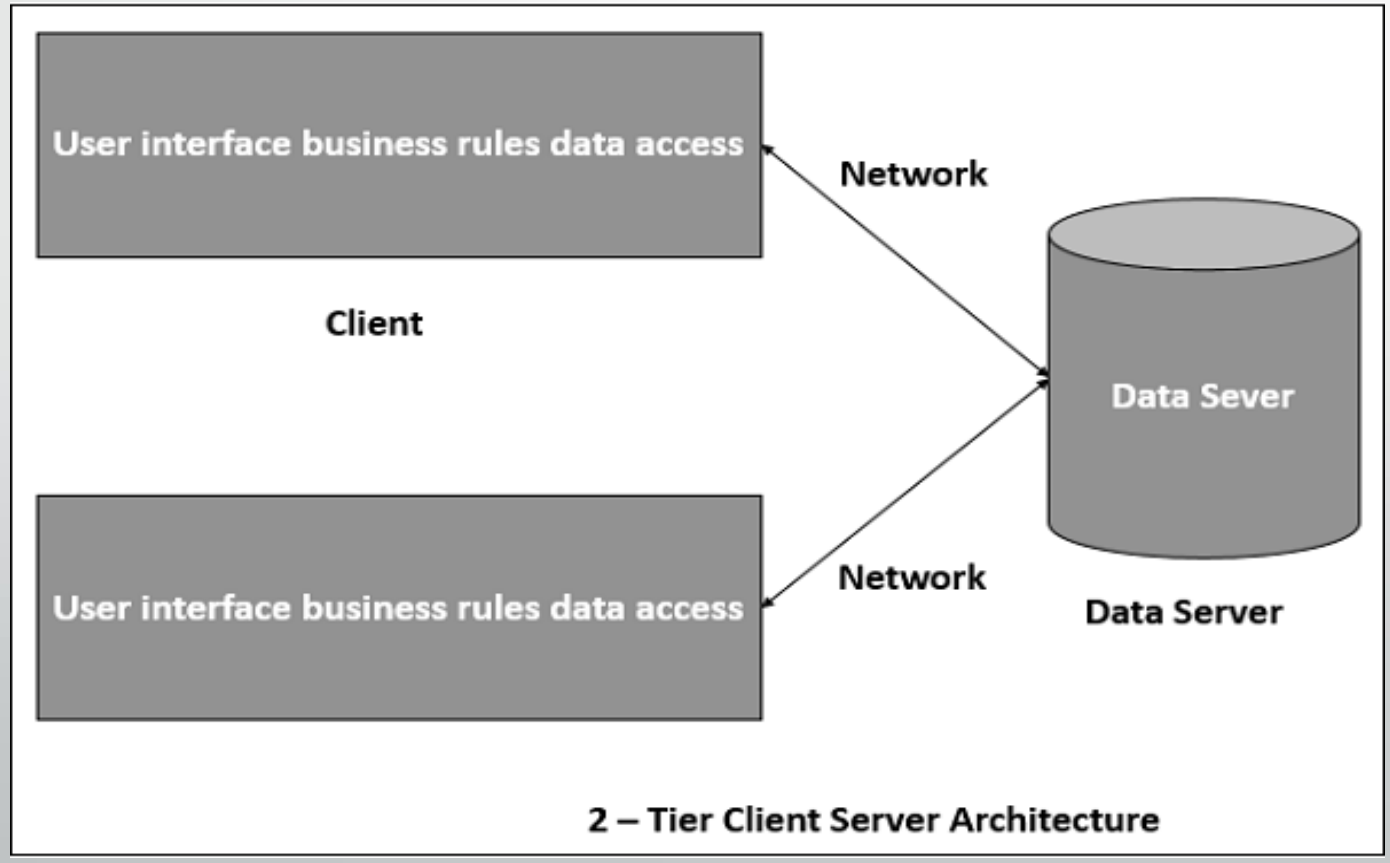
Distributed Architecture

- There are several technology frameworks to support distributed architectures, including J2EE and CORBA.
- *Middleware* is an infrastructure that appropriately supports the development and execution of distributed applications. It provides a buffer between the applications and the network. It sits in the middle of system and manages or supports the different components of a distributed system.

Client-Server Architecture

- The client-server architecture is the most common distributed system architecture which decomposes the system into two major subsystems or logical processes:
  - **Client**: This is the first process that issues a request to the second process i.e. the server.
  - **Server**: This is the second process that receives the request, carries it out, and sends a reply to the client.
- In this architecture, the application is modelled as a set of services that are provided by servers and a set of clients that use these services.

Client-Server Architecture

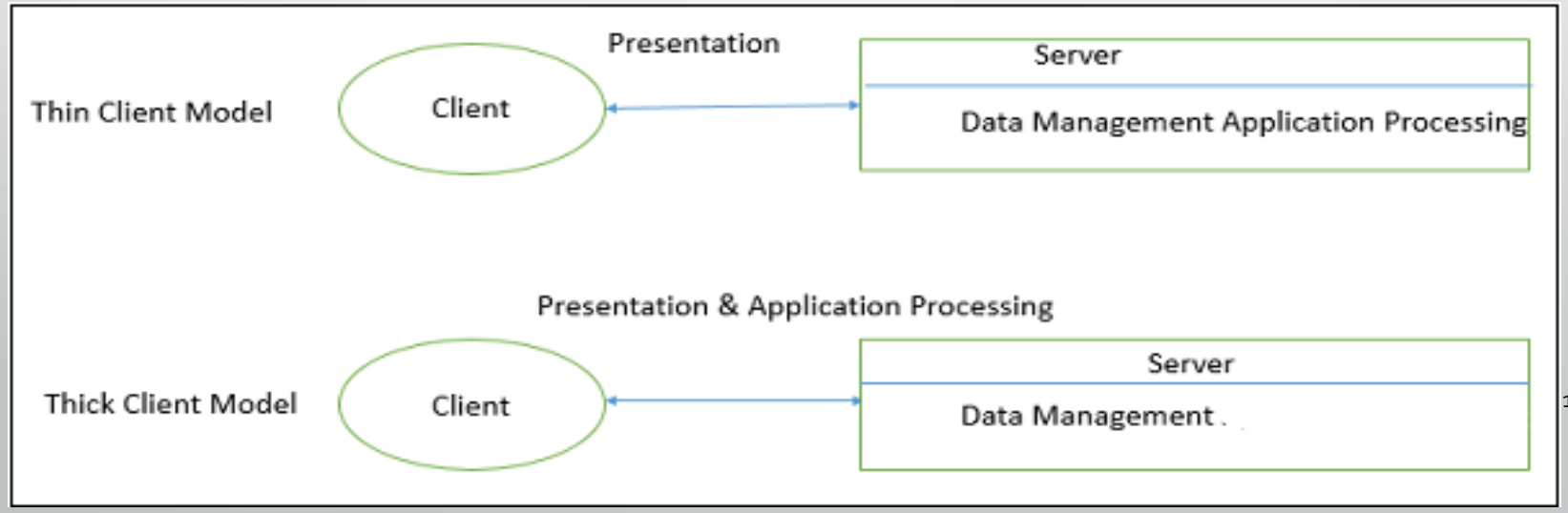


Client-Server Architecture

- Client-server Architecture can be classified into two models based on the functionality of the client:
  - *Thin-client model*: In this model, all the *application processing* and *data management* is carried by the server. The client is simply responsible for running the *presentation* software.
  - Used when legacy systems are migrated to client server architectures in which legacy system acts as a server in its own right with a graphical interface implemented on a client.
  - A major disadvantage is that it places a heavy processing load on both the server and the network.

Client-Server Architecture

- In *thick-client model*, the server is only in charge for *data management*. The software on the client implements the *application logic* and the *interactions with the system user*.
- More complex than a thin client model especially for management. New versions of the application have to be installed on all clients.



Client-Server Architecture

- Advantages:
  - Separation of responsibilities such as user interface presentation and business logic processing.
  - Reusability of server components.
  - Simplifies the design and the development of distributed applications.
  - It makes it easy to migrate or integrate existing applications into a distributed environment.
  - It also makes effective use of resources when a large number of clients are accessing a high-performance server.
- Disadvantages:
  - Security complications.
  - Limited server availability and reliability.
  - Limited testability and scalability.
  - Fat clients with presentation and business logic together.

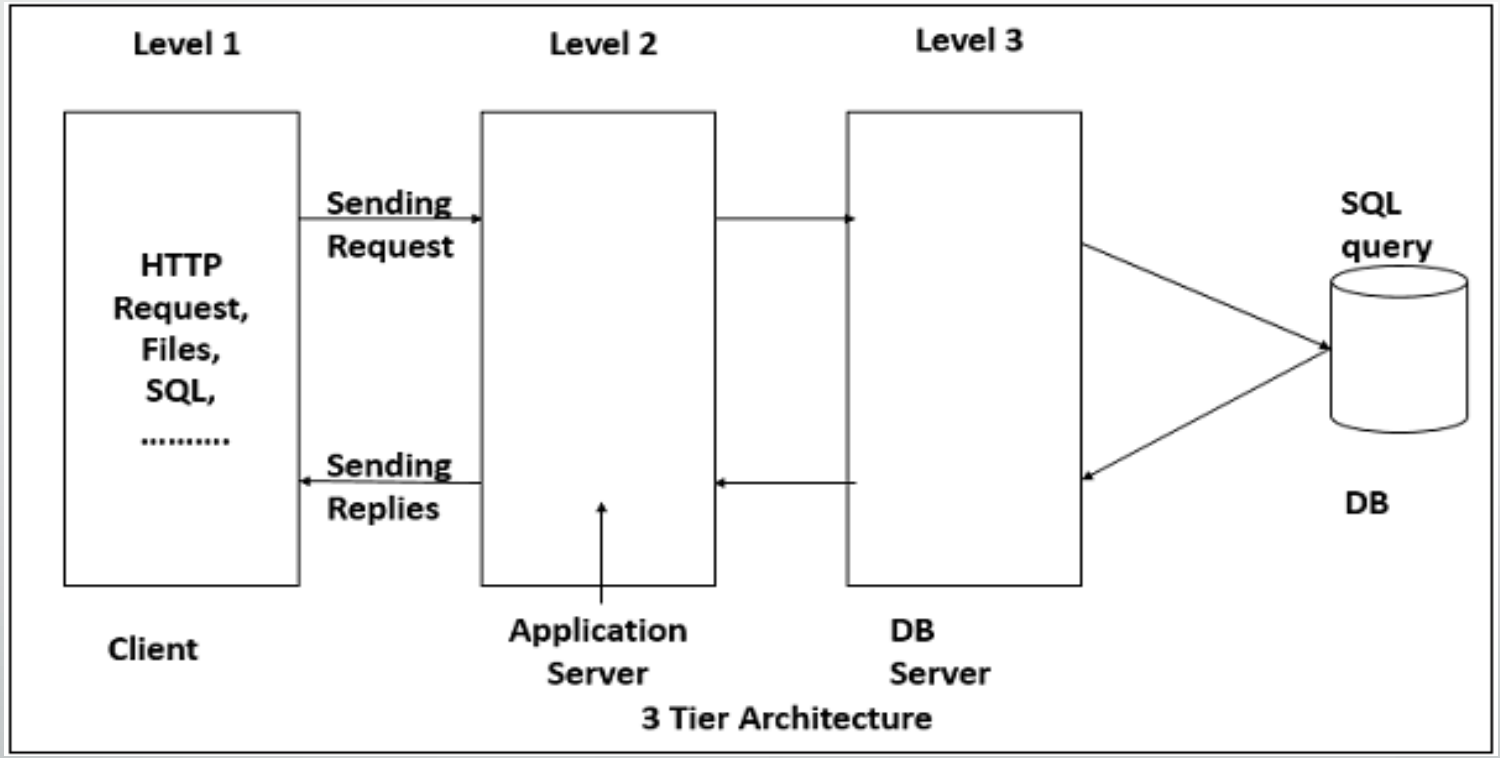
Multi-Tier Architecture (n-tier Architecture)

- Multi-tier architecture is a client–server architecture in which the functions such as presentation, application processing, and data management are *physically* separated.
- By separating an application into tiers, developers obtain the option of changing or adding a specific layer, instead of reworking the entire application.
- It provides a model by which developers can create flexible and reusable applications.

Multi-Tier Architecture (n-tier Architecture)

- The most general use of multi-tier architecture is the three-tier architecture.
- A three-tier architecture is typically composed of a presentation tier, an application tier, and a data storage (discussed in information system design):
  - *Presentation Tier*: It translates the tasks and results to something that user can understand.
  - *Application Tier* (Business Logic, Logic Tier, or Middle Tier): It coordinates the application, processes the commands, makes logical decisions, evaluation, and performs calculations. It controls an application's functionality by performing detailed processing.
  - *Data Tier*: It includes the data persistence mechanisms (database servers, file shares, etc.) and provides *API (Application Programming Interface*) to the application tier which provides methods of managing the stored data.

Multi-Tier Architecture (n-tier Architecture)



Multi-Tier Architecture (n-tier Architecture)

- Advantages:
  - Better performance than a thin-client approach and is simpler to manage than a thick-client approach.
  - Enhances the reusability and scalability – as demands increase, extra servers can be added.
  - Reduces network traffic.
  - Provides maintainability and flexibility.
- Disadvantages:
  - Unsatisfactory testability due to lack of testing tools.
  - More critical server reliability and availability.

Broker Architectural Style

- *Broker Architectural Style* is a middleware architecture used in distributed computing to coordinate and enable the communication between registered servers and clients.
- Here, object communication takes place through a middleware system called an *object request broker (software bus).*
- Client and the server do not interact with each other directly. Client and server have a direct connection to its proxy which communicates with the mediator-broker.
- A server provides services by *registering and publishing their interfaces* with the broker and clients can request the services from the broker by *look-up.*
- *CORBA (Common Object Request Broker Architecture)* is a good implementation example of the broker architecture.

Broker Architectural Style

- Components of Broker Architectural Style:
- **Broker** *(software bus)*:
  - It is responsible for coordinating communication, such as forwarding and dispatching the results and exceptions.
  - It is responsible for brokering the service requests, locating a proper server, transmitting requests, and sending responses back to clients.
  - It retains the servers' registration information including their functionality and services as well as location information.
  - It provides APIs for clients to request, servers to respond, registering or unregistering server components, transferring messages, and locating servers.
- **Stubs:** Generated at the static compilation time and then deployed to the client side and used as a proxy for the client. Client-side proxy acts as a mediator between the client and the broker and provides additional transparency between them.

Broker Architectural Style

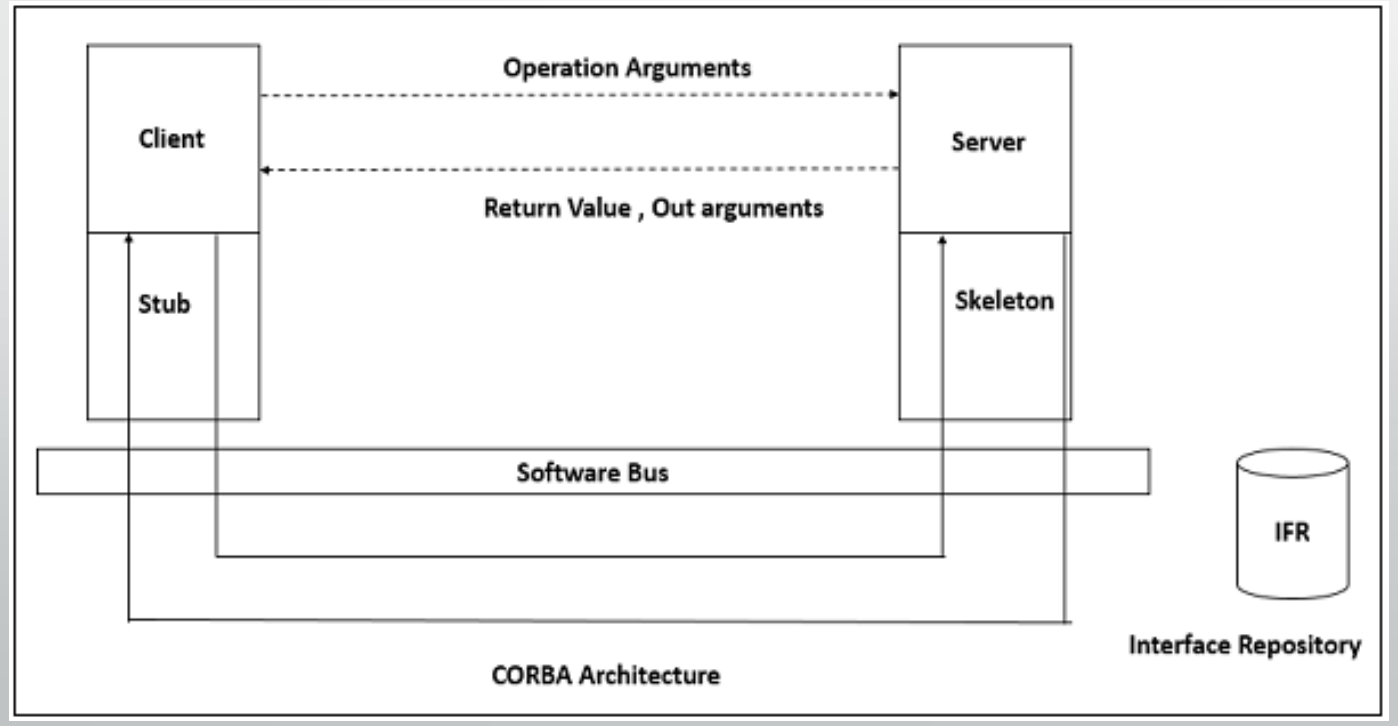
- **Skeleton:** It is generated by the service interface compilation and then deployed to the server side, which is used as a proxy for the server. Server-side proxy encapsulates low-level system-specific networking functions and provides high-level APIs to mediate between the server and the broker.
- It receives the requests, unpacks the requests, *unmarshals* the method arguments, calls the suitable service, and also *marshals* the result before sending it back to the client.
- *Marshalling*: process of taking a collection of data items and assembling them into a form suitable for transmission in a message.
- *Unmarshalling*: process of disassembling a collection data items from a message at the destination.

Broker Architectural Style

- **Bridge:** It can connect two different networks based on different communication protocols and it mediates different brokers.
  - Bridges are optional component, which hides the implementation details when two brokers interoperate and take requests and parameters in one format and translate them to another format.

Broker Implementation in CORBA

- CORBA is an international standard for an *Object Request Broker.*
- It is a middleware to manage communications among distributed objects defined by OMG (Object Management Group).

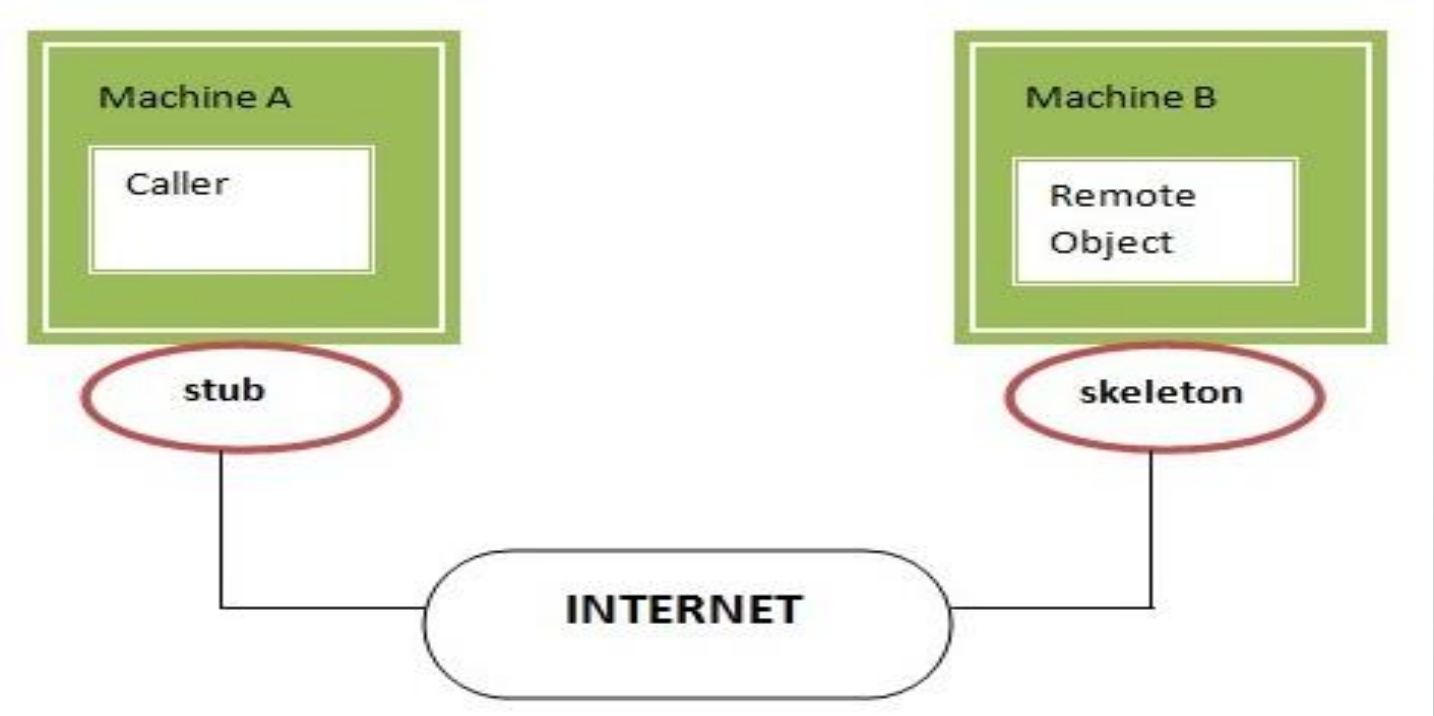


Remote Method Invocation (RMI)

- The **RMI** (Remote Method Invocation) is an API that provides a mechanism to create distributed application in java. The RMI allows an object to invoke methods on an object running in another Java Virtual Machine.
- The RMI provides remote communication between the applications using two objects *stub* and *skeleton*.
- RMI uses stub and skeleton object for communication with the remote object.
- A **remote object** is an object whose method can be invoked from another JVM.

Remote Method Invocation (RMI)

- Let's understand the stub and skeleton objects:



Remote Method Invocation (RMI)

- The stub is an object, acts as a gateway for the client side. All the outgoing requests are routed through it. It resides at the client side and represents the remote object. When the caller invokes method on the stub object, it does the following tasks:
- It initiates a connection with remote Virtual Machine (JVM),
- It writes and transmits (marshals) the parameters to the remote Virtual Machine (JVM),
- It waits for the result,
- It reads (unmarshals) the return value or exception, and
- It finally, returns the value to the caller.

Remote Method Invocation (RMI)

- The skeleton is an object, acts as a gateway for the server side object. All the incoming requests are routed through it. When the skeleton receives the incoming request, it does the following tasks:
- It reads the parameter for the remote method,
- It invokes the method on the actual remote object, and
- It writes and transmits (marshals) the result to the caller.
- In the Java 2 SDK, a stub protocol was introduced that eliminates the need for skeletons.

Remote Method Invocation (RMI)

- If any application performs these tasks, it can be a distributed application.
- The application needs to locate the remote method.
- It needs to provide the communication with the remote objects.
- The application needs to load the class definitions for the objects.
- The RMI application has all these features, so it is called a distributed application.

Remote Method Invocation (RMI)

- There are 6 steps to write java RMI program:
- Create the *remote interface*.
  - Interfaces: definition of the signatures of a set of methods without their implementation.
  - Remote interfaces: which of the object methods can be invoked remotely.
- Compile the implementation class and create the stub and skeleton objects using *rmic tool*.
  - rmic tool is the Java RMI compiler and it generates stubs, skeletons, and ties classes for remote objects using specific protocols.

Remote Method Invocation (RMI)

- Start the registry service using *a binder*.
  - The binder is a separate service that maintains a table containing mappings from textual names to *remote object references*.
  - Remote object reference: identifier that can be used throughout a distributed system to refer to a particular unique remote object.
  - Remote object references may be passed as arguments and results of remote method invocations.
  - The binder is used by: servers to register their remote objects by name and by clients to look them up.
- Create and start the remote application (Server).
- Create and start the client application.

References

- Coulouris, G., Dollimore, J., Kindberg, T., and Blair, G. (2012). Distributed Systems. Concepts and Design. Pearson, 5th edition.
- Tanenbaum, A. S. and van Steen, M. (2007). Distributed Systems. Principles and Paradigms. Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition.