VISION SHARE - SCREEN BROADCASTING USING RMI

(Remote Method Invocation)

A Report Submitted

In Partial Fulfillment of the Requirements

For the degree of

BACHELOR OF TECHNOLOGY

Pooja Agarwal	**Roll No.:201410101110010**
Arushi Tiwari	**Roll No.:201410101110013**
Chandra Shekhar Singh Yadav	**Roll No.: 201410101110017**
Shubhankar Rastiogi	**Roll No.: 201410101110040**

Guide: **Ms. Neha Agarwal**

FACULTY OF COMPUTER SCIENCE & ENGINEERING

INSTITUTE OF TECHNOLOGY,

SHRI RAMSWAROOP MEMORIAL UNIVERSITY

Apr, 2018

Certificate

It is certified that the work contained in the project report titled **SCREEN BROADCASTING USING RMI (Remote Method Invocation)**, by Pooja Agarwal, Chandra Shekhar Singh Yadav, Shubhankar Rastogi and Arushi Tiwari has been carried out under my supervision and that this work has not been submitted elsewhere for any other degree.

Ms. Neha Agarwal

Computer Science and Engineering Department

Shri Ramswaroop Memorial University

Apr,2018

Dr Shalini Agarwal

Computer Science and Engineering Department

Shri Ramswaroop Memorial University

Abstract

In current computing settings, effective communication and collaboration remain essential. This is especially the case in educational and professional environments, where multiple users often need to exchange information in real time. The project Screen Broadcasting using RMI or Remote Method Invocation, develops a network driven approach. It enables a single machine to share its display with various clients linked on the local network.

This initiative draws on Java's RMI API to support real time exchanges across distributed setups. The broadcasting process involves capturing the server's screen at set intervals. It then encodes those images and sends them via the network to client devices, then the frames get displayed as received. The design follows a server centered structure. This setup allows for focused oversight and potential expansion.

Such a method delivers an affordable, cross platform option suited to classroom demos, laboratory activities, and workplace talks. In these situations, several machines must access a shared view. What distinguishes it from commercial applications is the full dependence on Java's core libraries alone. That reliance adopts ease of transfer, straightforward operation, and strong learning potential.

Acknowledgement

A research work owes its success from commencement to completion to the people in love with researchers at various stages. Let me, in this page, express my gratitude to all those who helped us in different stages of this study.

I wish to express my sincere gratitude and indebtedness to **Ms. Neha Agarwal** (Department of Computer Science Engineering, SRMU) for introducing the present topic and for his inspiring guidance, constructive criticism, and valuable suggestions throughout this project work.

I am also thankful to my Parents for their true help and inspiration. Last but not least, I pay my sincere thanks and gratitude to all the staff members and my project partner at **SRMU** for their support and for making our training valuable and fruitful.

Date: Apr 2018	Pooja Agarwal (201410101110010)
Place: Lucknow, UP	Arushi Tiwari (201410101110013)
	Chandra Shekhar Singh Yadav (201410101110017)
	Shubhankar Rastogi (201410101110040)

Declaration

We hereby declare that the project work entitled “**VISION SHARE SCREEN BROADCASTING USING RMI(Remote Method Invocation)**” submitted to the Shri Ramswaroop Memorial University, is a record of original work done by us under the guidance of **Ms. Neha Agarwal**, Faculty of Computer Science and Engineering, Shri Ramswaroop Memorial University, Lucknow and this project work is submitted in the partial fulfilment of the requirements for the award of the degree of Bachelor of technology in Computer Science and Engineering. The results embodied in this report have not been submitted to any other University or Institute for the award of any degree.

Signature:

Date:

Table Of Contents

Table Of Figures

Chapter 1
Introduction

1.1 Introduction

Have you ever felt the need to share the display contents of a computer with others, especially when all participants cannot gather in the same room due to physical constraints or time limitations? Video conferencing may be a good solution, but it is not always ideal when the flow of information is primarily one-way — for instance, when sharing presentation slides or demonstrations. In such cases, display sharing over a local network provides an efficient and practical way to broadcast content so that other users can view it on their own screens without leaving their workstations.

The main inspiration for this project arises from the challenges commonly faced in classrooms, laboratories, and office meetings, where projectors are typically used to display information. But projector-based presentations often suffer from limited visibility, especially for individuals sitting farther away from the screen. The visual details, code snippets, or diagrams projected may not be clearly visible to everyone, leading to reduced understanding and engagement.

To address this problem, one effective approach involves a setup for real-time screen sharing across a local network. This method shares the screen from a single computer to all devices linked to the same LAN. As a result, participants gain a clear view of the current task or display right on their individual screens. Client devices connect easily to the main broadcasting machine, which functions as the server. They pull in the live feed with no need for extra equipment or configuration.

This project focuses on developing a screen broadcasting setup through Java Remote Method Invocation, or RMI. That framework supports smooth interactions across distributed systems. It does this by letting one Java Virtual Machine call methods on objects in a different JVM. In this project, RMI is used to transmit the captured screen images from the server system to all connected client systems in a synchronized manner.

Unlike traditional socket programming, where developers need to handle data streams and protocols manually, RMI simplifies the process by providing an object-oriented approach to remote communication. It takes care of network connections, data serialization, and transfer internally, which makes the system easier to develop and maintain.

At present, quite a few screen-sharing applications out there are commercial in nature. They demand administrative privileges to operate, or they lean on external servers linked up through the internet. Solutions like these tend to fall short in educational settings and lab configurations. Network restrictions and spotty internet access were pretty standard in those places. This project stepped in to tackle such drawbacks. It develops a Java-based tool for broadcasting over local networks. The setup allows screen sharing with no reliance on third-party platforms or any paid services.

This setup demonstrates how distributed computing ideas are applied in real-world group work situations. It highlights how various machines connected on the same network manage to share image information smoothly. They accomplish this using basic Java tools. No extra add-ons or setup steps are needed at all. The solution is scalable, portable, and applicable across different operating systems that support Java.

This project basically delivers an affordable option for broadcasting screens live across a local network. It works without depending on specific platforms. Plus, it stays easy for users to handle. All of this helps improve how people share information in talks, classes, and meetings. Everyone gets the same clear view of the visuals. In turn, that boosts engagement and makes communication run smoothly overall.

1.2 <u>Objective of the Project</u>

The main objective of this project is to design and develop a system that can share the screen of one computer with other computers connected to the same local network. The project aims to make the sharing process simple, reliable, and efficient without using any external hardware devices or internet-based services.

The specific objectives of the project are:

- To capture and transmit the screen of the server computer in real time.
- To allow multiple client systems in the same network to view the broadcasted screen simultaneously.
- To maintain smooth performance and low delay while broadcasting over a local area network (LAN).
- To demonstrate the practical implementation of distributed computing using Java RMI.

This project also helps students and developers understand how remote method invocation can be used to create communication between systems in a distributed environment.

1.3 <u>Purpose</u>

This project aims to offer a straightforward, affordable, and open-source option instead of those paid screen-sharing or remote-desktop programs you see on the market. A lot of the tools out there cost money, take time to set up properly, or depend on an internet connection, so they do not fit well in local school or work settings.

The setup works best in classrooms and labs, where instructors can share their screen with every student's machine all at the same time. That way, there is no reliance on a projector, and each student gets a clear view of the material being covered. On top of that, the whole thing shows how Java RMI helps create network applications that run across various platforms with just a little initial configuration.

1.4 <u>Project Scope</u>

The scope of this project covers the design and implementation of a screen broadcasting system that works over a local network using Java RMI. The system can be used in different environments, such as:

- Classrooms and laboratories for teaching and demonstration purposes.
- Corporate offices to share presentations and reports during internal meetings.
- Training centers or workshops where all participants need to view the same content.

The system is developed to work on any computer that supports Java, making it platform-independent. It can be further enhanced by adding new features like remote control, user authentication, data encryption, and cross-platform support. Such improvements will make the system more secure and capable of handling larger networks with multiple clients.

Chapter 2
Java RMI

2.1 Introduction

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 JVM. RMI is used to build distributed applications; it provides remote communication between Java programs. It is provided in the package java.rmi.

The **Java Remote Method Invocation (Java RMI)** is a Java API that performs remote method invocation, the object-oriented equivalent of Remote procedure calls (RPC), with support for direct transfer of serialized Java classes and distributed garbage-collection.

The original implementation depends on Java Virtual Machine (JVM) class-representation mechanisms and it thus only supports making calls from one JVM to another. The protocol underlying this Java-only implementation is known as **Java Remote Method Protocol (JRMP)**.

2.2 Architecture of RMI

In an RMI application, we write two programs, a server program (resides on the server) and a client program (resides on the client).

- Inside the server program, a remote object is created and reference of that object is made available for the client (using the registry).
- The client program requests the remote objects on the server and tries to invoke its methods.

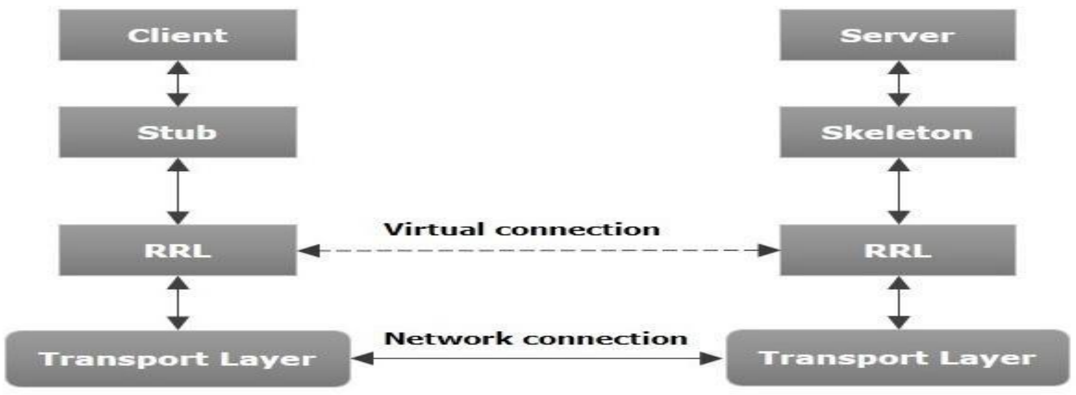


Figure 2.2.1 Architecture of RMI Application

The Components of RMI are as follows-

- ***Transport Layer*** − This layer connects the client and the server. It manages the existing connection and also sets up new connections.

- ***Stub*** − A stub is a representation (proxy) of the remote object at client. It resides in the client system; it acts as a gateway for the client program.
- ***Skeleton*** − This is the object which resides on the server side. Stub communicates with this skeleton to pass request to the remote object.
- ***RRL (Remote Reference Layer)*** − It is the layer which manages the references made by the client to the remote object.

2.3 <u>Working of RMI Application</u>

The following points summarize how an RMI application works −

- When the client makes a call to the remote object, it is received by the stub which eventually passes this request to the RRL.
- When the client-side RRL receives the request, it invokes a method called invoke() of the object remoteRef. It passes the request to the RRL on the server side.
- The RRL on the server side passes the request to the Skeleton (proxy on the server) which finally invokes the required object on the server.
- The result is passed all the way back to the client.

2.4 <u>Understanding Stub and Skeleton</u>

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. Let's understand the stub and skeleton objects:

2.4.1 stub

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.

2.4.2 skeleton

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.

2.5 Requirement for Distributed Application

If any application performs these tasks, it can be distributed application.

- The application needs to locate the remote method
- It needs to provide the communication with the remote objects, and
- The application needs to load the class definitions for the objects.

The RMI application have all these features, so it is called the distributed application.

2.6 Java RMI Example

The is given the 6 steps to write the RMI program.

1. Create the remote interface
2. Provide the implementation of the remote interface
3. Compile the implementation class and create the stub and skeleton objects using the rmic tool
4. Start the registry service by rmiregistry tool
5. Create and start the remote application
6. Create and start the client application

2.7 RMI Example

In this example, we have followed all the 6 steps to create and run the rmi application. The client application need only two files, remote interface and client application. In the rmi application, both client and server interact with the remote interface. The client application invokes methods on the proxy object, RMI sends the request to the remote JVM. The return value is sent back to the proxy object and then to the client application.

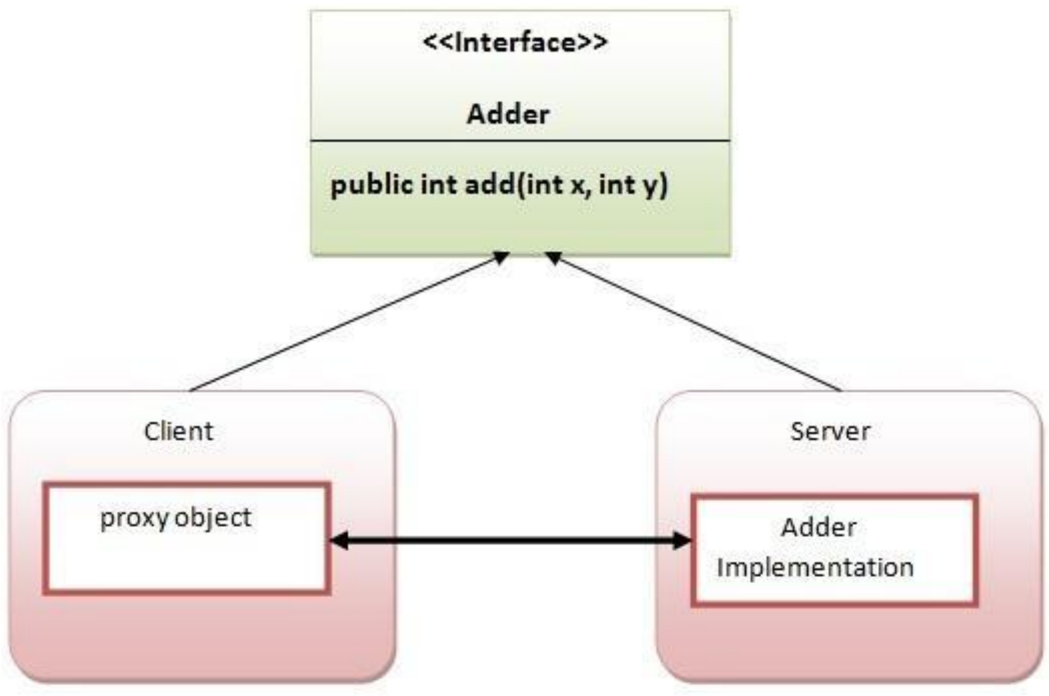


Figure 2.7.1 Implementation of RMI Example

1) Create the Remote Interface

For creating the remote interface, extend the Remote interface and declare the RemoteException with all the methods of the remote interface. Here, we are creating a remote interface that extends the Remote interface. There is only one method named add() and it declares RemoteException.

```
import java.rmi.*;

public interface Adder extends Remote {

        public int add(int x,int y)throws RemoteException;

}
```

2) Provide the implementation of the Remote Interface

Now provide the implementation of the remote interface. For providing the implementation of the Remote interface, we need to either extend the UnicastRemoteObject class, or use the exportObject() method of the UnicastRemoteObject class.

In case, you extend the UnicastRemoteObject class, you must define a constructor that declares RemoteException.

```
import java.rmi.*;
import java.rmi.server.*;
public class AdderRemote extends UnicastRemoteObject implements Adder {
```

```
            AdderRemote() throws RemoteException {
                    super();
            }
            public int add(int x,int y){
                    return x+y;
            }
    }
```

3) Create the stub and skeleton objects using the rmic tool.

Next step is to create stub and skeleton objects using the rmi compiler. The rmic tool invokes the RMI compiler and creates stub and skeleton objects.

```
rmic AdderRemote
```

4) Start the registry service by the rmiregistry tool

Now start the registry service by using the rmiregistry tool. If you don't specify the port number, it uses a default port number. In this example, we are using the port number 5000.

```
rmiregistry 5000
```

5) Create and run the server application

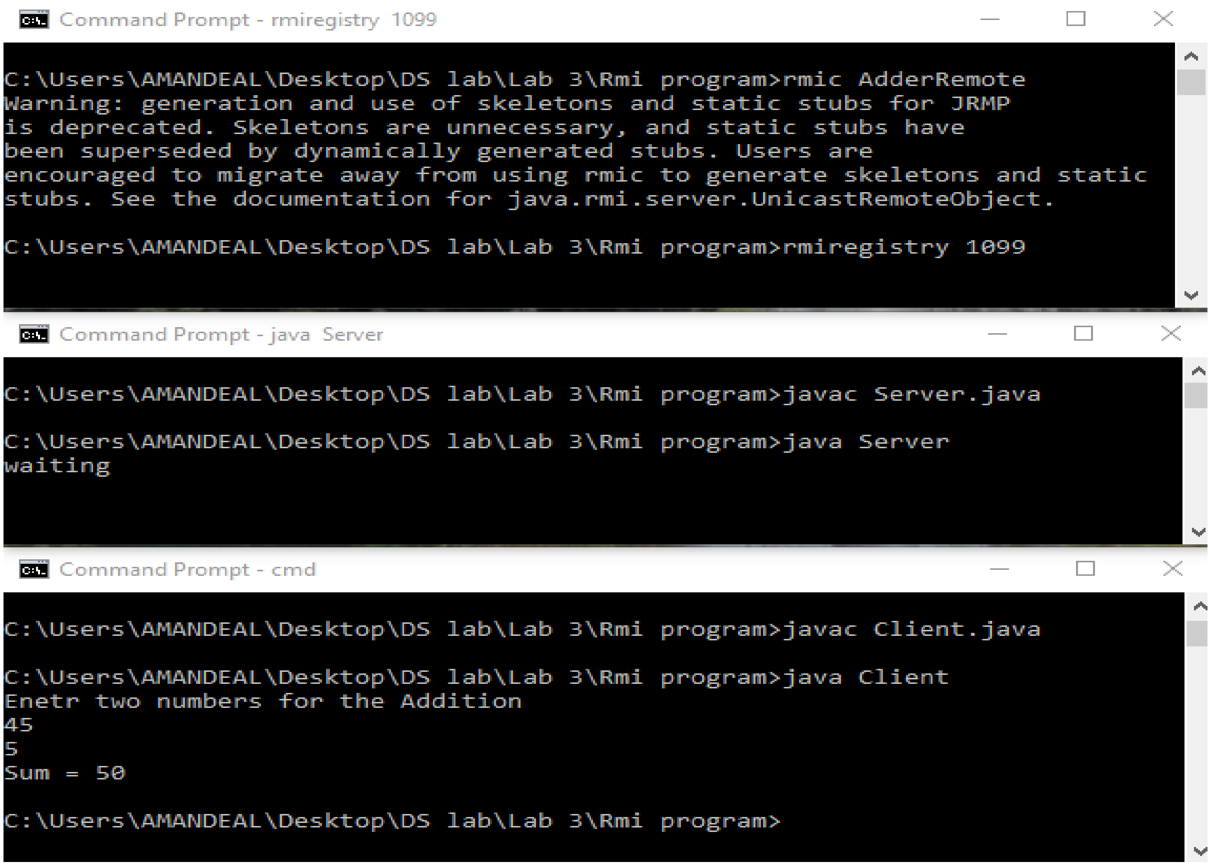
Now rmi services need to be hosted in a server process. The Naming class provides methods to get and store the remote object.

```
import java.rmi.*;
import java.rmi.registry.*;
public class MyServer {
        public static void main(String args[]) {
                try {
                        Adder stub=new AdderRemote();
                        Naming.rebind("rmi://localhost:5000/sonoo",stub);
                } catch(Exception e) {
                        System.out.println(e);
                }
        }
}
```

6) Create and run the client application

At the client we are getting the stub object by the lookup() method of the Naming class and invoking the method on this object. In this example, we are running the server and client applications, in the same machine so we are using localhost. If you want to access the remote object from another machine, change the localhost to the host name (or IP address) where the remote object is located.

```
import java.rmi.*;
public class MyClient{
        public static void main(String args[]) {
                try {
                  Adder stub=(Adder)Naming.lookup("rmi://localhost:5000/sonoo");
                  System.out.println(stub.add(34,4));
                } catch(Exception e) { }
        }
}
```



Command Prompt - rmiregistry 1099

```
C:\Users\AMANDEAL\Desktop\DS lab\Lab 3\Rmi program>rmic AdderRemote
Warning: generation and use of skeletons and static stubs for JRMP
is deprecated. Skeletons are unnecessary, and static stubs have
been superseded by dynamically generated stubs. Users are
encouraged to migrate away from using rmic to generate skeletons and static
stubs. See the documentation for java.rmi.server.UnicastRemoteObject.

C:\Users\AMANDEAL\Desktop\DS lab\Lab 3\Rmi program>rmiregistry 1099
```

Command Prompt - java Server

```
C:\Users\AMANDEAL\Desktop\DS lab\Lab 3\Rmi program>javac Server.java

C:\Users\AMANDEAL\Desktop\DS lab\Lab 3\Rmi program>java Server
waiting
```

Command Prompt - cmd

```
C:\Users\AMANDEAL\Desktop\DS lab\Lab 3\Rmi program>javac Client.java

C:\Users\AMANDEAL\Desktop\DS lab\Lab 3\Rmi program>java Client
Enetr two numbers for the Addition
45
5
Sum = 50

C:\Users\AMANDEAL\Desktop\DS lab\Lab 3\Rmi program>
```

Figure 2.7.2 Output of the RMI Example

2.8 Marshalling and Unmarshalling

Whenever a client invokes a method that accepts parameters on a remote object, the parameters are bundled into a message before being sent over the network. These parameters may be of primitive type or objects. In case of primitive type, the parameters are put together and a header is attached to it. In case the parameters are objects, then they are serialized. This process is known as **marshalling**.

At the server side, the packed parameters are unbundled and then the required method is invoked. This process is known as **unmarshalling**.

2.9 RMI Registry

RMI registry is a namespace on which all server objects are placed. Each time the server creates an object, it registers this object with the RMIregistry (using bind() or reBind() methods). These are registered using a unique name known as bind name.

To invoke a remote object, the client needs a reference of that object. At that time, the client fetches the object from the registry using its bind name (using lookup() method).

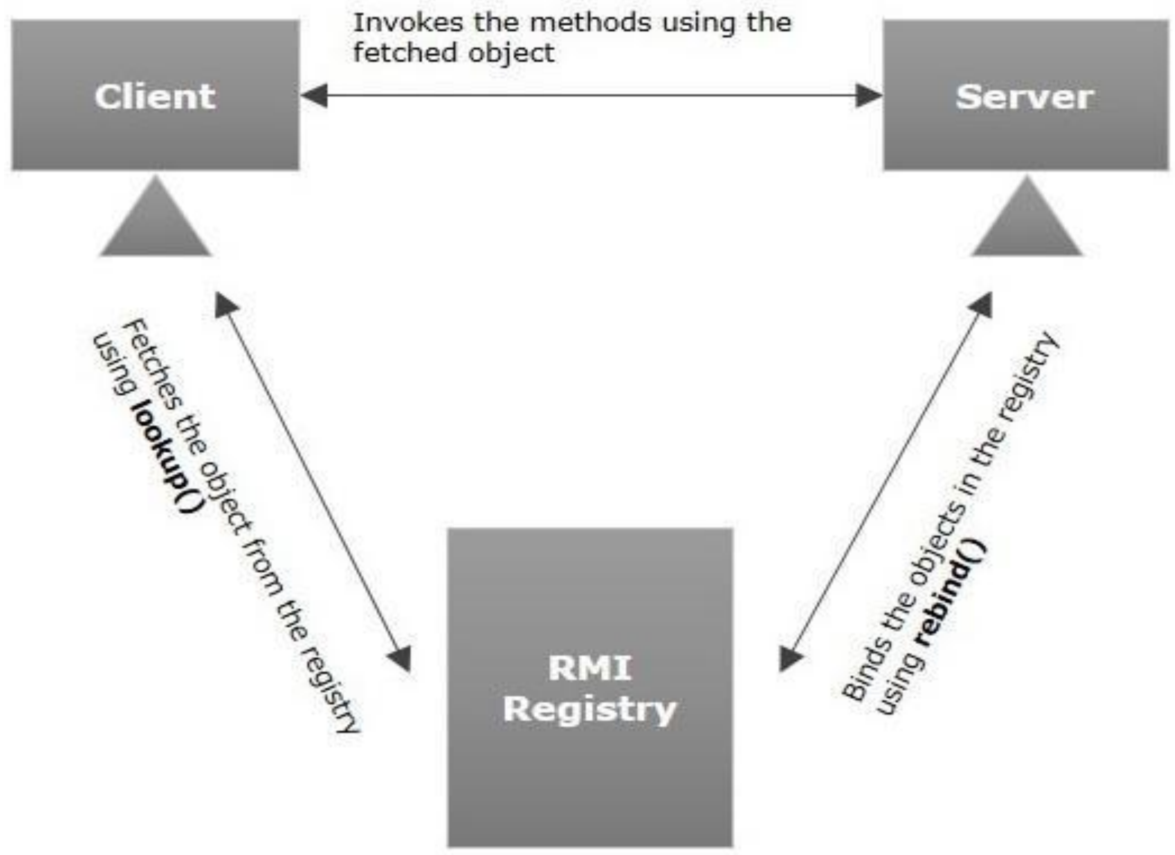

Client
Invokes the methods using the fetched object
Server
Fetches the object from the registry using **lookup()**
Binds the objects in the registry using **rebind()**
RMI Registry

Figure 2.9.1 Process of RMI Registry

Chapter 3
Flowchart

3.1 Flow Diagram

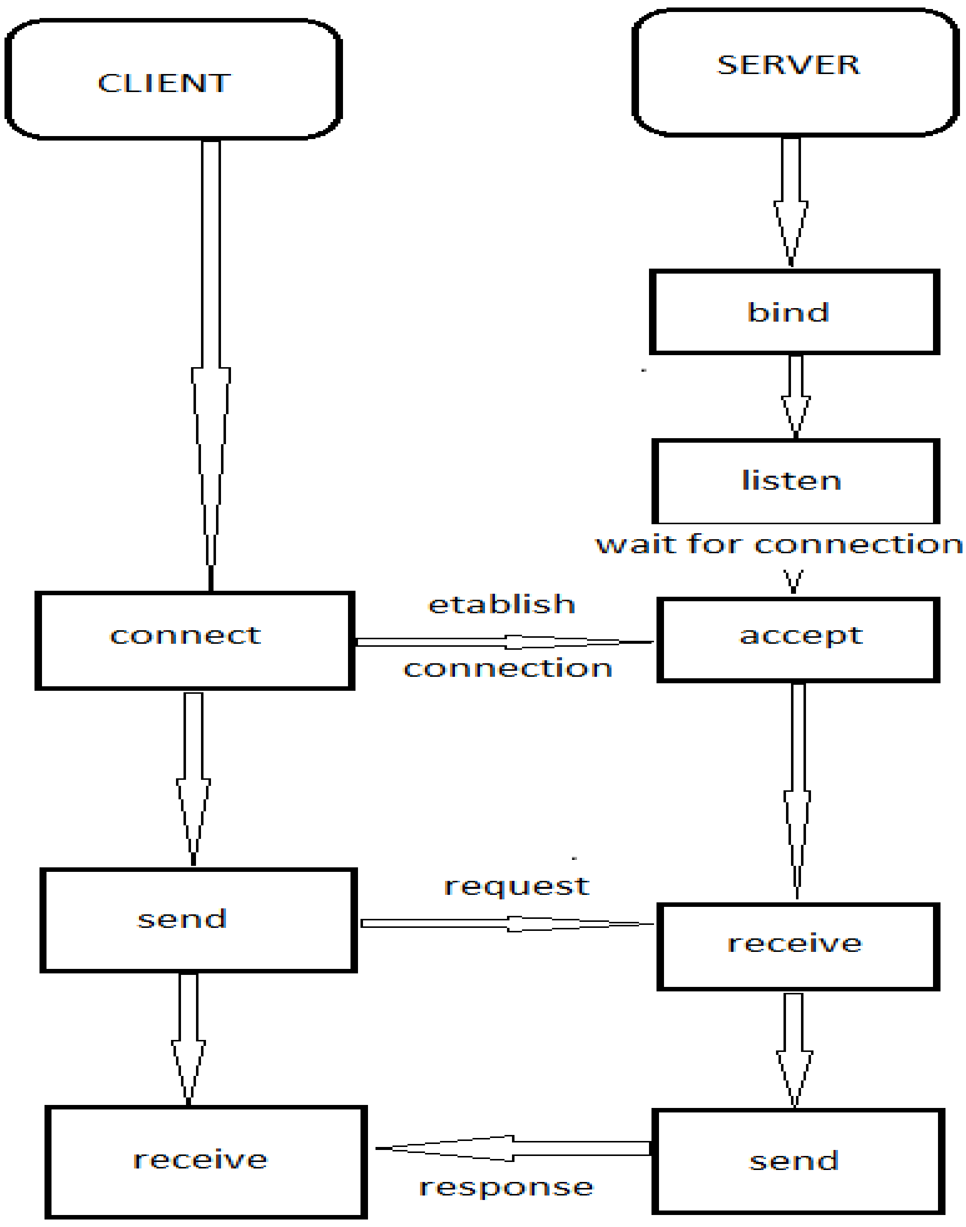


Figure 3.1.1 Flow Diagram of the Screen Broadcasting Process

3.2 Steps Involved in Screen Broadcasting Process

1) Initialize the server and wait for connection

2) Start listening: start the registry for listening the server is ready to take request from the clients
3) Wait for connection: now server waits for the request from the clients
4) Open the client site enter the IP of the server to connect and wait for the response.
5) Server receive the requests and the in response broadcasts its screen to the connected client
6) The client receives the response able to see the screen of the server from remote location

Chapter 4
Software and Hardware Requirements

4.1 Platforms Used

- Eclipse
- Java Development kit
- Libraries Used: RMI-IIOP, RMIIO

4.1.1 RMI-IIOP

- It is read as "RMI over IIOP".
- It denotes the Java Remote Method Invocation (RMI) interface over the Internet Inter-Orb Protocol (IIOP), which delivers CORBA distributed computing capabilities to the Java platform.
- RMI-IIOP allows developers to pass any Java object between application components either by reference or by value.
- IIOP eases legacy application and platform integration by allowing the application components written in any CORBA supported languages to communicate with components running on the Java platform.
- Initially it was based on two specifications:
- Java Language Mapping to OMG IDL (Where OMG is "Object Management Group" and IDL is "Interface Definition Language").
- CORBA/IIOP
- With this inherited feature of CORBA, it supports multiple platforms and can make remote procedure calls to execute, subroutines on another computer as defined by RMI.

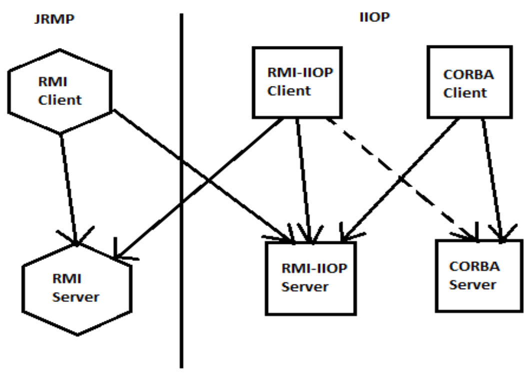


Figure 4.1.1 Introduction to RMI-IIOP

In Fig:4.1.1 above,

- The left part is representing the RMI (JRMP) model, the middle part the RMI-IIOP model, and the right part represents the CORBA model.
- Arrows are representing a situation in which a client can call a server. RMI-IIOP belongs in the IIOP world below the vertical line.
- The diagonal arrows crossing the border between the JRMP world and the IIOP world, implies that an RMI (JRMP) client can call an RMI-IIOP server, and vice versa.
- RMI-IIOP supports both JRMP and IIOP protocols.

4.1.2 RMIIO

RMIIO is a library which makes it as simple as possible to stream large amounts of data using the RMI framework (or any RPC framework for that matter). Who needs this? Well, if you have ever needed to send a file from an RMI client to an RMI server, you have faced this problem. And, if you did manage to implement a basic solution, it probably threw an OutOfMemoryError the first time someone tried to send a 2GB file. Due to the design of RMI, this common and deceptively simple problem is actually quite difficult to solve in an efficient and robust manner.

The RMI framework makes it very easy to implement remote communication between java programs. It takes a very difficult problem (remote communication) and presents a fairly easy to use solution. However, the RMI framework is designed around sending and receiving groups of objects which are all immediately available in memory.

The Features of RMIIO are as follows –

- Remote input and output streams
- Remote Iterator implementation for streaming collections of objects (SerialRemoteIteratorServer)
- Optional GZIP compression over the wire (input and output streams)
- Stream progress monitoring hooks (RemoteStreamMonitor)
- Optional low-latency streaming (noDelay)
- Serializable InputStream and OutputStream wrappers for remote input and output streams
- Pluggable RPC integration which can be used to integrate with frameworks other than RMI (see the RemoteStreamExporter section below for details)
- Utilities to facilitate robust RMI usage (RemoteRetry)

4.2 <u>Hardware Requirement</u>

- Windows 10,8.1,7
- 4–6 GB for a typical installation.

- Any Intel or AMD x86-64 processor
- No specific graphics card is required

Chapter 5
Literature Review

5.1 Introduction

Screen broadcasting and remote display systems had been an active area of study for many years. Several software solutions were developed to provide remote screen access using different communication protocols and network architectures. This chapter presents an overview of the existing technologies related to screen broadcasting and explains how Java RMI fits into this context.

5.2 Existing Technologies for Remote Screen Sharing

Several tools and protocols have been developed to provide remote screen sharing and desktop access. Some of the commonly used technologies are described below.

VNC, developed by AT&T Laboratories, transmitted screen updates as images using the Remote Frame Buffer (RFB) protocol. It was cross-platform and widely used but its performance was limited on low-speed networks because of the high volume of image data transferred.

Microsoft RDP, a proprietary protocol built into the Windows operating system, provided faster screen encoding and better session management. However, it was platform-dependent and worked mainly on Windows-based systems.

Other tools such as **TeamViewer** and **AnyDesk** also became popular because of their user-friendly interfaces and internet-based connectivity. These tools allowed remote access and file sharing but required constant internet connection and were not open-source.

Although these applications worked effectively, they were not mainly designed for educational or local network demonstrations. Most of them required administrative access or external servers to establish connections, which made them unsuitable for simple LAN-based broadcasting.

5.3 Distributed Systems and Java Technologies

Java RMI had become a widely used approach for implementing distributed applications. Alternatives included:

- **CORBA (Common Object Request Broker Architecture),** which allowed interoperability across languages but was complex to configure.
- **Java Socket Programming**, which provided lower-level control but required manual management of connections, serialization, and data handling.

- **JMS (Java Message Service)**, suited for asynchronous communication but less efficient for real-time streaming.

RMI provided an optimal balance between ease of development and functionality. It enabled method invocation across JVMs without dealing with network-level complexities, making it ideal for academic demonstrations and prototypes like *Vision share - Screen Broadcasting using RMI.*

5.4 Comparison of Technologies

Many technologies are available for developing distributed applications and remote communication systems. Each technology has its own advantages and disadvantages depending on its use and the level of complexity involved. The following section compares Java RMI, CORBA, Socket Programming, and VNC / RDP based on their working and suitability for screen broadcasting applications.

5.4.1 Java RMI (Remote Method Invocation)

Java RMI is a Java-based API used to build distributed applications where one computer can call methods on another computer in the same or different machine. It automatically manages network communication, object transfer, and remote method calls. RMI is easy to implement and understand for Java programmers because it uses standard Java classes and interfaces.

It is best suited for small-scale or educational projects that work within a local area network (LAN). However, RMI works only in Java environments and may not perform well for large multimedia data without extra optimization.

5.4.2 CORBA (Common Object Request Broker Architecture)

Socket programming is one of the oldest ways to connect computers over a network. It allows the direct sending and receiving of data using the TCP/IP protocol. It gives the programmer full control over how data is transmitted, which makes it flexible and fast.

However, it also increases the complexity because the developer has to manage all parts of communication manually, such as connection handling, data transfer, and synchronization. It is useful for real-time applications that need custom communication, but not ideal for simple or demonstration-based systems.

5.4.3 VNC (Virtual Network Computing) / RDP (Remote Desktop Protocol)

VNC and RDP are popular tools for remote desktop access. VNC is an open-source protocol that sends screen updates as images to the client system. It is platform-independent but can be slow on low-speed networks because large image data needs to be transmitted.

RDP, developed by Microsoft, provides better performance by compressing graphical data. It also supports file transfer and clipboard sharing. However, RDP is limited to Windows platforms, while VNC, though open-source, is not designed mainly for one-way screen broadcasting. Both are useful for remote control but not suitable for simple local broadcasting setups.

5.5 Overall Comparison

Technology	Advantages	Limitations	Suitable Use Case
Java RMI	Easy to use, Java-based, automatic communication	Works only in Java, not for heavy data	Educational or LAN projects
CORBA	Cross-platform, supports multiple languages	Complex setup and configuration	Enterprise distributed systems
Socket Programming	High control, fast communication	Requires manual handling, more coding	Custom or real-time applications
VNC / RDP	Mature technology, stable and tested	Bandwidth dependent, not ideal for LAN broadcasting	Remote desktop and administration

5.6 Summary

From this comparison it can be seen that **Java RMI** is the most suitable technology for implementing a simple and effective screen broadcasting system within a local network. It is easy to develop, understand, and maintain, and it provides all the features needed for communication between server and client machines. Other technologies like CORBA or socket programming are more complex, while VNC and RDP are designed for remote control rather than educational broadcasting. Hence, RMI provides the best balance between simplicity and functionality for this project.

Chapter 6
System Design and Implementation Details

6.1 System Architecture

The system is based on a client–server architecture, where one computer acts as the server and broadcasts its screen to other computers that act as clients. The server captures its screen, converts it into an image format, and sends the data to all connected clients through Java RMI. Each client receives the transmitted data and displays it as a continuous stream of images, creating the effect of real-time screen sharing.

The Java RMI framework is used for communication between the server and clients. It allows the server to expose remote objects and the clients to invoke methods on these objects. The RMI registry acts as a directory service where the server registers its remote objects, and clients use this registry to look up and connect to the broadcasting server.

This architecture provides a simple and efficient way to implement screen broadcasting without using external libraries or complex network configurations. The design also makes the system portable and easy to deploy on any platform that supports Java.

6.2 Major Components

The system mainly consists of the following components:

Server Module: The server module captures the computer screen using Java's built-in Robot class and converts the image into a compressed format. The image is then sent to the connected clients through RMI remote method calls. The server also handles multiple client connections and ensures that data is transmitted continuously.

Client Module: The client module connects to the server using the RMI registry and receives the image data. The received data is converted back into image format and displayed on the client's screen using Java's BufferedImage and graphical components. The client continuously updates the display to reflect the latest screen images received from the server.

Network Communication Layer: This layer handles all communication between the server and the clients. It is managed automatically by the RMI framework, which takes care of establishing connections, transferring data, and handling remote method invocations.

RMI Registry: The RMI registry is a naming service that stores references to remote objects. It allows clients to locate and access the broadcasting server by looking up the registered object name or IP address.

6.3 Implementation Details

The system was developed using the Java programming language in the Eclipse IDE environment with JDK 8. The following key features and classes were used in the implementation:

Screen Capture: The screen was captured using the Robot class and the createScreenCapture(Rectangle) method to obtain the display image.

Image Conversion: The captured screen image was converted into a compressed format (JPEG or PNG) using the ImageIO.write() method. This reduced the data size before transmission.

Data Transmission: The converted image data was sent from the server to the clients using the remote interface defined in Java RMI. RMI automatically handled object serialization and network communication.

Client Display: The client received the image data in byte format, converted it back into an image using the ImageIO.read() method, and displayed it using Java Swing components.

Threading: Separate threads were used to handle screen capture and data transmission on the server side to ensure smooth and uninterrupted broadcasting.

This implementation successfully demonstrated the use of Java RMI for transmitting visual data between systems. The system performed effectively in a local network environment and required no additional configuration other than the RMI setup.

6.5 Performance Considerations

The system was designed to work efficiently within a local area network (LAN) environment. Since the communication between the server and client systems takes place over the network, the overall performance of the application depends on factors such as network speed, image size, and screen refresh rate.

During testing, it was observed that the broadcasting delay mainly depended on the resolution of the captured screen and the number of connected clients. The use of image compression helped to reduce the size of transmitted data, which improved the response time. Threading was used on the server side to handle multiple client connections simultaneously and to ensure continuous data transmission without blocking the main process.

Although the system performed well under LAN conditions, its efficiency may decrease with an increase in the number of clients or in lower network bandwidths. The performance can be further improved by implementing advanced compression techniques and by optimizing the screen capture rate according to network conditions.

6.6 Security Aspects

The Java RMI framework provides a built-in security manager that helps to prevent unauthorized access and execution of code on remote systems. In this project, the default RMI security manager was used to manage permissions and to ensure that only the required classes and objects were executed during remote communication.

Since the main focus of this project was on developing and testing the screen broadcasting functionality, advanced security features such as encryption and authentication were not implemented. However, these features can be added in future improvements to make the system more secure and suitable for real-world use.

6.7 Limitations

Although the project successfully achieved its main objectives, there are some limitations in the current implementation. The performance of the system decreases when the number of clients increases, as the server needs to process and transmit more data simultaneously. Continuous screen capturing and image transmission also consume considerable CPU and memory resources, especially on systems with limited hardware capabilities.

The system works efficiently in a local area network but has not been tested for wide area networks (WAN) or internet-based broadcasting. The absence of encryption and authentication makes it unsuitable for transmitting confidential information. Moreover, the system does not include audio streaming or control features, which could make it more interactive.

These limitations can be addressed in future work by adding data compression, encryption, and improved synchronization mechanisms to enhance the overall performance and security of the system.

Chapter 7
Testing, Results and Future Scope

7.1 Testing Strategy

The testing of the system was carried out in a local area network (LAN) environment using multiple client computers connected to a single server. The main purpose of testing was to check the functionality, performance, and stability of the system. Different types of tests were performed to ensure that the application works correctly and efficiently under normal network conditions.

The following types of tests were conducted:

Functional Testing: This test was performed to check the basic operations of the system such as establishing connections between the server and clients, starting the broadcasting process, and verifying that the screens of the clients were updating correctly.

Performance Testing: This test was done to monitor important performance factors such as screen refresh rate, delay between server and client, CPU usage, and memory consumption.

Stability Testing: This test was carried out to ensure that the system remains stable during continuous broadcasting sessions and that there are no crashes or connection drops during long usage periods.

7.2 Observations and Results

After testing, the following observations were made:

- The average delay between the server and the client systems was around 0.8 to 1.2 seconds.
- The screen refresh rate was maintained at around 1 to 2 frames per second, depending on the screen resolution and network speed.
- No data loss, corruption, or connection failure was observed during LAN testing.
- The CPU usage on the server system remained between 30% to 40% during medium-resolution broadcasting.

From these results, it can be concluded that the system worked successfully in transmitting the server screen to all connected clients in real-time within the same local network. The performance was stable, and the system responded well under normal LAN conditions.

7.3 <u>Challenges Encountered</u>

During the development and testing of the project, several challenges were faced, which helped in understanding the limitations and possible improvements of the system. Some of the major challenges were as follows:

- Managing synchronization between multiple clients connected to the server.
- Reducing delay by optimizing image compression and transmission.
- Handling exceptions and errors that occurred when clients disconnected suddenly during broadcasting.

These challenges provided valuable learning experiences and helped in identifying areas for further enhancement of the system.

7.4 <u>Future Scope</u>

Although the system achieved its main objectives, there is still scope for improvement and further development. Some of the enhancements that can be made in the future are:

- Integration of advanced data compression techniques such as JPEG Delta Encoding to reduce data size and improve transmission speed.
- Implementation of encryption and authentication to make the data transmission more secure.
- Addition of audio broadcasting and chat features to enable real-time communication between users.
- Extension of the system to work over wireless networks and the internet with buffering mechanisms to handle variable speeds.
- Development of a more user-friendly graphical interface for easier configuration, control, and monitoring of the broadcasting process.

These improvements would make the system more robust, secure, and suitable for a wider range of real-world applications.

7.5 <u>Conclusion</u>

The project **“Vision Share - Screen Broadcasting Using RMI”** successfully demonstrates the use of Java RMI for building distributed multimedia applications. It shows how the features of Java’s standard libraries can be used to develop a simple yet effective system for screen sharing over a local network.

The system is easy to implement, does not require any external tools, and can be used for educational and professional purposes. The results of the testing proved that the system performs well under LAN conditions and provides a practical solution for real-time screen broadcasting.

Overall, the project serves as a good example of applying distributed computing concepts in real-life applications and can be further improved and extended for advanced research or industrial use.

Chapter 8
Scientific Evaluation and Discussion

8.1 Introduction

This chapter presents the scientific evaluation of the Screen Broadcasting Using RMI system. The main purpose of this evaluation was to study how efficiently Java RMI can be used for real-time screen broadcasting within a local area network (LAN) environment. The analysis focuses on performance factors such as delay, frame rate, CPU usage, and network utilization. The evaluation also discusses the results in relation to distributed system concepts and identifies the areas where the system can be improved.

8.2 Objective of the Evaluation

The objective of this evaluation was to verify whether the proposed system could achieve stable and efficient screen broadcasting using Java RMI and to determine how well it performs when multiple clients are connected to the same network.
The key questions addressed were:

1. Can Java RMI provide acceptable latency for real-time screen broadcasting in a LAN?
2. How does the system behave when the number of connected clients increases?
3. What are the resource requirements in terms of CPU and bandwidth utilization?

8.3 Experimental Setup

The testing was conducted in a laboratory environment with one system acting as a server and up to four client systems connected over a LAN. The setup details are as follows:

Parameter	Description
Server Configuration	Intel Core i5, 2.6 GHz CPU, 8 GB RAM, Windows 10, JDK 8
Client Configuration	Intel Core i3, 4 GB RAM, Windows 10, JDK 8
Network Type	Wired LAN (100 Mbps)
Image Format	JPEG (Medium Compression)
Frame Capture Rate	1 frame per second

During testing, the server captured and broadcasted the screen to all connected clients using Java RMI. The clients displayed the received frames continuously, simulating real-time broadcasting.

8.4 Observations and Results

The following observations were recorded during the evaluation process:

Metric	Average Value	Remarks
Average Delay	0.9 – 1.2 seconds	Acceptable for classroom or demonstration use
Frame Refresh Rate	1–2 frames per second	Dependent on image resolution
CPU Utilization (Server)	35–40%	Stable during continuous broadcasting
Memory Utilization (Server)	Around 500 MB	Moderate for medium image resolution
Maximum Clients Supported	4	Performance degraded beyond 4 clients

8.5 Discussion

The performance analysis showed that Java RMI is suitable for implementing real-time broadcasting in small LAN-based environments. The delay observed was primarily caused by two factors:

Image Encoding and Transmission Delay: The conversion of captured screens into compressed images before transmission added processing time on the server side.

Client-Side Rendering Delay: The received image needed to be decoded and redrawn on the client's graphical interface, causing a small rendering delay.

Despite these delays, the system performed efficiently in local networks with sufficient bandwidth. Compared to traditional socket programming, RMI simplified communication by automatically handling object serialization and data transfer. This reduced coding complexity and minimized errors related to manual data handling.

The results also confirmed that the system was stable during long-duration tests and that RMI's internal thread management handled multiple clients effectively. However, as the number of clients increased beyond four, network traffic and processing load on the server began to affect the frame rate and response time.

8.6 Comparative Analysis

A brief comparison between Java RMI and alternative technologies is presented below:

Technology	Performance	Ease of Implementation	Suitability for LAN Broadcasting

Java RMI	Good	Easy	Highly Suitable
Socket Programming	Excellent	Complex	Suitable but harder to implement
CORBA	Moderate	Complex	Suitable for enterprise systems
VNC / RDP	Excellent	Ready-to-use	Not ideal for educational demonstration

The comparison indicates that Java RMI provides a good balance between simplicity and performance. While socket programming offers better speed, it requires more effort to develop and maintain. RMI, on the other hand, is more convenient for academic or small-scale applications where simplicity and clarity are preferred.

8.7 Interpretation of Results

From a distributed systems perspective, the evaluation demonstrated that RMI can efficiently handle real-time multimedia data transfer in a controlled LAN environment. The findings support the concept that high-level middleware, such as RMI, can be used for multimedia broadcasting tasks when optimized for limited data size and controlled network conditions.

Although the system was not designed for high-definition or internet-scale streaming, it performed well for its intended purpose — educational and office-based screen broadcasting. The results also highlight that further optimization, such as adaptive frame rate control and data compression, could make the system more scalable and efficient.

8.8 Conclusion of Evaluation

The scientific evaluation confirmed that the Screen Broadcasting Using RMI system successfully met its objectives of providing reliable and real-time screen broadcasting within a local network. The use of Java RMI proved effective for implementing distributed multimedia communication with acceptable delay and stable performance.

While the system has limitations in scalability and security, it serves as a strong foundation for future research in distributed broadcasting technologies. The results show that Java RMI is a practical choice for lightweight, LAN-based screen broadcasting and can be extended for further academic and industrial applications.

References

1. Sun Microsystems, *Java Remote Method Invocation (RMI) Specification*, Oracle Corporation, 2017.
2. AT&T Laboratories, *The RFB Protocol – Virtual Network Computing (VNC)*, Technical Report, 2003.
3. Microsoft Corporation, *Remote Desktop Protocol (RDP) Technical Overview*, Microsoft Developer Network (MSDN), 2016.
4. TeamViewer GmbH, *TeamViewer – Remote Control and Desktop Sharing Software*, User Documentation, Version 12, 2017.
5. AnyDesk Software GmbH, *AnyDesk Remote Desktop Application*, Product Overview, 2018.
6. Tanenbaum, A. S., & Van Steen, M., *Distributed Systems: Principles and Paradigms*, 2nd Edition, Prentice Hall, 2007.
7. Deitel, H. M., & Deitel, P. J., *Java: How to Program*, 10th Edition, Pearson Education, 2015.
8. Oracle Documentation, *Java Platform Standard Edition 8 API Specification*, Oracle Corporation, 2018.
9. Liang, Y. D., *Introduction to Java Programming: Comprehensive Version*, 10th Edition, Pearson, 2015.
10. Stevens, W. R., *UNIX Network Programming*, Volume 1: The Sockets Networking API, 3rd Edition, Addison-Wesley, 2003.
11. Object Management Group (OMG), *Common Object Request Broker Architecture (CORBA) Specification*, Version 3.3, 2012.
12. Oracle Java Tutorials, *Trail: RMI – The Java Remote Method Invocation API*, Oracle Corporation, 2017.
13. Eckel, B., *Thinking in Java*, 4th Edition, Prentice Hall, 2006.
14. IEEE Computer Society, *IEEE Standards for LAN/MAN (IEEE 802.3)*, 2012.
15. W. Stallings, *Data and Computer Communications*, 10th Edition, Pearson, 2014.