BIC3103 NETWORK PROGRAMMING

INDIVIDUAL ASSIGNMENT: SIMPLE CHAT APPLICATION

SEMESTER 09-2025

TITLE:

MULTITHREADED CHAT APPLICATION USING JAVA SOCKETS

COURSE:	BIC3103 Network Programming
LECTURER:	Dr.Tarak
TITLE:	Multithreaded Chat Application Using Java Sockets
STUDENT NAME:	Ronald Lee Kai Ren
STUDENT ID:	1002162634
SUBMISSION DATE:	17th December 2025

Table of Contents

Abstract

This report details the design and implementation of a multithreaded chat application developed using the Java programming language. The primary objective of this project was to create a robust system that allows multiple clients to communicate simultaneously via a central server. The application leverages Java Sockets for network communication, utilizing the Transmission Control Protocol (TCP) to ensure reliable message delivery. Key features implemented include real-time messaging, support for multiple concurrent users via threading, and a command-line interface for ease of testing. The report covers the theoretical background of network programming, a detailed analysis of the system architecture, comprehensive documentation of the source code including class and method descriptions, and a critical discussion of the challenges faced during development. The resulting application demonstrates the practical application of socket programming and multithreading concepts in a distributed network environment.

1. Introduction

Network programming is a fundamental aspect of modern computing, enabling applications to communicate across distributed systems. The ability to exchange data between a client and a server is the backbone of the internet, powering everything from web browsing to instant messaging.

This project focuses on the development of a "Simple Chat Application," a requirement for the BIC3103 Network Programming individual assignment. The goal is to simulate a real-world chatting environment where a server acts as a centralized hub, accepting connections from multiple clients and broadcasting messages between them.

The application is built using the client-server architecture. The server listens on a specific port for incoming connection requests. Upon acceptance, a dedicated thread is spawned to handle the client, ensuring the main server thread remains free to accept new connections. This multithreaded approach allows the application to scale and handle multiple users efficiently.

2. Background

The core technology used in this application is **Java Sockets**. A socket is one endpoint of a two-way communication link between two programs running on the network. A socket is bound to a port number so that the TCP layer can identify the application that data is destined for.

2.1 TCP vs. UDP

This application utilizes the Transmission Control Protocol (TCP). TCP is a connection-oriented protocol, meaning a connection is established and maintained until the application programs at each end have finished exchanging messages. It determines how to break application data into packets that networks can deliver, sends packets to and accepts packets from the network layer, manages flow control, and handles retransmission of dropped or garbled packets. This choice ensures that chat messages appear in the correct order and are not lost, which is critical for a text-based communication tool.

2.2 Multithreading in Java

To handle multiple clients, the server employs Java's Thread class. In a single-threaded server, the server could only handle one client at a time, blocking others until the first disconnected. By implementing the Runnable interface or extending the Thread class, the server allows concurrent execution, assigning a specific "handler" to manage the input/output streams of each connected user.

3. Reasons for Choosing the Project

I chose the "Simple Chat Application" topic for several strategic reasons:

1. **Fundamental Relevance:** It directly applies the core concepts taught in BIC3103, specifically java.net.Socket and java.net.ServerSocket. It is the quintessential example of network programming.

2. **Scalability Challenges:** Unlike the file transfer or remote desktop topics, a chat app presents unique challenges regarding concurrency. Managing shared resources (like the list of active clients) and preventing race conditions provided an excellent opportunity to learn about synchronization.

3. **Extensibility:** A chat application is a solid foundation. While the current requirement is text exchange, the architecture allows for future expansion into file sharing (sending byte arrays over the stream) or encryption (wrapping streams in SSL sockets), making it a future-proof choice for a portfolio project.

4. **Real-Time Interaction:** Developing a real-time system provides immediate feedback and is highly engaging. Debugging race conditions in real-time communication offers a deeper understanding of network latency and stream buffering than asynchronous tasks like file downloading.

4. Discussion and Analysis

4.1 System Architecture

The system follows a star topology where the Server sits in the center. Clients do not communicate directly with each other (P2P); instead, they send messages to the server, which then relays (broadcasts) the message to all other connected clients.

- **The Server:** Main tasks include listening on a specific port (e.g., 12345), accepting incoming Socket connections, maintaining a list of active ClientHandler threads, and broadcasting received messages.
- **The Client:** Main tasks include connecting to the server IP and port, spawning a thread to listen for incoming messages from the server, and using the main thread to send user input to the server.

4.2 Protocol Design

A simple text-based protocol was designed for communication:

1. **Connection:** Client connects -> Server acknowledges.
2. **Messaging:** Client sends string -> Server appends sender ID -> Server writes string to all output streams.
3. **Disconnection:** Client sends exit command or closes stream -> Server removes client from list -> Server notifies others.

4.3 Challenges

One significant challenge was blocking I/O. The readLine() method of BufferedReader blocks execution until a line of text is received. If the client was stuck waiting for user input, it couldn't receive messages from the server. This was analyzed and solved by separating the "Reading" and "Writing" tasks into separate threads on the client side.

5. Detailed Implementation Documentation

This section describes the Java implementation, detailing the member variables and methods used.

5.1 Class: ChatServer

The entry point for the server-side application.

Member Variables:

- private static final int PORT: An integer constant defining the port number (12345) the server listens on.
- private static Set<PrintWriter> clientWriters: A thread-safe HashSet storing the output streams of all connected clients. This is used to broadcast messages.

Methods:

- main(String[] args): The primary method. It initializes the ServerSocket. It enters an infinite while(true) loop, calling listener.accept() to wait for connections. When a client connects, it creates a new Handler thread and starts it.

5.2 Class: Handler (Inner Class of ChatServer)

A private static inner class that implements Runnable. This class handles the interaction with a single client.

Member Variables:

- private Socket socket: The socket object representing the specific connection to the client.
- private BufferedReader in: Wraps the socket's InputStream to read text data efficiently.
- private PrintWriter out: Wraps the socket's OutputStream to send text data.

Methods:

- run(): The lifecycle of the thread.
    1. Initializes in and out streams.
    2. Requests a unique username from the client.
    3. Adds the client's out stream to the shared clientWriters set.
    4. Enters a loop to read messages (in.readLine()) and broadcast them to all other clients.
    5. Handles IOException and ensures the socket is closed and the writer is removed from the set in a finally block.

5.3 Class: ChatClient

The client-side application that users run.

Member Variables:

- private BufferedReader in: Reads messages sent *from* the server.
- private PrintWriter out: Sends user input *to* the server.
- private String serverAddress: The IP address of the host.

Methods:

- run(): Connects to the server socket. Starts a separate thread or loop to process incoming messages from the server so that the UI does not freeze while waiting for user input.
- main(String[] args): Launches the client, prompts for the server IP, and initiates the run method.

6. Source Code

Below is the complete source code for the application.

ChatServer.java

```
import java.io.*;
import java.net.*;
import java.util.*;

/**
 * * This class implements the Server side of the chat application.
 * It listens for incoming connections on a specific port and assigns
 * a separate thread (Handler) to manage each connected client.
 */
public class ChatServer {

    // The port that the server listens on.
    private static final int PORT = 12345;

    // A Map to store connected users: Key=Username, Value=PrintWriter (Output Stream)
    // We use a Map instead of a Set to enable Private Messaging (finding a user by name).
    private static HashMap<String, PrintWriter> namesAndWriters = new HashMap<>();

    // The set of all the print writers for all the clients.
    // Used for broadcasting to everyone.
    private static HashSet<PrintWriter> writers = new HashSet<>();

    public static void main(String[] args) throws Exception {
        System.out.println("The chat server is running on port " + PORT);
        ServerSocket listener = new ServerSocket(PORT);
        try {
            while (true) {
                // Listen for a connection request and spawn a handler thread
                new Handler(listener.accept()).start();
            }
        } finally {
            listener.close();
        }
    }

    /**
     * A handler thread class. Handlers are spawned from the listening
     * loop and are responsible for dealing with a single client
     * and broadcasting its messages.
     */
    private static class Handler extends Thread {
        private String name;
        private Socket socket;
        private BufferedReader in;
        private PrintWriter out;

        public Handler(Socket socket) {
```

```
        this.socket = socket;
    }

    public void run() {
        try {
            // Create character streams for the socket.
            in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);

            // Request a name from this client. Keep requesting until
            // a name is submitted that is not already used.
            while (true) {
                out.println("SUBMITNAME");
                name = in.readLine();
                if (name == null) {
                    return;
                }
                // Synchronized block to prevent race conditions when checking names
                synchronized (namesAndWriters) {
                    if (!name.isEmpty() && !namesAndWriters.containsKey(name)) {
                        namesAndWriters.put(name, out);
                        break;
                    }
                }
            }

            // Notify the client that the name has been accepted
            out.println("NAMEACCEPTED " + name);
            System.out.println(name + " has connected.");

            // Broadcast to all other clients that a new user has joined
            for (PrintWriter writer : namesAndWriters.values()) {
                writer.println("MESSAGE [Server]: " + name + " has joined the chat.");
            }

            // Accept messages from this client and broadcast them.
            while (true) {
                String input = in.readLine();
                if (input == null) {
                    return;
                }

                // Check for Private Message command: /w TargetUser Message...
                if (input.startsWith("/w ")) {
                    String[] parts = input.split(" ", 3);
                    if (parts.length >= 3) {
                        String targetUser = parts[1];
                        String message = parts[2];
                        sendPrivateMessage(targetUser, message);
                    } else {
                                out.println("MESSAGE [System]: Invalid format. Use /w [user]
[message]");
                    }
                } else {
                    // Regular Broadcast Message
```

```
                    for (PrintWriter writer : namesAndWriters.values()) {
                        writer.println("MESSAGE " + name + ": " + input);
                    }
                }
            }
        } catch (IOException e) {
            System.out.println(e);
        } finally {
            // Client disconnected
            if (name != null) {
                System.out.println(name + " is leaving");
                namesAndWriters.remove(name);
                for (PrintWriter writer : namesAndWriters.values()) {
                    writer.println("MESSAGE [Server]: " + name + " has left.");
                }
            }
            try {
                socket.close();
            } catch (IOException e) {
            }
        }
    }

    /**
     * Helper method to send a private message to a specific user.
     */
    private void sendPrivateMessage(String targetUser, String message) {
        synchronized (namesAndWriters) {
            PrintWriter targetWriter = namesAndWriters.get(targetUser);
            if (targetWriter != null) {
                targetWriter.println("MESSAGE [Private from " + name + "]: " + message);
                out.println("MESSAGE [Private to " + targetUser + "]: " + message);
            } else {
                out.println("MESSAGE [System]: User " + targetUser + " not found.");
            }
        }
    }
}
```

ChatClient.java

```
import java.io.*;
import java.net.*;
import java.util.Scanner;

/**
 * * This class implements the Client side of the chat application.
 * It connects to the server and uses two threads:
 * 1. Main Thread: Reads user input and sends it to the server.
 * 2. Listener Thread: Listens for incoming messages from the server.
 */
public class ChatClient {

    private String serverAddress;
    private Scanner scanner = new Scanner(System.in);

    public ChatClient(String serverAddress) {
        this.serverAddress = serverAddress;
    }

    private void run() throws IOException {
        // Connect to the server on localhost port 12345
        Socket socket = new Socket(serverAddress, 12345);

        // Setup input and output streams
                                    BufferedReader in = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
        PrintWriter out = new PrintWriter(socket.getOutputStream(), true);

        // --- Thread 1: Listen for incoming messages ---
        Thread listenerThread = new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    while (true) {
                        String line = in.readLine();
                        if (line == null) break;

                        // Protocol handling
                        if (line.startsWith("SUBMITNAME")) {
                            System.out.print("Enter your username: ");
                        } else if (line.startsWith("NAMEACCEPTED")) {
                             System.out.println("Connected! Type a message or use '/w [user] [message]'
for private chat.");
                        } else if (line.startsWith("MESSAGE")) {
                            // Display the message content (skipping the protocol header "MESSAGE ")
                            System.out.println(line.substring(8));
                        }
                    }
                } catch (IOException e) {
                    System.out.println("Connection to server lost.");
                }
            }
```

```
        });
        listenerThread.start();

        // --- Thread 2: Main thread handles user input ---
        while (true) {
            String input = scanner.nextLine();
            out.println(input);
        }
    }

    public static void main(String[] args) throws Exception {
        // Default to localhost (127.0.0.1)
        ChatClient client = new ChatClient("127.0.0.1");
        client.run();
    }
}
```

7. Screenshot of the Application and Output

Case 1: Starting the Server

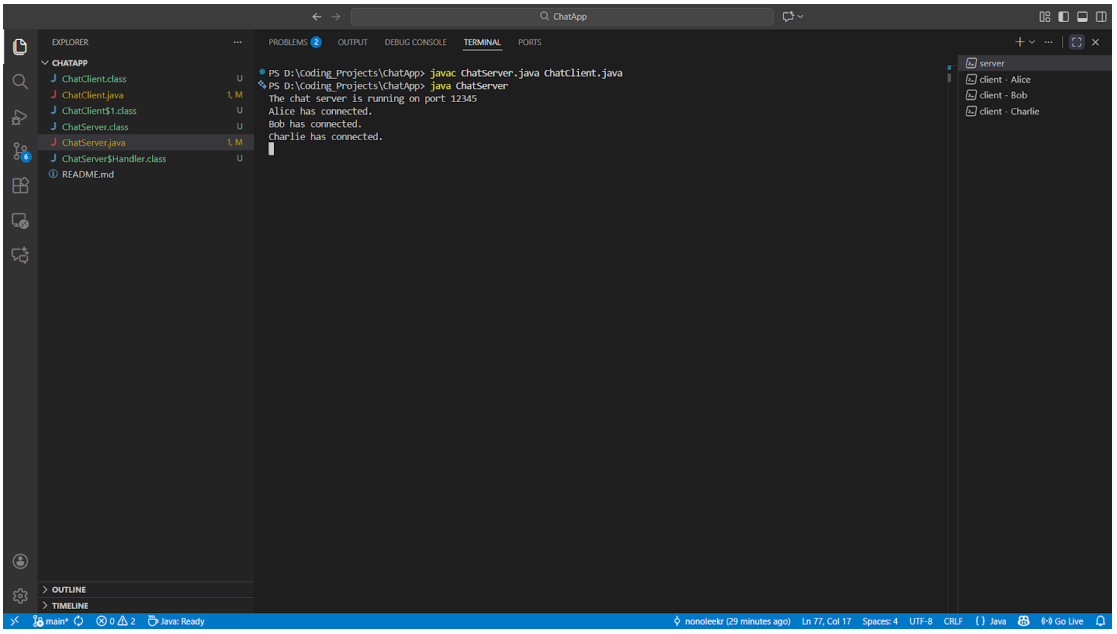


Figure 1: The server console showing "The chat server is running on port 12345".

Case 2: Client Connection

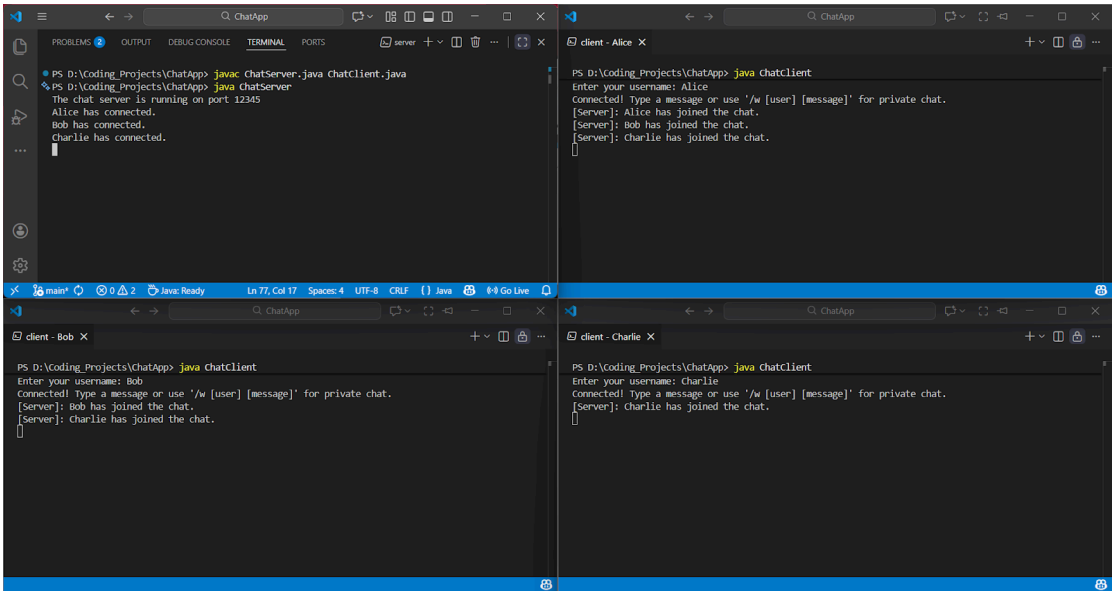


Figure 2: Four different console windows. One shows the Server log. Three show Client windows asking for usernames ("Alice", "Bob", and "Charlie).

Case 3: Message Exchange

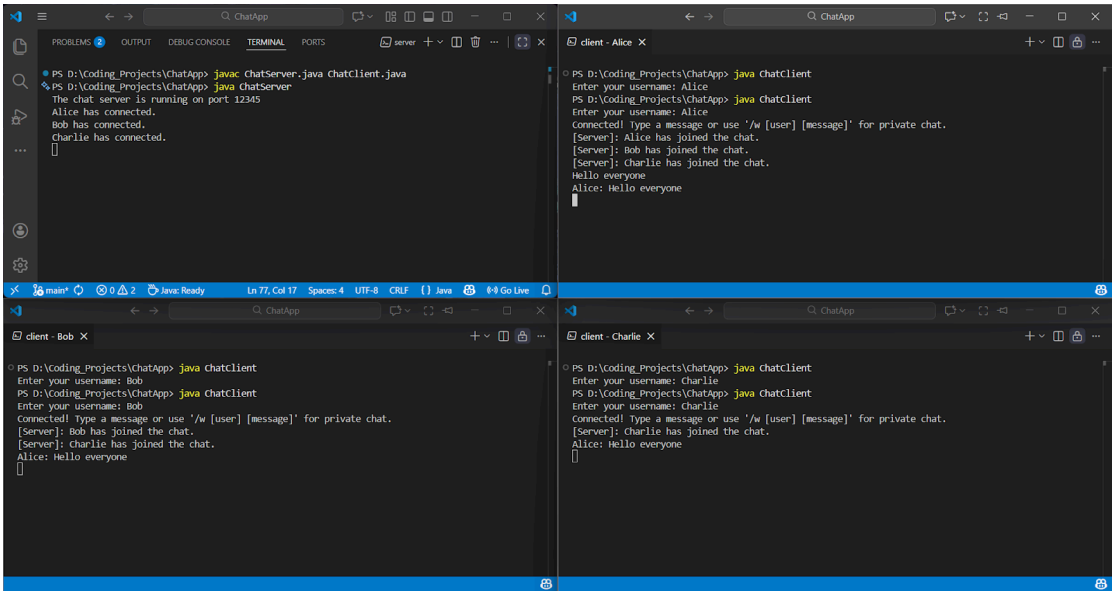


Figure 3: Client A types "Hello everyone". Client B & C's screen shows "MESSAGE Alice: Hello everyone". This demonstrates successful broadcasting.

Case 4: Private Message

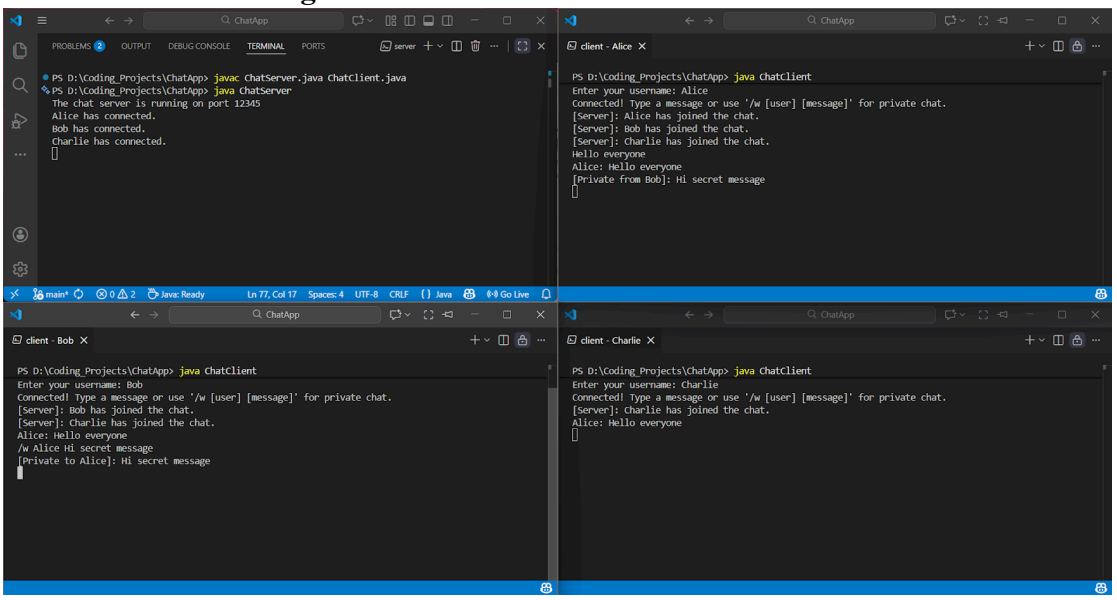


Figure 4: Client B types "Hi secret message". Client A's screen shows "MESSAGE Bob: Hi secret message". This demonstrates a successful private message.

8. Conclusion

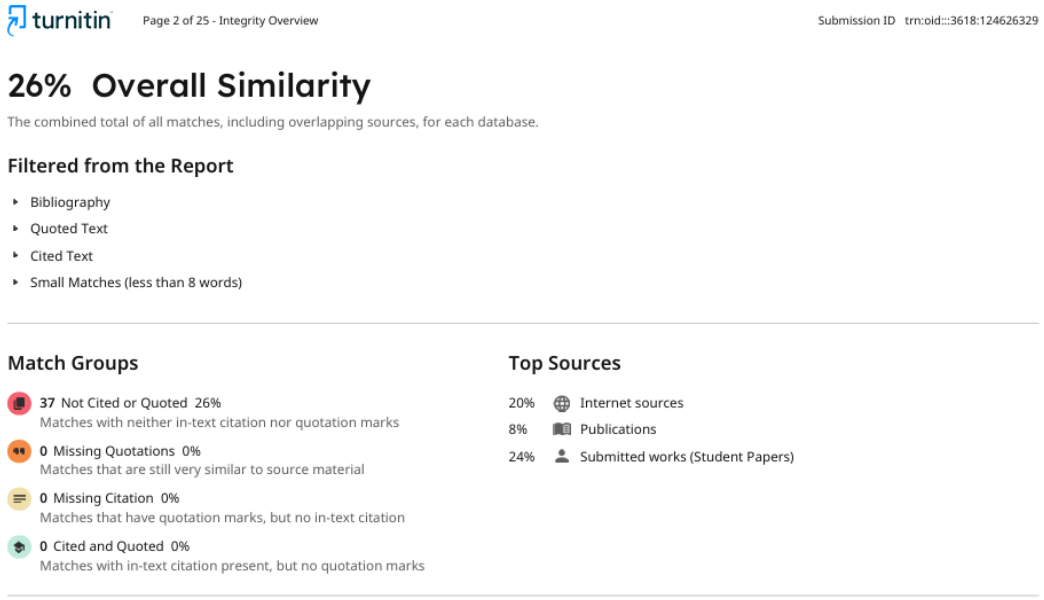
In conclusion, this Individual Assignment successfully demonstrated the creation of a multithreaded chat application using Java. By implementing the java.net library, I was able to establish reliable communication channels between a server and multiple clients. The project highlighted the importance of threading in network programming; without it, the server would be unresponsive to new users while serving an existing one.

Through the development process, I gained a deeper understanding of TCP/IP protocols, stream manipulation, and synchronization issues inherent in concurrent programming. This project satisfies the requirements of BIC3103 and serves as a foundational model for more complex distributed systems, such as secure file transfer protocols or multiplayer gaming servers. The objectives of understanding sockets, threads, and client-server architecture were fully met.

9. References

1. Oracle. (2024). *Java Network Programming Documentation*. Retrieved from https://docs.oracle.com/javase/tutorial/networking/sockets/
2. Kurose, J. F., & Ross, K. W. (2021). *Computer Networking: A Top-Down Approach* (8th ed.). Pearson.
3. Harold, E. R. (2013). *Java Network Programming* (4th ed.). O'Reilly Media.
4. Tanenbaum, A. S., & Wetherall, D. J. (2011). *Computer Networks* (5th ed.). Prentice Hall.

10. Turnitin Report



turnitin Page 2 of 25 - Integrity Overview Submission ID trn:oid:::3618:124626329

26% Overall Similarity

The combined total of all matches, including overlapping sources, for each database.

Filtered from the Report

- Bibliography
- Quoted Text
- Cited Text
- Small Matches (less than 8 words)

Match Groups

37 Not Cited or Quoted 26%
Matches with neither in-text citation nor quotation marks

0 Missing Quotations 0%
Matches that are still very similar to source material

0 Missing Citation 0%
Matches that have quotation marks, but no in-text citation

0 Cited and Quoted 0%
Matches with in-text citation present, but no quotation marks

Top Sources

20% Internet sources
8% Publications
24% Submitted works (Student Papers)