Experiment No: 06

Aim: Program to demonstrate token ring algorithm for distributed mutual exclusion

Theory:

- Concurrent access to a shared resource by several uncoordinated user-requests is serialized to secure the integrity of the shared resource.
- The actions performed by a user on a shared resource must be atomic.
- For correctness, it is necessary that the shared resource be accessed by a single site (or process) at a time (eg. Directory management)

"Mutual exclusion is a fundamental issue in the design of distributed systems and an efficient and robust technique for mutual exclusion is essential to the viable design of distributed systems."

Classification of Mutual Exclusion Algorithms:

1. Non-token based:
   - A site/process can enter a critical section when an assertion (condition) becomes true.
   - Algorithm should ensure that the assertion will be true in only one site/process.
2. Token based:
   - A unique token (a known, PRIVILEGE message) is shared among cooperating sites/processes.
   - Possessor of the token has access to critical section.
   - Need to take care of conditions such as loss of token, crash of token holder, possibility of multiple tokens, etc.

A token ring algorithm:

A logical ring is constructed in which each process is assigned a position in the ring. The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is.

Assumptions:

- all processes are equal
- each process has a unique "priority" number
- all processes know all unique priority numbers
- they don't know which has crashed
- a process that has failed is recovering knows that it has failed and takes appropriate action to rejoin the group

ADVANTAGES:

1. The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a CS.

2. Since the token circulates among processes in a well-defined order, starvation cannot occur.

DISADVANTAGES:

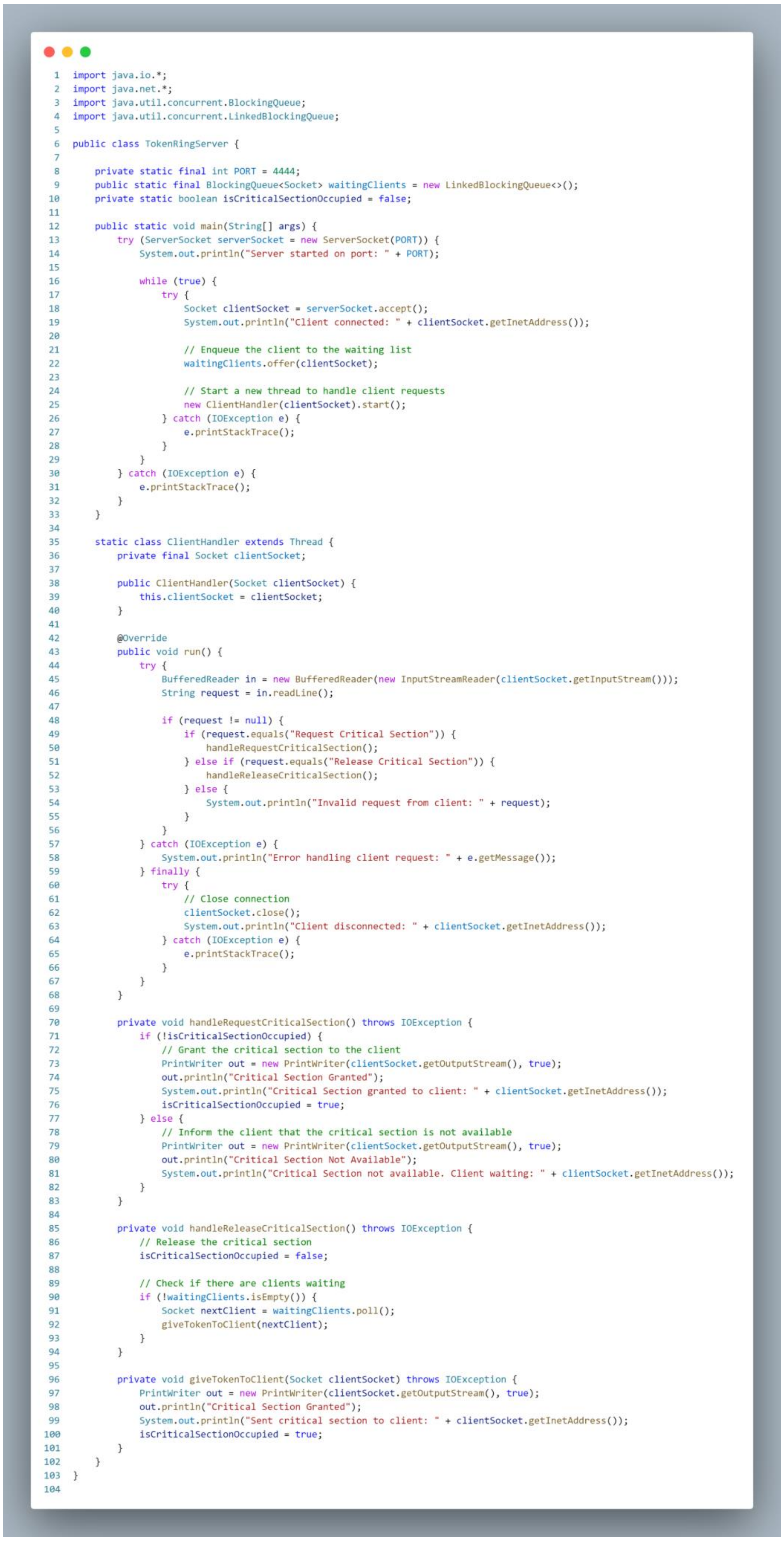
1. Once a process decides it wants to enter a CS, at worst it will have to wait for every other process to enter and leave one critical region.
2. If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is not a constant. The fact that the token has not been spotted for an hour does not mean that it has been lost; some process may still be using it.
3. The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbour tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can pass the token to the next member down the line.

Algorithm:

1. When the ring is initialized, process A (say) is given a token. The token circulates around the ring. It is passed from process k to process k+1 in point-to-point messages.
2. When a process acquires the token from its neighbour, it checks to see if it is attempting to enter a CS. If so, the process enters the CS, does all the work it needs to, and leaves the section.
3. After it has exited, it passes the token along the ring. It is not permitted to enter a second CS using the same token.
4. If a process is handed the token by its neighbour and is not interested in entering a CS, it just passes it along. As a consequence, when no processes want to enter any CS, the token just circulates at high speed around the ring.

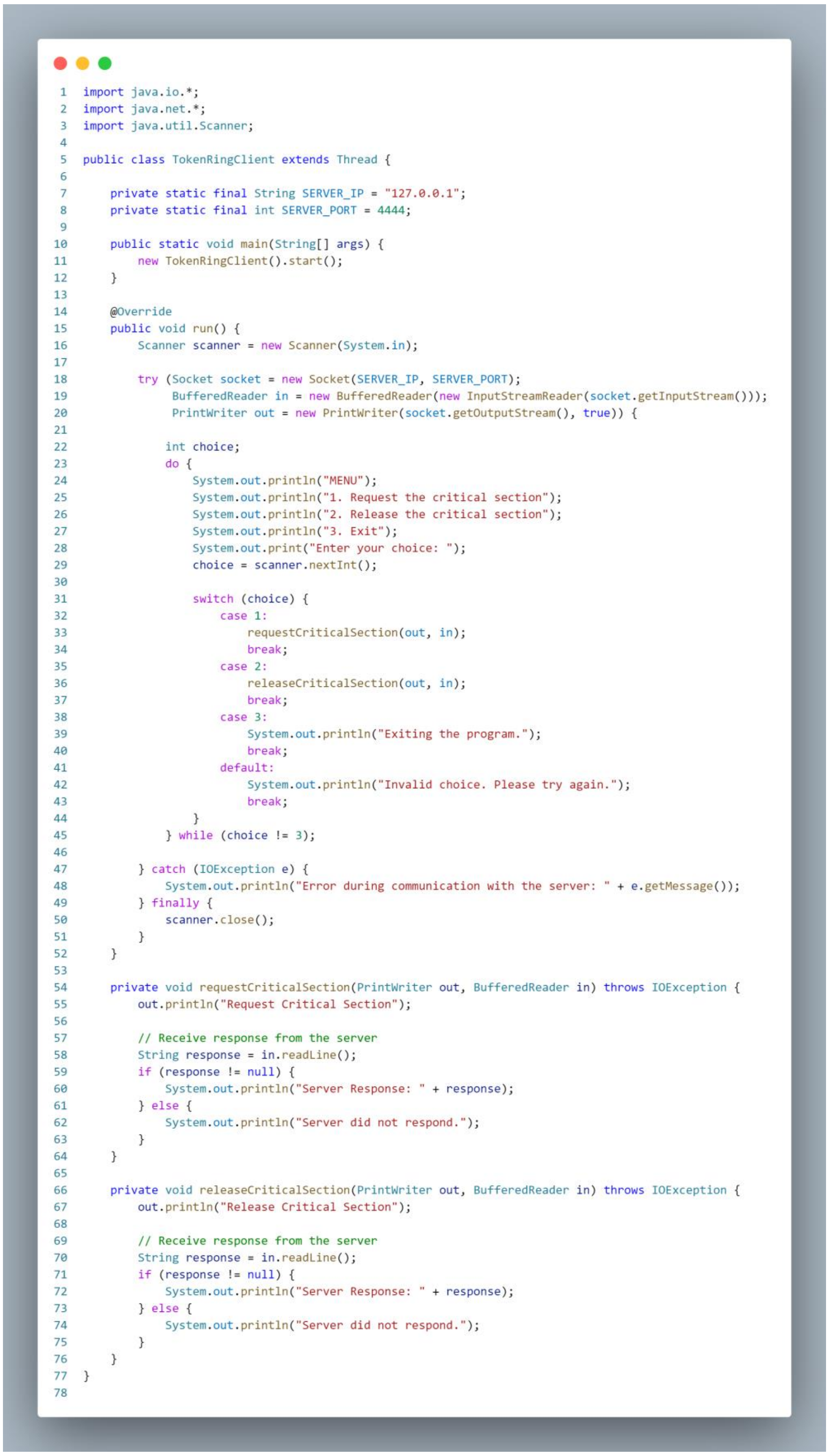
Code:

TokenRingServer.java



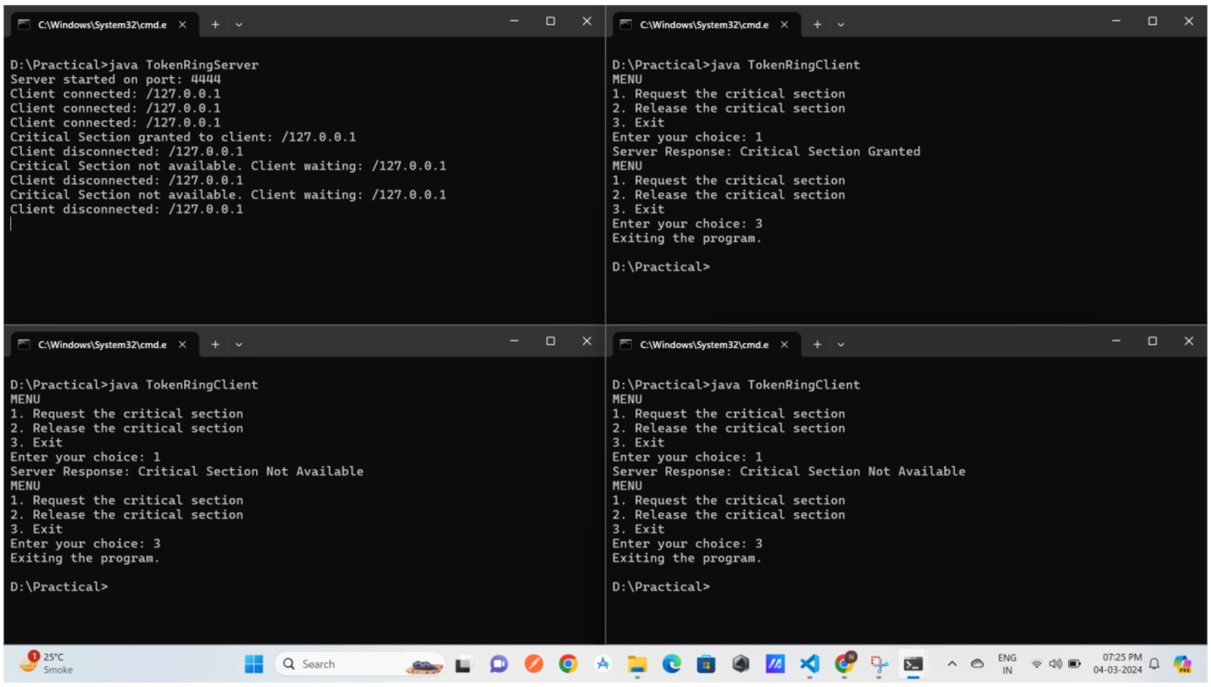
```java
import java.io.*;
import java.net.*;
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;

public class TokenRingServer {

    private static final int PORT = 4444;
    public static final BlockingQueue<Socket> waitingClients = new LinkedBlockingQueue<>();
    private static boolean isCriticalSectionOccupied = false;

    public static void main(String[] args) {
        try (ServerSocket serverSocket = new ServerSocket(PORT)) {
            System.out.println("Server started on port: " + PORT);

            while (true) {
                try {
                    Socket clientSocket = serverSocket.accept();
                    System.out.println("Client connected: " + clientSocket.getInetAddress());

                    // Enqueue the client to the waiting list
                    waitingClients.offer(clientSocket);

                    // Start a new thread to handle client requests
                    new ClientHandler(clientSocket).start();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    static class ClientHandler extends Thread {
        private final Socket clientSocket;

        public ClientHandler(Socket clientSocket) {
            this.clientSocket = clientSocket;
        }

        @Override
        public void run() {
            try {
                BufferedReader in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
                String request = in.readLine();

                if (request != null) {
                    if (request.equals("Request Critical Section")) {
                        handleRequestCriticalSection();
                    } else if (request.equals("Release Critical Section")) {
                        handleReleaseCriticalSection();
                    } else {
                        System.out.println("Invalid request from client: " + request);
                    }
                }
            } catch (IOException e) {
                System.out.println("Error handling client request: " + e.getMessage());
            } finally {
                try {
                    // Close connection
                    clientSocket.close();
                    System.out.println("Client disconnected: " + clientSocket.getInetAddress());
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        private void handleRequestCriticalSection() throws IOException {
            if (!isCriticalSectionOccupied) {
                // Grant the critical section to the client
                PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
                out.println("Critical Section Granted");
                System.out.println("Critical Section granted to client: " + clientSocket.getInetAddress());
                isCriticalSectionOccupied = true;
            } else {
                // Inform the client that the critical section is not available
                PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
                out.println("Critical Section Not Available");
                System.out.println("Critical Section not available. Client waiting: " + clientSocket.getInetAddress());
            }
        }

        private void handleReleaseCriticalSection() throws IOException {
            // Release the critical section
            isCriticalSectionOccupied = false;

            // Check if there are clients waiting
            if (!waitingClients.isEmpty()) {
                Socket nextClient = waitingClients.poll();
                giveTokenToClient(nextClient);
            }
        }

        private void giveTokenToClient(Socket clientSocket) throws IOException {
            PrintWriter out = new PrintWriter(clientSocket.getOutputStream(), true);
            out.println("Critical Section Granted");
            System.out.println("Sent critical section to client: " + clientSocket.getInetAddress());
            isCriticalSectionOccupied = true;
        }
    }
}
```

TokenRingClient.java



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

public class TokenRingClient extends Thread {

    private static final String SERVER_IP = "127.0.0.1";
    private static final int SERVER_PORT = 4444;

    public static void main(String[] args) {
        new TokenRingClient().start();
    }

    @Override
    public void run() {
        Scanner scanner = new Scanner(System.in);

        try (Socket socket = new Socket(SERVER_IP, SERVER_PORT);
             BufferedReader in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
             PrintWriter out = new PrintWriter(socket.getOutputStream(), true)) {

            int choice;
            do {
                System.out.println("MENU");
                System.out.println("1. Request the critical section");
                System.out.println("2. Release the critical section");
                System.out.println("3. Exit");
                System.out.print("Enter your choice: ");
                choice = scanner.nextInt();

                switch (choice) {
                    case 1:
                        requestCriticalSection(out, in);
                        break;
                    case 2:
                        releaseCriticalSection(out, in);
                        break;
                    case 3:
                        System.out.println("Exiting the program.");
                        break;
                    default:
                        System.out.println("Invalid choice. Please try again.");
                        break;
                }
            } while (choice != 3);

        } catch (IOException e) {
            System.out.println("Error during communication with the server: " + e.getMessage());
        } finally {
            scanner.close();
        }
    }

    private void requestCriticalSection(PrintWriter out, BufferedReader in) throws IOException {
        out.println("Request Critical Section");

        // Receive response from the server
        String response = in.readLine();
        if (response != null) {
            System.out.println("Server Response: " + response);
        } else {
            System.out.println("Server did not respond.");
        }
    }

    private void releaseCriticalSection(PrintWriter out, BufferedReader in) throws IOException {
        out.println("Release Critical Section");

        // Receive response from the server
        String response = in.readLine();
        if (response != null) {
            System.out.println("Server Response: " + response);
        } else {
            System.out.println("Server did not respond.");
        }
    }
}
```

Output:



```
D:\Practical>java TokenRingServer
Server started on port: 4444
Client connected: /127.0.0.1
Client connected: /127.0.0.1
Client connected: /127.0.0.1
Critical Section granted to client: /127.0.0.1
Client disconnected: /127.0.0.1
Critical Section not available. Client waiting: /127.0.0.1
Client disconnected: /127.0.0.1
Critical Section not available. Client waiting: /127.0.0.1
Client disconnected: /127.0.0.1
```

```
D:\Practical>java TokenRingClient
MENU
1. Request the critical section
2. Release the critical section
3. Exit
Enter your choice: 1
Server Response: Critical Section Granted
MENU
1. Request the critical section
2. Release the critical section
3. Exit
Enter your choice: 3
Exiting the program.

D:\Practical>
```

```
D:\Practical>java TokenRingClient
MENU
1. Request the critical section
2. Release the critical section
3. Exit
Enter your choice: 1
Server Response: Critical Section Not Available
MENU
1. Request the critical section
2. Release the critical section
3. Exit
Enter your choice: 3
Exiting the program.

D:\Practical>
```

```
D:\Practical>java TokenRingClient
MENU
1. Request the critical section
2. Release the critical section
3. Exit
Enter your choice: 1
Server Response: Critical Section Not Available
MENU
1. Request the critical section
2. Release the critical section
3. Exit
Enter your choice: 3
Exiting the program.

D:\Practical>
```

Conclusion:

The algorithm does not allow the circulation of the token along the ring, when there is no need (i.e. when no process wants to enter in its critical section).