



**School of Computer Science and Engineering**

**CSE 2005**

# Multi-threading using client server interaction

Faculty: Prof. Sendhil Kumar K.S.

Slot: A1

NAME	REGISTRATION NUMBER
Daksh Bardia	16BCE0783
Aditya Firoda	16BCE2184
Aryan Soni	16BCI0198
Abhishek Dube	16BCI0196

## CERTIFICATE

This is to certify that the project work entitled “Multi-threading using client-server interaction” that is being submitted by “Daksh, Aryan, Aditya and Abhishek” for Operating Systems

(CSE2005) is a record of bonafide work done under my supervision. The contents of this Project work, in full or in parts, have neither been taken from any other source nor have been submitted for any other CAL course.

Place: Vellore

Date: 2<sup>nd</sup> May, 2016

**Signature of Faculty:**

**Signature of Students:**

16BCE0783 Daksh –

16BCI0198 Aryan –

16BCI0196 Abhishek –

16BCE2074 Aditya-

## ACKNOWLEDGEMENT

We would like to thank VIT University for providing us an opportunity to carry out this research project. We also thank to our batch mates and seniors in helping us to carry out our project work.

We thank Prof. SENDHIL KUMAR K.S. to be our project guide and guided us at various stages in completing the project. Without his support it would be very difficult for us to do the project. We would like to pay our gratitude to Sir for sharing his pearls of wisdom with us during the project.

## INTRODUCTION

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but in most cases a thread is a component of a process. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time.

Systems with a single processor generally implement multithreading by time slicing: the central processing unit (CPU) switches between different software threads. This context switching generally happens very often and rapidly enough that users perceive the threads or tasks as running in parallel. On a multiprocessor or multi-core system, multiple threads can execute in parallel, with every processor or core executing a separate thread simultaneously; on a processor or core with hardware threads, separate software threads can also be executed concurrently by separate hardware threads.

Threads made an early appearance in OS/360 Multiprogramming with a Variable Number of Tasks (MVT) in 1967, in which context they were called "tasks". The term "thread" has been attributed to Victor A. Vyssotsky. Process schedulers of many modern operating systems directly support both time-sliced and multiprocessor threading, and the operating system kernel allows programmers to manipulate threads by exposing required functionality through the system call interface. Some threading implementations are called kernel threads, whereas light-weight processes (LWP) are a specific type of kernel thread that share the same state and information. Furthermore, programs can have user-space threads when threading with timers, signals, or other methods to interrupt their own execution, performing a sort of ad hoc time slicing.

### Single Threading:

In computer programming, single-threading is the processing of one command at a time. The opposite of single-threading is multithreading. While it has been suggested that the term single-threading is misleading, the term has been widely accepted within the functional programming community.

### Multi-Threading:

Multithreading is mainly found in multitasking operating systems. Multithreading is a widespread programming and execution model that allows multiple threads to exist within the context of one process. These threads share the process's resources, but are able to execute independently. The threaded programming model provides developers with a useful abstraction of concurrent execution. Multithreading can also be applied to one process to enable parallel execution on a multiprocessing system.

Multithreaded applications have the following advantages:

- **Responsiveness:** multithreading can allow an application to remain responsive to input. In a one-thread program, if the main execution thread blocks on a long-running task, the entire application can appear to freeze. By moving such long-running tasks to a worker

thread that runs concurrently with the main execution thread, it is possible for the application to remain responsive to user input while executing tasks in the background. On the other hand, in most cases multithreading is not the only way to keep a program responsive, with [non-blocking I/O](#) and/or [Unix signals](#) being available for gaining similar results.<sup>[6]</sup>

- **Faster execution:** this advantage of a multithreaded program allows it to operate faster on [computer systems](#) that have multiple [central processing units](#) (CPUs) or one or more [multi-core processors](#), or across a [cluster](#) of machines, because the threads of the program naturally lend themselves to parallel execution, assuming sufficient independence (that they do not need to wait for each other).
- **Lower resource consumption:** using threads, an application can serve multiple clients concurrently using fewer resources than it would need when using multiple process copies of itself. For example, the [Apache HTTP server](#) uses [thread pools](#) a pool of listener threads for listening to incoming requests, and a pool of server threads for processing those requests.
- **Better system utilization:** as an example, a file system using multiple threads can achieve higher throughput and lower latency since data in a faster medium (such as cache memory) can be retrieved by one thread while another thread retrieves data from a slower medium (such as external storage) with neither thread waiting for the other to finish.
- **Simplified sharing and communication:** unlike processes, which require a [message passing](#) or shared memory mechanism to perform [inter-process communication](#) (IPC), threads can communicate through data, code and files they already share.
- **Parallelization:** applications looking to use multicore or multi-CPU systems can use multithreading to split data and tasks into parallel subtasks and let the underlying architecture manage how the threads run, either concurrently on one core or in parallel on multiple cores. GPU computing environments like [CUDA](#) and [OpenCL](#) use the multithreading model where dozens to hundreds of threads run in [parallel across data](#) on a [large number of cores](#).

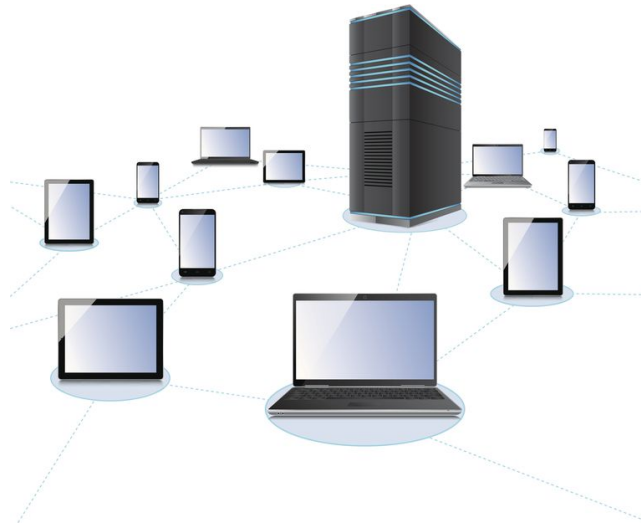
Multithreading has the following drawbacks:

- **[Synchronization](#):** since threads share the same address space, the [programmer](#) must be careful to avoid [race conditions](#) and other non-intuitive behaviors. In order for data to be correctly manipulated, threads will often need to [rendezvous](#) in time in order to process the data in the correct order. Threads may also require [mutually exclusive](#) operations (often implemented using [semaphores](#)) in order to prevent common data from being simultaneously modified or read while in the process of being modified. Careless use of such primitives can lead to [deadlocks](#).
- **Thread crashes a process:** an illegal operation performed by a thread crashes the entire process; therefore, one misbehaving thread can disrupt the processing of all the other threads in the application.

## **Client – Server Model:**

A client/server application is a piece of software that runs on a client computer and makes requests to a remote server. Many such applications are written in high-level visual programming languages where UI, forms, and most business logic reside in the client application.

Often such applications are database applications that make database queries to a remote central database server (this can, however, get much more complicated than that and involve other communication methods)

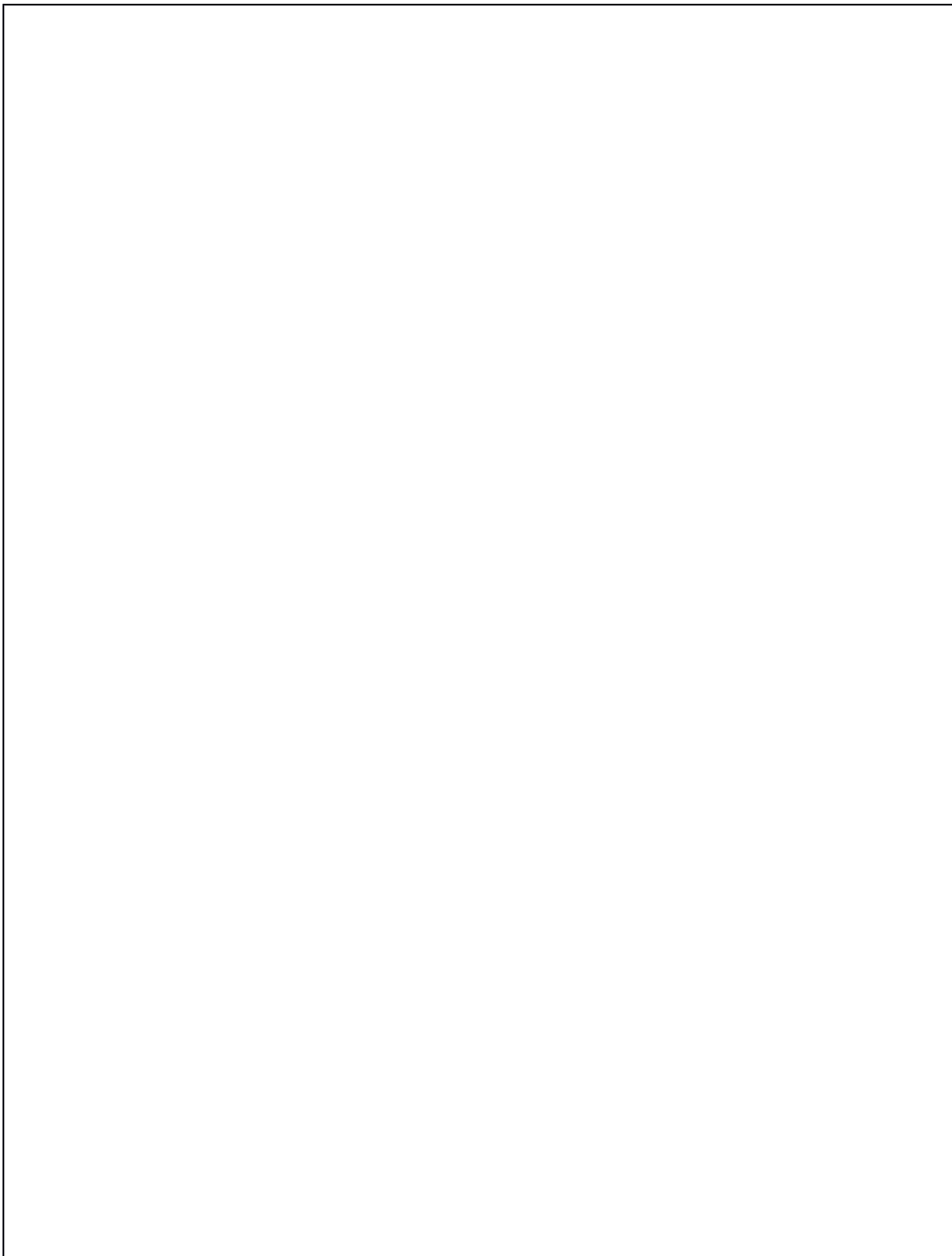


## **Client – Server Hardware -**

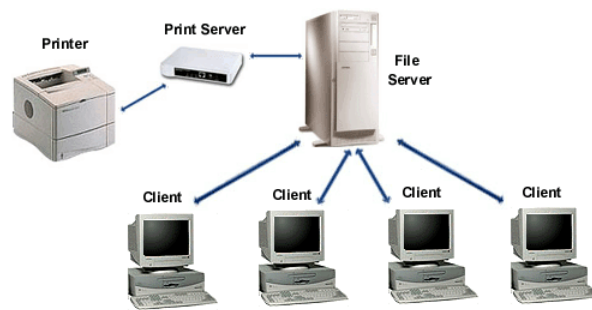
Client/server networking grew in popularity many years ago as personal computers (PCs) became the common alternative to older mainframe computers.

Client devices are typically PCs with network software applications installed that request and receive information over the network. Mobile devices, as well as desktop computers, can both function as clients.

A server device typically stores files and databases including more complex applications like Web sites. Server devices often feature higher-powered central processors, more memory, and larger disk drives than clients.



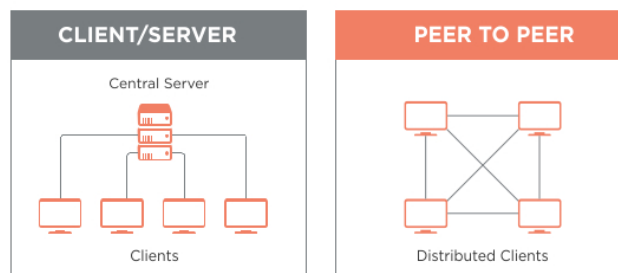
**Local Client-Server Networks:** Many home networks utilize client-server systems on a small scale. Broadband routers, for example, contain DHCP servers that provide IP addresses to the home computers (DHCP clients). Other types of network servers found in home include print servers and backup servers.



## **Client-Server vs. Peer-to-Peer and Other Models:**

The client-server model of networking was originally developed to share access to database applications among larger numbers of users. Compared to the mainframe model, client-server networking gives better flexibility as connections can be made on-demand as needed rather than being fixed. The client-server model also supports modular applications that can make the job of creating software easier. In so-called two tier and three tier types of client-server systems, software applications are separated into modular components, and each component is installed on clients or servers specialized for that subsystem.

Client-server is just one approach to managing network applications. The primary alternative to client-server, peer-to-peer networking, treats all devices as having equivalent capability rather than specialized client or server roles. Compared to client-server, peer to peer networks offer some advantages such as better flexibility in expanding the network to handle a large number of clients. Client-server networks generally offer advantages over peer-to-peer as well, such as the ability to manage applications and data in one centralized location.





## **Applications -**

- A client / server application can be cross platform if it is written in a cross platform language, or it can be platform specific. In the case of a cross platform language there is an advantage that the application can potentially provide a user interface that is native in appearance to the OS or platform environment it is running under.
- In a database application, data related number crunching can occur on the remote database server where the processing is close to physical data. An example of a database query might be to return the sum of a field named "dollar amount" where the field name year is "2001". There may be hundreds of thousands of records but the client computer does not have to worry about fetching or sorting through all of them itself. The database server will sort through that and just return one small record with the result to the client.
- A client / server application can be cross platform if it is written in a cross platform language, or it can be platform specific. In the case of a cross platform language there is an advantage that the application can potentially provide a user interface that is native in appearance to the OS or platform environment it is running under.
- Client / server applications, either run locally on a client computer or through something like Terminal Server, Citrix, or VNC, can work closely with the underlying OS to provide a rich, powerful, robust, easy to use interface.
- By running locally on a client computer applications can also work with other local hardware such as scanners, bar code readers, modems, removable media, accelerated video for multimedia, or 3d video acceleration.

## **Libraries Used:**

- **Java.io.BufferedReader** - The Java.io.BufferedReader class reads text from a character-input stream, buffering characters so as to provide for the efficient reading of characters, arrays, and lines.
- **Java.io.IOException** - In general, I/O means Input or Output. Those methods throw the IO Exception whenever an input or output operation is failed or interpreted. Note that this won't be thrown for reading or writing to memory as Java will be handling it automatically using this Class.
- **Java.io.InputStreamReader** - An InputStreamReader is a bridge from byte streams to character streams. It reads bytes and decodes them into characters using a specified charset.
- **Java.io.PrintWriter** - The Java PrintWriter class ( java.io.PrintWriter ) enables you to write formatted data to an underlying Writer . For

instance, writing int , long and other primitive data formatted as text, rather than as their byte values.

- **Java.net.Socket** - Class java.net.Socket. java.lang.Object  
java.net.Socket public class Socket extends Object. This class implements client sockets (also called just "sockets"). A socket is an endpoint for communication between two machines. The actual work of the socket is performed by an instance of the SocketImpl class.
- **Java.net.UnknownHostException** - Java.net.UnknownHostException:  
Host is unresolved: Thrown to indicate that the IP address of a host could not be determined. This exception is also raised when you are connected to a valid wifi but router does not receive the internet.
- **Java.net.ConnectException** - This exception usually occurs when there is no service listening on the IP/port you are trying to connect to. ...  
Your server is not listening for connections. Your server has too many pending connections waiting to be accepted. A firewall is blocking your connection before it reaches your server.

-

## **CLIENT CODE:**

```
import java.util.*;
import java.io.*;
import java.util.concurrent.atomic.AtomicLong;
public class Client {
    private static String hostName;
    private static Thread thrd = null;
    private static LinkedList<Thread> list = new LinkedList<Thread>();
    private static AtomicLong totalTime = new AtomicLong(0);
```

```

private static AtomicLong runningThreads = new AtomicLong(0);
private static boolean printOutput = true;

public static void main(String[] args) {
    int menuSelection = 0;
    int numProcesses = 1;
    if (args.length == 0) {
        System.out.println("User did not enter a host name. Client program
exiting.");
        System.exit(1);
    }
    else while (menuSelection != 6) {
        menuSelection = mainMenu();
        if (menuSelection == 6) {
            System.out.println("Quitting.");
            System.exit(0);
        }
        if (menuSelection == 5) {
            printOutput = false;
            menuSelection = benchmarkMenu();
            numProcesses = numProcessesMenu();
        }

        totalTime.set(0);
        runningThreads.set(numProcesses);
        for (int i = 0; i < numProcesses; i++) {

            thrd = new Thread(new ClientThread(args[0], menuSelection,
totalTime, printOutput, runningThreads));
            thrd.start();
            list.add(thrd);

        }

        for (int i = 0; i < numProcesses; i++) {
            try {

                list.get(i).join();
            } catch (InterruptedException e) {

                e.printStackTrace();
            }
        }
    }
}

```

```

    }

    while (runningThreads.get() != 0) {

        System.out.println("Average response time: " + (totalTime.get() /
numProcesses) + " ms\n");
        numProcesses = 1;
        printOutput = true;
    }

}

public static int mainMenu() {
    int menuSelection = 0;
    while ((menuSelection <= 0) || (menuSelection > 6)) {
        System.out.println("The menu provides the following choices to the user:
");
        System.out.println("1. Host current Date and Time \n"
            + "2. Host Netstat \n3. Host current users "
            + "\n4. Host running processes \n5. Benchmark (measure
mean response time)\n6. Quit ");
        System.out.print("Please provide number corresponding to the action you
want to be performed: ");
        Scanner sc = new Scanner(System.in);
        if (sc.hasNextInt()) menuSelection = sc.nextInt();
    }
    return menuSelection;
}

public static int benchmarkMenu() {
    int menuSelection = 0;
    while ((menuSelection <= 0) || (menuSelection > 4)) {
        System.out.println("Which command would you like to benchmark? ");
        System.out.println("1. Host current Date and Time \n"
            + "2. Host Netstat \n3. Host current users "
            + "\n4. Host running processes");
        System.out.print("Please provide number corresponding to the action you
want to be performed: ");
        Scanner sc = new Scanner(System.in);
        if (sc.hasNextInt()) menuSelection = sc.nextInt();
    }
    return menuSelection;
}

```

```

    }

    public static int numProcessesMenu() {
        int menuSelection = 0;
        while ((menuSelection <= 0) || (menuSelection > 100)) {
            System.out.print("How many connections to the server would you like to
open? [1-100]: ");
            Scanner sc = new Scanner(System.in);
            if (sc.hasNextInt()) menuSelection = sc.nextInt();
        }
        return menuSelection;
    }
}

```

## **CLIENT THREAD CODE:**

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.net.UnknownHostException;
import java.net.ConnectException;
import java.util.concurrent.atomic.AtomicLong;

public class ClientThread extends Thread {
    int menuSelection;
    String hostName;
    Socket socket = null;
    AtomicLong totalTime;
    AtomicLong runningThreads;
    boolean printOutput;
    long startTime;
    long endTime;

    ClientThread(String hostName, int menuSelection, AtomicLong totalTime, boolean
printOutput, AtomicLong runningThreads) {
        this.menuSelection = menuSelection;
        this.hostName = hostName;
        this.totalTime = totalTime;
        this.printOutput = printOutput;
    }
}

```

```

        this.runningThreads = runningThreads;
    }

    public void run() {
        PrintWriter out = null;
        BufferedReader input = null;
        try {

            socket = new Socket(hostName, 15432);
            if (printOutput) {
                System.out.print("Establishing connection.");
            }
            out = new PrintWriter(socket.getOutputStream(), true);
            input = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
            if (printOutput) System.out.println("\nRequesting output for the " +
menuSelection + " command from " + hostName);
            startTime = System.currentTimeMillis();
            out.println(Integer.toString(menuSelection));
            if (printOutput) System.out.println("Sent output");
            String outputString;
            while (((outputString = input.readLine()) != null) && (!
outputString.equals("END_MESSAGE"))) {
                if (printOutput) System.out.println(outputString);
            }

            endTime = System.currentTimeMillis();

            totalTime.addAndGet(endTime - startTime);

        }
        catch (UnknownHostException e) {
            System.err.println("Unknown host: " + hostName);
            System.exit(1);
        }
        catch (ConnectException e) {
            System.err.println("Connection refused by host: " + hostName);
            System.exit(1);
        }
        catch (IOException e) {
            e.printStackTrace();
        }
        finally {
            if (printOutput) System.out.println("closing");

```

```

        try {
            socket.close();
            runningThreads.decrementAndGet();
            System.out.flush();
        }
        catch (IOException e ) {
            System.out.println("Couldn't close socket");
        }
    }
}
}
}

```

## **COMMAND EXECUTOR CODE:**

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
public class CommandExecutor {

    static String run(String commandString) {
        String result = "";
        String line;
        try {
            Process child =
Runtime.getRuntime().exec(parseCommand(commandString));
            BufferedReader output = new BufferedReader(new
InputStreamReader(child.getInputStream()));
            while ((line = output.readLine()) != null) {
                result = result.concat(line);
                result = result.concat("\n");
            }

            result = result.concat("\n");
            result = result.concat("END_MESSAGE");
            output.close();

        } catch (IOException e) {
            e.printStackTrace();
        }

        return result;
    }
}

```



```

static String parseCommand(String inputString) {
    int inputInt = Integer.parseInt(inputString);
    String commandString = "";
    switch (inputInt) {
        case 1:
            commandString = "date";
            break;
        case 2:
            commandString = "netstat";
            break;

        case 3:
            commandString = "who";
            break;
        case 4:
            commandString = "ps -e";
            break;
    }

    return commandString;
}
}

```

## **SERVER CODE OUTPUT:**

```

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.concurrent.atomic.AtomicInteger;

public class Server {

    public static void main(String[] args) {
        AtomicInteger numThreads = new AtomicInteger(0);

        ArrayList<Thread> list = new ArrayList<Thread>();

        try {
            ServerSocket socket = new ServerSocket(15432);
            System.out.println("Server listening on port 15432");
            while(true) {
                Socket client = socket.accept();
                Thread thrd = new Thread(new ServerThread(client));
            }
        }
    }
}

```

```

        list.add(thrd);
        thrd.start();
        numThreads.incrementAndGet();
        System.out.println("Thread " + numThreads.get() + " started.");
    }
}
catch (IOException ioe){
    ioe.printStackTrace();
}
}
}

```

## **SERVER-THREAD CODE:**

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.Socket;
import java.util.concurrent.atomic.AtomicInteger;

public class ServerThread extends Thread {
    Socket client = null;
    PrintWriter output;
    BufferedReader input;

    public ServerThread(Socket client) {
        this.client = client;
    }

    public void run() {
        System.out.print("Accepted connection. ");

        try {

            output = new PrintWriter(client.getOutputStream(), true);
            input = new BufferedReader(new
InputStreamReader(client.getInputStream()));
            System.out.print("Reader and writer created. ");

            String inString;
            // read the command from the client

```

```

while ((inString = input.readLine()) != null);
    System.out.println("Read command " + inString);

    // run the command using CommandExecutor and get its output
    String outString = CommandExecutor.run(inString);
    System.out.println("Server sending result to client");
    // send the result of the command to the client
    output.println(outString);
}
catch (IOException e) {
    e.printStackTrace();
}
finally {

    try {
        client.close();
    }
    catch (IOException e) {
        e.printStackTrace();
    }
    System.out.println("Output closed.");
}

}
}

```

## **Output:**

In Case of option 1 -

```
MINGW32/c/Users/Aryan Soni/Pictures/Simple-Java-Client-Server-master/client
Aryan_Soni@ARYAN MINGW32 /c/Users/Aryan Soni/Pictures/Simple-Java-Client-Server-master/client
$ javac Client.java
Aryan_Soni@ARYAN MINGW32 /c/Users/Aryan Soni/Pictures/Simple-Java-Client-Server-master/client
$ java Client ARYAN
The menu provides the following choices to the user:
1. Host current Date and Time
2. Host Netstat
3. Host current users
4. Host running processes
5. Benchmark (measure mean response time)
6. Quit
Please provide number corresponding to the action you want to be performed: 1
Establishing connection.
Requesting output for the '1' command from ARYAN
Sent output
Thu, Nov 2, 2017 6:33:40 PM

closing
Average response time: 237 ms

The menu provides the following choices to the user:
1. Host current Date and Time
2. Host Netstat
3. Host current users
4. Host running processes
5. Benchmark (measure mean response time)
6. Quit
Please provide number corresponding to the action you want to be performed: ]

MINGW32/c/Users/Aryan Soni/Pictures/Simple-Java-Client-Server-master/server
Aryan_Soni@ARYAN MINGW32 /c/Users/Aryan Soni/Pictures/Simple-Java-Client-Server-master/server
$ java Server
Server listening on port 15432
Thread 1 started.
Accepted connection. Reader and writer created. Read command 1
Server sending result to client
Output closed.
```

## In Case of option 2 -

```
MINGW32/c/Users/Aryan Soni/Pictures/Simple-Java-Client-Server-master/client
Sent output
Active Connections

Proto Local Address Foreign Address State
TCP 127.0.0.1:3306 ARYAN:50027 ESTABLISHED
TCP 127.0.0.1:3306 ARYAN:50028 ESTABLISHED
TCP 127.0.0.1:49669 ARYAN:49669 ESTABLISHED
TCP 127.0.0.1:49670 ARYAN:49669 ESTABLISHED
TCP 127.0.0.1:50027 ARYAN:3306 ESTABLISHED
TCP 127.0.0.1:50028 ARYAN:3306 ESTABLISHED
TCP 172.16.42.84:15432 ARYAN:50367 ESTABLISHED
TCP 172.16.42.84:50252 122.15.34.52:http CLOSE_WAIT
TCP 172.16.42.84:50253 172.16.1.1:http CLOSE_WAIT
TCP 172.16.42.84:50254 122.15.34.49:http CLOSE_WAIT
TCP 172.16.42.84:50335 maa03s22-in-f14:https TIME_WAIT
TCP 172.16.42.84:50338 ARYAN:15432 TIME_WAIT
TCP 172.16.42.84:50339 prg34-010:http TIME_WAIT
TCP 172.16.42.84:50341 204.79.197.229:https TIME_WAIT
TCP 172.16.42.84:50345 204.79.197.229:https TIME_WAIT
TCP 172.16.42.84:50346 204.79.197.229:https TIME_WAIT
TCP 172.16.42.84:50347 204.79.197.229:https TIME_WAIT
TCP 172.16.42.84:50348 204.79.197.229:https TIME_WAIT
TCP 172.16.42.84:50349 204.79.197.229:https TIME_WAIT
TCP 172.16.42.84:50350 204.79.197.229:https TIME_WAIT
TCP 172.16.42.84:50351 204.79.197.229:https TIME_WAIT
TCP 172.16.42.84:50352 204.79.197.229:https TIME_WAIT
TCP 172.16.42.84:50353 111.221.29.254:https TIME_WAIT
TCP 172.16.42.84:50354 111.221.29.254:https TIME_WAIT
TCP 172.16.42.84:50355 204.79.197.229:https TIME_WAIT
TCP 172.16.42.84:50356 111.221.29.254:https TIME_WAIT
TCP 172.16.42.84:50359 111.221.29.254:https TIME_WAIT
TCP 172.16.42.84:50360 r-252-58-45-5:http TIME_WAIT
TCP 172.16.42.84:50361 fra02-003:http TIME_WAIT
TCP 172.16.42.84:50362 1oft9316:http CLOSE_WAIT
TCP 172.16.42.84:50367 ARYAN:15432 ESTABLISHED

closing
Average response time: 78036 ms

The menu provides the following choices to the user:
1. Host current Date and Time
2. Host Netstat
3. Host current users
4. Host running processes
5. Benchmark (measure mean response time)
6. Quit
Please provide number corresponding to the action you want to be performed: ]

MINGW32/c/Users/Aryan Soni/Pictures/Simple-Java-Client-Server-master/server
Aryan_Soni@ARYAN MINGW32 /c/Users/Aryan Soni/Pictures/Simple-Java-Client-Server-master/server
$ java Server
Server listening on port 15432
Thread 1 started.
Accepted connection. Reader and writer created. Read command 1
Server sending result to client
Output closed.
2Thread 2 started.
Accepted connection. Reader and writer created. Read command 2
Server sending result to client
Output closed.
```

## In case of option 3 -

```
MINGW32/c/Users/Aryan Soni/Pictures/Simple-Java-Client-Server-master/client
TCP 172.16.42.84:50335 172.16.42.84:50335 TIME_WAIT
TCP 172.16.42.84:50338 172.16.42.84:50338 TIME_WAIT
TCP 172.16.42.84:50339 172.16.42.84:50339 TIME_WAIT
TCP 172.16.42.84:50341 172.16.42.84:50341 TIME_WAIT
TCP 172.16.42.84:50345 172.16.42.84:50345 TIME_WAIT
TCP 172.16.42.84:50346 172.16.42.84:50346 TIME_WAIT
TCP 172.16.42.84:50347 172.16.42.84:50347 TIME_WAIT
TCP 172.16.42.84:50348 172.16.42.84:50348 TIME_WAIT
TCP 172.16.42.84:50349 172.16.42.84:50349 TIME_WAIT
TCP 172.16.42.84:50350 172.16.42.84:50350 TIME_WAIT
TCP 172.16.42.84:50351 172.16.42.84:50351 TIME_WAIT
TCP 172.16.42.84:50352 172.16.42.84:50352 TIME_WAIT
TCP 172.16.42.84:50353 172.16.42.84:50353 TIME_WAIT
TCP 172.16.42.84:50354 172.16.42.84:50354 TIME_WAIT
TCP 172.16.42.84:50355 172.16.42.84:50355 TIME_WAIT
TCP 172.16.42.84:50356 172.16.42.84:50356 TIME_WAIT
TCP 172.16.42.84:50359 172.16.42.84:50359 TIME_WAIT
TCP 172.16.42.84:50360 172.16.42.84:50360 TIME_WAIT
TCP 172.16.42.84:50361 172.16.42.84:50361 TIME_WAIT
TCP 172.16.42.84:50362 172.16.42.84:50362 CLOSE_WAIT
TCP 172.16.42.84:50367 172.16.42.84:50367 ESTABLISHED

closing
Average response time: 78036 ms

The menu provides the following choices to the user:
1. Host current Date and Time
2. Host Netstat
3. Host current users
4. Host running processes
5. Benchmark (measure mean response time)
6. Quit
Please provide number corresponding to the action you want to be performed: 3
Establishing connection.
Requesting output for the '3' command from ARYAN
Sent output

closing
Average response time: 185 ms

The menu provides the following choices to the user:
1. Host current Date and Time
2. Host Netstat
3. Host current users
4. Host running processes
5. Benchmark (measure mean response time)
6. Quit
Please provide number corresponding to the action you want to be performed: |

MINGW32/c/Users/Aryan Soni/Pictures/Simple-Java-Client-Server-master/server
$ java Server
Server listening on port 15432
Thread 1 started.
Accepted connection. Reader and writer created. Read command 1
Server sending result to client
Output closed.
Thread 2 started.
Accepted connection. Reader and writer created. Read command 2
Server sending result to client
Output closed.
Thread 3 started.
Accepted connection. Reader and writer created. Read command 3
Server sending result to client
Output closed.
```

## In case of Option 4:

```
MINGW32/e/vit/os/os_project_final/server
$ java Server
Server listening on port 15432
Thread 1 started.
Accepted connection. Reader and writer created. Read command 4
Server sending result to client
Output closed.

MINGW32/e/vit/os/os_project_final/client
$ java Client.java
Error: Could not find or load main class Client.java

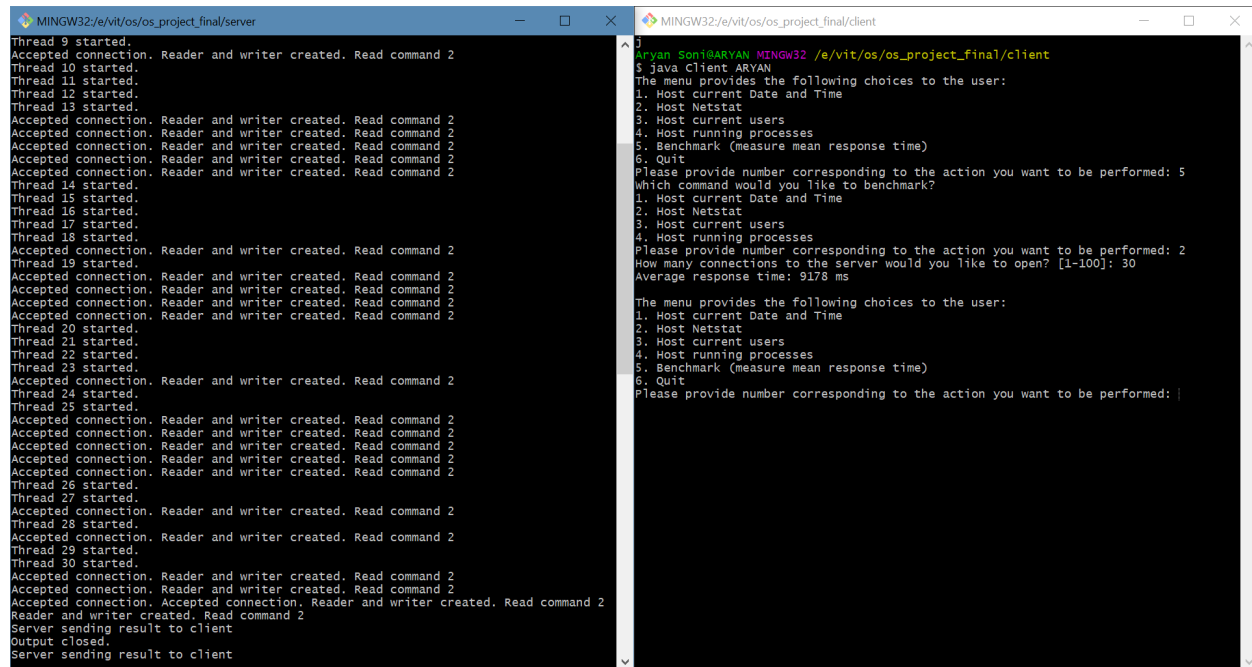
$ java client ARYAN
The menu provides the following choices to the user:
1. Host current Date and Time
2. Host Netstat
3. Host current users
4. Host running processes
5. Benchmark (measure mean response time)
6. Quit
Please provide number corresponding to the action you want to be performed: 4
Establishing connection.
Requesting output for the '4' command from ARYAN
Sent output

  PID  PPID  PGID  WINPID  TTY  UID  STIME  COMMAND
11980  6308  11980  16640  pty0  197609  19:23:29  /usr/bin/bash
2356  11980  2356  14356  pty0  197609  19:23:52  /c/ProgramData/Oracle/JAVA/javapath/java
1976  1  1976  1976  ?  197609  19:24:21  /usr/bin/mintty
1332  1  1332  1332  ?  197609  19:24:39  /usr/bin/ps
15044  1976  15044  6860  pty1  197609  19:24:21  /usr/bin/bash
12124  15044  12124  17308  pty1  197609  19:24:27  /c/ProgramData/Oracle/JAVA/javapath/java
6308  1  6308  6308  ?  197609  19:23:29  /usr/bin/mintty

closing
Average response time: 95 ms

The menu provides the following choices to the user:
1. Host current Date and Time
2. Host Netstat
3. Host current users
4. Host running processes
5. Benchmark (measure mean response time)
6. Quit
Please provide number corresponding to the action you want to be performed: |
```

## IN CASE OF OPTION 5:



The image shows two terminal windows side-by-side. The left window, titled 'MINGW32/e/vit/os\_project\_final/server', displays the output of a multi-threaded Java server. It shows 30 threads starting sequentially, each accepting a connection, creating a reader and writer, and reading command 2. The threads are numbered 9 through 30. The right window, titled 'MINGW32/e/vit/os\_project\_final/client', shows the output of a Java client. It displays a menu of options: 1. Host current Date and Time, 2. Host Netstat, 3. Host current users, 4. Host running processes, 5. Benchmark (measure mean response time), and 6. Quit. The user selects option 5, and the client displays the average response time: 9178 ms. The client then displays the same menu again.

```
MINGW32/e/vit/os_project_final/server
Thread 9 started.
Accepted connection. Reader and writer created. Read command 2
Thread 10 started.
Thread 11 started.
Thread 12 started.
Thread 13 started.
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Thread 14 started.
Thread 15 started.
Thread 16 started.
Thread 17 started.
Thread 18 started.
Accepted connection. Reader and writer created. Read command 2
Thread 19 started.
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Thread 20 started.
Thread 21 started.
Thread 22 started.
Thread 23 started.
Accepted connection. Reader and writer created. Read command 2
Thread 24 started.
Thread 25 started.
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Thread 26 started.
Thread 27 started.
Accepted connection. Reader and writer created. Read command 2
Thread 28 started.
Accepted connection. Reader and writer created. Read command 2
Thread 29 started.
Thread 30 started.
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Accepted connection. Reader and writer created. Read command 2
Reader and writer created. Read command 2
Server sending result to client
Output closed.
Server sending result to client

MINGW32/e/vit/os_project_final/client
Aryan Soni@ARYAN MINGW32 /e/vit/os_project_final/client
$ java Client ARYAN
The menu provides the following choices to the user:
1. Host current Date and Time
2. Host Netstat
3. Host current users
4. Host running processes
5. Benchmark (measure mean response time)
6. Quit
Please provide number corresponding to the action you want to be performed: 5
which command would you like to benchmark?
1. Host current Date and Time
2. Host Netstat
3. Host current users
4. Host running processes
Please provide number corresponding to the action you want to be performed: 2
How many connections to the server would you like to open? [1-100]: 30
Average response time: 9178 ms
The menu provides the following choices to the user:
1. Host current Date and Time
2. Host Netstat
3. Host current users
4. Host running processes
5. Benchmark (measure mean response time)
6. Quit
Please provide number corresponding to the action you want to be performed: |
```

## Conclusion:

Java sockets API (Socket and ServerSocket classes) is a powerful and flexible interface for network programming of client/server applications.

On the other hand, Java threads is another powerful programming framework for client/server applications. Multi-threading simplifies the implementation of complex client/server applications. However, it introduces synchronization issues. These issues are caused by the concurrent execution of critical sections of the program by different threads.

The `synchronized(this){}` statement allows us to synchronize the execution of the critical sections. Using this statement, however, requires a good understanding of the synchronization issues. The incorrect use of `synchronized(this){}` statement can cause other problems, such as deadlocks and/or performance degradation of the program.