Contents

Overview

In this project, I was tasked with creating a Centralized Computing System (CCS) server that provides the following key functionalities:

Service Discovery (UDP): Clients can discover the server in a local network by broadcasting a "CCS DISCOVER" message. The server replies with "CCS FOUND".

Client Communication (TCP): Clients connect via TCP on the same port, issue arithmetic requests (ADD, SUB, MUL, DIV), and receive results or "ERROR".

Statistics Reporting: The server keeps track of global and interval-based statistics (e.g., number of requests, operations performed, errors) and prints them every 10 seconds.

Project implementation

1. **The server is started with:**

java -jar CCS.jar port

where port is a port number.

2. **Service Discovery**

The server opens a UDP socket on the specified <port>.

It continuously listens for any DatagramPacket.

If the incoming message starts with "CCS DISCOVER", the server responds with "CCS FOUND".

Important:

I used a dedicated thread (UDP-Discovery-Thread) that never terminates, always listening.

If a malformed message arrives or doesn't match "CCS DISCOVER", it is ignored.

3. **TCP Communication and Arithmetic Operations**

The server simultaneously opens a TCP ServerSocket on the same <port>.

Whenever a client connects, a new thread handles that client's requests:

1. Receives lines formatted as: <OPER> <ARG1> <ARG2>
2. Tries to parse the integers and the operation.
3. Computes the result or returns "ERROR" if something is invalid (for example: division by zero).
4. Continues until the client disconnects.
5. Operations implemented are "ADD", "SUB", "MUL", and "DIV" (integer division).

I used threads so that many clients can connect to the server simultaneously.

4. **Statistics Reporting**

I created a Statistics class that uses atomic variables to track both lifetime and interval counts. Atomic values allow me to perform thread-safe operations. Every 10 seconds, a Timer triggers and prints two sets of data:

Lifetime Statistics: total number of connected clients, total requests, total sums, etc.

Interval Statistics: same categories, but only for the last 10 seconds, which then get reset.

What was implemented

1. UDP Discovery: Fully functional; sends "CCS FOUND" in response to "CCS DISCOVER".
2. TCP Server: Listens for multiple client connections; spawns a thread per connection.
3. Request Processing: Handles four operations (ADD, SUB, MUL, DIV), checks argument validity, division by zero, etc.
4. Periodic Statistics: A timer-based thread prints cumulative and interval-based stats.
5. Multithreaded: The server can handle multiple concurrent clients.

What wasn’t implemented or could be improved

Thread Pool: Currently, each client spawns a new thread. This works but could be inefficient or potentially unsafe under extreme load. Virtual threads could be used to support the connection of many users.

If I stop the server, existing client connections are simply cut off. Implementing a more controlled shutdown sequence would be better (e.g., draining current requests, then closing connections). But it works, so I believe that there is no need for further development.

Right now, whenever there is an error, the program prints out stack traces or simple error messages. More robust logging could be integrated, to allow easier debugging.

The assignment didn’t require a full client program, only to handle the server side. I tested with a very simple java client class. But I didn’t create a polished client.

Difficulties encountered

Concurrency: Ensuring that multiple threads do not interfere with each other when updating statistics. I learned about AtomicInteger and AtomicLong to avoid data corruption by threads.

UDP Discovery: Implementing the broadcast-based discovery was a major hurdle. I had to set up a DatagramSocket bound to the same port as the TCP service, and then I had to correctly parse incoming discovery messages ("CCS DISCOVER") and respond ("CCS FOUND"). Initially, I struggled with ensuring the server could handle broadcast packets and with choosing the right IP address for broadcasting, but then I remembered that the project is about a “Centralized Computing System” so the IP is the local address. In addition, I learned that socket.receive() is a blocking method, meaning the server must continuously listen for discovery packets in a separate thread. Once I understood how to bind a UDP socket to a specific port and respond to the packet’s source address and port, I overcame the challenge by structuring an infinite listening loop. This approach now reliably allows clients on the local network to discover the server without knowing its IP address in advance.

Parsing Requests: Handling malformed inputs (e.g., not enough arguments, non-integer arguments) required careful validation. I initially forgot to check the argument length, causing ArrayIndexOutOfBoundsException errors.

Timer Scheduling: Learning about Java’s Timer and TimerTask was new for me. I had to ensure the periodic task wasn’t blocking anything critical.

Errors or bugs that still exist

Interrupted Shutdown: If the main thread is interrupted, the server tries to shut down, but any client threads that are still running could cause exceptions or partial disconnects.

New Concepts Learned

Java Networking (Sockets): Using DatagramSocket for UDP and ServerSocket/Socket for TCP was a significant learning curve.

Multithreading: Learned how to manage concurrency and share data (via atomic variables) safely between threads.

Timers: Discovered Java's Timer and TimerTask classes for periodic tasks, as well as the alternative ScheduledExecutorService.

Atomic Classes: Gained experience in concurrency-safe counters to avoid race conditions.