

CS 7610 Programming Assignment 1

Overview

Through this assignment, you will learn how to build a client-server program which is the basis for any networked program including distributed systems. You will learn,

- how to connect software components over the network using sockets,
- how to hide network details by building a simple RPC-like abstractions using stubs,
- how to use threads and synchronization primitives to create multi-threaded client and server programs, and
- how to measure the basic performance metrics that can help you explore design trade-offs.

Before you start to program, first read through this entire document and understand the tasks that you should do for this assignment. You will be running and testing the code in the Khoury VDI cluster and your code will be graded in the same environment. Thus, your code **MUST** compile and run in this Linux environment (Note that the MacOS environment had some issues in the past).

Computer factory

You own a computer factory that produces laptops on-demand. Your main customers are retailers who sell laptops at their stores and the customers continuously place orders to your factory. You will create interfaces to receive orders and ship your laptops, model the behavior of your customers, and manage your factory employees to assemble laptops.

1 Create an order-and-delivery system (simple stubs)

For your factory to operate, customers should be able to place orders and receive laptops. Similarly, your factory should be able to receive orders and ship laptops. For this process, your employees and your customers do not need to know the detailed steps of how orders and laptops are delivered. You will implement a system that sends and receives orders and laptops, but hides the low-level communication details.

The client program models your customers and the server program corresponds to your factory. You should implement the low-level communication code using TCP sockets and hide them under high-level client and server wrappers. You should design `ClientStub` and `ServerStub` classes for the client and the server, respectively. Each client and server thread will later hold their own instances of these classes for communications.

The following describes the basic requirement and you can define more features and functions as needed.

1.1 Client stubs

- `ClientStub.Init(std::string ip , int port)`: initializes the client stub and establishes a new TCP connection to the server.
- `ClientStub.Order(/* order details + any */)`: sends an order to the server and receives the laptop (or laptop information) from the server.

1. Once the function is called the order details should be marshalled into a byte stream.
2. The byte stream is then sent through a socket connection to the server.
3. Next, the function should wait for a server response.
4. Once the server responds back with the laptop information in a byte stream format, the byte stream should be unmarshalled to the laptop information and returned.

1.2 Server stubs

- `ServerStub.Init(/* connected socket to a client*/):` initializes the server stub. It should directly take a socket or a class that includes a socket. In either case, the socket should be the one “accepted” from a listening socket and should be already connected to a client.
- `ServerStub.ReceiveOrder(/* any */):` receives an order from the client.
 1. The function waits for the client order to arrive through the socket connection.
 2. Once the order is received through the socket in a byte stream format, the byte stream should be unmarshalled to an order and should be returned.

The order information will be used to assemble the laptop (i.e., filling in laptop information; see later sections). Once the laptop assembly is finished, the laptop information should be sent back to the customer.

- `ServerStub.ShipLaptop(/* laptop information + any */):` sends the laptop information to the client who ordered it. It should marshal the laptop information into a byte stream, and send the byte stream through the connected socket to the customer.

1.3 Orders and laptop information

This section describes the information that must be included in the order and the laptop information. Both order and laptop information should be designed as objects using structs or classes.

Orders Each order must include,

```
int customer_id;    // customer id
int order_number;   // # of orders issued by this customer so far
int laptop_type;    // either 0 - regular or 1 - custom
```

laptop information Each laptop information message must include,

```
int customer_id;    // copied from the order
int order_number;   // copied from the order
int laptop_type;    // copied from the order
int engineer_id;    // id of the engineer who created the laptop
int expert_id;      // id of the expert who added a custom module
                   // -1 indicates that there is no custom module
```

You may add extra information/variables if needed.

1.4 Hints

- If you are not familiar with C++11 and socket programming, study the resources in the syllabus: C++ programming books (mostly available online through NEU library) and Beej's guide to network programming should be helpful.
- It is recommended that you first build communication classes that encapsulates the socket-level code for clients and servers and then use these classes to build the client and server stub classes. You will be able to reuse the communication class for later assignments.
- Write a simple client and server program that simply sends and receives (order and laptop info) messages to first test whether your communication class works.
- To test network communications while coding, you do not need two different computers. You can use the localhost IP address to test your client and server interactions within a single machine. Use "127.0.0.1" as the IP address and use port number of your choice (but do not use preassigned ports by other services such as 21, 22, 80, etc.).

2 Model your customers and factory

Now that you have the communication interface, you can interact with your customers. The next goal is to model your customer behaviors and to create your factory.

2.1 Customers (client program)

Your laptop is very popular, but each customer is only allowed to issue one order at a time one. Your laptop sells out very quickly in your customers' retail stores and all of your customers place the next order immediately after they receive the laptop from the factory.

Client program structure

You will model each customer as a thread and your client program will model multiple customers that concurrently issue orders to you factory. Your client program should take command line arguments that specify the ip address of the server, the port number of the server, the number of customers, how many orders each customer will place, and the laptop type that the customers want. The client program should take these as arguments in the following format:

```
./client [ip addr] [port #] [# customers] [# orders] [laptop type]
```

For example,

```
./client 123.456.789.123 12345 16 1000 0
```

means that the server ip address is 123.456.789.123, the server port number is 12345, the number of customers is 16, and each customer will place 1000 orders, where the laptop type is regular.

When the client program starts up,

1. The program should create the customer threads as many as the specified customer number.
2. Each customer thread should have a unique `customer_id`.
3. Either the main thread or each customer thread can instantiate connection to the server, but the socket connection should be made once per client stub and each customer should have its own client stub instance.

4. The customer thread should start issuing orders and receiving laptop information as many times as the input argument using the client stub described in the previous section. The order should include the corresponding `customer_id`, `order_number`, and `laptop_type`. For now, only use laptop type 0 (regular type).
5. Once the thread completed all of its tasks, the connection to the server should be closed and the thread should terminate.

By varying the number of customers you should be able to control the amount of concurrent loads on the server.

Performance statistics

Each customer thread should measure how much time it took for each order (i.e., latency, or elapsed time from issuing an order to receiving the laptop information) using `std::chrono::high_resolution_clock` in a microsecond scale. This time period is the latency for an order. Compute the mean latency, maximum latency, and minimum latency based on the measured time information for all orders. For example, if you have 100 customers who place 100 orders each, then you should have 10,000 latency records to compute these numbers. In addition, measure the throughput of your factory (server program) from your client program (i.e., `orders/second`). Print these numbers at the end of each client program execution in the following tab separated format:

```
[avg latency]    [min latency]    [max latency]    [throughput]
```

Hints for performance statistics

1. When computing the mean latency, you should not get 0 as a result: getting the 0 value might mean that you are dividing an integer value with a larger integer value.
2. Collecting performance statistics should not significantly affect the performance of your program.

2.2 Laptop factory (server program)

Based on years of order history, you know that every new customer will place many orders. So you decided to hire and assign an engineer who produces laptops for each customer. Currently, your laptop factory only produces regular type laptops.

Server program structure

The server program is your factory and threads in the server program are your engineers. Each engineer should have a unique id and once an engineer is assigned to a customer, the engineer should directly communicate with the customer (using the stub).

The server program should take port number as a command line argument so that it can accept client connections through the port.

```
./server [port #]
```

For example, for a server to listen to a new connection through port 12345, you should run,

```
./server 12345
```

The main thread of the server program should work as the following.

1. The server program creates sockets to accept new connections and waits for new connections from client programs in a loop.
2. Once a new connection is made, it creates a new engineer thread for the connection. The engineer thread should be given access to a server stub that uses the new connection to directly communicate with the customer thread.

Since your server waits for clients in a loop, you can type Ctrl+c in the terminal to terminate your server program.

Each engineer thread implements a loop that continuously processes orders until the client closes the connection. The thread uses the server stub to communicate with the client.

1. The thread waits for the client's order to arrive.
2. Once the order is received, it starts filling in the laptop information for the order. It copies the information in the order to the laptop information and adds its own `engineer_id` and `expert_id` (use value -1 for now). Then the laptop information is sent back to the client.

If the connection is closed by the client, the thread can escape the loop and terminate. A client closing the connection can be noticed by the return value of `socket recv()` function. When `recv` error occurs (including client connection close) `recv` returns -1. You can figure out more details about the `recv` errors by inspecting the `errno` (<https://man7.org/linux/man-pages/man2/recv.2.html>, <https://man7.org/linux/man-pages/man3/errno.3.html>).

3 Custom laptops

You were only selling regular laptops with no customization and you want to add customized laptops to your production line. Custom laptops require an expert engineer who can add and test customized module to the regular laptop. However, expert engineers are difficult to hire so you cannot assign them per customer.

Therefore, to create a custom laptop, engineers should create the regular laptop and request the expert engineer to attach the custom module. Because there are only a few expert engineers, requests should be queued and processed as the expert engineer becomes available.

Adding expert engineer workflow to laptop factory

To add expert engineer workflow, the server program should take an additional command line argument, which is the number of expert engineers.

```
./server [port #] [# experts]
```

For example,

```
./server 12345 2
```

indicates that the server uses port number 12345 and has 2 expert engineers.

Expert engineers should be added to the server program as a thread pool. Recall that threads in a thread pool wait for tasks to arrive using guards (condition variables). You should use a single FIFO queue that is shared among all expert engineer threads and regular engineer threads. The regular engineers will enqueue requests to the queue and the expert engineers will dequeue the request, process it, and notify completion to the engineer who requested the task. The request sent to the expert engineer should include the laptop information that the engineer was working on,

and the response by the expert engineer should send back the laptop information with the expert engineer's id filled in.

Once the regular engineer issued a request to expert engineers, it should wait until an expert engineer notifies completion of the request, and then send back the laptop information back to the customer.

Each regular engineer thread should look at the customer order and send expert engineer requests only if the laptop type is custom. For regular laptop types, the engineer should solely create the laptop without expert engineer's involvement as described in Section 2.

Now, the main thread of the server program should be modified. The following indicates additional functions that you need to implement.

1. The server program creates the expert engineer request queue and the expert engineer thread pool. Expert engineer threads should be given unique ids that do not overlap with regular engineer ids.
2. The regular engineer threads should be given ways to access the expert request queue so that requests can be sent to the expert engineer threads.

The regular engineer thread should be modified so that it can communicate with the expert engineers, when needed. Necessary additions include,

1. Checking the laptop type in the order and if it is a regular laptop type, then follow the workflow in the previous section.
2. If the laptop type is custom, then send a request to expert engineer threads by enqueueing the request to the expert engineer request queue. Next, wait for an expert engineer thread to respond back. Once the response is received, the regular engineer thread sends back the laptop information with added expert id to the customer.

Each expert engineer thread in the thread pool should work as follows.

1. It waits for requests to arrive in the queue.
2. Once a request is detected and successfully received, the expert engineer thread works on the laptop for at least 100 microseconds (**implement this using `std::this_thread::sleep_for()`**) and add expert id into the laptop information. Then **it responds back to the regular engineer thread that originally sent the request** with the completed laptop information.

Hints Note that the expert engineer must return the response to the regular engineer who initially sent the request; this ensures that the same regular engineer who received the request from the customer can return the final response back to the client. The key components to support the expert engineer workflow are implementing:

1. A shared request queue.
2. An expert engineer thread pool.
3. Mechanisms for regular engineers to enqueue requests to the request queue safely.
4. Mechanisms for expert engineers to wake up and process the request only when there are pending requests in the queue.
5. Mechanisms for regular engineers to wait until any expert engineer processes the request.

6. Mechanisms for expert engineers to wake up the specific engineer who sent the request.
7. Mechanisms for expert engineers to notify their ids to the regular engineer who sent the request.

There can be many ways to implement the key components above. One way to implement a similar function to 1-4 was covered during the class. 5-7 requires your design skills which may involve using various synchronization primitives (e.g., mutex, condition variables, futures, promises, etc.) or shared variables. To make sure that the pair of regular engineer and the expert engineer who work on the same laptop can exchange the wake up handles and ids, you can design the request to include rich information in addition to the laptop information.

4 Playing with the implementation

Now that you have all the components, it's time to play with them and observe the performance trends that are reported from the client program. **Make sure your client and server programs are running on separate Khoury VDI machines (see the next section for more details).** Enter an appropriate number of orders so that your client program runs for at least 10 seconds. (e.g., place orders as many as 10,000.), but not more than 1 minute to save time and shared compute resources.

Experiment 1 Use the regular laptop type and vary the number of customers: 1, 2, 4, 8, 16, 32, 64, 128, and 256. What are the measured mean latency, min latency, max latency, and throughput for each case?

Experiment 2 Use the custom laptop type and set the number of expert engineer to 1. Vary the number of customers: 1, 2, 4, 8, 16, 32, 64, 128, and 256. What are the measured mean latency, min latency, max latency, and throughput for each case?

Experiment 3 Use the custom laptop type and fix the number of expert engineer to 8. Vary the number of customers: 1, 2, 4, 8, 16, 32, 64, 128, and 256. What are the measured mean latency, min latency, max latency, and throughput for each case?

Experiment 4 Use the custom laptop type and set the number of expert engineer to be the same number as the customers. Vary the number of customers and expert engineers: 1, 2, 4, 8, 16, 32, 64, 128, and 256. What are the measured mean latency, min latency, max latency, and throughput for each case?

Repeat experiments Repeat each experiment at least 3 times and average the results. Repetition makes the experimental results more accurate.

Plot the numbers For each experiment, provide one table and two graphs: 1) a table showing min, mean, and max latency numbers; 2) a latency graph that shows the mean latency; and 3) a throughput graph. The columns of the table should be the number of customers and the rows should be min, mean, and max latencies. For the graphs, the y-axis should be the latency or throughput and the x-axis should be the number of customers. For each graph, write a couple of sentences that explain what is happening. Note that you do not have to write C++ code for this. Use any tools you like to create the tables and graphs and include them in the report

5 Running your client and server programs on Khoury VDI

You will use a cluster of shared machines with in the college. If you are accessing the machines from off campus, you should first log on to the Northeastern VPN (search for VPN on Northeastern ITS webpage for detailed information). Use your Khoury account and password (if you do not have them, contact khoury-systems@northeastern.edu) to access the Khoury VDI machines.

You can access different VDI machines via ssh by specifying their addresses. VDI machine addresses range from `vdi-linux-030.ccs.neu.edu` to `vdi-linux-070.ccs.neu.edu`. Refrain from using the “vdi-linux-030” machine and choose random numbers between 030 and 070 so that you and your peers are evenly distributed to different machines.

Before running the client and server programs, make sure both programs are running on different machines. Type “hostname” in each terminal and it will return the machine name that your terminal is connected to. Type “hostname -i” to get the ip address of the machine that you will run the server program on. Use port numbers between 10000 and 65535 for your server program: these are freely usable ports between VDI machines and other port numbers are blocked.

Khoury VDI machines are shared resources, so be mindful of other users.

6 What to submit

1. A report in a pdf format.
 - Summary of your software design, what works and what doesn’t work, if any, and how to run your binary, if it is different from the specification above.
 - 4 tables and 8 plots and their explanations as described in Section 4.
 - Everything should fit into maximum 3 pages. Concise and clear descriptions are the best and the report can be shorter than 3 pages.
2. All source code files with a Makefile. The code should compile with a “make” command to create both “server” and “client” programs.

Place all files in a folder and create a zip file. Name the zip file “first name initials + last name.zip” (e.g., for Ji-Yong Shin create a file `jshin.zip`). Upload the file to the Canvas assignment section.

7 Due date

10/03/2022 (Monday), 11:59 pm. Refer to the syllabus on Canvas for the late policy.

8 Grading - max 15 points

1. The client and server stubs work (3 pt).
2. The client program works (3 pt).
3. The server program works for the regular laptop type (3pt).
4. The server program works for the custom laptop type (3pt).
5. Report (3pt).