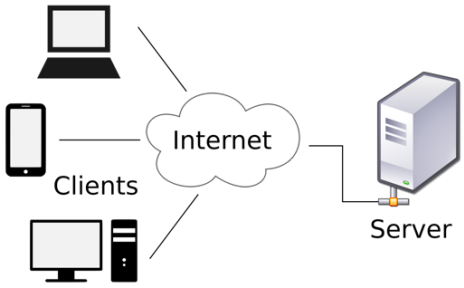
CMPUT 379 Winter 2021

Assignment 3

Client-Server / Monitoring

> **Client–server model** is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters, called clients. Often clients and servers communicate over a computer network on separate hardware, but both client and server may reside in the same system. A server host runs one or more server programs, which share their resources with clients. A client does not share any of its resources, but it requests content or service from a server. Clients therefore initiate communication sessions with servers, which await incoming requests. Examples of computer applications that use the client–server model are Email, network printing, and the World Wide Web.
> [Wikipedia]

You are to implement a client-server architecture. The clients receive transactions that they pass on to the server. Think of a client as a web page where a user initiates a transaction (such as purchasing an airline ticket). The request gets sent to a server (the airline company) for processing. This is shown in the following diagram.



If there are lots of transactions coming in, the server could be multi-threaded, assigning tasks to threads for execution. That is exactly what Assignment 2 was about. In effect, Assignment 3 is the front-end interface to your work in Assignment 2. Do not worry – this is a standalone assignment, which means you do not integrate your Assignment 2 solution into your Assignment 3 submission. However, as discussed below, you may be able to re-use some of your Assignment 2 code.

Specifications

For this assignment you will implement a client-server architecture. Clients generate transactions (`T<n>`) that get sent to the server to execute. Clients may have lulls between transactions, so they wait (`S<n>`) for a transaction to happen. The server receives the transactions, processes them, and then waits for the next transaction.

You will write a program called `server` that accepts one command-line argument:

```
server port
```

where `port` is the port number that the server listens to for communications from clients. The port must be in the range 5,000 to 64,000.

You will also write a program called `client` that accepts two command line arguments:

```
client port ip-address
```

where `ip-address` is the IP (Internet Protocol) address of the machine that is hosting the server, and `port` is the port number that the server will read from (in the range 5000 to 64,000, the same number that the server uses).

For example, executing the command

```
./server 6000
```

on machine ug11 (IP address `129.128.29.41`) results in the server listening for messages on port `6000`. When a message is received, it will process it and then wait for another message.

Executing the following command on ug12

```
./client 6000 129.128.29.41 <client.in
```

creates a `client` process that will send its transactions to ug11's port `6000` for processing. The client process reads in a series of `T<n>` and `S<n>` commands. The `T<n>` commands get sent to the server to be executed, while the `S<n>` command causes the `client` to wait.

Note that you can have multiple clients on multiple machines, all sending to the same server. Try it!

More specifically, a client process receives input (either from the keyboard or redirected from a file) containing two commands:

`T<n>` Transaction. The parameter is an integer > 0. This command will be sent to the server for processing. The client sits idle waiting for the server to complete the transaction.

`S<n>` Sleep. This command simulates having the client wait between receiving new transactions to process. The client waits for a time determined by integer parameter `n`, with `n` between `0` and `100`. You will have a routine that gets called for each sleep request: `void Sleep(int n)`.

The server process reads its input from the port number specified on its command line invocation. Specifically, these messages from clients are transaction requests:

`T<n>` Execute a transaction with integer parameter `n`, with `n` > 0. You will have a routine that gets called for each transaction: `void Trans(int n)`. The parameter n is used by Trans to determine how long to simulate the execution of the transaction.

We will guarantee that the input files used are in the correct format. The code for `Trans` and `Sleep` is available on *eclass* (same as Assignment 2). Do not change these functions!

The server maintains a transaction # that starts at 0 and is incremented with each transaction performed by the server. When a transaction is complete, the server sends back to the client the message D<n>, that the transaction is "D"one, and that its transaction number is n.

When the client reaches the end of input, then it will exit. The server, by definition, does not exit – it does not know all the potential clients that might try to communicate with it.

Sample Output

Note the following in the output below:

- Log file times are actual times (UNIX Epoch times), with two decimal places for the partial seconds. Epoch times can be converted to normal dates, for example using the converter at www.epochconverter.com.
- Each client records all its Trans calls – when sent to the server and when the server acknowledges that the transaction is complete.
- Each client records when it does a Sleep.
- In the log file, T=Trans, S=Sleep, and D=Done.
- The server gives each transaction it receives a unique # (starting at 1). The server puts the transaction number in its log file, and sends the transaction number back to the client (a receipt acknowledging completion of the transaction).
- The semantics of T and S events are exactly the same as in Assignment 2.
- Clients do not do any work between the sending of a transaction request (T) to the server and the acknowledgement that the transaction is done (D). In the real world, the client might itself be a multi-threaded program, dealing with other requests while it waits for a transaction to complete (that is **NOT** part of this assignment).
- Note the impact of long transactions on the client and on the server. Some of this overhead could be mitigated if the server was multi-threaded (as in Assignment 2), but that is **NOT** part of this assignment.

Here is a sample run with a server and two clients running on the same machine. The clients ("./client 5002 127.0.0.1", both on ug11) executed the following commands with the following outputs:

In	ug11.20295 Output	In	ug11.20296 Output
	Using port 5002		Using port 5002
	Using server address 127.0.0.1		Using server address 127.0.0.1
	Host ug11.20295		Host ug11.20296
T1	1583256161.99: Send (T 1)	T100	1583256161.99: Send (T100)
	1583256162.00: Recv (D 1)		1583256162.08: Recv (D 2)
T20	1583256162.00: Send (T 20)	T1	1583256162.08: Send (T 1)
	1583256162.09: Recv (D 3)		1583256162.09: Recv (D 4)
T500	1583256162.09: Send (T500)	T2	1583256162.09: Send (T 2)
	1583256162.49: Recv (D 5)		1583256162.49: Recv (D 6)
T1	1583256162.49: Send (T 1)	T3	1583256162.49: Send (T 3)
	1583256162.49: Recv (D 7)		1583256162.49: Recv (D 8)
T1	1583256162.49: Send (T 1)	T4	1583256162.49: Send (T 4)
	1583256162.49: Recv (D 9)		1583256162.49: Recv (D 10)
T1	1583256162.49: Send (T 1)	S75	Sleep 75 units
	1583256162.50: Recv (D 11)	T5	1583256163.24: Send (T 5)

T10	1583256162.50: Send (T 10)		1583256163.25: Recv (D 17)
	1583256162.50: Recv (D 12)	T6	1583256163.25: Send (T 6)
T10	1583256162.50: Send (T 10)		1583256163.27: Recv (D 18)
	1583256162.51: Recv (D 13)	T7	1583256163.27: Send (T 7)
S50	Sleep 50 units		1583256163.28: Recv (D 20)
T60	1583256163.01: Send (T 60)	T8	1583256163.28: Send (T 8)
	1583256163.06: Recv (D 14)		1583256163.33: Recv (D 22)
T1	1583256163.06: Send (T 1)	T9	1583256163.33: Send (T 9)
	1583256163.06: Recv (D 15)		1583256163.34: Recv (D 24)
T1	1583256163.06: Send (T 1)	T1	1583256163.34: Send (T 1)
	1583256163.07: Recv (D 16)		1583256163.34: Recv (D 25)
S20	Sleep 20 units	T1	1583256163.34: Send (T 1)
T1	1583256163.27: Send (T 1)		1583256163.34: Recv (D 26)
	1583256163.27: Recv (D 19)	T1	1583256163.34: Send (T 1)
S1	Sleep 1 units		1583256163.35: Recv (D 27)
T50	1583256163.28: Send (T 50)	T1	1583256163.35: Send (T 1)
	1583256163.33: Recv (D 21)		1583256163.35: Recv (D 28)
T1	1583256163.33: Send (T 1)	T1	1583256163.35: Send (T 1)
	1583256163.34: Recv (D 23)		1583256163.35: Recv (D 29)
	Sent 14 transactions		Sent 15 transactions

The server ("`./server 5002`", running on ug11) has the following output as it serves the above two clients:

Server Output

```
Using port 5002
1583256161.99: #  1 (T  1) from ug11.20295
1583256162.00: #  1 (Done) from ug11.20295
1583256162.00: #  2 (T100) from ug11.20296
1583256162.08: #  2 (Done) from ug11.20296
1583256162.08: #  3 (T 20) from ug11.20295
1583256162.09: #  3 (Done) from ug11.20295
1583256162.09: #  4 (T  1) from ug11.20296
1583256162.09: #  4 (Done) from ug11.20296
1583256162.09: #  5 (T500) from ug11.20295
1583256162.49: #  5 (Done) from ug11.20295
1583256162.49: #  6 (T  2) from ug11.20296
1583256162.49: #  6 (Done) from ug11.20296
1583256162.49: #  7 (T  1) from ug11.20295
1583256162.49: #  7 (Done) from ug11.20295
1583256162.49: #  8 (T  3) from ug11.20296
1583256162.49: #  8 (Done) from ug11.20296
1583256162.49: #  9 (T  1) from ug11.20295
1583256162.49: #  9 (Done) from ug11.20295
1583256162.49: # 10 (T  4) from ug11.20296
1583256162.49: # 10 (Done) from ug11.20296
1583256162.49: # 11 (T  1) from ug11.20295
1583256162.50: # 11 (Done) from ug11.20295
1583256162.50: # 12 (T 10) from ug11.20295
1583256162.50: # 12 (Done) from ug11.20295
1583256162.50: # 13 (T 10) from ug11.20295
1583256162.51: # 13 (Done) from ug11.20295
1583256163.01: # 14 (T 60) from ug11.20295
1583256163.06: # 14 (Done) from ug11.20295
1583256163.06: # 15 (T  1) from ug11.20295
1583256163.06: # 15 (Done) from ug11.20295
1583256163.06: # 16 (T  1) from ug11.20295
```

```
1583256163.07: # 16 (Done) from ug11.20295
1583256163.24: # 17 (T  5) from ug11.20296
1583256163.25: # 17 (Done) from ug11.20296
1583256163.25: # 18 (T  6) from ug11.20296
1583256163.27: # 18 (Done) from ug11.20296
1583256163.27: # 19 (T  1) from ug11.20295
1583256163.27: # 19 (Done) from ug11.20295
1583256163.27: # 20 (T  7) from ug11.20296
1583256163.28: # 20 (Done) from ug11.20296
1583256163.28: # 21 (T 50) from ug11.20295
1583256163.33: # 21 (Done) from ug11.20295
1583256163.33: # 22 (T  8) from ug11.20296
1583256163.33: # 22 (Done) from ug11.20296
1583256163.33: # 23 (T  1) from ug11.20295
1583256163.34: # 23 (Done) from ug11.20295
1583256163.34: # 24 (T  9) from ug11.20296
1583256163.34: # 24 (Done) from ug11.20296
1583256163.34: # 25 (T  1) from ug11.20296
1583256163.34: # 25 (Done) from ug11.20296
1583256163.34: # 26 (T  1) from ug11.20296
1583256163.34: # 26 (Done) from ug11.20296
1583256163.34: # 27 (T  1) from ug11.20296
1583256163.35: # 27 (Done) from ug11.20296
1583256163.35: # 28 (T  1) from ug11.20296
1583256163.35: # 28 (Done) from ug11.20296
1583256163.35: # 29 (T  1) from ug11.20296
1583256163.35: # 29 (Done) from ug11.20296

SUMMARY
  14 transactions from ug11.20295
  15 transactions from ug11.20296
21.3 transactions/sec  (29/1.36)
```

Implementation Notes

The code for creating a `client` and a `server` process that interact through sockets can be found on many places on the web. The code demonstrated in class is based on the code taken from:

`binarytides.com/server-client-example-c-sockets-linux`

This code works – with a couple of minor changes.

Clients are known by the name `machinename.pid` where `machinename` is the name of the machine the client is running on and `pid` is the process id of the client. Thus, multiple clients on the same machine have different names. Each client maintains a log file that has the name `machinename.pid`.

All UNIX machines support the IP address 127.0.0.1. This is the "loop back" address, which is used to refer to the current machine. In other words, if you run the clients and server on the same machine, the IP-address 127.0.0.1 will always work for you. Make sure your program also works if the clients and server are on different machines.

A server is usually implemented as an infinite loop – it sits around waiting for a client to send it a message. There is a danger that students will log out leaving server processes running in the

background. You are to implement a timer in your program. After 30 seconds of no incoming messages, the server should exit.

The Summary statistics reported by the server are computed using real time (not CPU time) from the start of the first transaction until the end of the last transaction (i.e., the 30 seconds used to exit the server is not included).

Lastly, you are to produce manual pages for your client and the server programs. These need to be produced using the program *groff* with the "*man*" macro package – the same tools that UNIX/Linux systems use to produce their manual pages. There are many resources on the web to show you how to do this.

Warning

There are two caveats you should be careful to watch for when testing your program:

- Two users might unwittingly use the same port number on the same machine. If something goes wrong, try changing your port number.
- Although the client should be able to connect to a server on any machine, this might not be possible for a machine outside the undergraduate lab environment. Because of security measures (including restrictions on port usage and firewalls), your program might not be able to make the connections needed.
- Please use tools to test your code before submission. `Valgrind` checks for memory leaks (usage of this tool would have saved several students marks on Assignments #1 and #2). Compilation using -W warning flags can help identify bugs in advance. Check the return status of all your system calls. Use a debugger! Use these tools to save you time.

Grading

Here are some important things to watch out for:

- The client and server must execute as separate processes.
- Your program should work with multiple concurrent client processes.
- Be careful about boundary conditions.
- Your `makefile` should do the minimum amount of work required to produce the requisite executables. Please use the -O compilation flag for your default client and server executables.
- Do not leave any processes running after you log out. Students who leave processes running – any time before the assignment deadline – will be penalized.

Submission

Submit the following as a single `tar` file:

- All source code files (C, C++, headers) for `client` and `server`,
- The source for your man pages,
- A `README` file (see below), and
- A `Makefile` the produces your executables and a pdf version of your man pages.

The `README` file will help us grade your assignment. It increases your chance of getting a better mark by letting the grader know more about your submission. In the file, please include to see the following:

- Give the names of each of the files in your submission and a summary of their contents.
- List any assumptions you made in doing your assignment. It's easier if you tell us, rather than us having to find out the hard way.
- A summary of the approach you took to solving the assignment.
- Instructions on how to compile and run your program. Yes, this should be standard, but it wasn't with some of the Assignment #1 and #2 submissions.

The assignment is due no later than 10:00 PM on Sunday, November 28. Late assignments received before 10:00 PM on Monday, November 29 will have their assignment grade lowered by 20%.