Internet Programming

Programming Assignment 2: Distributed Programming with Sockets

Deadline: Friday 2 October 2015, midnight

1 A Content-Preserving Server

In this part of the assignment, you will have to implement a simple TCP-based server and its client. The server must listen for connections on **port 4444**. It maintains a **4-byte integer counter** of the number of clients that have connected to it. Each time a client connects to the server, the server increments its counter, writes the counter (as a 4-byte integer) to the client, and closes the connection.

The numbers must be transferred in binary form. E.g., number 55060 must be transferred as four bytes representing one `int`, not as 6 bytes representing the `chars` of string `"55060"`.

The client must connect to the server, read the integer and print it to the standard output in the following format: "`I received: X`", where `X` is the received integer. The client must accept exactly one command-line argument, representing the name of the server it should connect to.

A typical series of client invocations should look like this:

```
$ ./client dastardly.few.vu.nl
I received: 1
$ ./client dastardly.few.vu.nl
I received: 2
$ ./client dastardly.few.vu.nl
I received: 3
$
```

1.1 Client/Server Implementation

Implement three versions of the server described as well a client that can be used with all three of them. The client binary must be called `client`. The three server versions must be called `serv1`, `serv2` and `serv3`. Each version must be implemented using a different structure, as following:

1. **Iterative server**: `serv1` must be implemented using an iterative structure. I.e., it processes client connections one at a time.
2. **On-demand forking server**: `serv2` must support concurrent processing of multiple client connections. It must use the **one-process-per-client structure**.
3. **Preforking server**: Finally, `serv3` must be implemented using a **pre-forking structure**.

1.2 Questions

Q-A Your hacker friend got hold of a vintage Sun SPARCstation 10 workstation[1]. You compile your assignment on both your laptop and the Sun workstation.

Would there be any problems when running the server part of the assignment on your laptop and the client part on the Sun workstation? If yes, can you describe them? What would you do to fix them with minimal changes on your code? If not, why?

Q-B What would happen if 20 clients attempted to contact one of your servers at the same time? Which version of the server would behave best, which worst, and why?

Q-C In the implementation of `serv2`, did you face any challenge in making your server behave correctly when concurrently serving client requests? How did you address this challenge? Can you propose a different approach to addressing it?

Q-D In the implementation of `serv3`, did you face a challenge similar to those described in **Q-C**? Are the approaches you proposed in **Q-C** applicable for the case of `serv3`? Why?

2 A Talk Program

In the second part of this assignment, you will have to implement a talk program that allows two users to chat over the network. Everything the first user types is sent to the second user to be displayed on his/her screen, and vice-versa.

For establishing a connection, one of the programs must be launched in *server mode* and the other in *client mode*. Both modes must be implemented in a single program called `talk`. After the initial connection establishment, the two programs behave identically. During the chat session data must be sent as they become available, **without waiting for the next carriage return**.

Remember that when using `write()`/`send()` on a network file descriptor you are not guaranteed that the whole specified buffer will be transfered. Consult the lecture notes for handling this problem.

If no command-line arguments are supplied, `talk` must start in server mode, listening for incoming connections on **port 5555**. If a command-line argument is supplied, the program must start in client mode and interpret the argument as the hostname of the server it should connect to.

The end of the session is signaled by a Ctrl + C keystroke. After the end of the session, both programs must exit.

2.1 Implementation

You can follow the steps below for the implementation of your `talk` program. **Only submit the most advanced version of your program.**

1. Start by addressing the issue of establishing connections between the two parties. When the connection is established, close it without exchanging any data.

On the program operating as server make sure that you properly close both the socket used for listening for connections as well as the socket created for the communication of the two parties.

[1] SPARCstation 10 features a 32-bit SPARC V8 processor.

2. Update your `talk` program to allow users to chat. At this stage, do not worry if the characters typed by the two parties intermingle.

3. **Optional:** Update your program to separate the text written by the two parties. Use the `ncurses` library to split the screen of your terminal in two parts as seen in Figure 1. The top half should show the text typed by the local user while the bottom half the text typed by the remote user.

The `curses` library **is not thread-safe**! If your program uses multiple threads, make sure you use the proper synchronization primitives.

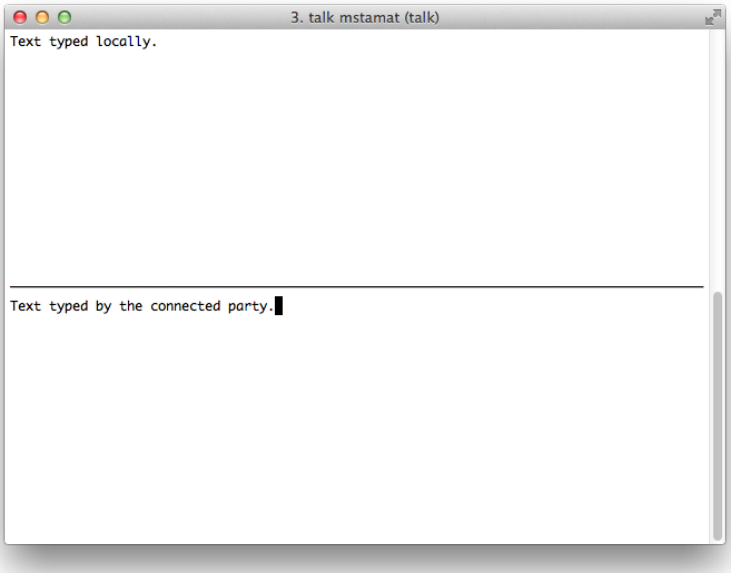

Figure 1: Split-terminal chat using `ncurses`.

2.2 Questions

Q-D Which kind of server do you think is the most appropriate for the server mode of `talk`: (*a*) *iterative*, (*b*) *one-process-per-client*, or (*c*) *preforked*? Why?

Q-E Can you run two instances of your `talk` program on the same machine? Consider the following two cases: (*a*) the first instance runs as server and the second as client, (*b*) both instances run as server. If not, explain how you would modify your program to allow for this.

Q-F The `talk` program needs to receive input from both the keyboard and the network. Describe two different approaches for handling this situation and avoiding blocking on `read()`. Consider the following two cases: (*a*) your program is limited to using a single process/thread, (*b*) your program may use more than one processes/threads. Which of the two approaches did you choose to implement? Why?

A Submission and Grading

A.1 Grading

Your final grade for each assignment will range from 0 to 100. Most assignments will include *optional sub-components* that make the assignment sum-up to a figure larger than 100. Solving the optional parts will only count towards compensating for any other errors.

Grading for the different parts of this assignment is as following:

Item	Points	Note
`client`	6	
`serv1`	6	
`serv2`	12	
`serv3`	12	
`talk`	40	
split screen `talk`	10	
Questions	24	
Total	**100+10**	

A.2 What to submit

You must create and submit a zip file (name doesn't matter) with your assignment. Unless otherwise required, **all files must be stored on the top directory** of the archive. Make sure you don't include un-needed files in your submission (object files, executables, class files, etc.).

You can use the following commands to create the zip file:

```
cd assignment2
zip -r ~/assignment2.zip *
```

The contents of the zip file should be as following:

- `Makefile` → A makefile containing rules that compile all the submitted programs. A `build` recipe must be provided that compiles everything. I.e., issuing the following command must result in all the required binaries being compiled:

  ```
  make build
  ```

- `client.c`, `serv1.c`, `serv2.c`, `serv3.c` → The programs described in Section 1.
- `talk.c` → The most advanced implementation of the program described in Section 2.
- `report.pdf` or `report.txt` → A pdf or plain ASCII text document containing: (*a*) a *short* report (less than one page) on the implementation of this assignment, (*b*) your answers to the questions in this assignment.

A.3 How to submit

Please read the respective section in the first assignment.

A.4 Disclaimers

Make sure you follow the archive structure described above in A.2. Otherwise, automatic evaluation will not work and you will end up with a null grade.

You may submit your assignment multiple times. **Only the last submitted version will be taken into account for your grade.**

Passing all the automated tests does not mean your submission is perfect! **Submissions may also be subject to manual inspection.**