Lab 2: Simple, flawed Server

General Instructions

In this second part, you will start by analyzing a simple (but somewhat flawed) server application. In the next part (Lab 3), you will change the server to fix some of the flaws.

To pass, you will have to answer some questions. The answers to the exercises are to be handed-in in a single PDF document that is uploaded to pingpong. Try to provide concise answers - unless otherwise noted, no exercise should require more than a few lines of text to answer correctly. Some exercises ask you to discuss the question with your partner. To these questions, you do not need to provide written answers.

You will have to make some minor changes to the program in this lab. You don't have to hand in the modified source code in Lab 2.

You should complete the Lab on the BSD Network API in groups of two persons (use the groups you've created in pingpong!). You're allowed to discuss problems with other persons (outside of your group), but everything that you hand in must have been written by either you or your partner. Upload your solution (a PDF document) to pingpong - the PDF document you upload should identify both members of the group.

Source code for the provided programs is available in the appendices, and as downloads from pingpong. (In e.g. Acrobat Reader, right-click on the text-file icon in the appendices and select *"Save Embedded File to Disk"* to get the source code from the PDF file.)

It is assumed that you do the labs in the linux environment at Chalmers. You can, for instance, use the remote machines, which are accessible via SSH. See Appendix D for instructions. Finally, you may use your own computers, however we might not be able to provide support in that case.

If you're a Mac user and wish to complete these labs on your Mac, that's possible. You'll run into one small problem, though: **`MSG_NOSIGNAL`** doesn't exist.
You can work around this by using `SO_NOSIGPIPE` as described at
- http://stackoverflow.com/a/450130.
If you're a Windows user, you can try to port the program to use Winsock2, which is mostly equivalent to the BSD sockets API. If you attempt this, you should already be somewhat comfortable with C/C++ and Windows programming at the very least, though.

Part I - A simple, but flawed server

This first server is based on the iterative echo-server presented during lectures. The full source code with instructions on how to build and use the server program is included in Appendix A. The source code is also available for download from Pingpong.

▶ I.a - Looking at the server code

Grab the source code and take a look at it. Based on the source code and documentation (course literature, *man*-pages and the internet), answer the following questions (and remember, short and concise answers, please!).

Exercise I.a.1 The listening socket is bound to a specific address. What address is this? (Give both the symbolic name used in the code, and the corresponding IPv4 address in numeric or dotted notation).

Exercise I.a.2 In the code, there is a call to `recv()` as follows:

```
ret = recv( cd.sock, cd.buffer, kTransferBufferSize, 0 );
```

The return value `ret` will be one of the following:

(a) ret $= -1$
(b) ret $= 0$
(c) $0 <$ ret $<$ `kTransferBufferSize`
(d) ret $=$ `kTransferBufferSize`

Describe the implications of each case! Additionally, why is `cd.buffer` (see `ConnectionData` declaration) defined to be of size `kTransferBufferSize+1` rather than just plain `kTransferBufferSize`?

Exercise I.a.3 Sending is performed using the `send()` method as follows:

```
ret = send( cd.sock,
  cd.buffer+cd.bufferOffset,
  cd.bufferSize-cd.bufferOffset,
  MSG_NOSIGNAL
);
```

How does the `send()` method indicate that the connection in question has been closed/reset? How does `MSG_NOSIGNAL` relate to this (on linux machines)?

Two different strategies are used to handle errors. During setup, the server program will exit with an error code after encountering and reporting an error. In the loop that handles incoming connections, errors cause the server to stop processing the active client (and close its connection if necessary).

Exercise I.a.4 Discuss the reasons for this behaviour with your partner.

Also, quickly look through the error codes (values of `errno`) possible after `accept()`, `send()`, and `recv()` (check the *man*-pages!). Can you identify conditions where attempting to continue execution might be unreasonable?

(Note: you don't need to provide a written answer for this exercise.)

Time to test the server

Build and start the server! Refer to Appendix A for instructions.

A simple client is available in Appendix B. Build and run the simple client. Make sure that you can connect the client to the server, especially if the server is running on a different machine. If necessary, use the remote machines for these tests (for instructions, see Appendix D).

If you're running on one of the remote machines (or any other shared server), there's a chance that the default port is in use by a different group. If you tell the server to listen on port 0, it'll choose a free port automatically. The actual port the server is listening on is printed to `stdout`.

Any text the client sends should be returned verbatim by the server. In fact, the client will tell you if the response matched the original query. Make sure that the response matches - if not, you're probably connecting to the wrong server. Disconnect the client by signalling it with EOF (in a linux terminal running bash you can press *ctrl-D* to signal EOF to the current foreground program).

Additionally, a second program is available: the multi-client emulator. You can use the multi-client emulator to simulate many concurrent clients that perform a number requests. It is available in Appendix C, where its usage is also described.

▶ I.b - Looking at the clients

Study the source code of the simple client (you may skip the timing code at the very end). Quickly look at the source of the multi-client emulator.

In the client code, the input buffer is defined to be 256 bytes:

```
const size_t kInputBufferSize = 256;
```

Thus, the largest query a client can send is around 256 bytes.

On the server, the size of the transfer buffer is much smaller:

```
const size_t kTransferBufferSize = 64;
```

What happens if the clients sends a message larger than 64 bytes? Try it! (You may temporarily reduce the buffer size on the server if 64+ bytes is too long of message for your taste!)

The multi-client emulator can attempt to establish many TCP connections concurrently. For this, it performs a *non-blocking* connect.

Exercise I.b.1 Discuss with your partner: How is the program notified that a connection attempt has failed or succeeded?

Hint: the process is described in the course book!

(No written answer is required for this exercise.)

▶ I.c - A few simple experiments

Start a single simple client and let it connect to your server. Inspect the state of the connection using e.g. the *netstat* utility:

```
netstat --inet -p
```

The flag `--inet` tells netstat that we're interested in IPv4 connections, and the `-p` flag indicates that (when possible) *netstat* should print the PID and the name of the program and to which the connection belongs. (Note: the flags might be different on other systems. Check your system's documentation!) Make sure that the simple client shows up in the list - see Listing 1 for an example.

```
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address    Foreign Address            State       \
        PID/Program name
...
tcp        0      0 localhost:59868  remote11.chalmers.se:5703  ESTABLISHED \
        8209/client-simple
...
```

Listing 1: *Example netstat output. The client program name is `client-simple`, and its PID (process ID) is 8209. Additional instances of the client program will have the same name, but each instance has a different PID - this allows you to keep track of the instances when you run several at once.*

Now, attempt to connect a second client (the first client should still be connected).

Exercise I.c.1 Try to send messages with each of the clients. Describe the results – do you receive a response immediately?.

Check with *netstat* and document the status of the connection from each client.

Disconnect the first client.

Exercise I.c.2 When you disconnected the first client, what happened? Explain why.

In the source code of the simple client, change the line

```
#define MEASURE_ROUND_TRIP_TIME 0
```

to

```
#define MEASURE_ROUND_TRIP_TIME 1
```

and recompile the client (see Appendix B if you have trouble compiling or linking). Now the client will measure the time it takes to send a query and receive the response from the server.

Exercise I.c.3 Measure the round trip time when the client and server are running on the same machine. Also measure the round trip time when they are on different machines.

Can you observe any differences? Write down the times. (Note: take the average of a few (> 5) attempts.)

Exercise I.c.4 Measure the round trip times for two concurrently connected simple clients (similar to exercise I.c.1).

Discuss with your partner: What is the largest factor in the measured round trip time of the second client?

(No written answer required.)

► I.d - Simulating many clients and requests

Time to switch to the multi-client emulator. Compile the program, if you have not already (see Appendix C for instructions). The multi-client emulator can be used to perform some simple stress tests, where you can choose to emulate a specific number of clients that attempt to connect concurrently, each of which can send a specific number of queries.

For instance, to simulate 7 clients of which each sends 255 queries, you'd run

```
./client-multi <hostname> <port> 7 255
```

The program keeps track of some simple data, including the time required to establish a connection, and the time taken to perform the queries. It will, by default, print the average and minimal/maximal times.

Exercise I.d.1 Run the above command (make sure that the server is still running), and note the results.

Increase the number of clients a few times (try, for example, using 10, 15, 30, 50 and 100 clients). What happens to the minimal/maximal times required to establish a connection? What happens to the round-trip times? Did any errors occur?

Exercise I.d.2 Take note of the timing results. You will want to compare them to results in the next Lab/Exercise.

(You don't have to hand in the results, though.)

Reduce the number of clients to 7, and instead increase the number of messages (try e.g. 1000, 5000, 10000 messages). Compare the timings as you did above.

Now, run with 100 clients and 10000 queries. This is also an excellent time to grab a cup of coffee. Take note of the times that are reported. You'll want to compare them to the ones you will take in Lab 3.

Finally, you're going to perform a (very simple) denial of service attack. Connect a simple client to the server and make sure that the server is handling that client actively. **Note the current wall-time!** Use the multi-client to open a large number (> 50) connections to the server.

At some point, connections of the multi-client should start to time out. This may take several minutes. **Write down the time when the connections timed out!**

Exercise I.d.3 How long did it take for the connection attempts to time out?

Troubleshooting: If you don't get time outs, you can check the status with netstat. Some connections will display the status `ESTABLISHED`*, and some should have the status* `SYN_SENT`*. The*

latter group will eventually time out. If there are no connections with the status `SYN_SENT`*, try increasing the number of attempted connections with the multi-client.*

The simple client is blocking the server completely. Stop it. Some pending connections from the multi-client should now complete.

In the next Lab/Exercise you'll improve the server to be able to handle the situations that we've created more gracefully. Stay tuned.

A- A simple, but flawed server - iterative server program


Iterative Server Source Code


http://www.cse.chalmers.se/~billeter/courses/server-iterative.cpp

Build the server program using following command:

```
g++ -Wall -Wextra -g3 server-iterative.cpp -o server
```

This command will produce the executable binary called *server*, from the source file *server-iterative.cpp*. The flags `-Wall` and `-Wextra` enable additional diagnostic warnings during compilation (`-Wall`: enable all Warnings; `-Wextra`: enable extra Warnings). The `-g3` flag instructs the compiler to generate debugging information, which is embedded into the executable binary. (Note: `g++` is a C++ compiler; the code uses some C++ features!)

Start the server by issuing either of the following two commands:

```
./server
./server 31337
```

The first command starts the server on the default port, 5703. In some cases, this port may already be used, in which case you can use the second form to specify a custom port explicitly (here 31337). **Remember which port (and machine) your server runs on — some of the tests performed in this lab might interfere with other groups' servers, or indeed completely unrelated servers!**

B- Simple, Single-Client Program


Single-Client Source Code


http://www.cse.chalmers.se/~billeter/courses/client-simple.cpp

Build the standard single-client program with the following command:

```
g++ -Wall -Wextra -g3 client-simple.cpp -o client-simple
```

Refer to Appendix A for an explanation of the flags.

Start the client by issuing the following command:

```
./client-simple remote.example.com 5703
```

The first argument identifies the target host (here `remote.example.com`), and the second argument identifies the target port (here `5703`). The client attempts to establish a single TCP connection to the target address (host+port).

The client program can be configured to measure the round-trip time of a single message. This is achieved by changing the line

```
#define MEASURE_ROUND_TRIP_TIME 0
```

to

```
#define MEASURE_ROUND_TRIP_TIME 1
```

The timing code may require some additional libraries. For instance, on Linux, you have to link against the *rt* library. Thus, the command to build the program becomes

```
g++ -Wall -Wextra -g3 client-simple.cpp -o client-simple -lrt
```

The additional flag, `-lrt`, tells the linker to additionally consider the library "*rt*" during linking.

C- Multi-Client Emulation Software - The Destroyer of Worlds

Multi-Client Emulator Source Code

http://www.cse.chalmers.se/~billeter/courses/client-multi.cpp

Build the multi-client emulator program with the following command:

```
g++ -Wall -Wextra -g3 client-multi.cpp -o client-multi -lrt
```

Refer to Appendix A for an explanation of the flags.

The multi-client emulator accepts up to five arguments on startup:

- remote host, e.g. `remote.example.com`
- remote port, e.g. 31337
- number of concurrent connections
- *(optional)* number of times the message is sent
- *(optional)* the message that is sent

By default, the message is sent once, and the default message is equal to "*client%d*". The message may contain a single '*%d*' placeholder, which is replaced by a connection ID before the message is sent. For example, the default message would be expanded to "*client0*", "*client1*", and so on.

For example, the following command would emulate 255 clients that each send 1705 messages to a host called `remote.example.com` on port 5703. The message template used is "*Client %d says hello.*":

```
./client-multi remote.example.com 5703 255 1705 'Client %d says hello.'
```

Note: Avoid pointing the multi-client emulator at servers that you do not control/own!

D- Instructions for accessing the remote machines at Chalmers

There are five central machines accessible to students via *ssh* at Chalmers:

- `remote1.student.chalmers.se`
- `remote2.student.chalmers.se`
- `remote3.student.chalmers.se`
- `remote4.student.chalmers.se`
- `remote5.student.chalmers.se`

Login requires a valid CID (username) and password.

Most GNU/Linux and Mac OS X installations include a SSH client by default. For example, to login to the *remote1* machine, issue the following command in a terminal:

```
ssh CID@remote1.student.chalmers.se
```

You should replace `CID` with your actual CID in the example above!

Some additional examples, including using SCP to transfer files between different machines, are available at

 http://en.flossmanuals.net/command-line/ch029_ssh/

Windows users can use the *Putty* SSH client, freely available at

 http://www.chiark.greenend.org.uk/~sgtatham/putty/

Windows users may want to use *WinSCP* to transfer files. WinSCP is freely available at SourceForge:

 http://sourceforge.net/projects/winscp/

.