Lab 1.4: Scaling up to more connections

General Instructions

In the last lab, you improved the simple server by introducing I/O multiplexing. Using **`select()`**, the new server can handle multiple connections concurrently. However, this server is not designed to process more than a few hundreds of simultaneous clients. In this lab, we will allow the server to scale for more incoming clients.

Continue working in the same group as with the previous lab. You will start making improvements on your code from lab 3, but save a copy of your original code for comparison later.

When you are done remember to submit your quizz answers on Canvas.

Part I - Identifying the problem

In the previous lab, when you where measuring the response times of the server, you might have noticed that the performance deteriorates rapidly when the number of concurrent clients exceeds a certain point.

The reason is that **`select()`** was not designed to scale for a large number of file descriptors. It is not only that the internal implementation of **`select()`** is slow, but also that the way the server handles connections is not optimal.

Remember how **`select()`** works: it requires a set of read and a set of write file descriptors (that need to be re-created after every **`select()`** call). After calling **`select()`**, the read and write sets are modified by including only the file descriptors that have work to do (read from or write to). Then, the server need to iterate over ALL file descriptors to figure out which are the ones that are active. Imagine a server with many open sockets. Every time something is written in one socket and **`select()`** returns, the server needs to iterate over all connections to serve only one.

Servers using **`select()`** could not scale to many concurrent connections and that was known as the C10K problem. The solution was the introduction of system calls that can handle more file descriptors. The Linux implementation of such a system call is called **`epoll()`**.

Your next task is to modify the server from lab 3, so that it uses **`epoll()`** instead of **`select()`**. The are a couple of differences in the API that you need to keep in mind. You will find relevant information in the man pages of **`epoll()`** and **`epoll_wait()`**.

> Note that, in this lab, we will use **`epoll()`** in level triggered mode, in order to re-use the code from the previous labs.

In the following link you can find an interesting article about the C10K problem and many server design techniques.

http://www.kegel.com/c10k.html

General design

In general, instead of relying on file descriptor sets, **`epoll()`** creates a list of events, related ONLY to the file descriptors that have work to do. Then the server knows exactly which connections to serve.

Lets take it step by step. First create a new instance of **`epoll()`**, using **`epoll_create()`**. Set the maximum number of events to 8. The file descriptor that **`epoll_create()`** returns is where we

will attach all other file descriptors to.

Monitoring file descriptors

Every time we have a new file descriptor that we need to monitor, we have to attach it to the file descriptor of **epoll()**, using **epoll_ctl()** (see the man page of **epoll_ctl()** for more info).

epoll() uses the data structure `epoll_event` to handle file descriptors:

```
struct epoll_event {
    __uint32_t   events;      /* Epoll events */
    epoll_data_t data;        /* User data variable */
};
```

The field `events` describes the state of the file descriptor (in our case it will be either EPOLLIN or EPOLLOUT). The field `data` is defined as

```
typedef union epoll_data {
    void        *ptr;
    int          fd;
    __uint32_t   u32;
    __uint64_t   u64;
} epoll_data_t;
```

That means that it can be either a pointer or an integer (signed or unsigned, 32 or 64 bit). You will have to use that field to store all the information you want about the specific socket (use the integer as an index to some array where you keep track of connections or use the pointer to point to that connection record).

Waiting for an event with **epoll_wait()**

The general design of the server stays the same: We will use the system call **epoll_wait**,instead of **select()**, that waits for any event that is of interest to our server. However the behaviour of **epoll_wait()** is slightly different. The input argument of **epoll_wait()** is an array of `epoll_event` structs. Upon return, that array will contain the events that happened during waiting time. The return value of **epoll_wait()** is the number of event that happened. That way, the server can iterate through the active event records and serve the connections.

The example found in the man page of **epoll()** is very relevant and will give you an understanding of how **epoll()** is used. Note that, other than the main processing loop of the server, the rest of the code should probably be the same as in lab 3 (e.g. you don't have to change anything in the logic of `process_client_recv()` or `process_client_send()`).

▶ I.a - Experiments with the concurrent server

If you followed the comments above and studied a few examples, you should be able to make your code **epoll** - compatible. When you have a working version, go through the simple tasks below.

Exercise I.a.1 Make sure that you have a running server that works with **epoll()** and has the same functionality as before.

Try connecting with a few simple clients, send messages and see the responses. Connect with a few hundred clients using the multi-client emulator, just to make sure that everything works as intended.

Exercise I.a.2 Time to see if we gained any performance improvement over the old server.

Use the multi-client emulator to run a few tests on both servers, comparing against the same number of clients and messages each time. Measure the values you find interesting(e.g total time, connection time, round trip time) and create a plot, showing the performance of the new server compared to that of lab 3.

In order to make the measurement process a little bit simpler, we have provided a naive python script that performs many runs one after the other and summarises the results.

The script usage looks like the way you run the multi client emulator, except that you can provide a series of client numbers instead of only one. Here is an example:

```
python measurement_script.py client-multi remote.example.com 5703 100 200 300 10
```

This will run the multi-client emulator for 100 , 200 and 300 clients with 10 messages per client (each run is repeated 5 times and an average is computed). At the end, the script will output the average connection and total times for 100, 200 and 300 clients respectively.

The C10M problem

If you are interested to know what is the next step in highly scalable server design, this is an article, along with a video presentation, about the next generation of servers:

http://highscalability.com/blog/2013/5/13/the-secret-to-10-million-concurrent-connections-the-kernel-i.html

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

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

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

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 two central machines accessible to students via *ssh* at Chalmers:

- `remote11.chalmers.se`
- `remote22.chalmers.se`

Login requires a valid CID (username) and password. These machines run Linux, so you should be able to complete this lab there.

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@remote11.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:


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, which is freely available at:

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

E- Using STD::VECTOR to track connections

A dynamically sized vector (similar to an array) containing `ConnectionData` elements can be declared as

```
std::vector<ConnectionData> connections;
```

Before any items are added, the vector is empty and has size zero. You can query this with the `clear()` and **`size`**`()` methods, respectively:

```
connections.empty() // <-- will return (bool) true
connections.size() // <-- will return (size_t) 0
```

A new element can be added to the vector using the **`push_back`**`()` method:

```
ConnectionData connData;
// initialize connData ...
connections.push_back( connData );
```

Elements of a **`std::vector`** are accessed like the elements of an ordinary C-array:

```
ConnectionData d = connections[0]; // get first element
size_t i;
// assign some value 0 <= i < connections.size() to i
connections[i].sock = -1; // set i:th connection's sock member to -1
//...
```

Of course, like an ordinary C-array, behaviour when accessing out-of-bounds elements is undefined (=bad)!

A simple method to iterate over the elements in the vector is as follows:

```
for( size_t i = 0; i < connections.size(); ++i )
{
  printf( "Connection %zu: in state %d and has socket %d\n",
    i, connections[i].state, connections[i].sock );
}
```

(although seasoned C++ programmers might prefer using (const) iterators.)[1]

Connections with invalid sockets (i.e. the `sock` member is equal to -1) can be removed using the following code snippet:

```
connections.erase(
    std::remove_if(
        connections.begin(), connections.end(), &is_invalid_connection
    ),
    connections.end()
);
```

[1] The `%zu` modifier is used to format the **`size_t`** variable `i`. Technically, this is a C99 extension, and not necessarily supported by all C++ standard libraries.

The `is_invalid_connection()` function is not a standard function, but is defined at the very end of the iterative server's source code!

Important: You might be tempted to use the **`erase()`** method to remove elements while looping over the elements in the vector.
Don't do this! The **`erase()`** method will invalidate the current iterator (if you use iterators) and *shift* elements following the element that was erased. (Which means, that unless you're careful, you will miss elements or remove the wrong ones.)

Documentation is available at e.g.  http://www.cplusplus.com/reference/vector/vector/ .