Lab 3: I/O multiplexing for concurrent connection handling

General Instructions

In the last lab, you analyzed a simple server. You probably discovered that it's not very good at handling many requests concurrently. In this lab you will modify the server to handle this better.

Continue working in the same group as with the previous lab. This time, you will mainly hand in the modified source code as a single `.c` or `.cpp` file.

> This manual uses the code from Lab 1.2 as a basis (iterative server) to guide you towards a solution.
> We have also added a skeleton code (check the Canvas page of the labs) that shows the main structure of the solution, with placeholders for you to add your code.

Part I - Improving the server

As you may have noticed, the simple iterative server has a few flaws. Clients are handled in a serial manner, and a single client can block the server more-or-less indefinitely.

Your next task is to turn the simple iterative server into a concurrent server using I/O multiplexing with **select**(). If you're unsure about **select**() and I/O multiplexing (event-based IO), you should -at this point- review some of the relevant course material.

Additionally, some help might be found in the source code of multi-client emulator program. The multi-client emulator also uses multiplexed I/O with **select**() to handle the large number of connections it attempts to establish.

Non-Blocking Modes

So far, most network related operations in the iterative server are blocking, i.e. they wait for a certain event to occur, and return when this event has taken place. For example, the line

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

would wait for data to become available on the socket `cd.sock`. In a concurrent server, blocking operations must be avoided: for example, during the time the program waits for data to become available on the above socket, a new client might attempt to connect and perform its transaction.

To make sure that no operations are blocking, sockets can be put into non-blocking mode (see `set_socket_nonblocking()` in the server's source code). Change the line

```
#define NONBLOCKING 0
```
to
```
#define NONBLOCKING 1
```
and calls to `set_socket_nonblocking()` will be made at the appropriate locations.

Recompile and run the server. The server should now repeatedly print error messages. For example:

```
./server-concurrent
accept() failed: Resource temporarily unavailable
accept() failed: Resource temporarily unavailable
```

. . .

Since the listening socket is in non-blocking mode, attempts to call **accept**() return immediately and indicate that no new connection could be accepted (there was none!) using an error condition of **EWOULDBLOCK** or **EAGAIN** (the error conditions are equivalent and, on some systems, they are represented by the same error code).

> Later, as you work through the lab, the error messages are going to disappear. Right now, if you see those message, everything is as it should be.
>
> At no point, however, should you try to "fix" problems by disabling/commenting parts of the source code that print error messages - that is never the right answer. If your program produces error messages after you think you've fixed a certain problem, ask for assistance, rather than glossing it over.
>
> If you remove the parts printing error messages anyway, we will glare at you angrily (at the very least)!

Waiting for an event with SELECT()

Using **select**(), it is possible to wait for some event (e.g. an incoming connection, or data becoming available, and so on) to occur - on one or more sockets. The **select**() method will also identify on which sockets the event(s) took place.

Start by just adding the listening socket, listenfd, to the set of file-descriptors watched for non-blocking reads to become possible (named readfds in the *man*-page of **select**())[1]. Also add an appropriate call to **select**().

When **select**() returns, its return value describes the number of events that have occurred (or -1 if there was an error). Additionally, **select**() will have changed the file descriptor sets passed to it - they will only contain the file descriptors which are ready for reading (readfds) or writing (writefds). Thus, after the call to select, you can check, with **FD_ISSET**, if the listening socket remains a member of the readfds - if so, a new connection is ready to be accepted.

Compile and run the server with these modifications. Make sure that **select**() returns when you attempt to connect to the server. The program should only call **accept**() when this event occurs (and listenfd is set in the readfds).

At this point, the server program should no longer produce the "*accept() failed: Resource ...*" error messages (although if you establish a connection, a different error will probably be raised).

Tracking open connections

Currently, the server will attempt to receive from and send to a client directly after accepting the connection. Of course, that is a bad idea in a concurrent server - the server should wait until data is available for reception (and conversely, that it is possible to send data without blocking). In order to do this, the server needs to keep track of open connections.

A ConnectionData structure is already defined in the server - each open connection will require a copy of that structure. A convenient way to give each connection a unique copy is to dynamically allocate the structure and store it in a data structure of some kind. A simple such data structure is the linked list - if you wish, you may (re-) implement a linked list and use that.

Of course, if you're lazy (you should be!), you may simply use a C++ container such as the STL **std::vector**<> or **std::list**<> template classes. Some hints on this are given in Appendix E, if you're unfamiliar with C++ STL containers.

---

[1]Accepting a new connection is, in this case, considered to be equivalent with reading from the socket. Accepting the new connection is still performed with **accept**(), though!

More SELECT()

So far, only the listening socket is added to the file descriptor sets for **select**(). But, the server also needs to be notified when data can be received from or sent to any open connection.

Before calling **select**(), also insert the sockets from open connections into the correct sets. Connections that are expected to receive data (i.e. their state is eConnStateReceiving) should be added to the readfds; connections that need to send data (their state is eConnStateSending) should instead be added to the writefds.

The state of new connections is initialized to eConnStateReceiving by the code. Additionally, the methods process_client_recv() and process_client_send() will change the state when appropriate - that is, you don't have to worry about that part.

What you *do* have to worry about is *calling* the above methods at the correct times. The method for receiving data, process_client_recv(), should be called when data is available to be received on a given connection. The process_client_send() method should be called when the socket is ready for sending. Implement this.

Additionally, if either method returns **false**, the connection should be closed (so that the socket's file descriptor can be reused for a new connection), and the connection's unique copy of the ConnectionData structure should be removed from the list of open connections. Also implement this.

▶ I.a - Experiments with the concurrent server

Congratulations - you should now have a working concurrent server. The next step is to repeat some of the experiments that were performed earlier.

> **Exercise I.a.1** Establish two connections to the improved server using two instances of the simple client program.
>
> 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.

> **Exercise I.a.2** Repeat the measurement with 100 clients and 10000 queries, using the multi-client emulator. Compare to the timings you wrote down in the previous Lab.

> **Exercise I.a.3** Consider the very simple denial of service attack we performed in the previous lab, where a single client (using the simple client program) would block the server for all other clients (emulated using the multi-client program).
>
> Is it still possible to block the server using a single client? (Try it!)

> **Exercise I.a.4** Discuss with your partner: There may still be some issues with the server. Can you identify any? (For instance, is it still possible for a determined person to deny service to other persons? If so, under what conditions? Suggest possible solutions!)
>
> (You don't have to provide a written answer to this exercise.)

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 <u>all</u> <u>W</u>arnings; `-Wextra`: enable <u>extra</u> <u>W</u>arnings). 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) <u>your</u> 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 <u>l</u>ibrary "<u>*rt*</u>" 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:

```
1 ConnectionData connData;
2 // initialize connData ...
3 connections.push_back( connData );
```

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

```
1 ConnectionData d = connections[0]; // get first element
2 size_t i;
3 // assign some value 0 <= i < connections.size() to i
4 connections[i].sock = -1; // set i:th connection's sock member to -1
5 //...
```

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:

```
1 for( size_t i = 0; i < connections.size(); ++i )
2 {
3   printf( "Connection %zu: in state %d and has socket %d\n",
4     i, connections[i].state, connections[i].sock );
5 }
```

(although seasoned C++ programmers might prefer using (const) iterators.)[2]

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

```
1 connections.erase(
2     std::remove_if(
3         connections.begin(), connections.end(), &is_invalid_connection
4     ),
5     connections.end()
6 );
```

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/ .

---

[2]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.