Software Architecture

Hand-in Assignment

Week 6

Instructions:

- Submit your solutions through GitHub Classroom and **remember** to submit a link to your repository in BrightSpace. Otherwise, I will have a hard time distinguishing between the different groups.
- When submitting, you should push your solutions to the existing repository. In this assignment you only need to add code where **"Implement this"** is stated.
- In this assignment there is a possibility to test your solutions with **autograde.py**. I hope it works on your machine. If not, you should spend some time setting up Python and CMake. Again, it's optional. Regardless, it's a huge advantage to read and understand the two test files in order to implement.
- You **must** add comments in your programs to explain your code. Any solutions submitted without any comments will not be graded.

Description

During this exercise we will take a look at how network programming can be implemented in C/C++. Specifically we will implement a client-server pattern which allows a client to download an image from a server.

The client-server is a common architecture used throughout the internet, a classical example being a web-browser loading a webpage. Here the web-browser acts as the client and the server is a program deployed to some machine, which loads the webpage from a database and sends the response to the client.

It should be stressed that the server and client are simply two computer programs that communicate with each other. Traditionally, we think of servers as running on different machines, but there is no limitation on where these programs are running. **During this exercise you will be running both client and server on your own local machine.**

System Requirements & Setup

For this assignment, we will run the code directly in the terminal instead of using a Virtual Machine. This approach reflects industry standards and makes it significantly easier to install the necessary third-party networking packages (such as the Boost library) directly on your operating system, avoiding the overhead and complexity of setting up a virtualized environment.

Depending on your operating system, follow the instructions below to set up your environment:

Windows (WSL)

Windows users must use Windows Subsystem for Linux (WSL) to get a native Ubuntu terminal.

1. Open PowerShell as Administrator and run: `wsl -install`
2. Restart your computer. Upon restarting, a terminal will open and ask you to create a UNIX username and password.
3. Update your new Linux environment and install the required tools by running:

```
sudo apt update && sudo apt upgrade
sudo apt-get install build-essential libboost-all-dev
```

macOS (Homebrew)

Link: https://brew.sh/
Mac users can use the Homebrew package manager to install the necessary libraries.

1. Install the Command Line Tools by opening your terminal and running: `xcode-select -install`
2. Install Homebrew by running the following in your terminal:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew
/install/HEAD/install.sh)"
```

3. Install the networking library: `brew install boost`

Linux (Ubuntu/Debian)

If you are already running Linux natively, simply open your terminal and run:

```
sudo apt-get install build-essential libboost-all-dev
```

Installing Networking Library

Since TCP and UDP sockets are not currently part of the C++ standard library, we must install a third-party library that takes care of this for us. We will be using the `boost::asio` library which is part of the boost ecosystem of packages, which provides portable code for some of the functionality that is not part of the C++ standard library.

Note that different networking-related frameworks such as `boost::asio` target different levels of networking abstractions. In other words, if you are implementing a new video streaming protocol you would pick another library than if you were writing a program that fetches an image using HTTP.

The choice of `boost::asio` lets us build the application directly on top of the transport layer that does not rely on other application layer protocols such as HTTP. This is a purely didactical choice, hopefully leading to a better understanding of what a client and server really are.

Compiling the Code

Because we are working directly in the terminal, you will need to compile your C++ files manually. The exact compilation command depends on your operating system, as the Boost library is installed in different locations.

For Linux and Windows (WSL):
The package manager installs Boost in the default system path, so you can simply run the standard compile command:

```
g++ -std=c++11 src/daytime_server.cpp -o daytime_server
```

For macOS (Homebrew):
Homebrew installs packages in a custom directory, so you must explicitly tell the compiler where to find the Boost header files using the `-I` flag:

```
g++ -std=c++11 src/daytime_server.cpp -o daytime_server -I/opt/homebrew/include
```

Note: When compiling the client files, simply replace `daytime_server` with `daytime_client` in the commands above.

Automated Testing of Networking

One challenging aspect of developing networked applications is testing. The fact that client and server have to be running on separate threads makes it somewhat harder to do automated testing.

As such the test cases you are used to seeing will not be provided for this exercise and we will instead rely on manual testing, e.g., inspecting the transmitted image to see if things went well.

Exercise

The exercise is split into two parts: (1) you will review an example of a simple network application for fetching the current time from a server. (2) you will extend the example to send an image as a series of bytes rather than a text string describing the date.

Daytime server

1. Inspect the files `daytime_client.cpp` and `daytime_server.cpp`. Add your own notes in the form of comments describing what the different methods do. Focus on the high-level structure like where data structures are initialized and where the reading and writing to the socket occurs.
2. Compile both programs using the instructions in the "Compiling the Code" section.
3. Test the programs. Start the server from a separate terminal (the server program will loop indefinitely):

   ```
   ./daytime_server
   ```

   Then from another terminal start the client, specifying "127.0.0.1" (this is a special loopback address):

   ```
   ./daytime_client 127.0.0.1
   Fri Mar 12 11:07:48 2021
   ```

Image Transmission

The next step in this exercise is to modify the existing programs to send an image instead of a string. Like any other file on your computer, an image is represented by a number of bytes. What makes it an image rather than a blob of bytes is simply that we encode and interpret these bytes as an image. A file extension like `.png` is one way to indicate that the contents of a file should be treated as an image.

1. Examine the files `image_client.cpp` and `image_server.cpp`.
2. Implement the `get_image` function in `image_server.cpp`. This loads the contents of the `cat.jpg` file into a vector of bytes. You may have to copy the `cat.jpg` image into the directory from which the program is launched.
3. Implement `save_image` in `image_client.cpp`. This should store the received bytes as an image `copycat.jpg`.
4. Start the image server and then the image client. Does the `copycat.jpg` appear? If not, make the necessary changes to the programs and verify that the image is transmitted.

Hint: You can implement all the code for reading and writing images in `main.cpp` first and integrate later to avoid the hassle of dealing with sockets.