Project 0: Environment Setup, Sockets

Due on Friday, October 18th, 2024 at 11:59 pm (Week 3)

Thank you to Professor Lixia Zhang; former CS 118 TAs Xinyu Ma, Varun Patil, Tianyuan Yu; and former CS 118 LA Paul Serafimescu for the original version of this project.

Quick Links

Starter Code: https://github.com/CS118F24/project0starter
Socket Programming Tips
Non-blocking file descriptors

Overview

Welcome to CS 118! We hope you have a fun experience working on our projects this quarter. To make things easier, we've designed a local environment that replicates the Gradescope autograder. In other classes, you might have used the autograder and submitted hundreds of times to perfect your solution. Since you can run the exact environment on your machine, you'll have a much better (and faster) time making improvements to your solution. In addition to setting up the local environment, we'll also help you set up your IDE and gain basic familiarity with socket programming.

This document will go over the setup for Windows and macOS (with Visual Studio Code as an IDE). If you're using Linux (or any other operating system/IDE), we'll assume you know what you're doing. Nonetheless, feel free to attend TA office hours if you need any help.

By the end of the project, you'll have:

1. Set up WSL if you're on Windows & Developer Tools if you're on macOS
2. Installed Docker Desktop
3. Installed Visual Studio Code and the C/C++ extension
4. Installed OpenSSL 3 (useful for a later project)
5. Explored Gradescope's grading harness and ran individual tests with Python's *unittest*
6. Created both a client and server that communicate with each other using UDP

Please note that the only graded part of this project is the last step (6). The rest is purely to help you set up your machine to work with our grading harness/write code. The grading harness isn't necessary to complete the project; it just makes it easier. It is more likely than not that your machine is already equipped with the technologies in any or all of these steps. **If you're confident that you're ready to go, skip to this section.**

This project is one of three in this class. They will all build on top of each other. **It's important that you finish this project in order to work on the next one.** (If you're not able to finish by the deadline, we do plan on releasing the reference solution to individuals that need it to start on the next projects.)

Also, let any instructor know if a certain part of this guide is confusing. This is the very first iteration of this *version* of the project—we'll be more than happy to update this document if necessary.

Notices

1. If you're using GitHub or a similar tool to track your work, please make sure that your repository is **set to private** until the due date. You're welcome to make your solution public after the matter.
2. As per the syllabus, the use of AI assisted tools is not allowed. Again, we can't prove that such tools were used, but we do reserve the right to ask a couple questions about your solution and/or add project related questions to the exam.
3. In the provided autograding Docker container, there are reference binaries. Please do not reverse engineer the binaries and submit that code—which would be obvious and clear academic dishonesty. Remember, we do manually inspect your code.

Setting up WSL (Windows Only)

WSL stands for Windows Subsystem for Linux—this essentially creates a Linux virtual machine that's deeply integrated into your Windows system. If you're on Windows, **please use WSL**. We've had students who used MinGW and other runtimes—they had a much harder time developing their projects.

These steps were verified to work on Windows 11 x64 23H2. They may work on previous versions of Windows (down to Windows 10), but it's not guaranteed. Please seek help in office hours if you get stuck.

Here are the steps to get WSL set up:

1. Go to the Microsoft Store and search for Ubuntu (feel free to install any distro, but we've just tested with Ubuntu). Install it.
2. Go to Control Panel (Win + R, type `control`, hit enter) > Programs and Features > Update Windows Features. Select Windows Subsystem for Linux.
3. Reboot your computer.
4. Go to your command prompt and enter the following commands:
   ```
   wsl.exe --update
   wsl.exe --install Ubuntu
   ```
5. Reboot your computer.
6. Go to your command prompt and enter `ubuntu`. Follow the prompts to set up your UNIX username and password.

7. Once you're in your Ubuntu install, run the following commands:
   ```
   sudo apt update
   sudo apt install -y make
   ```

You've now got a working WSL install!

Setting up Command Line Tools + Extras (macOS Only)

macOS uses the Darwin kernel—which is based on BSD[1] (a cousin of Linux). As such, a lot of APIs are similar—which makes macOS a fairly compatible development machine right out of the box. However, there are a couple of other tools you need to install first.

1. Open Terminal.
2. Install the Command Line Tools by entering `xcode-select --install`. Follow the instructions.
3. Install Homebrew, a package manager for macOS. Follow the instructions at https://brew.sh.

Installing Docker

How do we replicate the exact autograding environment? We use Docker, which is a form of containerization software that creates isolated environments according to certain specifications. Feel free to read more about it here. All you really need to know is that it's very similar to a virtual machine.

Let's install Docker on your system. Head over to https://www.docker.com and click on the install button for your machine. Follow the installation instructions. (**On Windows, make sure the WSL 2 option is selected.** If you forgot to do this/you already have Docker Desktop installed, visit this page for more information.)

Installing OpenSSL 3

In Project 2, you'll be using `libcrypto` from OpenSSL 3 to write a makeshift TLS-like layer to add security to this quarter-long project. We won't be doing that now, but we'd like to make sure you've got it installed beforehand.

If you haven't already, on WSL (or any Debian based system), run `sudo apt install -y libssl-dev`.

On macOS, run:
`brew install openssl`

[1] Saying Darwin is a kernel that's based on BSD is a bit of an oversimplification and honestly kind of inaccurate. Check out this Wikipedia article for more information.

`ln -s /usr/local/Cellar/openssl@3/3.3.2/include/openssl/ /usr/local/include/openssl` (if it says that the file already exists, that's fine. You can ignore it.)

Installing Visual Studio Code

As said in the overview, the exact IDE you use really isn't that important. However, we'll provide instructions here as official support (Visual Studio Code (VSC) also has great WSL support for Windows users).

Head over to https://code.visualstudio.com to download and install the appropriate version of VSC. Follow the instructions and any prompts.

Launch VSC. Follow these instructions if you're on WSL:

1. On first launch, VSC will recommend installing the WSL extension. (If the prompt doesn't show up, install it in the Extensions tab.) Follow the prompts to install it and restart VSC.
2. In the bottom, left-hand corner of your window, there should be a colored remote connect button. Select it and click on Connect to WSL.
3. This will open up a new window that's connected to your WSL installation! You'll have to repeat step 2 any time you open up a new VSC workspace.

In your VSC terminal (open it up by pressing Ctrl + `. Note that it's the same for macOS), clone our **Starter Code** using Git. Launch the repo in a new workspace by running `code project0starter`.

Now, follow these steps to set up code completion:

1. Select the file `project > client.c` in the left hand side navigator. VSC will prompt you to install the C/C++ extension. (If it doesn't prompt you, check in the Extensions tab.)
2. Once it's done installing, restart VSC.
3. Select the same file. There might be a warning in the bottom right corner to install a new compiler. Follow those steps and restart VSC.
4. Add the directive `#include <openssl/evp.h>`. If there's a red squiggly line under it, you don't have OpenSSL in your include path.
    a. VSC may prompt you to fix this issue. Follow its suggestion.
    b. If you're on macOS, try adding `/usr/local/include/openssl` and `/usr/local/Cellar/openssl@3/3.3.2/include/openssl` to your include path.
    c. You may add paths to the include path by going to the Extension Settings for the C/C++ extension.
    d. Please visit office hours if you're still unable to fix this issue.
5. Remove `#include <openssl/evp.h>`. We're just using this line to test your include path.

Exploring the Grading Harness + Unit Tests

Now that your environment has been set up, we can now demonstrate how to use the local autograding harness. Make sure Docker Desktop is running. If you haven't already, clone the **Starter Code** onto your machine.

Autograding

Run `make run`. This will pull the Docker image from Docker Hub, build your submission (in the `project` directory), and run the autograding code.

After it's completed, you'll see JSON formatted output of your results. You should be failing all the test cases since there's only the starter code. See Autograder Test Cases for more info. This is also available in `results > results.json`

You may also see the output of `stderr` for each run. Each file in the `project` directory will have the test case name (example: `test_self_ascii`) appended with either `refserver`, `refclient`, `yourserver`, and `yourclient`.

Custom Testing

Run `make interactive`. This will drop you into the autograding harness's shell.

In the `/autograder` directory, there are three important subdirectories.

One of them is `/autograder/`**`source`**. This is where all the autograding logic resides. If you want to run individual tests, you may:

1. Go to the `/autograder/source/tests` directory.
2. Run the compilation test case first: `python3 -m unittest test_0_compilation`
3. Run `python3 -m unittest <file_name>.<class_name>.<test_name>`
   For example, to run the first data transport test case:
   `python3 -m unittest test_1_reliable_data_transport.TestRDT.test_self_ascii`

Another is `/autograder/`**`submission`**. This is linked directly to the `project` directory on your local machine (any file you change in `project` changes in `submission` in the autograder and vice-versa).

The last one is `/autograder/`**`results`**. Just like `submission`, it's linked to the `results` directory on your local machine.

Client and Server over UDP[2]

Great! We're now all set up to start on your first CS 118 project! If you haven't already, clone the **Starter Code** onto your machine. These **Socket Programming Tips** are a great way to learn how to use BSD sockets.

Motivation

You may be familiar with the program `netcat`. Essentially, this program takes input from standard input and sends it to a network host of your choosing. It will also listen for input from said network host and output it on standard output.

Here's an example of two hosts communicating with `netcat`:

Host 1 (Client)

```
host1:~/project0 $ nc -4u localhost
8080
> Message 1


Message 2
> Message 3
> Message 4
> George Varghese



Lixia Zhang
Songwu Lu
```

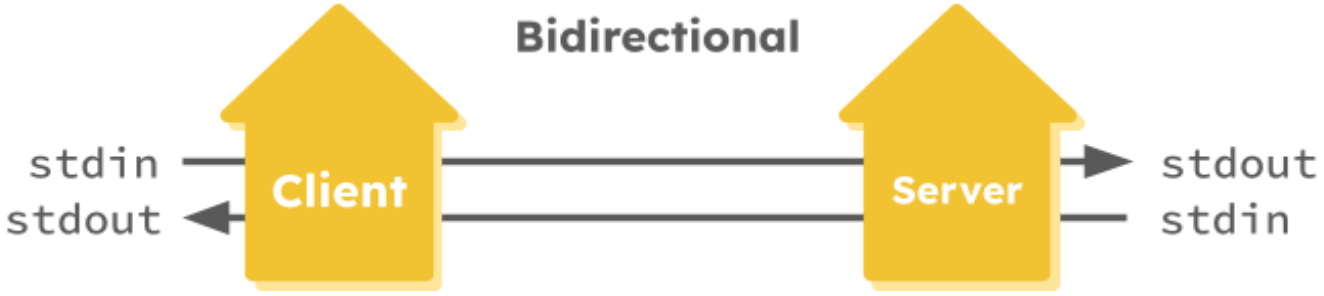
Host 2 (Server)

```
host2:~/project0 $ nc -4ul 8080


Message 1
> Message 2



Message 3
Message 4
George Varghese
> Lixia Zhang
> Songwu Lu
```

(Lines starting with > represent input into standard input (what you type). The extra lines between the messages are to show which messages happen in what sequence. These both don't actually show in the real prompt.)

Here's a diagram that (hopefully) makes it a bit more clear as to what these programs are doing.

[2] If you're familiar with UDP, you might be wondering why we're using it instead of TCP (since UDP has no reliability guarantees). Since we're only going to test on small files, it shouldn't really matter. The next project requires you to add this reliability.



Your goal is to create two programs (aptly named client and server) that do the exact same thing as the example above:

Host 1 (Client)

```
host1:~/project0 $ ./client localhost
8080
> Message 1


Message 2
```

Host 2 (Server)

```
host2:~/project0 $ ./server 8080


Message 1
> Message 2
```

Specification

Create two programs written in C/C++ (up to version 17). Both programs will use BSD sockets using IPv4 and UDP to listen/connect. While you may make both programs multi-threaded, it's not recommended. Use **non-blocking file descriptors** to help. **See Socket Programming Tips for more details**. No other third party libraries are allowed.

Server

The first program, `server`, takes one argument: the port number. The usage of `server` is as follows:

```
./server <port>
```

`server` will listen on the given port until a client establishes a UDP connection. We know a connection has been established as soon as the client has sent over data. **The server does not have to accept any other connections.** Once this connection is established, it will:

1. start reading from the socket and output its contents to standard output.
2. At the same time, it will start reading from standard input and forwarding its contents to the newly connected client.

Even though `server` is doing two things at the same time, it's possible to keep it single threaded. All you have to do is make both standard input and the socket **non-blocking**.

Here's some example pseudocode for `server`:

```
Set up socket
Make socket non-blocking
Make standard input non-blocking

Infinite loop:
      recvfrom client
      if no data has been received and client has not connected:
            continue

      // Now, the client has connected (data has been sent at some point)
      Set client connected to true

      if data has been received:
            write to standard out

      read from standard in
      if data available from standard in:
            sendto client
```

Client

The second program, `client`, takes two arguments: the hostname and the port number. The usage of `client` is as follows:

```
./client <hostname> <port>
```

Note: While we ask you to accept a hostname, we'll only test you on `localhost`. In short, your logic can literally check if the second argument is "`localhost`" and use the IP "`127.0.0.1`"—otherwise, pass in the second argument verbatim to `inet_addr`.

`client` will connect to the server over a UDP connection. **The client can always assume that the server has already started and is immediately ready for sending/receiving. We will always test it such that the server has started much before the client.** Immediately, it will:

1. start reading from the socket and output its contents to standard output.
2. At the same time, it will start reading from standard input and forwarding its contents to the server.

Just like `server`, even though `client` is doing two things at the same time, it's possible to keep it single threaded. All you have to do is make both standard input and the socket **non-blocking**.

Here's some example pseudocode for `client`:

```
Set up socket
Make socket non-blocking
Make standard input non-blocking

Infinite loop:
      recvfrom server

      if data has been received:
            write to standard out

      read from standard in
      if data available from standard in:
            sendto server
```

Note that the code for the client is very similar to the server, but it doesn't worry about whether or not it's connected.

Potential Improvements

To make your life easier before you attempt Project 1, try to find a way to abstract your "listen loop" code into another file. After all, both the client and server use very similar code.

Common Problems (list will be frequently updated)

- Don't ever use `printf`. Just write directly to standard output.
- If you want to write out debugging messages, use `fprintf` to `stderr`.
- Test on ports >=1024. Under that requires root access.

Autograder Test Cases

0. Compilation (`class TestCompilation`)

This test case passes if:

1. Your code compiles,
2. yields two executables named `server` and `client`,
3. and has no files that aren't source code.

1. Data Transport (`class TestRDT`)

1. **Data Transport (Your Client <-> Your Server): Small, ASCII only file (2 KB): `test_self_ascii` (25 points)**
   This test case runs your server executable then your client executable. It will then input 2 KB of random data (ASCII only) in both programs' stdin and check if the output on the respective other side matches.

2. **Data Transport (Your Client <-> Your Server): Small file (2 KB): `test_self` (25 points)**
   Same as 1, but could be any sort of data (just completely random bytes).

3. **Data Transport (Your Client <-> Reference Server): Small file (2 KB): `test_client_normal` (25 points)**
   Same as 1, but uses the reference server instead.

4. **Data Transport (Reference Client <-> Your Server): Small file (2 KB): `test_server_normal` (25 points)**
   Same as 1, but uses the reference client instead.

Submission Requirements

Submit a ZIP file, GitHub repo, or Bitbucket repo to Gradescope by the posted deadline. You have unlimited submissions; we'll choose your best scoring one. As per the syllabus, remember that all code is subject to manual inspection.

In your ZIP file, include all your source code + a Makefile that generates two executables named `server` and `client`. **Do not include any other files; the autograder will reject submissions with those.** If you're using the starter repository, the default Makefile inside the `project` directory includes a `zip` directive that automatically creates a compatible ZIP file.