CS341 Network Lab 1: Basic Socket Programming

Logistics

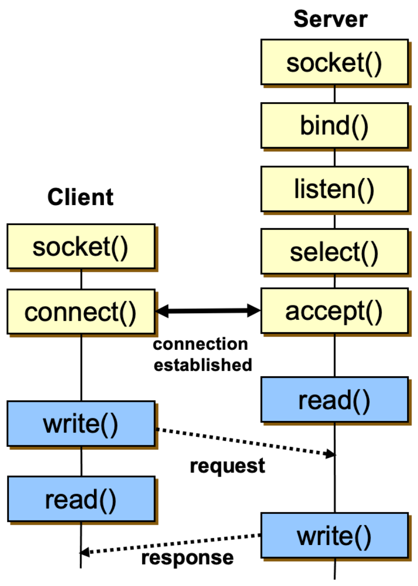
- The due date is 11:59pm (KST) on September 19th, 2024
- Submit `EchoAssignment.cpp` via KLMS gradescope.
- Q&A: Refer to the Discussion and Q&A section of this document.

Overview

In Lab 1, you will implement a variant of a simple echo server and a client. An echo server uses TCP via POSIX network sockets to receive and handle requests from its clients. Clients issue requests to servers and receive their responses. The purpose of this project is to learn how to use the POSIX network socket.

POSIX Socket Programming

The Linux socket networking layer provides BSD socket functions, and they are the user interface between the user process and the network protocol stacks in the kernel. The BSD socket functions contain `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `read()`, `write()`, `getsockname()`, `getpeername()`, and `close()`. In this project you learn to use these functions. The figure below shows the overall control flow of the socket functions.



`socket()`

It creates a new socket and returns its socket descriptor.
[IMPORTANT] KENS only supports `socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)`. Other parameters will not work.

`bind()`

It associates a socket with a local port number and an IP address.
Try think about following:
What does it mean by binding to 0.0.0.0?

`listen()`

It prepares a socket for incoming connections.

`accept()`

It accepts a received incoming attempt from a client. It creates a new socket associated with a new TCP connection.

`connect(sockfd, addr, addrlen)`

It binds the address specified by `addr` to the socket referred to by the file descriptor `sockfd`.

`read()` / `write()`

These functions are used for data transfer using a socket.

`getsockname()`

It returns the current address to which a socket is bound.

`getpeername()`

It returns the address of the peer connected to a socket.

`close()`

It closes the connection.

You should refer to the Linux manual pages for the detailed instructions on the socket functions. You may want to use `inet_addr` or `inet_ntop`.

Getting Started

To set up KENS, please follow instructions in the KENS github wiki.
Depending on your environment, follow proper Getting-Started page:

- For Linux users, follow Getting-Started: CLI (Linux)
- For macOS users, follow Getting Started: CLI (macOS)
- For Windows users choose from followings:
    - Install WSL2, then follow Linux setup
    - Install Docker, then follow Getting Started: Docker

Note that M1 Docker is not supported.

Some images or documents could be outdated; for example, we do not use the `kens-part-x` format now.

The only file you should edit is `EchoAssignment.cpp`

Simple Echo Server

An echo server is a very simple application that receives requests and sends back the requests as is. In Lab 1, you will implement a variant of the simple echo application. Instead of simply sending back what it received, the server will process three requests differently and compose replies that are not the same as the request. The server should repeat following indefinitely:

1. The server application accepts a new connection: `accept()`.
2. It receives a request terminated by a new line character (`\n`).
3. The server responds differently depending on the request.
    a. if request := `hello\n`, response := `server-hello(server-hello variable in the skeleton code)\n`
    b. if request := `whoami\n`, response := client's IP address`\n`
    c. if request := `whoru\n`, response := server's IP address`\n`
    d. for all other requests, the response is identical to the request.
4. All responses must terminate with a new line character (`\n`).
5. Server logs (or submits) requests via submitAnswer method.
    a. For all requests, the server must call the `submitAnswer` method with a client's IP address and the request's data.
    b. e.g.) `submitAnswer(client_ip, content);`
    c. Note that both IP and data must be null terminated C strings (`\0`) not new line terminated strings (`\n`).

Server is designed to run indefinitely, accepting clients' connection forever. You do not need to worry about stopping them; KENS testing system handles it. If you want to gracefully handle the end of the KENS test, you could add logic to stop the server when `accept()` fails, returning -1. Usually you do not need to worry about how the function ends.

Skeleton Code

All of your server source code must lie in the `serverMain` method (see comments in the skeleton code). The first and second parameters are used for the `listen()` call. The third parameter (`const char *server_hello`) is used for `server-hello` messages.

```
int EchoAssignment::serverMain(const char *bind_ip, int port,
                               const char *server_hello) {
    // Your server code
    // !IMPORTANT: do not use global variables and do not define/use
functions
    return 0;
}
```

Client

You will also implement a simple client application for the echo server. It connects to the server and sends a request. It also uses the `submitAnswer` method to log the server's responses. The client's control flow is as below.

1. A client connects to an echo server: use `connect()`
2. Client sends a request to the server.
3. Client receives a response from the server.
4. Client logs (or submits) the response via `submitAnswer` method.
    a. For all the responses, the client must call the `submitAnswer` method with a server's IP address and the reponse's data.
    b. See the server's instructions for more detail about the `submitAnswer` method.

Skeleton Code

All of your client source code must lie in the `clientMain` method (see comments in the skeleton code). The first and second parameters are used for the `connect()` call. The third parameter (`const char *command`) is sent to the echo server.

```
int EchoAssignment::clientMain(const char *server_ip, int port,
                               const char *command) {
    // Your client code
    // !IMPORTANT: do not use global variables and do not define/use
functions
    return 0;
}
```

Build

Build instructions are described in the Getting Started pages:
https://github.com/ANLAB-KAIST/KENSv3/wiki.
After build completes, two binaries are produced in `build/app/echo/` directory: `echo` and `echo-non-kens`.
For GCC version > 11, you might face `cannot download kens_solution` error. This happens because we have not prepared KENS solution binaries for GCC version > 11. Please refer to https://github.com/ANLAB-KAIST/KENSv3/discussions/94 for fix.

Test

To run test cases, execute the `echo` binary. We will use this result for grading; there are no hidden cases. The `echo-non-kens` is your echo server and client application without KENS. You can use this binary for testing with a real environment (we will not use this binary for grading).

Example Test Output:

```
[==========] Running 15 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 15 tests from EchoTesting
[ RUN      ] EchoTesting.SingleEcho
[       OK ] EchoTesting.SingleEcho (12 ms)
[ RUN      ] EchoTesting.SingleWhoRU
[       OK ] EchoTesting.SingleWhoRU (6 ms)
[ RUN      ] EchoTesting.SingleWhoAmI
[       OK ] EchoTesting.SingleWhoAmI (4 ms)
[ RUN      ] EchoTesting.SingleHello
[       OK ] EchoTesting.SingleHello (2 ms)
[ RUN      ] EchoTesting.OnetoManyEcho
[       OK ] EchoTesting.OnetoManyEcho (39 ms)
[ RUN      ] EchoTesting.OnetoManyWhoRU
[       OK ] EchoTesting.OnetoManyWhoRU (37 ms)
[ RUN      ] EchoTesting.OnetoManyWhoAmI
[       OK ] EchoTesting.OnetoManyWhoAmI (38 ms)
[ RUN      ] EchoTesting.OnetoManyHello
[       OK ] EchoTesting.OnetoManyHello (39 ms)
[ RUN      ] EchoTesting.OnetoManyAll
[       OK ] EchoTesting.OnetoManyAll (153 ms)
[ RUN      ] EchoTesting.ManytoOneEcho
[       OK ] EchoTesting.ManytoOneEcho (12 ms)
[ RUN      ] EchoTesting.ManytoOneWhoRU
[       OK ] EchoTesting.ManytoOneWhoRU (11 ms)
[ RUN      ] EchoTesting.ManytoOneWhoAmI
[       OK ] EchoTesting.ManytoOneWhoAmI (13 ms)
[ RUN      ] EchoTesting.ManytoOneHello
[       OK ] EchoTesting.ManytoOneHello (11 ms)
[ RUN      ] EchoTesting.ManytoOneAll
[       OK ] EchoTesting.ManytoOneAll (43 ms)
[ RUN      ] EchoTesting.ManytoMany
[       OK ] EchoTesting.ManytoMany (242 ms)
[----------] 15 tests from EchoTesting (668 ms total)

[----------] Global test environment tear-down
[==========] 15 tests from 1 test suite ran. (668 ms total)
[  PASSED  ] 15 tests.
```

Usage of echo-non-kens:

```
Usage: ./echo-non-kens <mode> <ip-address> <port-number>
<command/server-hello>
Modes:
    c: client
    s: server
```

```
Client commands:
    hello : server returns <server-hello>
    whoami: server returns <client-ip>
    whoru : server returns <server-ip>
    others: server echos
Note: each command is terminated by newline character (\n)
Examples:
    server: ./echo-non-kens s 0.0.0.0 9000 hello-client
    client: ./echo-non-kens c 127.0.0.1 9000 whoami
```

Frequently Asked Questions

You can browse previously discussed Q&As from the legacy github discussions:
https://github.com/ANLAB-KAIST/KENSv3/discussions/categories/-project-q-a-1-application-layer-socket

I have no idea what I am supposed to do

Search for `socket` API examples. There are lots of guides and how-to about `socket` APIs.
Learn how servers and clients interact, how to use `socket` API, then start the project.
You may use following 4-steps:

1. Try looking at `app/kens/testhandshake.cpp` to start understanding the conventions and how KENS wants you to code.
2. Use Google to find websites for building a backbone for your code.
3. Once done with the backbone, start debugging with help from the linux official manpage and previous discussions (Refer to Discussions and Q&A section)
    a. Build and run `./echo` until you get at least 1 test case down
    b. When stuck, customize `./echo-non-kens` parameters to check if your code is working on your local machine
    c. If your code only works on your local machine, check the github board for answers.
4. Repeat step 3 until ./echo shows all pass

Following three small Q&A might also help:

All socket programming examples from the Internet show that the server and client are to be done on separate files. May I have some clarification on this?

Usually, socket programming guides write server and client on different files for easier understanding: two binaries, two functions, two files. There is no problem to write two functions in a single file: `EchoAssignment.cpp`.

Where exactly are we supposed to save the `EchoAssignment.cpp` file in our directory such that we can proceed with the code writing and testing?

Do not replace nor move any files, just edit `EchoAssignment.cpp` in `/app/echo/`. After you edit `EchoAssignment.cpp`, try building binaries again, and test it by executing binaries to check if your implementation is correctly working.

What are we supposed to see as a success test case?

When you correctly implement your server and client code and run the binary `/build/app/echo/echo`, you will see messages described at `Example Test Output:` in our project document (number of tests and execution may be different). When correctly implemented, there should be no `FAIL` or similar messages, and the last line should say `[ PASSED ] X tests`.

How does this system work?

KENS is a network simulator. For this assignment, server and client run on the KENS TCP stack, communicating with each other. `echo` binary will test your server and client implementation by connecting them, providing proper arguments.

Do you provide virtual machines for this lab?

No. KENS supports various environments, so that you can do this lab on your machine.

Are there hidden test cases?

No. Result of `./app/echo/echo` will be your grade.

Can I get help from ChatGPT, or other LLMs?

Yes, **as long as you are getting "help"** from them. You cannot ask LLMs to do your entire lab. Copy-pasting codes from ChatGPT and other LLMs is **prohibited**, regarded as cheating. Note that TAs can access ChatGPT, generate codes, and compare them with yours.

How can I use `echo-non-kens`?

`echo-non-kens` executes your client or server in real network stack. It will print out your `submitAnswer` parameters. Check if you are correctly submitting answers.

Should the server run in an infinite loop?

Yes. Servers should try accepting all incoming connections. You do not need to worry about how to stop the server, the KENS testing system handles it. You may make the server gracefully stop when accepting a new connection fails.

`accept()` returns negative value

This can happen while the KENS testing system kills the server while ending a test, so accept() ends with error. If all tests pass, there is no problem.

Am I OK to use this function? What does it mean by "do not define/use functions"?

Do not include additional libraries, and use only functions that are supported by the Ubuntu environment. Please **avoid** functions that are **not** supported by Ubuntu, for example, `strlcpy()` function and `sin_len` field, which are **only** supported in BSD environments.
You have following pre-included modules, so check if all functions you use are included in them:

- Defined at `app/echo/EchoAssignment.cpp`
    - `<cstdio>`
    - `<cstdlib>`
    - `<cstring>`
    - `<arpa/inet.h>`
- Defined at `app/echo/EchoAssignment.hpp`
    - `<sys/socket.h>`
        - This will be replaced with KENS socket API while testing, such as socket(), and bind()
    - `<unistd.h>`
        - This will be replaced with KENS system call API while testing, such as read(), and write()
    - `<map>`
    - `<string>`
    - `<vector>`

Some unexpected behaviors (`` Assertion `nsIter != appIter->second.fdToDomain.end()' failed. ``, etc)

This might happen due to the KENS bug, which happens while allocating large stack variables. For example, a `char` array larger than 20 bytes might trigger the bug. In some cases, the following error message shows up: `` Assertion `nsIter != appIter->second.fdToDomain.end()' failed ``. Try allocating the variable from the heap, using `malloc` or `new` syntax.

How should I implement error handling/edge cases?

Most test cases do not test error handling nor edge cases. You do not worry about such cases; just try to run and check if your code passes.

`socket()` halts!

KENS only supports `socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)`. Other parameters will not work.

Is there a way to test a single test case?

There is no simple command to do so; however, you can comment out test case configuration code in `/app/echo/testecho.cpp`, which starts with `TEST_F`.

Testing takes too long time

Try followings:

- remove unintended loop
- remove printing out to console
- close connections (opened sockets) that are no more used

A function does not work as expected

Search for the exact behavior of the function. You can access the Linux manual page by using keyword: “<function name> man page” on Google, or Linux manual pages from KENS github. The manual page describes all details about the function. In most cases, you are giving inappropriate parameters to the function, or you are using the return value inappropriately. **Some functions reuse the same buffers** for storing results, so you need to copy the results after invoking the functions.

Cannot access Gradescope; Turnitin shows following error: App could not be opened. Launch type not supported. Please contact your administrator.

Sometimes KLMS fails to open Gradescope. In most cases, the problem persists for a very short time; try a bit later. If the problem persists, contact us.

Is there a penalty for making more than one submission to gradescope?

No. Just note that the last submission is used for grading.

Discussion and Q&A

Before asking questions, **please** try followings carefully:

1. For general socket related questions, search through the Internet (Google, ChatGPT, etc)
2. Browse Frequently Asked Questions section in this document
3. Browse previously discussed Q&As from the legacy github discussions
4. Browse previously discussed Q&As from our Classum board

If you are sure that your problem has never been discussed before, follow How to ask questions to make a new question, then post it to our Classum board.

We will **not** use KLMS Q&A board, KENS github discussion, e-mail, or others for accepting new questions. Use Classum for new questions.

Grading

We will use an automated testing environment that executes tests with `EchoAssignment.cpp` you submit. The result will be exactly the same as the test you can run, by executing `/app/echo/echo`. There is no hidden case, and we will not check your detailed implementation. Just in case, please consider testing your code in an Ubuntu 22.04 machine, which we will use for grading.

Submission

Submit your `EchoAssignment.cpp` to KLMS gradescope assignment, before 11:59 PM on September 12th. **DO NOT CHANGE THE FILENAME.**