# Programming Assignment #1: Basic Socket Programming

## Logistics

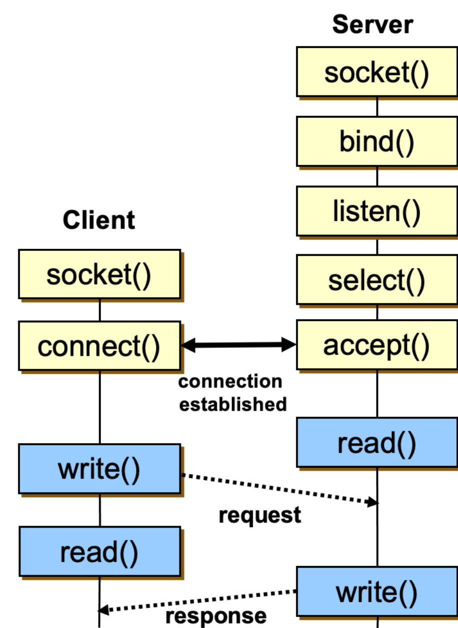
- The due date is 11:59pm on Tue, Mar. 29th.
- Submit `EchoAssignment.cpp` via KLMS per team.
- Discussion and Q&A: https://github.com/ANLAB-KAIST/KENSv3/discussions

## Overview

In PA #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 and to set up KENS for later assignments.

## 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 and in later projects learn to actually implement them. The figure below shows the overall control flow of the socket functions.




**socket()**
It creates a new socket and returns its socket descriptor.

**bind()**
It associates a socket with a local port number and an IP address.

**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:
https://github.com/ANLAB-KAIST/KENSv3/wiki/Misc:-External-Resources#linux-manuals .

## Getting Started

To set up KENS, please follow instructions in the following link:
https://github.com/ANLAB-KAIST/KENSv3/wiki .

## Simple Echo Server

**Update #1:** please use `socket(AF_INET, SOCK_STREAM, IPPROTO_TCP)` for KENS. The other parameters would not work.
**Update #2:** For your convenience, you may want to use `inet_addr` or `inet_ntop`.

An echo server is a very simple application that receives a request and sends back the request as is. In PA#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. Below is the description.

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`).

**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](https://github.com/ANLAB-KAIST/KENSv3/wiki) .

After build completes, two binaries are produced in `build/app/echo/` directory: `echo` and `echo-non-kens`.

## Test

To run test cases, executes the `echo` binary. We will use this result for grading. 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 9 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 9 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 (6 ms)
[ RUN      ] EchoTesting.SingleHello
[       OK ] EchoTesting.SingleHello (6 ms)
```

```
[ RUN      ] EchoTesting.OnetoManyEcho
[       OK ] EchoTesting.OnetoManyEcho (134 ms)
[ RUN      ] EchoTesting.OnetoManyWhoRU
[       OK ] EchoTesting.OnetoManyWhoRU (103 ms)
[ RUN      ] EchoTesting.OnetoManyWhoAmI
[       OK ] EchoTesting.OnetoManyWhoAmI (94 ms)
[ RUN      ] EchoTesting.OnetoManyHello
[       OK ] EchoTesting.OnetoManyHello (93 ms)
[ RUN      ] EchoTesting.OnetoManyAll
[       OK ] EchoTesting.OnetoManyAll (448 ms)
[----------] 9 tests from EchoTesting (902 ms total)

[----------] Global test environment tear-down
[==========] 9 tests from 1 test suite ran. (903 ms total)
[  PASSED  ] 9 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
```