Programming Assignment #2: Context Management and 3-way Handshaking in TCP

Logistics

- The due date is 1pm on Tue, April 12th.

In Programming Assignment #1, you have installed KENS and learned to do basic socket programming in C++ on top of KENS. On the client side, you have used the socket APIs in the following order: socket() - bind() - connect() - send()/recv() - close(). On the server side, socket() - bind() - listen() - accept() - send()/recv() - close().

From Chapter 3 in the textbook, you are learning what the transport layer does to provide reliable delivery to the application layer. In this programming assignment, you will implement the very basics of the transport layer; that is, not only use the socket APIs but implement what goes on below the socket API.

Below is the key reference to the inner workings of TCP/IP. For detailed information on the socket API, look up the corresponding manual pages.

[1] TCP/IP Illustrated, Volume 2, Gary R. Wright and W. Richard Stevens, 1995, Addison-Wesley.

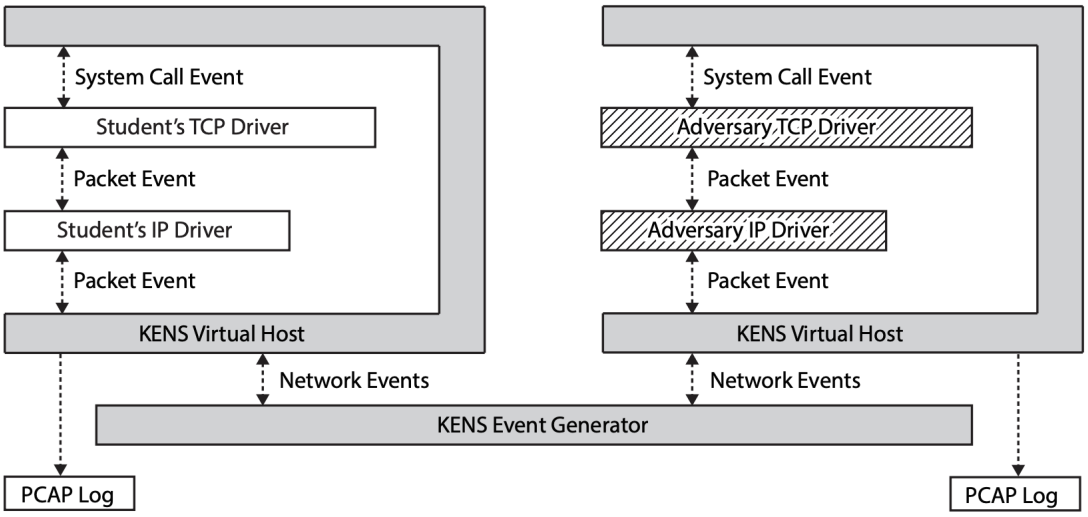
Brief Introduction to KENSv3

In most operating systems, the transport and lower layers are implemented in the kernel and are very hard to modify. A bug in the kernel will incur a system panic and you will have to reboot your system. Building and testing in userspace is far easier than in kernel, because restarting a process is far easier than rebooting the entire OS: voila, KENS.

We have developed a network simulator called KENS (KAIST Educational Network System). KENS is an event-driven network simulator that provides a virtual environment for students to build and test TCP and IP stacks. KENS provides the application layer as well as the IP and the layer below. Also included in KENS are reference binaries for the TCP and IP layers so that students can test their own against.

The overall KENS architecture is in the figure below. You only implement specific APIs in the given TCPAssignment.cpp file. You do not write your application code, IP layer code, nor the

counterpart (“adversary”). No need to even write main(). The only files you will modify in PA #2 and PA #3 are: **TCPAssignment.cpp** and **TCPAssignment.hpp**.

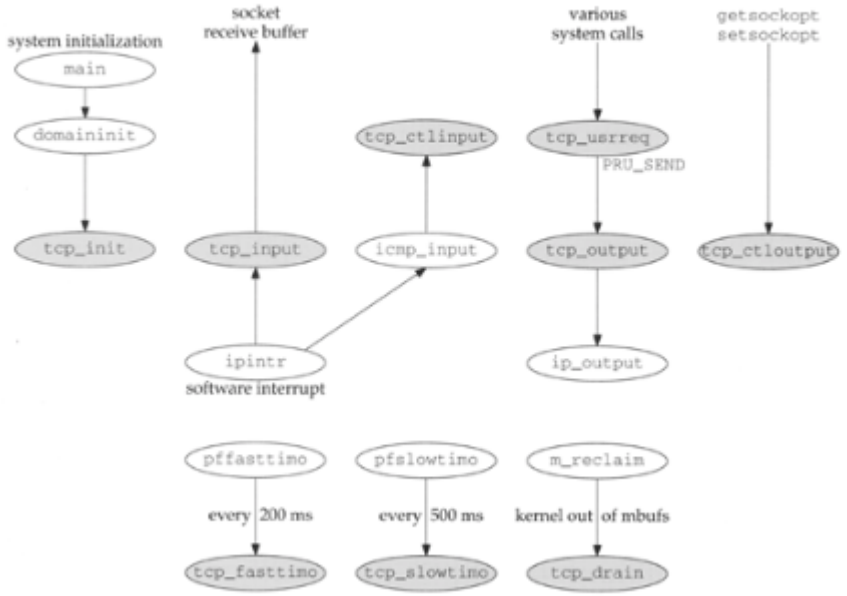


A First Step to Build Your Own TCP

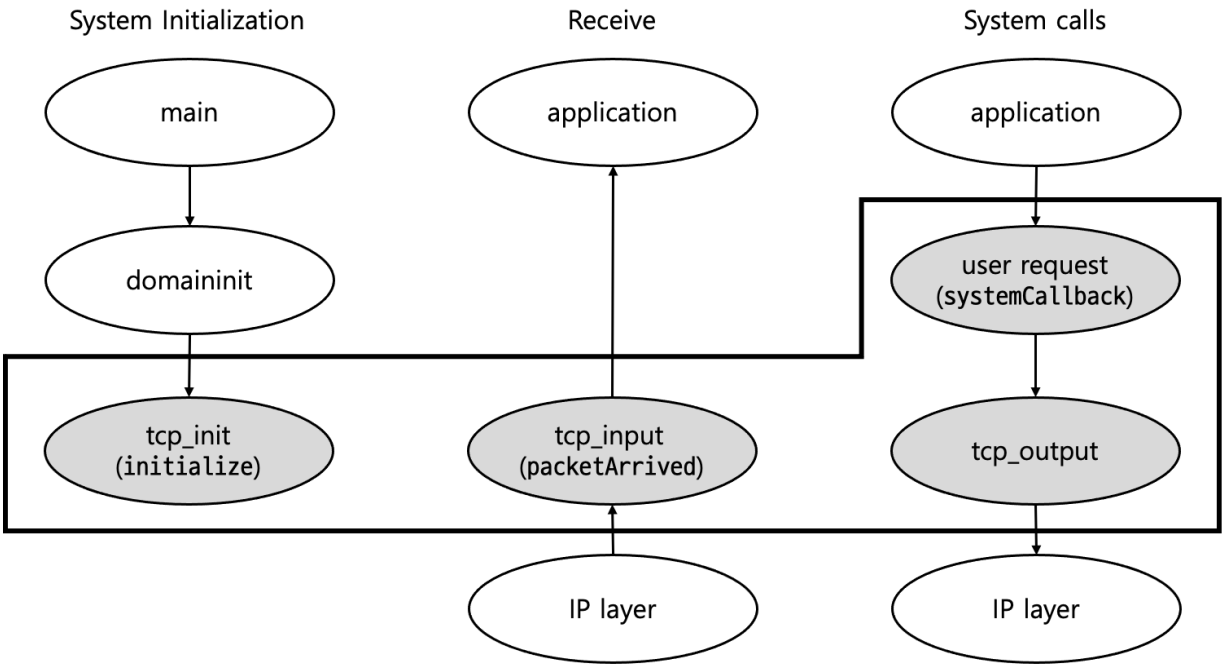
In PA #2 and PA #3 we are not building the complete TCP, but a bare minimum of the protocol. Yet any reference describes TCP in full. In order to help you understand the limited scope of PA #2, we will refer to the TCP implementation in [1] and present the scope you should implement in PA #2.

First, the TCP layer interacts with the application layer above and the IP layer below and its relationship is drawn in Figure 24.2 of [1].

Figure 24.2. Relationship of TCP functions to rest of the kernel.



In PA #2 you only need to implement only the following gray areas in KENS:



You will implement in `TCPAssignment::initialize()` all the code necessary to initialize data structures you will later use in socket call processing.

In an operating system, when a user mode process invokes a system call, the CPU switches to kernel mode and starts the execution of a kernel function. The execution is a jump to an assembly language function called the system call handler. The user mode process passes a parameter called the system call number for the kernel to identify the required system call handler. KENS provides a simulation framework that mimics this user-kernel mode switching. The system call handler in an operating system maps to a switch statement in the systemCallback method. The system call number is `param.syscallNumber` in the code below. We ask you to implement only the `TCPAssignment::systemCallback not` the entire mode switching of the operating system.

```
void TCPAssignment::systemCallback(UUID syscallUUID, int pid, const
SystemCallParameter &param) {//}
```

`UUID syscallUUID`	`Unique identifier allocated for every system call initiation`
`int pid`	`process id`
`const SystemCallParameter &param`	`system call parameter, including the system call number. see` https://github.com/ANLAB-KAIST/KENSv3/blob/master/include/E/Networking/E_Host.hpp `for details.`

The `TCPAssignment::systemCallback` method does not have a return value. Those system calls that may not return immediately, such as `accept()`, need to be blocked until ready. You should store the corresponding `UUID` to return it later. For all system calls to return a value, you must use a return`SystemCall` method.

```
void TCPAssignment::systemCallback(UUID syscallUUID, int pid,
                                   const SystemCallParameter &param)
{
    this->returnSystemCall(syscallUUID, -ENOSYS);
    // returns -ENOSYS (Function not implemented) from system call
`syscallUUID`
}
```

When a packet arrives from the IP layer to the TCP layer in KENS, the packet must be processed in `TCPAssignment::packetArrived`:

```
void TCPAssignment::packetArrived(std::string fromModule, Packet
&&packet) {
  (void)from Module;
  (void)packet;
}
```

The tcp_output module in Figure 24.2 [1] prepares actual packets from user data. It must fill the header fields and payload. KENS provides methods to compute the checksum and obtain the source IP address in the header. KENS even provides a utility function to convert between integer arrays. Refer to the following link for further details.

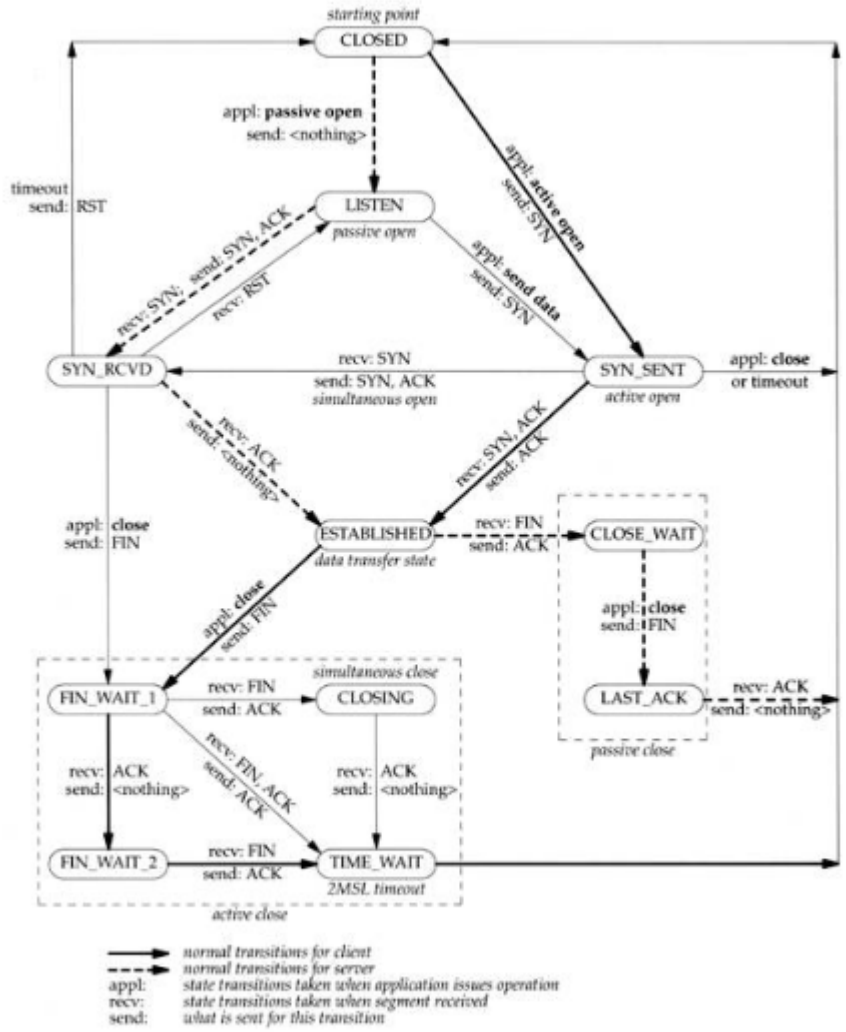
- Figuring out the source IP address
- Networking Utilities

Once a packet is ready, the tcp_output module must call `sendPacket()`. You will find details on `sendPacket()` in the following link.

- Sending and Receiving a Packet

Within `TCPAssignment::packetArrived`, `TCPAssignment::systemCallback()`, and tcp_output, you will implement the following state transitions.

Figure 24.15. TCP state transition diagram.



Basics: socket(), open(), getsockname()

As you have programmed in PA #1, the POSIX APIs for socket() and bind() are as follows:

```
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
```

```
#include <sys/types.h>          /* See NOTES */
```

```
#include <sys/socket.h>

int bind(int sockfd, const struct sockaddr *addr,
         socklen_t addrlen);
```

```
#include <sys/socket.h>
int getsockname(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

Corresponding APIs in KENS would be implemented in `systemCallback`:

```
void TCPAssignment::systemCallback(UUID syscallUUID, int pid,
                                   const SystemCallParameter &param) {


  switch (param.syscallNumber) {
  case SOCKET:
    // this->syscall_socket(syscallUUID, pid, param.param1_int,
    // param.param2_int, param.param3_int);
    break;

...

  case BIND:
    // this->syscall_bind(syscallUUID, pid, param.param1_int,
    //            static_cast<struct sockaddr *>(param.param2_ptr),
    //            (socklen_t) param.param3_int);
    break;

...

  case GETSOCKNAME:
    // this->syscall_getsockname(syscallUUID, pid, param.param1_int,
    //            static_cast<struct sockaddr *>(param.param2_ptr),
    //            static_cast<socklen_t*>(param.param3_ptr));
    break;

...

 }
}
```

This assignment is to build the bare minimum of the transport layer, namely create data structures, assign parameter values to the correct data structure fields, and return correct values. The testing code will call those APIs and check to see the return values are correct. It is not easy to check if the internals of your code are correct. They will be tested more rigorously in PA #3. So we recommend you to take a look at the test code in PA #3 in advance and design your data structures accordingly.

Socket Function

```
  case SOCKET:
    // this->syscall_socket(syscallUUID, pid, param.param1_int,
    // param.param2_int, param.param3_int);
    break;
```

The socket() call receives 3 parameters from the application layer. Now it should create a file descriptor and store the domain and the protocol in the data structure indexed by the file descriptor. It returns the file descriptor. More details about the socket call are described at: https://linux.die.net/man/2/socket and https://linux.die.net/man/3/socket. In KENS, you need to implement only domain AF_INET, type SOCK_STREAM, and protocol IPPROTO_TCP.

The testing code in .app/kens/testopen.cpp calls socket() many times and checks if the return values are correct.

Bind Function

```
  case BIND:
    // this->syscall_bind(syscallUUID, pid, param.param1_int,
    //            static_cast<struct sockaddr *>(param.param2_ptr),
    //            (socklen_t) param.param3_int);
    break;
```

The bind() call receives 3 parameters from the application layer. Now it should assign an address to the socket. More details about the socket call are described https://linux.die.net/man/2/bind and https://linux.die.net/man/3/bind .
In KENS, you need to implement only sockaddr_in type for sockaddr.

```
struct sockaddr_in {
    sa_family_t    sin_family; /* address family: AF_INET */
    in_port_t      sin_port;   /* port in network byte order */
    struct in_addr sin_addr;   /* internet address */
};
```

```
/* Internet address. */
struct in_addr {
    uint32_t       s_addr;     /* address in network byte order */
};
```

The only value you should assign to sin_family is AF_INET. The two fields, sin_port and sin_addr, must follow the network byte order. The sin_addr field must be either an IP address or INADDR_ANY. You should implement both cases.

When an IP address and a port number are bound to a socket, they should not overlap with IP addresses and port numbers that are already bound to other sockets. The client side does not call bind() – a random port number is assigned upon connect(). Examples are:

- 143.248.234.2:5555 and 143.248.234.3:5555 (IP addresses differ; OK)
- 143.248.234.2:5555 and 143.248.234.2:5556 (Port numbers differ; OK)
- 143.248.234.2:5555 and 0.0.0.0:5555 (0.0.0.0 in case of INADDR_ANY; not OK)
- No need to check against closed sockets

GetSockName Function

```
  case GETSOCKNAME:
    // this->syscall_getsockname(syscallUUID, pid, param.param1_int,
    //             static_cast<struct sockaddr *>(param.param2_ptr),
    //             static_cast<socklen_t*>(param.param3_ptr));
    break;
```

The getsockname() call receives 3 parameters from the application layer. It should return the current address to which the socket is bound. More details about the socket call are described https://linux.die.net/man/2/getsockname and https://linux.die.net/man/3/getsockname. As in the case of bind(), you need to implement only the sockaddr_in type for sockaddr.

Every time you create a socket, you create a file descriptor and have to manage file descriptors per process. KENS provides the file descriptor management methods in `class SystemCallInterface`. The parameter processID to `createFileDescriptor` specifies the process associated with the file descriptor. You can create a new file descriptor using `createFileDescriptor`. This method allocates a new file descriptor in a POSIX-compliant way and returns it. Also, you can remove a specific file descriptor using `removeFileDescriptor`.

3-way Handshake

As we are yet to cover in the lectures, TCP connection setup is done via 3-way handshake. When the client side initiates the connection setup by calling connect(),

Connect Function

```
case CONNECT:
  // this->syscall_connect(syscallUUID, pid, param.param1_int,
  //            static_cast<struct sockaddr*>(param.param2_ptr),
  //           (socklen_t)param.param3_int);
  break;
```

The connect() call receives 3 parameters from the application layer. It connects the file descriptor to the address specified by addr. More details about the socket call are described https://linux.die.net/man/2/connect and https://linux.die.net/man/3/connect.

Listen Function

```
case LISTEN:
  // this->syscall_listen(syscallUUID, pid, param.param1_int,
  // param.param2_int);
  break;
```

The listen() call receives 2 parameters from the application layer. It marks the socket as a passive socket, that is, as a socket that will be used to accept incoming connection requests using accept. KENS requires you to implement the backlog parameter. It defines the maximum length to which the queue of pending connections for sockfd may grow. More details about the socket call are described https://linux.die.net/man/2/listen and https://linux.die.net/man/3/listen.

Accept Function

```
case ACCEPT:
  // this->syscall_accept(syscallUUID, pid, param.param1_int,
  //            static_cast<struct sockaddr*>(param.param2_ptr),
  //            static_cast<socklen_t*>(param.param3_ptr));
  break;
```

The accept() call receives 3 parameters from the application layer. It extracts the first connection on the queue of pending connections. It creates and returns a new file descriptor

for the connection. It also fills the address parameter with connecting client's information. More details about the socket call are described https://linux.die.net/man/2/accept and https://linux.die.net/man/3/accept .

Close Function

```
case CLOSE:
  // this->syscall_close(syscallUUID, pid, param.param1_int);
  break;
```

The close() call receives a parameter from the application layer. It closes the file descriptor's connection and deallocates the file descriptor. More details about the socket call are described https://linux.die.net/man/2/close and https://linux.die.net/man/3/close.

Using connect() or close(), you need a timer in order not to wait indefinitely. Check out the following link to use timers.

- Using Timer

More Resources

https://github.com/ANLAB-KAIST/KENSv3/wiki/Misc:-External-Resources

Submission

The test code is in the following files. Your code will be graded according to the test results from them (**only reliable test cases**).

- ./app/kens/testopen.cpp (`test-kens-open`)
- ./app/kens/testbind.cpp (`test-kens-bind`)
- ./app/kens/testhandshake.cpp (`test-kens-handshake`)
- ./app/kens/testclose.cpp (`test-kens-close`)

You should submit only three files: **readme.txt**, **TCPAssignment.cpp**, **TCPAssignment.hpp**. TCPAssignment.cpp and TCPAssignment.hpp should contain your implementation. Upload the files on KLMS. There is no designated template for readme.txt; just briefly explain what you have implemented and how you have progressed to complete this assignment. It does not have to be long and detailed. A brief summary will suffice.