Programming Assignment #2: Basics of KENSv3

Logistics

- Each team has to submit 3 questions up on the discussion board by **Sunday (3/28)**. All questions, answers, and reviews must be in English.
- Each team has to answer the questions of the following two teams by **10:30am Tuesday (3/30)**.
- Each team has to write feedbacks (reviews/encouragements) about the answers of the following two teams by **10:30am Thursday (4/1)**.
- That is, Team N answers questsions of N+1 and N+2 teams and writes feedbacks (reviews/encouragements) to the answers of N+3 and N+4 teams.
- Beware that there are gaps in team numbers.
- By 7pm next Thu, April 1st, submit a snapshot of `$./app/kens/kens-all-run-solution` on your platform.

In Programming Assignment #1, you have learned to do basic socket programming. 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 have learned 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 we list the key references.

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

Introduction to KENSv3

In most operating systems, the transport and lower layers are implemented in the kernel and are very hard to modify. We have developed a simulation environment called KENSv3 (KAIST Educational Network System) so that you can build and test basic functions of the transport layer not in the kernel but in userspace.

Let's first start installing KENS on your platform. Go to https://github.com/ANLAB-KAIST/KENSv3/wiki and follow the link to "Getting Started" for your own system.

KENSv3 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. Through KENS, students build only the TCP module as if it lies in the kernel. In a typical system, the TCP module is called upon by applications and the lower layer via three events:

(1) a system call from an application
(2) an upcall from the IP layer when a packet arrives
(3) timer expiration.

In order to implement (1) to (3), the following portions of code must be implemented.

(1) When an application makes a system call, KENS turns it into the following callback function.

```
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;

...

}
```

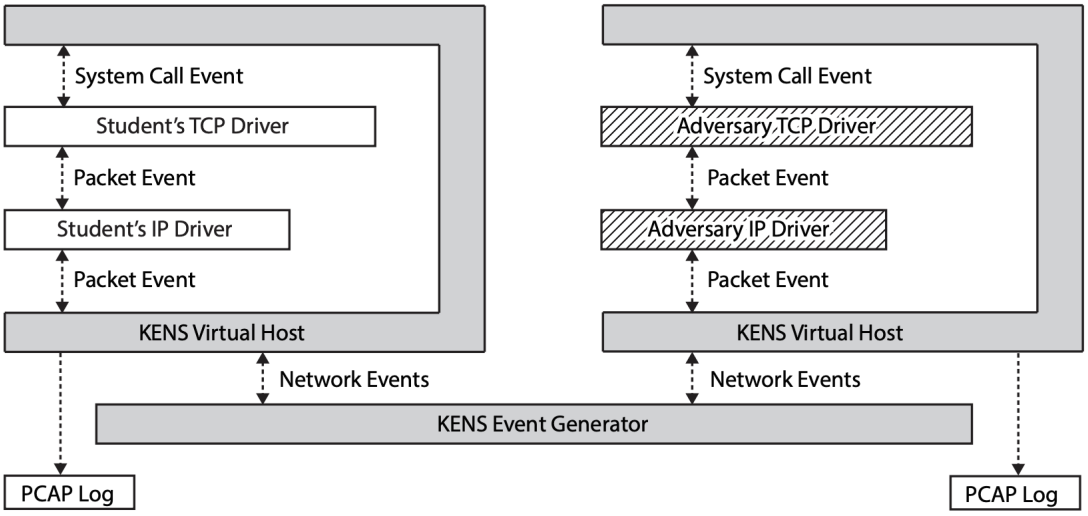
(2) When a packet arrives in the IP layer, the IP layer makes the following upcall:

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

(3) When a timer expires, just as in (1) a timer callback is called.

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 build process differs from platform to platform. Please follow instructions in “Getting Started “at: https://github.com/ANLAB-KAIST/KENSv3/wiki

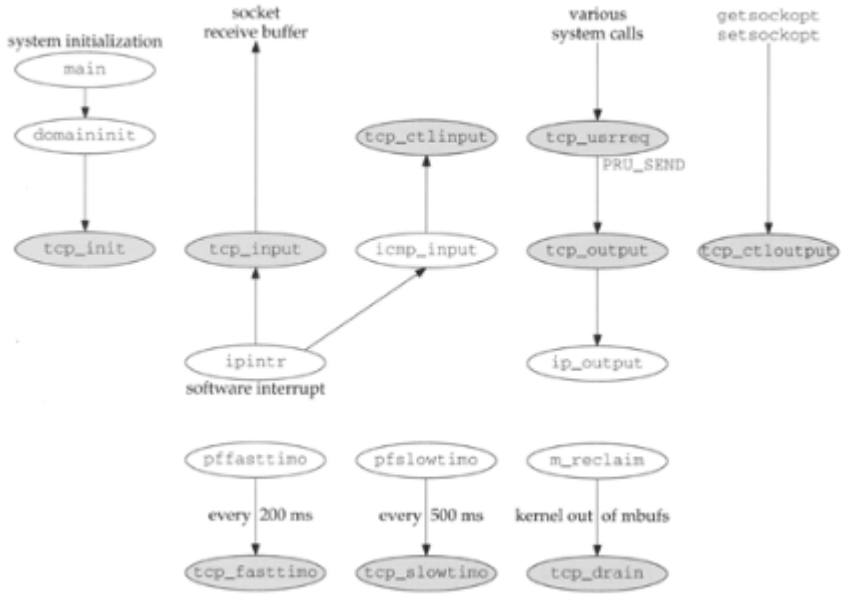


A First Step to Build Your Own TCP

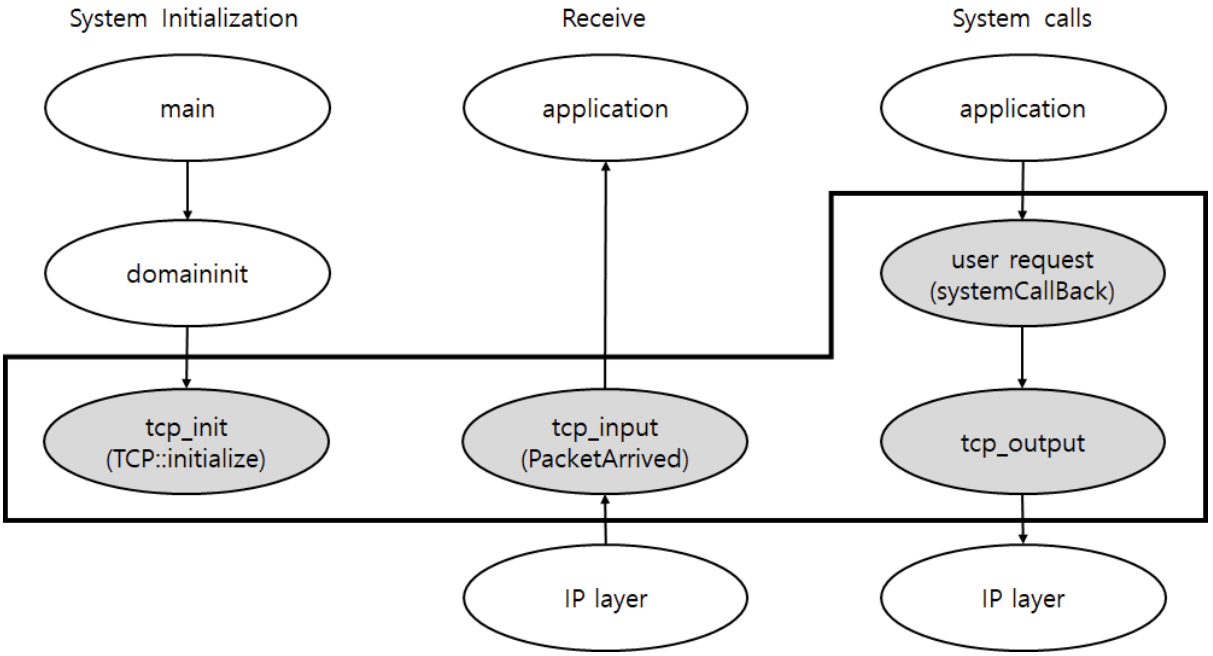
In PA #2 and PA #3 we are not building the complete TCP, but a bare minimum of the protocol. Yet any reference describes TCP in full. In order to help you understand the limited scope of PA #2, we will refer to the TCP implementation in [1] and present the scope you should implement in PA #2.

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 in KENS:



The corresponding part in KENS to tcp_input is:

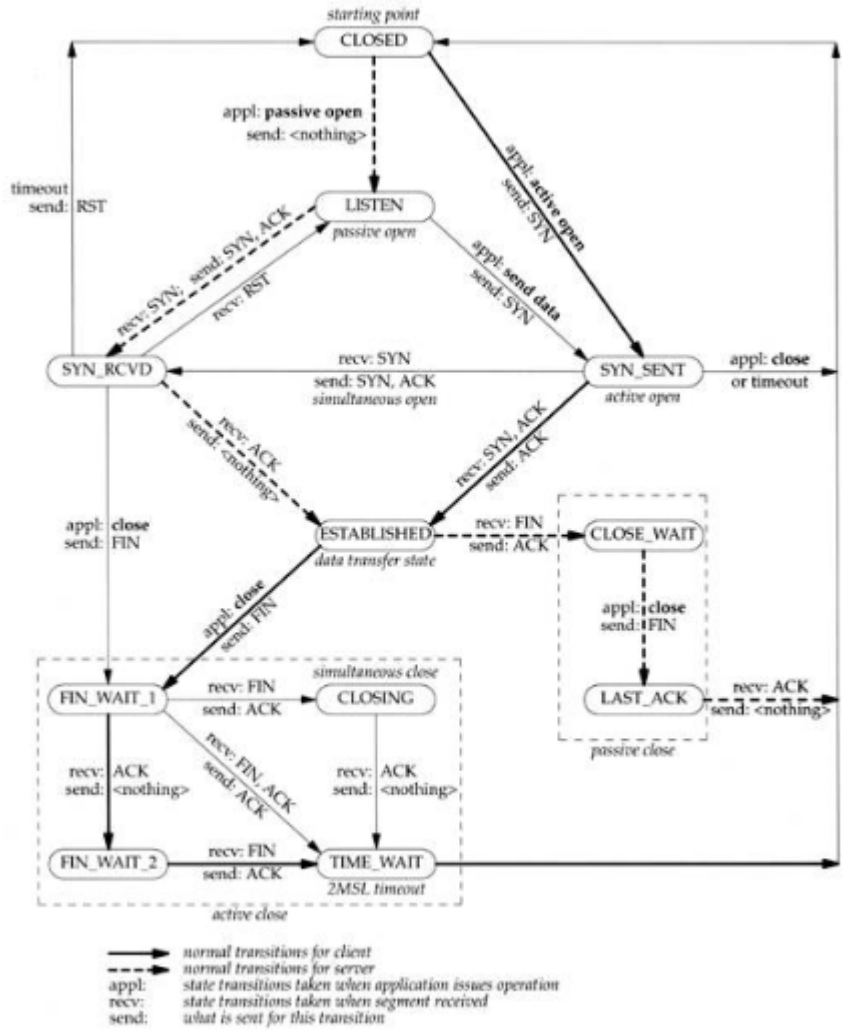
```
void TCPAssignment::packetArrived(std::string fromModule, Packet
&&packet) {
  (void)from Module;
  (void)packet;
}
```

The corresponding part in KENS to user_request is:

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

Within TCPAssignment::packetArrived() and TCPAssignment::systemCallback(), you will implement the following state transition diagram.

Figure 24.15. TCP state transition diagram.



Part 1: 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.

In PA #1 you have used the socket API. In PA #2 and PA #3, you build your own TCP layer below the socket API. You will implement the functions listed above. The skeletal code is in TCPAssignment.cpp.

The goal here is to implement the following parts in TCPAssignment.cpp and run testopen.cpp and testbind.cpp successfully.

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 testopen.cpp calls socket() many times and checks if the return values are correct.

Caution

You have to implement a very simple `close()` call to pass `testopen`. The `close()` removes its file descriptor and the latter `socket()` call would return the descriptor's value .

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.

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.

Tips

- Managing File Descriptors
- Returning a System Call

Part 2: 3-way Handshake

As we have covered 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.

Caution

In KENS, the listen()'s backlog specifies the queue length for incomplete connection requests as before Linux 2.2. See NOTES section of listen(2).

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.

Tips

- Sending and Receiving a Packet
- Figuring out the Source IP Address to Use
- Using a Timer
- Networking Utilities

More Resources

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

Submission

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 implemented in this assignment. If you wish to use token(s) for this assignment, please state clearly the number of tokens to use for each member in readme.txt.

In your readme.txt, describe how you have progressed to complete this assignment. It does not have to be long and detailed. A brief summary will suffice.