EECS 325/425 - Computer Networks I

Project 2: Transmission Control Protocol

1 Introduction

In this project you will be implementing the Transmission Control Protocol, as specified in [RFC793]. However, instead of implementing it inside the operating system itself, you will be implementing it inside a system called χTCP [1]. This system allows you to write socket-based applications that rely on your TCP implementation instead of the one included in your operating system. To do this, χTCP provides an alternate socket library, χsocket , that provides the same functions as the standard socket library (`connect`, `send`, `recv`, etc.). Although the χsocket functions have the same expected behaviour as the standard socket functions, they do not implement the entire functionality provided by standard sockets (e.g., non-blocking sockets are not supported). You will be provided with a code skeleton for your implementation. **Section 5 contains the instructions to install χTCP . Section 4 outlines the data structures and functions that you might need for your implementations.**

In χTCP , the socket layer and all the messy details of how TCP interacts with the other layers of the protocol stack (including how packets are handed down to the network layer, and how data is passed up to the application layer) are already implemented for you. In this project, you will focus on implementing the TCP protocol itself.

2 The χTCP architecture

When using χTCP in place of the operating system's TCP/IP stack, applications use χsocket functions to perform standard socket operations, like connect, send, recv, etc. Instead of including the `socket.h` system header, a χTCP header must be included:

```
#include "chitcp/socket.h"
```

The functions provided by the χsocket library are the same as the ones in the standard socket library, except they are prefixed with `chisocket_`. They provide essentially the same functionality, although they do not support every possible flag and error code. However, this is enough to write simple clients and servers. For example:

```
clientSocket = chisocket_socket(PF_INET,       // Family: IPv4
                                SOCK_STREAM,   // Type: Full-duplex stream (reliable)
                                IPPROTO_TCP);  // Protocol: TCP;

chitcp_addr_construct(host, port, &serverAddr);

chisocket_connect(clientSocket, (struct sockaddr *) &serverAddr, sizeof(struct sockaddr_in));

chisocket_send(clientSocket, "Hello!", 6, 0);

chisocket_close(clientSocket);
```

[1] This project is forked from χTCP project developed by UChicago. `http://chi.cs.uchicago.edu/chitcp/index.html`

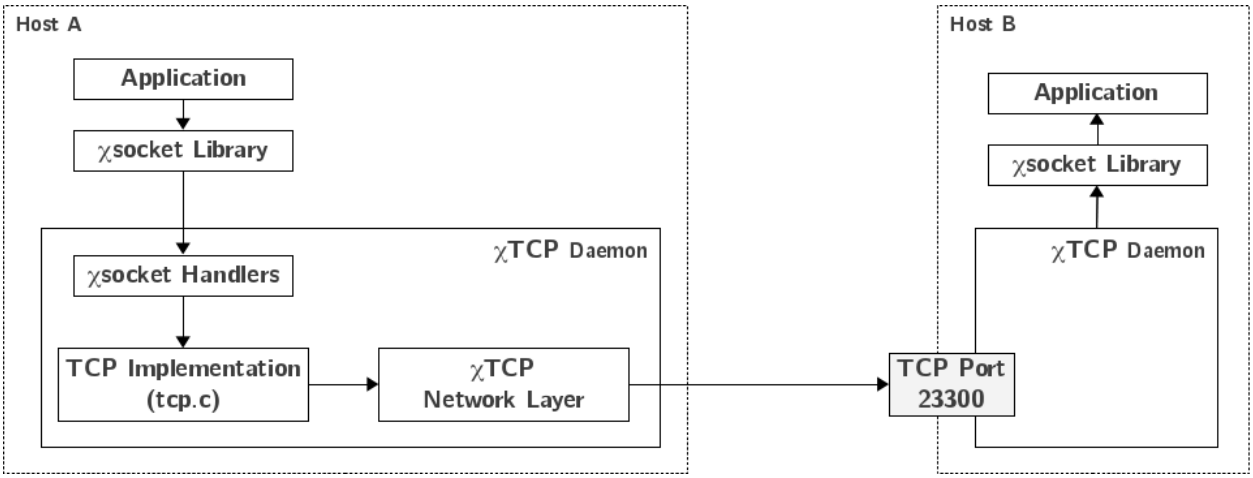

Figure 1: The χTCP architecture

For a host to be able to use χTCP or, more specifically, to be able to write servers and clients based on the χsocket library instead of the regular socket library, that host must run a χTCP daemon (called `chitcpd`). This daemon is where your implementation of TCP will reside, with the χsocket library performing the standard socket operations (connect, send, recv, etc.) through the χTCP daemon, instead of accessing the operating system's TCP/IP stack directly.

Figure 1 summarizes the χTCP architecture. Applications that want to use χTCP call the socket functions in the χsocket library, which communicates with the χTCP daemon. This daemon includes three important components:

- The χsocket Handlers: This contains the implementation of the socket layer, and is the interface between an application and your TCP implementation.
- The TCP Implementation (file `tcp.c`, described in more detail in Section 4): Your TCP implementation.
- The χTCP Network Layer: This part of the daemon is responsible for getting your TCP packet from one χTCP daemon to another, the same way that, when using your operating system's TCP/IP stack, IP is responsible for getting your TCP packet from your host to another host.

The χTCP Network Layer is actually just regular TCP (i.e., the operating system's TCP, *not* the one you are implementing). So, when χTCP needs to get one of your TCP packets to another host, it does so by establishing a (real) TCP connection to that other host's χTCP daemon on port 23300. Figure 2 shows the packet encapsulation that happens in χTCP . Notice how, from χTCP 's perspective, (real) TCP is essentially the *Network* layer of the protocol stack, while your implementation of TCP is the *Transport* layer. If we looked at this from a standard TCP/IP perspective, your TCP would simply be the payload of a (real) TCP packet.

3 Implementing RFC 793

In this project, you are going to implement a substantial portion of [RFC793]. In particular, you will be focusing on [RFC793 §3.9] (Event Processing), which provides a detailed description of how TCP should behave (whereas the preceding sections focus more on describing use cases, header specifications, example communications, etc.). The second paragraph of this section sums up pretty nicely how a TCP implementation should behave:

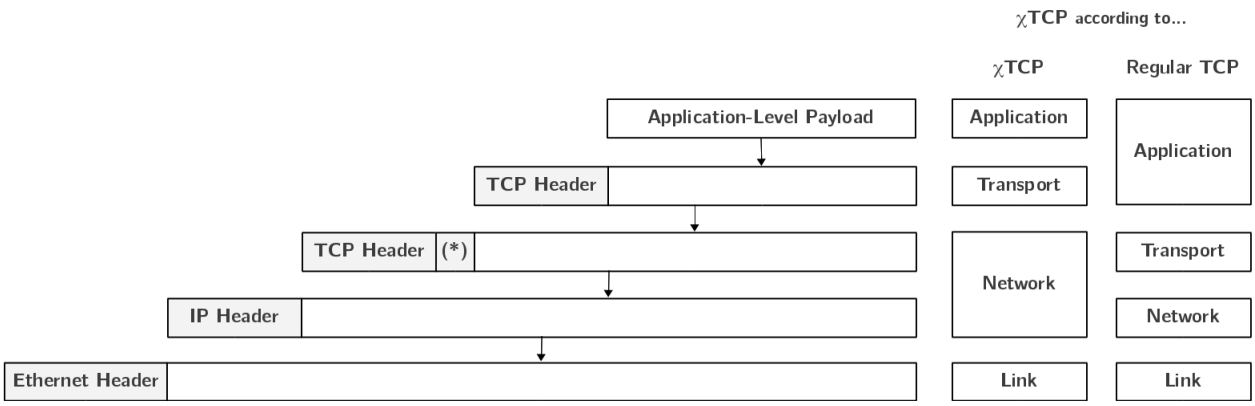

Figure 2: The χTCP layers.
(*) χTCP inserts a special header between the two TCP headers that contains χTCP -specific information.

```
The activity of the TCP can be characterized as responding to events.
The events that occur can be cast into three categories:  user calls,
arriving segments, and timeouts.  This section describes the
processing the TCP does in response to each of the events.  In many
cases the processing required depends on the state of the connection.
```

So, we can think of TCP as a state machine where:

- The states are CLOSED, LISTEN, SYN_SENT, etc.
- The inputs are a series of events defined in [RFC793] (described below)
- The transition from one TCP state to another is based on the current state, an event, *and* a series of TCP variables (SND.NXT, SND.UNA, etc. These terms are explained in [RFC793])
- Transitions from one TCP state to another result in actions, typically sending a TCP packet with information dependent on the state of the TCP variables and the send/receive buffers.

The events defined in [RFC793 §3.9] are:

- `OPEN`: χTCP will generate this event when the application layer calls `chisocket_connect`.
- `SEND`: χTCP will generate this event when the application layer calls `chisocket_send`.
- `RECEIVE`: χTCP will generate this event when the application layer calls `chisocket_recv`.
- `CLOSE`: χTCP will generate this event when the application layer calls `chisocket_close`.
- `ABORT`: Not supported by χTCP .
- `STATUS`: Not supported by χTCP .
- `SEGMENT ARRIVES`: χTCP will generate this event when a TCP packet arrives.
- `USER TIMEOUT`: Not supported by χTCP .
- `RETRANSMISSION TIMEOUT`: Not supported by χTCP .
- `TIME-WAIT TIMEOUT`: Not supported by χTCP .

As described in the next section, your work in χTCP will center mostly on a file called `tcp.c` where you are provided with functions that handle events in given TCP states. These functions are initially mostly empty, and it is up to you to write the code that will handle each event in each state.

Of course, a TCP implementation would have to consider every possible combination of states and events. However, many of these are actually invalid combinations. For example, [RFC793 §3.9] specifies that that if the `SEND` event happens in the following states:

```
FIN-WAIT-1 STATE
FIN-WAIT-2 STATE
CLOSING STATE
LAST-ACK STATE
TIME-WAIT STATE
```

Then the following action must be taken:

```
Return "error:  connection closing" and do not service request.
```

Actions like this are actually handled in the χsocket layer, and you will not have to worry about them. For example, in the above case, the `chisocket_send` function will set `errno` to `ENOTCONN`.

Section 6 carves out exactly what state/event combinations you will have to implement. Additionally, your implementation should take the following into account:

- You can assume a reliable network. You do not need to implement retransmissions or timeouts.
- You do not need to support delayed acknowledgements. An acknowledgement packet is sent immediately when data is received, although you can piggyback any data in the send buffer that is waiting to be sent (but you do not need to wait for a timeout to increase the probability that you'll be able to piggyback data on the acknowledgement).
- You do not need to support the `RST` bit.
- You do not need to support the `PSH` bit.
- You do not need to support the Urgent Pointer field or the `URG` bit in the TCP header. This also means you do not need to support the `SND.UP`, `RCV.UP`, or `SEG.UP` variables.
- You do not need to support TCP's "security/compartment" features, which means you can assume that `SEG.PRC` and `TCB.PRC` always have valid and correct values.
- You do not need to support the checksum field of the TCP header.
- You do not need to support TCP options.
- You do not need to support the `TIME_WAIT` timeout. You should still update the TCP state to `TIME_WAIT` when required, but do not have to implement a timeout. Instead, you should immediately transition to `CLOSED` from the `TIME_WAIT` state.
- You do not need to support simultaneous opens (i.e., the transition from `SYN_SENT` to `SYN_RCVD`).

4 Implementing the `tcp.c` file

Since TCP is essentially a state machine, χTCP 's implementation boils down to having a handler function for each of the TCP states (CLOSED, LISTEN, SYN_RCVD, etc.), all contained in the `src/chitcpd/tcp.c` file. If an event happens (e.g., a packet arrives) while the connection is in a specific state (e.g., ESTABLISHED), then the handler function for that state is called, along with information about the event that just happened. You will only have to worry about writing the code inside the handler function; the rest of the scaffolding (the socket library, the actual dispatching of events to the state machine, etc.) is already provided for you.

Each handler function has the following prototype:

```
int chitcpd_tcp_state_handle_STATENAME(serverinfo_t *si,
                                       chisocketentry_t *entry,
                                       tcp_event_type_t event);
```

The parameters to the function are:

- `si` is a pointer to a struct with the χTCP daemon's runtime information (e.g., the socket table, etc.). You should not need to access or modify any of the data in that struct, but you will need the `si` pointer to call certain auxiliary functions.
- `entry` is a pointer to the socket entry for the connection that is being handled. The socket entry contains the actual TCP data (variables, buffers, etc.), which can be accessed like this:

  ```
  tcp_data_t *tcp_data = &entry->socket_state.active.tcp_data;
  ```

  The contents of the `tcp_data_t` struct are described below. You should not access or modify any other information in `entry`.
- `event` is the event that is being handled. The list of possible events corresponds roughly to the ones specified in [RFC793 §3.9]. They are:
  - `APPLICATION_CONNECT`: Application has called `chisocket_connect()` and a three-way handshake must be initiated.
  - `APPLICATION_SEND`: Application has called `chisocket_send()`. The socket layer (which is already implemented for you) already takes care of placing the data in the socket's TCP send buffer. This event is a notification that there may be new data in the send buffer, which should be sent if possible.
  - `APPLICATION_RECEIVE`: Application has called `chisocket_recv()`. The socket layer already takes care of extracting the data from the socket's TCP receive buffer. This event is a notification that there may now be additional space available in the receive buffer, which would require updating the socket's receive window (and the advertised window).
  - `APPLICATION_CLOSE`: Application has called `chisocket_close()` and a connection tear-down should be initiated once all outstanding data in the send buffer has been sent.
  - `PACKET_ARRIVAL`: A packet has arrived through the network and needs to be processed (RFC 793 calls this "SEGMENT ARRIVES")
  - `TIMEOUT`[2]: A timeout (e.g., a retransmission timeout) has happened.

To implement the TCP protocol, you will need to implement the handler functions in `tcp.c`. You should not need to modify any other file. However, you will need to use a number of functions and structs defined elsewhere.

[2]Not currently supported in χTCP

4.1 The `tcp_data_t` struct

This struct contains all the TCP data for a given socket:

The pending packet queue

```
list_t pending_packets;
pthread_mutex_t lock_pending_packets;
pthread_cond_t cv_pending_packets;
```

As TCP packets arrive through the network, the χTCP daemon places them in the pending packet queue of the appropriate socket (you do not need inspect the origin and destination port of the TCP packet; this is taken care of for you). The list contains pointers to `tcp_packet_t` structs (described below) in the heap. It is your responsibility to free this memory when you are done processing a packet.

The queue is implemented with the SimCList[3] library, which is already included in the χTCP code, and the head of the queue can be retrieved using SimCList's `list_fetch` function. The `lock_pending_packets` mutex provides thread-safe access to the queue. The `cv_pending_packets` condition variable is used to notify other parts of the χTCP code that there are new packets in the queue; you should not wait or signal this condition variable.

The TCP variables

```
/* Send sequence variables */
uint32_t ISS;      /* Initial send sequence number */
uint32_t SND_UNA;  /* First byte sent but not acknowledged */
uint32_t SND_NXT;  /* Next sendable byte */
uint32_t SND_WND;  /* Send Window */

/* Receive sequence variables */
uint32_t IRS;      /* Initial receive sequence number */
uint32_t RCV_NXT;  /* Next byte expected */
uint32_t RCV_WND;  /* Receive Window */
```

These are the TCP sequence variables as specified in [RFC793 §3.2].

The TCP buffers

```
circular_buffer_t send;
circular_buffer_t recv;
```

These are the TCP send and receive buffers for this socket. The `circular_buffer_t` type is defined in the `include/chitcp/buffer.h` and `src/libchitcp/buffer.c` files. You are provided with the implementation of this type, which will be enough to run some basic tests with your TCP implementation.

The management of these buffers is already partially implemented:

- The `chisocket_send()` function places data in the send buffer and generates an `APPLICATION_SEND` event.
- The `chisocket_recv()` function extracts data from the receive buffer and generates an `APPLICATION_RECV` event.

You do not need to implement the above functionality; it is already implemented for you. On the other hand, you will be responsible for the following:

- When an `APPLICATION_SEND` event happens, you must check the send buffer to see if there is any data ready to send, and you must send it out if possible (i.e., if allowed by the send window).

[3] http://mij.oltrelinux.com/devel/simclist/

- When a `PACKET_ARRIVAL` event happens (i.e., when the peer sends us data), you must extract the packets from the pending packet queue, extract the data from those packets, verify that the sequence numbers are correct, and put the data in the receive buffer.
- When an `APPLICATION_RECV` event happens, you do not need to modify the receive buffer in any way, but you do need to check whether the size of the send window should be adjusted.

4.2 The `tcp_packet_t` struct

The `tcp_packet_t` struct is used to store a single TCP packet:

```
typedef struct tcp_packet
{
    uint8_t *raw;
    size_t  length;
} tcp_packet_t;
```

This struct simply contains a pointer to the packet in the heap, along with its total length (including the TCP header). You will rarely have to work with the TCP packet directly at the bit level. Instead, the `include/chitcp/packet.h` header defines a number of functions, macros, and structs that you can use to more easily work with TCP packets. More specifically:

- Use the `TCP_PACKET_HEADER` to extract the header of the packet (with type `tcphdr_t`, also defined in `include/chitcp/packet.h`, which provides convenient access to all the header fields. Take into account that all the values in the header are in network-order: you will need to convert them to host-order before using using (and viceversa when creating a packet that will be sent to the peer).
- Use the `TCP_PAYLOAD_START` and `TCP_PAYLOAD_LEN` macros to obtain a pointer to the packet's payload and its length, respectively.
- Use the `SEG_SEQ`, `SEG_ACK`, `SEG_LEN`, `SEG_WND`, `SEG_UP` macros to access the `SEG.*` variables defined in [RFC793 §3.2]. Take into account that these macros *do* convert the values from network-order to host-order.
- Finally, although this header file provides a `chitcp_tcp_packet_create` function, you should not use this function directly. Instead, use `chitcpd_tcp_packet_create` (note the `chitcpd` prefix, not `chitcp`) defined in `src/chitcpd/serverinfo.h`, which is a wrapper around `chitcp_tcp_packet_create` (besides creating a packet, it will also correctly initialize the source and destination ports to match those of the socket).

4.3 The `chitcpd_update_tcp_state` function

This function is defined in `src/chitcpd/serverinfo.h`. Whenever you need to change the TCP state, you **must use this function**. For example:

```
chitcpd_update_tcp_state(si, entry, ESTABLISHED);
```

The `si` and `entry` parameters are the same ones that are passed to the TCP handler function.

4.4 The `chitcpd_send_tcp_packet` function

This function is defined in `src/chitcpd/connection.h`. Whenever you need to send a TCP packet to the socket's peer, you **must use this function**. For example:

```
tcp_packet_t packet;

/* Initialize values in packet */

chitcpd_send_tcp_packet(si, entry, &packet);
```

The `si` and `entry` parameters are the same ones that are passed to the TCP handler function.

4.5 The logging functions

The χTCP daemon prints out detailed information to standard output using a series of logging functions declared in `src/include/log.h`. We encourage you to use these logging functions instead of using `printf` directly. More specifically, you should use the printf-style `chilog()` function to print messages:

```
chilog(WARNING, "Asked send buffer for %i bytes, but got %i.", nbytes, tosend);
```

And the `chilog_tcp()` function to dump the contents of a TCP packet:

```
tcp_packet_t packet;

/* Initialize values in packet */

chilog(DEBUG, "Sending packet...");
chilog_tcp(DEBUG, packet, LOG_OUTBOUND);
chitcpd_send_tcp_packet(si, entry, &packet);
```

The third parameter of `chilog_tcp` can be `LOG_INBOUND` or `LOG_OUTBOUND` to designate a packet that is being received or sent, respectively (this affects the formatting of the packet in the log). `LOG_NO_DIRECTION` can also be used to indicate that the packet is neither inbound or outbound.

In both functions, the first parameter is used to specify the log level:

- `CRITICAL`: Used for critical errors for which the only solution is to exit the program.
- `ERROR`: Used for non-critical errors, which may allow the program to continue running, but a specific part of it to fail (e.g., an individual socket).
- `WARNING`: Used to indicate unexpected situation which, while not technically an error, could cause one.
- `INFO`: Used to print general information about the state of the program.
- `DEBUG`: Used to print detailed information about the state of the program.
- `TRACE`: Used to print low-level information, such as function entry/exit points, dumps of entire data structures, etc.

The level of logging is controlled by the `-v` argument when running `chitcpd`:

- No `-v` argument: Print only `CRITICAL` and `ERROR` messages.
- `-v`: Also print `WARNING` and `INFO` messages.
- `-vv`: Also print `DEBUG` messages.
- `-vvv`: Also print `TRACE` messages.

5 Building and testing the χTCP code

We will use the same Virtual Machine image for developing this project. First, clone χTCP repository from GitHub to your home directory in VM:

```
git clone https://github.com/uchicago-cs/chitcp.git
```

Then, download `install-chitcp.sh` from Canvas and execute the script:

```
./install-chitcp.sh
```

You should only need to rerun the above commands if you modify CMake's `CMakeLists.txt` (which you should not need to do as part of this project). Once you have done this, simply run `make` inside the build directory to build χTCP . This will generate the χTCP daemon (`chitcpd`), some sample programs, as well as the test executables (all starting with `test-`). Take into account that you must run these programs from inside the `build` directory.

By default, `make` will only print the names of the files it is building. To enable a more verbose output (including the exact commands that make is running during the build process), just run `make` like this:

```
make VERBOSE=1
```

This will generate two files:

- `chitcpd`: The χTCP daemon. You can verify that it works correctly by running the following:

  ```
  ./chitcpd -v
  ```

  You should see the following output:

  ```
  [2014-02-02 11:36:07]   INFO lt-chitcpd chitcpd running. UNIX socket: /tmp/chitcpd.socket. TCP socket: 23300
  ```

  Note that, by default, `chitcpd` will run on port 23300. **You do not need to run this daemon for testing your implementation.** Instead, you should run the test suite discussed next.

- `./.libs/libchitcp.so`: The `libchitcp` library. Any applications that want to use the χsocket library will need to link with this library.

Finally, to run the χTCP test suite, run the following command inside the `build` directory for testing connection initiation:

```
LOG=MINIMAL ./test-tcp --filter 'conn_init/*'
```

or use for colorizing this output for extra readability:

```
LOG=MINIMAL ./test-tcp --filter 'conn_init/*' | ../tests/colorize-minimal.sh
```

To test all the tasks you implemented for this project, run:

```
LOG=MINIMAL ./test-tcp --filter '?(conn_init|data_transfer|conn_term)/*'
```

Take into account that, until you implement TCP, many of these tests will fail.

6 TCP over a Reliable Network

This project will be divided into three sub-tasks:

- Implementing the TCP 3-way handshake (**checkpoint-1: March 27^{th}**)
- Sending and receiving data (**checkpoint-2: April 3^{st}**)
- Connection termination (**submission deadline: April 10^{th}**)

There will be 2 checkpoints during this project. You will be provided with the implementations for subtask 1 and 2 after the checkpoint dates so that you could continue working on the remaining tasks with the provided code. **Thus, it is important to submit your code on time!**

NO LATE submissions will be accepted for checkpoint-1 and checkpoint-2. Final late submissions are accepted within 3 days after the deadline, with 20% late penalty.

Implementing the TCP 3-way handshake

In `tcp.c` you must implement the following:

- Handling event `APPLICATION_CONNECT` in `chitcpd_tcp_state_handle_CLOSED`. This corresponds to handling the `OPEN Call` in the `CLOSED STATE` in [RFC793 §3.9].
- Handling event `PACKET_ARRIVAL` in:
  - `chitcpd_tcp_state_handle_LISTEN`
  - `chitcpd_tcp_state_handle_SYN_SENT`
  - `chitcpd_tcp_state_handle_SYN_RCVD`

  As described in the `SEGMENT ARRIVES` portion of [RFC793 §3.9].

Suggestion: Instead of writing separate pieces of code in each of the handler functions where you're handling the `PACKET_ARRIVAL` event, you may want to write a single function whose purpose is to handle packets in any TCP state, following the general procedure described in pages 64-75 of [RFC793]. This will also make it easier to implement the rest of the project.

Sending and receiving data

In this task, you will complete your TCP implementation. This involves handling the following events in `chitcpd_tcp_state_handle_ESTABLISHED`:

- Event `PACKET_ARRIVAL`, as described in the `SEGMENT ARRIVES` portion of [RFC793 §3.9], but without handling `FIN` packets.
- Event `APPLICATION_SEND`. This corresponds to handling the `SEND Call` in the `ESTABLISHED STATE` in [RFC793 §3.9]. Take into account that the χsocket layer already takes care of putting data in the send buffer. So, this event notifies your TCP implementation that there is new data in the send buffer, and that it should be sent if possible.
- Event `APPLICATION_RECEIVE`. This corresponds to handling the `RECEIVE Call` in the `ESTABLISHED STATE` in [RFC793 §3.9]. Take into account that the χsocket layer already takes care of retrieving data from the receive buffer and handing it to the application layer. This event notifies your TCP implementation that space has become available in the buffer, and you should update the TCP internal variables accordingly.

Connection tear-down

This involves handling the `APPLICATION_CLOSE` event in the following handlers:

- `chitcpd_tcp_state_handle_ESTABLISHED`
- `chitcpd_tcp_state_handle_CLOSE_WAIT`

Both of these correspond to handling the `CLOSE Call` in the `ESTABLISHED STATE` and `CLOSE-WAIT STATE` in [RFC793 §3.9].

You also need to handle the `PACKET_ARRIVAL` event in the following handlers:

- `chitcpd_tcp_state_handle_FIN_WAIT_1`
- `chitcpd_tcp_state_handle_FIN_WAIT_2`
- `chitcpd_tcp_state_handle_CLOSE_WAIT`
- `chitcpd_tcp_state_handle_CLOSING`
- `chitcpd_tcp_state_handle_LAST_ACK`
- Modify the handling of this event in `chitcpd_tcp_state_handle_ESTABLISHED` to handle `FIN` packets.

All of these are described in the `SEGMENT ARRIVES` portion of [RFC793 §3.9].

7 Testing your implementation

To test your implementation, we have provided some basic tests. To run these tests, just run:

```
LOG=MINIMAL ./test-tcp --filter 'conn_init/*'
```

```
LOG=MINIMAL ./test-tcp --filter 'data_transfer/*'
```

```
LOG=MINIMAL ./test-tcp --filter 'conn_term/*'
```

More details could be found: `http://chi.cs.uchicago.edu/chitcp/testing.html`

8 Submission deadlines

- Implementing the TCP 3-way handshake (**checkpoint-1: March 27^{th}, 11:59 pm**)
- Sending and receiving data (**checkpoint-2: April 3^{st}, 11:59 pm**)
- Connection termination (**submission deadline: April 10^{th}, 11:59 pm**)

Again, **NO LATE submissions for checkpoint-1 and checkpoint-2 will be accepted**.