Lab 3: Reliable Transport (15 Points)

1 Download Lab

Download the Lab 3 files from Brightspace. The source code will be inside the directory "`Lab3/`". You must use the environment from Lab 0 to run and test your code. Next, open a terminal and "`cd`" into the "`Lab3/`" directory. Now you are ready to run the lab!

2 Task

For this lab, your task is to implement a reliable transport layer protocol similar to TCP. We will call it `tinytcp`. Just like TCP, `tinytcp` must deliver *a stream of bytes* ***reliably*** *and* ***in-order*** to applications over the network. To accomplish this, `tinytcp` must implement connection set up and termination, application multiplexing, data acknowledgments and retransmissions, similar to TCP. However, unlike TCP, `tinytcp` need not implement flow control and congestion control.

2.1 File transfer application

We will run a file transfer application over `tinytcp`. The files will be transferred from a client to a server. You can assume that there will be a single client and a single server. However, the client can initiate multiple (up to a maximum of 5) file transfers in parallel. Internally, `tinytcp` would create multiple parallel streams (one for each file transfer) between the client and the server. Your implementation must be capable of multiplexing between multiple `tinytcp` streams.

3 Building and Running the Code

You must build, run, and test your code on `ecegrid` using the environment from Lab 0. If your code does not run in that environment, you will not get any credit!

`tinytcp` is a real system that can be used to transfer files reliably between any two machines over a network. However, for the ease of testing, you will run both the client and the server on the same machine, connected over the loopback network interface (IP: 127.0.0.1, defined in "`SERVADDR`" parameter in "`tinytcp.c`"). For remote testing, you would change "`SERVADDR`" value to the IP address of the remote server (not required for this lab). Since there would not be any physical link between the client and the server in our loopback network, we simulate the link inside "`tinytcp.c`". The link delay is configured using the parameter "`DELAY`" in "`tinytcp.h`". The link can also be configured to be lossy as described below.

You should open two terminals. Compile the code by running "`make`" in one of the terminals,

```
$ make
```

Next, to start the server, run the following command in one of the terminals,

```
$ ./bin/tinytcp server [loss_probability]
```

IMPORTANT: Always start the server before starting the client! Also, re-start both the client and the server for each new experiment run.

Here, "`loss_probability`" must be a number between 0 and 100. This value configures the probability with which a packet will be dropped/reordered over the simulated link. For example, if you set the

loss probability value to 0, no packets will be dropped/reordered over the simulated link. Similarly, if you set the value to 100, all packets will be dropped. You can assume that the three connection set up packets (`SYN`, `SYN-ACK`, `ACK`) and the three connection termination packets (`FIN`, `FIN-ACK`, `ACK`) are **never** dropped, regardless of the loss probability value.

NOTE: The link delay in the code is simulated using packet buffers. So if your implementation is too slow, i.e., you do not process received packets fast enough, then the buffer at the receiver will overflow and packets will be dropped **regardless** of the loss probability value. This is a very serious issue, and generally means that your code is stuck in some deadlock or your packet receive implementation is too slow. If this happens, you should see the following messages printed on stdout:
`ALERT!! link delay buffer overflow at client!! Packet from server dropped!!`
`ALERT!! link delay buffer overflow at server!! Packet from client dropped!!`
You must debug your code to get rid of any such messages.

Finally, to start the client, run the following command in the other terminal,

```
$ ./bin/tinytcp client [loss_probability] [file1 file2 ...]
```

Here, "`file1`", "`file2`" etc. are the filenames that you want to transfer. You can specify up to a maximum of 5 files. Also note that the loss probability values *can* be different for the client and the server. Client's loss probability value determines the probability with which packets sent by the client will be dropped/reordered (similarly for the server).

IMPORTANT: All files that you want to transfer **must** be stored inside the "`sendfiles/`" directory. We have already provided you with a set of files that you can use for testing. All the files received at the server are stored inside the "`recvfiles/`" directory.

Debugging Tip: Each run of the code generates four .dump files inside the directory "`dumps/`". These files store the information about all the packets that were sent and received by the client and the server during the run of the experiment. You can use these files to debug your implementation.

To clean the working directory, run the following command in one of the terminals,

```
$ make clean
```

This will clean all the unnecessary files, including the dump files and the received files from last run.

4 Output

Figure 4 and Figure 5 show sample terminal outputs for the client and the server respectively. Note that these outputs are for a fully implemented `tinytcp`. The output of a partially implemented `tinytcp` might not show all the information.

In the sample output, client is sending three files in parallel. The loss probability is 30% on both the client and the server side. The output shows the information about the connection set up and termination packets. In addition, the output also shows the progress of file transfer using the character "=". On the client side, the character "=" is printed every time a packet is sent to the network. On the server side, character "=" is printed every time some data is written to the receiving file. You can use this to make sure that your code is making progress. Note that it is normal for the client to print more "=" characters than the server, since not all packets sent by the client will reach the server if the link is lossy, and the server might also batch data from received packets before writing to the receiving file. Once all the files have been transferred, the code will print the transfer statistics on the client terminal. This includes the total time of transfer and the total bytes sent (including the ones that were dropped). At the end, the code matches the received files against the corresponding sent files, and prints either "SUCCESS" or "FAILURE" depending upon whether the sent and received files are identical or not.

5 Milestones and Grading

This lab has four milestones. For grading milestones 1–3, we will run multiple experiments per milestone with different files, loss probabilities, and configuration parameter values (inside `.h` files), and for each milestone, your grade will be the percentage of experiments you passed. For grading milestone 4, we will consider the runtime performance of your code over the experiments used to grade milestones 1–3.

5.1 Milestone 1: Connection set up and termination (4 Points)

For the first milestone, you are required to implement only the connection set up and termination part of the code, without worrying about any actual file data transfer. **You must choose the initial sequence number (ISN) for both the client and the server randomly**. You can use the "`empty.txt`" file inside the "`sendfiles/`" directory to test this milestone. For example, you can run,

```
$ ./bin/tinytcp client 0 empty.txt
```

```
$ ./bin/tinytcp server 0
```

Your code must work with multiple empty files sent in parallel.

5.2 Milestone 2: File transfer over a lossless link (3 Points)

For the second milestone, you must be able to successfully transfer file data between the client and the server. You can assume that the link is lossless for this milestone, i.e., set the loss probability value to 0 on both the client and the server side. Your code must be able to successfully transfer multiple files in parallel.

5.3 Milestone 3: File transfer over a lossy link – Performance Agnostic (4 Points)

For the third milestone, you must be able to successfully transfer multiple files in parallel over a lossy link. Your code must work with any loss probability value on both the client and the server side. Note that the focus of this milestone is only on correctness, and not on performance. So you are free to implement even the most naive reliability protocol, for example, send a single data packet and wait for an ACK until timeout. If the ACK arrives before timeout, send the next packet in line, else retransmit the first packet.

5.4 Milestone 4: File transfer over a lossy link – Performance Aware (4 Points)

For the final milestone, you must be able to successfully transfer multiple files in parallel over a lossy link using Cumulative ACK with Go-Back-N, and retransmit on receiving 3 Duplicate ACKs (Fast Retransmit) or timeout, whichever happens first. We will test the performance of your implementation against our own implementation. To receive full credit, both the total time taken for file transfer and the total bytes sent must be in the same ball-park as our implementation. Also, do not use the performance numbers from the sample output in Figure 4 as the benchmark, as they were generated in a different environment. Instead, we recommend comparing your performance numbers with other groups to gauge the efficiency of your implementation.

IMPORTANT: The retransmission timeout value is already configured in the code provided to you (`RTO` value in file "`include/tinytcp.h`"). You must **not** retransmit packets using some different timeout value. This is to ensure that the performance comparison between your implementation and our implementation is fair. **Violation of this guideline will result in significant grade penalty!**

6 Source Code

The abstract communication architecture for this lab is shown in Figure 1. Both the client and server applications run over `tinytcp` transport layer. The physical link between the client and the server is simulated using a buffer, called "`link_delay_buffer`" inside "`tinytcp.c`". The buffer size is configured using "`LINK_DELAY_BUF_SIZE`" inside "`include/tinytcp.h`". The buffer also introduces a delay (configured using "`DELAY`" inside "`tinytcp.h`") to simulate the link propagation delay. Packets sent from client to server and vice-versa are buffered in the link delay buffers for "`DELAY`" amount of time before being received at the other end. If these buffers overflow, probably because you are sending faster than you can process at the receiver, you will see the "`ALERT!!`" messages (ref. Section 3) printed on stdout alerting you of packet drops due to link delay buffer overflow.

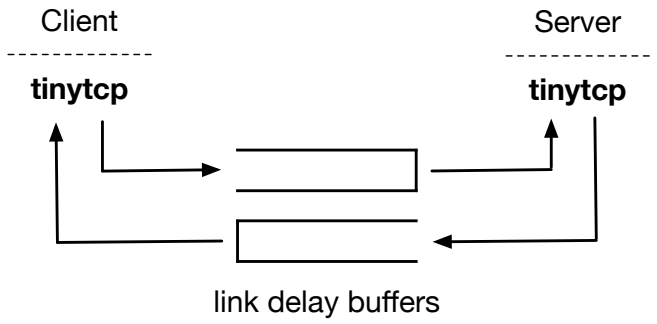

Figure 1: Communication Architecture.

The source files for this lab can be found inside the "`src/`" directory. You are provided with three source files — "`tinytcp.c`", "`ring_buffer.c`", and "`handle.c`". You will do all your implementation inside "`handle.c`". You must **not** modify any other files. You must also **not** delete/modify any existing code inside "`handle.c`". The "`handle.c`" file is tagged with several TODO comments that guide you in the right direction in terms of what to implement. Read those comments very carefully. The header files are inside "`include/`" directory. In particular, you should go through "`tinytcp.h`" header file to familiarize yourself with all the macros and structure definitions used in `tinytcp`.

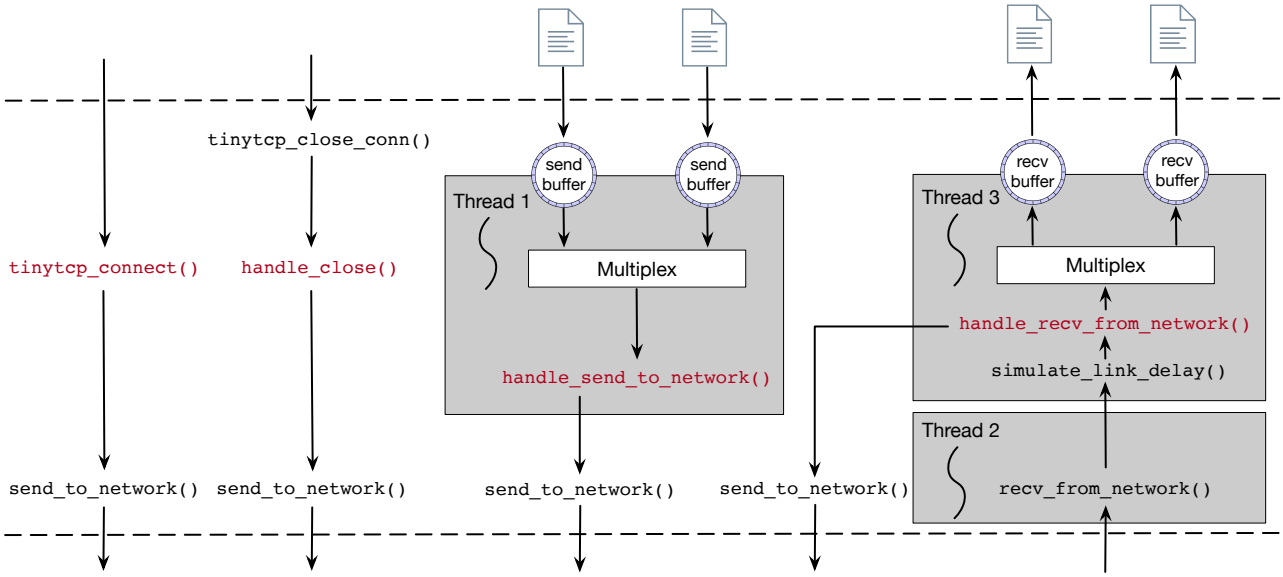

Figure 2: Workflow of tinytcp. Functions in red need to be implemented as part of this lab.

6.1 Event Driven Programming

The same source code runs on both the client and the server. This is similar to the real world TCP, where every computer runs the same TCP code regardless of whether the computer acts as a client or a server. The reason for this is that there is no concept of a client and a server at the transport layer; this distinction is made only at the application layer.

The key to writing a symmetric transport layer code is to focus on the transport layer **events** and not on the application characteristics (which could be asymmetric like in this lab). Example events could be receiving a SYN packet, or sending a DATA packet, or receiving an ACK packet, etc. The key here is to implement the transport behavior for each of these events in isolation within a single source file, and then depending upon whether you are a client application or a server application, the appropriate events will be triggered automatically. For example, if you are a client in this lab initiating a `tinytcp` connection, an example event triggered at the transport layer would be sending of a SYN packet, where as if you are the server running the same exact code, this event would never be triggered in this lab, and instead events such as receiving a SYN packet and sending a SYN-ACK packet would be triggered.

Guideline: You should implement `tinytcp` using the principled approach described above, instead of trying to use hacks to distinguish between the client and server operations, that might work for the given lab, but will not work if we changed the application. **Under the OSI model, an ideal transport layer implementation is independent of the application characteristics running on top!** However, you will not be penalized if you do not follow this guideline, as long as your code works.

6.2 Workflow

To start a file transfer, the client side application first calls "`tinytcp_create_conn()`". This creates a "`tinytcp_conn_t`" structure (defined in "`include/tinytcp.h`"). This structure stores all the connection-specific information in `tinytcp`.

Next, the client calls "`tinytcp_connect()`" inside "`handle.c`". This starts the 3-way handshake with the server. It first sends a `SYN` packet, waits to receive a `SYN-ACK` packet, and finally sends an `ACK` packet, and returns. After this function successfully returns, the connection is established. **You are supposed to implement** `tinytcp_connect()`**.**

`tinytcp` runs a separate thread for receiving packets. The receive thread ("`recv_from_network()`" inside "`tinytcp.c`") runs in an infinite loop, and on receiving a packet, it enqueues the packet into a buffer (simulating the link delay). A separate thread ("`simulate_link_delay()`" inside "`tinytcp.c`") dequeues the packet from the buffer after "`DELAY`" time interval (defined in "`include/tinytcp.h`"), and calls the "`handle_recv_from_network()`" function inside "`handle.c`". **You are supposed to implement** `handle_recv_from_network()`**.**

For file data transfer, the client side application periodically reads small chunks of data from the specified file, and writes it to that particular connection's send buffer. In case of multiple file transfer connections, data is read concurrently into the respective send buffers. This is implemented inside "`tinytcp.c`". In a separate thread, `tinytcp` runs "`handle_send_to_network()`" function. This function multiplexes between all the active send buffers in an infinite loop, and chooses one send buffer at a time in a round robin manner. It then sends some data from that buffer into the network (by calling "`create_tinytcp_pkt()`" followed by "`send_to_network()`", both implemented in "`tinytcp.c`"), before moving to the next send buffer. **You are supposed to implement** `handle_send_to_network()`**.**

NOTE: You can send at most "`MSS`" bytes (defined in "`include/tinytcp.h`") of payload data in a single `tinytcp` packet.

On the server side, on receiving some data from the network, the "handle_recv_from_network()" function inside "handle.c" writes the data into the corresponding connection's receive buffer. The server side application periodically reads data from all the active receive buffers, and writes it to the respective files inside the "recvfiles/" directory.

Once the application has read the entire file on the client side, it calls "tinytcp_close_conn()" function inside "tinytcp.c". This function changes the state of the connection to "READY_TO_TERMINATE" and returns immediately. The "handle_send_to_network()" function inside "handle.c" on realizing that the connection is ready to terminate, waits till all the remaining data inside the connection's send buffer has been reliably transferred, and then initiates connection termination by calling "handle_close()" function in "handle.c". "handle_close()" function sends a FIN packet, waits to receive a FIN-ACK packet, and finally sends an ACK packet and returns. After this function successfully returns, the connection is terminated. **You are supposed to implement** handle_close().

6.3 Send and Receive Buffers

The send and receive buffers in tinytcp are instances of the **ring buffer** data structure implemented in "ring_buffer.c", to match real world implementations. A ring buffer is a fixed-capacity First-In-First-Out (FIFO) queue. The ring buffer has a head and a tail pointer. New data is always added at the tail of the ring buffer, and the tail pointer is then moved to the end of the new data ("ring_buffer_add()" function). Similarly, data is always removed (read) from the head of the ring buffer, and the head pointer is moved ahead by the amount of data read ("ring_buffer_remove()" function). All the valid data inside a ring buffer lies between the head and the tail pointers, and the ring buffer is considered empty if the head and the tail pointers coincide. We provide you with the implementations of all the basic ring buffer operations, but you may **need to implement a few extra ring buffer operations** for Cumulative ACKing and Go-Back-N. If so, do it inside "handle.c" file.

6.4 State Machine

A tinytcp connection can be in one of 11 different states, as defined in enum "tinytcp_conn_state_t" in "include/tinytcp.h" file. The state of a connection is configured using the "curr_state" variable in "tinytcp_conn_t" structure. For your implementation, you will have to constantly transition between different states. Below we describe the various state transitions in tinytcp.

A tinytcp connection starts in an undefined state. Just before sending a SYN packet, the current state changes to SYN_SENT. On receiving a SYN packet, the current state changes to SYN_RECVD. Just before sending a SYN-ACK packet, the current state changes to SYN_ACK_SENT. On receiving a SYN-ACK packet, the current state changes to SYN_ACK_RECVD. Just before sending the ACK packet in connection set up, the current state changes to CONN_ESTABLISHED. Similarly, on receiving an ACK packet in connection set up, the current state changes to CONN_ESTABLISHED.

All file data transfers happen when both the client and the server side connections in tinytcp are in the CONN_ESTABLISHED state. Once the client has read the entire file into the send buffer, it changes the current connection state from CONN_ESTABLISHED to READY_TO_TERMINATE. The client remains in this state until the send buffer becomes empty, after which it sends a FIN packet.

Just before sending a FIN packet, the current state changes to FIN_SENT. On receiving a FIN packet, the current state changes to FIN_RECVD. Just before sending a FIN-ACK packet, the current state changes to FIN_ACK_SENT. On receiving a FIN-ACK packet, the current state changes to FIN_ACK_RECVD. Just before sending the ACK packet in connection termination, the current state changes to CONN_TERMINATED. Similarly, on receiving an ACK packet in connection termination, the state changes to CONN_TERMINATED.

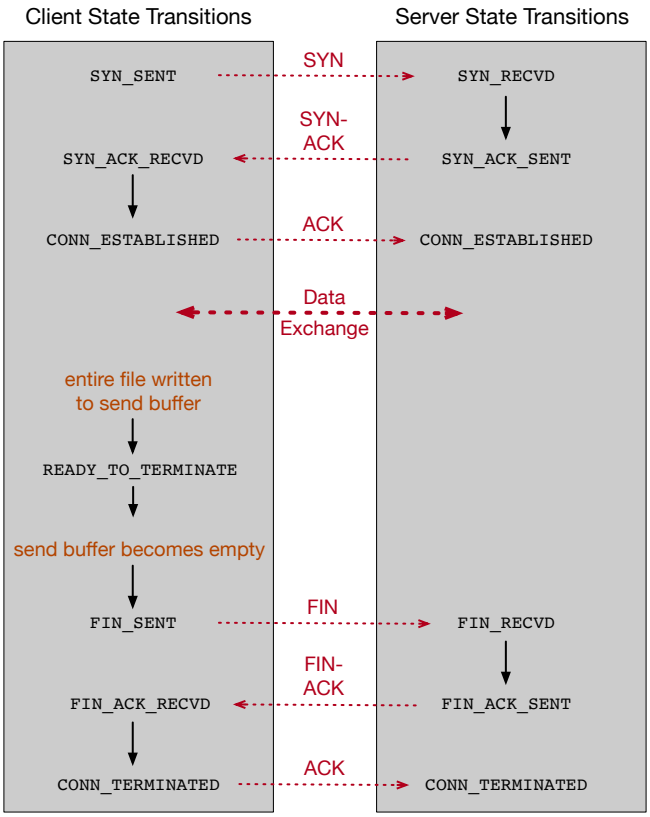

Figure 3: State transitions in tinytcp.

7 Memory Leaks

The start-up code provided to you does not leak any memory at runtime. However, tools such as "`valgrind`" can report memory leaks when the program terminates. This is because `tinytcp` code is written to run forever in an infinite loop (similar to a real TCP implementation). So there is no mechanism for a graceful termination of the program, and it needs to be terminated abruptly, e.g., using exit(1). As a result, some of the memory might not be freed before the code was abruptly terminated. In particular, we do not garbage collect the "`tinytcp_conn_t`" data structure, do not close the dump files, and do not destroy send, receive, and link delay threads, which are designed to run forever. All these will be reported as memory leaks by "`valgrind`", and should be ignored. The OS will reclaim this memory once the program terminates.

However, you should ensure that your code is not leaking memory during the runtime, or else your program may run out of heap memory and crash! This will result in an automatic 0 on the lab. Note that we will not explicitly check for any memory leaks, so you will not be penalized for any memory leaks in your implementation, as long as your program does not crash.

8 Multi-threading

`tinytcp` code is multi-threaded, and has four threads running in parallel, including the "main (default)" thread. So if you have a state or a variable that can be accessed concurrently from multiple threads (resulting in a race condition), you must make sure to "lock" the state before accessing it, and then "unlock" it once you are done. This ensures that at any given time, at most one thread can

access the state, to avoid any indeterminant behavior. Locking and unlocking of state can be done using a mutex. "`tinytcp_conn_t`" structure defines a mutex (of type spinlock), called "`mtx`", that can be used to lock any "`tinytcp_conn_t`" attribute, if there is a possibility of a race condition. Look at "`tinytcp_close_conn()`" function in "`tinytcp.c`" for an example of spinlock locking / unlocking.

Note: Accessing the same state from multiple threads does not automatically result in a race condition, for example, the state might be inside some "if" conditions in each thread, such that only one of those conditions can be true at a given time. You should only use locks if there is a possibility of a race condition. Using locks unnecessarily may affect your code's performance, as locking has overheads.

9 Frequently Asked Questions

9.1 How to get rid of error "bind: address already in use"?

Wait for a few seconds, and it will go away. Or, run the following command:

```
kill $(lsof -i :53005 | awk 'NR>1 {print $2}')
```

9.2 How to handle out-of-order packets at the receiver?

Maintain a separate data structure to buffer and order packets, and then feed the packets from this new data structure to the provided receive ring buffer in-order. This is what a typical TCP implementation would do (although the details may vary based on implementation).

Or, simply drop any out-of-order packets at the receiver, but still send back the appropriate ACK to generate 3 Dup ACKs, and rely on Go-Back-N at the sender after it receives 3 Dup ACKs to re-transmit dropped packets. This simplifies implementation, at the cost of efficiency.

9.3 What is the optimal sender window size for this lab?

Sender window size (i.e., packet sending rate) will depend upon how fast the receiver can process packets (i.e., your implementation of "`handle_recv_from_network()`"). The sender must not send packets at a faster rate than the receiver could process. Or else, the queue inside link delay buffers will overflow, resulting in "`ALERT!!`" messages on stdout. Thus, if your implementation is slow, your sender window size (and hence the sending rate) will be small. An optimal implementation of "`handle_recv_from_network()`" will ensure that the sender can continuously keep sending packets back-to-back (i.e., at maximum sending rate) without ever overflowing the link delay buffers.

10 Submission

You are required to submit a single file "`handle.c`" on Brightspace.
Do not submit a compressed (e.g., .zip) file.

Note: Your final code must not print any custom / debug statements to the terminal other than what the starter code already prints. **Violations of this guideline will result in a 10% grade penalty.**

IMPORTANT: Configuration parameter values `MSS`, `DELAY`, `RTO`, and `CAPACITY` in the header files, as well as the client and server port number values are subject to change. Your code must not make any assumptions about these values. However, you can assume that `MAX_CONNS` value will always be 5 and `TINYTCP_HDR_SIZE` will always be 20.

```
vagrant@ubuntu:/vagrant/assignment3-sol$ ./bin/tinytcp client 30 file3.txt file1.txt file2.txt
### started recv thread
### started send thread
### started link delay thread

SYN sending (src_port:3001 dst_port:5001 seq_num:466 ack_num:0)
=
SYN-ACK recvd (src_port:5001 dst_port:3001 seq_num:499 ack_num:467)

ACK sending (src_port:3001 dst_port:5001 seq_num:467 ack_num:500)
=
connection established...sending file file3.txt


SYN sending (src_port:3002 dst_port:5002 seq_num:412 ack_num:0)
=
SYN-ACK recvd (src_port:5002 dst_port:3002 seq_num:306 ack_num:413)

ACK sending (src_port:3002 dst_port:5002 seq_num:413 ack_num:307)
=
connection established...sending file file1.txt


SYN sending (src_port:3003 dst_port:5003 seq_num:222 ack_num:0)
=
SYN-ACK recvd (src_port:5003 dst_port:3003 seq_num:724 ack_num:223)

ACK sending (src_port:3003 dst_port:5003 seq_num:223 ack_num:725)
=
connection established...sending file file2.txt

================================================
FIN sending (src_port:3002 dst_port:5002 seq_num:781 ack_num:307)
=
FIN-ACK recvd (src_port:5002 dst_port:3002 seq_num:307 ack_num:782)

ACK sending (src_port:3002 dst_port:5002 seq_num:782 ack_num:308)
=
file file1.txt sent...connection terminated

================================================================================================
================================================================================================
================================================================================================
================================================================================================
FIN sending (src_port:3001 dst_port:5001 seq_num:45415 ack_num:500)
=
FIN-ACK recvd (src_port:5001 dst_port:3001 seq_num:500 ack_num:45416)

ACK sending (src_port:3001 dst_port:5001 seq_num:45416 ack_num:501)
=
file file3.txt sent...connection terminated

==
FIN sending (src_port:3003 dst_port:5003 seq_num:5599 ack_num:725)
=
FIN-ACK recvd (src_port:5003 dst_port:3003 seq_num:725 ack_num:5600)

ACK sending (src_port:3003 dst_port:5003 seq_num:5600 ack_num:726)
=
file file2.txt sent...connection terminated

total time: 10 sec
total bytes sent: 246547
SUCCESS: Files sendfiles/file3.txt and recvfiles/file3.txt are identical!
SUCCESS: Files sendfiles/file1.txt and recvfiles/file1.txt are identical!
SUCCESS: Files sendfiles/file2.txt and recvfiles/file2.txt are identical!
```

Figure 4: A sample client side output.

```
vagrant@ubuntu:/vagrant/assignment3-sol$ ./bin/tinytcp server 30
### started recv thread
### started send thread
### started link delay thread

SYN recvd (src_port:3001 dst_port:5001 seq_num:466 ack_num:0)

SYN-ACK sending (src_port:5001 dst_port:3001 seq_num:499 ack_num:467)

ACK recvd (src_port:3001 dst_port:5001 seq_num:467 ack_num:500)

connection established...receiving file file3.txt


SYN recvd (src_port:3002 dst_port:5002 seq_num:412 ack_num:0)

SYN-ACK sending (src_port:5002 dst_port:3002 seq_num:306 ack_num:413)

ACK recvd (src_port:3002 dst_port:5002 seq_num:413 ack_num:307)

connection established...receiving file file1.txt


SYN recvd (src_port:3003 dst_port:5003 seq_num:222 ack_num:0)

SYN-ACK sending (src_port:5003 dst_port:3003 seq_num:724 ack_num:223)

ACK recvd (src_port:3003 dst_port:5003 seq_num:223 ack_num:725)

connection established...receiving file file2.txt

=======
FIN recvd (src_port:3002 dst_port:5002 seq_num:781 ack_num:307)

FIN-ACK sending (src_port:5002 dst_port:3002 seq_num:307 ack_num:782)

ACK recvd (src_port:3002 dst_port:5002 seq_num:782 ack_num:308)

file file1.txt received...connection terminated

================================================================================================================
FIN recvd (src_port:3001 dst_port:5001 seq_num:45415 ack_num:500)

FIN-ACK sending (src_port:5001 dst_port:3001 seq_num:500 ack_num:45416)

ACK recvd (src_port:3001 dst_port:5001 seq_num:45416 ack_num:501)

file file3.txt received...connection terminated


FIN recvd (src_port:3003 dst_port:5003 seq_num:5599 ack_num:725)

FIN-ACK sending (src_port:5003 dst_port:3003 seq_num:725 ack_num:5600)

ACK recvd (src_port:3003 dst_port:5003 seq_num:5600 ack_num:726)

file file2.txt received...connection terminated
```

Figure 5: A sample server side output.