CSCI4430 Data Communication and Computer Networks (2016-2017 2nd term)
Project 1 - µTCP Protocol

Deadline: 23:59 22nd March 2017

1. Introduction

In this project you are required to implement an internet protocol library such that you can complete a client program and a server program where the client program can send a file to the server program through a "stripped-down" TCP protocol called micro TCP (µTCP). Similar to a typical TCP, µTCP is a reliable data transfer protocol. µTCP guarantee its reliability under a lossy network through (1) having 3-way handshake during establishing connection, (2) having 4-way handshake during tearing down connection and (3) having a simple loss packet retransmission mechanism.

2. Basic flow of server and client program

You are required to complete a client program and server program which make use of the µTCP protocol library you have implemented. Since the main purpose of this project is to implement a simplified TCP, you are NOT allowed to invoke any TCP related function calls. Rather, we are going to implement a set of functions which emulates these function calls in a limited fashion. The basic flow of the client program and the server program is shown in figure 1 and 2 below:

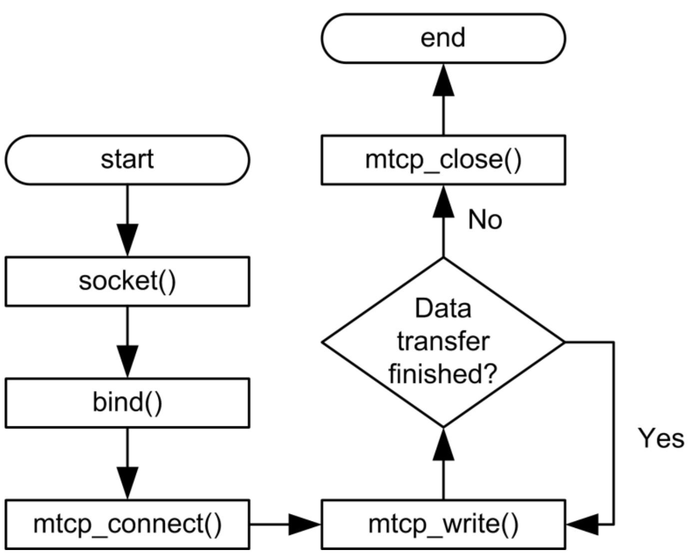


Figure 1: The basic flow of the client process.

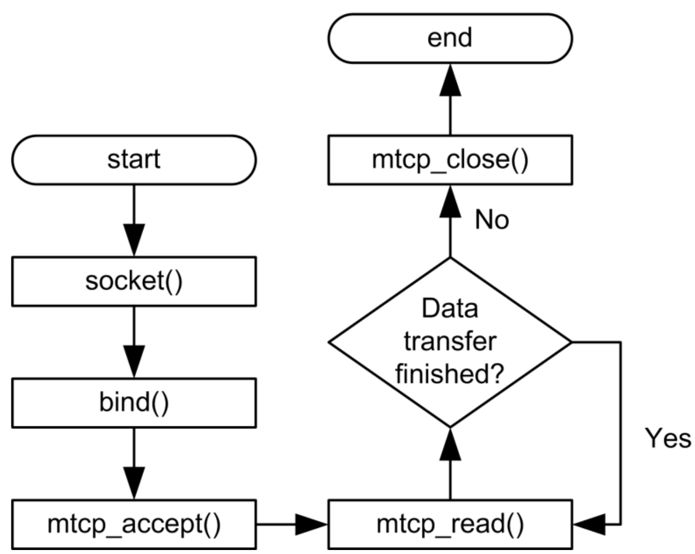


Figure 2: The basic flow of the server process.

3. Data Flow Overview

The connection between the client and the server is unidirectional as shown in figure 3 below. Under the µTCP protocol, the client can send data to the server but the server does not send data to the client. In the µTCP protocol, µTCP packets are sent between the client and the server as UDP packets. The structure of a µTCP packet will be discussed in section 4. You would **NOT GAIN ANY POINTS** for this assignment if you are implementing the assignment using the TCP protocol.

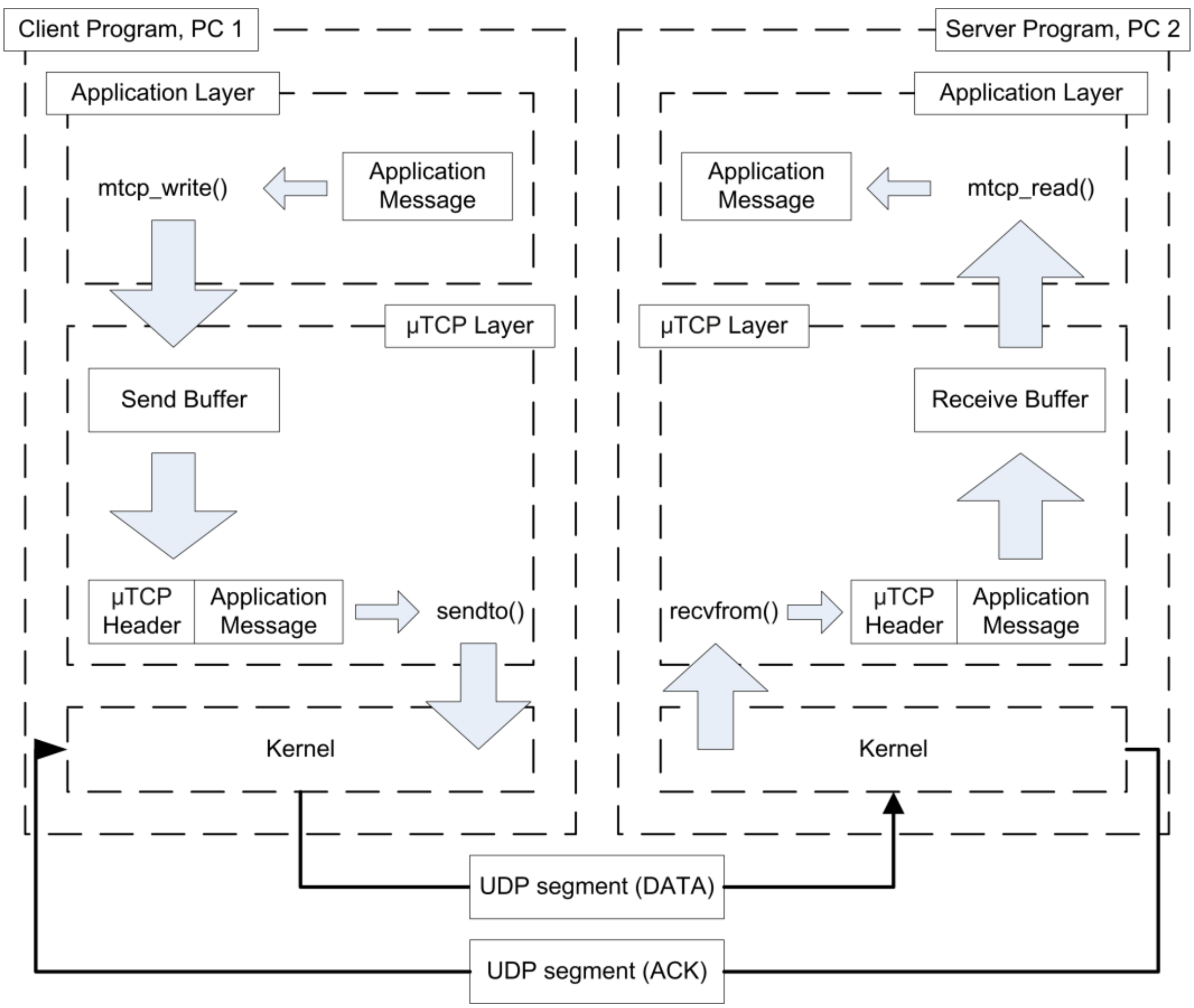


Figure 3: An overview of the data flow between two programs after the connection is establish

4. Definition of µTCP Packet

For each UDP packet sent under µTCP protocol, you have to insert a µTCP header before each application-layer message. The structure of a µTCP packet is shown in figure 4 and all possible values for 4 bit integer variable "type" in the µTCP header is shown in Table 1.

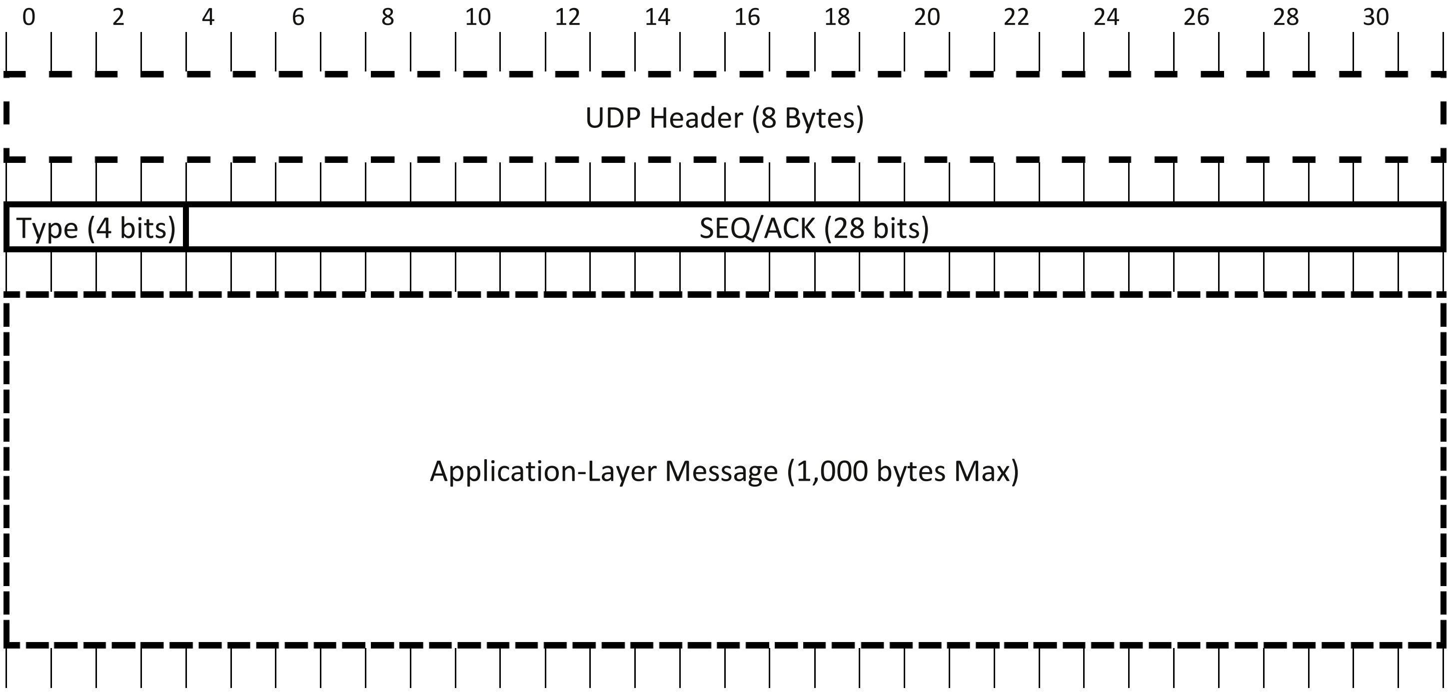


Figure 4: The structure µTCP packet where µTCP header is highlighted with solid borders.

Value	Name	Description
0	SYN	1st packet in the 3-way handshake; no application data.
1	SYN-ACK	2nd packet in the 3-way handshake; no application data.
2	FIN	1st packet in the 4-way handshake; with application data.
3	FIN-ACK	2nd and 3rd packet in the 4-way handshake; no application data.
4	ACK	An acknowledgement packet; no application data.
5	DATA	A packet with application data.

Table 1: Definition of 6 different possible values of "type" variable in the µTCP header.

5. Assumption and Regulations

- The server can only be connected from at most one client and the client can only be connected to at most one server.
- The connection between the client and server is unidirectional. Only the client can send data packet to the server.
- The client sends data packet to the server in a sequential manner and without any pipelining.
- Any out-of-order µTCP packet should be discarded.
- There is no sequence number wrap-around.
- It is assumed that the server is always ready to respond to the 4 way handshake. Therefore, the 2nd and 3rd handshake messages are merged in our implementation (i.e. Type FIN-ACK).
- During the 3-way handshake and 4-way handshake process, no packet will be dropped.
- Since it is assumed that there will be no packet loss during 4-way handshake, the client program doesn’t need to wait for a specific time for retransmissing ACK or FIN packet after the last ACK.
- The client must send some data to the server after 3-way handshake is completed.
- The 4-way handshake can only be initiated by the client program.
- The server and client will not close unless the 4-way handshake is completed.

- The mtcp_write() of the client program will never block.
- After the connection is established, the maximum size of the file to be sent is bounded by $2^{28} - 3$ bytes.
- The maximum µTCP packet size (MSS) is bounded to be 1004 byte. It includes both the header of the µTCP (4 bytes) and the application-layer message (at most 1,000 bytes).
- The timeout for retransmission is 1 second.
- All µTCP functions are implemented in the user space, not in the kernel space.
- Once the server has established a connection with client, data will be received and acknowledged even mtcp_read() is not being called. (i.e. data received are stored to the receive buffer automatically by a separate receiving thread).
- Once the client has established a connection with server and the sending buffer is not empty, the client will keep sending data in the sending buffer to the server even long after mtcp_write() is called. (i.e. data stored in the send buffer will be sent automatically by a separate sending thread).

6. Specification on µTCP functions

6.1. Server-side Functions

6.1.1. mtcp_accept()

You should finish the implementation of this function in the µTCP protocol API such that it can act like listen() and accept() of a typical socket programming API. However, this function will not produce any extra file descriptors since one server program can only be connected from at most one client program. Moreover, it is a blocking call and it waits until the 3-way handshake has been completed before it returns. The definition of the function prototype is given in table 3.

Function Prototype	
void mtcp_accept (int sock_fd, struct sockaddr_in *client_addr)	
Argument	**Description**
int sock_fd	It is the file descriptor of the opened **UDP** socket. It is assumed that the value of this argument is also correct. No error checking is needed.
struct sockaddr_in *server_addr	This is a pointer to the address structure for the client ad- dress. Note that this is a **value-result variable** and it is expected that by the time this function returns, the client IP address and port number obtained will be written to this variable.

Table 3: Definition of function prototype of mtcp_accept()

6.1.2. mtcp_read()

You should finish the implementation of this function in the µTCP protocol API such that it can act like read() of a typical socket programming API. If there is data in the receive buffer, the call will write number Min(Num-of-bytes-stored-in-the-received-buffer, len-of-buffer) to the buffer given in the argument. On the other hand, if the receive buffer is empty, the call is blocked until there is data received. The definition of the function prototype and return value is given in table 4 and table 5 respectively.

Function Prototype	
int mtcp_read(int sock fd, char *buffer, int len);	
Argument	**Description**
int sock_fd	It is the file descriptor of the opened **UDP** socket. It is assumed that the value of this argument is correct and no error checking is needed.
char *buffer	It is a pointer to an **allocated buffer**. This is a **value-result variable** and it is expected that by the time this function returns, the data read from the sender will be stored into this allocated buffer.
int len	It is the amount of data, measured in terms of bytes that the process wants to read. It is assumed that "len" is always smaller than or equal the allocated size of "buffer".

Table 4: Definition of function prototype of mtcp_read()

Return Value	Meaning
Positive integer	The data length measured in terms of number of bytes copied to from receive buffer to local buffer
Zero	The return value will be zero iff both of the following two conditions are satisfied: 1. The receive buffer has no data 2. The 4 way handshake has been triggered. Note: ▪ If there is some unread data in the receive buffer after 4 way handshake, the mtcp read() call should not return zero. ▪ When the client has started the 4 way handshake, the handshake cannot be undone and it is clear that no further inbound data would be coming. Therefore, when the receive buffer is empty, the mtcp_read() call should return zero.
Negative integer	The underlying recvfrom() system call returns −1 The process invokes mtcp_read() after calling mtcp_close().

Table 5: Definition of return value of mtcp_read()

6.1.3. mtcp_close()

You should finish the implementation of this function in the µTCP protocol API such that it can release the memory and port allocated by the µTCP protocol API. It will wait for the sending thread and receiving thread to be terminated before releasing any memory and ports.

Function Prototype
void mtcp_close(void);

Table 9: Definition of return value of mtcp_close()

6.2. Client-side Functions

6.2.1. mtcp_connect()

You should finish the implementation of this function in the µTCP protocol API such that it can act like connect() of a typical socket programming API. It starts a µTCP connection and performs the 3-way handshake. Moreover, it is a blocking call and will not stop until the 3-way handshake is complete. Its function prototype is given in table 6.

Function Prototype	
void mtcp_connect(int sock_fd, struct sockaddr_in *server addr)	
Argument	**Description**
int sock_fd	It is the file descriptor of the opened **UDP** socket. It is assumed that the value of this argument is always correct. No error checking is needed.
struct sockaddr_in *server_addr	It is a pointer to the address structure for the server address. It is assumed that the value of this argument is always cor- rect. No error checking is needed.

Table 6: Definition of function prototype of mtcp_accept()

6.2.2. mtcp_write()

You should finish the implementation of this function in the µTCP protocol API such that it can act like write() of a typical socket programming API. When mtcp_write() is invoked, new application data will be copied to the send buffer of the µTCP layer (see Figure 2). Then, the call returns. Meanwhile, the µTCP layer then asynchronously sends the data to the server side. The definition of the function prototype and return value is given in table 7 and table 8 respectively.

Function Prototype	
int mtcp_write(int sock fd, char *buffer, int len);	
Argument	**Description**
int sock_fd	It is the file descriptor of the opened **UDP** socket. It is assumed that the value of this argument is always correct. No error checking is needed.
char *buffer	It is a pointer to an allocated buffer. The content of the buffer will be sent to the receiver by this function.
int len	It is the amount of data, measured in terms of bytes that the process wants to send. It is assumed that "len" is always smaller than or equal the allocated size of "buffer".

Table 7: Definition of function prototype of mtcp_write()

Return Value	Meaning
Positive integer	The data length measured in terms of number of bytes copied from local buffer to send buffer
Zero	The 4 way handshake has been triggered
Negative integer	The underlying sendto() system call returns −1 The process invokes mtcp_write() after calling mtcp_close().

Table 8: Definition of return value of mtcp_write()

6.2.3. mtcp_close()

You should finish the implementation of this function in the µTCP protocol API such that it can act like close() of a typical socket programming API. If the connection is not closed, it starts the 4-way handshake. Otherwise, it does nothing. Note that you can safely assume that the client will always invoke this call before the server in order to guarantee that it is always the client which triggers the 4-way handshake. Moreover, it is a blocking call when the connection is not closed and does not return until the 4-way handshake process has been finished. After the process is completed, both programs should close the UDP socket. Its function prototype is given in table 9.

Function Prototype
void mtcp_close(void);

Table 9: Definition of return value of mtcp_close()

7. Packet flow of 3-way handshake

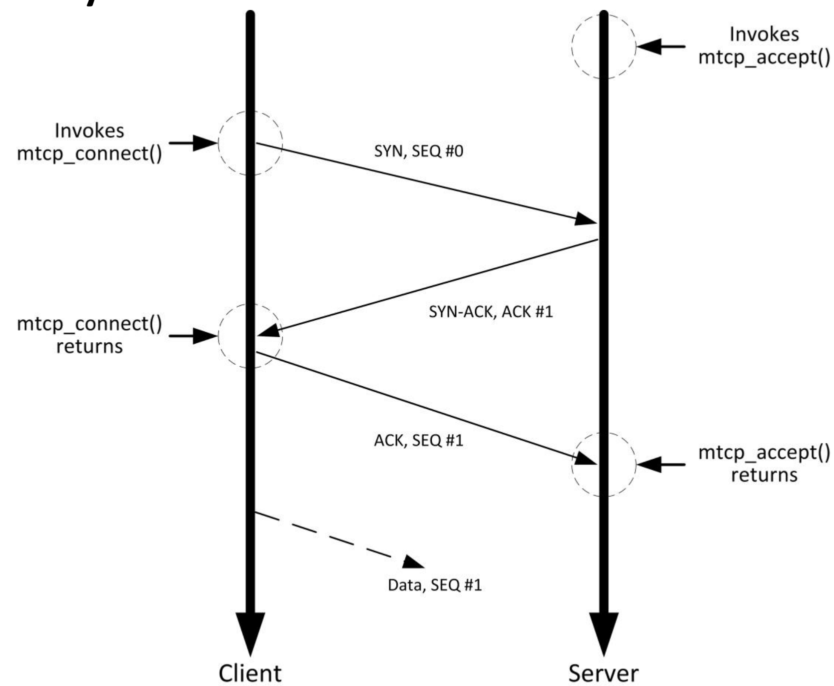


Figure 5: Illustration of the flow of 3-way handshake.

8. Packet flow of 4-way handshake

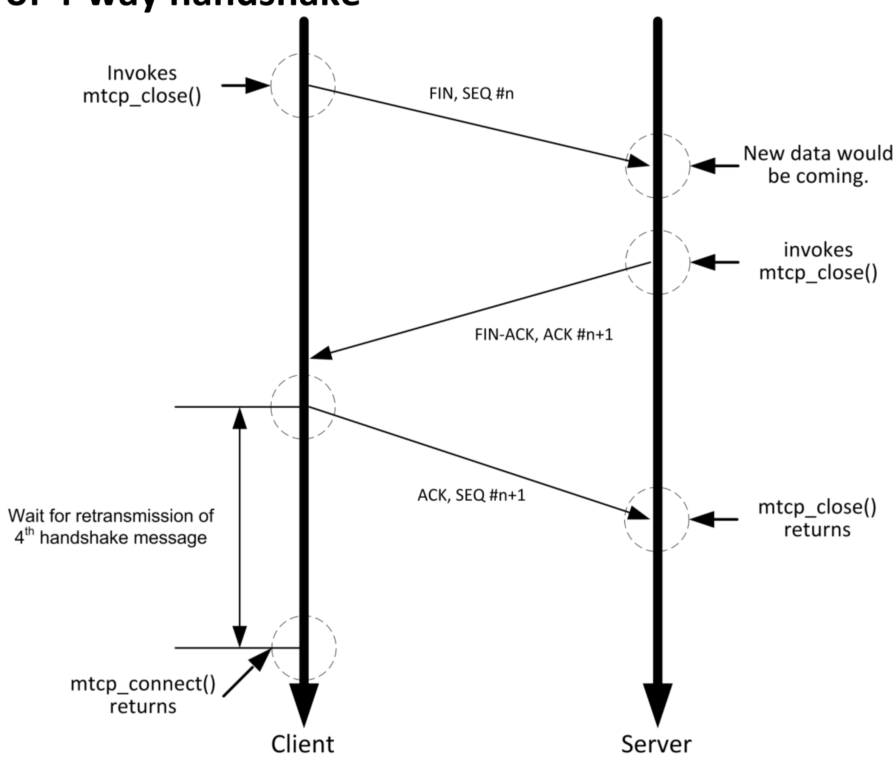


Figure 6: Illustration of the flow of 4-way handshake.

9. Specification on Threading

You should implement both the client program and the server program as multi-threading processes. Both client program and server program should have 3 threads namely application thread, receiving thread, sending thread. The interaction between these three threads under different scenario is shown below:

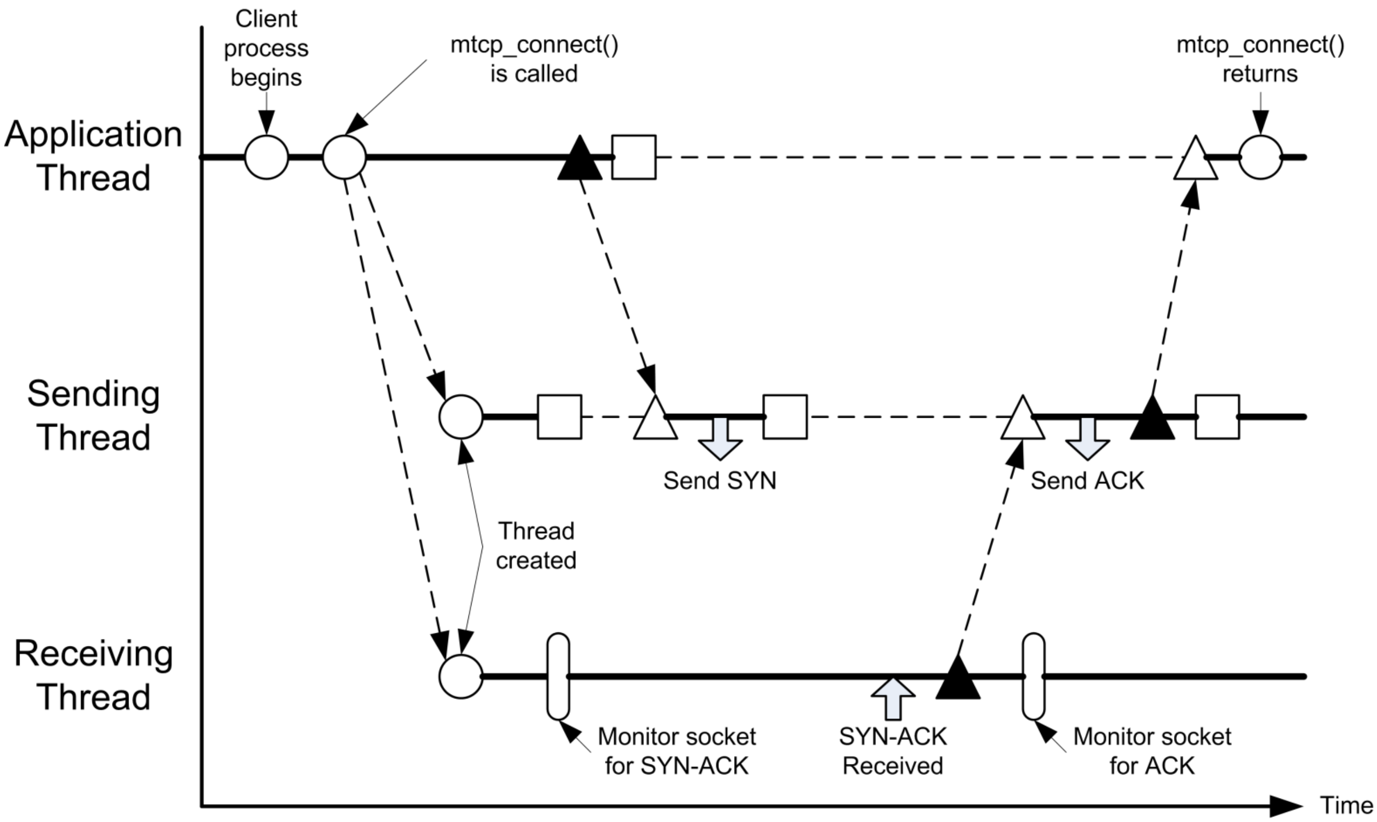
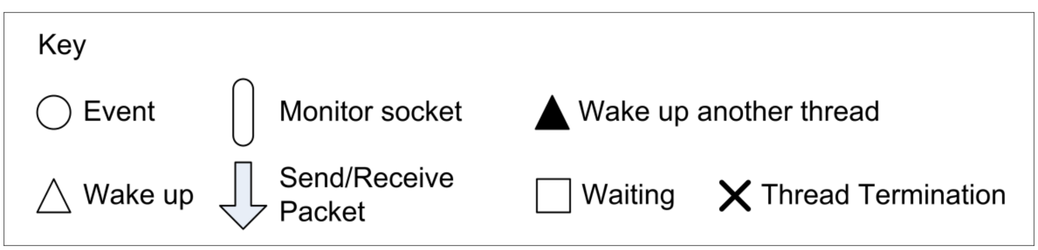


Figure 7: Interactions among the threads of the client program when mtcp_connect() is invoked.



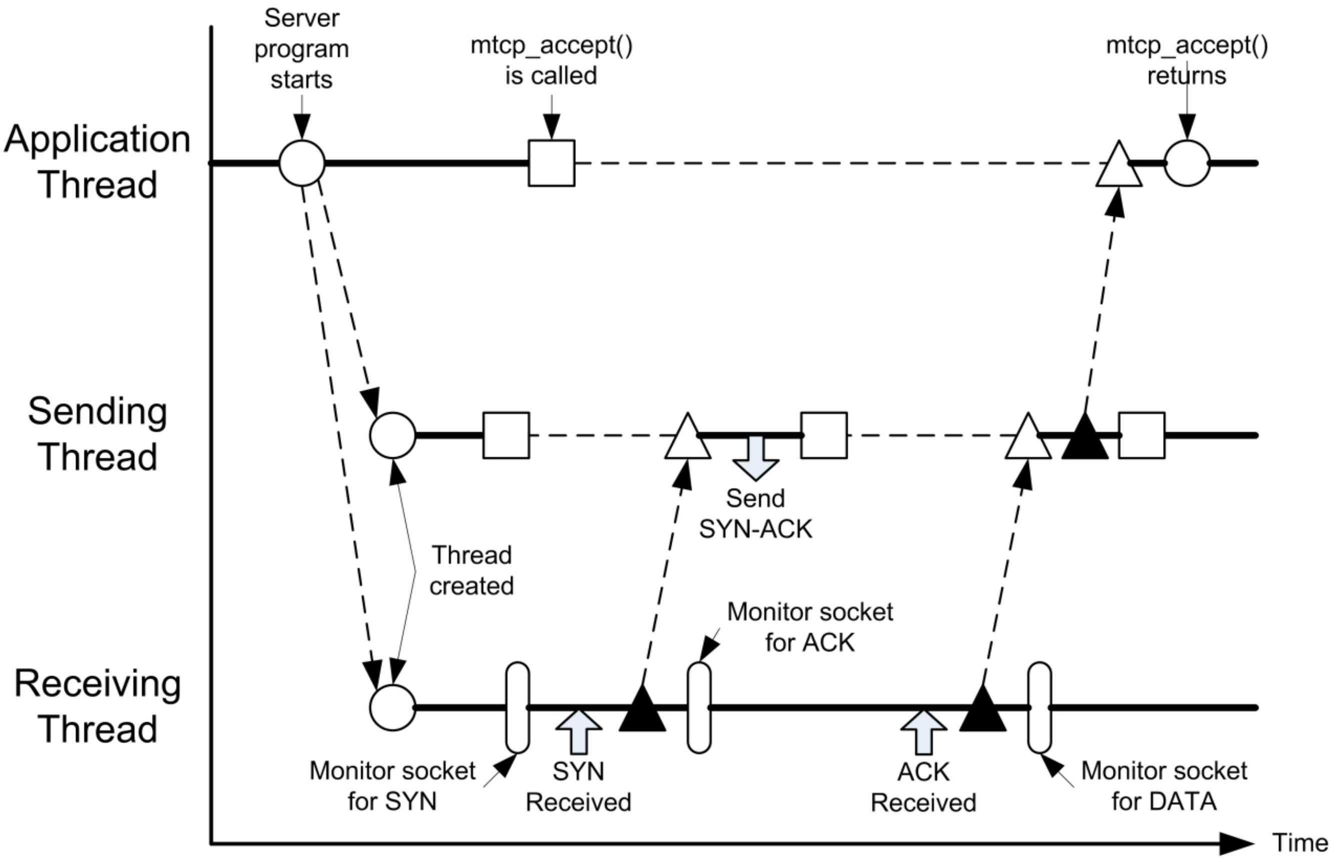
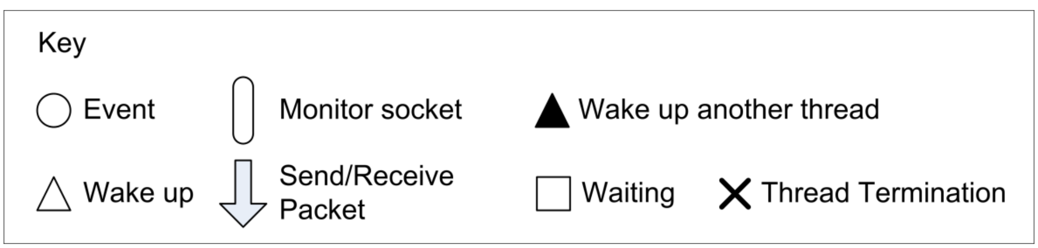


Figure 8: Interactions among the threads of the server program when mtcp_accept() is invoked.



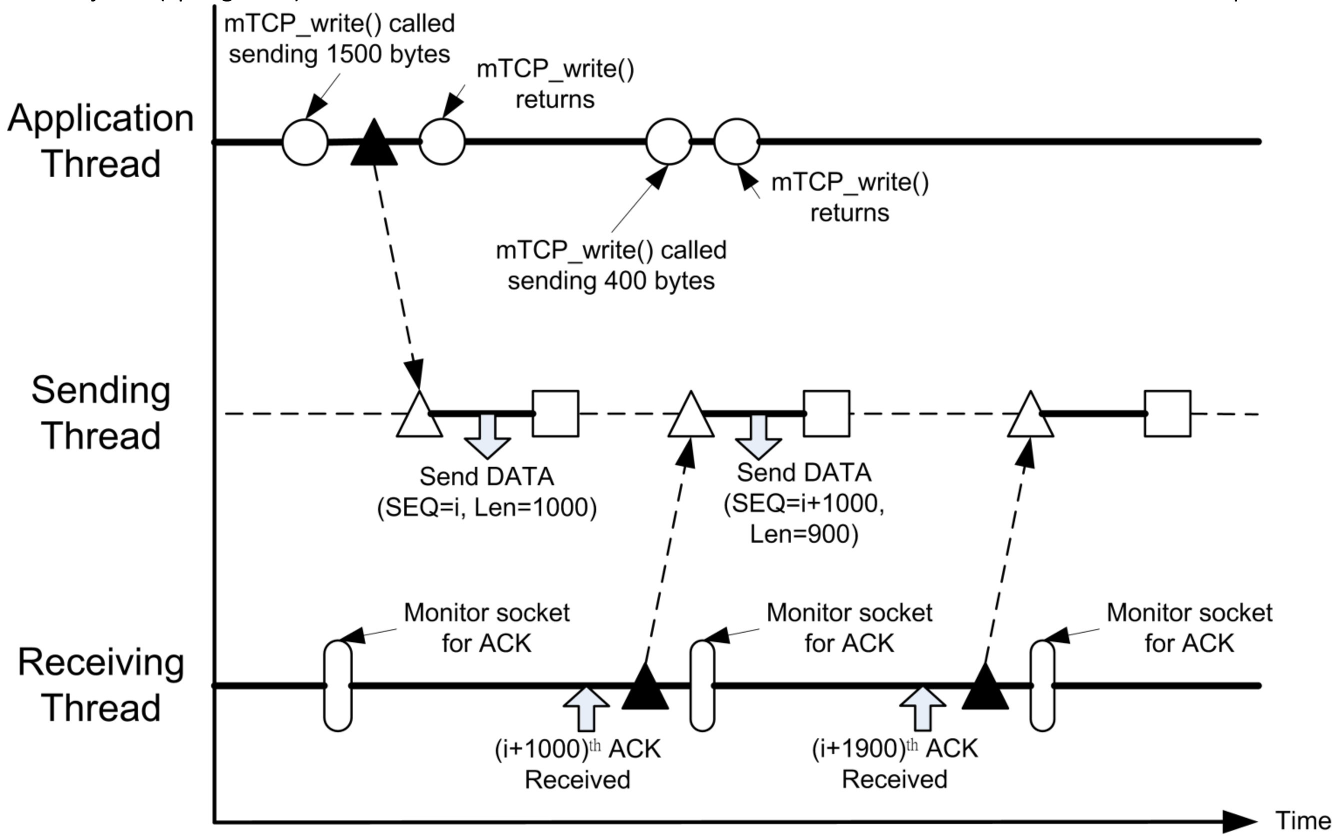
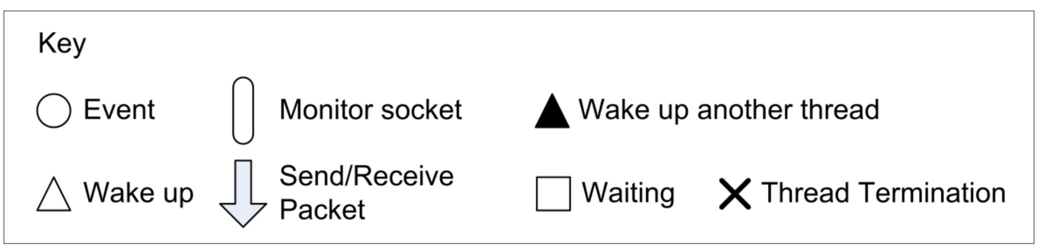


Figure 9: The interactions among the 3 threads when the client program has involved mtcp_write() twice to send two message. As shown in this figure, the message may not sent immediately right after mtcp_write is involved. The two messages are stored in a buffer and wait to be sent by the sending thread. Moreover, the sending thread will sent the messages stored in the buffer through a number of packets regardless when the application thread involves mtcp_write().



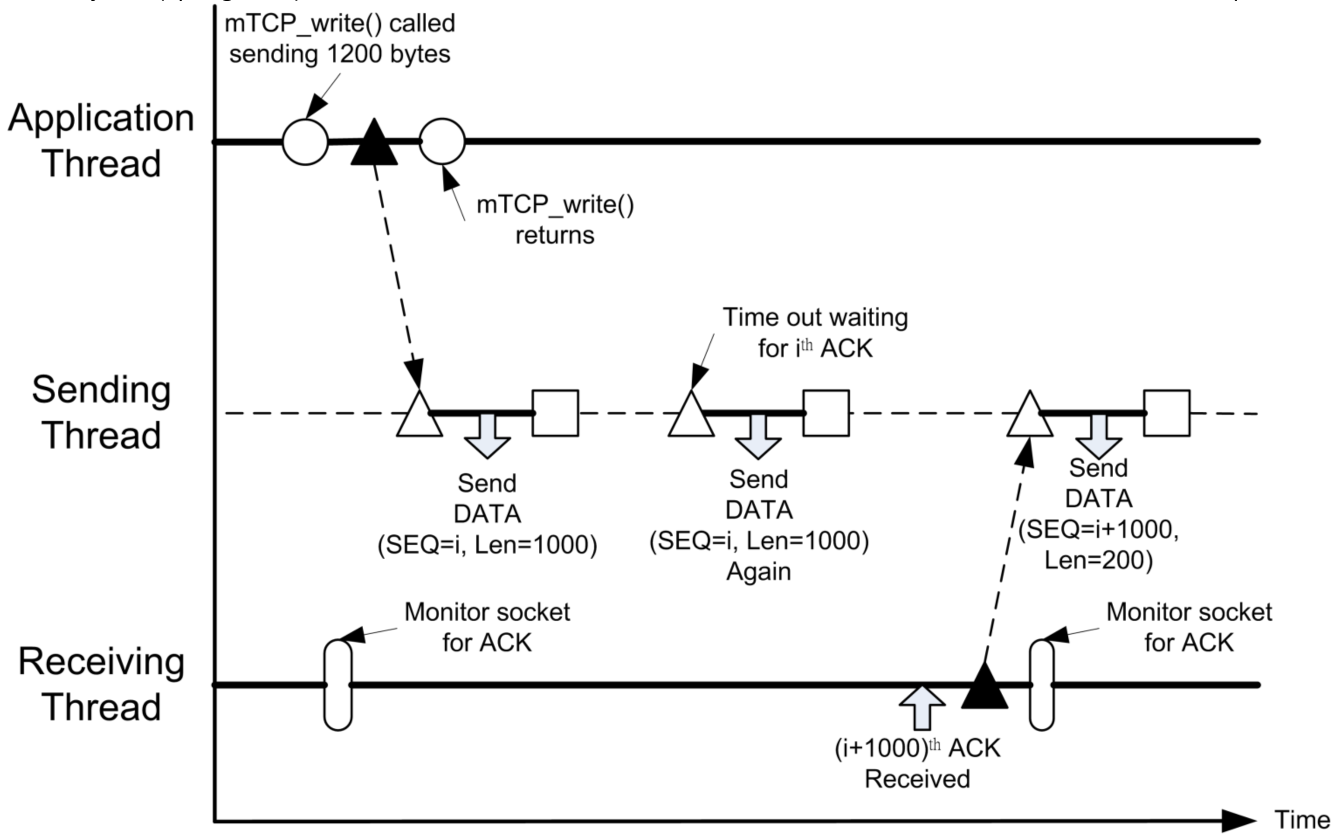
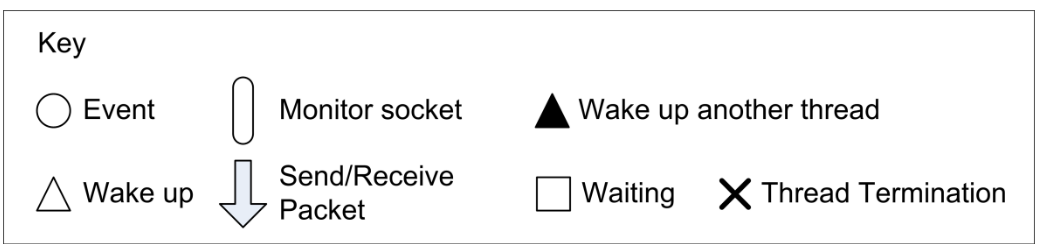


Figure 10: The interactions among the threads when (1) the client program invokes rdtp write() and (2) the first sent packet is lost. If there is a packet lost (i.e. The correct ACK packet is not received within the time-out period), the sender thread will re-transmit it.



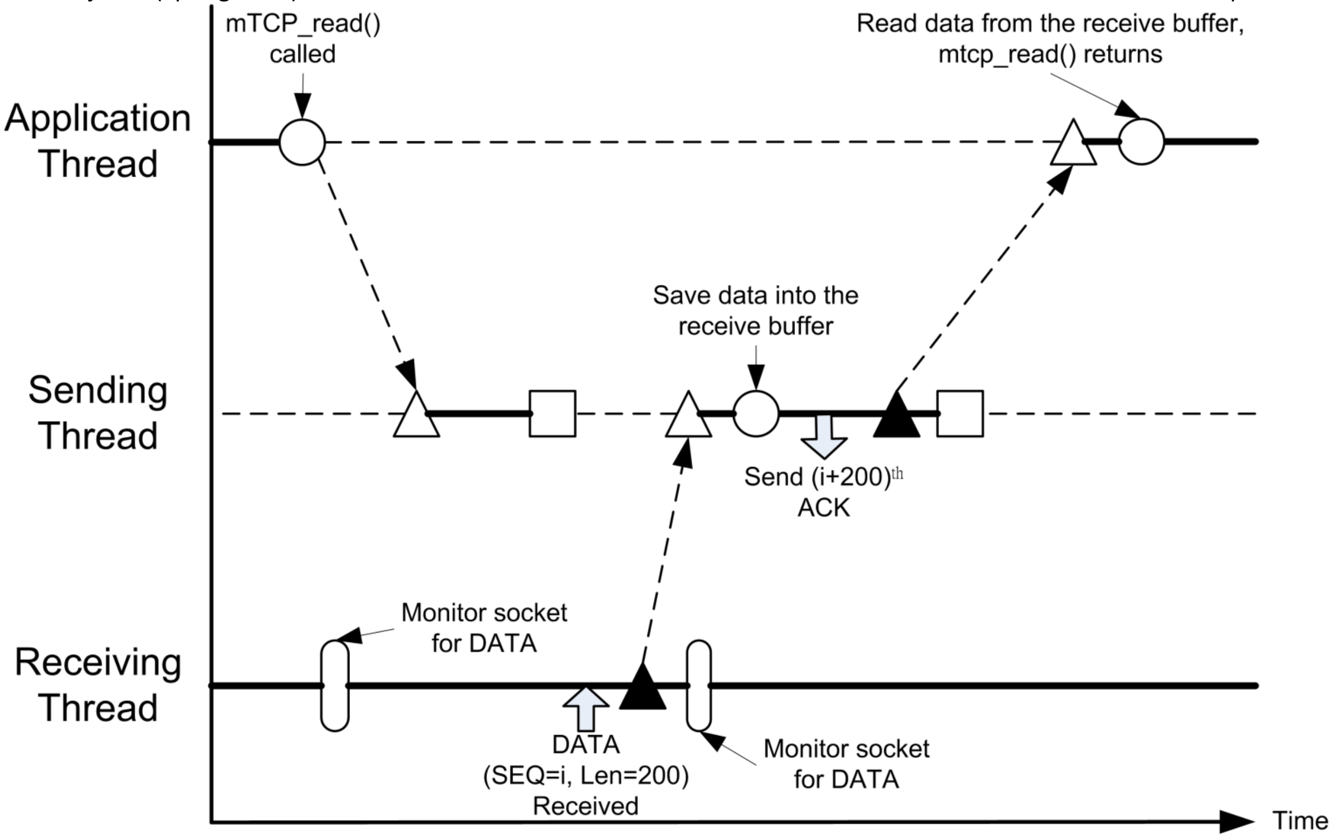
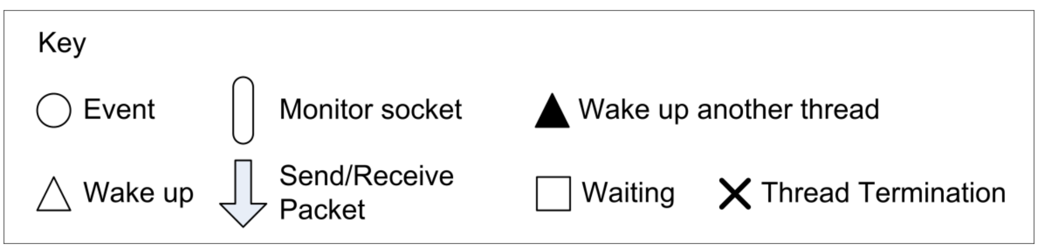


Figure 11: The interactions among the threads when the server program invokes rdtp_read(). It returns whenever a DATA package arrives.



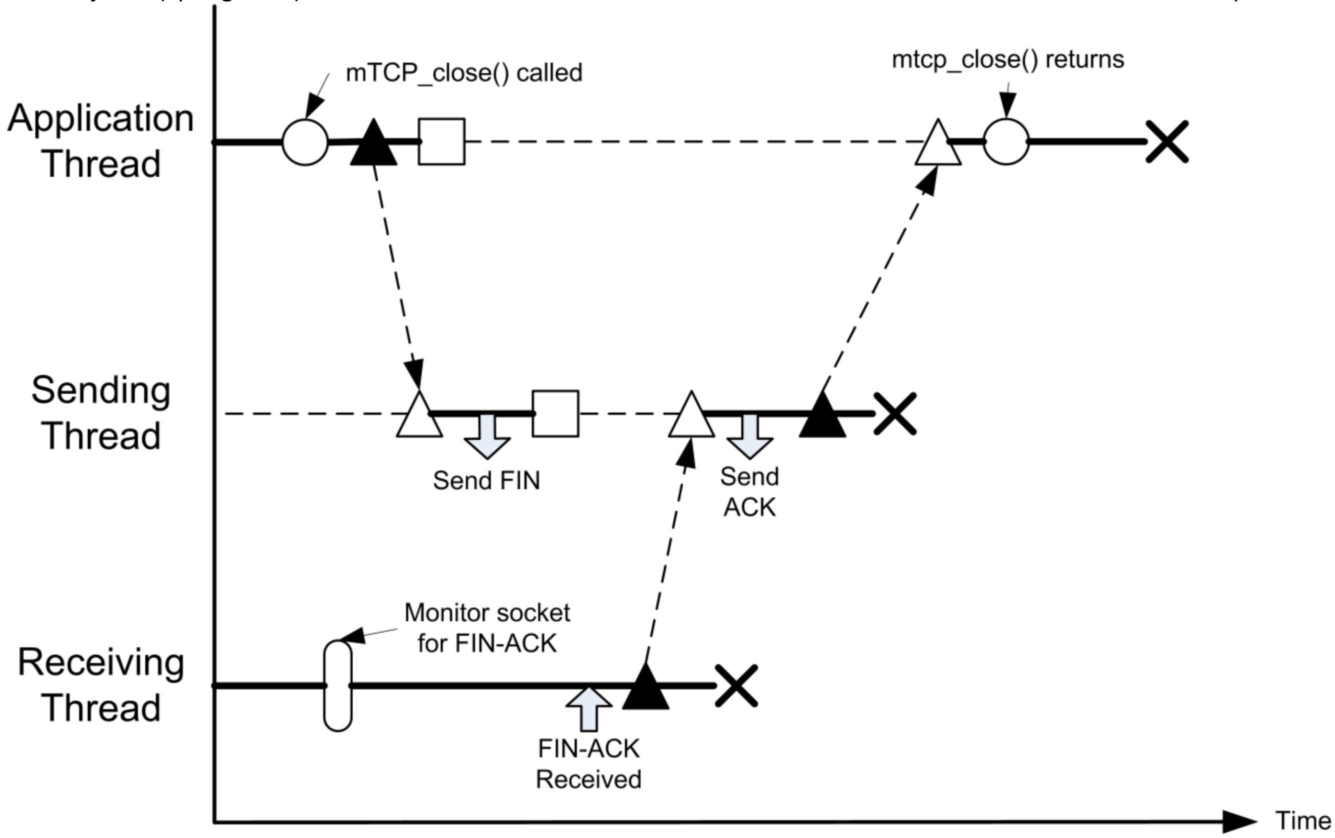
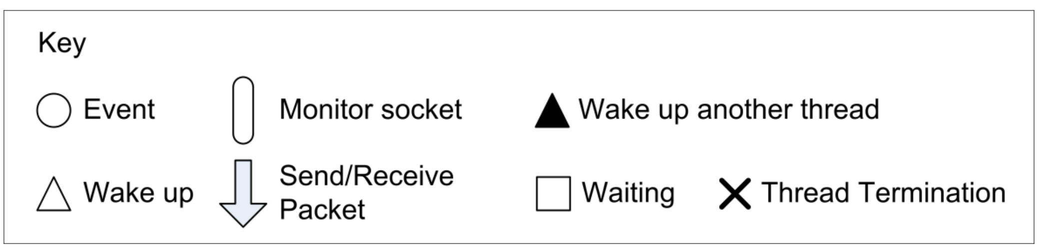


Figure 12: The interactions among the threads when the client program invokes the rdtp_close() process and the pseudo 4-way handshake process afterward.



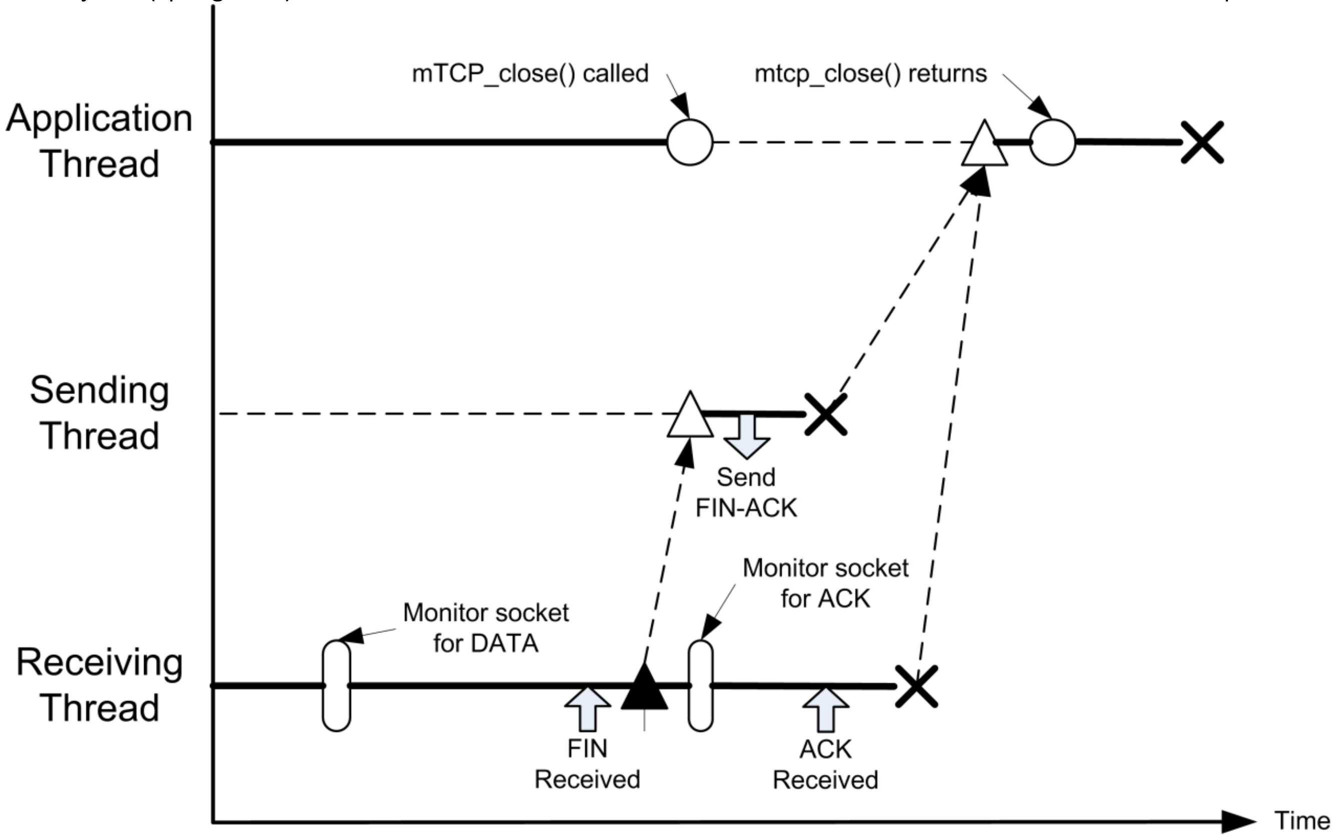
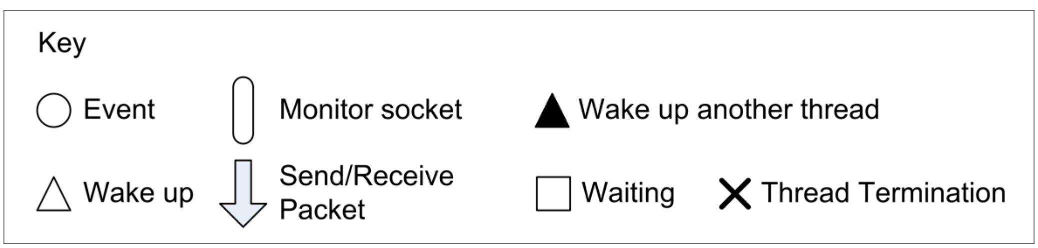


Figure 15: The interactions among the threads when the server program invokes the rdtp_close() process and the pseudo 4-way handshake process afterward.



10. Demonstration

- All groups need to sign up for a demonstration. The registration page would be posted on the course website later.
- All group members should attend the demonstration.
- The duration for the demonstration for each group is about 20 minutes.
- Your program will be tested in the virtual machines (VMs) with the same configuration as the VMs given to you.
- The dataset used in the demonstration may be different from the dataset provided.
- Packets between client and server program may be dropped randomly during the project demonstration.

11. Submission

- Submit a ZIP file (one copy for each group) to the collection box at eLearning platform. The ZIP file should consist of all your source codes and a README file (README.txt), which contains:
  - The group number of your group
  - The name and the student ID of each group members of your group
  - List of files with description
  - Methods of compilation and execution