COMP3234/CSIS0234 Computer and Communication Networks

Programming Project

Total 16 points

Version 1.0

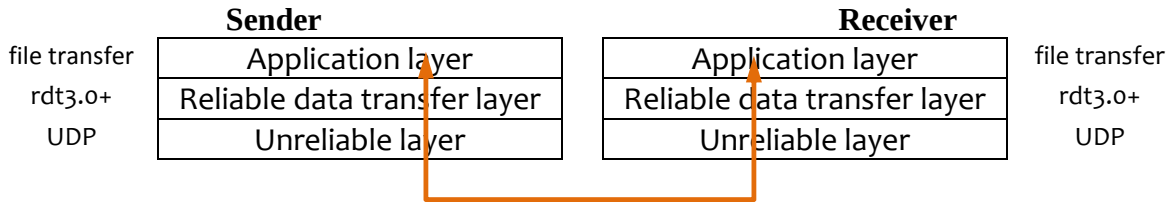
Final Due Date: 5pm April 15, 2015

Hand-in the assignment via the Moodle System.

Stop-and-Wait (rdt3.0) ARQ

Overview

In this project, you are going to design and implement a reliable data transfer layer using a modified version of **Stop-and-Wait** (rdt 3.0) protocol. This RDT layer supports **connectionless** reliable **duplex** data transfer on top of unreliable UDP. Our reliable communication system has three layers. They are the Application layer, Reliable data transfer layer, and Unreliable layer.



The Application layer invokes the service primitives provided by RDT layer in order to send/receive messages to/from remote application processes. Upon receiving message from Application layer, the RDT layer encapsulates the application message with control header to form packet, and passes the packet to Unreliable layer by invoking the service primitives exported by the Unreliable layer. The Unreliable layer is responsible for sending the packets to the Unreliable layer of a remote host by means of UDP datagram. To simulate transmission errors, the Unreliable layer may randomly discard or corrupt datagrams.

Objectives

1. An assessment task related to ILO4 [Implementation] – "be able to demonstrate knowledge in using Socket Interface to design and implement a reliable data transfer protocol".
2. A learning activity to support ILO1, ILO2a, & ILO4.
3. The goals of this programming project are:
   - to get better understanding of the principles behind Stop-and-Wait protocol;
   - to gain hands-on experience in implementing a reliable protocol;
   - to understand the performance difference between a pipelined protocol and the Stop-and-Wait protocol;
   - to gain experience in using UNIX socket functions to implement a real-life protocol.

Specification

This programming project consists of three parts:

- Part 1 – **Assume** UDP is reliable (i.e., the Unreliable layer is indeed "reliable"), you are going to implement the RDT layer (rdt-part1.h) directly on top of UDP. This is true when running the application within the virtual machine or in local area network.
- Part 2 – Assume UDP is unreliable, which suffers with packet losses and transmission errors. You are going to implement the RDT layer (rdt-part2.h) using Stop-and-Wait protocol directly on top of UDP but with a wrapper function ***udt_send()*** to simulate the loss and corruption.
- Part 3 - To improve the performance, you are going to implement a pipelined protocol - ***Extended Stop-and-Wait*** in our RDT layer (rdt-part3.h), which works on top of the unreliable layer.

PART 1 (2 POINTS)

Download the compressed file – Part1.zip from the Course web site at moodle.hku.hk. This zip file contains four files:

1. A Makefile to compile the testing application programs.
2. A header file – rdt-part1.h, which defines all functions provided by the RDT layer to the application layer. However, rdt-part1.h file is not completed. ***Your task in this part is to complete rdt-part1.h file.***

   It consists of six functions. They are:

rdt_socket()	Application process calls this function to create the RDT socket.
rdt_bind()	Application process calls this function to specify the IP address and port number used by itself and assigns them to the RDT socket.
rdt_target()	Application process calls this function to specify the IP address and port number used by remote process and associates them to the RDT socket.
rdt_send()	Application process calls this function to transmit a message (up to a limit of 1000 bytes) to targeted (rdt_target) remote process through the RDT socket; when returned from this function, the caller can assume that the message has been successfully delivered to the remote process.
rdt_recv()	Application process calls this function to wait for a message from targeted remote process; the caller will be blocked waiting for the arrival of the message. Upon arrival of a message (of maximum size 1000 bytes), the RDT layer will immediately return the message to the caller.
rdt_close()	Application process calls this function to close the RDT socket.

   As we **assume UDP is reliable**, you do not need to implement any reliable protocol in this part; thus, you can simply ***complete these functions by using standard UNIX socket functions***.
3. Two files – test-client.c and test-server.c. These two programs require the rdt-part1.h header file to compile and work correctly. This is the driver application of your RDT protocol as they make use of the functions provided by the RDT layer. The task of this driver application is to upload a file from the client program to the server program and store the file at a pre-fixed storage location ("Store") at the server side.

   The client program accepts two arguments:

   ./tclient 'server hostname' 'filename'

   The server program accepts one argument

   ./tserver 'client hostname'

Before executing the server program, you should create the directory "Store" under the same directory of the server program. The server program won't start up if this directory is missing. [Note: These two files will also be used in parts 2 and 3; thus, you can safely ignore the settings on PACKET_LOSS_RATE and PACKET_ERR_RATE in this part.]

Testing

Test your programs on the Ubuntu platform (14.04 or 12.04) with three different file sizes: (i) small file (around 30 KB), (ii) moderate size (around 500 KB), and (iii) large file (around 10 MB).

Sample Output

To aid the programming and debugging, please print some log information to the screen to dynamically show what's happening inside the system during the transfer. We suggest you generating some output in response to the send/receive activities. Please refer to Figure 1 of the document "Project-sample-output.pdf" for the sample output.

Submission

You only need to submit the rdt-part1.h header file. Submit the file to the Course's web site at moodle.hku.hk.

PART 2 (7 POINTS)

Download the compressed file – Part2.zip from the Course web site at moodle.hku.hk. This zip file contains six files:

1. A Makefile to compile the testing application programs.
2. Two application files – test-client.c and test-server.c. They are the same as in part 1 except that they require rdt-part2.h header file to compile and work correctly.
3. Two shell scripts – run-client.sh and run-server.sh.

   To simulate packet losses and transmission errors, we use two environment variables (**PACKET_LOSS_RATE** and **PACKET_ERR_RATE**) to control the behavior of the ***udt_send()*** function. By default, these two variables are set to 0.0; thus, the system does not experience any loss or transmission error. If either one is set to be greater than 0, this means we want to simulate the loss or corruption. The higher the number, the higher the chance of the errors.

   Before running the client and server programs, we need to set up these two environment variables. The two scripts are for you to set the environment variables and start the client and server programs. To start the client program, run the run-client.sh script. It accepts 4 arguments:

   sh run-client.sh 'PACKET_LOSS_RATE' 'PACKET_ERR_RATE' 'server hostname' 'filename'

   To start the server program, run the run-server script. It accepts 3 arguments:

   sh run-server.sh 'PACKET_LOSS_RATE' 'PACKET_ERR_RATE' 'client hostname'
4. The header file – rdt-part2.h, which defines all functions provided by the RDT layer to the application layer as well as two utility functions – *udt_send()* and *checksum()*. However, rdt-part2.h file is not completed. ***Your task in this part is to complete rdt-part2.h file.***

   Same as rdt-part1.h, the service interface consists of six functions: rdt_socket(), rdt_bind(), rdt_target(), rdt_send(), rdt_recv() and rdt_close(). As we still make use of UDP to carry the file data, you can reuse your implementation of rdt_socket(), rdt_bind() and rdt_target() in part1. For rdt_send(), rdt_recv and rdt_close(), you need to implement the Stop-and-Wait (rdt3.0) reliable logic in these functions.

rdt_send()	Application process calls this function to transmit a message (up to a limit of 1000 bytes) to targeted remote process through the RDT socket. This function will return only when it knows that the application message has been successfully delivered to remote process.
rdt_recv()	Application process calls this function to wait for a message from targeted remote

process; the caller will be blocked waiting for the arrival of message. Upon arrival of a message (of maximum size 1000 bytes), the RDT layer will immediately return the message to the caller.

rdt_close() The behavior of this function is slightly different from part 1. Application process calls this function to close the RDT socket. However, before closing the RDT socket, the reliable layer needs to wait for TWAIT time units before closing the socket.

Simulate Packet Losses and Corruptions

Although UDP is an unreliable transport service, we hardly encounter datagram losses within local network. To test whether your rdt3.0 implementation is working correctly under erroneous situations, you have to randomly generate loss or error events when transmitting UDP datagrams. One simple method to simulate that is to use a wrapper function ***udt_send()***, which in turn, calls the *send()* function with some probability of packet loss or corruption. Below is the code fragment of this wrapper function for your reference. In your implementation of rdt-part2.h, you are required to use udt_send() for all message transmissions in the rdt_send(), rdt_recv(), and rdt_close() functions.

```
int udt_send(int fd, void * pkt, int pktLen, unsigned int flags) {
       double randomNum = 0.0;

       /* simulate packet loss */
       //randomly generate a number between 0 and 1
       randomNum = (double)rand() / RAND_MAX;
       if (randomNum < LOSS_RATE){
              //simulate packet loss of unreliable send
              printf("WARNING: udt_send: Packet lost in unreliable layer!!!!!!\n");
              return pktLen;
       }

       /* simulate packet corruption */
       //randomly generate a number between 0 and 1
       randomNum = (double)rand() / RAND_MAX;
       if (randomNum < ERR_RATE){
              //clone the packet
              u8b_t errmsg[pktLen];
              memcpy(errmsg, pkt, pktLen);
              //change a char of the packet
              int position = rand() % pktLen;
              if (errmsg[position] > 1) errmsg[position] -= 2;
              else errmsg[position] = 254;
              printf("WARNING:    udt_send:    Packet    corrupted    in    unreliable
layer!!!!!!\n");
              return send(fd, errmsg, pktLen, 0);
       } else       // transmit original packet
              return send(fd, pkt, pktLen, 0);
}
```

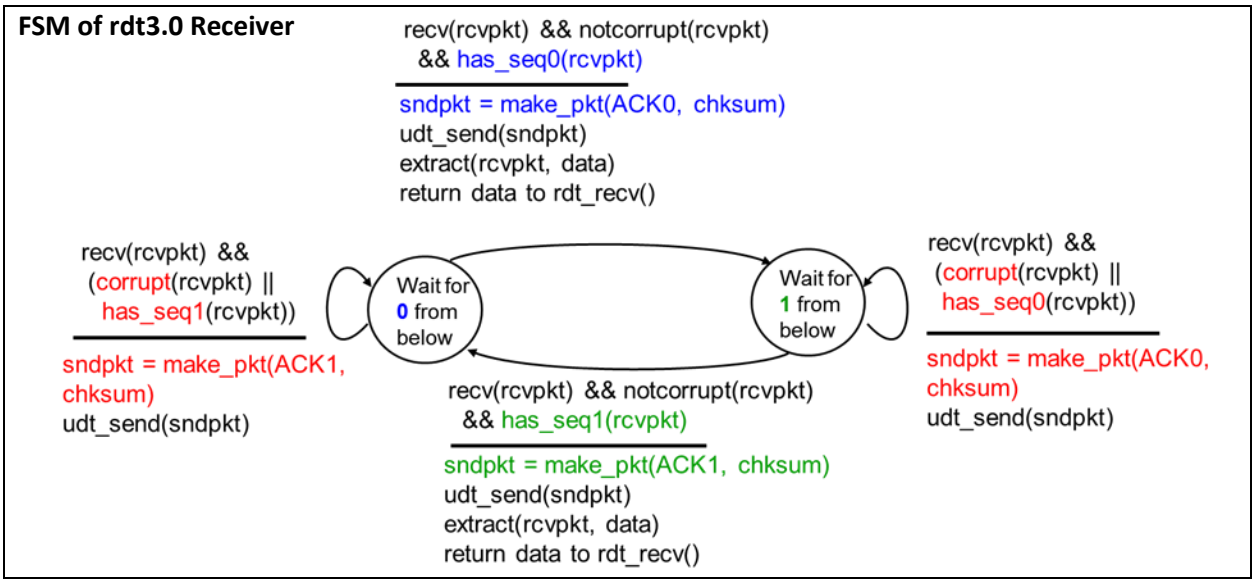
Checksum Calculation

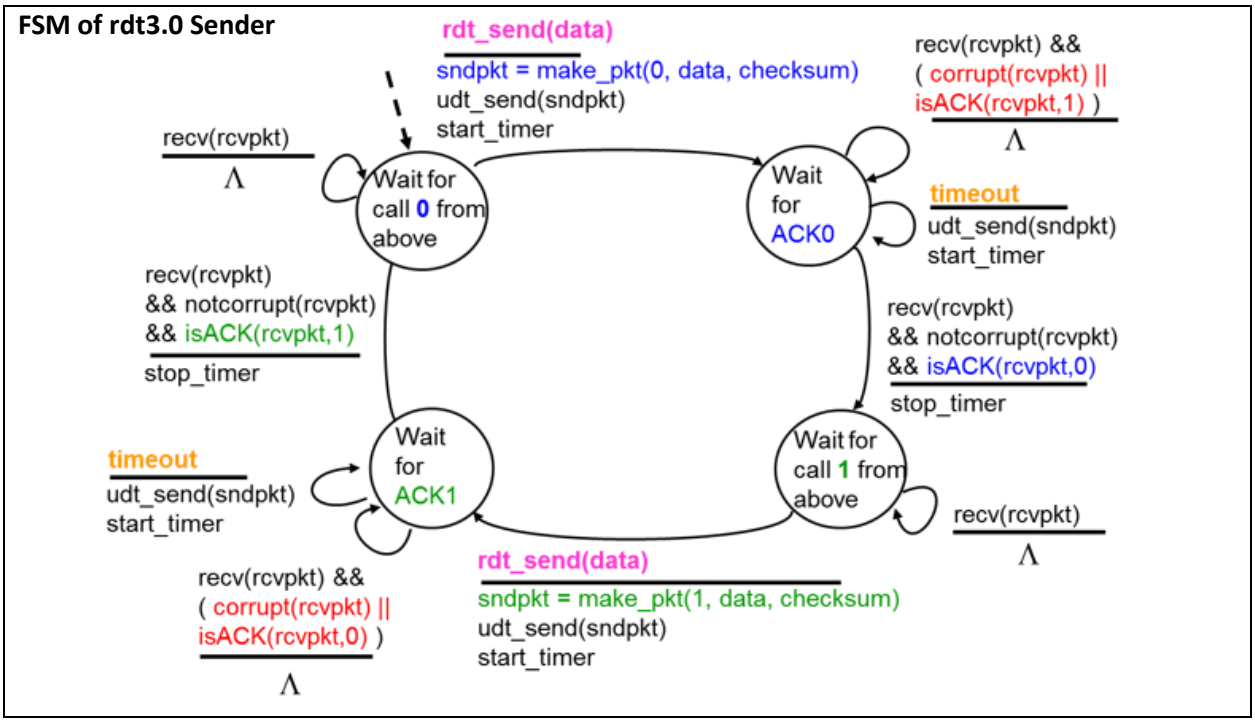
As transmissions may suffer with corruption, you need to use an error detection method to detect transmission errors. The Internet checksum is a good candidate for this exercise. Below is the code fragment of the checksum function for your reference. Before transmitting a packet, the system passes the packet to the checksum function to calculate a checksum value; then it stores the checksum value in the packet's header. After receiving the packet, the system passes the packet to the checksum function to determine whether it is corrupted or not. Please be remembered that the payload as well as the header of the packet can be corrupted.

```
//----- Type defines ----------------------------------------------------------
typedef unsigned char            u8b_t;       // a char
typedef unsigned short           u16b_t;      // 16-bit word
typedef unsigned int             u32b_t;      // 32-bit word

u16b_t checksum(u8b_t *msg, u16b_t bytecount)
{
       u32b_t sum = 0;
       u16b_t * addr = (u16b_t *)msg;
       u16b_t word = 0;

       // add 16-bit by 16-bit
       while(bytecount > 1)
       {
              sum += *addr++;
              bytecount -= 2;
       }

       // Add left-over byte, if any
       if (bytecount > 0) {
              *(u8b_t *)(&word) = *(u8b_t *)addr;
              sum += word;
       }

       // Fold 32-bit sum to 16 bits
       while (sum>>16)
              sum = (sum & 0xFFFF) + (sum >> 16);

       word = ~sum;

       return word;
}
```

Stop-and-Wait (rdt3.0) protocol

The rdt_send() and rdt_recv() are the two functions that support reliable data transfer between the two ends. Below diagrams show the finite state machines of the sender and receiver. In principle, you implement the sender logic in rdt_send() function and the receiver logic in rdt_recv() function. However, please be aware that the sender and receiver may encounter events that are not covered in its FSM; e.g., a sender, while waiting for the ACK, may receive a data packet from its peer, or vice versa.





In addition, instead of passing application data directly to UDP using udt_send() function, your protocol layer needs to encapsulate application data together with RDT header to form the PDU of your reliable protocol. Below diagram is a suggestion for your implementation. You can define two packet types, they are the data packets and ACK packets. Both packets have the same header fields such as sequence number and checksum value. Data packets have the payload field, while ACK packets do not. To differentiate between the two types, we add another field in the header.

Implementing rdt_close()

Our RDT layer is a connectionless reliable layer; thus, we don't need to implement any connection teardown procedure. However, a peer cannot close the RDT socket immediately after it finished all data transfer; in particular, this is crucial to the sender. This is because the last ACK sent by a receiver may be lost or corrupted; if a receiver closes its socket and leaves, nobody is going to handle the retransmitted packet from the sender and the sender is forced to hang there forever.

We suggest that before a peer closes it RDT socket, the peer stays in the TIME_WAIT state for TWAIT time units before closing the socket. In the TIME_WAIT state, the peer reacts to the retransmitted packet by returning an ACK packet and then re-enters the TIME_WAIT state again. Only until the peer does not receive any retransmissions within TWAIT time units, it quits the TIME_WAIT state and closes the socket.

Implementing Timeout

The sender needs to wait for acknowledgment after it sent out a packet. The sender invokes recv()/recvfrom() function to wait for the acknowledgment; however, these two functions are blocking function call, which blocks the calling process until a message arrived. If the sender is blocked, how can it react to the timeout event? One simple method is to use the ***select()*** function call. Although select() is

a blocking function, it can be set to block waiting for a fixed duration; thus, you can set this duration as the timeout duration. Please refer to Linux man page as well as to Lab 2 notes on how to use select() function.

Testing

Test your programs on the Ubuntu platform (14.04 or 12.04) with two different file sizes: (i) small file (around 30 KB), and (ii) large file (around 10 MB) with different combinations of PACKET_LOSS_RATE and PACKET_ERR_RATE:

PACKET_LOSS_RATE	PACKET_ERR_RATE
0.0	0.0
0.2	0.0
0.0	0.2
0.2	0.2
0.3	0.3

Sample Output

We suggest you generating an output statement whenever the RDT layer sends out or receives a packet (include the packet type and some control information in the output). In addition, you can generate an output statement whenever the RDT layer detects or experiences an expected event or unexpected event or error situation. Please refer to Figures 2 & 3 of the document "Project-sample-output.pdf" for the sample output.

Submission

You only need to submit the rdt-part2.h header file. Submit the file to the Course's web site at moodle.hku.hk.

PART3 (6 POINTS)

Download the compressed file – Part3.zip from Course web site at moodle.hku.hk. Same as Part2.zip, it contains six files: Makefile, run-client.sh, run-server.sh, test-client.c, test-server.c, and rdt-part3.h. Most of the files (except rdt-part3.h) are the same except changes are made to include rdt-part3.h for compilation.

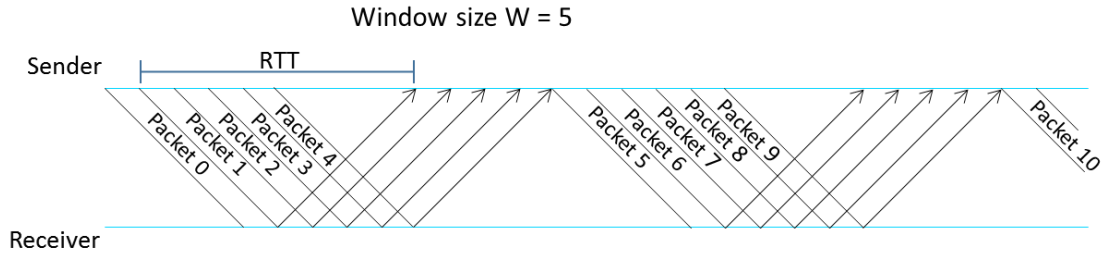
1. The header file – rdt-part3.h, which defines all functions provided by the RDT layer to the application layer as well as two utility functions – *udt_send()* and *checksum()*. However, rdt-part3.h file is not completed. ***Your task in this part is to complete rdt-part3.h file.***

   Same as rdt-part2.h, the service interface consists of six functions: rdt_socket(), rdt_bind(), rdt_target(), rdt_send(), rdt_recv() and rdt_close(). You can reuse your implementation of rdt_socket(), rdt_bind() and rdt_target() in part1. For rdt_send(), rdt_recv and rdt_close(), you need to modify those functions in part2 to include the Extended Stop-and-Wait reliable logic.

   - rdt_send() — Application process calls this function to transmit a message (up to a limit of $\mathbf{1000} \times \boldsymbol{W}$ bytes) to targeted remote process through the RDT socket. This function will return only when it knows that the **whole** message has been successfully delivered to remote process.
   - rdt_recv() — Application process calls this function to wait for a message from targeted remote process; the caller will be blocked waiting for the arrival of message. Upon arrival of a message (**which is carried in one packet**), the RDT layer will immediately return the message to the caller.
   - rdt_close() — Application process calls this function to close the RDT socket. The behavior of this function is the same as in part 2 except that a peer may receive retransmitted packets with different sequence numbers.

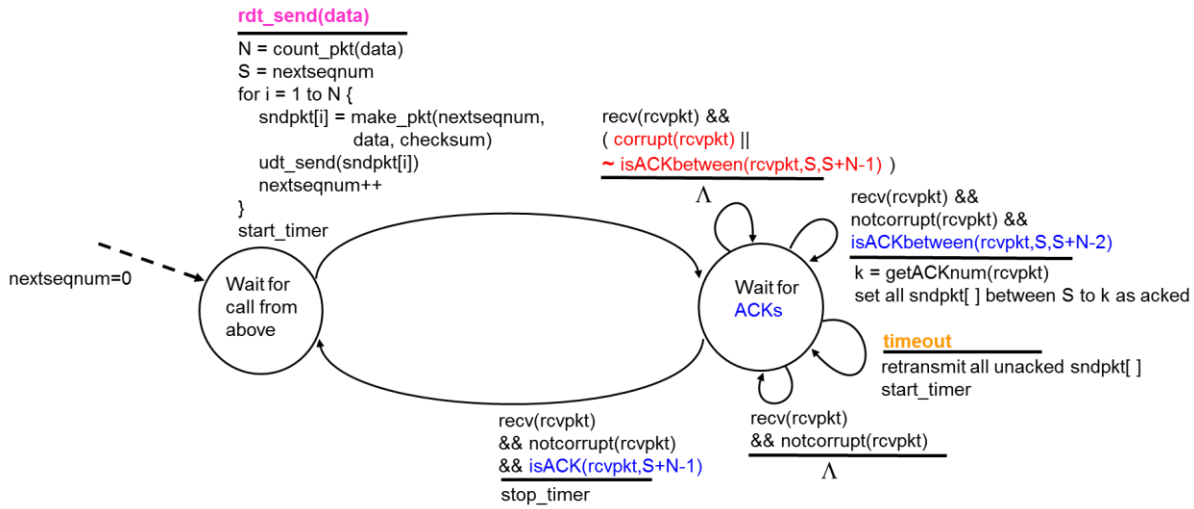
Extended Stop-and-Wait protocol

The rdt3.0 Stop-and-Wait protocol allows only one segment in transit from the sender to the receiver, i.e., the sender waits for the acknowledgment before sending the next segment. The extended version of Stop-and-Wait protocol allows a full window of segments to be in-transited before stop-and-wait for the acknowledgments. The window size W is a constant defined in rdt-part3.h; we can adjust it during compilation time. Strictly speaking, the Extended Stop-and-Wait protocol is a pipelined protocol, but it is not a sliding window scheme since its window does not slide forward until received all acknowledgments.



Below diagrams show the finite state machines of the sender and receiver. In principle, you implement the sender logic in rdt_send() function and the receiver logic in rdt_recv() function. However, please be aware that the sender and receiver may encounter events that are not covered in its FSM; e.g., a sender, while waiting for the ACK, may receive a data packet from its peer, or vice versa.

FSM of E-S&W sender

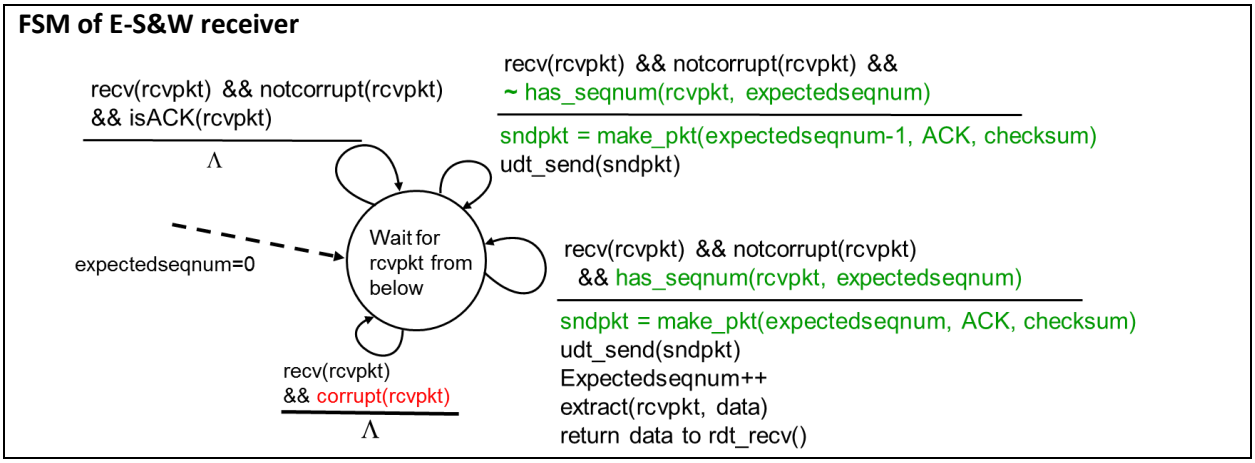


Basically, the E-S&W sender responds to three types of events:

1. Call by application - invocation of rdt_send(). Based on the length of the message, the sender determines how many packets (***N***) it is going to be transmitted in this round, and it is always true that $N \leq W$. Each packet consumes a sequence number so as to uniquely identify a packet. After sent out all ***N*** packets, the sender starts the Timeout timer and waits for all acknowledgments to come back.
2. Receipt of an ACK. In the E-S&W protocol, we adopt the **cumulative acknowledgment** approach. Each ACK packet carried a sequence number ***k***, which indicates that all packets with a sequence number up to and including ***k*** have been correctly received at the receiver. Once the sender knows that all ***N*** packets have been correctly received, it clears the Timeout timer and waits for call from above.
3. A timeout event. If a timeout occurs, the sender resends all packets that have been previously sent in this round but have not yet been acknowledged. One single Timeout timer is used for keeping track

of all ***N*** packets. If an ACK is received but there still have pending to be acknowledged packets, the timer is restarted.

FSM of E-S&W receiver



The behavior of E-S&W receiver closely resemble the behavior of GBN receiver. If a data packet with sequence number ***j*** is received correctly and ***j*** is the current expected in-order sequence number, the receiver sends an ACK packet with sequence number ***j*** and delivers the data to the application layer. Otherwise, the receiver discards that data packet and resends an ACK for the most recently received in-order packet.

Testing

Test your programs on the Ubuntu platform (14.04 or 12.04) with two different file sizes: (i) small file (around 30 KB), and (ii) large file (around 10 MB) with different combinations of W, PACKET_LOSS_RATE and PACKET_ERR_RATE:

W	PACKET_LOSS_RATE	PACKET_ERR_RATE
1	0.0	0.0
5	0.0	0.0
9	0.0	0.0
1	0.1	0.1
5	0.1	0.1
9	0.1	0.1
1	0.3	0.3
5	0.3	0.3
9	0.3	0.3

Sample Output

We suggest you generating an output statement whenever the RDT layer sends out or receives a packet (include the packet type and some control information in the output). In addition, you can generate an output statement whenever the RDT layer detects or experiences an expected event or unexpected event or error situation. Please refer to Figures 4 to 6 of the document “Project-sample-output.pdf” for the sample output.

Submission

You only need to submit the rdt-part3.h header file. Submit the file to the Course's web site at moodle.hku.hk.

Implementation requirements

DO NOT implement the reliable layer on top of other protocols (e.g. TCP). You must implement the reliable layer on top of UDP and use the udt_send() function that we have provided to you to simulate transmission errors. We will definitely check on this issue, and we will consider implementations that using other means as a fail case and will receive ZERO mark.

Computer Platform to Use

For this project assignment, you are expected to develop and test your program under Ubuntu 14.04 or 12.04. You can use C/C++ to implement the program, and it should be successfully compiled with g++ (or gcc).

Format for the documentation

1. At the head of the submitted header files, state clearly the
   - Student name:
   - Student No. :
   - Date and version:
   - Development platform:
   - Development language:
   - Compilation:
2. Inline comments (try to be detailed so that your code could be understood by others easily)

Grading Policy

1. The tutor will first test your final submission after the project deadline. If your program fully complies with the project specification, you'll get all marks for all three parts automatically. Otherwise, tutor will test your Part 2 submission (if any). If it passes all test cases, you'll get all marks for Parts 1 and 2. Otherwise, tutor will test your Part 1 submission (if any) to give marks.
2. After submission of your program(s) at Parts 1 or 2, if you want to know whether your program works correctly, you are welcome to make an appointment with the tutor and demonstrate your program to the tutor. Therefore, you could receive feedback to improve your program for next phase.
3. As the tutor will check your source code, please write your program with good readability (i.e., with good code convention and sufficient comments) so that you will not lose marks due to possible confusions.

Documentation (1 point)	High Quality [0.5/1] - only apply to Parts 2 and 3 • Include necessary documentation to clearly indicate the logic of the program
	Standard Quality [0.5/1] • Include required program and student's info at the beginning of the program • Include minimal inline comments
Part 1 (2 points)	• The program should be compiled and executed successfully. • The program can transfer data correctly in the local area network (assume no packet loss). If your implementation cannot transfer the file correctly for some test cases, you can only get at most 1 point in this part.

Part 2 (7 points)	• The program should be compiled and executed successfully. • The program can transfer data and terminate correctly in an environment without packet loss and corruption. [2/7] • The program can transfer data and terminate correctly in an environment with packet loss but no corruption ($0.0<LOSS\leq0.3$, $ERR=0.0$). [1.5/7] • The program can transfer data and terminate correctly in an environment with packet corruption but no loss ($LOSS=0.0$, $0.0<ERR\leq0.3$). [1.5/7] • The program can transfer data and terminate correctly in an environment with packet loss and corruption ($0.0<LOSS\leq0.3$, $0.0<ERR\leq0.3$). [2/7]
Part 3 (6 points)	• The program should be compiled and executed successfully. • The program can transfer data and terminate correctly with $W=1$ in an environment without packet loss and corruption [1/6] and in an environment with loss and corruption [1/6]. • The program can transfer data and terminate correctly with $1<W\leq9$ in an environment without packet loss and corruption [1.5/6] and in an environment with loss and corruption ($0.0<LOSS\leq0.3$, $0.0<ERR\leq0.3$) [2.5/6].

Background Readings and Preparation

- Lectures - Transport Layer: Parts 2 and 3
- Section 3.4 of textbook – Computer Networking: A Top-Down Approach (6th ed.) by J.F. Kurose et. al
- Lecture - Socket Programming in C and the Labs
- Project briefing notes

Plagiarism

Plagiarism is a very serious academic offence. Students should understand what constitutes plagiarism, the consequences of committing an offence of plagiarism, and how to avoid it. **Please note that we may request you to explain to us how your program is functioning as well as we may also make use software tools to detect software plagiarism.**