COMP3234 Computer and Communication Networks

Programming Project

Total 16 points

Version 1.0

Part 2 Due Date: 5pm March 14, 2018

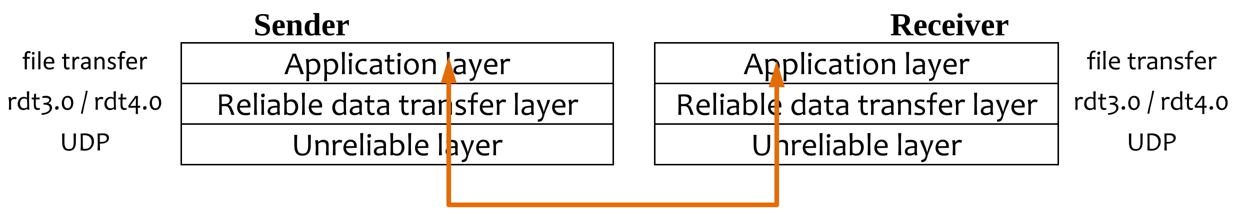
Part 3 Due Date: 5pm April 11, 2018

Hand-in the assignment via the Moodle System.

Stop-and-Wait (rdt3.0) ARQ
and
Extended Stop-and-Wait (rdt4.0) ARQ

Overview

In this project, you are going to design and implement two reliable data transfer layers, one is based on the Stop-and-Wait (rdt3.0) protocol and the other is based on the extended **Stop-and-Wait** (rdt 4.0) protocol. Both RDT protocols support **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 the RDT layer in order to send/receive messages to/from remote application processes. Upon receiving a message from the Application layer, the RDT layer encapsulates the application message with control header to form a 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 packet. To simulate transmission errors, the Unreliable layer may randomly discard or corrupt packets.

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 socket functions to implement a real-life protocol.

Specification

This programming project consists of three parts:

- Part 1 [Warm up] – **Assume** UDP is reliable (i.e., the Unreliable layer is indeed "reliable"), you are going to implement the RDT layer (rdt1.0) directly on top of UDP with the wrapper functions ***__udt_send()*** and ***__udt_recv()***. This is true when running the application within the virtual machine or in local area network .
- Part 2 [Submission] – Assume UDP is unreliable, which suffers with packet losses and transmission errors. You are going to implement the RDT layer using Stop-and-Wait protocol (rdt3.0) directly on top of UDP but with the wrapper function ***__udt_send()*** that simulates the loss and corruption.
- Part 3 [Submission] – To improve the performance, you are going to implement a pipelined protocol - ***Extended Stop-and-Wait*** (rdt4.0) in the RDT layer, which works on top of the unreliable layer.

Part 1 – No Submission is Required

The objective of Part 1 is to help you get familiar with the RDT API. Please download the compressed file – Part1.zip from the course web site at moodle.hku.hk. This zip file contains seven files:

1. A python module file – rdt1.py, which defines all functions provided by the RDT layer to the application layer. However, rdt1.py file is not completed. ***Your task in this part is to complete rdt1.py file.***

   You have to complete 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_peer()	Application process calls this function to specify the IP address and port number used by the remote peer.
rdt_send()	Application process calls this function to transmit a message (up to a limit of PAYLOAD bytes) to the targeted remote peer 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 the targeted remote peer; the caller will be blocked waiting for the arrival of the message. Upon arrival of a message (of maximum size PAYLOAD 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 most of the functions by using standard socket functions except rdt_send() and rdt_recv(). To simulate the underlying unreliable layer, two internal functions are provided: *__udt_send()* and *__udt_recv()*. They will be used by rdt_send() and rdt_recv() for sending and receiving messages.
2. A pair of test files – test-client1.py and test-server1.py. These two programs require the rdt1.py module 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:

```
python3 test-client1.py  'server hostname'  'filename'
```

The server program accepts one argument

```
python3 test-server1.py  '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.

3. Three script files – (1) run-simulation1.bat for Windows, (2) run-simulation1-OSX.sh for Mac OSX, and (3) run-simulation1-Ubuntu.sh for Ubuntu. By running the script file on corresponding platform, the system creates two terminals for you and starts running the server and client programs in each terminal for you. For example, to run the simulation on Windows, on a terminal, type: run-simulation1.bat 'filename'
4. Part1-sample-output.pdf - 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 this document for the sample output.

Testing

Test your programs with three different file sizes: (i) small file (around 30 KB), (ii) moderate size (around 500 KB), and (iii) large file (around 10 MB).

Submission is not required

You are not required to submit Part1 program. This is just a warm-up exercise. The sample rdt1.py program will be released to the class after Chinese New Year Break.

PART 2 (9 POINTS) – SUBMISSION DEADLINE: MARCH 14 17:00

You are going to implement the RDT3.0 protocol in this part. Download the compressed file – Part2.zip from the course web site at moodle.hku.hk. This zip file contains seven files:

1. Two test program files for part 2 – test-client2.py and test-server2.py. They are similar to the part 1 testing set except a few minor differences. They require rdt3.py module file to work correctly.

   The client program accepts three arguments:

   ```
   python3 test-client2.py  'server hostname'  'filename'  'packet loss rate'  'packet error rate'
   ```

   The server program accepts two arguments:

   ```
   python3 test-server2.py  'client hostname'  'packet loss rate'  'packet error rate'
   ```

2. Three script files – one for Windows, one for Mac OSX, and one for Ubuntu.

   To start the simulation, you have to run the script with 3 arguments. For example, on the Windows terminal, type: run-simulation2.bat 'filename' 'packet loss rate' 'packet error rate'.

   The packet loss rate and packet error rate are used by the __udt_send() function to control the degree of packet loss and corruption respectively. If these two variables are set to 0.0; thus, the system does not experience any loss or corruption. If their values are set to be greater than 0, this means we want to simulate the loss and corruption. The higher the number, the higher the chance of the errors / losses.

3. The module file – rdt3.py, which defines all functions provided by the RDT3.0 layer to the application layer as well as three internal functions – *__udt_send()*, *__udt_recv()* and *__IntChksum()*,

and a few global variables. However, rdt3.py file is not completed. ***Your task in this part is to complete rdt3.py file.***

Same as rdt1.py, the service interface consists of seven functions: rdt_network_init(), rdt_socket(), rdt_bind(), rdt_peer(), rdt_send(), rdt_recv() and rdt_close(). However, we have already completed the rdt_network_init() function. You are going to implement the Stop-and-Wait (rdt3.0) reliable logic in these functions.

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_peer()	Application process calls this function to specify the IP address and port number used by the remote peer.
rdt_send()	Application process calls this function to transmit a message (up to a limit of PAYLOAD bytes) to the targeted remote peer through the RDT socket. This function will return only when it knows that the application message has been successfully delivered to the remote peer.
rdt_recv()	Application process calls this function to wait for a message from the targeted remote peer; the caller will be blocked waiting for the arrival of the message. Upon arrival of a message (of maximum size PAYLOAD bytes), the RDT layer will immediately return the message to the caller.
rdt_close()	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.

4. Part2-sample-output.pdf – 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 this document for the sample output.

Simulate Packet Losses and Corruptions

Although UDP is an unreliable transport service, we hardly encounter packet losses within a local network. To test whether your rdt3.0 implementation is working correctly under erroneous situations, we have to randomly generate loss or error events when transmitting UDP packets. One simple method to simulate that is to use a wrapper function ***__udt_send()***, which in turn, calls the *sendto()* function with some probability of packet loss or corruption. In your implementation of rdt3.py, you are required to use __udt_send() and __udt_recv() for sending and receiving messages in the rdt_send(), rdt_recv(), and rdt_close() functions.

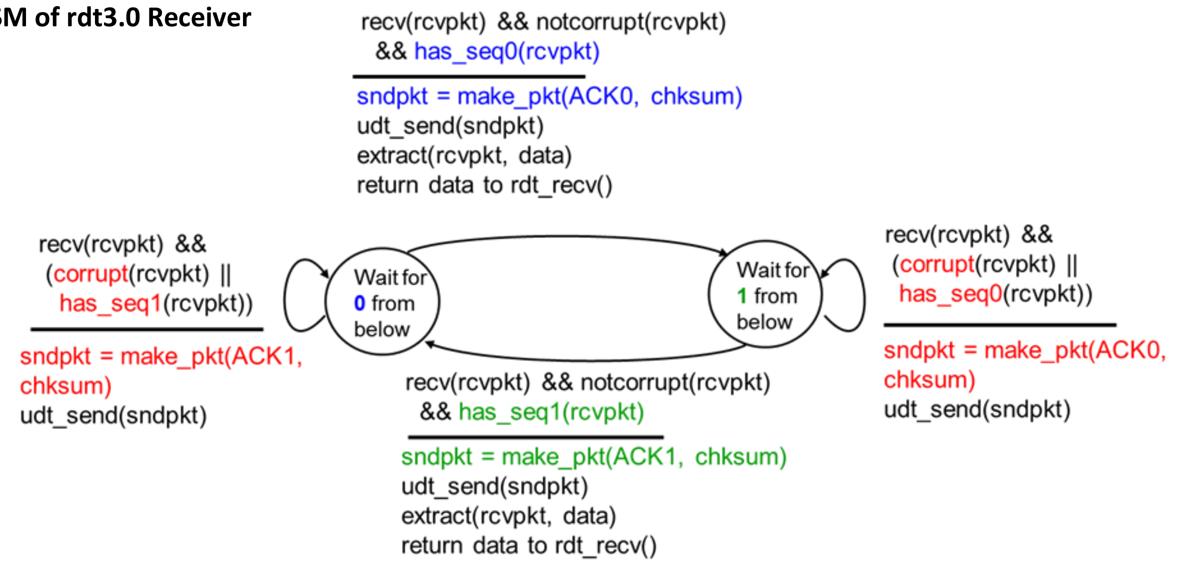
Checksum Calculation

As transmissions may suffer from corruption, you need to use an error detection method to detect transmission errors. The Internet checksum algorithm is a good candidate for this exercise. We have implemented the algorithm as an internal function – *__IntChksum()* for the project. 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 and the header of the packet can be corrupted.

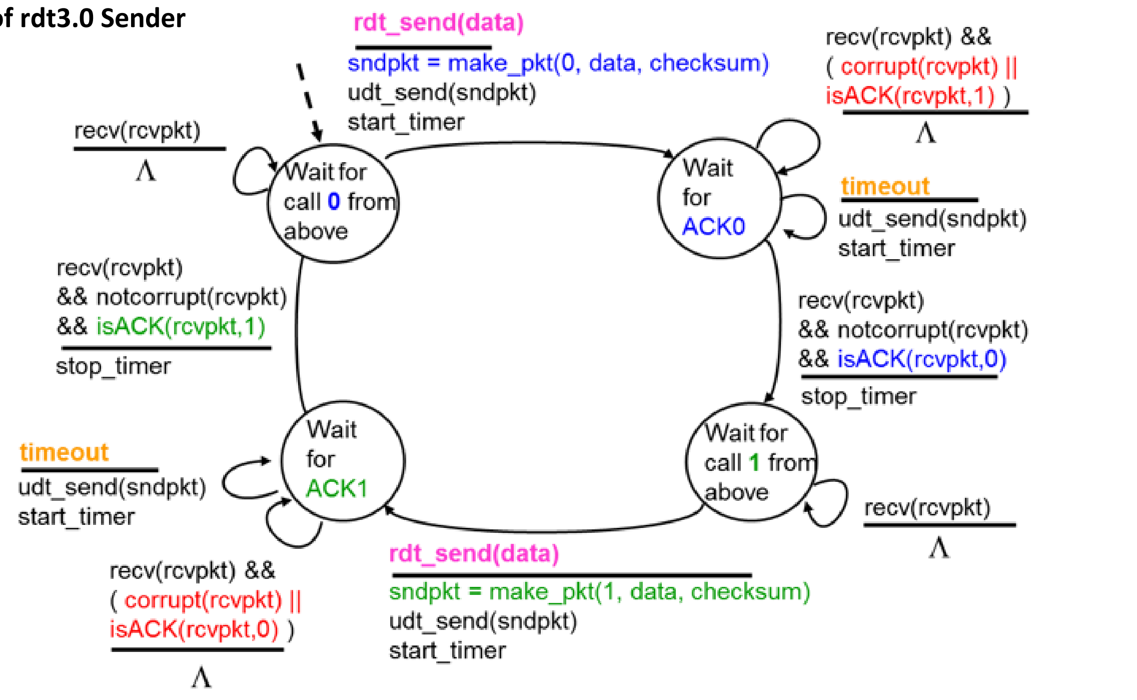
Stop-and-Wait (rdt3.0) protocol

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.

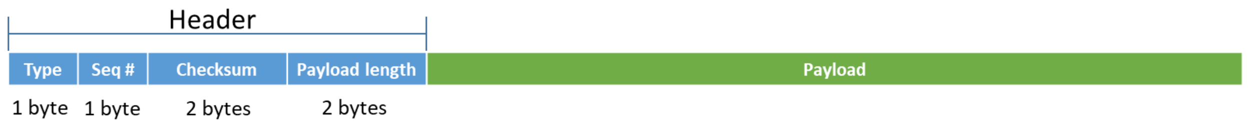
FSM of rdt3.0 Receiver


recv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
sndpkt = make_pkt(ACK0, chksum)
udt_send(sndpkt)
extract(rcvpkt, data)
return data to rdt_recv()
recv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
recv(rcvpkt) && (corrupt(rcvpkt) || has_seq0(rcvpkt))
sndpkt = make_pkt(ACK0, chksum)
udt_send(sndpkt)
recv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)
extract(rcvpkt, data)
return data to rdt_recv()

FSM of rdt3.0 Sender


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
recv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
Λ
recv(rcvpkt)
Λ
Wait for call 0 from above
Wait for ACK0
timeout
udt_send(sndpkt)
start_timer
recv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
recv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
timeout
udt_send(sndpkt)
start_timer
Wait for ACK1
Wait for call 1 from above
recv(rcvpkt)
Λ
recv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
Λ
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer

Instead of passing application data directly to UDP using __udt_send() function, your protocol layer needs to encapsulate application data by the RDT header to form the PDU of the reliable protocol. Below diagram shows the message format of the reliable protocol.



The protocol header consists of 4 fields: Type (size: 1 byte), Sequence no. (size: 1 byte), Checksum (size: 2 bytes), and Payload length (size: 2 bytes). There are two types of packets – data packets and ACK packets. Data packets carry application data in the Payload field, while ACK packets do not have any payload. To differentiate between the two types, we use the Type field in the header. This is the suggested setting of the Header fields:

Type	ACK = 11, Data = 12
Sequence no.	0 or 1
Checksum	Use __IntChksum() to calculate the checksum for the whole packet (Header + Payload)
Payload length	0 to 1000

Implementing rdt_close()

Our RDT layer is a connectionless reliable layer; thus, we don't need to implement any connection set-up procedure. However, a peer cannot close the RDT socket immediately after it finished all data transfer; 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 packets within TWAIT time units, it quits the TIME_WAIT state and closes the socket.

Implementing Timeout

The sender needs to wait for the acknowledgment after it sent out a packet. The sender invokes __udt_recv() function to wait for the acknowledgment; however, this function invokes the blocking recvfrom() function call, which blocks the calling process until a message arrived. If the sender is blocked, how can it react to the timeout event? A simple method is to use the socket timeout mode or select.select() function. Although recvfrom() is a blocking function, by setting the socket with the timeout mode, the function is set to block waiting for a fixed duration; thus, you can set this duration as the timeout duration. Please refer to Lab 2 worksheet on how to use the socket timeout mode or select.select() function.

Testing

Test your programs with two different file sizes on any platform installed with Python 3: (i) small file (around 30 KB), and (ii) large file (around 10 MB) with different combinations of packet loss rate and packet error rate:

PACKET LOSS RATE	PACKET ERROR RATE
0.0	0.0
0.2	0.0
0.0	0.2
0.2	0.2
0.3	0.3

Submission

You only need to submit the rdt3.py program. Submit the file to the Course's web site at moodle.hku.hk.

PART 3 (7 POINTS) – SUBMISSION DEADLINE: APRIL 11 17:00

You are going to implement the Extended Stop-and-Wait (rdt4.0) protocol in this part. Download the compressed file – Part3.zip from course web site at moodle.hku.hk. Same as Part2.zip, it contains seven files:

1. Two test program files for part 3 – test-client3.py and test-server3.py. They are similar to the parts 1 and 2 testing sets except for a few minor differences. They require rdt4.py module file to work correctly.

   The client program accepts four arguments:

   python3 test-client3.py 'server hostname' 'filename' 'packet loss rate' 'packet error rate' 'window size'

   The server program accepts three arguments:

   python3 test-server3.py 'client hostname' 'packet loss rate' 'packet error rate' 'window size'
2. Three script files – one for Windows, one for Mac OSX, and one for Ubuntu.

   To start the simulation, you have to run the script with 4 arguments. For example, on the Windows terminal, type: run-simulation3.bat 'filename' 'packet loss rate' 'packet error rate' 'window size'. The packet loss rate and packet error rate have the same meaning/usage as in Part 2. The window size, which is an integer, defines the number of packets a sender can send before stop-and-wait for the acknowledgments to come back.
3. The module file – rdt4.py, which defines all functions provided by the RDT4.0 layer to the application layer as well as three internal functions – *__udt_send()*, *__udt_recv()* and *__IntChksum()*, and a few global variables. However, rdt4.py file is not completed. ***Your task in this part is to complete rdt4.py file.***

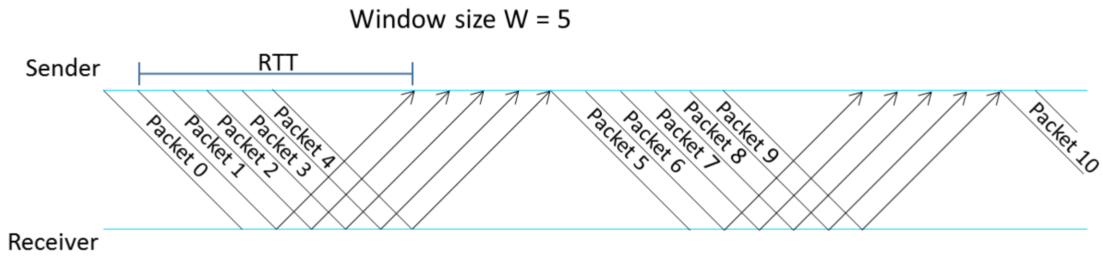
   Same as rdt3.py, the service interface consists of seven functions: rdt_network_init(), rdt_socket(), rdt_bind(), rdt_peer(), rdt_send(), rdt_recv() and rdt_close(). We have completed the rdt_network_init() function. You can reuse your implementation in Part 2 and just modify those functions to include the Extended Stop-and-Wait (rdt4.0) reliable logic; in particular, rdt_send(), rdt_recv(), and rdt_close().

rdt_send()	Application process calls this function to transmit a message (up to a limit of **W × PAYLOAD** bytes) to the targeted remote peer through the RDT socket. This function will return only when it knows that the **whole** message has been successfully delivered to the remote peer.
rdt_recv()	Application process calls this function to wait for a message from the targeted remote peer; the caller will be blocked waiting for the arrival of the 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.

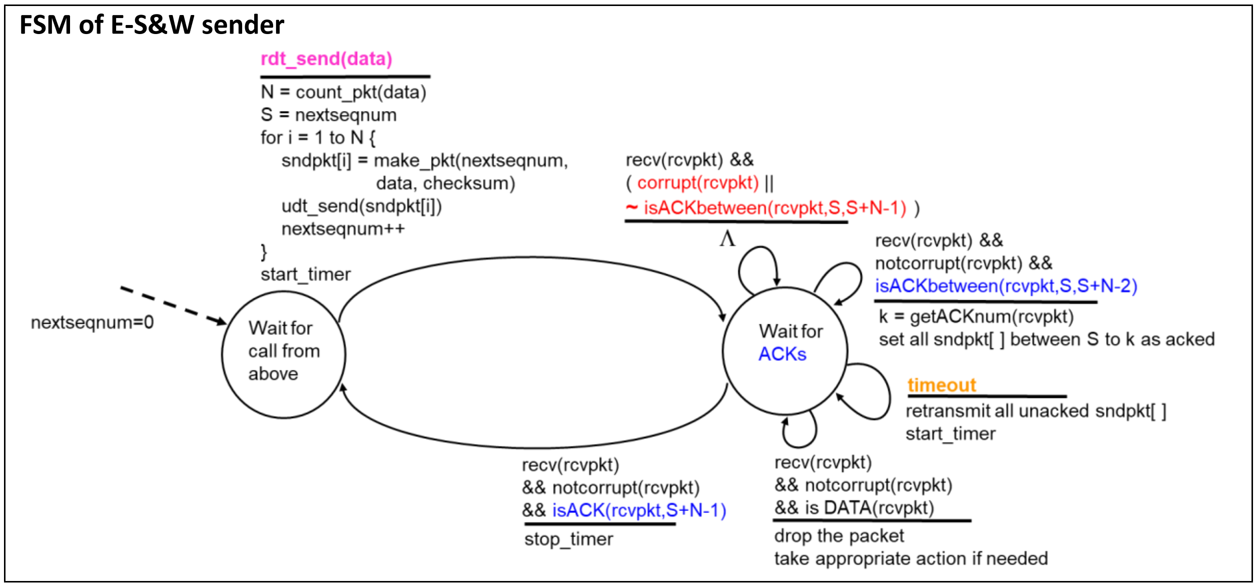
4. Part3-sample-output.pdf - We suggest you generating an output statement whenever the RDT4.0 layer sends out or receives a packet (include the packet type and some control information in the output). In addition, you can generate an output statement whenever the RDT layer detects or experiences an expected event or unexpected event or error situation. Please refer to this document for the sample output.

Extended Stop-and-Wait protocol

The rdt3.0 Stop-and-Wait protocol allows only one packet in transit from the sender to the receiver, i.e., the sender waits for the acknowledgment before sending the next packet. The extended version of Stop-and-Wait protocol allows a full window of packets to be in-transited before stop-and-wait for the acknowledgments. The window size W is defined by the user; we use the rdt_network_init() function to pass it to the RDT4.0 layer. Strictly speaking, the Extended Stop-and-Wait protocol is a pipelined protocol, but it is not the traditional sliding window schemes (e.g., Go-Back-N and Selective Repeat) since its window only slides forward after received all acknowledgments in the current window.



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
rdt_send(data)
N = count_pkt(data)
S = nextseqnum
for i = 1 to N {
sndpkt[i] = make_pkt(nextseqnum, data, checksum)
udt_send(sndpkt[i])
nextseqnum++
}
start_timer
recv(rcvpkt) && (corrupt(rcvpkt) || ~ isACKbetween(rcvpkt,S,S+N-1))
Λ
recv(rcvpkt) && notcorrupt(rcvpkt) && isACKbetween(rcvpkt,S,S+N-2)
k = getACKnum(rcvpkt)
set all sndpkt[ ] between S to k as acked
nextseqnum=0
Wait for call from above
Wait for ACKs
timeout
retransmit all unacked sndpkt[ ]
start_timer
recv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,S+N-1)
stop_timer
recv(rcvpkt) && notcorrupt(rcvpkt) && is DATA(rcvpkt)
drop the packet
take appropriate action if needed

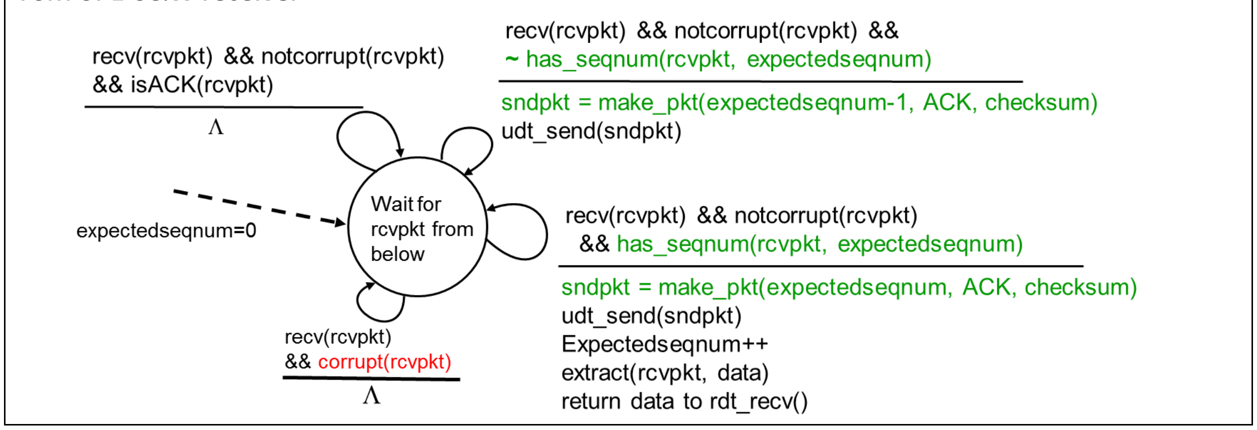
Basically, the E-S&W sender responds to three types of events:

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 by 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



recv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt)
Λ
recv(rcvpkt) && notcorrupt(rcvpkt) && ~ has_seqnum(rcvpkt, expectedseqnum)
sndpkt = make_pkt(expectedseqnum-1, ACK, checksum)
udt_send(sndpkt)
expectedseqnum=0
Wait for rcvpkt from below
recv(rcvpkt) && notcorrupt(rcvpkt) && has_seqnum(rcvpkt, expectedseqnum)
sndpkt = make_pkt(expectedseqnum, ACK, checksum)
udt_send(sndpkt)
Expectedseqnum++
extract(rcvpkt, data)
return data to rdt_recv()
recv(rcvpkt) && corrupt(rcvpkt)
Λ

The behavior of E-S&W receiver closely resembles 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 with two different file sizes on any platform installed with Python 3: (i) small file (around 30 KB), and (ii) large file (around 10 MB) with different combinations of packet loss rate and packet error rate:

W	PACKET LOSS RATE	PACKET ERROR 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

Submission

You only need to submit the rdt4.py program. 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 on any platform installed with Python 3 or above.

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:
   - Python version:
2. Inline comments (try to be detailed so that your code could be understood by others easily)

Grading Policy

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.

Part 2 (9 points)	• The program can transfer data and terminate correctly in an environment without packet loss and corruption. [2.5/9] • The program can transfer data and terminate correctly in an environment with packet loss but no corruption ($0.0<LOSS\leq 0.3$, ERROR=0.0). [2/9] • The program can transfer data and terminate correctly in an environment with packet corruption but no loss (LOSS=0.0, $0.0<ERROR\leq 0.3$). [2/9] • The program can transfer data and terminate correctly in an environment with packet loss and corruption ($0.0<LOSS\leq 0.3$, $0.0<ERROR\leq 0.3$). [2/9] • Documentation [0.5/9] Include necessary documentation to clearly indicate the logic of the program; include required student's info at the beginning of the program

Part 3 (7 points)	• The program can transfer data and terminate correctly with W=1 in an environment without packet loss and corruption [1/7] and in an environment with loss and corruption [1/7]. • The program can transfer data and terminate correctly with 1<W≤10 in an environment without packet loss and corruption [2/7] and in an environment with loss and corruption (0.0<LOSS≤0.3, 0.0<ERR≤0.3) [2.5/7]. • Documentation [0.5/7] Include necessary documentation to clearly indicate the logic of the program; include required student's info at the beginning of the program

Background Readings and Preparation

- Lectures – 03b-Transport-S&W.pdf
- Section 3.4 of textbook – Computer Networking: A Top-Down Approach (6th & 7th ed.) by J.F. Kurose et. al
- Lecture - Socket Programming in Python 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.**