2020-21 COMP3234A

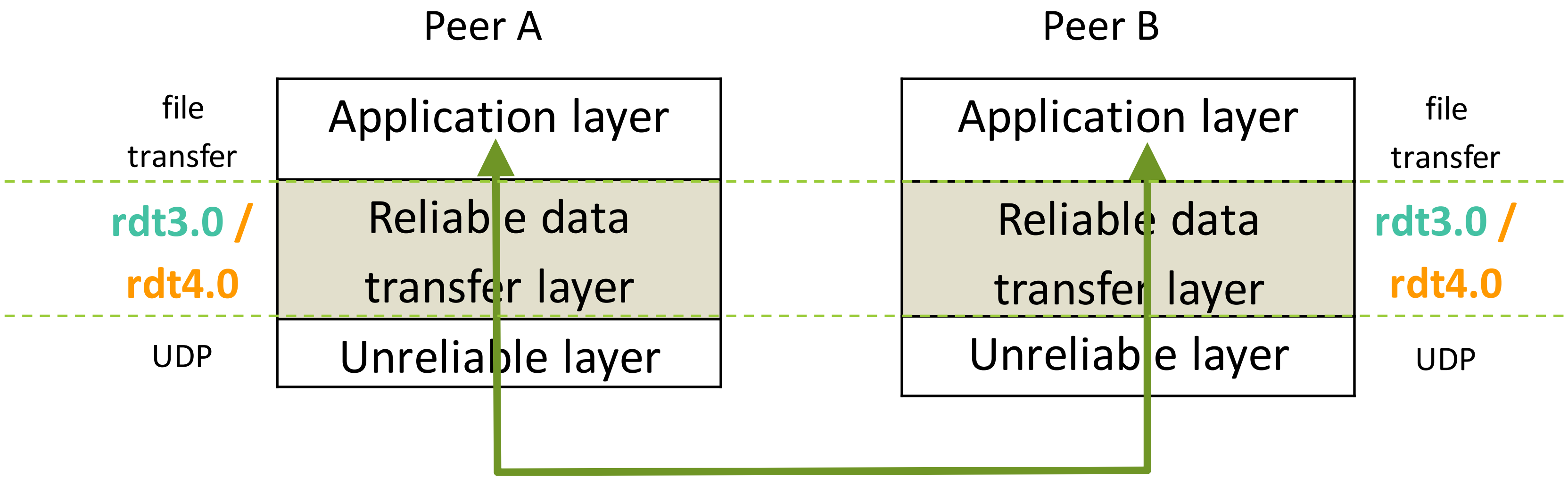
Programming Project

STOP-AND-WAIT (RDT3.0) ARQ AND

EXTENDED-STOP-AND-WAIT (RDT4.0) ARQ

Overview

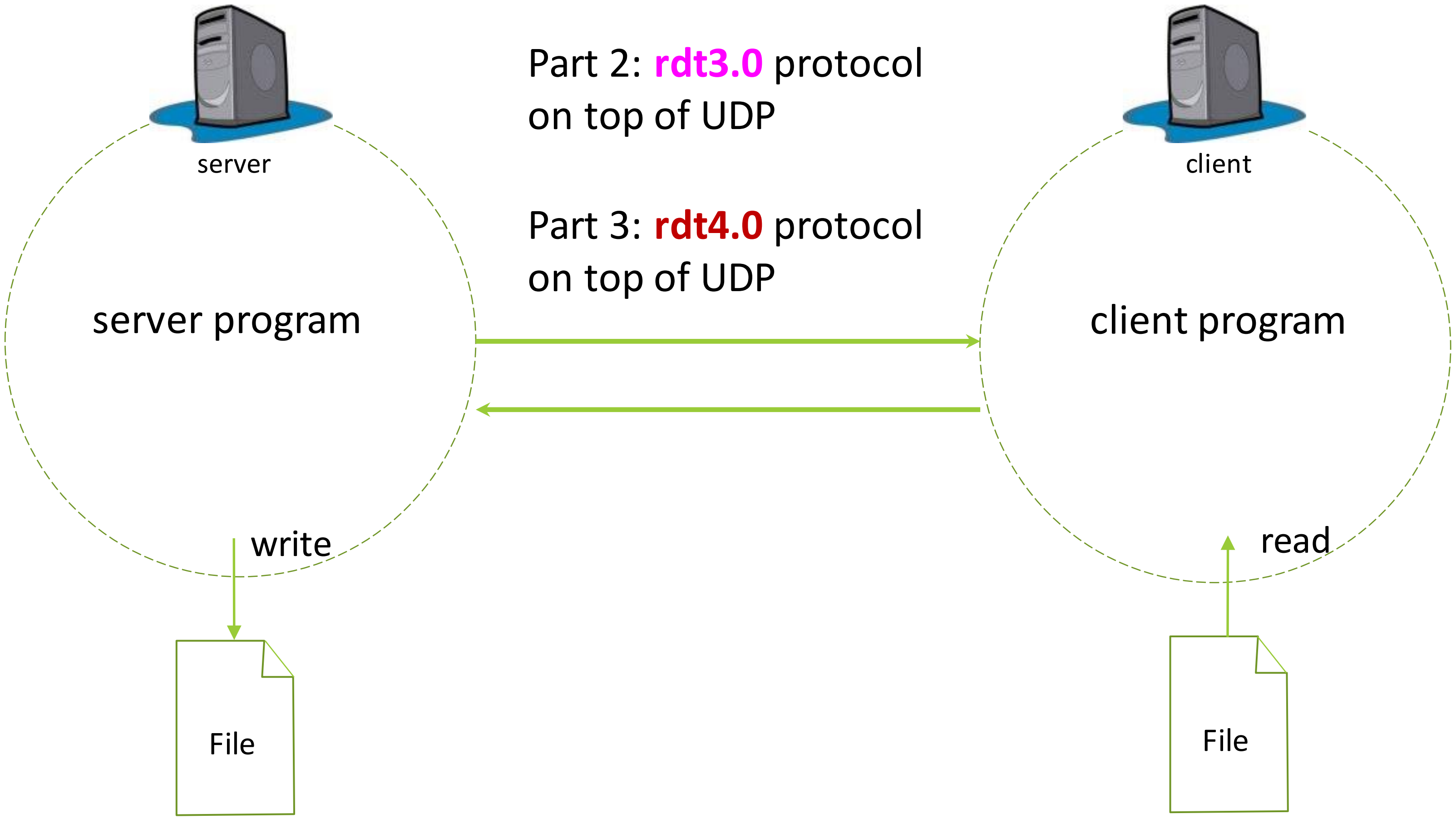
Our RDT protocols support **connectionless** reliable **duplex** data transfer on top of unreliable UDP



Objectives

- To get better understanding of the **principles** behind Stop-and-Wait 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
- The project is divided into **three parts** to make it more attainable
- An assessment task related to ILO4 [Implementation]

Application



Service Interface of the RDT layer

rdt_network_init()	To set up the simulation environment.
rdt_socket()	To create a RDT socket.
rdt_bind()	To assign address info used by this RDT socket.
rdt_peer()	To inform the system the address info of a remote peer.
rdt_send()	To reliably transmit an application message to the targeted remote peer through this RDT socket.
rdt_recv()	To block and wait for a message from the targeted remote peer.
rdt_close()	To close this RDT socket.

Note: Our RDT layer offers a slightly different Service Interface as compared to TCP and UDP; in particular, we have the functions rdt_network_init() and rdt_peer() that do not exist in standard socket interface.

Structure of the Project

Part 1 Warm up

Examine rdt1.py

- **Assume UDP is reliable**
- Implement the reliable layer directly on top of UDP without adding extra functionality to UDP

rdt_send(),
rdt_recv(),
rdt_close()

Part 2 [9 points]

Implement rdt3.py

- **UDP is unreliable** with packet losses and corruptions
- Implement the reliable layer using **Stop-and-Wait** (rdt3.0) ARQ on top of UDP

rdt_send(),
rdt_recv(),
rdt_close()

Part 3 [9 points]

Implement rdt4.py

- **UDP is unreliable** with packet losses and corruptions
- Implement the reliable layer using **Extended-Stop-and-Wait** (rdt4.0) on top of UDP to improve performance

rdt_send(),
rdt_recv(),
rdt_close()

rdt_socket(), rdt_bind(), rdt_peer()

Part 1 – rdt1.py

Download Part1.zip

Examine and Test

- rdt1.py
  - There are six rdt_xxxxx() functions
    - rdt_socket(), rdt_bind(), rdt_peer(), rdt_send(), rdt_recv(), & rdt_close()
    - rdt_send() and rdt_recv() use __udt_send() and __udt_recv() for all communications
      - We have implemented these two internal functions, which consists of the main logic that simulates the underlying unreliable network
- Take note of the difference between rdt_bind() and rdt_peer()
  - rdt_bind() is for setting the address info ("mailbox address“) of the UDP socket; so the socket can be used for receiving message
  - rdt_peer() is for specifying the address info of the remote peer, so that we can reference to the peer’s address info easily during communication

Part 1 – rdt1.py

- Testing platform – any platform with python3 installed
  - To run the server: python3 test-server1.py localhost
  - To run the client: python3 test-client1.py localhost 〈〈filename〉〉
    - Always start server process first before executing client program otherwise, you may experience intermittent transmission errors in the client process if server process is missing
- Test cases
  - small file (around 30 KB)
  - medium size (around 500 KB)
  - large file (around 10 MB)
- Script files
  - run-simulation1.bat, run-simulation1-OSX.sh, run-simulation1-Ubuntu.sh

Submission Deadline:
April 7 17:00

Part 2 – rdt3.py

Download Part2-template.zip

> test-client2.py, test-server2.py
> run-simulation2 script files
> **rdt3.py**
> Part2-sample-output.pdf

Task

- Complete rdt3.py
  - For rdt_socket(), rdt_bind(), & rdt_peer(), you can reuse Part 1 code. But you may have to make minor changes to fit for rdt3.0 logic.
  - Add the reliable logic (rdt3.0) to
    - rdt_send(), rdt_recv(), & rdt_close()
    - must use __udt_send() and __udt_recv() for all communications

Part 2 – Simulate Losses & Errors

```
def __udt_send(sockd, peer_addr, byte_msg):
        :
        :
    else:
        #Simulate packet loss
        drop = random.random()
        if drop < __LOSS_RATE:
          #simulate packet loss of unreliable send
          print("WARNING: udt_send: Packet lost in unreliable layer!!")
          return len(byte_msg)

        #Simulate packet corruption
        corrupt = random.random()
        if corrupt < __ERR_RATE:
          err_bytearr = bytearray(byte_msg)
          pos = random.randint(0,len(byte_msg)-1)
          val = err_bytearr[pos]
          if val > 1:
              err_bytearr[pos] -= 2
          else:
              err_bytearr[pos] = 254
          err_msg = bytes(err_bytearr)
          print("WARNING: udt_send: Packet corrupted in unreliable layer!!")
          return sockd.sendto(err_msg, peer_addr)
        else:
          return sockd.sendto(byte_msg, peer_addr)
```

You are required to use __udt_send() and __udt_recv() for all message communications in the rdt_send(), rdt_recv(), and rdt_close() functions.

Part 2 – Message format

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	Network-byte order

You learn how to assemble a header with binary data thru Workshop 3

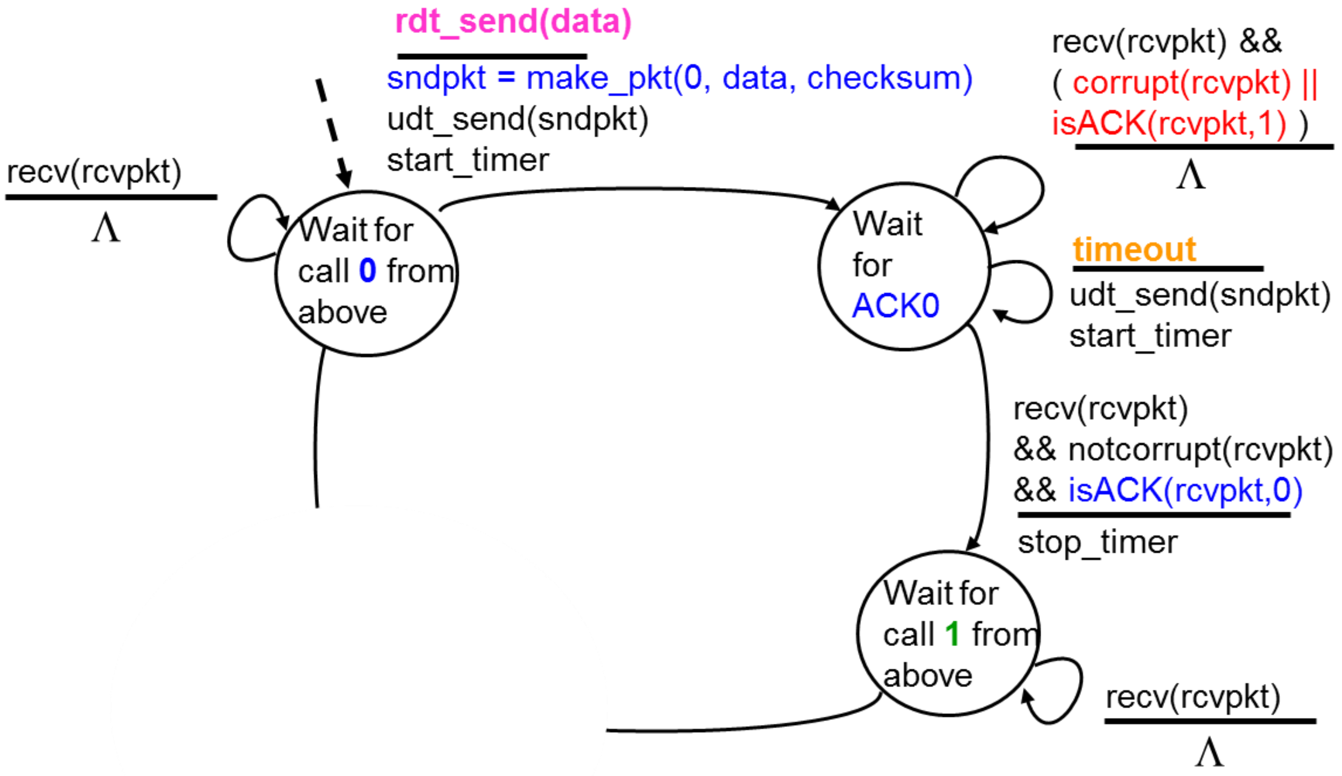
Part 2 – Checksum Calculation

```
def __IntChksum(byte_msg):

    total = 0
    length = len(byte_msg)  #length of the byte message object
    i = 0
    while length > 1:
        total += ((byte_msg[i+1] << 8) & 0xFF00) + ((byte_msg[i]) & 0xFF)
        i += 2
        length -= 2

    if length > 0:
        total += (byte_msg[i] & 0xFF)

    while (total >> 16) > 0:
        total = (total & 0xFFFF) + (total >> 16)

    total = ~total

    return total & 0xFFFF
```

This function treats the whole message as a sequence of bytes and calculates the 16-bit checksum value.

This function is also being used at the receiving end to check whether the received message is unimpaired.

Part 2 – rdt_send()




```
Make the packet
   assemble the packet header
   set checksum field to zero
   copy application data to payload field
   calculate checksum for whole packet
   store checksum value in packet header

udt_send(packet)
do
   wait for ACK or timeout
   if is timeout
      retransmit the packet
   endif
   take appro. action if packet is corrupted
   if is ACK
      check for correctness of ACK
   else is DATA
      take appropriate action
   endif
repeat until received expected ACK
```

“wait for ACK or timeout” How to do that?

Wait for ACK

- The process has to call __udt_recv() to wait for incoming packet
- __udt_recv() is a blocking call
- How can it return after waiting for a fixed duration?

Use socket timeout mode or select.select()

- We learn the technique thru Workshop 2

Part 2 – rdt_recv()

```
do
   receive(packet)
   take appro. action if packet is corrupted
   if is DATA
      if is expected packet
         send ACK
         extract message and return it to upper layer
      else
         take appropriate action
      endif
   else is ACK
      take appropriate action
   endif
repeat until received expected DATA
```

recv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq0(rcvpkt)

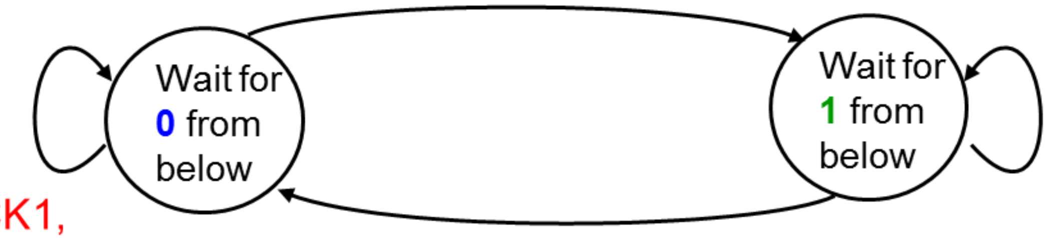
---

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)



recv(rcvpkt) &&
(corrupt(rcvpkt) ||
has_seq0(rcvpkt))

---

sndpkt = make_pkt(ACK0, chksum)
udt_send(sndpkt)

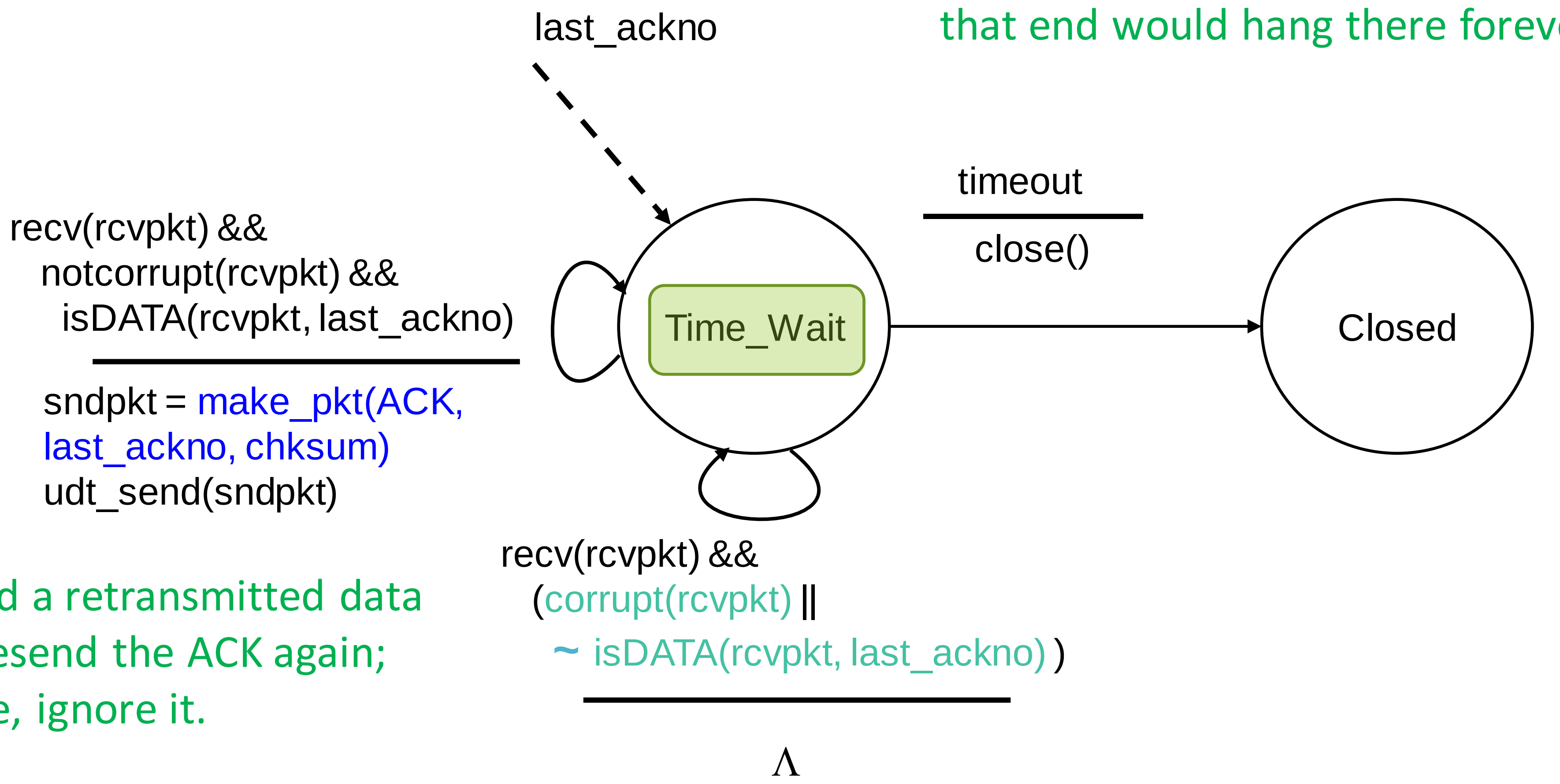
recv(rcvpkt) && notcorrupt(rcvpkt)
&& has_seq1(rcvpkt)

---

sndpkt = make_pkt(ACK1, chksum)
udt_send(sndpkt)
extract(rcvpkt, data)
return data to rdt_recv()

Part 2 – rdt_close()

The last ACK sent by this peer may be lost or corrupted; if this peer closes its socket and leaves, nobody is going to handle the retransmitted packet from the other end and that end would hang there forever !!!



If received a retransmitted data packet, resend the ACK again; otherwise, ignore it.

Implementation Requirements

Cannot use TCP

Require to simulate losses and errors by using the __udt_send() function to transmit all outgoing packets

Zero mark will be given if we find that

- Your implementation makes use of TCP
- Your implementation does not call __udt_send()

Part 2 – rdt3.py

Test

- Testing platform – any platform with python3
  - To run the server: python3 test-server1.py localhost ⟨⟨loss rate⟩⟩ ⟨⟨error rate⟩⟩
  - To run the client: python3 test-client1.py localhost ⟨⟨filename⟩⟩ ⟨⟨loss rate⟩⟩ ⟨⟨error rate⟩⟩
- Test cases
  - small file (around 30 KB)
  - large file (around 10 MB)
  - 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

- Script files
  - run-simulation2.bat, run-simulation2-OSX.sh, run-simulation2-Ubuntu.sh

Output Display

Very important – that helps you to check the correctness of your logic as well as to aid the debugging

Recommendation

- Generate an output statement whenever the RDT layer sends or receives a packet
- Generate an output statement to identify the type of packet and some control information
- Generate an output statement whenever the RDT layer detects or experiences an expected event or unexpected event or error situation

Please refer to Part2-sample-output.pdf for the sample output

Part 3 – rdt4.py

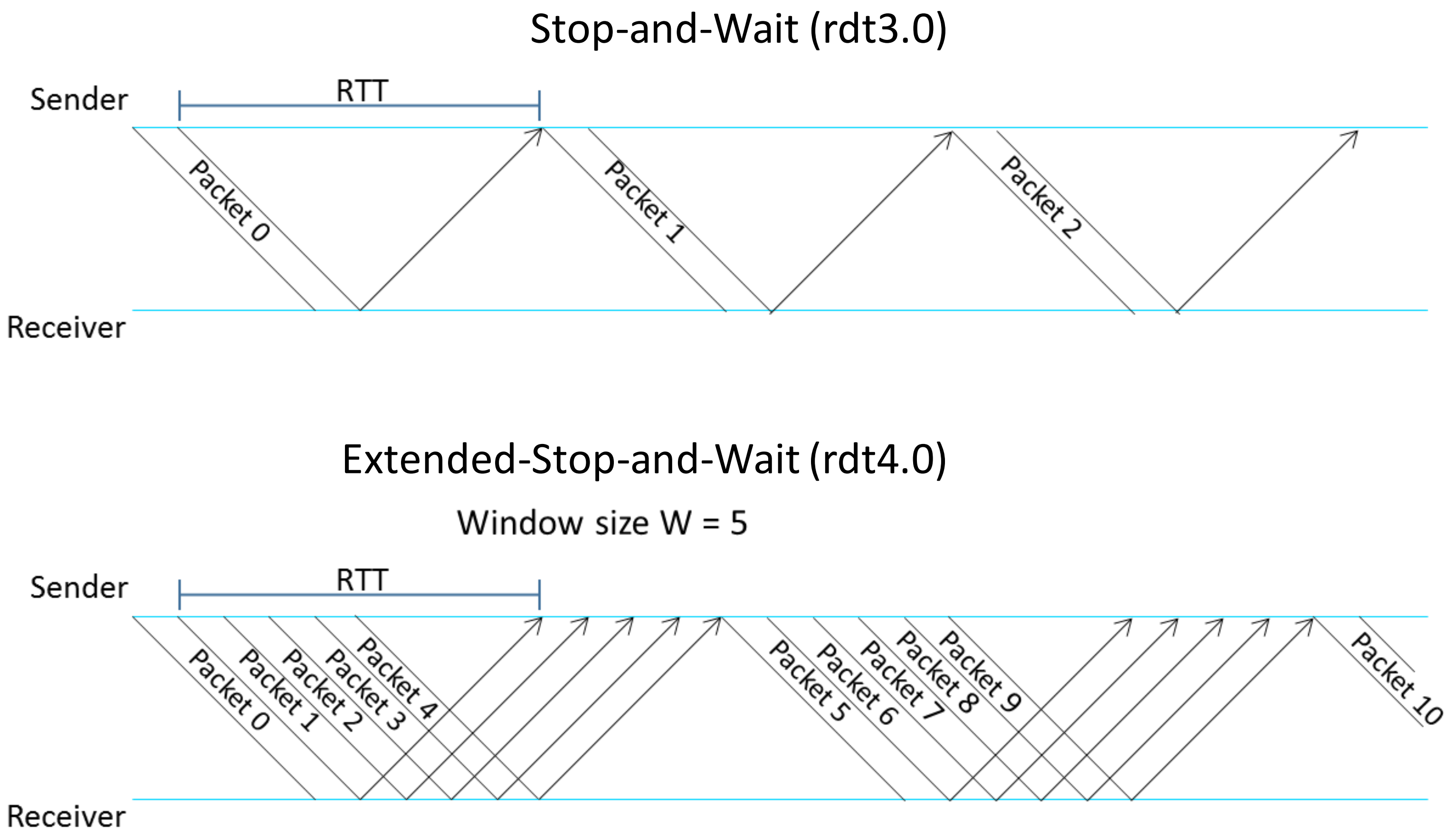
Submission Deadline: May 3 17:00

Download Part3-template.zip

Task

- Complete rdt4.py
  - For rdt_socket(), rdt_bind(), & rdt_peer(), you can reuse Part 2 implementation
  - Enhance rdt3.0 logic to include the Extended-Stop-and-Wait (rdt4.0) logic in
    - rdt_send(), rdt_recv()
  - For rdt_close()
    - The behavior of this function is the same as in Part 2 except that a peer may receive retransmitted packets with different sequence numbers within previous window

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



Part 3 – Message format (Same as Part 2)

DATA

Header				
Type	Seq #	Checksum	Payload length	Payload
1 byte	1 byte	2 bytes	2 bytes	

ACK

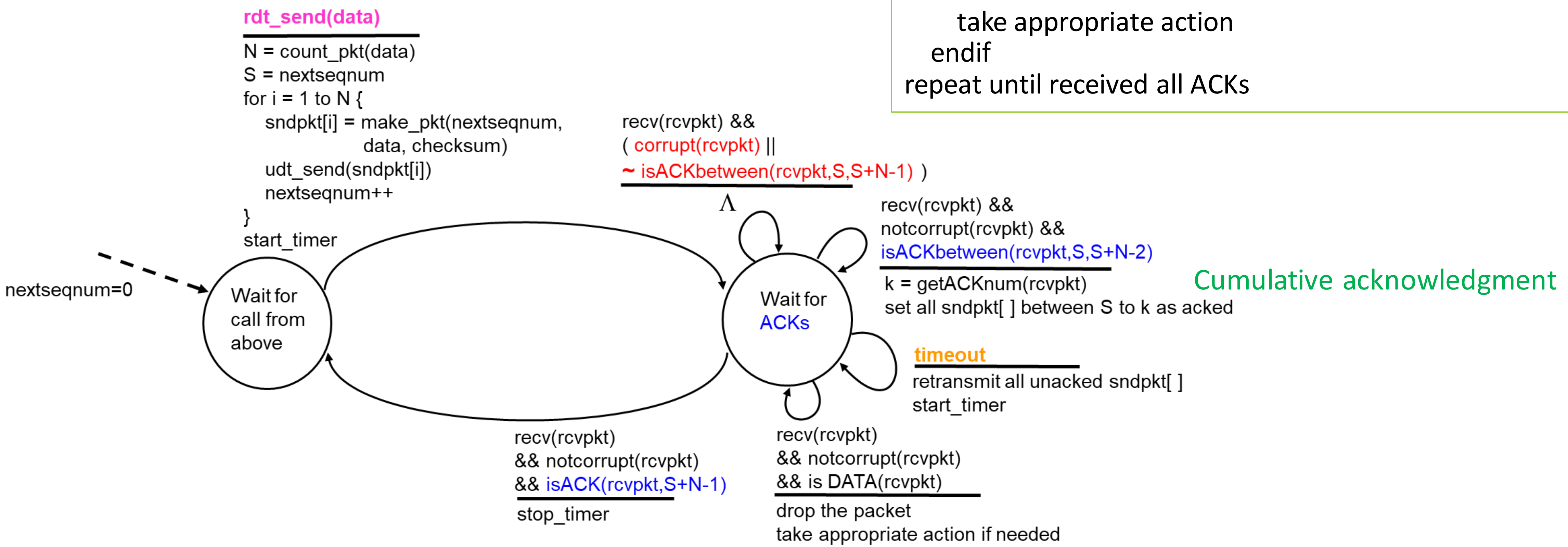
Header			
Type	Seq #	Checksum	Payload length
1 byte	1 byte	2 bytes	2 bytes

Type	ACK = 11, DATA = 12
Sequence no.	**0 to 255**
Checksum	Use __IntChksum() to calculate the checksum for the whole packet (Header + Payload)
Payload length	0 to 1000

Part 3 – rdt_send()

Count no. of packets that will be generated
Compose and send all packets; each with unique sequence number
do
    wait for ACK or timeout
    if is timeout
        retransmit all unACKed packets
    endif
    take appropriate action if packet is corrupted
    if is ACK
        check for correctness of ACK and take appro. action
    else is DATA
        take appropriate action
    endif
repeat until received all ACKs

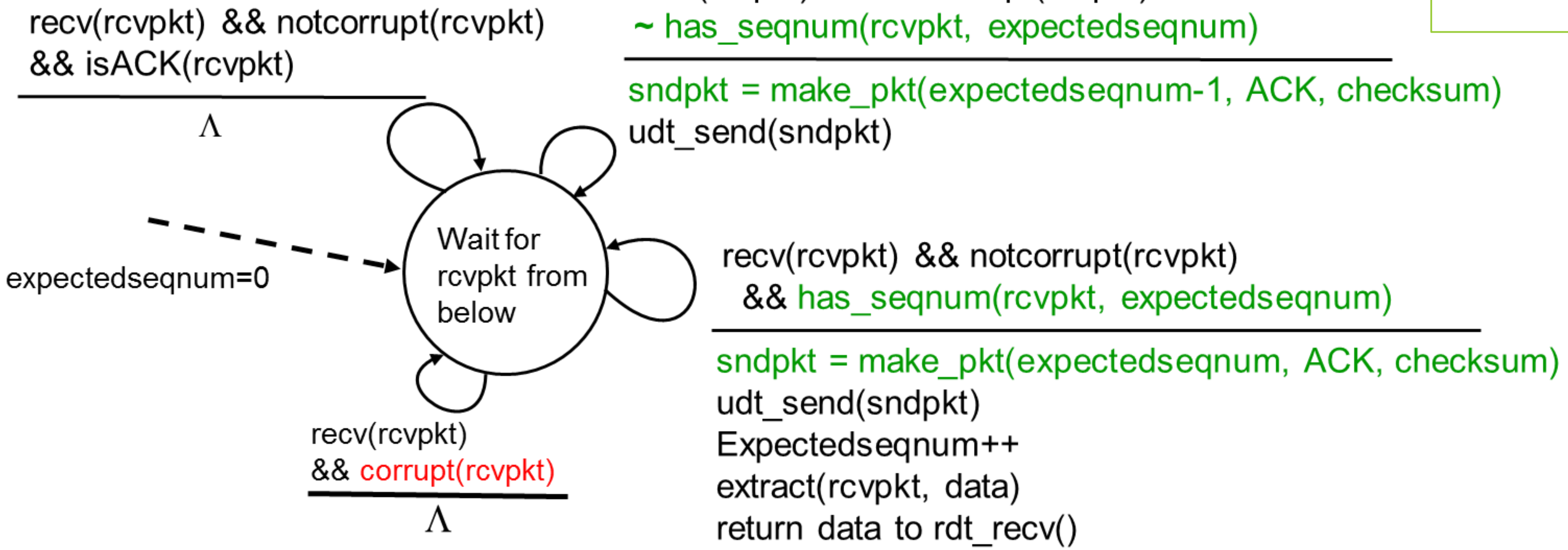
Application process calls this function to transmit a message (up to a limit of PAYLOAD $\times$ $\boldsymbol{W}$ bytes) to targeted remote process



Part 3 – rdt_recv()

```
do
   receive(packet)
   take appro. action if packet is corrupted
   if is DATA
      if is expectedseqnum
         send ACK
         return message to upper layer
      else
         take appropriate action
      endif
   else is ACK
      take appropriate action
   endif
repeat until received the expected DATA
```

Please note that the receiver at RDT layer does not know how many packets are coming. Thus, it simply accepts one packet at a time and passes it to upper layer.



Part 3 – rdt4.py

Testing platform – any platform with python3

- To run the server: python3 test-server3.py localhost ⟨⟨loss rate⟩⟩ ⟨⟨error rate⟩⟩ ⟨⟨window size⟩⟩
- To run the client: python3 test-client3.py localhost ⟨⟨filename⟩⟩ ⟨⟨loss rate⟩⟩ ⟨⟨error rate⟩⟩ ⟨⟨window size⟩⟩

Test cases

- small file (around 30 KB)
- large file (around 10 MB)
- different combinations of W, LOSS_RATE and ERR_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

Script files

- run-simulation3.bat, run-simulation3-OSX.sh, run-simulation3-Ubuntu.sh

Please refer to Part3-sample-output.pdf for the sample output

Submissions

Part 2

- Deadline – 5:00pm, April 7 (Wednesday)
- Submit file: rdt3.py

Part 3

- Deadline – 5:00pm, May 3 (Monday)
- Submit file: rdt4.py

Late submission policy:

- At most 3 days with 10% penalty for each day of delay.

Grading Policy

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≤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≤0.3). [2/9] • The program can transfer data and terminate correctly in an environment with packet loss and corruption (0.0<LOSS≤0.3, 0.0<ERROR≤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 (9 points)	• The program can transfer data and terminate correctly with W=1 in an environment without packet loss and corruption [1/9] and in an environment with loss and corruption [1.5/9]. • The program can transfer data and terminate correctly with 1<W≤10 in an environment without packet loss and corruption [2/9] and in an environment with loss and corruption (0.0<LOSS≤0.3, 0.0<ERR≤0.3) [4/9]. • 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