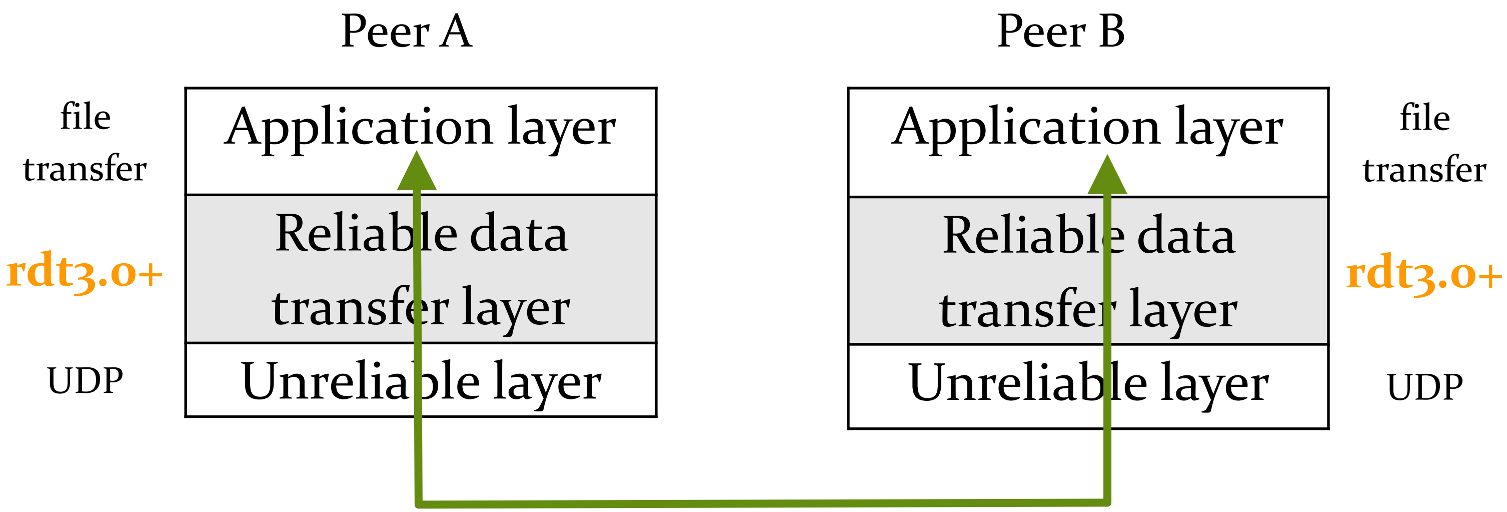
Programming Project

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

Overview



- 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 UNIX 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]

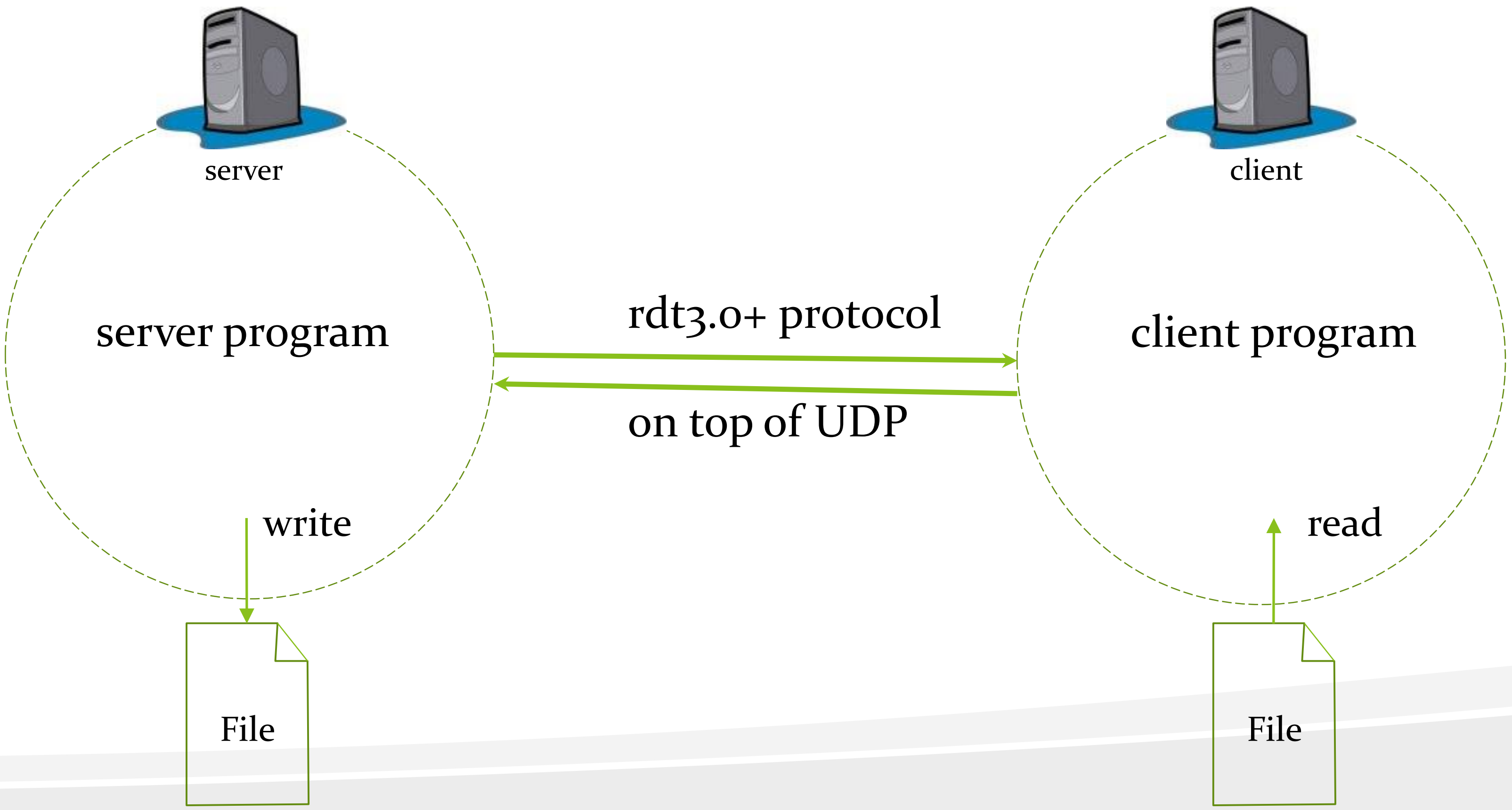
Application

client hostname or IP

```
test-server 147.8.176.14
```

server hostname or IP   upload this file

```
test-client belief.cs.hku.hk ata.png
```

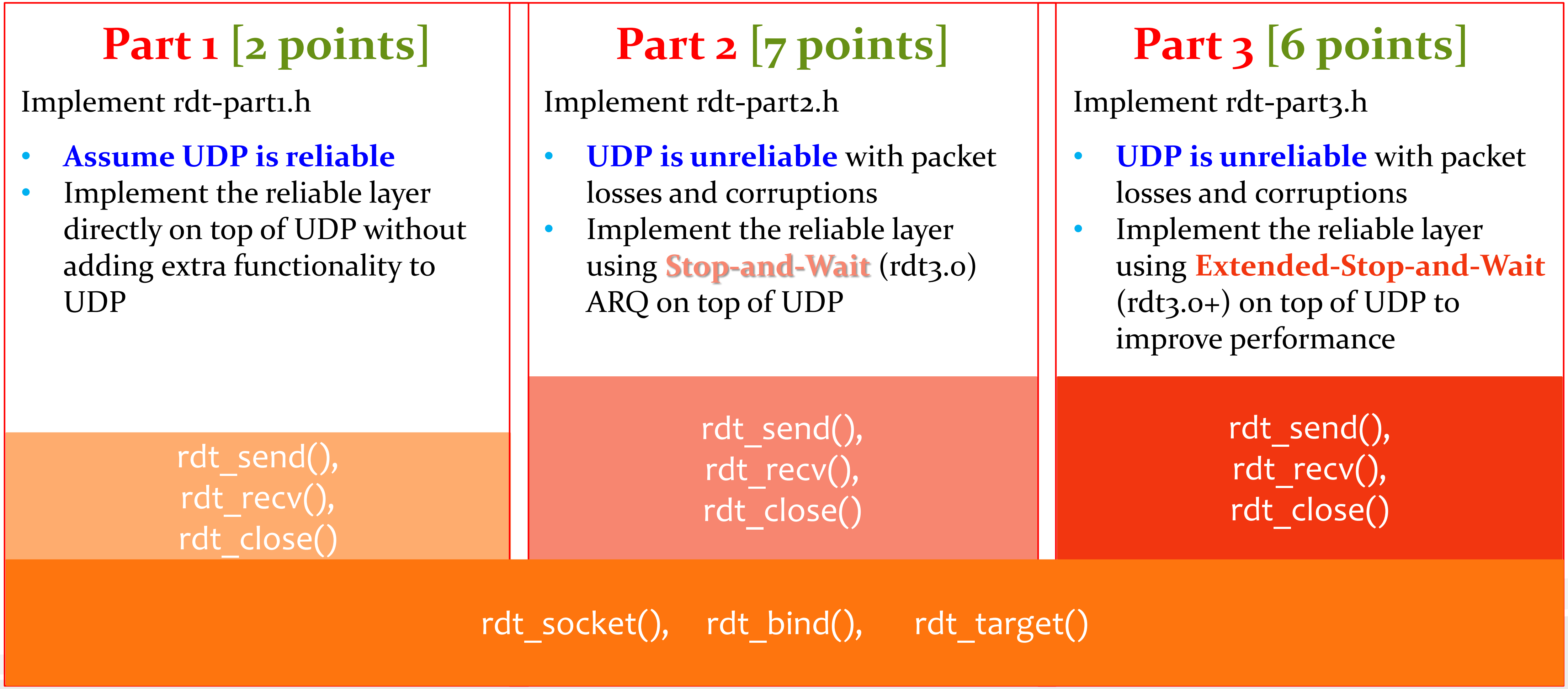


Service Interface of the RDT layer

rdt_socket()	To create a RDT socket.
rdt_bind()	To assign address info used by this RDT socket.
rdt_target()	To inform the system the address info of a remote peer that “pair-up” to this RDT socket.
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 function rdt_target() that does not exist in standard socket interface.

Structure of Assignment



Part 1 [2 points]

Implement rdt-part1.h

- **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 [7 points]

Implement rdt-part2.h

- **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 [6 points]

Implement rdt-part3.h

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

rdt_send(),
rdt_recv(),
rdt_close()

rdt_socket(), rdt_bind(), rdt_target()

Part 1 – rdt-part1.h

- Download Part1.zip
- Task
  - Complete rdt-part1.h
    - Treat those six rdt_xxxxx() functions as “wrapper” functions
      - Fill in the corresponding socket function(s) for each rdt_xxxxx()
  - Take note of the difference between rdt_bind() and rdt_target()
    - rdt_bind() is for setting the address info ("mailbox address“) of the UDP socket; so the socket can be used for receiving message
    - rdt_target() is for “associating” the address info of the remote process to this UDP socket; so the system can simple use send() function (not sendto()) to transmit a message to the target remote process

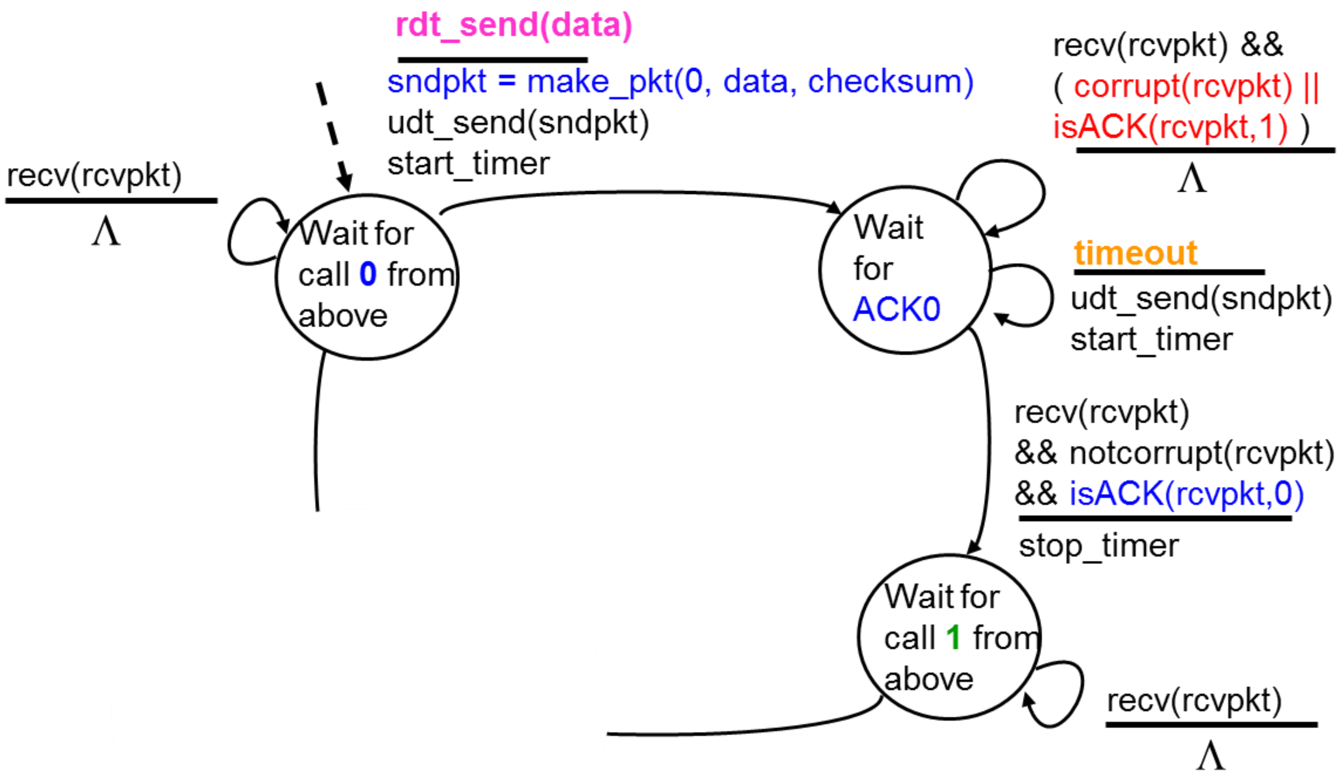
Part 1 – rdt-part.h

- Test
  - Testing platform – Ubuntu 14.04 or 12.04
    - To run the server: ./tserver localhost
    - To run the client: ./tclient localhost ⟨⟨filename⟩⟩
  - Makefile – compile the programs by just entering the command “make”
  - Test cases
    - small file (around 30 KB)
    - medium size (around 500 KB)
    - large file (around 10 MB)
  - 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

Part 2 – rdt-part2.h

- Download Part2.zip
- Task
  - Complete rdt-part2.h
    - Add necessary data structures (e.g., packet, . . .) and variables (e.g., expectedseqnum, last_acknum, . . .)
    - Add the reliable logic (rdt3.0) to
      - rdt_send()
      - rdt_recv()
      - rdt_close()

Part 2 – rdt_send()



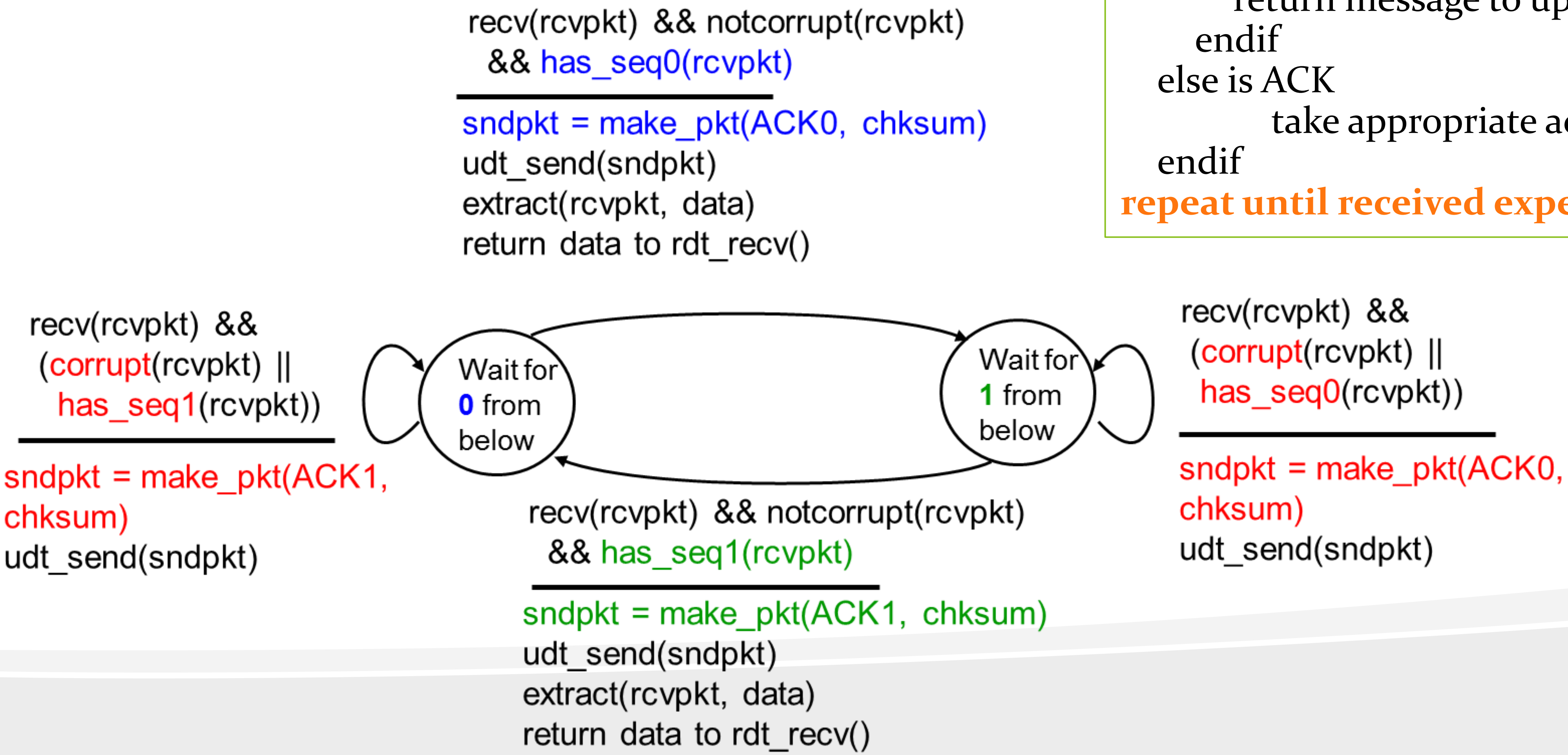
```
Make the packet
    enter control info in packet header
    copy application data to payload field
    set checksum field to zero
    calculate checksum for whole packet
    store checksum value in packet header

udt_send(packet)
do
    wait for ACK or timeout
    take appro. action if packet is corrupted
    if is timeout
        retransmit the packet
    else if is ACK
        check for correctness of ACK
    else if 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 recv() to wait for incoming packet
  - recv() is a blocking call
  - How can it return after waiting for a fixed duration?
- select() system call
    - a blocking system call that can be set up to wait for more than one socket/file descriptor(s) for I/O events
    - AND
    - a countdown timer for unblocking the process if no I/O event happened within that duration
  - Will talk about this system call in Lab 2

Part 2 – rdt_recv()

```
do
    receive(packet)
    take appro. action if packet is corrupted
    if is DATA
        check for correctness of DATA
        if is correct
            send ACK
            return message to upper layer
        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)

Wait for 0 from below

Wait for 1 from below

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

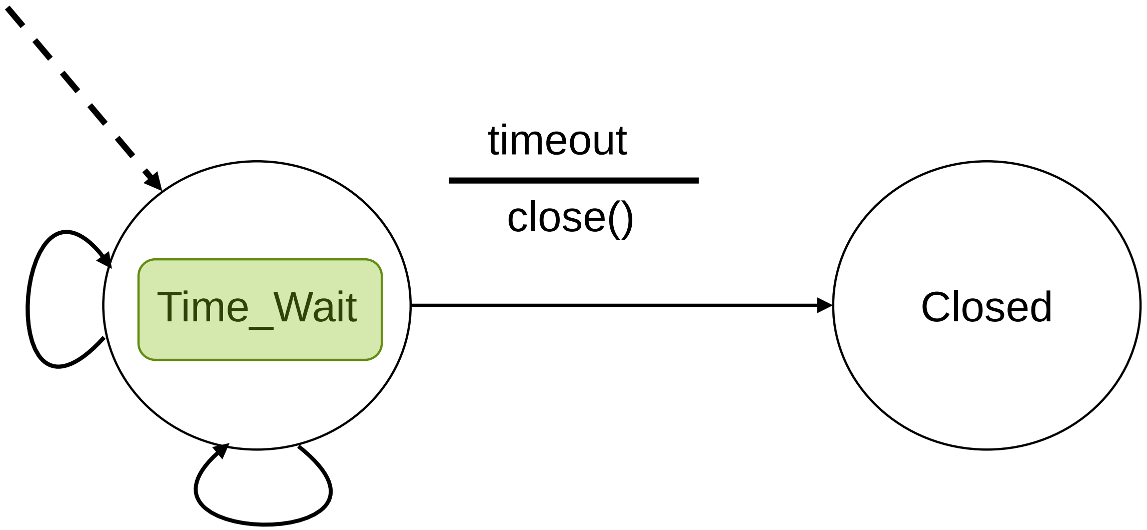
sndpkt = make_pkt(ACK0, chksum)
udt_send(sndpkt)

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

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

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 !!!



last_ackno

Time_Wait

Closed

$$\frac{\text{timeout}}{\text{close()}}$$

recv(rcvpkt) &&
  notcorrupt(rcvpkt) &&
    isDATA(rcvpkt, last_ackno)

---

sndpkt = make_pkt(ACK, last_ackno, chksum)
udt_send(sndpkt)

recv(rcvpkt) &&
  (corrupt(rcvpkt) ||
   ~ isDATA(rcvpkt, last_ackno))

Λ

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

Part 2 – Simulate Losses & Errors

```
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);
}
```

You are required to use udt_send() for all message transmissions in the rdt_send(), rdt_recv(), and rdt_close() functions.

Part 2 – Checksum Calculation

```
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;
}
```

This function treats the whole message at 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.

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 – rdt-part2.h

- Test
  - Testing platform – Ubuntu 14.04 or 12.04
  - Makefile – compile the programs by just entering the command "make"
  - Test cases
    - small file (around 30 KB)
    - large file (around 10 MB)
    - 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

Control Loss and Error Rates

- Two environment variables
  - PACKET_LOSS_RATE **0.0** (default)
  - PACKET_ERR_RATE **0.0** (default)

- To adjust loss and error rates, change these two variables before starting the server and client processes
  - Example: export PACKET_LOSS_RATE=0.1
- Two shell scripts are provided for you to adjust the LOSS_RATE and ERR_RATE

- To run server and client:
  - sh run-server.sh ⟨⟨LOSS_RATE⟩⟩ ⟨⟨ERR_RATE⟩⟩ localhost
  - sh run-client.sh ⟨⟨LOSS_RATE⟩⟩ ⟨⟨ERR_RATE⟩⟩ localhost ⟨⟨filename⟩⟩

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 Figures 2 & 3 of the document “Project-sample-output.pdf” for the sample output

Useful Tip

- At test-server.c and test-client.c, there is a statement that adjusts the random seed

```
srand(time(NULL));
```

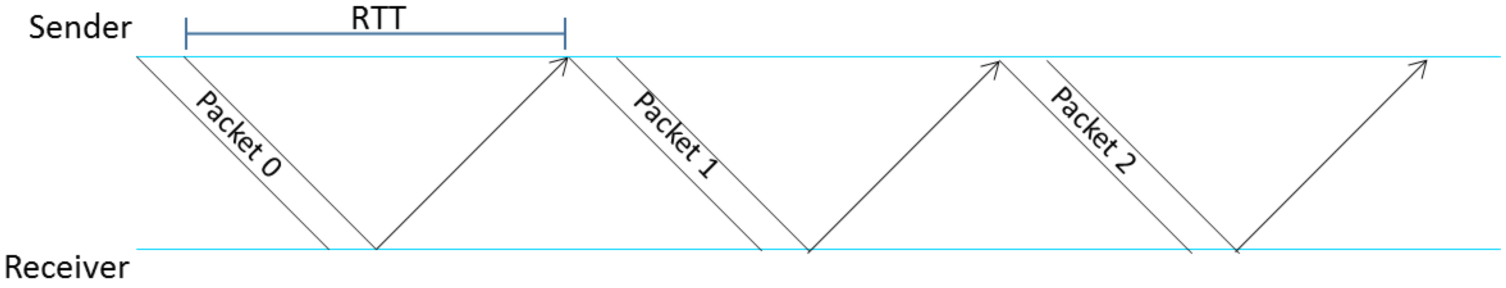
- The purpose of this statement is to get a different random number sequence for each execution
- However, this also affects your debugging as you might not be able to regenerate the losses and errors at the same timing
- Remove or comment it out when you start working on debugging
  - This helps to get the same random number sequence for each run
- Add it back in the final tests

Part 3 – rdt-part3.h

- Download Part3.zip
- Task
  - Complete rdt-part3.h
    - Enhance the rdt3.0 logic to include the Extended-Stop-and-Wait (rdt3.0+) logic in
      - rdt_send()
      - rdt_recv()
      - 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

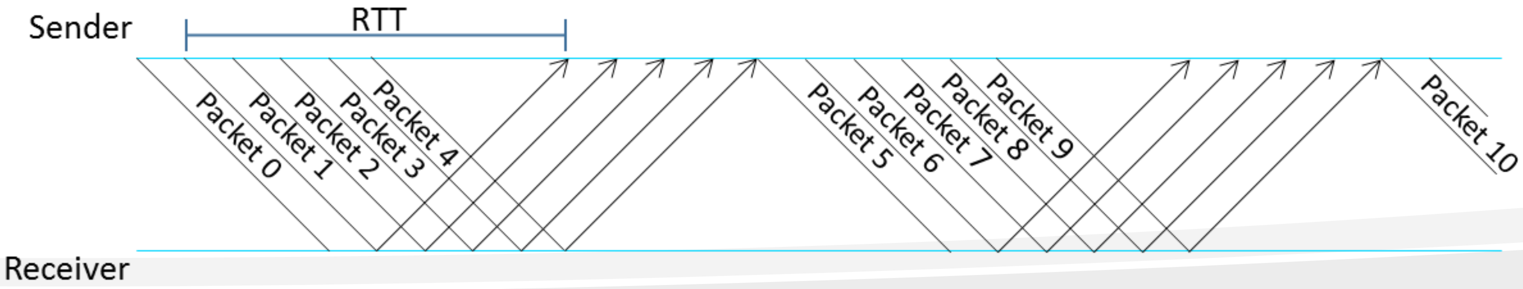
Extended-Stop-and-Wait (rdt3.0+)

Stop-and-Wait (rdt3.0)



Extended-Stop-and-Wait (rdt3.0+)

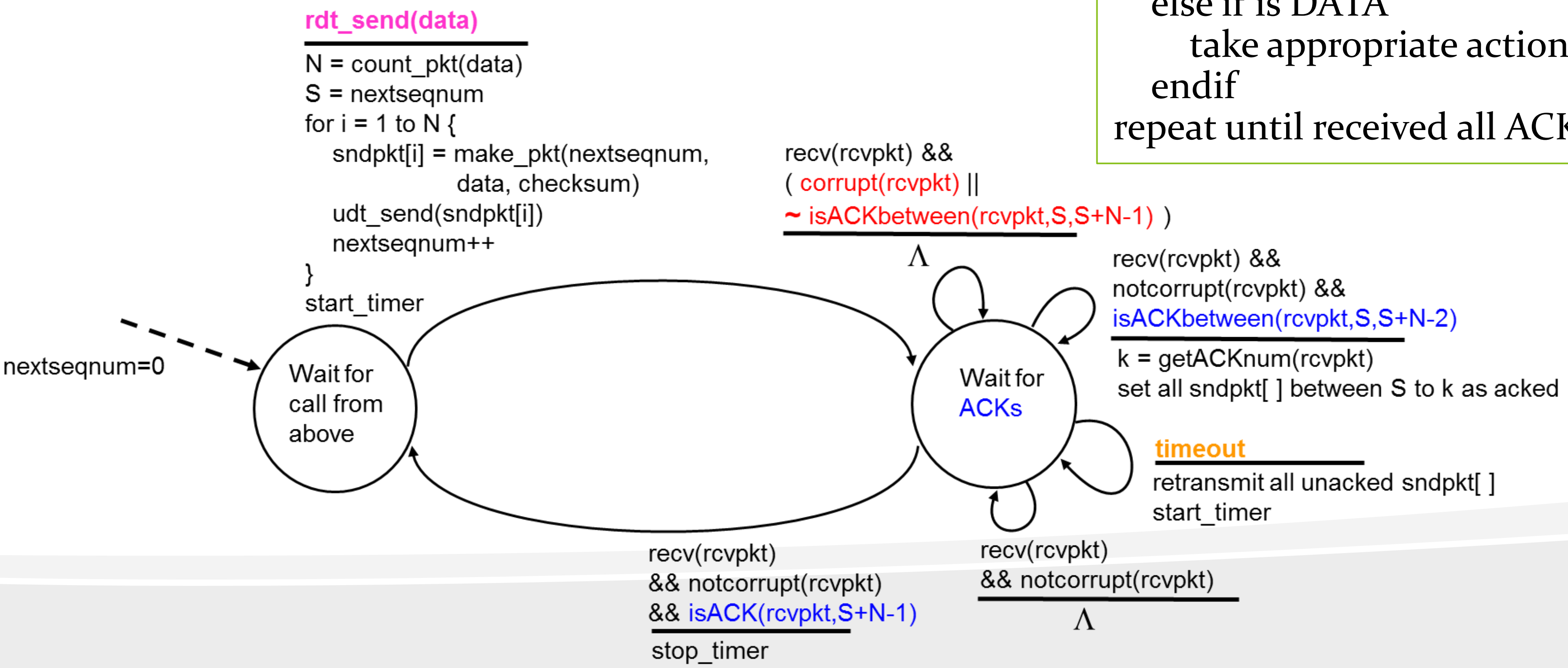
Window size W = 5



Part 3 – rdt_send()

Application process calls this function to transmit a message (up to a limit of **1000** × ***W*** bytes) to targeted remote process

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



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
```

nextseqnum=0

Wait for call from above

Wait for ACKs

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

timeout

retransmit all unacked sndpkt[ ]
start_timer

recv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,S+N-1)

stop_timer

recv(rcvpkt) && notcorrupt(rcvpkt)

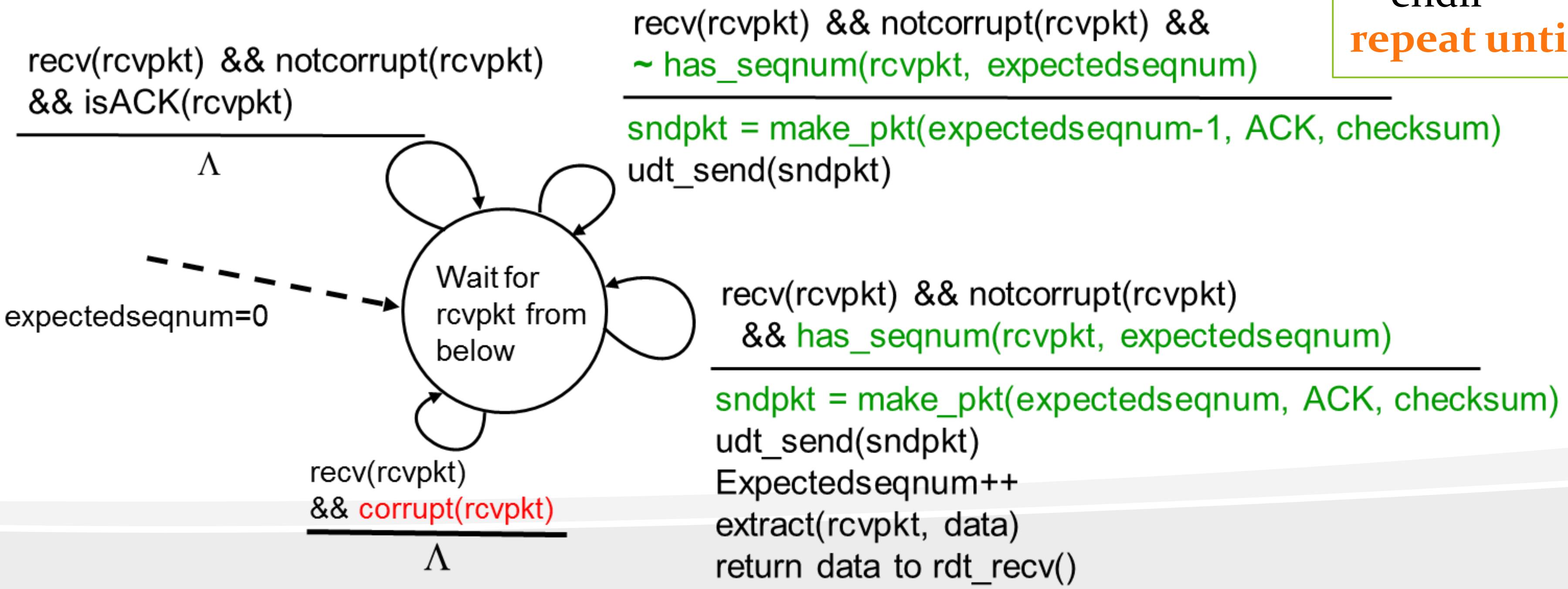
Λ

Cumulative acknowledgment

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 not the expected one
            send duplicate ACK
        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.



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

Part 3 – rdt-part3.h

- Implementation requirement, Output display, Control Loss and Error rates
  - Same as Part 2
- 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_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

Part 3 – Control Window Size

- A constant is defined in rdt-part3.h
  - #define W 5 //For Extended S&W - define pipeline window size

- To adjust the W, you have to change it in rdt-part3.h AND recompile the programs again (using “make”)

Submissions

- Part 1 [Optional]
  - (Soft) Deadline – 5:00pm, March 2, 2015 (Monday)
  - Submit file: rdt-part1.h
- Part 2 [Optional]
  - (Soft) Deadline – 5:00pm, March 30, 2015 (Monday)
  - Submit file: rdt-part2.h
- Part 1 & Part 2 submissions are optional but highly recommended
  - Will accept submissions even after the deadlines but will be labelled as *late* (with no penalty)
- Part 3 [Compulsory]
  - (Hard) Deadline – 5:00pm, April 15, 2015 (Wednesday)
  - Submit file: rdt-part3.h
- Final Cut-off for all three parts is 5:00pm, April 15, 2015 (Wednesday)

Grading Policy

- The tutor will first test your Final submission after project deadline
- If your Part 3 submission fully complies with the project specification, you'll get all the 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 the marks for Parts 1 & 2.
- Otherwise, tutor will test your Part 1 submission (if any) to give marks.
- 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.

Grading Policy

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≤0.3, 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≤0.3**). [1.5/7] • The program can transfer data and terminate correctly in an environment with packet loss and corruption (**0.0<LOSS≤0.3, 0.0<ERR≤0.3**). [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≤9** in an environment without packet loss and corruption [1.5/6] and in an environment with loss and corruption (**0.0<LOSS≤0.3** **0.0<ERR≤0.3**) [2.5/6].