Name_______________ Student ID____________________ Department/Year___________

Midterm Examination

Introduction to Computer Networks (Online)
Class#: EE 4020, Class-ID: 901E31110
Spring 2020

10:20-12:10 Thursday
April 23, 2020

Cautions

1. There are in total 100 points to earn. You have 100 minutes to answer the questions. Skim through all questions and start from the questions you are more confident with.
2. Use only English to answer the questions. Misspelling and grammar errors will be tolerated, but you want to make sure with these errors your answers will still make sense.

1. (Golang) Consider the following Go program: server-midterm-1.go. Execute the server-midterm-1.go first and then client-101.go.

server-midterm-1.go

```
package main

import "fmt"
import "bufio"
import "net"

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    fmt.Println("Launching server...")
    ln, _ := net.Listen("tcp", ":<your port#>")
    conn, _ := ln.Accept()
    defer ln.Close()
    conn.Close()

    scanner := bufio.NewScanner(conn)
    message := “”
    if scanner.Scan() {
        message = scanner.Text()
        fmt.Println(message)
    }

    writer := bufio.NewWriter(conn)
    newline := fmt.Sprintf("%d bytes received\n", len(message))
    _, errw := writer.WriteString(newline)
    check(errw)
    writer.Flush()
}
```

client-101.go

```
package main

import "fmt"
import "bufio"
import "net"


func check(e error) {
   if e != nil {
      panic(e)
   }
}

func main() {
   conn, errc := net.Dial("tcp", "127.0.0.1:<your port#>")
   check(errc)
   defer conn.Close()

   writer := bufio.NewWriter(conn)
   len, errw := writer.WriteString("Hello World!\n")
   check(errw)
   fmt.Printf("Send a string of %d bytes\n", len)
   writer.Flush()

   scanner := bufio.NewScanner(conn)
   if scanner.Scan() {
      fmt.Printf("Server replies: %s\n", scanner.Text())
   }
}
```

(1) Tell the output on screen of server-midterm-1.go (1%).
(2) Tell the output on screen of client-101.go (1%).
(3) Explain why we see the output on screen server-midterm-1.go (3%).
(4) Explain why we see the output on screen client-101.go (3%).

Sample Solution:

(1) “Launching server...”

(2) “Send a string of 13 bytes”

(3) conn.Close() closes the conn socket before the reader/writer wrap. No messages can be read or written back onto the socket.

(4) conn is closed from the server side. Therefore, the client is not able to receive a message back, and not able to print the message on the screen.

2. (Golang) Consider the following Go program: server-midterm-2.go. Execute the server-midterm-2.go first and then client-101.go.

server-midterm-2.go

```
package main

import "fmt"
import "bufio"
import "net"

func check(e error) {
    if e != nil {
        panic(e)
    }
}

func main() {
    fmt.Println("Launching server...")
    ln, _ := net.Listen("tcp", ":<your port#>")
    conn, _ := ln.Accept()
    ln.Close()
    defer conn.Close()

    scanner := bufio.NewScanner(conn)
    message := “”
    if scanner.Scan() {
        message = scanner.Text()
        fmt.Println(message)
    }

    writer := bufio.NewWriter(conn)
    newline := fmt.Sprintf("%d bytes received\n", len(message))
    _, errw := writer.WriteString(newline)
    check(errw)
    writer.Flush()
}
```

(1) Tell the output on screen of server-midterm-2.go (1%).
(2) Tell the output on screen of client-101.go (1%).
(3) Explain why we see the output on screen server-midterm-2.go (3%)..
(4) Explain why we see the output on screen client-101.go (3%).

Sample Solution:

(1) “Launching server...”
    “Hello World!”
(2) “Send a string of 13 bytes”
    “Server replies: 12 bytes received”
(3) ln.Close() closes the listening socket early, but the conn socket has already been established. The reader/writer wrapping and message reading/writing will go on.
(4) conn is open for sending and receiving of messages, no problem.

3. (Golang) Consider the following Go program: server-midterm-3.go. Execute the server-midterm-3.go first. Then start two extra terminals. Execute client-102.go on the two terminals back to back.

server-midterm-3.go

```
package main

import "fmt"
import "bufio"
import "net"
import "time"

func check(e error) {
   if e != nil {
      panic(e)
   }
}

func handleConnection (c net.Conn) {
   reader := bufio.NewReader(c)
   message, errr := reader.ReadString('\n')
   check(errr)
   fmt.Printf("%s", message)

   time.Sleep(10 * time.Second)

   writer := bufio.NewWriter(c)
   newline := fmt.Sprintf("%d bytes received\n", len(message))
   _, errw := writer.WriteString(newline)
   check(errw)
   writer.Flush()
}

func main() {
   fmt.Println("Launching server...")
   ln, _ := net.Listen("tcp", ":<your port#>")
   defer ln.Close()
   i := 1
   for {
      conn, _ := ln.Accept()
      defer conn.Close()

      fmt.Printf("%d ", i)
      go handleConnection(conn)
      i++
   }
}
```

client-102.go

```
package main

import "fmt"
import "bufio"
import "net"


func check(e error) {
   if e != nil {
      panic(e)
   }
}

func main() {
   conn, errc := net.Dial("tcp", "127.0.0.1:<your port#>")
   check(errc)
   defer conn.Close()

   writer := bufio.NewWriter(conn)
   len, errw := writer.WriteString("Hello World!\n")
   check(errw)
   fmt.Printf("Send a string of %d bytes\n", len)
   writer.Flush()
```

```
    reader := bufio.NewReader(conn)
    message, errr := reader.ReadString('\n')
    check(errr)
    fmt.Printf("Server replies: %s", message)
}
```

(1) Tell the output on screen of server-midterm-3.go (1%).
(2) Tell the time gap between the output lines on screen of server-midterm-3.go (2%).
(3) Explain why we see the time gap on screen server-midterm-3.go (3%).

Sample Solution:

(1) "Launching server..."
    "1 Hello World!"
    "2 Hello World!"
(2) The time gap between "1 Hello World!" and "2 Hello World!" is very small.
(3) handleConnection() being a goroutine, allows concurrent running of the function. The 2nd execution of client-102.go forks another handleConnection() process which prints the message before entering the time.Sleep(), and back to back of the 1st client-102.go's message printout.

4. (Golang) Consider the following Go program: server-midterm-4.go. Execute the server-midterm-4.go first. Then start two extra terminals. Execute client-102.go on the two terminals back to back.

server-midterm-4.go

```
package main

import "fmt"
import "bufio"
import "net"
import "time"

func check(e error) {
   if e != nil {
      panic(e)
   }
}

func handleConnection (c net.Conn) {
   reader := bufio.NewReader(c)
   message, errr := reader.ReadString('\n')
   check(errr)
   fmt.Printf("%s", message)

   time.Sleep(10 * time.Second)

   writer := bufio.NewWriter(c)
   newline := fmt.Sprintf("%d bytes received\n", len(message))
   _, errw := writer.WriteString(newline)
   check(errw)
   writer.Flush()
}

func main() {
   fmt.Println("Launching server...")
   ln, _ := net.Listen("tcp", ":<your port#>")
   defer ln.Close()
```

```
    i := 1
    for {
        conn, _ := ln.Accept()
        defer conn.Close()

        fmt.Printf("%d ", i)
        handleConnection(conn)
        i++
    }
}
```

(1) Tell the output on screen of server-midterm-4.go (1%).
(2) Tell the time gap between the output lines on screen of server-midterm-4.go (2%).
(3) Explain why we see the time gap on screen server-midterm-4.go (3%).

Sample Solution:

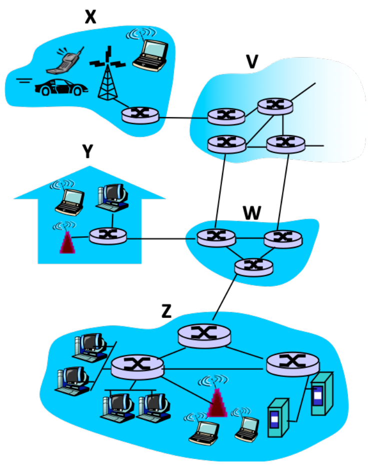
(1) "Launching server..."
"1 Hello World!"
"2 Hello World!"
(2) The time gap between "1 Hello World!" and "2 Hello World!" is substantially long – about the sleep time 10 seconds.
(3) handleConnection() now not a goroutine, will need to sleep through the 10 seconds time.Sleep(), triggered by the 1st execution of client-102.go. The 2nd execution of client-102.go will enter the handleConnection() function and print the message at least 10 seconds apart from the 1st printout.

5. (Golang) Compare and contrast server-midterm-3.go and server-midterm-4.go.
   (1) As a user, would you prefer the service by server-midterm-3.go or server-midterm-4.go (1%)? And why (3%)?
   (2) As a service provider, would you prefer your server running server-midterm-3.go or server-midterm-4.go (1%)? And why (3%)?

Sample Solution

Take your pick and justify your answer

6. (Overview) Consider a micro-Internet consisting of 5 subnets, namely V, W, X, Y and Z as labeled below.



(1) Which of the subnets should be classified as the Internet edge? (1%)

(2) Which of the subnets should be classified as the Internet core? (1%)

Sample Solution:

(1) X, Y, Z

(2) V, W

7. (Overview) Based on your understanding of packet switching and circuit switching principle, select the keywords that fit better the characteristics of a circuit switching network. (6%)

    (a) Contention
    (b) Congestion
    (c) Idle resource
    (d) Reservation
    (e) Call setup
    (f) Delay guarantee

Sample Solution:

(c)(d)(e)(f)

8. (Overview) Internet engineers like to group the protocols into layers. This is known as the layered reference model. Below are a few that may (or may not) be a part of the reference model.

   (a) Overview
   (b) Application
   (c) Session
   (d) Transport
   (e) Transmission
   (f) Network
   (g) Routing
   (h) Link
   (i) Wireless
   (j) Multimedia
   (k) Physical

   (1) List the Internet protocol layers top down. (1%)
   (2) Discuss the benefits and drawbacks of the layered reference model. (4%)

Sample Solution:

(1) (b) (d) (f) (h) (k)
    OK to have (c) in between (b) (d) (The original OSI model includes (c))

(2) Benefits:
    1. ease of discussion – such modularization makes it easy to see the relationships between the modules in a complex system.
    2. ease of maintenance/progression – such modularization makes it easy to update a module with minimum influence to other modules in a complex system.

    Drawback:
    1. initial learning curve – not easy for a newbie to understand the engineer's language and therefore hard to identify the module to blame
    2. performance suboptimal – hard to find exploits spanning multiple modules to optimize performance

9. (Application) Complexity at the edge, is another design principle the Internet engineers exercise.

   (1) Tell what it means to leave the complexity at the edge. (1%)
   (2) Tell the benefits of leaving the complexity at the edge. (2%)
   (3) Recall the protocol designed with the principle in mind. (1%)

Sample Solution:

(1) Pushing functionality that's yet to evolve in the future to the edge of the Internet.
(2) It is easy to evolve/upgrade the functionality without the need to reboot the core that some parts of the Internet may depend critically on. Or to keep the core simple and therefore fast and reliable.
(3) DNS

10. (Application) HTTP with non-persistent connection requires 2RTT (round trip time) + Tx (file transmission time) to download a Web object. For simplicity, let's assume all objects are the same size and the connection closing time is negligible. For a page that contains 1 base html object and 10 additional embedded objects, the total page response time will be 22RTT + 11Tx.

    What's nice about modern HTTP (1.1 and beyond) is this – it implements the persistent connection with pipelining mode, which offers a lower page response time in general. How much lower? This depends on how widespread the objects are. Now assume the Web page is requested from a Web client that allows one open connection at a time. Estimate the page response time for the following scenarios and derive its general form.

    (1) Tell the total page response time when the main html and the 10 embedded objects are on the same physical server. (1%)
    (2) Tell the total page response time when the main html, 5 of the embedded objects, and the rest of the embedded objects are on 3 different physical servers. (1%)
    (3) Tell the total page response time when the main html, 1 of the embedded objects, and the rest of the embedded objects are on 3 different physical servers. (1%)
    (4) Tell the total page response time when the main html and the embedded objects, are on 5 different physical servers. (1%)
    (5) Tell the total page response time when the main html, and all 10 additional embedded objects are on 11 different physical servers. (1%)
    (6) Derive the total page response time in general form when the main html and the k additional embedded objects spread on n different physical servers (n > 1). (2%)
    (7) For a frequently requested page containing multiple embedded objects where the object contents are static, would you rather (a) copy the embedded objects over and store all objects together or (b) have the client redirected to each of the remote servers? (1%) And why? (1%)

Sample Solution:

(1) 3RTT+11Tx
(2) 2RTT+Tx+2RTT+5Tx+2RTT+5Tx = 6RTT+11Tx
(3) 2RTT+Tx+2RTT+aTx+2RTT+(10-a) Tx = 6RTT+11Tx (2RTT to each physical server)
(4) 5*(2RTT)+11Tx = 10RTT+11Tx (2RTT to each physical server)
(5) 2RTT+Tx+10*(2RTT+Tx) = 22RTT+11Tx
(6) 2nRTT+(k+1)Tx.

(7) Copy them over locally. To minimize the page response time. Or just take your pick and justify for yourself.

11. (Application) We know smtp.gmail.com is the mail server of gmail.com and ns[1-4].google.com are the authoritative DNS servers for google.com. The questions are about the following DNS RRs where the entry types are masked.

(a)	(gmail.com,	smtp.gmail.com,	---,	2 days)
(b)	(smtp.gmail.com,	108.177.125.10,	---,	2 days)
(c)	(google.com,	ns1.google.com,	---,	2 days)
(d)	(google.com,	ns2.google.com,	---,	2 days)
(e)	(ns1.google.com,	216.239.32.10,	---,	2 days)
(f)	(ns2.google.com,	216.239.34.1,	---,	2 days)

(1) Which of the above are type A entries? (1%)
(2) Which of the above are type MX entries? (1%)
(3) Which of the above are type NS entries? (1%)
(4) Which of the above are type CNAME entries? (1%)
(5) Which entries are stored at the .com TLD servers? (1%)
(6) Which entries are stored at the authoritative DNS servers? (1%)

Sample Solution:

(1) (b)(e)(f)
(2) (a)
(3) (c)(d)
(4) none
(5) (c)(d)(e)(f)
(6) (a)(b)

12. (Application) Let's design the PetTube a video streaming platform that allows users to upload and share videos of their beloved pets. Similar to YouTube, we build our own CDN, which consists of 9 edge servers. Stored on the edge servers are the videos uploaded by the users. When a user click on a pet video after browsing, a list of edge servers containing the requested pet video is returned to the PetTube client.

    Implemented also in the client is the measurement of (RTT, std of RTT, BW, std of BW) to each edge server. RTT: round trip time. BW: available bandwidth. std: standard deviation. The 4-tuple for the 9 edge servers are as follows.

    (a) (10ms, 1ms, 5Mbps, 4.2Mbps)
    (b) (10ms, 1ms, 3Mbps, 0.34Mbps)
    (c) (10ms, 1ms, 1Mbps, 0.021Mbps)
    (d) (100ms, 0.01ms, 5Mbps, 4.2Mbps)
    (e) (100ms, 0.01ms, 3Mbps, 0.34Mbps)
    (f) (100ms, 0.01ms, 1Mbps, 0.021Mbps)
    (g) (50ms, 0.1ms, 5Mbps, 4.2Mbps)
    (h) (50ms, 0.1ms, 3Mbps, 0.34Mbps)
    (i) (50ms, 0.1ms, 1Mbps, 0.021Mbps)

    Provided the list of edge server (S) containing the requested video, tell which server you'd like the PetTube client to stream the video from.

    (1) S: {a, b, c} (1%)
    (2) S: {c, f, i} (1%)
    (3) S: {c, d, h} (1%)
    (4) Write your intuition out as an algorithm. (2%)
    (5) Apply your algorithm on S: {a, g, f}. (1%)
    (6) Are you comfortable with the result and why? (1%)

Sample Solution:

This question is very open ended. Feel free to speak of your mind.

13. (Transport) UDP/TCP checksum allows detection of bit errors in a packet. Provided the following two 16-bit data sequences in a packet, find the corresponding UDP/TCP checksum.

    (1) (1111 1111 1111 1111) and (0000 0000 0000 0000) (1%)
    (2) (0000 0000 1111 1111) and (1111 1111 0000 0000) (1%)
    (3) (0111 1111 1111 1111) and (1000 0000 0000 0000) (1%)

Sample Solution:

(1) (0000 0000 0000 0000)
(2) (0000 0000 0000 0000)
(3) (0000 0000 0000 0000)

14. (Transport) Find 5 pairs of 16-bit data sequences such that their UDP/TCP checksums are identical to the checksum of this pair – (1110 0110 0110 0110) and (1101 0101 0101 0101). (1%)(1%)(2%)(2%)(2%)

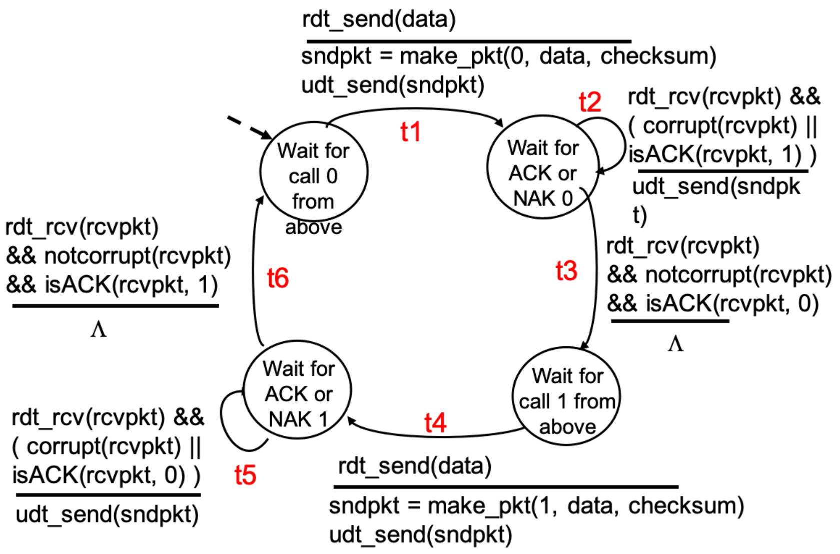
Sample Solution:

(1) (1101 0101 0101 0101) and (1110 0110 0110 0110) flipping 16 bits
(2) (1100 0110 0110 0110) and (1111 0101 0101 0101) flipping 2 bits
(3) (1111 0110 0110 0110) and (1100 0101 0101 0101) flipping 2 bits
(4) (1110 0100 0110 0110) and (1101 0111 0101 0101) flipping 2 bits
(5) (1110 0111 0110 0110) and (1101 0100 0101 0101) flipping 2 bits

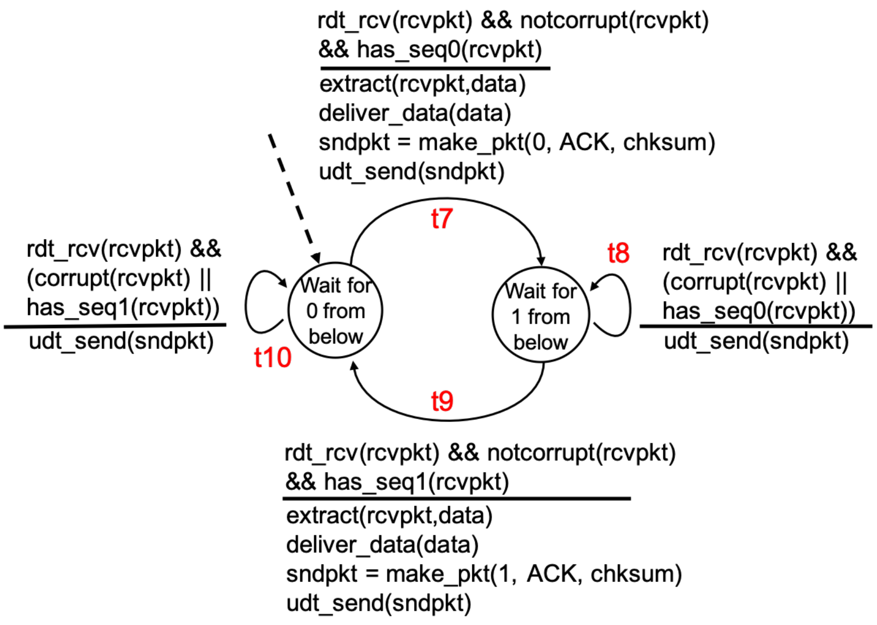
In case (2)-(5), we see numerous 2-bit flips to slip through the UDP/TCP checksum test. 2-bit flips aren’t exactly rare. Having the flips at specific positions is however harder.

15. (Transport) Provided below are the FSMs of rdt 2.2 sender and receiver. Indicate the order of the transitions (in terms of t1, t2, …, t12) taking places until the sender and receiver stabilize for the following scenarios. All scenarios in this problem set inherits the no packet loss assumption of rdt 2.2.

<u>rdt 2.2 sender:</u>


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
t1
t2
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 1))
udt_send(sndpkt)
Wait for call 0 from above
Wait for ACK or NAK 0
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 1)
Λ
t6
t3
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt, 0)
Λ
Wait for ACK or NAK 1
Wait for call 1 from above
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt, 0))
udt_send(sndpkt)
t5
t4
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)

<u>rdt 2.2 receiver:</u>


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(0, ACK, chksum)
udt_send(sndpkt)
t7
t8
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq1(rcvpkt))
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq0(rcvpkt))
udt_send(sndpkt)
t10
t9
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(1, ACK, chksum)
udt_send(sndpkt)

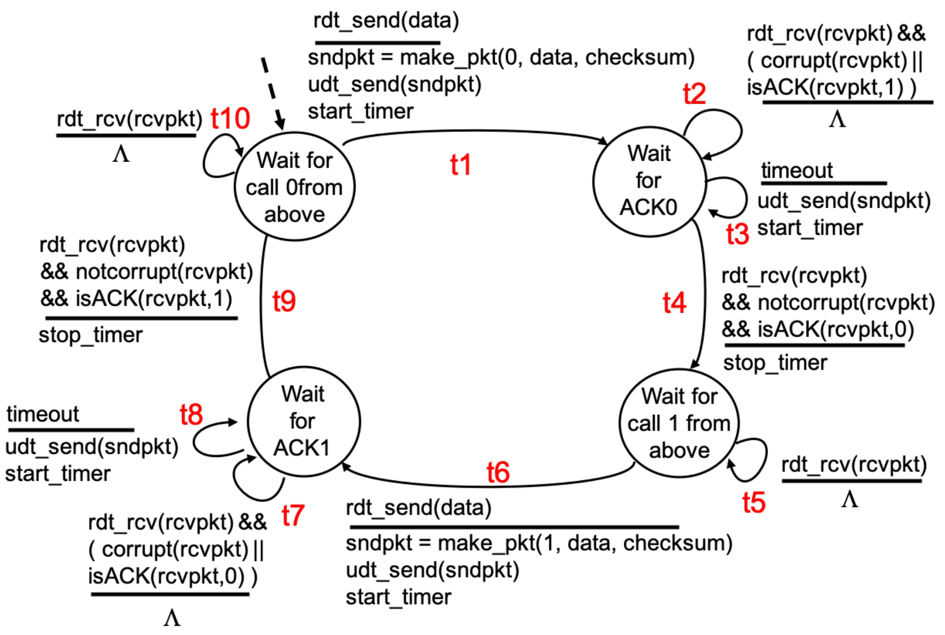
(1) Scenario 1: Both sender and receiver start from the initial state. The sender gets a call from above to send 1 data packet and there is no bit error at all. (1%)
(2) Scenario 2: Continue from Scenario 1. The sender gets another call from above to send 1 data packet and there is a bit error. There are no bit errors afterwards. (1%)
(3) Scenario 3: Both sender and receiver start from the initial state. The sender gets a call from above to send 1 data packet and there is a bit error in the data packet going to the receiver. There are no more bit errors afterwards. (1%)
(4) Scenario 4: Continue from Scenario 3. The sender gets another call from above to send 1 data packet and there is a bit error in the ACK 1 packet coming back to the sender. There are no more bit errors afterwards. (1%)
(5) Scenario 5: Continue from Scenario 3. The sender gets another call from above to send 1 data packet and there is a bit error in the ACK 0 packet coming back to the sender. There are no more bit errors afterwards. (1%)
(6) Describe the two scenarios that t10 will be triggered. (2%)

Sample Solution:

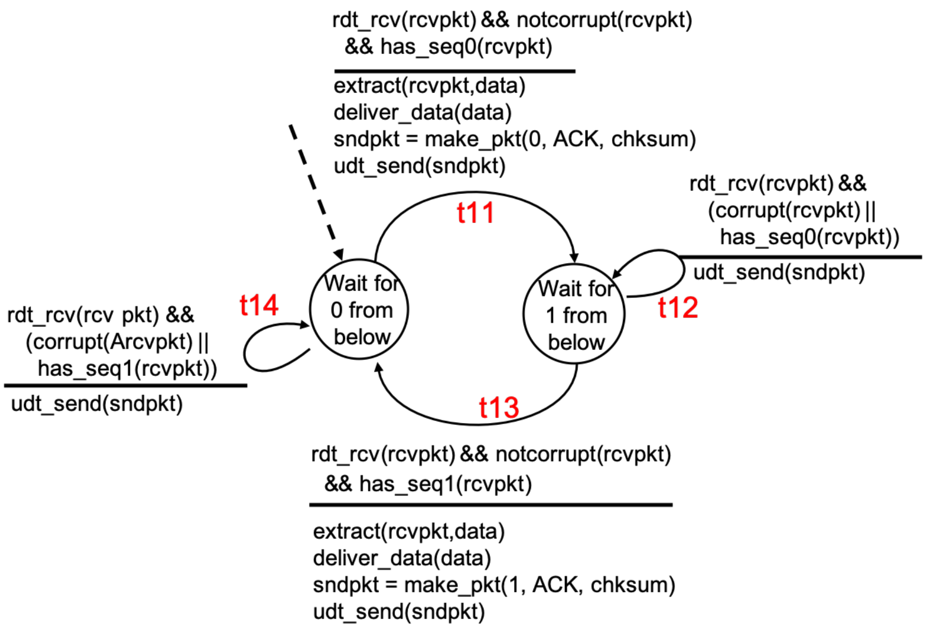
(1) t1, t7, t3
(2) t4, t8, t5, t9, t6
(3) t1, t10, t2, t7, t3
(4) t4, t9, t5, t10, t6
(5) t4, t8, t5, t9, t6
(6) S1: When the receiver is expecting data packet 0, but the incoming data packet (0 or 1) is corrupted.
S2: Data packet 0 has been received at the receiver earlier, and ACK 0 is sent back to the sender. In case the ACK 0 is corrupted at the sender. The sender retransmits the data packet 0. Next the receiver receives the data packet 0 when it’s expecting data packet 1. This triggers t10 and ACK 0 is transmitted again.

16. (Transport) Provided below are the FSMs of rdt 3.0 sender and receiver. Indicate the order of the transitions (in terms of t1, t2, …, t14) taking places until the sender and receiver stabilize for the following scenarios.

<u>rdt 3.0 sender:</u>


rdt_send(data)
sndpkt = make_pkt(0, data, checksum)
udt_send(sndpkt)
start_timer
t1
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
Λ
t2
timeout
udt_send(sndpkt)
start_timer
t3
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
stop_timer
t4
rdt_rcv(rcvpkt)
Λ
t5
rdt_send(data)
sndpkt = make_pkt(1, data, checksum)
udt_send(sndpkt)
start_timer
t6
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
Λ
t7
timeout
udt_send(sndpkt)
start_timer
t8
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
stop_timer
t9
rdt_rcv(rcvpkt)
Λ
t10
Wait for call 0from above
Wait for ACK0
Wait for call 1 from above
Wait for ACK1

<u>rdt 3.0 receiver:</u>


rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(0, ACK, chksum)
udt_send(sndpkt)
t11
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || has_seq0(rcvpkt))
udt_send(sndpkt)
t12
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(1, ACK, chksum)
udt_send(sndpkt)
t13
rdt_rcv(rcv pkt) && (corrupt(Arcvpkt) || has_seq1(rcvpkt))
udt_send(sndpkt)
t14
Wait for 0 from below
Wait for 1 from below

(1) Scenario 1: Both sender and receiver start from the initial state. The sender gets a call from above to send just 1 data packet. The data packet does not arrive at the receiver but all subsequent packet transmissions are fine, i.e., no bit error, no packet loss afterwards. (1%)
(2) Scenario 2: Continue from Scenario 1. The sender gets another call from above to send just 1 data packet. The ACK 1 packet does not arrive back at the sender while all other packet transmissions are fine. (1%)
(3) Scenario 3: Continue from Scenario 1. The sender gets another call from above to send just 1 data packet. The ACK 0 packet does not arrive back at the sender while all other packet transmissions are fine. (1%)
(4) Scenario 4: Continue from Scenario 1. The sender gets another call from above to send just 1 data packet. There is a bit error in the data packet but all subsequent packet transmissions are fine. (1%)
(5) Scenario 5: Continue from Scenario 1. The sender gets another call from above to send just 1 data packet. There is a bit error in the ACK 1 packet going back to the sender but all subsequent packet transmissions are fine. (1%)
(6) One can extend the t2 in rdt 3.0 sender such that when the ACK packet is corrupted or a duplicate (i.e., a NAK), the sender retransmits the data packet (instead of doing nothing). Discuss the benefits and drawbacks of the extension. (2%)

Sample Solution:

(1) t1, t3, t11, t4
(2) t6, t13, t8, t14, t9
(3) t6, t12, t8, t13, t9
(4) t6, t12, t7, t8, t13, t9
(5) t6, t13, t7, t8, t14, t9
(6) benefit – lower retransmission delay and therefore higher data throughput
drawback – multiple retransmission and therefore high network bandwidth consumption