[WR] Workshop 2

Topic	
Book chapter	
Covered in	
Date	
Main topic	Transport layer
Source	
URL	https://myuni.adelaide.edu.au/courses/85255/assignments/329601

Question 1 - Multiplexing/Demultiplexing

❓ The UDP header does not contain any information that is not present in the TCP header (verify this for yourself). That being the case, why do you, as a programmer, have to specify the header information (ports and IP addresses) whenever you send a UDP packet; but you don't when you send a TCP packet?

1. How is a UPD socket fully identified?
2. How is a TCP socket fully identified?
3. Assume a web application in Host A has a UDP socket bound to port number 8080. Now Host B and C each sends UDP segment to Host A with destination port number 8080. Will both segments be directed to the same socket at Host A? If so, how will the process at Host A know that these that these two segments originated from two different hosts?

When sending a UDP packet, we need to specify the header information (ports and IP addresses) but not for TCP because for TCP, the connection has to be established first before the packet is sent. Therefore, there is no need to specify the ports and IP again.

For UDP, which is a connectionless protocol, there is no connection and each packet is sent independently of each other. Therefore, each packet must contain information about ports and IP.

1. A UDP socket is fully identified by the combination of destination IP and destination port on the packet. Packets containing the same destination and destination port will be directed to the same socket.
2. A TCP socket is fully identified by 4 information: source IP, destination IP, source port number, destination port number
3. Both segments will be directed to the same port at host A because they have the same port number (and assuming the same IP which is host A's IP). The process at host A will use the source IP information inside each segment to tell which segment is sent from which host. If 2 segments have different source IPs then they come from 2 different hosts.

Question 2 - TCP and Protocol Design

❓ Protocol design decisions often have unexpected performance consequences. HTTP 1.0 is an example of a protocol design where lower layer protocol behaviour impacted directly on the performance of the higher layer protocol.

1. What is the difference between HTTP 1.1 and HTTP 1.0 in terms of transport layer connections?
2. What two transport layer issues do the changes to HTTP address?
3. How does the change improve HTTP performance?

1. Difference between HTTP 1.1 and HTTP 1.0 in terms of transport layer connection:
    a. HTTP 1.0 uses nonpersistent connection, meaning that for each packet of data, a connection need to be created. Meanwhile, for HTTP 1.1, there is persistent connection. The server leaves the connection open after sending the response,

thus saving the time needed to open a new connection for subsequent packets compared to HTTP 1.0

  b. HTTP 1.0 does not support pipelining (due to non persistent connection). HTTP 1.1 supports both with and without pipelining, and the default is with pipelining. In HTTP 1.1 with pipelining, client can send multiple requests without waiting for the response of the previous request to arrive first.

2. Improvement from HTTP 1.0 to 1.1 addresses these transport layer issues:
   a. First issue: HTTP is built upon TCP protocol for transport layer, which is a connection-oriented protocol via a 3-way handshake. Because a connection is required for TCP to work, the improvement from non-persistent to persistent connection allows multiple packets sharing the same initial and final connection requests, reducing the time needed to establish connection.
   b. Second issue (*I'm not exactly sure on this)*: using non-persistent connection, whenever the client sent a request, the both server and client need to create a new socket. While for persistent connection, server and client can just use the same socket for many packets. It can reduce the load on both sides, especially on server side since it potentially has to serve many clients.
3. How the change improve the HTTP performance:
   a. Reduce latency due to reduce the number of TCP connection requests for multiple packets.
   b. Reduce latency through pipelining by letting the client sending multiple requests at the same time, and server returns multiple requests at the same time.

Question 3 - Congestion Management/Control

❓ Consider congestion control in TCP Reno (most common algorithm).

How might application designers exploit the Internet's use of TCP to get higher data rates at the expense of other data flows that are using TCP?

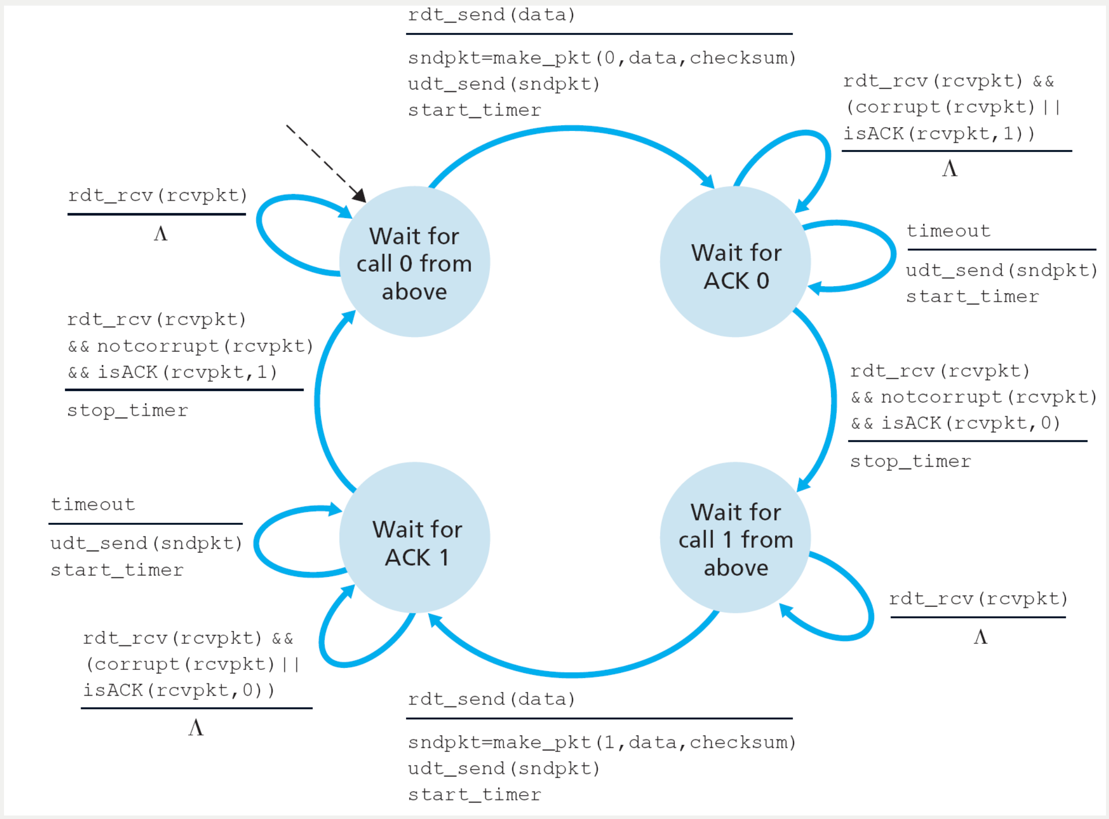
TCP Reno uses additive-increase, multiplicative-decrease (AIMD) form of congestion control. TCP linearly increases its congestion window size and hence the transmission rate until a triple duplicate ACK event occurs. It then decreases its congestion window size by a factor of two but then again begins increasing it linearly, probing to see if there is additional available bandwidth (Kurose & Ross 2020).

Application designers can exploit the Internet’s use of TCP to get higher data rates at the expense of other data flows by:

- setting a higher starting congestion window, so that during the slow start, the starting is not that “slow” and the rate increases significantly over a short period of time.
- increasing the congestion window increase step - also to make the slow start faster and thus occupies more bandwidth at a shorter period of time
- reducing the multiplicative decrease step - instead of reducing by 1/2, reduce the congestion window by 1/3 or 1/4 only, which will have the same effect as the above 2 methods
- decreasing retransmission timeout and make the packet retransmission faster, as the sender waits for a shorter time before deciding to re-send a packet that is not yet ACK-ed

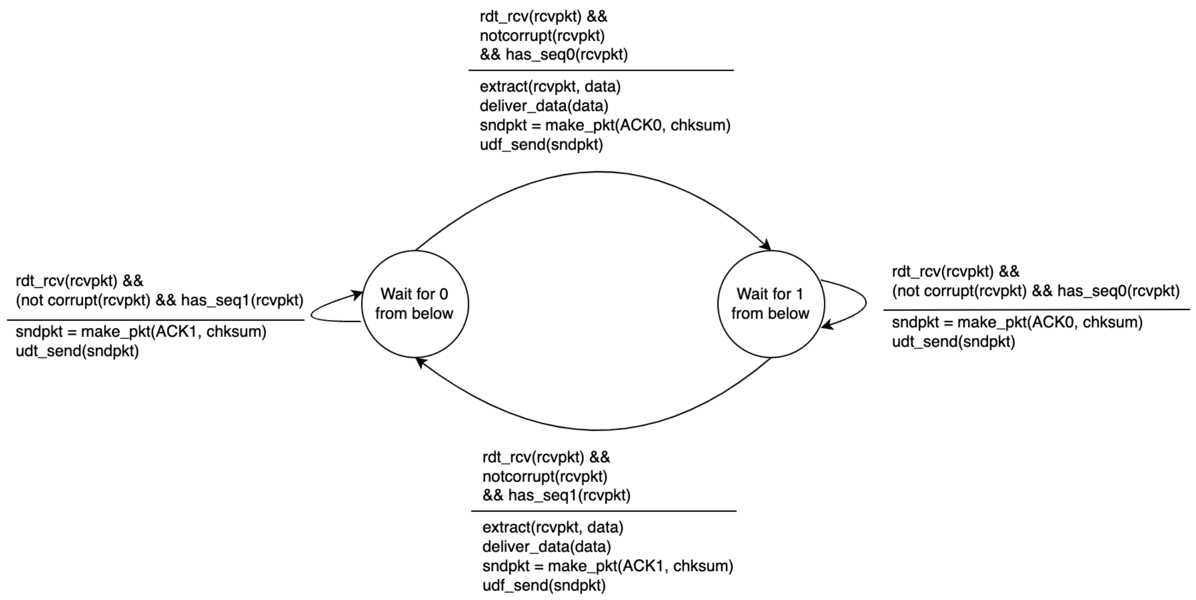
Question 4 - FSM and RDT

❓ Draw the FSM for the receiver side of protocol rdt3.0


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

Answer:

The receiver side should be the same as 2.2, because the only difference between 3.0 and 2.2 is the addition timer being implemented in the sender side. As the client side has no visibility of this timer, the receiver side should have no change.



Question 5 - Selective Repeat

❓ Consider the problem of implementing timers. Selective Repeat does not resend all packets on timeout, so it must timeout packets individually. However, in an implementation we are likely to only have access to one hardware timer.

How might you solve this problem?

- What overhead (extra tasks, that needs to be completed as a result, extra data structures that needs to be maintained) is your solution adding to the protocol?
- Can you think of a simpler solution?
    - Which packet, awaiting an ACK, in the sender window needs the timer?
    - What would happen if we re-start the timer every time a packet is sent?
    - What would happen if we re-start the timer when the oldest packet is ACKed?

1 solution I could think of is using the combination of go back N and selective repeat:

- Use the timer for the oldest packet inside the sender's window.
- When the ACK for all the packets inside the window is received, the window has already been moved forward, we restart the timer for the oldest packet of the new window - this is similar to go back N in the way that the time is only tagged to the oldest packet of the new window

If we re-start the timer every time a packet is sent, that means the timer is always used for the newest packet. Then the timer may never reach expiry because packets are sent continuously, unlike the Go Back N protocol. The timer should be used to track the time out of a specific packet only.

If we re-start the timer when the oldest packet is ACKed (similar to the solution I suggested above), then we need to build a timer that is long enough to accommodate for the delay of all the packets inside the window. This may cause a longer than expected wait until a packet is re-sent in case in lost packet, resulting in longer overall delay.

Question 6

❓ What would you like to discuss or review in the workshop?

Would be good to have some kind of overall review of what we learnt thus far in the course, as everything is connected but it’s hard to connect them all together.