CS2105 Live Class Lec4: UDP and Principle of RDT

$_{/}|/|U_Ch@NgRu!$

May 5, 2021

1 Transport services and protocols

Provide **logical communication** between app **processes** running on different hosts
Run in end systems:
send side: breaks app messages into **segments** passes to network layer
rcv side: reassembles **segments** into messages passes to app layer

2 Transport layer and network layer

2.1 transport layer

logical communication between **processes**
relies on network layer services

2.2 network layer

logical communication between **hosts**
unreliable, does not guarantee the pkt will be delivered

3 UDP(User Datagram Protocol)

3.1 UDP extra service

UDP adds very little service on top of network layer(IP):
1. connectionless multiplexing/de-multiplexing
2. Checksum

3.2 UDP chracteristic

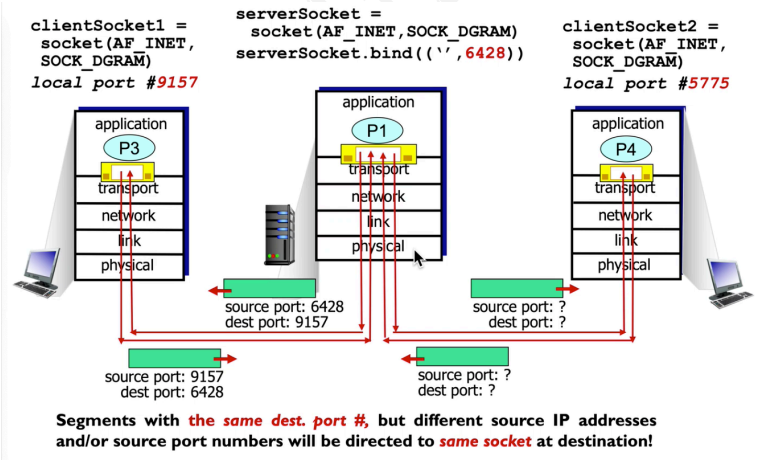
1. UDP transmission is unreliable: Often used by streaming multimedia apps(loss tolerant & rate sensitive)
2. UDP can still achieve reliable connection, but should be implemented in upper layer
3. IMPORTANT: it's wrong to say "make a UDP connection because UDP is not connection-oriented

3.3 multiplexing and demultiplexing

1. **Multiplexing:** (sometimes contracted to muxing) is a method by which multiple analog or digital signals are combined into one signal over a shared medium.
   In simple, multiple signal to single signal
2. **De-multiplexing** is the reverse of multiplexing, in which a multiplexed signal is decomposed in individual signals.
   In simple, single signal devides into multiple signal again

3.4 connectionless demultiplexing

Segments with the same distination port # but different source IP/port number will be directed to the same socket at destination, so what the UDP server get is multiplexed(i.e. the requests may come from different host) so it should de-multiplex the requests with the source IP and port #
(Isn't it similar to the welcome socket of TCP, but UDP really has the only socket for working, and the socket also response to client,this can also be seen as an advantage of UDP)

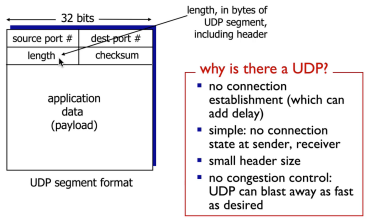


How UDP determines which process is responsible for certain requests when it receives multiple

requests

1. The client should specify the the port when creating the socket(door), in fact the OS will help to choose a port
2. The server should make its own port number to be known to all client(e.g.websites)
3. In the UDP request, the source port(help the server know who should response to) and the destination port(let the server host know which port to deliver to) are included

3.5 UDP: segmnent header



UDP header consists of 64 bits:

1. bit 1 $\sim$ 16: source port number (0 $\sim$ 65535)
2. bit 17 $\sim$ 32: destination port number (0 $\sim$ 65535)
3. bit 33 $\sim$ 48: length of UDP segment in bytes, including header
4. bit 49 $\sim$ 64: checksum

3.6 UDP checksum

Goal: detect "error" (e.g.flipped bits) in received segment

Computation:

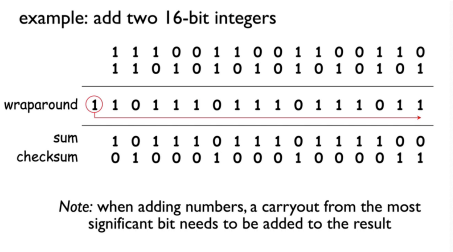
*1. Treat UDP segment as a sequence of 16-bit integer

*2. Apply **binary addition**(as shown in the image below) on every 16-bit integer

x	y	x ⊕ y	carry
0	0	0	-
0	1	1	-
1	0	1	-
1	1	0	1

*3. Carry from the most significant bit will be added to the result

*4. Compute 1's complement to get UDP checksum



IMPORTANT: The UDP can only detect error but cannot recover it

3.6.1 How is UDP checksum really calculated

you may find that the UDP checksum check the UDP header+data, while itself belongs to the UDP header, which seems wired

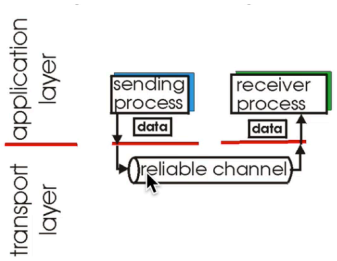
The basic idea is that the UDP checksum is a the complement of a 16-bit one's complement sum calculated over an **IP "pseudo-header" and the actual UDP data**. The **IP pseudo-header** is the source address, destination address, protocol (padded with a zero byte) and UDP length.

3.6.2 when the UDP checksum fail to check

If sender transmits the following two bytes: 01011100 and 01100101, and the two bits highlighted in red flip, then checksum remains unchanged and receiver will fail to detect this error.

4 Principle of reliable data transfer

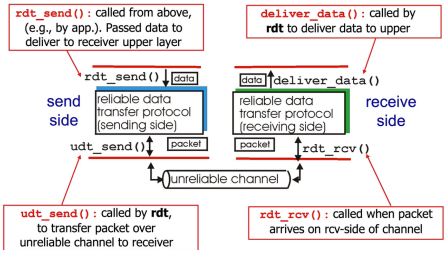
The principle is important in application, transportation, link layer and even lower



From the application's view, the transport layer provide reliable channel

The transportation layer applies lower layer service, which is unreliable channel(many wired thing will happen and hence numerous protocol strategies are needed to makes the upper layer get reliable data regardless of how shit the data in the layer

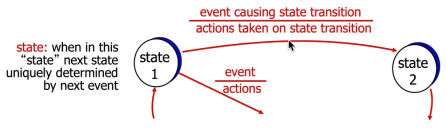
4.1 Reliable data transfer(rdt):interfaces



1. There is a method involved by the upper layer needs the layer to transfer data; the ”rdt_send()” means reliable data send
2. At the receiving side, the transport layer receiving side calls rdt_rcv() to receive data from the network layer(but the rdt_rcv() here does not mean that the data got by the transport layer is reliable because the lower layers are not reliable
3. The receiving side transportation layer will handle the received data according to protocol
4. The transportation receiver call deliver_data() to send the reliable data to higher layer get reliable data regardless of how shit the data in the layer

4.2 Finite state machine(FSM)

A diagram consisting with nodes(representing state) and arrows(transition/change of state) used to describe protocol



4.3 rdt1.0

4.3.1 Assumption

1. no bit error
2. no loss of packets

AIA: the lower layer is perfectly reliable

4.3.2 Seperate FSMs for sender and receiver

sender sends data into underlying channel
receiver reads data frome underlying channel
What the sender do is just break the data into block without checking and send it to the receiver
In the receiver side, just receive packet(assumed perfect) and reassemble it(if needed) and send it to higher layer ps: in a FSM, the state pointint to itself means reset initail state

4.4 rdt2.0

4.4.1 Assumption

underlying channel may flip bits in packet: checksum to detect bit errors

4.4.2 ACK(acknowledgement)

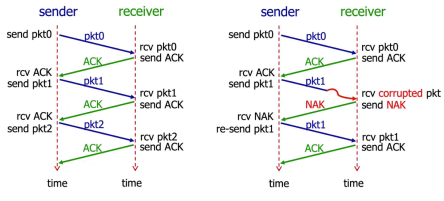
receiver explicitly tells sender that pkt reveived OK

4.4.3 NAK(negative acknowledgement)

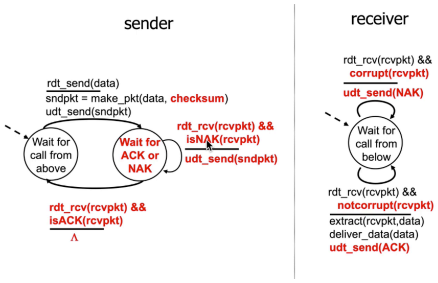
receiver explicitly tells sender that pkt had errors
sender retransmits pkt on receipt of NAK

4.4.4 new mechanisms in rdt2.0

1. error detection(sender checksum)
2. feedback: control msgs(ACK,NAK) from receiver to sender



4.4.5 The FSM



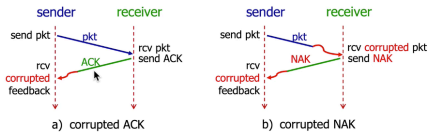
The red words are the difference comparing to the previous

1. there are two states for sender (wait for call and wait for ACK or NAK)
2. when the sender is waiting for the feedback, it receives the feedback and check: a: if it's the ACK, transfer to wait for call state

b: if it's the NAK, resend and wait for ACK or NAK

3. From the receiver side, it has only one single state, but it uses the checksum to check whether the received packet is corrupted or not

ps: What happens if ACK/NAK corrupted(lost)?



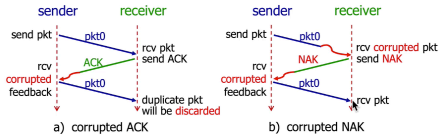
In the case, the sender did not receive any feedback or received garbled, the sender will re-transmit the packets again

But the above solution brings a new prob, in the case(a), the receiver cannot knwo whether it's the new packet or a repeated packet

4.5 rdt2.1

4.5.1 new mechanisms in rdt2.1

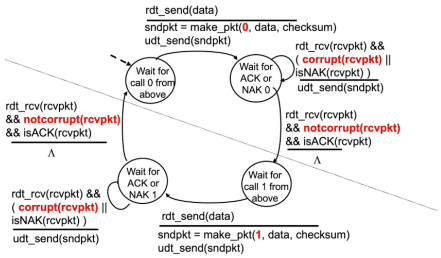
1. sender retransmits current packet if ACK/NAK is garbled
2. sender adds **sequence number** to each packet
3. receiver discards (doesn't deliver up) duplicate packet



In this case, it can be seen that in case (a) the two repeated packet can be differentiated The bit 'A' represents a bit indicate whether the pkt contains certain ACK, if it contain, it tells which sequence number associate with that ACK

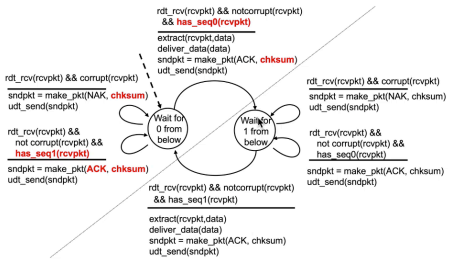
Here one bit is enough to differentiate the packet, but in reality it's not actually the case because the signal can be delayed the ACK of pkt 3 may arrive in advance to pkt1

4.5.2 FSM of rdt2.1 in class



The red 1 and 0 represent the sequence number

The FSM is divided into two symmetrical states, each half are identical to rdt2.0 The step flow(half):



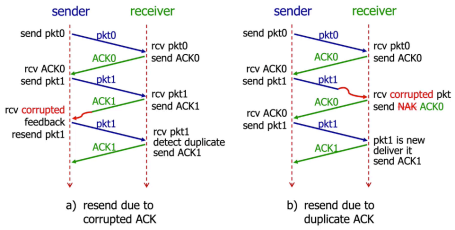
The receiver side of the FSM

IMPORTANT: the sender does not know whether the corrupted feedback is ACK or NAK, and it should not differentiate it. It only needs to send the pkt back with the right sequence number anyway.

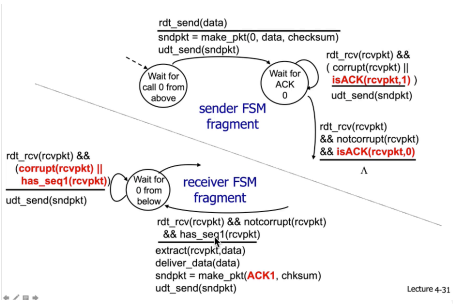
4.6 rdt2.2

4.6.1 new mechanisms in rdt2.2

1. Same functionality as rdt2.1, using **ACKs only**(receiver must explicitly include seq# of pkt being ACKed
2. duplicate ACK at sender results in the same action as NAK: retransmit current pkt



4.6.2 FSM of rdt2.2



4.6.3 Assumption in rdt2.2

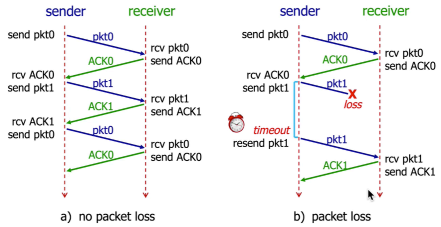
Underlying channel can also lose pkt(data, ACKs)
(ps: For stop-and-wait protocol, if the ACK will not be missed(i.e.The channel from R to S is reliable.); Then no sequence number is needed The sequence number is applied in case the ACK is corrupted and the duplicate pkt cannot be differentiated)
(checksum, seq. #,ACKs, transmission) in rdt2.2 are not enough.

4.6.4 New mechanism in rdt3.0

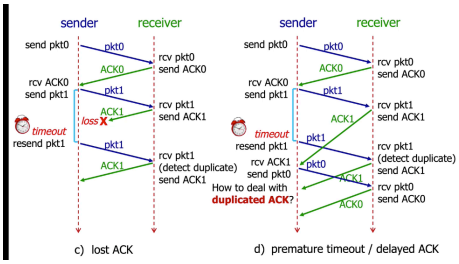
The sender waits "reasonable" amount of time for ACK:
1. requires countdown timer
2. retransmits if no ACK received in this time
3. if pkt (or ACK) just got delayed(not lost), retransmission will be duplicate, but seq 's already handles this; the problem is that the receiver also needs to specify the seq.# of pkt ACK(otherwise

the sender may be stuck

The below image shows the expected scenario



The pkt loss, after certain time the sender did not get any feed back and retransmit

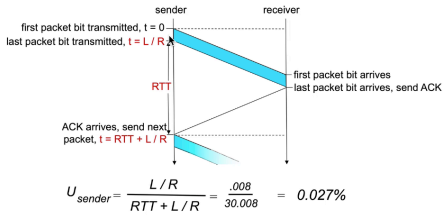


The ideal case for the 3.0 is in case(c), where the sender will retransmit the pkt if the ACK is lost

However, in the case d(called premature timeout), the ACK1 does not lost, but the sender is so lake of patience that it retransmit the pkt(maybe the time set is too short or the ACK is delayed), in this case, the ACK signal may be hashed, hence a large sequence number is needed, and the sender will ignore the ACK if it received duplicate ACK

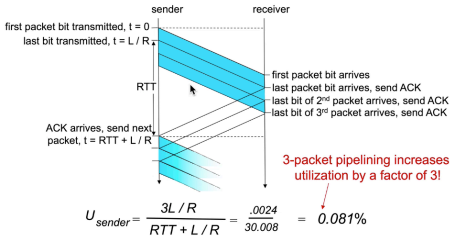
4.6.5 performance of rdt3.0

The stop-and-wait operation



The utilization is very inefficient because the sender wait for most of the time

The pipelining



Send multiple pkts rather than send the next until the ACK is received
In the untilization formula, why is the denominator $RRT + L/R$?

1. Because RTT>> $\frac{L}{R}$
2. since the first "run", when the first bit of the first pkt is sent out, say time1
3. To the second "run", when the first bit of the first pkt is sent out, say time2
4. time2-time1=RTT+L/R

In the statement above, there is a "run" concept, in fact, there is a window that decides how many pkt can be sent simultaneously without receiving the ACK, the windows here is more like a conceptual idea and later the Selective repeat and Go-back N will introduce one of the window
Here appears another problem related to this: what is the minimum window for the piplining so that the sender can be fully utilized(assume no corrup or pkt loss), the key idea is that, before the ACK of the first pkt of the first "run" reach the sender, the sender should not be idle, so

1. the time in first pkt's (denoted as pkt1 here) transmission view=time for trasmit pkt1 +time for propagate pkt1+ time for pkt1's ACK propagate
2. from the sender's view, the time for transmit all n in the window=time for trasmit single pkt*n

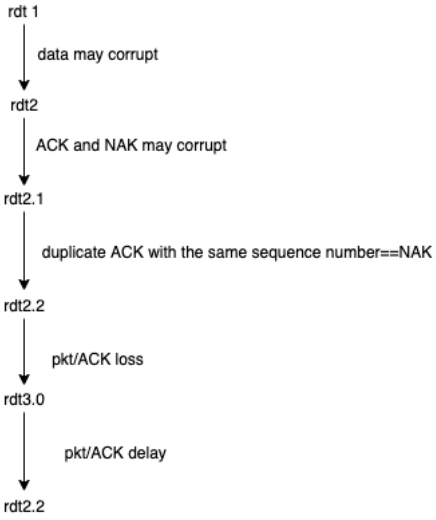
note that the (i) and (ii) are different view but the time take are the same
so time for transmit pkt1 +time for propagate pkt1+ time for pkt1's ACK propagate=time for trasmit single pkt*n
n=$\frac{\frac{L}{R}+RRT+RRT}{\frac{L}{R}}$

4.6.6 Factors that affect utilization

$U = \frac{NL/R}{RTT+L/R} = \frac{NL}{R*RTT+L} = \frac{N}{R*RTT/L+1}$

4.7 Conclusion

rdt version	network layer channel	new features introduced
1.0	no error	nothing
2.0	bit error in data	sender-side checksum, receiver-side ACK/NAK
2.1	bit error in data/feedback	receiver-side checksum, sender-side sequence number
2.2	same as 2.1	receiver-side sequence number
3.0	bit error/pkt loss in data/feedback	timeout/re-transmission



5 pipelined

5.1 Go-back-N in action

Introduced in details in Lec5

5.2 Selective repeat

Introduced in details in Lec5