Computer Networks Lecture

Scribe: alc3pf

October 6, 2021

Slides for the textbook can be found at: https://gaia.cs.umass.edu/kurose_ross/ppt.php

1 Python Demo

Code for the demo:

```
import socket
# Define socket host and port
SERVER_HOST = '0.0.0.0'
SERVER_PORT = 80
# Create socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.setsockopt(socket.SOL_SOCKET,
socket.SO_REUSEADDR, 1)
server_socket.bind((SERVER_HOST, SERVER_PORT))
server_socket.listen(1)
print('Listening on port %s ...' % SERVER_PORT)
while True:
    # Wait for client connections
    client_connection, client_address = server_socket.accept()
    # print client_address
    print("Connection from IP " + client_address[0] +
        " on port " + str(client_address[1]))
    # Get the client request
    request = client_connection.recv(1024).decode()
    print(request)
    # Send HTTP response
    response = 'HTTP/1.0 200 OK\n\nWelcome to Computer Networks'
    client_connection.sendall(response.encode())
    client_connection.close()
    # Close socket
    server_socket.close()
```

This demo simulates a web server that runs on port 80. When people submit requests to the server, the server will send back a response.

Code notes:

- import socket: socket is a python library that serves as a wrapper for the OS call
- AF_INET: specifies whether IPv4 or IPv6 socket should be used
- SOCK_STREAM: specifies UDP vs. TCP
- SOL_SOCKET: specifies that the options about to be supplied are socket-level options; generally, TCP layer options or presentation layer options can be supplied
- SO_REUSEADDR: tells the OS to give the program the socket to use; if the socket had been used previously, then there is some clean-up, and if the cleanup is still occurring, this option gives the socket to the program
- listen: controls how many people can connect simultaneously (e.g., listen with 10 means 10 people can connect simultaneously)
- Accept() does the three-way handshake
- Recv() parameter tells how much information (in bytes) you want to pull from the socket

The purpose of the network is to send a particular response whenever someone connects to it. In class, if students typed in their IP addresses into the browser, they would see the message ”Welcome to Computer Networks.” At the end of the day, we are writing an API, and we want to have a good design.

In-class squid game: be the 8th person to connect to the network; only the 8th person will survive and ”win” while everyone else ”dies.” How might we design the network to support this goal? Some considerations are to keep a counter to keep track of how many people have connected; we may also want to have a hash map to keep track of who has already connected, since if they already connected, and they are not the 8th person, then they lose and ”die.”

2 UDP

UDP (official documentation: RFC 768) is an Internet transport protocol that essentially pushes packets without checks.

- UDP uses a ”best effort” approach, so packets might get lost or be received out of order.
- UDP is typically used when losing information at times (e.g., a frame) is not a big deal. That is, UDP is tolerant to loss.

- UDP is **connectionless**, meaning that there is no handshake between the sender and the receiver. There is no connection establishment or connection state.
- There is no memory/state in UDP; all segments are independent from each other.

2.1 Advantages

- Simple and relatively quick, due to being connectionless (less overhead) - this also allows the size of the header to be smaller
- Still good with congestion - packets can be sent as fast as they can. This allows for more connections at a time.
- There is a little bit of help in terms of reliability (via the checksum).

2.2 Disadvantages

- Doesn't check when sending and receiving packets, so UDP is unreliable
  - If reliability is needed, then checks can be addressed at the application layer.

2.3 Uses

- Streaming videos, games, and other multimedia apps use UDP because it is fault tolerant.
  - Question: how does streaming work if things arrive out of order and can get lost?
  - Answer: You don't have to construct things perfectly in order; if order needs to be maintained, the application layer can be designed to be responsible for reordering things. Skype uses delta compression.
- DNS (associates name with IP) uses UDP because it is fast.
- SNMP (network management) uses UDP because it is good with congestion.
- HTTP/3 uses UDP; this allows more fine-tuned control of package exchange at the application layer.

2.4 Walking through UDP

- Sender actions: The application layer passes down message to the transport layer. UDP then attaches header fields, creating a UDP segment. The segment is then given to IP on the network layer. At this point, the segment becomes part of an IP datagram and is sent to another machine over the network.

- Receiver actions: An IP datagram is received at the network layer. The checksum is checked to test for data corruption. The message from the sender's application layer is retrieved and passed up through a socket to the receiver's application layer.

2.5 UDP Segment

The UDP segment has 4 header fields (2 bytes each) and the application message. The header fields are:

1. Source port number
2. Destination port number
3. Length: number of bytes in the segment. Note that the header length is not always included in the length field (sometimes will and sometimes will not be) - this is one of the things that you'll have to read the RFC really carefully for
4. Checksum: transport-layer check for packet corruption (bits were flipped).
    - Sender side: UDP segment is divided into 16-bit chunks, which are summed together using one's complement. The checksum is computer by taking the inverse of this sum. The checksum is then put into the segment header. Note that the chunks that are added together include the UDP header and some parts of the IP header.
    - Receiver side: the receiver does the same calculation on the segment. It checks this sum by adding it to the checksum value; the sum should be all 1s if the calculated sum is correct. It concludes no errors if the calculated sum is correct; it concludes that there was an error if the calculated sum is not correct.
    - When summing, overflow is addressed with wraparound. That is, the overflow is added back to the main sum, beginning with the least significant bit.
    - Checksum is a very weak check (almost as weak as parity). You can flip relatively few bits and pass the check. One might wonder why we have checksum if there is already CRC; this is because UDP is older, so checksum has not yet been officially removed.
    - **Example:** We can add the chunks (16-bit integers) 1110011001100110 and 1101010101010101. The sum is 11011101110111011 without wraparound; with wraparound, the sum is then 1011101110111100. The checksum is the inverse of this: 0100010001000011.

3 Reliable Data Transfer

Ideally, we want an abstraction of a reliable channel. We want to be able to send data down this channel and have it appear as if it is coming out the other end with no losses. We'll have sender-side and receiver-side reliable data transfer protocol (RDT); if something is lost, we'll have some mechanism of recovering it.

How complex the reliable data transfer protocol are dependent on the design of the channel. As a whole, the sender and receiver don't know each other's state; the sender never knows the state of the receiver unless the receiver sends information over the unreliable channel.

In defining our reliable data transfer protocol (RDT), we'll keep in mind that we are assuming unidirectional transfer of data (one machine is the dedicated receiver and the other is a dedicated sender). Also, we are defining the sender and receiver as finite state machines. That is, state will define what situation we are in, and we will only change state with some event.

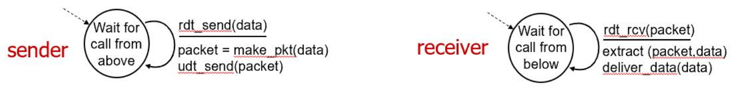
Interfaces of the RDT include:

- rdt_send(): this is a method used by the application level; it pushes the data down onto the transport layer
- : udt_send(): this method is used by RDT; it sends the packet down to the network layer and through the unreliable channel to the destination machine
- : rdt_rcv(): this method is used when the packet is received on the destination machine; the packet is pushed from the network layer to the transport layer
- deliver_data(): this method is used by RDT; it pushes information to the application layer from the transport layer.

rdt_send() and udt_send are used by the sender while rdt_rcv and deliver_rcv are used by the receiver.

3.1 RDT 1.0

The channel between the sender and the receiver is completely reliable. No losses or errors are incurred. The sender and receiver are completely separate finite state machines. The finite state machine for the sender will wait for some call from above (i.e., it will ask the application layer, "Do you have some data for me?"). If there is data, then the sender will make a packet out of the data and send it. the receiver will then extract data from the packet and deliver it to an application on the receiving side. This is the simplest case.



*Note that knowing how to read these diagrams is very important.

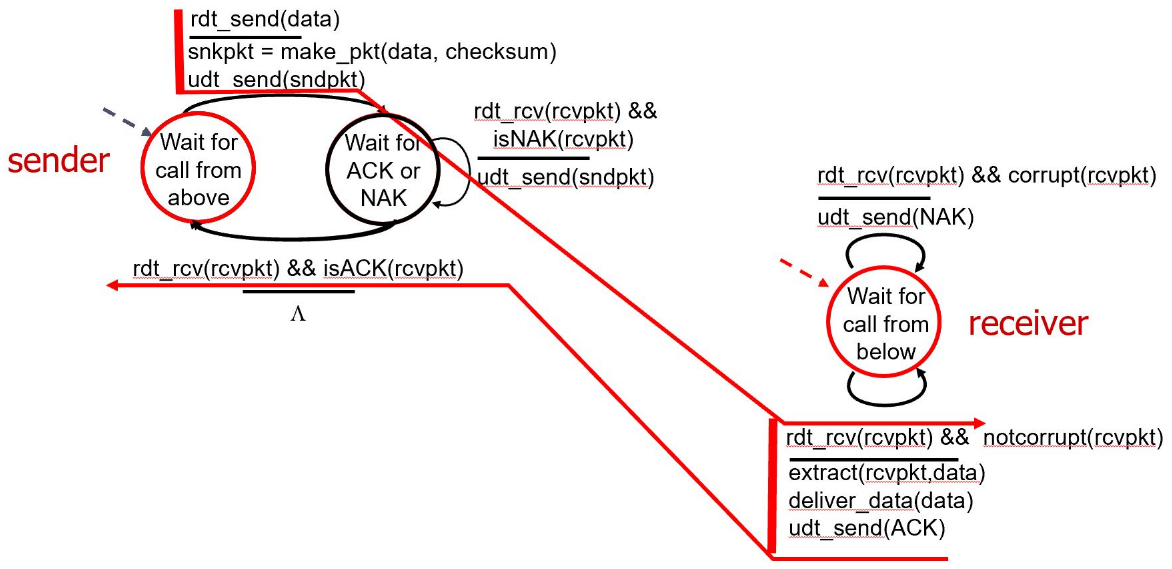
3.2 RDT 2.0

When sending packets through the channel, bit-wise errors may occur. We are still assuming at this point that the channel does not lose any packets. We know that we can check for errors by using checksum, but how do we fix these errors? We will introduce **acknowledgements (ACKs)** and **negative acknowledgements (NAKs)**.

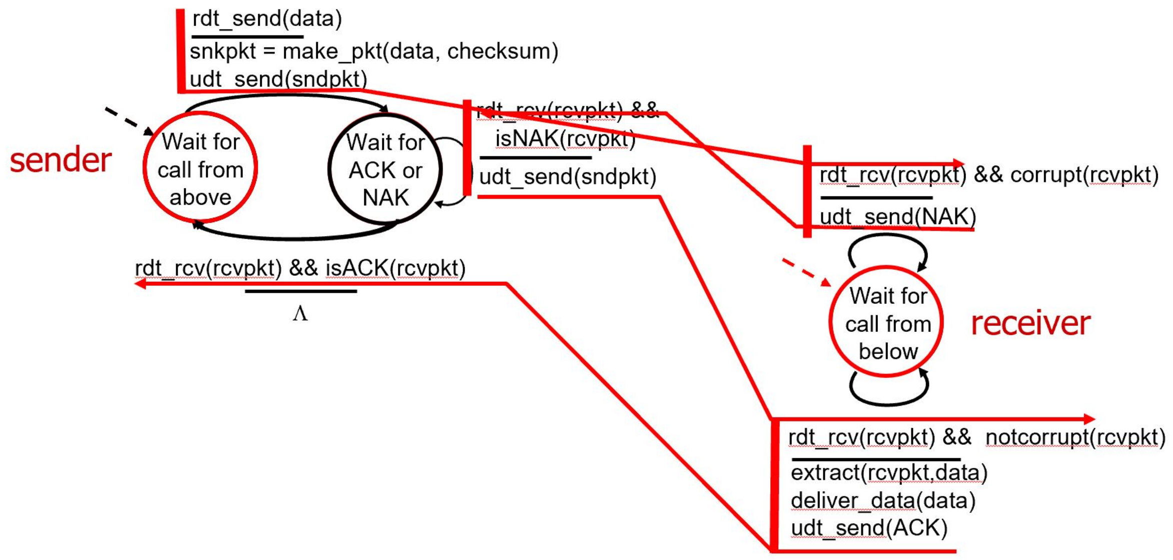
ACKs are a means for the receiver to let the sender know that the packet was received successfully without error. NAKs are a means for the receiver to let the sender know that there were errors in the packet received.

The organization of RDT 2.0 is such that the sender follows a **stop and wait** behavior. That is, the sender will transfer a packet over the channel and await a response from the receiver. If the sender receives a NAK, it will know to try to resend (retransmit) the packet.

The following diagram shows RDT 2.0 flow when the packet arrives without error:



The following diagram shows RDT 2.0 flow when the packets arrives with error:



There is still a wall between the sender and the receiver; the sender doesn't know what state the receiver is in unless we explicitly send some information, so this is one of the reasons we still need a protocol.
Disadvantages of RDT 2.0 include:

- How long do you wait for the receiver to respond? We have to set a timer, and this adds some complexity. Furthermore, the time can add up.
- There's a lot of overhead, as we are sending at least double the amount of packets for every packet with data. Moreover, ACK and NAK packets only need 1 bit of information to say whether or not the packet has been received without error. It seems a bit wasteful to devote entire packets just for one bit.

By far the biggest disadvantage of RDT 2.0 is **if the ACK or NAK packages are corrupted**. If the sender decides to retransmit without knowing whether the previous packet was received successfully, then there might be duplicate packets received in the end.

- One way to handle duplicates, however, is to have each packet have a sequence number (this is added by the sender). The receiver would then know which packets are duplicates; packets that are identified as duplicates are simply not pushed up to the application layer. When attaching sequence number, we know that we only need one bit of information, since we just need to know if the ACK or NAK was received correctly in order to know if the packet received is new or a duplicate.

3.3 RDT 2.1

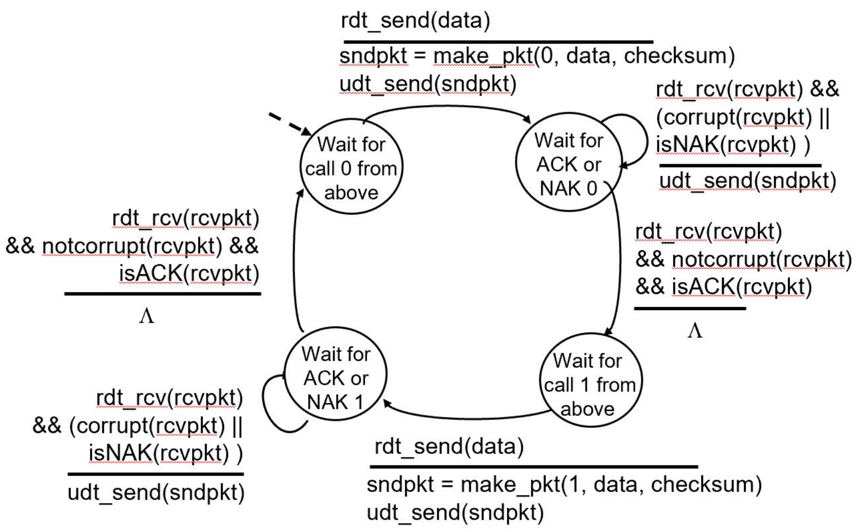
Here, we will take our infrastructure from RDT 2.0 and add additional complexity. At this point, each packet has an associated sequence number. With each new packet, the sequence number increments with modulo 2. So, packet 1 will have sequence number 1, packet 2 will have sequence number 0, packet 3 will have sequence number 1, and so on.

The receiver will check the sequence number of the latest packet received. If the sequence number is the same number as the packet received just prior, then the latest packet is a duplicate of the packet just prior. On the other hand, the sequence number of the latest packet is different from the last, then the latest packet is a new packet. This allows our sequence number to just be 0 or 1, since we just want to know if the current packet is the same as the last.

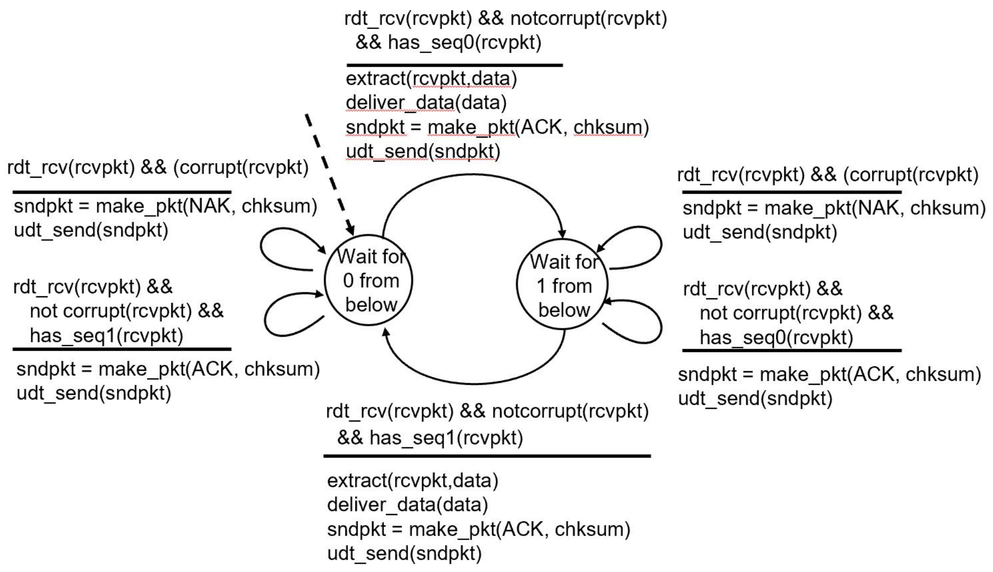
Because of the added complexity of the sequence number, we will now have double the states from before (for both sender and receiver). Additionally, the state for the sender and the state for the receiver must keep track of sequence number; the sender must know what sequence number to send for each packet, and the receiver must know what sequence number to expect.

One note to make is that the although the sender gets feedback on whether its packet was received, the receiver does not get feedback on whether its sent ACK or NAK packets arrived and were received successfully by the sender.

The following diagram shows the sender in RDT 2.1:


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

The following diagram shows the receiver in RDT 2.1:



rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq0(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq1(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
Wait for 0 from below
Wait for 1 from below
rdt_rcv(rcvpkt) && (corrupt(rcvpkt)
sndpkt = make_pkt(NAK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && not corrupt(rcvpkt) && has_seq0(rcvpkt)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && has_seq1(rcvpkt)
extract(rcvpkt,data)
deliver_data(data)
sndpkt = make_pkt(ACK, chksum)
udt_send(sndpkt)

If we establish a simulated RDT on Wireshark, we can see some of the concepts above concretely. That is, we can see each packet's sequence number and the contents of each packet. One note to point out is the content of ACK packages - it is apparent that only one bit is used and thus just how much space is wasted in that sense. Another note to point out is that the SYN, ACK, and SYNACK packages mentioned are the three-way handshake that will be discussed.

Next class, we will continue building up our RDT. f