Foundations of Computer Networks

Project 2(CSCI 651.01)

Aniket Giriyalkar

Ashish Paralkar

Table of Contents

Introduction

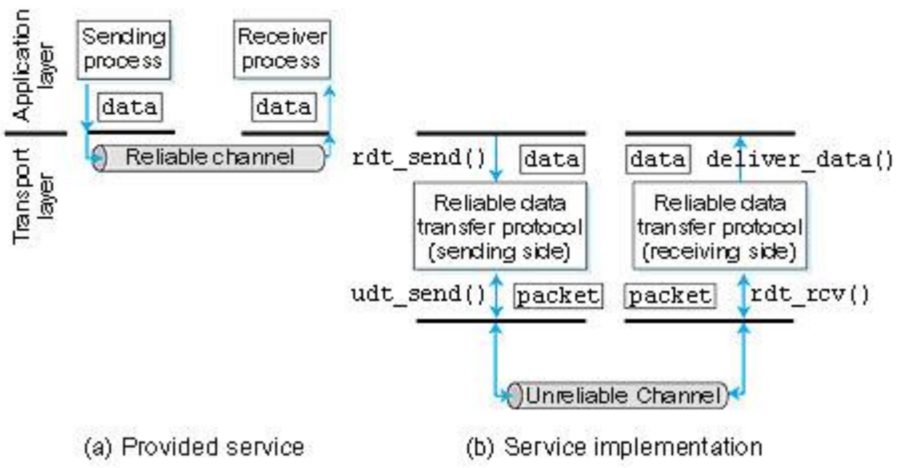


Figure 1: Reliable data transfer: Service model and service implementation

Characteristics of unreliable channel will determine the complexity of reliable data transfer protocol (rdt). rdt_send( ) is called from above, (e.g., by app.) passes data to deliver to receiver upper layer. deliver_data( ) is called by rdt to deliver data to upper. udt_send( ) is called by rdt, to transfer packet over unreliable channel to receiver. rdt_receive( ) is called when packet arrives on rcv-side of channel.
Rdt 1.0 was perfectly reliable, there were no bit errors and no loss of packets.
Rdt 2.0 added the features of error detection (using checksum) and feedback(control messages ACK,NAK from receiver to sender) to Rdt 1.0.

Rdt 2.1 corrects the flaws of Rdt 2.0, i.e. it handles corrupted ACK/NAK(sender retransmits current packet if ACK/NAK corrupted) and it also handles duplicate packets(receiver discards duplicate packets) . This follows a stop and wait approach i.e. the sender sends one packet, then waits for receiver response.
Rdt 2.2 has the same functionality as rdt2.1, but this uses ACKs only. Instead of NAK, receiver sends ACK for last packet received OK, duplicate ACK results in same action as NAK i.e. retransmit current packet.

In the implementation of rdt 3.0 the new assumption is that the underlying channel can also lose packets (data, ACKs). Checksum, Sequence number, ACK's will be of help but these won't be enough. The sender waits a "reasonable" amount of time for ACK. It retransmits if no ACK received in this time. If Packet (or ACK) is just delayed (i.e. not lost) then retransmission will be duplicate, but sequence numbers already handle this. Receiver must specify the packet number being ACKed. This requires the use of countdown timer

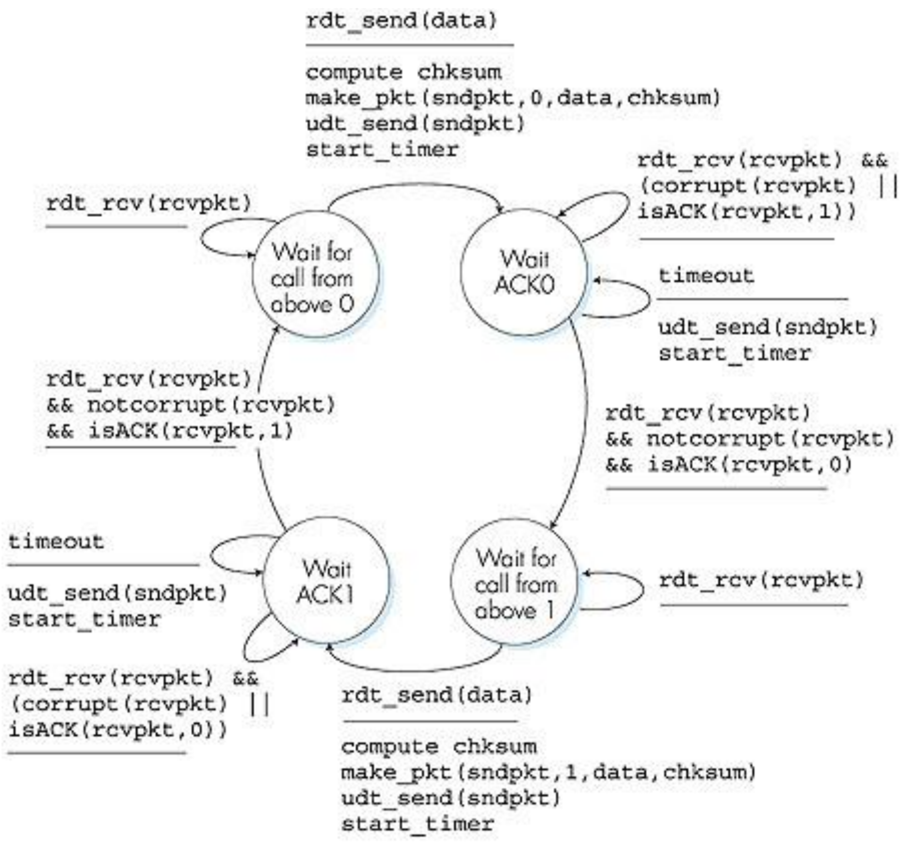

rdt_send(data)
compute chksum
make_pkt(sndpkt,0,data,chksum)
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,1))
rdt_rcv(rcvpkt)
Wait for call from above 0
Wait ACK0
timeout
udt_send(sndpkt)
start_timer
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,1)
rdt_rcv(rcvpkt) && notcorrupt(rcvpkt) && isACK(rcvpkt,0)
timeout
udt_send(sndpkt)
start_timer
Wait ACK1
Wait for call from above 1
rdt_rcv(rcvpkt)
rdt_rcv(rcvpkt) && (corrupt(rcvpkt) || isACK(rcvpkt,0))
rdt_send(data)
compute chksum
make_pkt(sndpkt,1,data,chksum)
udt_send(sndpkt)
start_timer

Figure 2: rdt3.0 sender FSM

We see in the figure below, as time moves forward from the top of the diagram toward the bottom of the diagram; note that a receive time for a packet is necessarily later than the send time for a packet as a result of transmission and propagation delays. Because packet sequence numbers alternate between 0 and 1, protocol rdt3.0 is sometimes known as the alternating-bit protocol. Checksums, sequence numbers, timers, and positive and negative acknowledgment packets each play a crucial and necessary role in the operation of this protocol.

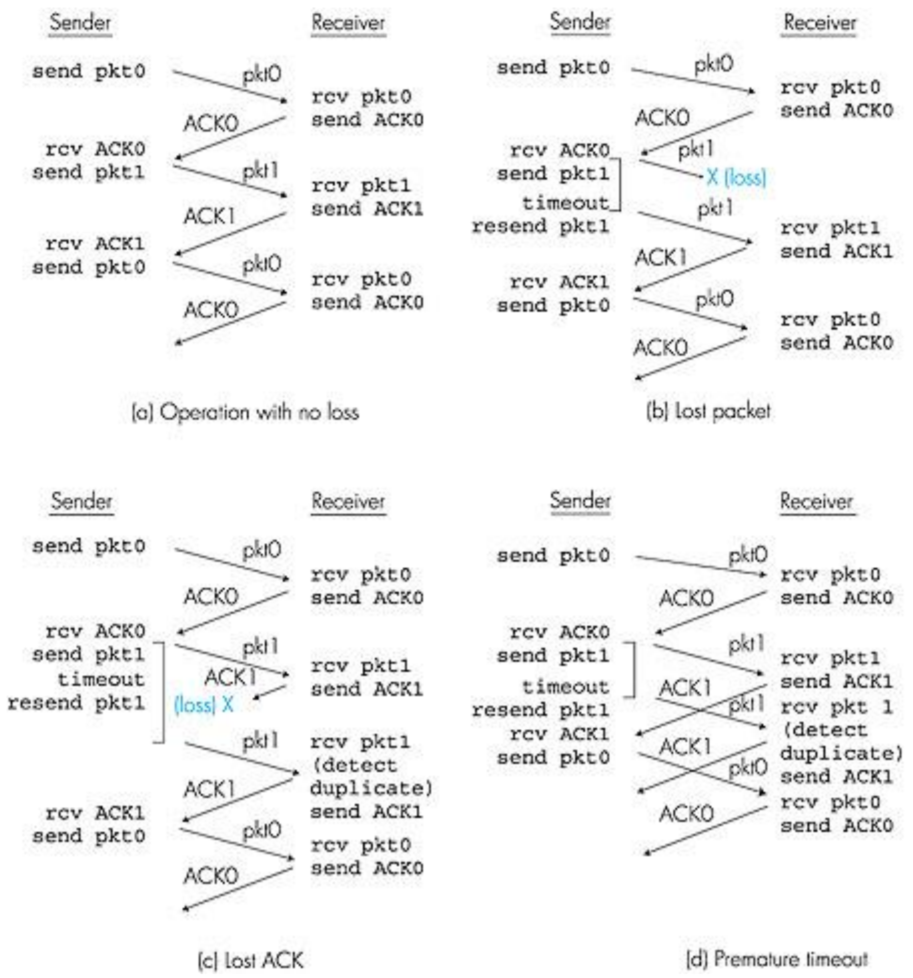


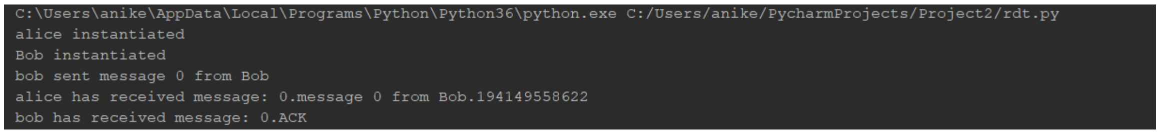
Figure 3: Operation of rdt3.0, the alternating-bit protocol

Procedures

We have implemented this protocol in our program using multithreading approach. We have used the package pydispatch for communication between the two threads.
In our program, we send a packet which consists of the sequence number, message and checksum. The checksum value is generated with the help of inbuilt hash function and the hash value generated is the value of checksum. At the receiver, in order to simulate this protocol, we purposely include different test cases(normal execution, corrupt message, corrupt ACK, incorrect sequence numbers, timeout) and each time a different test case is considered by choosing one from the all possible cases using random function. The cases are tested and appropriate results are obtained for the test cases which they correspond to.

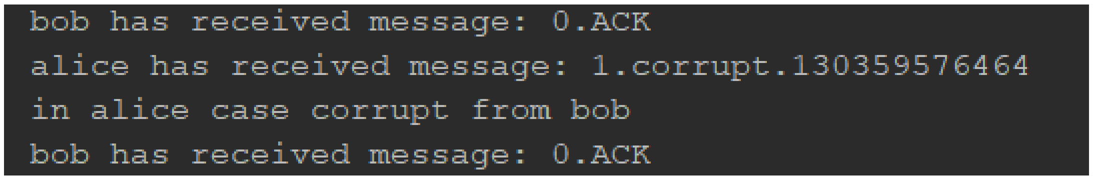
Results and Discussion

Case 1: Operation with no loss(Normal Execution)
In this case, the ACK is received and the package of next sequence is sent.



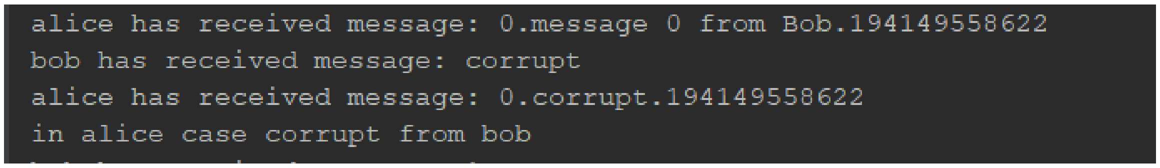
```
C:\Users\anike\AppData\Local\Programs\Python\Python36\python.exe C:/Users/anike/PycharmProjects/Project2/rdt.py
alice instantiated
Bob instantiated
bob sent message 0 from Bob
alice has received message: 0.message 0 from Bob.194149558622
bob has received message: 0.ACK
```

Case2: Corrupted message
In this case, the message received was corrupted and the packet is resent.



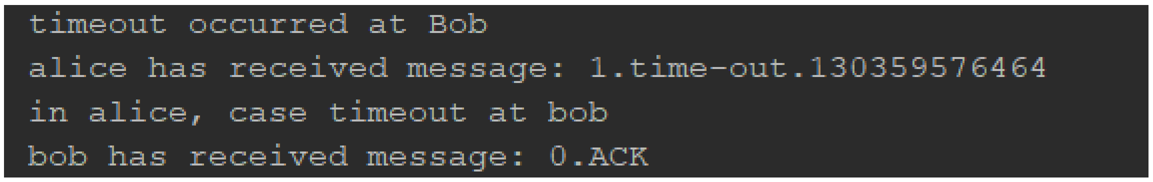
```
bob has received message: 0.ACK
alice has received message: 1.corrupt.130359576464
in alice case corrupt from bob
bob has received message: 0.ACK
```

Case 3: Corrupted Acknowledgement
In this case, the acknowledgement received is duplicate or corrupted, the packet is resent.



```
alice has received message: 0.message 0 from Bob.194149558622
bob has received message: corrupt
alice has received message: 0.corrupt.194149558622
in alice case corrupt from bob
```

Case 4: Timeout
The last ACKed packet is resent in case of a timeout.



```
timeout occurred at Bob
alice has received message: 1.time-out.130359576464
in alice, case timeout at bob
bob has received message: 0.ACK
```

Thus, we have successfully implemented a bit alternating rdt 3.0 protocol. It works successfully for the states 0 and 1, and thus can be successfully implemented on any number of packets.

Conclusion and Future work

Protocol rdt3.0 is a functionally correct protocol, but it is unlikely that it would be preferred for its performance, particularly in today's high-speed networks.

At the heart of rdt3.0's performance problem is the fact that it is a stop-and-wait protocol . So, rather than operating on a stop and wait manner, the sender is allowed to send multiple packets without waiting for acknowledgements. This technique is known as pipelining (as many in-transit sender-to-receiver packets can be visualized as filling a pipelining).

In future, different approaches of Pipelining (Go-Back-N or Selective repeat) can be used for a better performance than Rdt 3.0.