**Principles of Reliable Data Transfer**

Reliable Data Transfer = RDT

**Introduction**

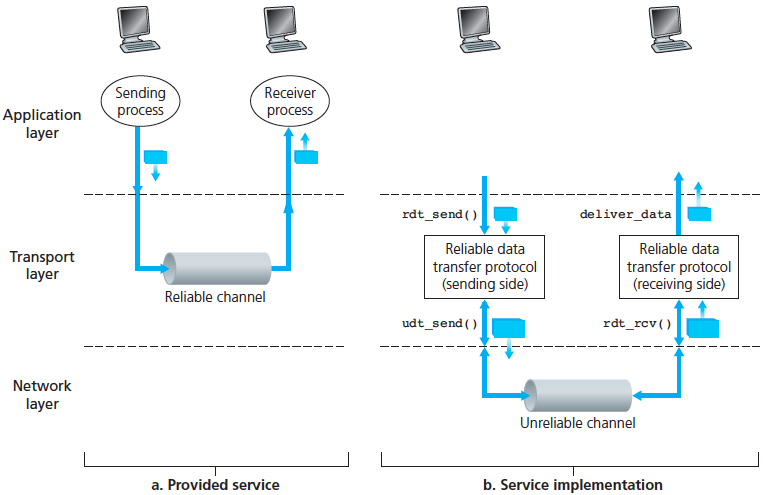
With RDT, transferred data bits are: 'not corrupted' | 'not lost' | 'delivered in order'

* TCP offers this service model to internet applications that use it
* However, layers below RDT Protocol may be unreliable.

**'rdt\_send()'** = Pass data from sender-side app to be delivered to the receiver-side app

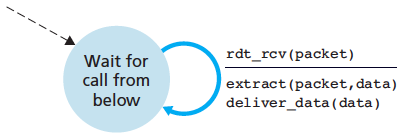
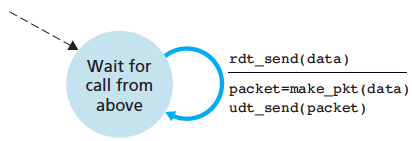
**'rdt\_rcv()'** = Called when packet arrives to receiver-side channel

**'deliver\_data()'** = Delivers the data to receiver-side app



**Building a Reliable Data Transfer Protocol**

**RDT 1.0 – Transfer over a perfectly reliable channel (not a realistic model)**



All packet flow is from sender 🡪 receiver.

No need for receiver-side to provide feedback to sender.

Also assumption that receiver is able to receive data as fast as the sender sending data, thus no need for flow/cong control.

**RDT 2.0 – Transfer over a channel with bit errors (more realistic model)**

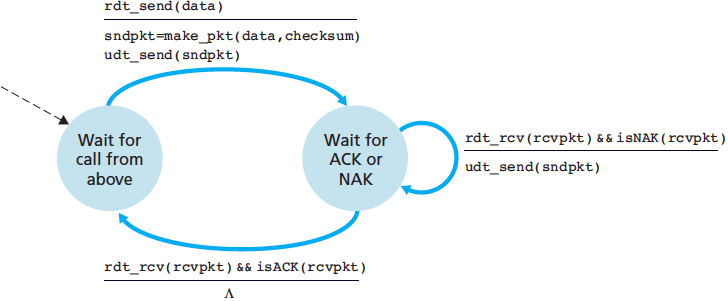
In this model, we assume packets can be corrupted. Bit errors occur in physical components of a network as a packet is transmitted, propagated or buffered.

This model is based on retransmissions of data, known as: ARQ: Automatic Repeat Request Protocols.

Three additional protocol features are required in ARQ protocols to handle bit errors:

1. Error Detection: E.g. UDP using internet checksum (gathered in checksum field of a data packet)
2. Receiver Feedback: Positive ACK and Negative NACK replies from receiver 🡪 sender.  
   Only needs to be 1 bit-long. 0 = NAK | 1 = ACK
3. Retransmission: Packet received in error will be retransmitted by the sender

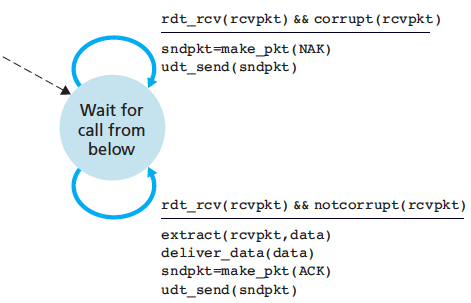
RDT 2.0 is



* Sender waits for data to be passed down
* When **rdt\_send() occurs**, sender creates a packet **sndpkt + checksum**
* Send packet via. **Udp\_send()**
* Wait for ACK packet: **rdt\_rcv(rcvpkt) && isACK**
* If true, go back to waiting for data from app layer.
* Else retransmit last packet + wait for ACK again

NOTE: If sender is waiting for ACK/NACK it can’t receive more data from upper layer. ( rdt\_send() can’t occur)

**This is a STOP-AND-WAIT protocol**.



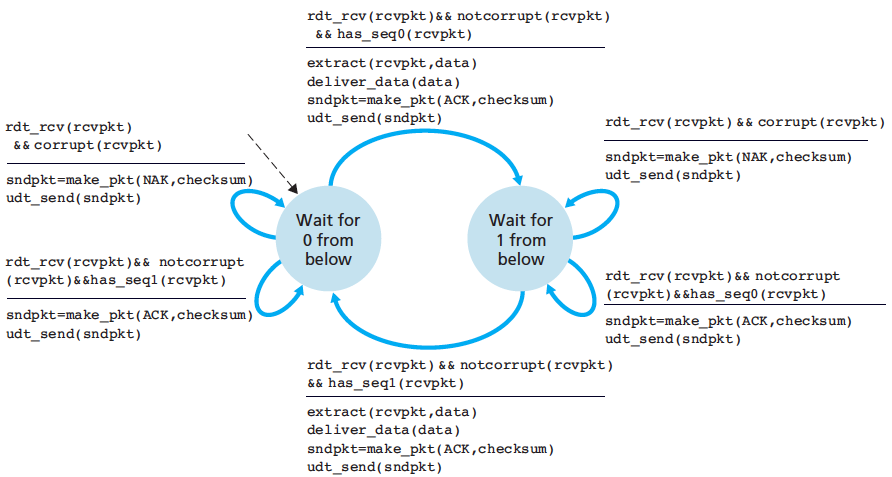
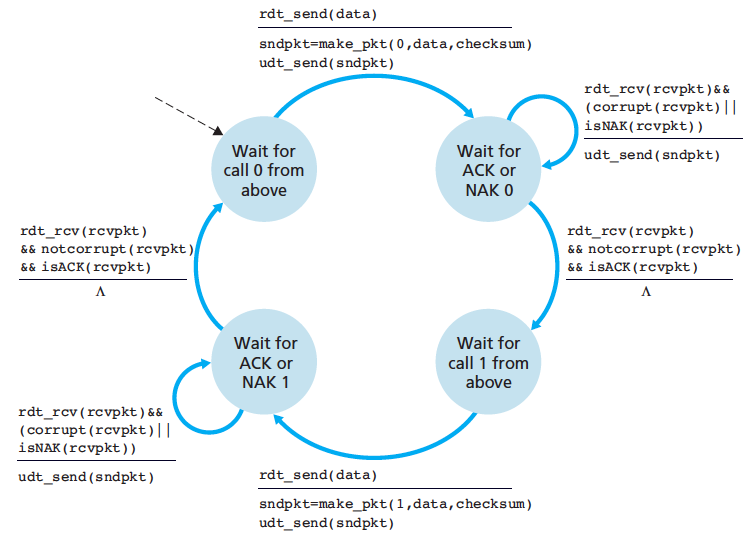
* On packet arrival, receiver replies with ACK/NACK depending on if packet is corrupted.
* Corrupted = **rdt\_rcv(rcvpkt) && corrupt()**  
  -> send NAK packet via. Udt\_send()
* Uncorrupted = **rdt\_rcv(rcvpkt) && notcorr()**  
  -> extract data  
  -> deliver to upper layer  
  -> send ACK packet via. Udt\_send()

Fatal flaw with RDT 2.0 = ACK/NACK packets themselves could be corrupted.

* Solution: add a new field into the data packet and have the sender number its data packets by putting a **sequence number**.
* Receiver only needs to check this sequence number to determine if the received packet is a retransmission.

**RDT 2.1 – Protocol state now reflects whether a packet being sent or expected should have sequence number 0 or 1**

0-numbered packet being sent or expected = 1-numbered packet is being sent or expected. (if correctly sending/recv)



* When out-of-order packet is received, receiver sends **pos acknowledgement** for the packet it has received.
* When corrupted packet is received, receiver sends **neg acknowledgement**.
* A sender that receives two ACKS for the same packet = receiver did not correctly receive the packet

**RDT 3.0 – Transfer over a LOSSY CHANNEL with BIT ERRORS**

Suppose in addition to corrupt bits, the underlying channel can lose packets. This means two additional concerns:

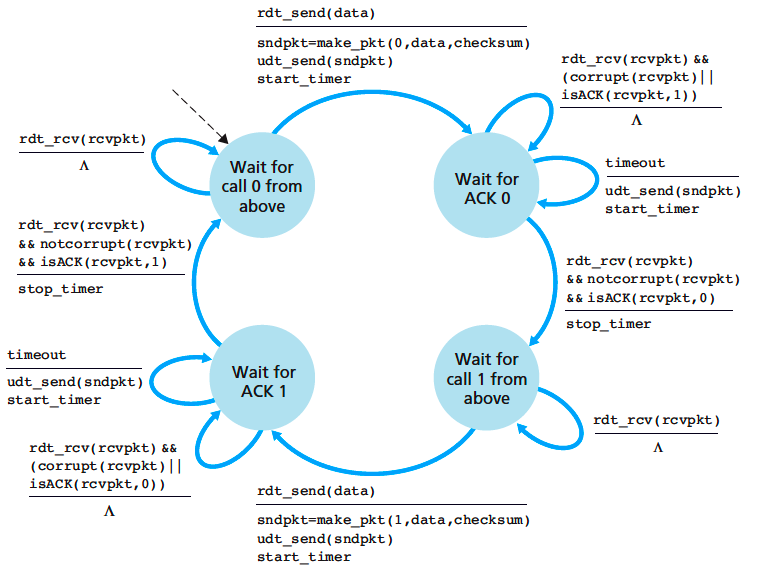
* (1) How to detect packet loss (2) What to do when packet loss occurs.
* Checksums, seq numbers, ACK packets and retransmission solve error 2.
* New protocol methods are needed to detect packet loss.
* We will assume both DETECTON and RECOVERING from lost packets will addressed by sender side.

Suppose the sender transmits a data packet:

* Either that packet or receiver’s ACK of the packet gets lost.
* If the sender is willing to wait long enough so that it is certain the packet is lost, it can simply re-transmit the packet.
* ^However, **large delays =** packet is not actually lost, but sender may retransmit the packet anyway = **duplicate packets**

From the sender’s viewpoint, retransmission is an all-in-one solution: doesn’t matter if packet/ACK was lost or overdelayed

* In all cases, the sender will just need to re-transmit the data.
* Implementing time-based re-transmission requires a **countdown timer** that can interrupt the sender after a given amount of time has expired.
* The sender will need to be able to:  
  **(1) start time after each packet – either 1st time or retransmission packet is sent  
  (2) respond to a timer interrupt – taking appropriate actions  
  (3) stop the timer**



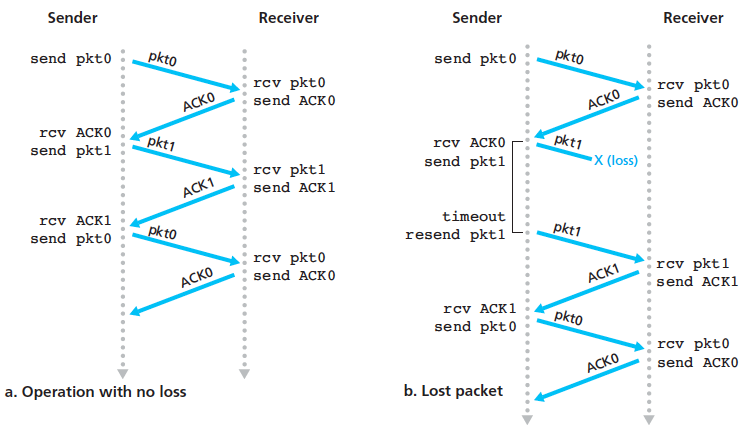
Now we have all the features required for a working, reliable data transfer protocol:

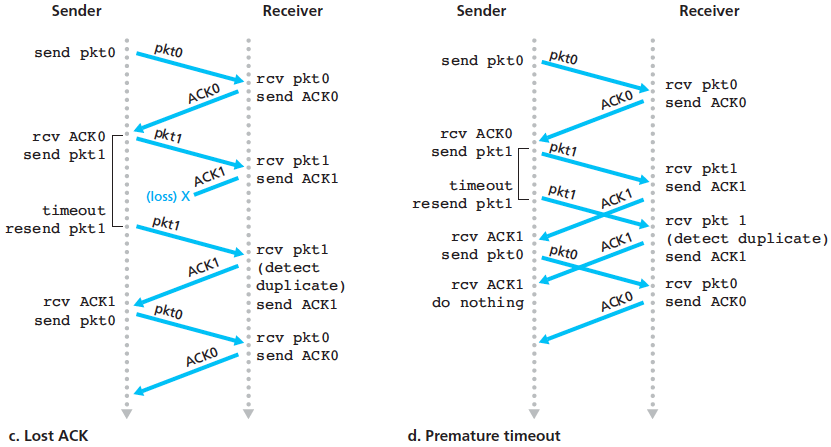
* **Checksums, sequence numbers, timers, positive and negative acknowledgement packets**.

**Pipelined Reliable Data Transfer Protocols**

The RDT 3.0 is a functionally correct protocol, however it is unlikely that anyone would be happy with its performance, due to the fact that RDT 3.0 is a **Stop-And-Wait** protocol.

Consider the performance of Stop-And-Wait behaviour: NO LOSS vs. LOST PACKET vs. LOST ACK vs. PREMATURE TIMEOUT





**SKIP TO 3.5 CONNECTION-ORIENTATED TRANSPORT: TCP**

**Reliable Data Transfer**

TCP creates a RDT on top of IP’s unreliable best-effort service.

* TCP ensures that the data stream is uncorrupted, without gaps, without duplication and in sequence (byte stream is the same byte stream sent by the end system on the other side of the connection)

Earlier, we assume that an individual timer is associated with each transmitted (but not ACK’d) segment.

* This is great in theory, but timer management can require considerable overehead.
* Recommended TCP timer mgmt. uses only a **single retransmission timer**, even if there are multiple transmitted but not yet acknowledged segments.

**SKIPPED SOME NOTES HERE**

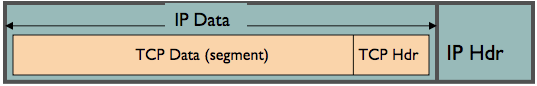
**SUMMARY: Components of Reliable Data Transfer**

1. **Checksums** (for error detection)
2. **Timers** (for loss detection)
3. **Acknowledgements**
   * Cumulative, Selective
4. **Sequence numbers** (duplicates, windows)
5. **Sliding Windows** (for efficiency)
   * Go-Back-N (GBN)
   * Selective Replay (SR)

**TCP is similar to previous components of RDT, but with some key differences**

* **Checksum:** TCP calculates checksum over both the HEADER (TPC “pseudo” header) and DATA (TCP segment)

<http://lateblt.tripod.com/bit34.txt>

* **Sequence & Acknowledgement numbers:** Are byte offsets
  + TCP segments are sent when (1) Segment is full (2) A timeout occurs
  + Segment Size:  
    
    - IP Data: No bigger than **Max Transmission Unit (MTU)**
    - TCP packet: IP packet with a TCP header and data
    - TCP segment: No more than **Max Segment Size (MSS)** **[ MSS = MTU – IP\_HEADER – TCP\_HEADER ]**

Point-To-Point Example: Client 🡪 Server (Assume Server always transmits a 0 byte payload)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **C-S** | **SYN** | **ACK** | **L** | **Description** |
| **Client** | 0 | 0 | 0 | I want to initiate a connection (SYN) |
| **Server** | 0 | 1 | 0 | I want to initiate a connection AND acknowledge your SYN packet (SYN-ACK) |
| **Client** | 1 | 0+1 | 0 | I acknowledge your SYN packet (ACK) |
| **Three-Way-Handshake Complete** | | |  | |
| **Client** | 1 | 1 | 25 | I am sending you data of size 25 bytes |
| **Server** | 1 | 1+25 | 0 | I acknowledge your data. I expect the next packet to have bytes starting at seq 25+1. |
| **Client** | 26 | 1 | 25 | I am sending you data of size 25 bytes |
| **Server** | 1 | 26+25 | 0 | I acknowledge your data. I expect the next packet to have bytes starting at seq 26 + 25 |
| **Client** | 51 | 1 | 25 | I am sending you data of size 25 bytes. |
| **Server** | 1 | 51+25 | 0 | I acknowledge your data. I expect the next packet to have bytes starting at seq 51 + 25 |
| **Begin Connection Tear-Down** | | |  | | |
| **C-S** | **FIN** | **ACK** | **L** | **Description** | |
| **Client** | 76 | 1 | 0 | I want to break the connection (FIN) | |
| **Server** | 1 | 76+1 | 0 | I acknowledge your break AND will initiate a break (ACK – FIN) | |
| **Client** | 77 | 1+1 | 0 | I acknowledge your FIN (ACK) | |

NOTE: Sequence number of next packet = number in last acknowledgement field

Continue from prev

* Receiver sends cumulative acknowledgement (like GBN)
* Receiver do not drop out-of-sequence packets (like SR)
* Sender maintains a single retransmission timer (like GBN) and retransmits on timeout

**DO MORE NOTES ON:**

* **Packet Loss**
* **Fast Retransmission**
* **Pipelined (Sliding Window) Protocols:**(sending multiple in-flight, yet-to-be-acknowledged pkts – TCP congestion and flow control set window size)
  + **Go-Back-N**
  + **Selective Repeat**

**TCP round trip time, timeout**Set up TCP timeout by choosing a value longer than RTT, however RTT varies.

* Choose value too short: premature timeout, unnecessary retransmission
* Choose value too long: slow reaction to segment loss and lower throughput for connection

**1. How to measure EstimatedRTT:**

Exponential Weighted Moving Average

EstimatedRTTCURR = (1 – a) \* EstimatedRTTPREV + a \* SampleRTTRECENT

* SampleRTT
  + Time measured from segment transmission until ACK receipt (ignoring retransmissions)
  + Current value of RTT
* Typical value of a = 0.125

**2. How to measure timeout interval: EstimatedRTT + “Safety Margin”**

RTT Deviation is calculated by

DevRTT = (1 – b) \* DevRTT + B \* |SampleRTT – EstimatedRTT|

* Typical value of b = 0.25

Timeout Value is calculated by

Timeout Interval = EstimatedRTT + 4 \* DevRTT

* 4 \* DevRTT = The “Safety Margin”

A good explanation of all the above:<https://www.quora.com/How-does-TCP-round-trip-time-RTT-estimation-work-How-different-is-the-implementation-across-operating-systems>