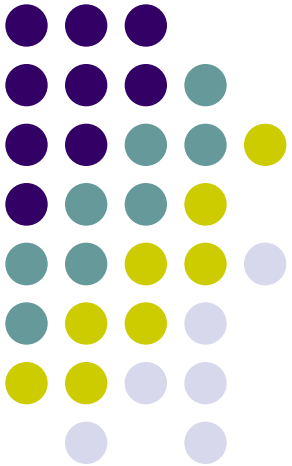
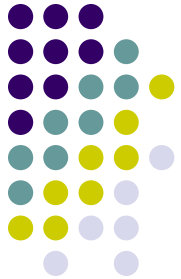
ELEN 4017

Network Fundamentals

Lecture 15



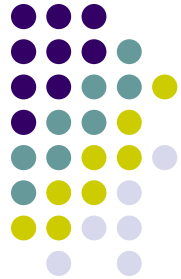
Purpose of lecture



Chapter 3: Transport Layer

- Reliable data transfer

Developing a reliable protocol



- Reliability implies:
  - No data is corrupted (flipped bits)
  - Data is delivered in order in which it was sent.
  - No data is lost.
- We will incrementally develop a reliable data transfer protocol – `rdt`
- Sending side of protocol is called `rdt_send`
- Receiving side is called `rdt_rcv`.
- The protocol will make use of an unreliable data transfer protocol at lower layers – `udt_send / udt_receive`
- To simplify we will consider uni-directional data transfer. Note: Protocol will exchange **control** messages in both directions.

Service abstraction to higher layers

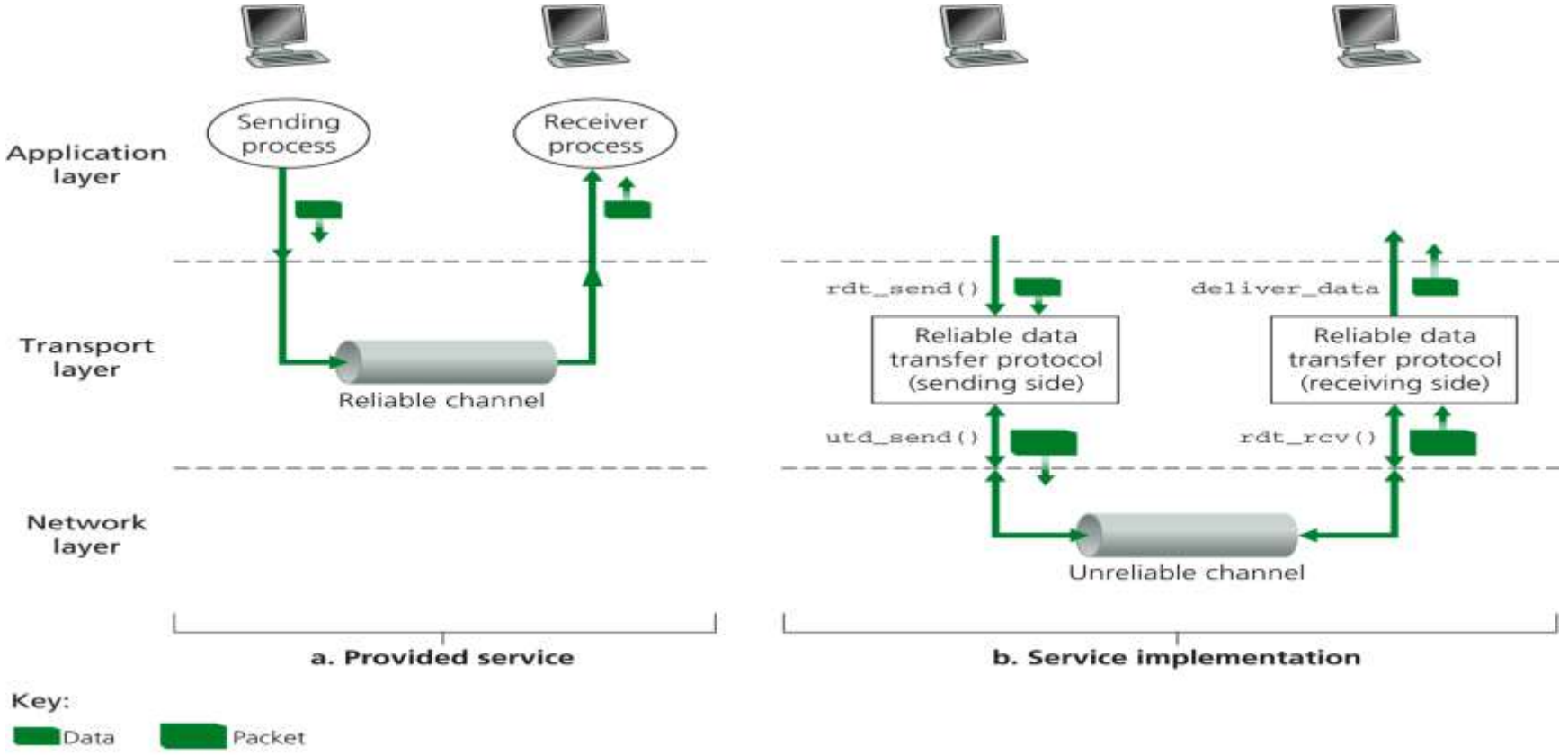
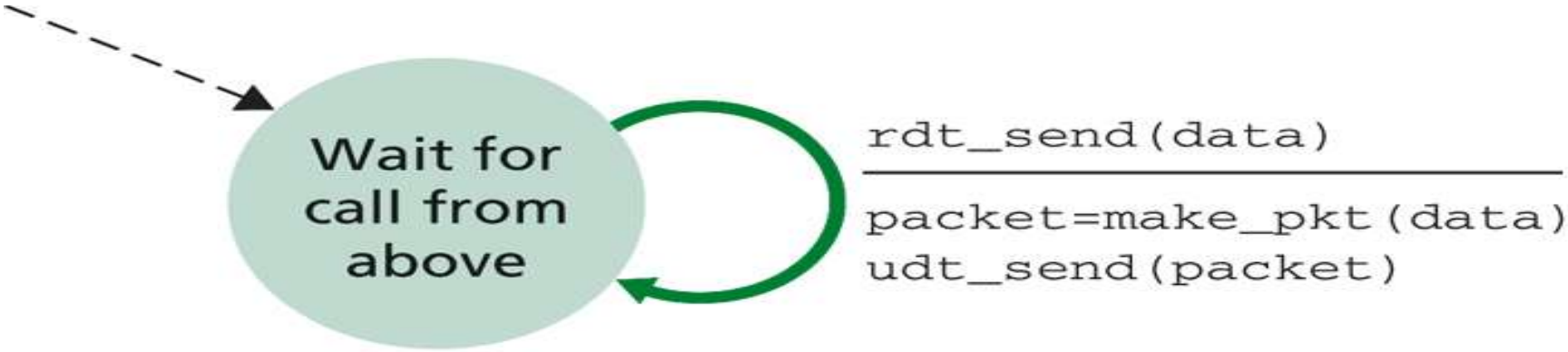


Figure 3.8 ◆ Reliable data transfer: Service model and service implementation

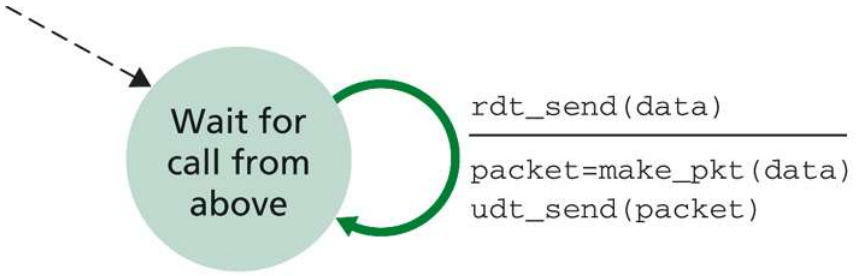
Finite State Machines (FSM)



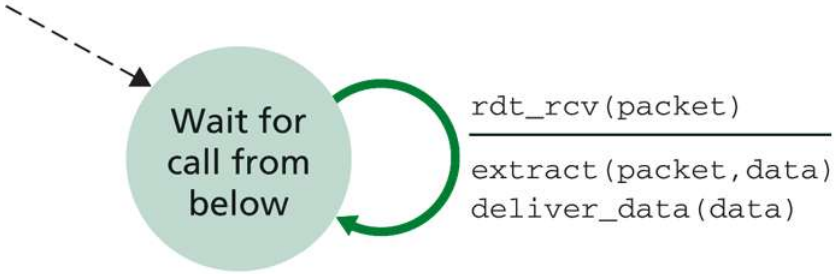
- Arrows indicate transition from 1 state to next.
- Event is shown above horizontal line.
- Actions taken are shown below. Λ means no action
- Dashed arrow is initial state.

rdt 1.0 – Completely reliable channel

- The sending side of rdt accepts data from upper layer, creates a packet and sends packet into channel.
- The call to rdt_send occurs from upper layer i.e. application.
- rdt_send is immediately available to process the next packet.
- On receiving side, rdt receives a packet from underlying channel, extracts data from packet, and delivers data to higher layer.

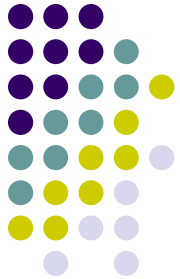


a. rdt1.0: sending side



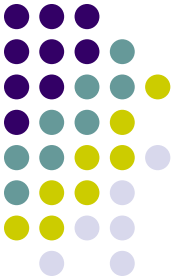
b. rdt1.0: receiving side

Channel with bit errors

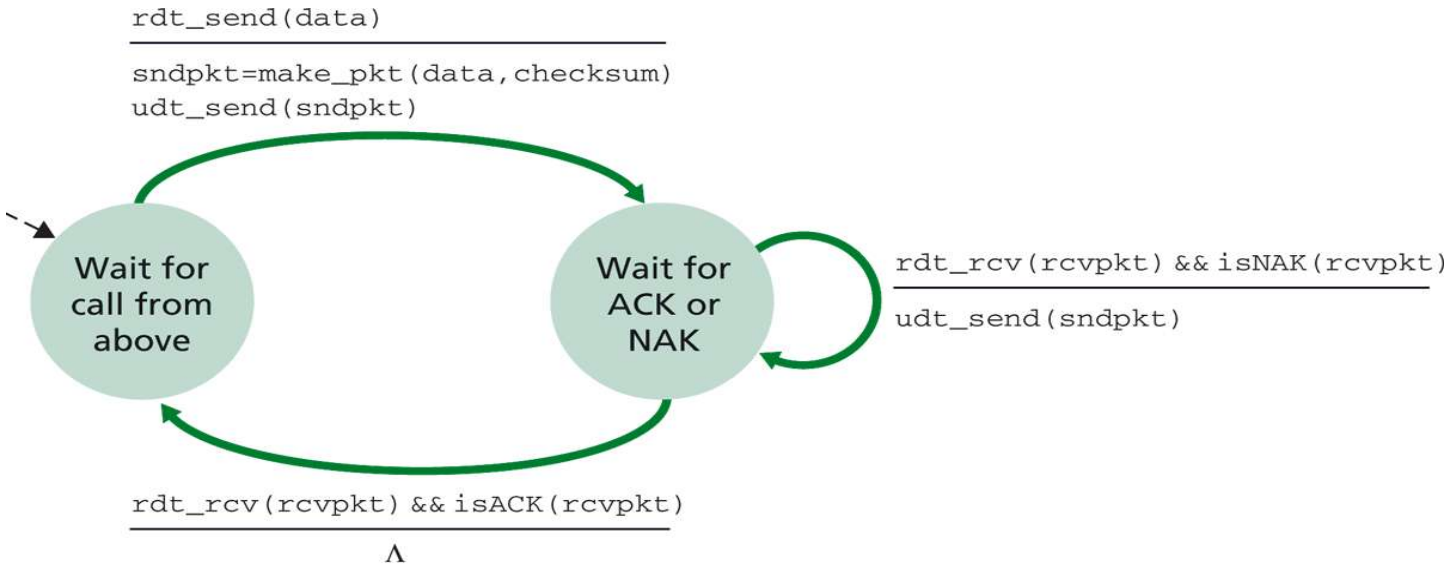


- Consider dictating a long message over the phone.
- In a typical scenario message taker might acknowledge each sentence with OK or “Please repeat that”.
- This protocol uses positive and negative acknowledgements.
- In computer networks, protocols that rely on retransmission are called Automatic Repeat ReQuest (ARQ) protocols.
- 3 capabilities are needed in ARQ:
  - Error detection
  - Receiver feedback (ACK and NAK messages)
  - Retransmission
- Lets consider rdt 2.0.

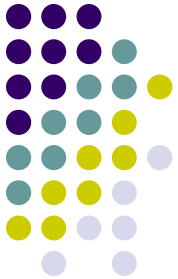
rdt 2.0: Reliable transfer over a channel with bit errors - Sender



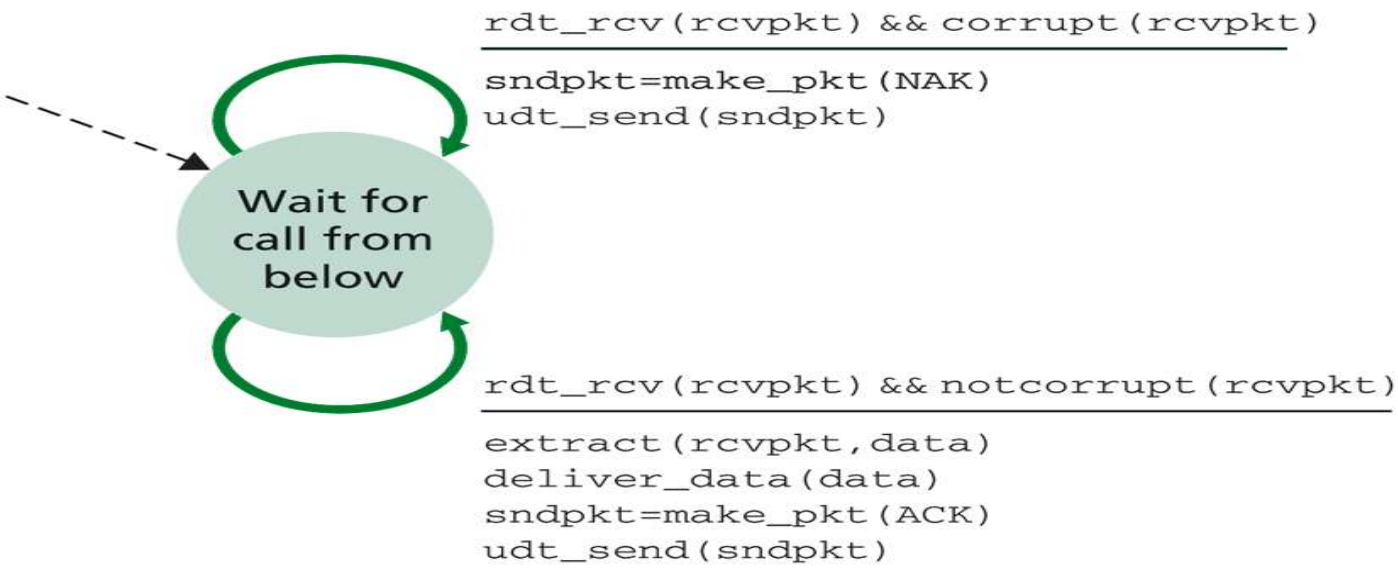
- In this case, assume that **no packets are lost**, but bits can be **corrupted** during transmission.
- A checksum is created by sender, so that receiver can detect if corruption has occurred.
- After packet is sent, wait for response.
- ACK → do nothing. NAK → retransmit.
- We can encode the ACK /NAK as a single bit.



rdt 2.0: Reliable transfer over a channel with bit errors - Receiver

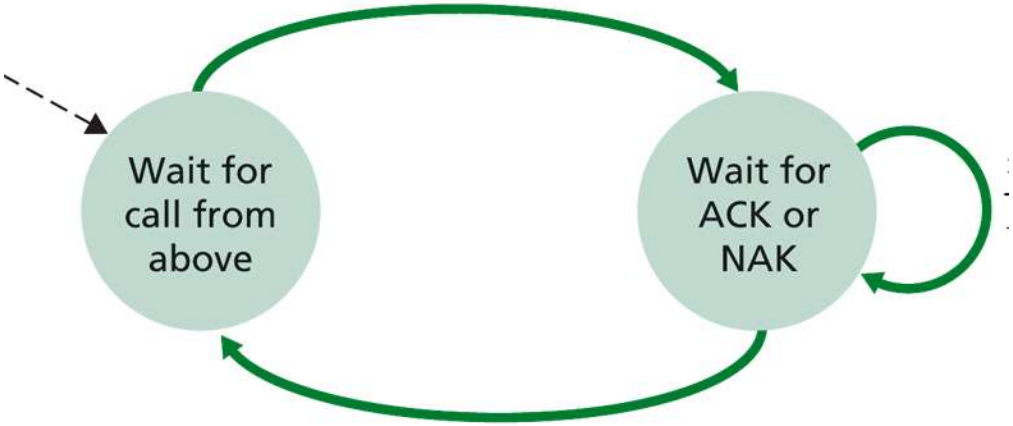


- Errors are detected by using the checksum.
- In case of no corruption, ACK is sent and data forwarded upwards
- In case of corruption, NAK is sent.

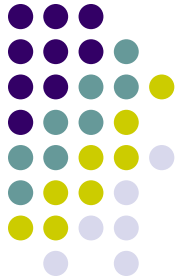


Stop-and-wait

- When rdt2.0_send is in state wait for ACK/NAK, it cannot process any more data, thus it is called a Stop-and-wait protocol.
- Thus throughput of this protocol is poor.

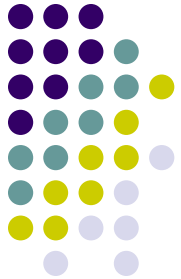


Corrupt ACK/NAK



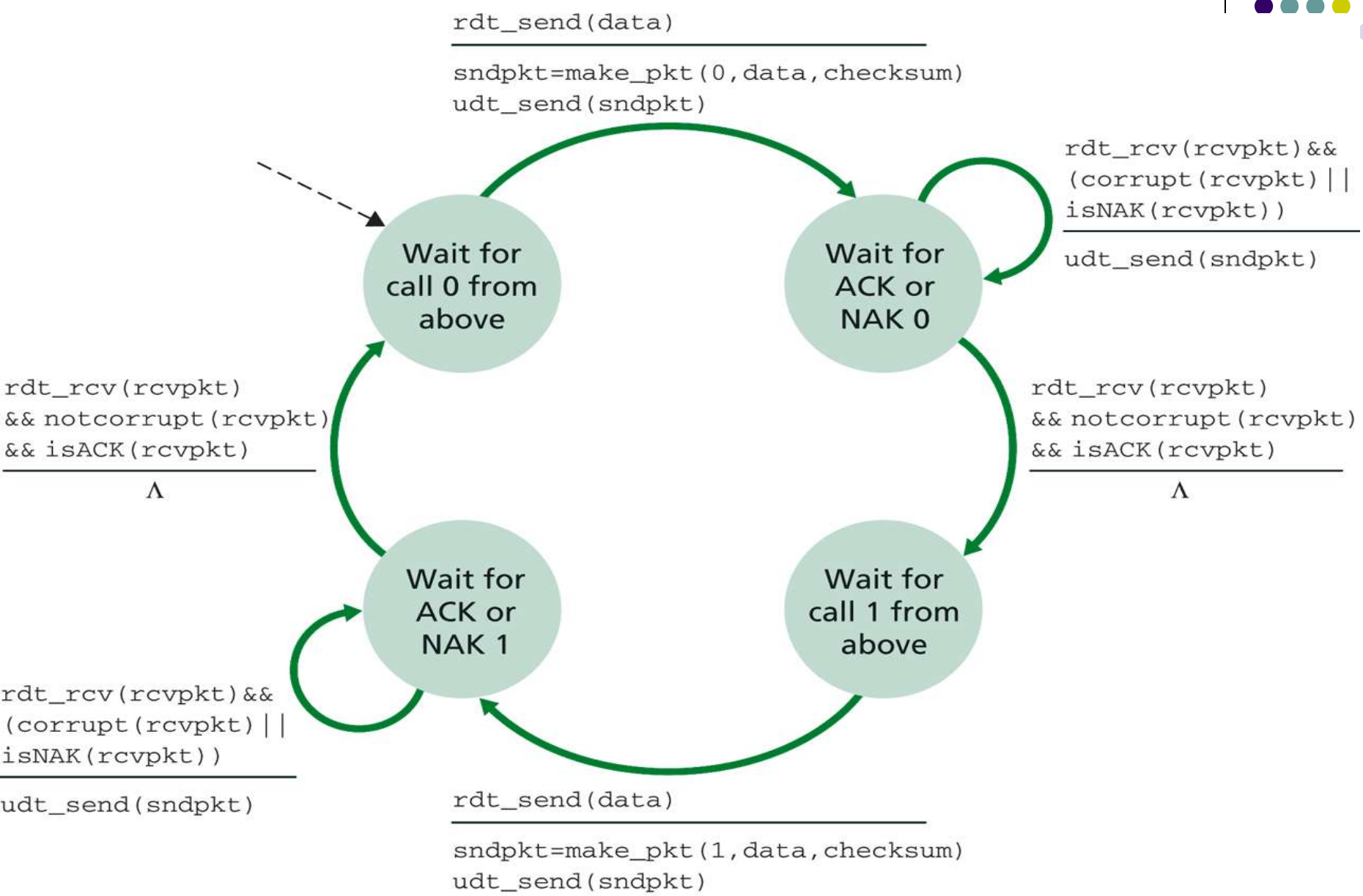
- A more serious problem is what happens in the case of the ACK/NAK message **itself being corrupted**.
- We could add a checksum for ACK/NAK to resolve this, but how do we recover from this state ?
- The difficulty is that in the case of a corrupt ACK/NAK, the sender has no way of knowing if the data was received correctly or not.
- 3 possibilities exist:
  - **Add new messages.** Consider the human analogy. Sender might say "What did you say?" (→ a new sender to receiver packet is sent). But this packet could also be garbled, and so we don't really solve anything with this approach.
  - **Forward Error Correction** – Add enough checksum bits so that receiver can recover from corrupt message without requesting a retransmit.
  - **Resend when corrupt ACK/NAK**- Sender must resend current data packet when garbled ACK/NAK received. This introduces the problem of **duplicate packets**. The receiver has no way of knowing whether the ACK/NAK sent was correctly received at the sender, and thus cannot know if the packet received is a new packet or not. The solution to this problem is to add a **sequence number** to the outgoing packet.

Sequence number

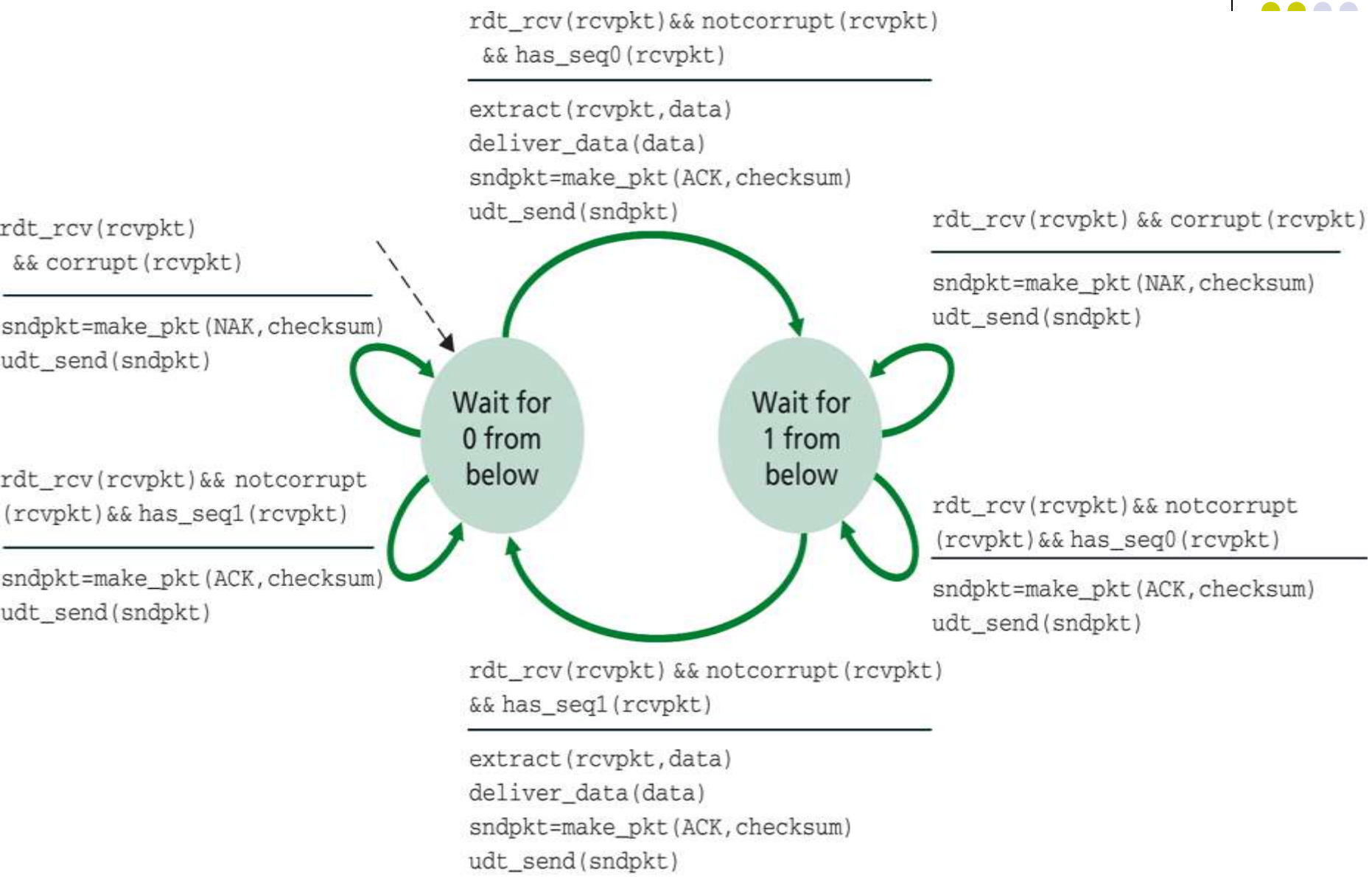


- Logic on receiver:
  - If sequence number of received packet is same as that of previously received packet → it must be a retransmit.
  - If it's a new sequence number then it's a new packet.
- For the case of stop-and-wait the sequence number can be 1 bit, and this value can alternate.

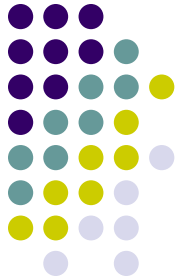
rdt 2.1 sender



rdt 2.1 receiver

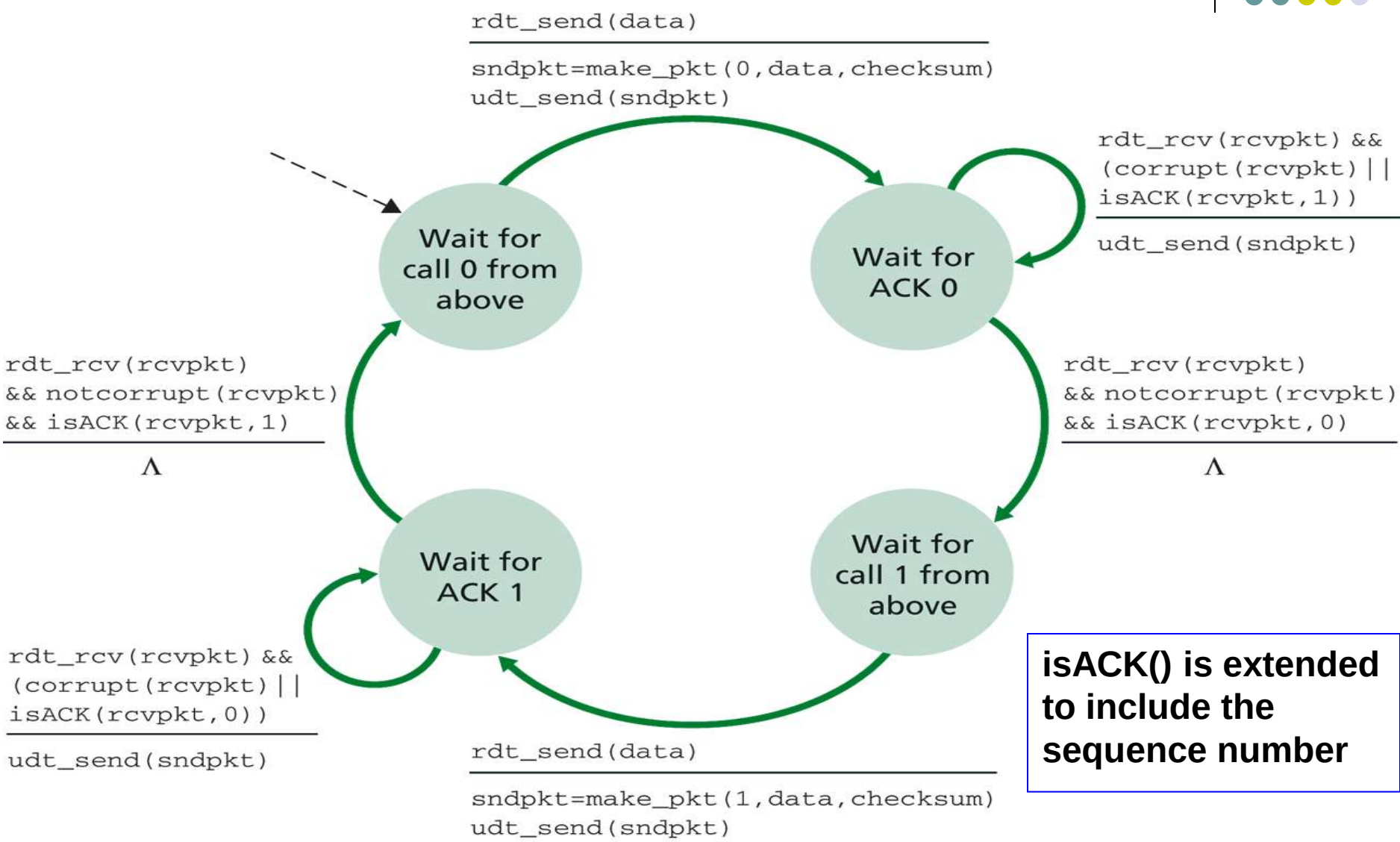


rdt 2.2: Replace NAK with ACK



- We can achieve same effect as a NAK, if we send an ACK for the **last correctly received packet**.
- Thus a sender receiving two ACK's for the same packet, knows that the subsequent packet was not received. To achieve this the ACK message must be extended to include the sequence number of the packet being acknowledged.

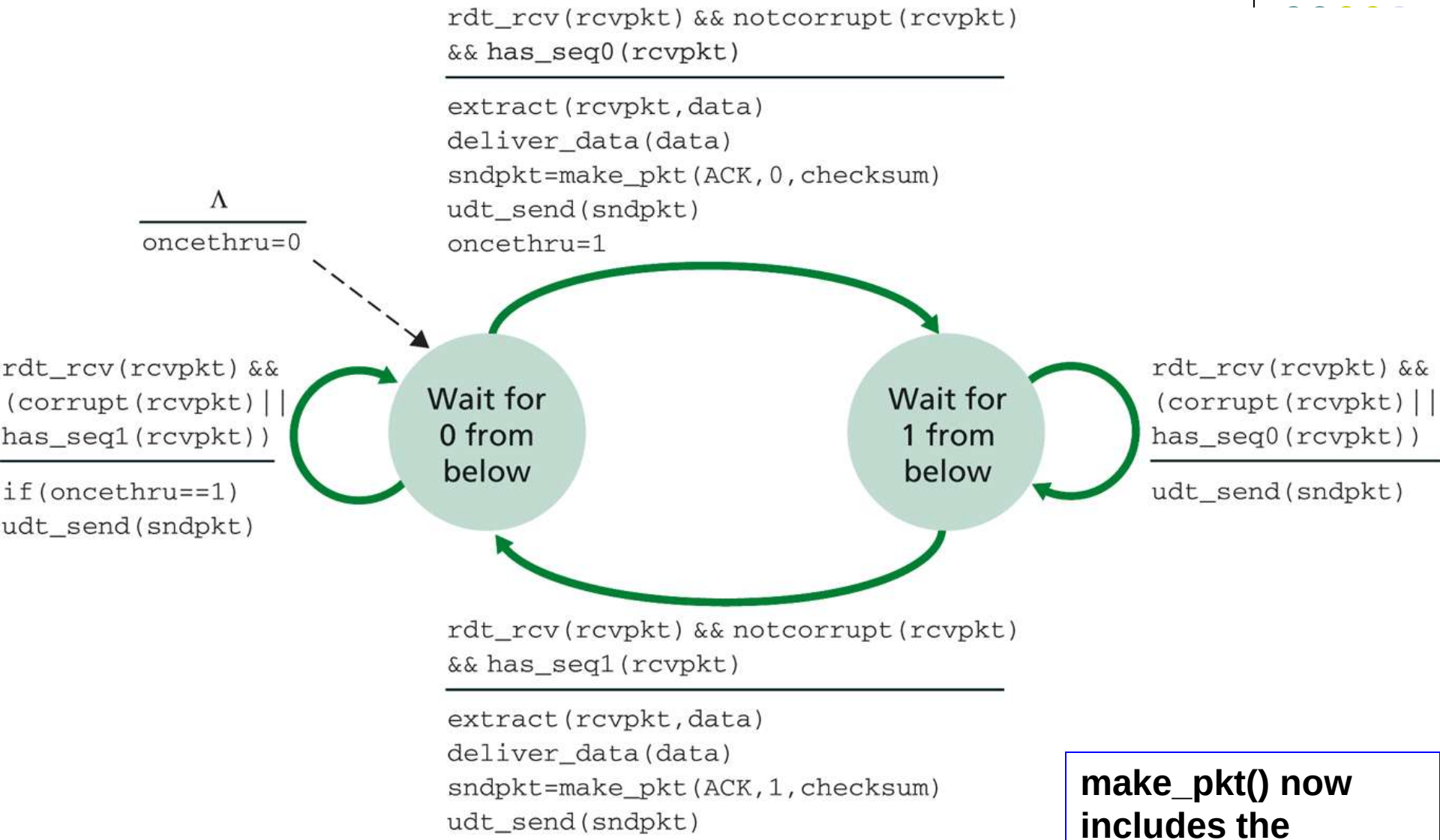
rdt 2.2 sender



isACK() is extended to include the sequence number

Figure 3.13 ♦ rdt2.2 sender

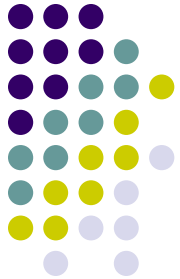
rdt 2.2 receiver



make_pkt() now includes the sequence number

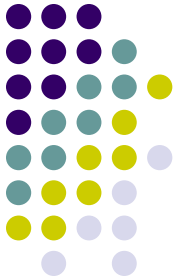
Figure 3.14 ♦ rdt2.2 receiver

rdt 3.0: Reliable transfer over a lossy channel with bit errors



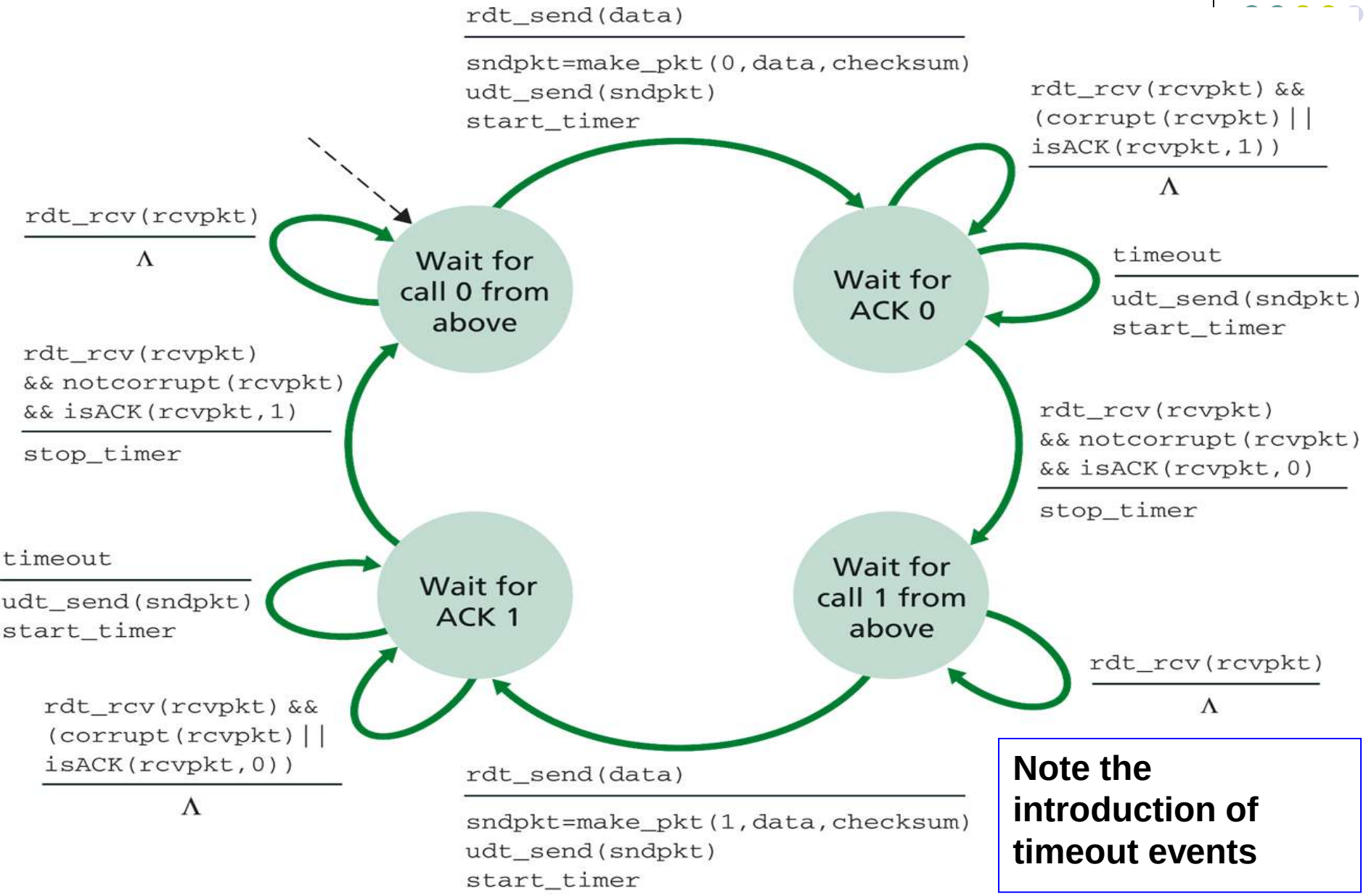
- In addition to bit corruption, the channel is **now allowed to lose packets**:
- Thus we must address:
  - How to detect packet loss?
  - How to recover from packet loss?
- There are many strategies to deal with this. We will consider case **where sender is responsible** for this.
- Consider the case where sender transmits a packet and either that packet or the receivers ACK for that packet is lost.
- In either case, no reply occurs at the sender.
- If the sender is willing to wait long enough so that it is **certain** that a packet has been lost, it can simply retransmit the packet.
- Our current rdt 2.2 receiver would support this.

Transmit timeout



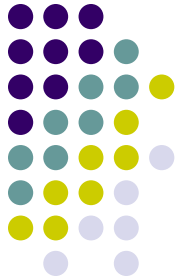
- How long should the sender wait ?
- Sender should wait at least 1 x RTT, but this is very difficult to estimate (router delays, different paths, …). Even a worst-case estimate is difficult.
- If we use the worst case estimate, this could result in a very long wait, but we would like the protocol to recover from packet loss as soon as possible.
- However, the rdt 2.2 protocol is able to cope with **duplicate packets** by using sequence numbers. Thus we can retransmit even if a packet loss has not occurred.
- A **countdown timer** is used to cater for **possible** packet loss.
- The sender starts the timer when a message is sent. In the case a response is received the timer is stopped. In the case of no response, the timer generates a new event, so that sender can handle this scenario.
- Since the sequence numbers alternate between 0 and 1, **rdt 3.0** is sometimes called the **alternating bit protocol.**

rdt 3.0 sender



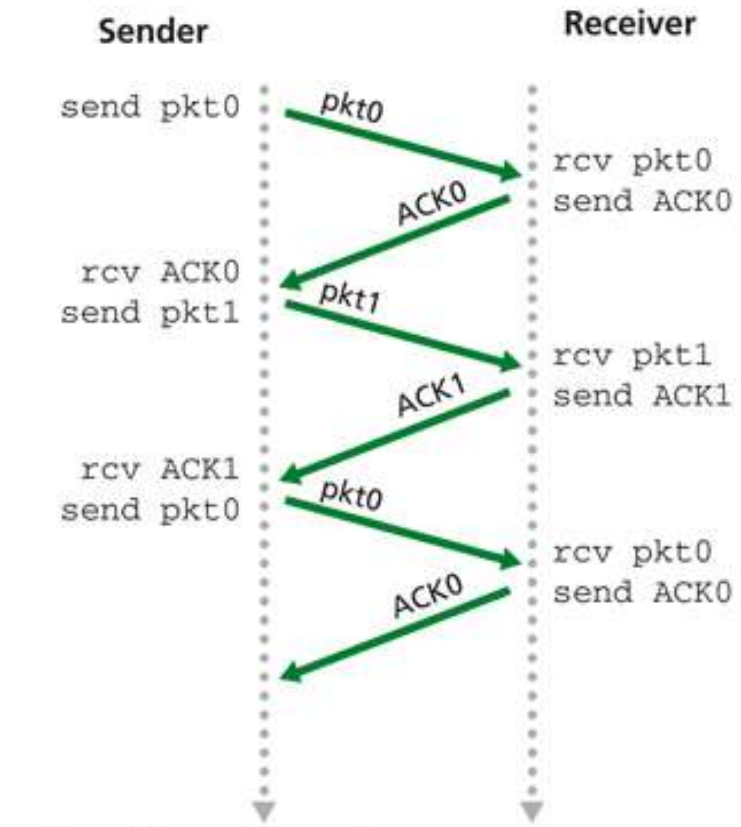
Note the introduction of timeout events

rdt3.0 receiver

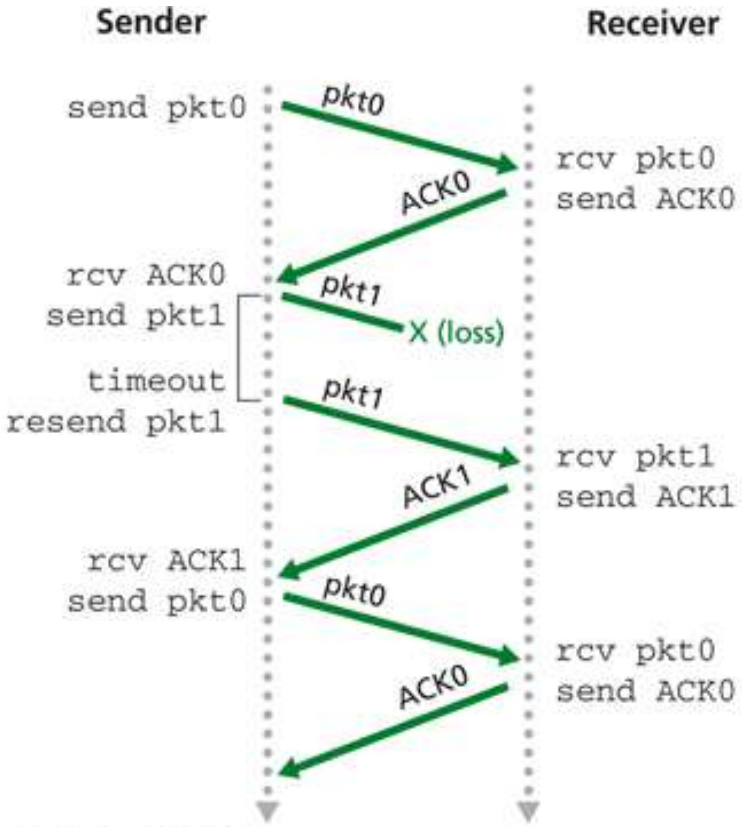


- What would this look like ? Any changes needed ?

Message flows

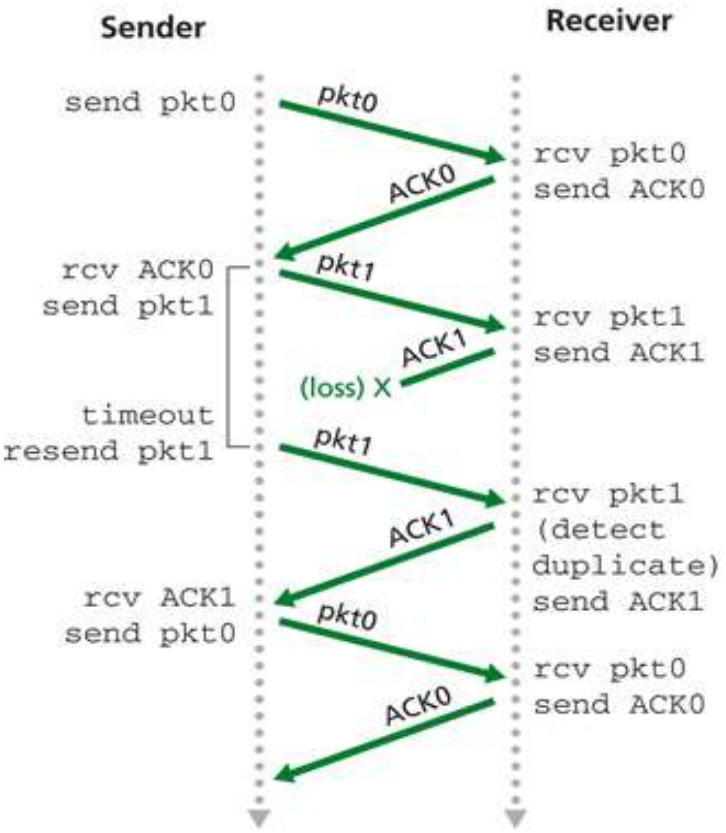


a. Operation with no loss

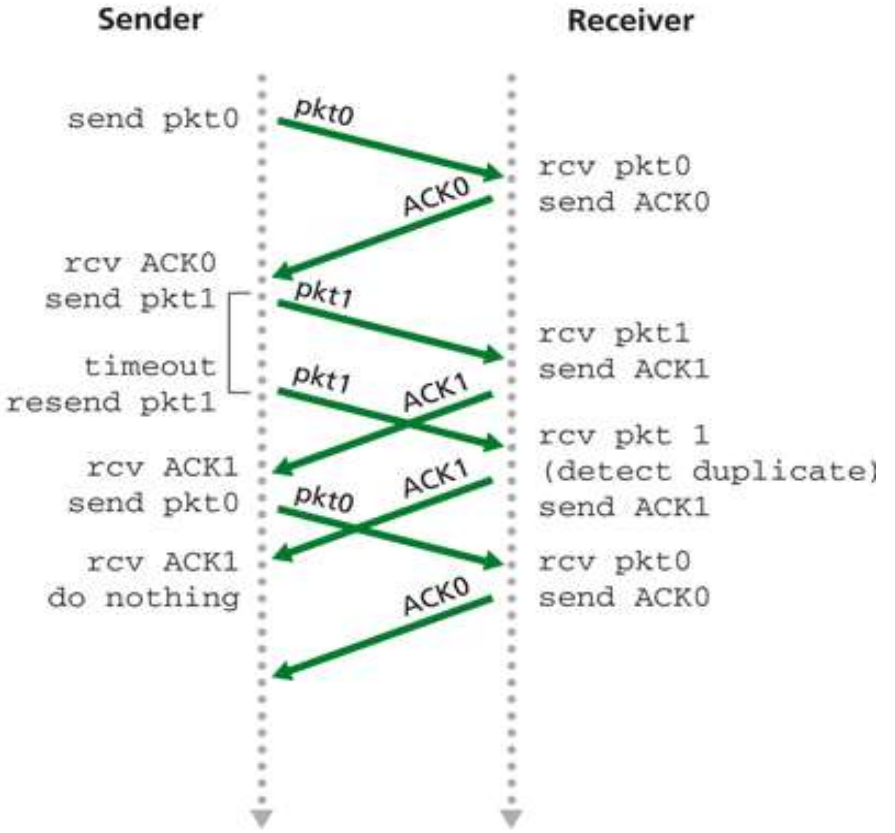


b. Lost packet

Message flows (contd)



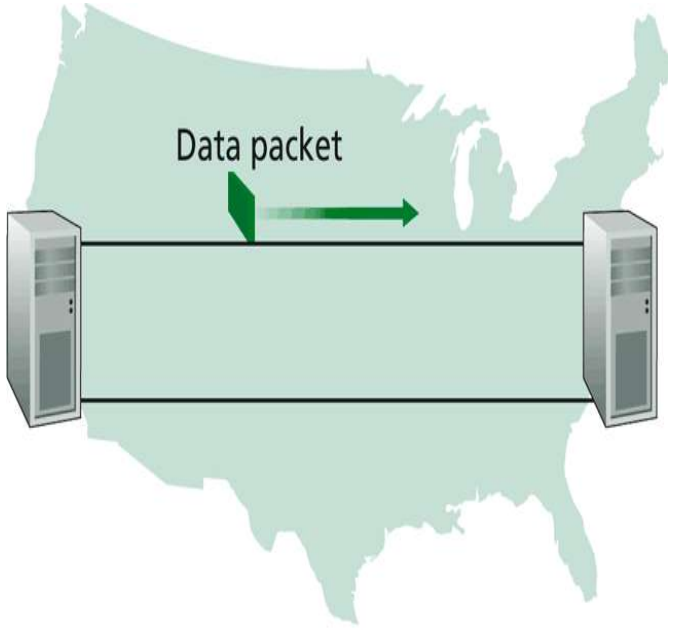
c. Lost ACK



d. Premature timeout

Utilization of a stop-and-wait protocol

- Consider two hosts located on opposite sides of U.S
- RTT t_{prop} = 30 * 10^{-3} s
- Transmission rate = 1Gbps
- For a packet size of 1000 bytes, transmission time t_{trans} = 8 *10^{-6} s
- Utilization (U) is the fraction of the time the sender is sending bits into the channel.
- U = $t_{trans}/(t_{trans}$ + RTT)
- For this case, utilization is very low (0.00027)
- Throughput is actually 267 kps for this 1Gbps link.
- **How do we improve this ?**



a. A stop-and-wait protocol in operation