Chapter 3- Part 3

The Data Link Layer

Many protocols/algorithms discussed in this chapter apply to other layers

Elementary Data Link Protocols

1. An Unrestricted Simplex Protocol (UTOPIA)
2. A Simplex Stop-and-Wait Protocol
3. A Simplex Protocol for a Noisy Channel

- Simplex means:
  - Data can flow in one direction only
  - Control packets can flow in both directions

Variations of protocols described in this section are used in other layers

Basic Assumptions

- Assume physical, data link, and network layers are independent processes that communicate only using messages
- The physical layer does **NOT reorder** packets.
  - In general, physical wires and media do not re-order packets.
  - Reordering usually occurs in virtual channels

- **Assume network layer at endpoints want *reliable* connection-oriented channel**
  - **In order delivery: Packets are always delivered to network layer in order**
  - **No duplicate packets: Never deliver duplicate packets to the network layer**
  - **No packets are lost: Never skip delivering a packet to the network layer.**
  - **No error packets: The payload delivered to the network layer of the receiver is identical to the payload sent by the network layer at the sender**
  - **No deadlock: Packets are delivered in a *finite amount of time*. E.g a packet never gets stuck at the sender or the receiver datalink layer**

    Note: Unbounded wait time (e.g. due to strict priority) is considered finite. Hence for a protocol to be reliable it need not provide bounded time but it has to be finite*)

Basic Assumptions

- Frame header is never given to network layer (to ensure layer independence)
- Assume medium is unreliable. Hence timeout mechanism is used by sender to retransmit.
- Assume timers can be (re)started and stopped at any time
- Assume receiver waits indefinitely and only wakes on events (e.g. packet arrive, received errored packet, timer expired)
- Subtraction and addition of sequence number is assumed to be *circular*
  - We *never* get negative sequence number
- Assume network layer transmitter has infinite supply of data
  - we will remove this assumption later

Link layer environment

- Link layer protocol implementations use library functions
  - See code (`protocol.h`) for more details

Group	Library Function	Description
Network layer	from_network_layer(&packet) to_network_layer(&packet) enable_network_layer() disable_network_layer()	Take a packet from network layer to send Deliver a received packet to network layer Let network cause “ready” events Prevent network “ready” events
Physical layer	from_physical_layer(&frame) to_physical_layer(&frame)	Get an incoming frame from physical layer Pass an outgoing frame to physical layer
Events & timers	wait_for_event(&event) start_timer(seq_nr) stop_timer(seq_nr) start_ack_timer() stop_ack_timer()	Wait for a packet / frame / timer event Start a countdown timer running Stop a countdown timer from running Start the ACK countdown timer Stop the ACK countdown timer

Protocol Definitions

```
#define MAX_PKT 1024                                  /* determines packet size in bytes */

typedef enum {false, true} boolean;                   /* boolean type */
typedef unsigned int seq_nr;                          /* sequence or ack numbers */
typedef struct {unsigned char data[MAX_PKT];} packet;/*    packet definition */
typedef enum {data, ack, nak} frame_kind;             /* frame_kind definition */

typedef struct {                                      /* frames are transported in this layer */
  frame_kind kind;                                    /* what kind of a frame is it? */
  seq_nr seq;                                         /* sequence number */
  seq_nr ack;                                         /* acknowledgement number */
  packet info;                                        /* the network layer packet */
} frame;
```

Frame header

The sequence number of the frame that we are sending in this packet

The sequence number of the frame to be received

Payload

Continued →

Some definitions needed in the protocols to follow.
These are located in the file protocol.h.

Protocol Definitions (ctd.)

```
/* Wait for an event to happen; return its type in event. */
void wait_for_event(event_type *event);

/* Fetch a packet from the network layer for transmission on the channel. */
void from_network_layer(packet *p);

/* Deliver information from an inbound frame to the network layer. */
void to_network_layer(packet *p);

/* Go get an inbound frame from the physical layer and copy it to r. */
void from_physical_layer(frame *r);

/* Pass the frame to the physical layer for transmission. */
void to_physical_layer(frame *s);

/* Start the clock running and enable the timeout event. */
void start_timer(seq_nr k);

/* Stop the clock and disable the timeout event. */
void stop_timer(seq_nr k);

/* Start an auxiliary timer and enable the ack_timeout event. */
void start_ack_timer(void);

/* Stop the auxiliary timer and disable the ack_timeout event. */
void stop_ack_timer(void);

/* Allow the network layer to cause a network_layer_ready event. */
void enable_network_layer(void);

/* Forbid the network layer from causing a network_layer_ready event. */
void disable_network_layer(void);

/* Macro inc is expanded in-line: Increment k circularly. */
#define inc(k) if (k < MAX_SEQ) k = k + 1; else k = 0
```

Separate timer for each sequence number "k"

Some definitions needed in the protocols to follow. These are located in the file protocol.h.

Circular increment

Unrestricted Simplex Protocol

- Perfect world ☺
- Error free communication channel
- Sender has infinite buffer
- Receiver has infinite buffer
- Receiver can process all input infinitely fast
- Sender network layer has infinite packet supply to send
- Propagation delay is negligible

Unrestricted Simplex Protocol

```
/* Protocol 1 (utopia) provides for data transmission in one direction only, from
   sender to receiver.  The communication channel is assumed to be error free,
   and the receiver is assumed to be able to process all the input infinitely quickly.
   Consequently, the sender just sits in a loop pumping data out onto the line as
   fast as it can. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender1(void)
{
  frame s;                              /* buffer for an outbound frame */
  packet buffer;                        /* buffer for an outbound packet */

  while (true) {
      from_network_layer(&buffer);      /* go get something to send */
      s.info = buffer;                  /* copy it into s for transmission */
      to_physical_layer(&s);            /* send it on its way */
  }                                      /*  Tomorrow, and tomorrow, and tomorrow,
                                            Creeps in this petty pace from day to day
                                            To the last syllable of recorded time
                                                - Macbeth, V, v */
}

void receiver1(void)
{
  frame r;
  event_type event;                     /* filled in by wait, but not used here */

  while (true) {
      wait_for_event(&event);           /* only possibility is frame_arrival */
      from_physical_layer(&r);          /* go get the inbound frame */
      to_network_layer(&r.info);        /* pass the data to the network layer */
  }
}
```

Simplex Stop-and-Wait Protocol

- Semi-perfect world
- Drop the unrealistic assumption at the receiver. Assume
  - Receiver has *finite buffer*
  - Receiver has *finite processing* capacity
- Error free channel
- Sender has infinite buffer
- Sender Network layer has infinite supply of packets to send
- *What and where error can occur?*

Packet drop at the receiver side

Simplex Stop-and-Wait Protocol

- We do *not need a timeout* mechanism in this protocol (*Why?*)
- We do not need sequence number (*why?*)

```
/* Protocol 2 (stop-and-wait) also provides for a one-directional flow of data from
   sender to receiver. The communication channel is once again assumed to be error
   free, as in protocol 1. However, this time, the receiver has only a finite buffer
   capacity and a finite processing speed, so the protocol must explicitly prevent
   the sender from flooding the receiver with data faster than it can be handled. */

typedef enum {frame_arrival} event_type;
#include "protocol.h"

void sender2(void)
{
 frame s;                          /* buffer for an outbound frame */
 packet buffer;                    /* buffer for an outbound packet */
 event_type event;                 /* frame_arrival is the only possibility */

 while (true) {
     from_network_layer(&buffer);  /* go get something to send */
     s.info = buffer;              /* copy it into s for transmission */
     to_physical_layer(&s);        /* bye bye little frame */
     wait_for_event(&event);       /* do not proceed until given the go ahead */
 }
}

void receiver2(void)
{
 frame r, s;                       /* buffers for frames */
 event_type event;                 /* frame_arrival is the only possibility */
 while (true) {
     wait_for_event(&event);       /* only possibility is frame_arrival */
     from_physical_layer(&r);      /* go get the inbound frame */
     to_network_layer(&r.info);    /* pass the data to the network layer */
     to_physical_layer(&s);        /* send a dummy frame to awaken sender */
 }
}
```

Simplex Stop-and-Wait Protocol

- We can see from the protocol code that we do *NOT have a timeout* on the sender side. *Why don't we need a timer even though the receiver has a finite buffer and hence packets may be dropped?*
  - Actually, there is *no possibility of packet drop* because the sender will send the next packet ONLY when it receives an ACK from the receiver.
  - Because we assumed perfect channel, the packet will arrive at the receiver and the ACK will arrive at the sender
  - Hence there is no need for timeout at the sender
  - In other words, we overcome the finite CPU and buffer capacity of the receiver by having the ***receiver control the transmission***.
- We do not need a sequence number , *why?*
  - because we do not lose packets
  - Because there is no possibility of packet duplication
  - Because there is no possibility of packet reorder
- *Why doesn't the sender inspect the packet type?*

A Simplex Protocol for a Noisy Channel

- Assume *non-zero* variable *propagation delay* along the channel in *both direction*
- Assume possible frame damage or loss in *both direction*
- Assume errored packets are *detected correctly* *(i.e. perfect error detection)*
- Simple modification to the previous protocol
- Use a **timeout** at the sender
- Sender only transmits next frame if it received *correct ACK* (damaged ACK are discarded)
- **If it does not receive a correct ACK, it times-out and re-transmits the same frame**

A Simplex Protocol for a Noisy Channel

- Assume *non-zero* variable *propagation delay* along the channel in *both direction*
- Assume possible frame damage or loss in *both direction*
- Assume errored packets are *detected correctly* (*i.e. perfect error detection)*
- Simple modification to the previous protocol
- Use a **timeout** at the sender
- Sender only transmits next frame if it received *correct ACK* (damaged ACK are discarded)
- **If it does not receive a correct ACK, it times-out and re-transmits the same frame**
- ***Is this correct?***

A Simplex Protocol for a Noisy Channel

- Assume *non-zero* variable *propagation delay* along the channel in *both direction*
- Assume possible frame damage or loss in *both direction*
- Assume errored packets are *detected correctly* (*i.e. perfect error detection*)
- Simple modification to the previous protocol
- Use a **timeout** at the sender
- Sender only transmits next frame if it received *correct ACK* (damaged ACK are discarded)
- **If it does not receive a correct ACK, it times-out and re-transmits the same frame**
- ***Is this correct?***

Duplicate Frame if ACK is lost

A Simplex Protocol for a Noisy Channel

- Assume *non-zero* variable *propagation delay* along the channel in *both direction*
- Assume possible frame damage or loss in *both direction*
- Assume errored packets are *detected correctly*
- Assume that the propagation delay is greater than zero
- Simple modification to the previous protocol
- Use a timeout at the sender
- Sender only transmits next frame if it received *correct ACK* (damaged ACK are discarded)
- If it does not receive a correct ACK, it times-out and re-transmits the same frame
- *Is this correct?* ***Duplicate Frame if ACK is lost***
- We need a sequence number (*what size?*)
- Protocols where sender waits to ACK from receiver and retransmits after time out are sometimes called
  - PAR: Positive Acknowledge with Retransmit
  - **ARQ: Automatic Repeat reQuest**
  - See for example RFC3366: Advice to Link Designers on Link ARQ
- Hence the rest of the protocols in this chapter belong to the class of protocols known as ARQ

A Simplex Protocol for a Noisy Channel

A positive acknowledgement with retransmission protocol.

*Why does the sender need to check the sequence number in **s.ack**?*

I.e. Can the sender just send the next frame as soon as it receives an ACK, assuming that the physical layer drops any corrupted packet?

- Assume that an errored frame generates the event "checksum_err". We will ignore this event.
- This is similar to assuming that physical layer discards errored packets

```
/* Protocol 3 (par) allows unidirectional data flow over an unreliable channel. */

#define MAX_SEQ 1                                   /* must be 1 for protocol 3 */
typedef enum  {frame_arrival, cksum_err, timeout} event_type;
#include "protocol.h"

void sender3(void)
{
  seq_nr next_frame_to_send;                        /* seq number of next outgoing frame */
  frame s;                                          /* scratch variable */
  packet buffer;                                    /* buffer for an outbound packet */
  event_type event;

  next_frame_to_send = 0;                           /* initialize outbound sequence numbers */
  from_network_layer(&buffer);                      /* fetch first packet */
  while (true) {
      s.info = buffer;                              /* construct a frame for transmission */
      s.seq = next_frame_to_send;                   /* insert sequence number in frame */
      to_physical_layer(&s);                        /* send it on its way */
      start_timer(s.seq);                           /* if answer takes too long, time out */
      wait_for_event(&event);                       /* frame_arrival, cksum_err, timeout */
      if (event == frame_arrival) {
          from_physical_layer(&s);                  /* get the acknowledgement */
          If (s.ack == 1- next_frame_to_send) {
              stop_timer(s.ack);                    /* turn the timer off */
              from_network_layer(&buffer);          /* get the next one to send */
              inc(next_frame_to_send);              /* invert next_frame_to_send */
          }
      }
  }
}
```

Circular increment

Continued →

A Simplex Protocol for a Noisy Channel (ctd.)

```
void receiver3(void)
{
  seq_nr frame_expected;
  frame r, s;
  event_type event;

  frame_expected = 0;
  while (true) {
      wait_for_event(&event);                  /* possibilities: frame_arrival, cksum_err */
      if (event == frame_arrival) {            /* a valid frame has arrived. */
          from_physical_layer(&r);             /* go get the newly arrived frame */
          if (r.seq == frame_expected) {       /* this is what we have been waiting for. */
              to_network_layer(&r.info);       /* pass the data to the network layer */
              inc(frame_expected);             /* next time expect the other sequence nr */
          }
          s.ack = frame_expected;              /* tell which frame is being acked */
          to_physical_layer(&s);               /* send acknowledgement */
      }
  }
}
```

- Assume that an errored frame generates the event "checksum_err". We will ignore this event.
- This is similar to assuming that physical layer discards errored packets

- Receiver sends ACK for next expected frame
- Receiver sends ACK ***every time*** it receives ANY packet

*•Suppose that the receiver does NOT send an ACK unless it receives a packet with the **expected sequence number**, will the protocol work correctly?*

Stop-and-Wait ARQ

Cases of Operations:

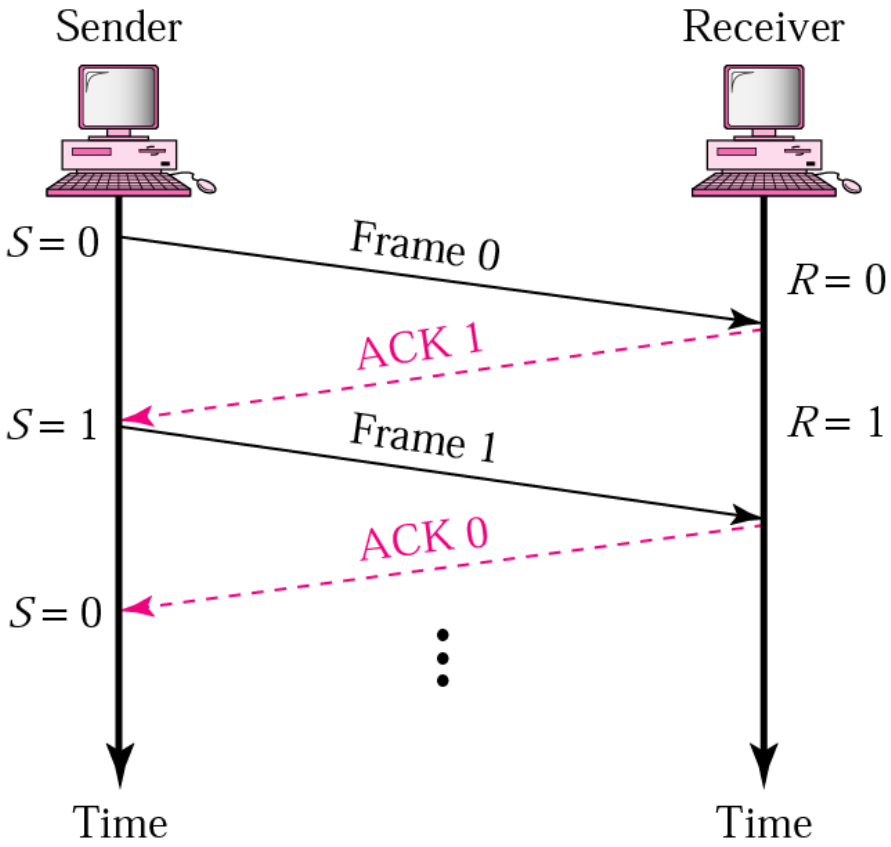
1. *Normal operation*
2. *The frame is lost*
3. *The Acknowledgment (ACK) is lost*
4. *The Ack is delayed*

S&W + Time out + Seq. No. + Ack No.

Stop-and-Wait ARQ

1. Normal operation

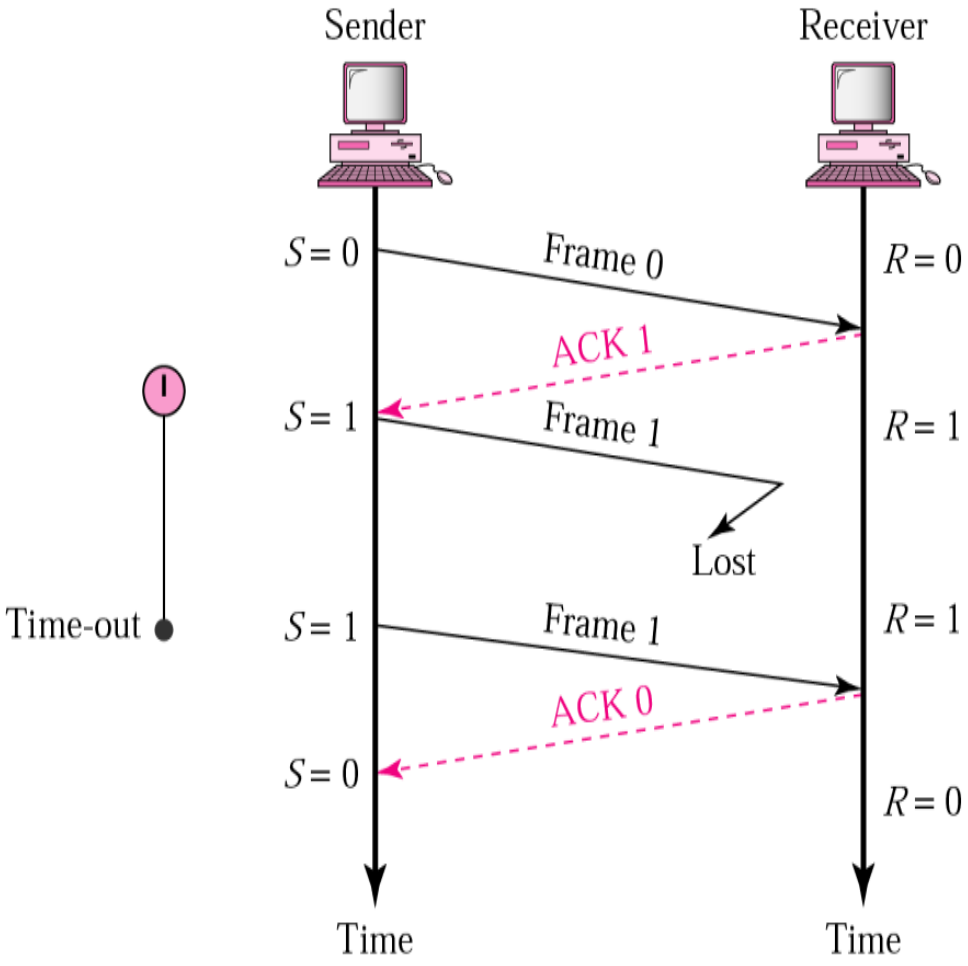
- *The sender will not send the next frame until it is sure that the current one is correctly receive*
- *sequence number is necessary to check for duplicated frames*



Stop and Wait ARQ

2. Lost or damaged frame

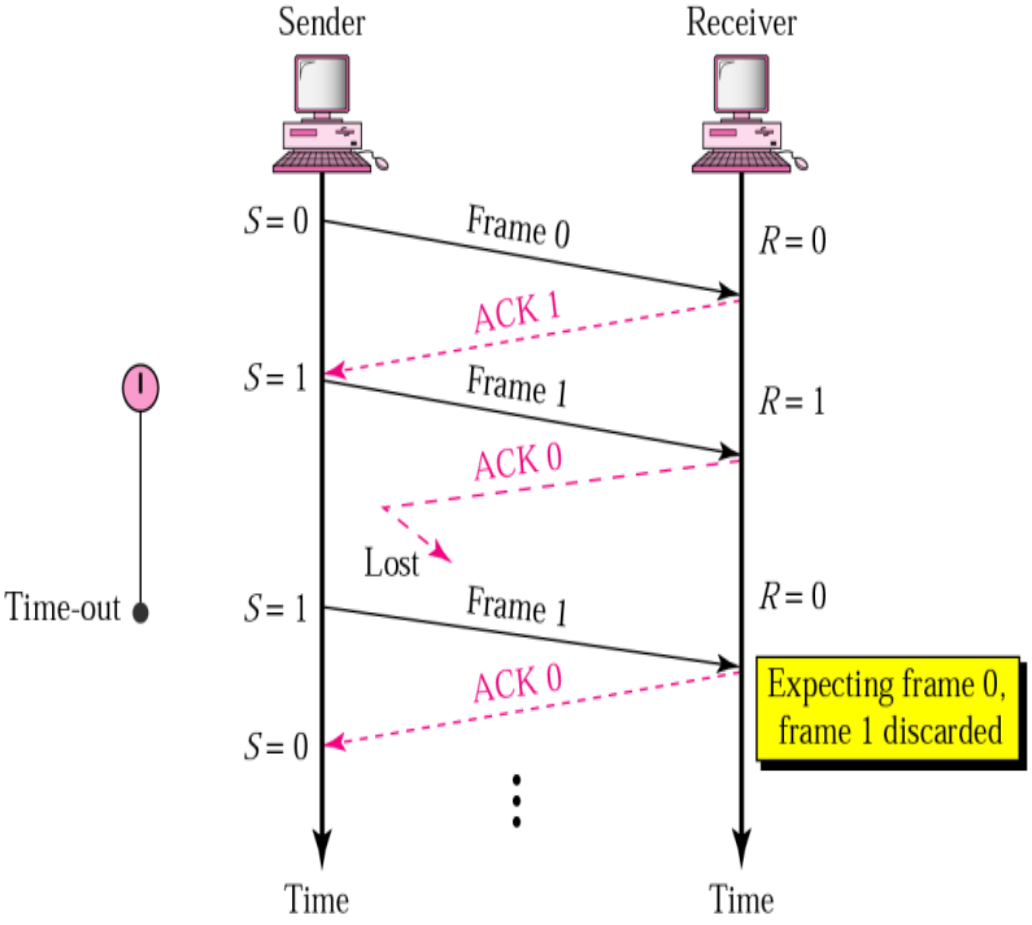
- *A damage or lost frame treated by the same manner by the receiver.*
- *No NACK when frame is corrupted / duplicate*



Stop-and-Wait ARQ

3. Lost ACK frame

- *Importance of frame numbering*



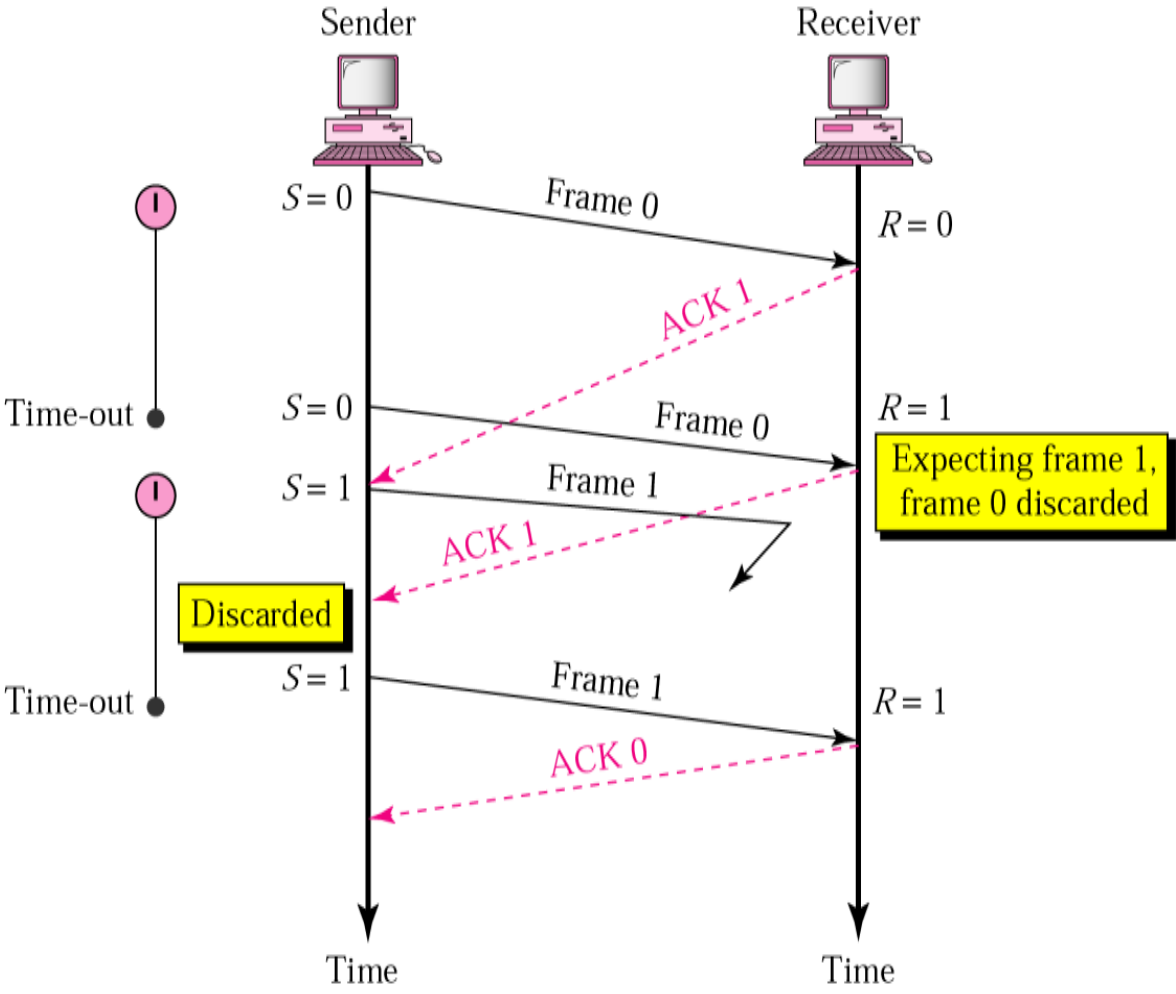
Note

In Stop and-Wait ARQ, numbering frames prevents the retaining of duplicate frames.

Stop-and-Wait ARQ

4. Delayed ACK and lost frame

- *Importance of frame numbering*

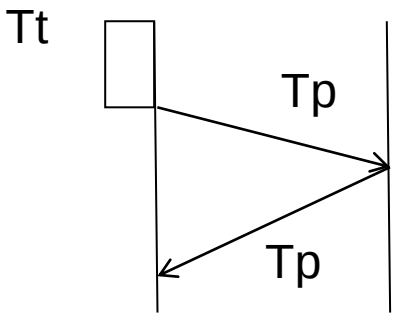


Note

Numbered acknowledgments are needed if an acknowledgment is delayed and the next frame is lost.

Stop &Wait Efficiency

- *Efficiency = useful time / total time*
  = Tt / (Tt + 2 Tp)
  = 1 / (1 + (2 Tp/ Tt)) = 1/ (1+2 a) *a= Tp/Tt*
  = 1/(1+2*(d/v)*(B/L))



- d = distance between source and receiver, v = velocity
- *B is channel Bandwidth; L is the packet length*

  Tp =d/v Tt= L/B
- *Efficiency* = 1/(1+2*(d/v)*(B/L))
- *Throughput : no. of bits sent / sec = L / (Tt + 2Tp)*
- *Throughput is called Effective Bandwidth or Bandwidth utilization*
- *Effective BW = (L *(B/B)/ (Tt+ 2Tp) = [Tt/Tt+2Tp] * BW*
  *= efficiency*B* *(Tt=L/B)*

Stop &Wait Efficiency Example

- *Example Tt = 1 msec, Tp = 1 msec , B= 6Mbps,*

  Then Efficiency is 33.3%

  Throughput = 2Mbps

  Tt = 2msec, Tp= 1 msec , Efficiency is 50%,

  Throughput =3Mbps

- *For efficiency >= 50% , then Tt >= 2Tp*
  - *Then (L/B) > = 2Tp then* ***L should be >= 2*Tp*BW***

S&W Efficiency

- In case of **S&W** protocol is only sending only one packet irrespective of the link bandwidth

Example: Tt= 1 msec, Tp = 1.5 msec

a= Tp /Tt =1.5

S&W Efficiency = 1/ (1+2a) = 25%

Retransmission packets in S&W ARQ protocol

- If P is the probability of loss packets
- If we want to transmit n packets then we will transmit the following:

  $n + n(p) + (n(p))(p) + (np\ p)\ p + \ldots.$

  $n(1 + p + p^2 + p^3 + p^4 + \ldots..)$

  $= n(1 / (1-p))$
- Ex: if n = 400 and p= 0.2 then we will transmit 500 packets

A Simplex Protocol for a Noisy Channel Discussion points

- *How can we argue that we only need 1 bit for sequence number?*
- *What anomaly could happen if the timeout at the sender is smaller (even temporarily when sending one packet only) than the round-trip delay?*
- *What is a alternative design for round-trip delay that is larger than sender timeout?*
- *Why does the sender re-send the last packet it sent when it gets duplicate ACK (I.e. what assumption is made)?*
- *Why don't we need a timeout mechanism at the receiver side? In other words, is it possible that the protocol deadlocks and the sender stops sending traffic?*
- *What is main new issue in this protocol?*