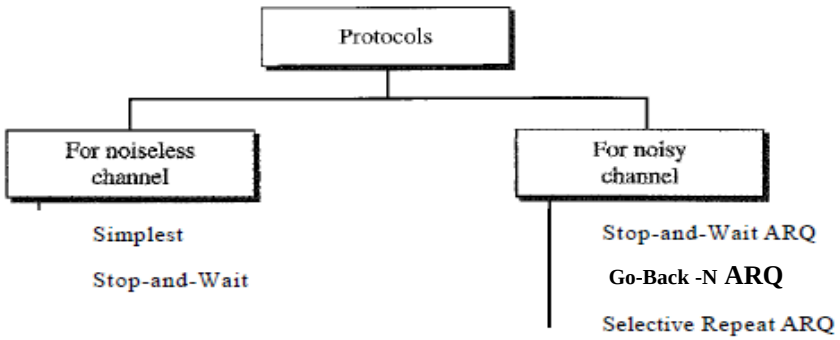
- **PROTOCOLS**



Protocols

- Noiseless channels
  .1. Simplest
  .2. Stop and Wait
- Noisy channels
  .1. Stop and Wait ARQ
  .2. Go Back-N ARQ
  .3. Selective Repeat ARQ
  .4. Piggy Backing.

NOISELESS CHANNELS

Let us first assume we have an ideal channel in which no frames are lost, duplicated, or corrupted. We introduce two protocols for this type of channel.
The first is a protocol that does not use flow control→ Simplex
The second is the one that use flow control →Stop-and-Wait
Neither has error control because we have assumed that the channel is a perfect noiseless channel.

Simplest Protocol

Our first protocol, which we call the Simplest Protocol is one that has no flow or error control. It is a unidirectional protocol in which data frames are travelling in only one direction-from the sender to receiver. We assume that the receiver can immediately handle any frame it receives with a processing time that is small enough to be negligible. The data link layer of the receiver immediately removes the header from the frame and hands the data packet to its network layer, which can also accept the packet immediately. In other words, the receiver can never be overwhelmed with incoming frames.

Design

There is no need for flow control in this scheme. The data link layer at the sender site gets data from its network layer, makes a frame out of the data, and sends it. The data link layer at the receiver site receives a frame from its physical layer, extracts data from the frame, and delivers the data to its network layer

We need to elaborate on the procedure used by both data link layers. The sender site cannot send a frame until its network layer has a data packet to send. The receiver site cannot deliver a data packet to its network layer until a frame arrives. If the protocol is implemented as a procedure, we need to introduce the idea of events in the protocol. The procedure at the sender site is constantly running; there is no action until there is a request from the network layer. The procedure at the receiver site is also constantly running, but there is no action until notification from the physical layer arrives. Both procedures are constantly running because they do not know when the corresponding events will occur.

Sender-site algorithm for the simplest protocol

```
//Send the frame
while (true)
{
WaitForEvent() //Sleep until an event occurs
if(Event(RequestToSend))
{
GetData()
MakeFrame()
SendFrame()
}
}
```

Receiver-site algorithm for the simplestprotocol

```
while(True)
{
WaitForEvent()
if(Event(ArrivalNotifications)
{
ReceiveFrame()
ExtractData()
DeliverData()
}
}
```

Stop-and-Wait Protocol

If data frames arrive at the receiver site faster than they can be processed, the frames must be stored until their use. Normally, the receiver does not have enough storage space, especially if it is receiving data from many sources. This may result in either the discarding of frames or denial of service. To prevent the receiver from becoming overwhelmed with frames, we somehow need to tell the sender to slow down. There must be feedback from the receiver to the sender. The protocol we discuss now is called the Stop-and-Wait Protocol because the sender sends one frame, stops until it receives confirmation from the receiver (okay to go ahead), and then sends the next frame. We still have unidirectional communication for data frames, but auxiliary ACK frames (simple tokens of acknowledgment) travel from the other direction. We add flow control to our previous protocol.

//Sender side algorithm for Stop-and-Wait Protocol

```
while(True)
canSend=True
{
WaitForEvent()    //Sleep until event occurs
if(Event(RequestToSend) and canSend)
{
GetData() //From upper layer
MakeFrame()
SendFrame() //Send data frame to receiver
canSend=False //cannot send until acknowledgement arrives
}
WaitForEvent() //sleep until event occurs
if(Event(ArrivalNotification)) //Acknowledgement has arrived
{
ReceiveFrame() //Receive ack frames
canSend=true //make flag true so that the next frame can be send
}
}
```

//Receiver side algorithm for Stop-and-Wait Protocol

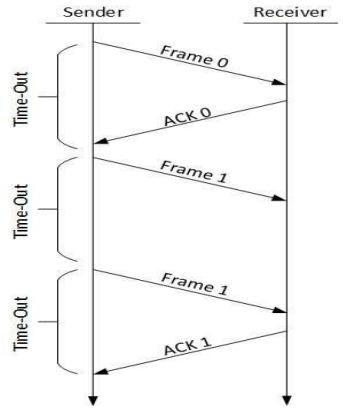
```
while(True)
WaitForEvent() //Sleep until event occurs
if(Event(ArrivalNotification )
{
ReceiveFrame()
ExtractData()
DeliverData()
SendFrame() //Ack frame
}
```

NOISY CHANNELS

Although the Stop-and-Wait Protocol gives us an idea of how to add flow control to its predecessor, noiseless channels are nonexistent. We can ignore the error (as we sometimes do), or we need to add error control to our protocols. We discuss three protocols in this section that use error control.

Stop-and-Wait Automatic Repeat Request

Our first protocol, called the Stop-and-Wait Automatic Repeat Request (Stop-andWaitARQ), adds a simple **error control mechanism** to the Stop-and-Wait Protocol. Let us see how this protocol detects and corrects errors. To detect and correct corrupted frames, we need to add redundancy bits to our data frame. When the frame arrives at the receiver site, it is checked and if it is corrupted, it is silently discarded. The detection of errors in this protocol is manifested by the silence of the receiver. Lost frames are more difficult to handle than corrupted ones. In our previous protocols, there was no way to identify a frame. The received frame could be the correct one, or a duplicate, or a frame out of order. The solution is to number the frames. When the receiver receives a data frame that is out of order, this means that frames were either lost or duplicated. The corrupted and lost frames need to be resent in this protocol. If the receiver does not respond when there is an error, how can the sender know which frame to resend? To remedy this problem, the sender keeps a copy of the sent frame. At the same time, it starts a timer. If the timer expires and there is no ACK for the sent frame, the frame is resent, the copy is held, and the timer is restarted. Since the protocol uses the stop-and-wait mechanism, there is only one specific frame that needs an ACK even though several copies of the same frame can be in the network.



Error correction in Stop-and-Wait ARQ is done by keeping a copy of the sent frame and retransmitting of the frame when the timer expires.

Since an ACK frame can also be corrupted and lost, it too needs redundancy bits and a sequence number. The ACK frame for this protocol has a sequence number field. In this protocol, the sender simply discards a corrupted ACK frame or ignores an out-of-order one.

Sequence Numbers

As we discussed, the protocol specifies that frames need to be numbered. This is done by using sequence numbers. A field is added to the data frame to hold the sequence number of that frame. One important consideration is the range of the sequence numbers. Since we want to minimize the frame size, we look for the smallest range that provides unambiguous communication. In this case sequence numbers can be 0 and 1 .
0 for first frame and 1 for second frame again 0 is used as sequence number for the third frame and 1 for fourth and so on . In this case we need only 1 bit to store the sequence number.

To show this, assume that the sender has sent the frame numbered 0. Three things can happen.
1. The frame arrives safe and sound at the receiver site; the receiver sends an acknowledgment. The acknowledgment arrives at the sender site, causing the sender to send the next frame numbered 1.
2. The frame arrives safe and sound at the receiver site; the receiver sends an acknowledgment, but the acknowledgment is corrupted or lost. The sender resends the frame (numbered 0) after the time-out. Note that the frame here is a duplicate. The receiver can recognize this fact because it expects frame I but frame 0 was received.
3. The frame is corrupted or never arrives at the receiver site; the sender resends the frame (numbered 0) after the time-out.

Acknowledgment Numbers
The acknowledgment numbers always announce the sequence number of the next frame expected by the receiver. For example, if frame 0 has arrived safe and sound, the receiver sends an ACK frame with acknowledgment 1 (meaning frame 1 is expected next). If frame 1 has arrived safe and sound, the receiver sends an ACK frame with acknowledgment 0 (meaning frame 0 is expected).

The following transition may occur in Stop-and-Wait ARQ:

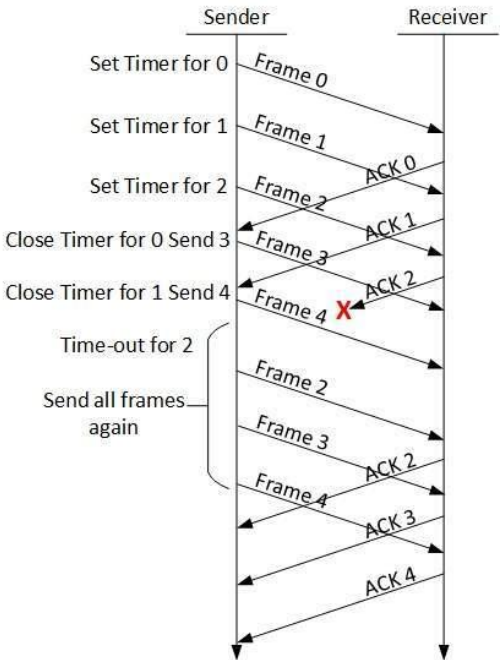
- The sender maintains a timeout counter.
- When a frame is sent, the sender starts the timeout counter.
- If acknowledgement of frame comes in time, the sender transmits the next frame in queue.
- If acknowledgement does not come in time, the sender assumes that either the frame or its acknowledgement is lost in transit. Sender retransmits the frame and starts the timeout counter.
- If a negative acknowledgement is received, the sender retransmits the frame.

Go-Back-N ARQ

To improve the efficiency of transmission multiple frames must be in transition while waiting for acknowledgment. In Go-Back-N Automatic Repeat Request we can send several frames before receiving acknowledgments; we keep a copy of these frames until the acknowledgments arrive.

Sequence Numbers
Frames from a sending station are numbered sequentially. However, because we need to include the sequence number of each frame in the header, we need to set a limit. If the header of the frame allows m bits for the sequence number, the sequence numbers range from 0 to 2^m - 1. For example, if m is 4, the only sequence numbers are 0 through 15 inclusive. However, we can repeat the sequence. So the sequence numbers are
0, 1,2,3,4,5,6, 7,8,9, 10, 11, 12, 13, 14, 15,0, 1,2,3,4,5,6,7,8,9,10, 11, ...

In Go-Back-N ARQ method, both sender and receiver maintain a window(window is an abstract concept to denote the range of sequence numbers)



The sending-window size enables the sender to send multiple frames without receiving the acknowledgement of the previous ones. The receiving-window enables the receiver to receive multiple frames and acknowledge them. The receiver keeps track of incoming frame's sequence number.

When the sender sends all the frames in window, it checks up to what sequence number it has received positive acknowledgement. If all frames are positively acknowledged, the sender sends next set of frames. If sender finds that it has received NACK or has not receive any ACK for a particular frame, it retransmits all the frames after which it does not receive any positive ACK.

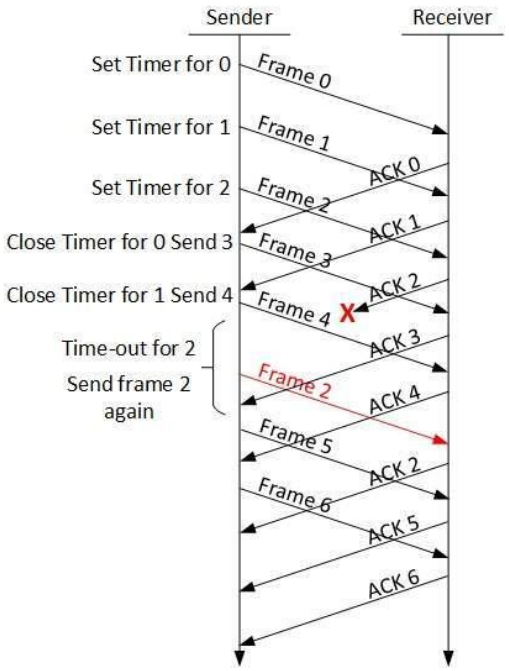
Selective Repeat ARQ

Go-Back-N ARQ is very inefficient for a noisy link. In a noisy link a frame has a higher probability of damage, which means the resending of multiple frames. This resending uses up the bandwidth and slows down the transmission. For noisy links, there is another mechanism that does not resend N frames when just one frame is damaged; only the damaged frame is resent. This mechanism is called Selective RepeatARQ. It is more efficient for noisy links, but the processing at the receiver is more complex.

In Go-back-N ARQ, it is assumed that the receiver does not have any buffer space for its window size and has to process each frame as it comes. This enforces the sender to retransmit all the frames which are not acknowledged.

In Selective-Repeat ARQ, the receiver while keeping track of sequence numbers, buffers the frames in memory and sends NACK (Negative Acknowledgment)for only frame which is missing or damaged.

The sender in this case, sends only packet for which NACK is received.



Piggybacking

The three protocols we discussed in this section are all unidirectional: data frames flow in only one direction although control information such as ACK and NAK frames can travel in the other direction. In real life, data frames are normally flowing in both directions: from node A to node B and from node B to node A. This means that the control information also needs to flow in both directions. A technique called **piggybacking** is used to improve the efficiency of the bidirectional protocols . Here, sending of acknowledgment is delayed until the next data frame is available for transmission. The acknowledgment is then hooked onto the outgoing data frame. The data frame consists of an *ack* field. The size of the *ack* field is only a few bits, while an acknowledgment frame comprises of several bytes. Thus, a substantial gain is obtained in reducing bandwidth requirement.

Suppose that there are two communication stations X and Y. The data frames transmitted have an acknowledgment field, *ack* field that is of a few bits length. Also there are frames for sending acknowledgments, ACK frames. The purpose is to minimize the use of ACK frames by piggybacking the acknowledgement in the data frames.

Consider the scenarios

- If station X has both data and acknowledgment to send, it sends a data frame with the *ack* field containing the sequence number of the frame to be acknowledged.
- If station X has only an acknowledgment to send, it waits for a finite period of time to see whether a data frame is available to be sent. If a data frame becomes available, then it piggybacks the acknowledgment with it. Otherwise, it sends a seperate ACK frame.
- If station X has only a data frame to send , station X may send the data frame with the *ack* field containing a bit combination denoting no acknowledgment.