**Chapter 11: Data Link Control**

Table of Contents

[11.1 Framing 2](#_Toc63258129)

[Character-Oriented Protocols 2](#_Toc63258130)

[Bit-Oriented Protocols 3](#_Toc63258131)

[11.2 Flow and Error Control 5](#_Toc63258132)

[11.3 Protocols 6](#_Toc63258133)

[11.4 Noiseless Channels 7](#_Toc63258134)

[Simplest Protocol 7](#_Toc63258135)

[Stop-and-Wait Protocol 9](#_Toc63258136)

[11.5 Noisy Channels 12](#_Toc63258137)

[Stop-and-Wait ARQ 12](#_Toc63258138)

[Go-Back-N ARQ 18](#_Toc63258139)

[Selective Repeat ARQ 21](#_Toc63258140)

## 11.1 Framing

The data-link layer takes the datagram from the transport layer and adds a header and a trailer. The result is called a frame. The header holds information like the physical addresses of the sender and receiver as well as some other information obtained from the ARP protocol. We will be discussing details about the format of the header in a later topic. The trailer holds the different error-detection and error-correction codes we have studied.

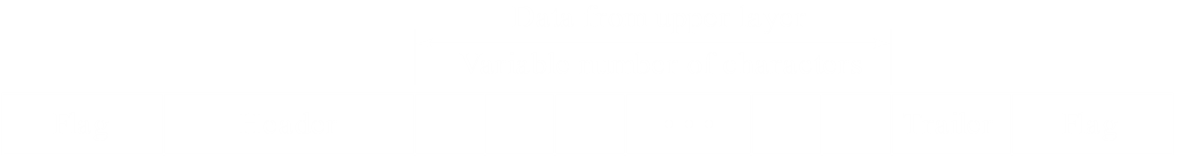
Frames can be of a fixed size or of a variable size.

Typically, fixed-size framing is used. Since the size of each frame is fixed, there is no need to use any delimiters. Say each frame is bytes. The receiver will simply take the data in groups of bytes each.

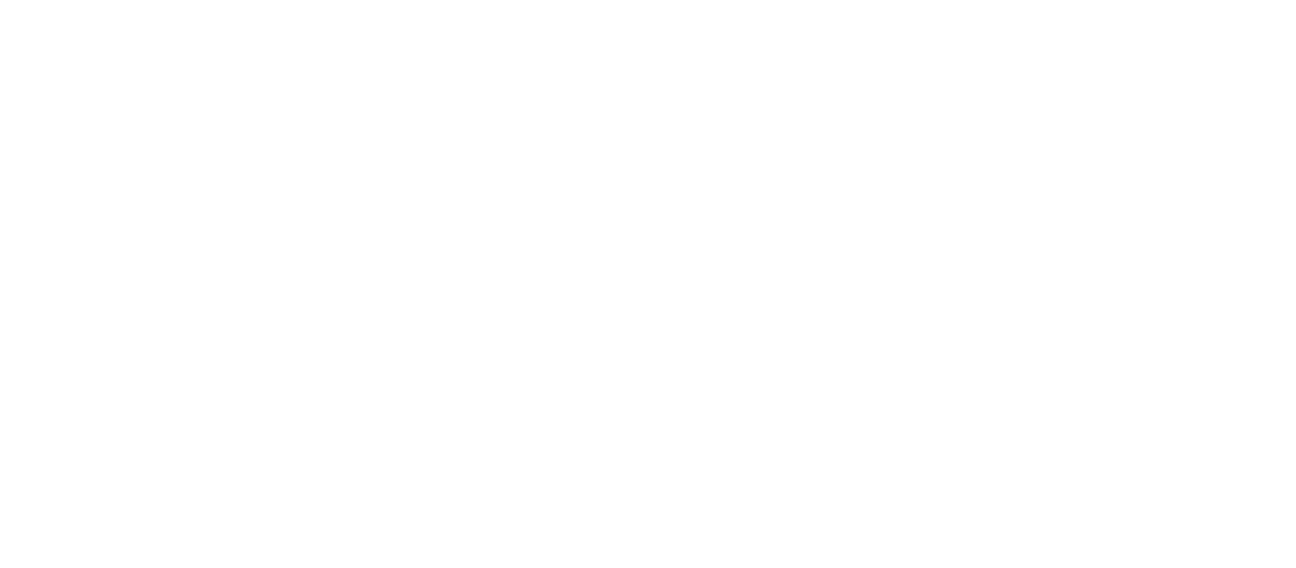
If instead variable-size framing is used, we also need to use delimiters. This is where character-oriented protocols and bit-oriented protocols come in.

### Character-Oriented Protocols

Character-oriented protocols are used when the data being carried are -bit characters from a coding system. Before and after the header and trailer respectively, an -bit flag is placed. These flags are used as delimiters to separate frames.



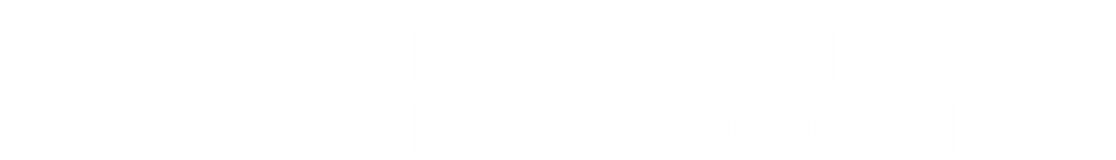
Of course, it is possible that the pattern used for the flags is also a part of the actual data. As such, the receiver would mistake that pattern in the data for a flag which would cause errors. To fix this, byte-stuffing is used. Essentially, an extra byte is added next to that pattern in the data. This byte, called the escape character, has a predefined pattern. Again, if the pattern for the escape character is also present in the actual data, this pattern must be escaped too.



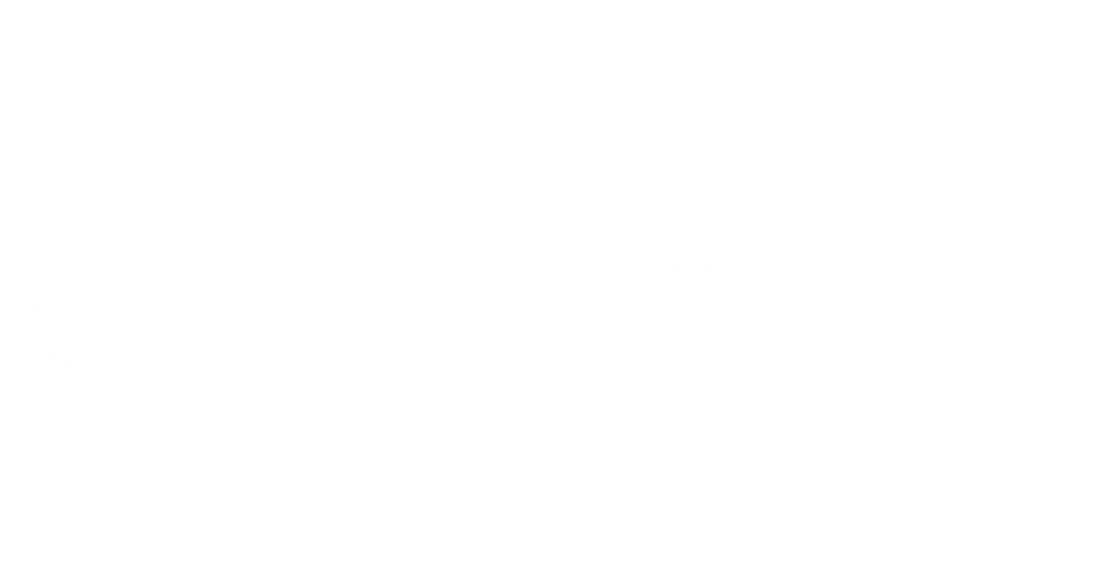
On the receiver’s end, the escape characters as simply removed.

### Bit-Oriented Protocols

Bit-oriented protocols are used when the data is a sequence of bits that can only be interpreted as text/graphics/audio/video/etc. by the upper layers. Again, we place a flag before and after the header and trailer respectively. In most protocols, the flag uses the special -bit pattern .



Again, the actual data may have the pattern used by the flags, which means we need bit stuffing. In bit stuffing, a single bit is stuffed instead of a byte. Whenever the pattern (with five s instead of six) is found, a is stuffed after it. This stuffed bit is removed at the receiver’s end.



## 11.2 Flow and Error Control

Two concepts that are extremely important in the data-link layer are flow control and error control.

Consider that the sender is sending data very fast, faster than the receiver can receive the data. At one point, the buffer available at the receiver will overflow, so any further incoming data will be lost.

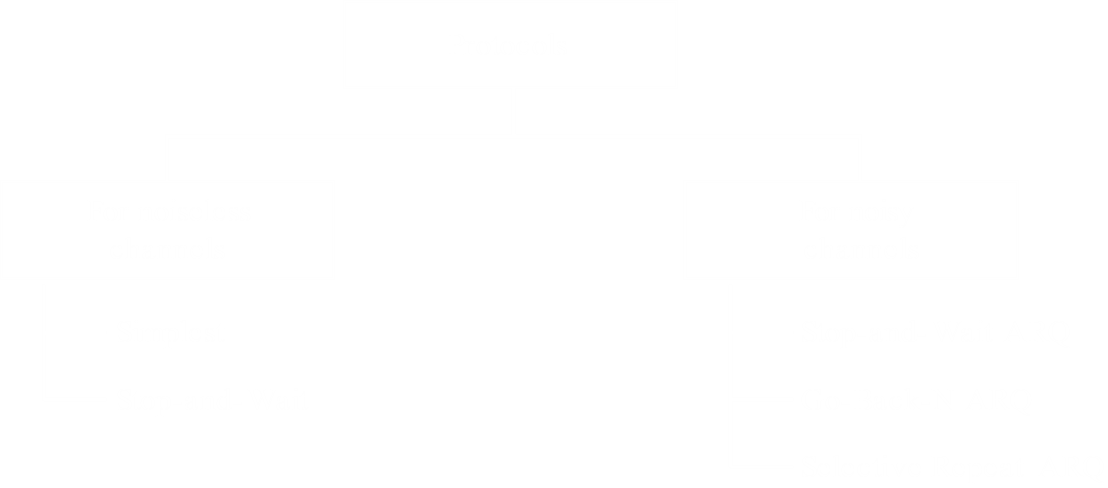
To avoid this, there are a set of procedures that limit the amount of data that the sender can send at once before waiting for acknowledgement that the data has been received. This set of procedures is called flow control.

Error control involves procedures for both error-detection and error-correction. It is based on the concept of automatic repeat requests, which is the retransmission of data.

## 11.3 Protocols

Now we will look at how the data-link layer achieves the different things we just saw that it is meant to do. This is where protocols come in. The protocols are implemented in software using one of the common programming languages. To keep things simple, we will be using pseudocode in place of the actually programming code in our discussion about protocols.

There are five different protocols we will study, divided into two categories.



The first two protocols are Simplest, and Stop-and-Wait. These deal with noiseless channels. Noiseless channels of course, do not exist in the real world, but these two protocols were still the starting point for the development of the other protocols.

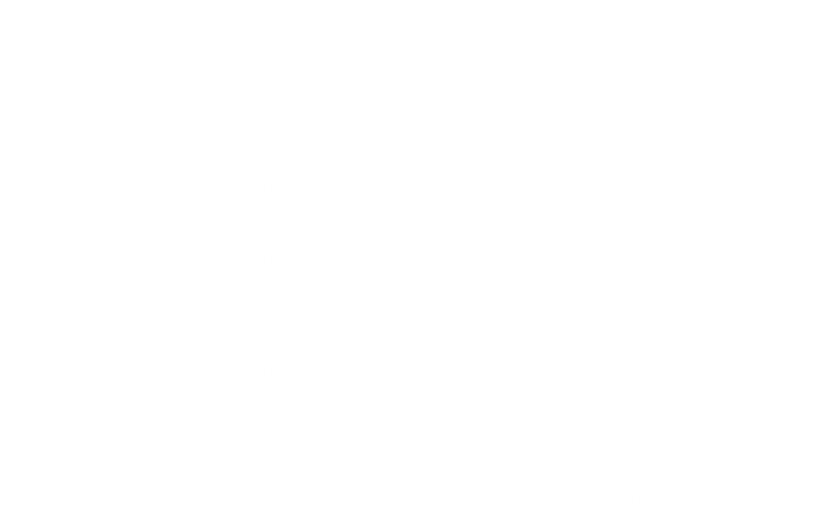
The three protocols that deal with noisy channels are Stop-and-Wait ARQ, Go-Back-N ARQ and Selective Repeat ARQ.

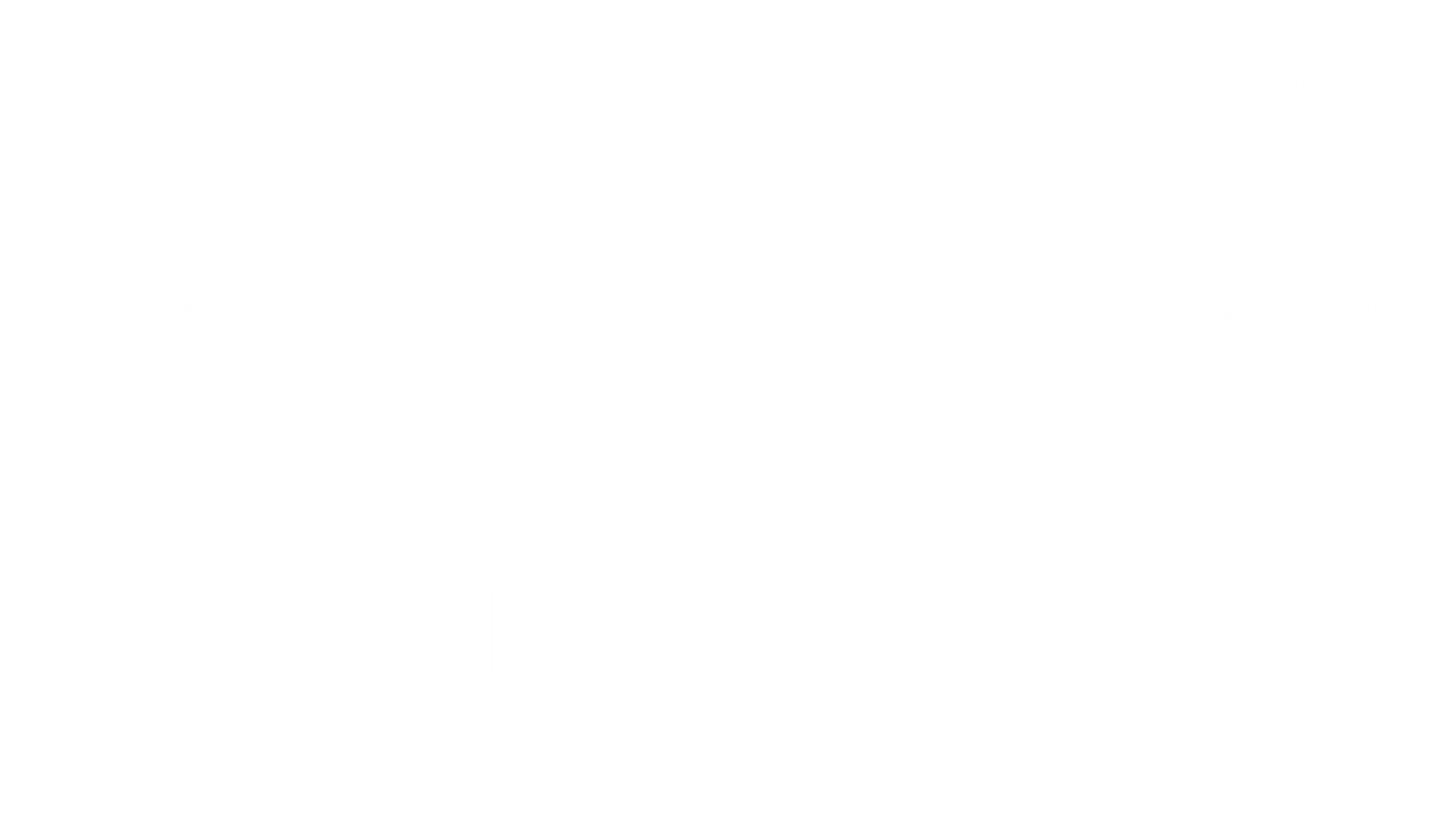
## 11.4 Noiseless Channels

Let’s assume we have an ideal noiseless channel. This means no frames are lost, duplicated or corrupted. Based on this assumption, we can have two different protocols.

### Simplest Protocol

This is literally the simplest protocol, thus the name. It has no flow control or error control mechanisms. This protocol is unidirectional, since frames only travel from the sender to the receiver. We assume that the receiver can immediately handle any frame that it receives with a processing time that is negligible. In other words, it is assumed the problem of the buffer at the receiver’s end overflowing does not occur, which removes the need for any flow control.





#### Pseudocode

Let’s take a look at what the pseudocode for the protocol might look like at the sender and receiver’s end.

On the sender’s side, we wait for an event to occur. This event is the request from the network layer to send some data. We take the data, make a frame and send it.

while(true) // Repeat forever  
{  
 WaitForEvent(); // Sleep until an event occurs  
 if(Event(RequestToSend)) // There is a packet to send  
 {  
 GetData();  
 MakeFrame();  
 SendFrame(); // Send the frame  
 }  
}

PSEUDOCODE

On the receiver’s end, a similar process occurs. The event this time is the notification from the physical layer that a frame has arrived. We take the frame, extract the data and deliver the data to the network layer.

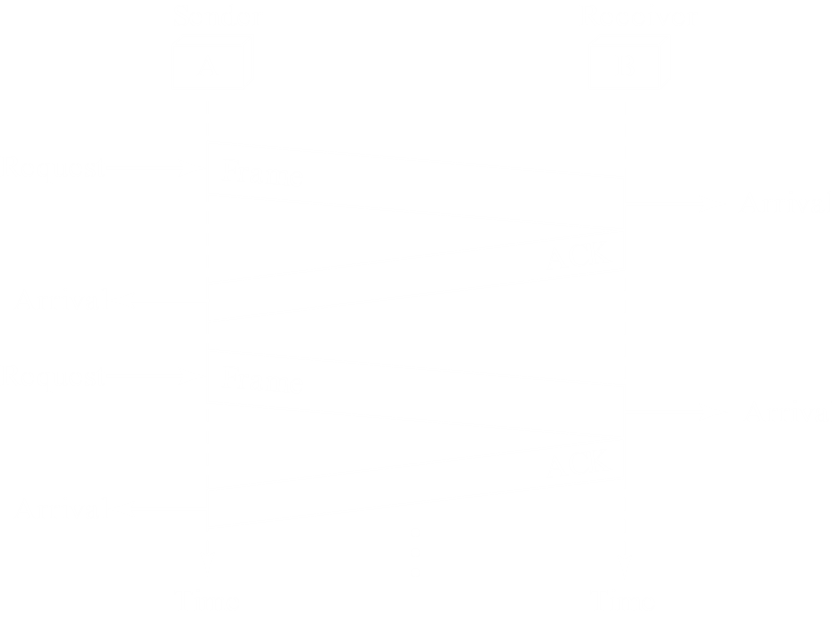
while(true) // Repeat forever  
{  
 WaitForEvent(); // Sleep until an event occurs  
 if(Event(ArrivalNotification)) // Data frame arrived  
 {  
 ReceiveFrame();  
 ExtractData();  
 DeliverData(); // Deliver data to network layer  
 }  
}

PSEUDOCODE

Notice that both the algorithms run forever. They keep looking for new events till the end of time.

### Stop-and-Wait Protocol

The stop-and-wait protocol takes into account that maybe the receiver’s buffer does get overflooded if we just keep sending data. Thus, flow control is added. The sender sends a frame, waits for an acknowledgement from the receiver, and then sends the next frame. We still have unidirectional communication for the data frames themselves, but we are also now receiving ACK (acknowledgement) frames from the other direction.





#### Pseudocode

In terms of pseudocode, the sender side’s algorithm gets a little bigger. Firstly, we now need to track two events, requests from the network layer and notifications from the physical layer for when we get ACKs. Secondly, we need to keep track of whether or not we got an acknowledgement. If we did not, we cannot send another frame, even if the network layer is making a request.

canSend = true; // Allow the first frame to go  
while(true) // Repeat forever  
{  
 WaitForEvent(); // Sleep until an event occurs  
 if(Event(RequestToSend) AND canSend)  
 {  
 GetData();  
 MakeFrame();  
 SendFrame(); // Send the data frame  
 canSend = false; // Cannot send until ACK arrives  
 }  
 WaitForEvent(); // Sleep until an event occurs  
 if(Event(ArrivalNotification)) // An ACK has arrived  
 {  
 ReceiveFrame(); // Receive the ACK frame  
 canSend = true;  
 }  
}

PSEUDOCODE

On the receiver’s side, the only change is that we need to send an ACK frame once we receive a frame.

while(true) // Repeat forever  
{  
 WaitForEvent(); // Sleep until an event occurs  
 if(Event(ArrivalNotification)) // Data frame arrives  
 {  
 ReceiveFrame();  
 ExtractData();  
 Deliver(data); // Deliver data to network layer  
 SendFrame(); // Send an ACK frame  
 }  
}

PSEUDOCODE

## 11.5 Noisy Channels

As discussed earlier, we don’t actually have noiseless channels in real life. So, we need protocols that deal with noisy channels. This is where the next three protocols come in. All three of them use error-control.

You’ll notice that all three have the term ARQ in them. ARQ stands for Automatic Repeat Request. When we send a frame, that frame is marked as an outstanding frame, meaning we have not yet received an ACK for it. A certain amount of time is required to send the frame and get an acknowledgement from the receiver. This time is called the round-trip time (RTT). The concept of ARQ is, if the sender does not get an acknowledgement for quite some time after the RTT (the exact time is predefined), it will automatically resend the packet.

### Stop-and-Wait ARQ

We know that Stop-and-Wait sends frames one at a time and waits for an acknowledgement before sending another, so adding error-control to this is relatively easy. A copy of the frame being sent is kept, and if the timer expires before an ACK is received for that frame, the frame is resent.

#### Frame Numbering

The thing to notice here is that the ACK must be for the frame currently being sent. What if, for some reason, we send one frame but the ACK is delayed so we send it again. Immediately after sending it the second time, we finally get the ACK for our first attempt and thus send out the second frame. However, we will soon get an ACK for the second attempt we made with the first frame. We need to be able to identify that this ACK is not for the second frame that we just sent. To do this, we use frame numbers.

In the Stop-and-Wait ARQ protocol in particular, we just need two numbers, and , since we are sending frames one by one. If the ACK we get tells us that frame has been received, we can send the next frame, which will be given the number . Once we get an ACK telling us that frame has been received, we can again number the next frame and send it.

This numbering scheme is based on modulo-2 arithmetic. The sequence of numbers is , where is the number of frames being sent at a time.

Another important fact to remember is that the ACK always tells us the number of the next frame that needs to be sent, not the number of the frame received. In Stop-and-Wait ARQ, if the ACK has the number , it means that the previous frame that had the number has been received and the next frame with the number should be sent. In other protocols, where the frame numbers can be higher, if the ACK has say the number , it means frames to have all been received. Note that the actual sequence of ACKs received is irrelevant here. If, the receiver sends an ACK with the number , it means it got all the frames prior to that, even if the sender did not get the ACKs for those frames. It is possible that the ACKs got lost.



Notice that at the top of the diagram we have images with arrows keeping track of which frame needs to be sent next. This uses the concept of a window, which will be explained in further detail soon. For now, just keep in mind that this is like having a window of size .

#### Pseudocode

Now for the pseudocode. The sender has three separate events it needs to deal with. Firstly, requests from the network layer are only entertained if there is nothing preventing us from sending another frame. Once we send a frame, we increment the frame number () and set the Boolean variable to false. We also start a timer.

If we receive an ACK, we must check that the ACK matches with the value of (since was incremented). If it does and the data that was sent was not corrupted, we stop the timer and throw away the copy of the frame we had stored. We also set the Boolean variable to true to indicate that we can send the next frame now.

If instead we do not receive an ACK for the correct frame that also tells us that the data was not corrupted, the timer will eventually run out. Once this happens, we resend frame and start the timer again.

Sn = 0; // Frame 0 sent first  
canSend = true; // Allow first request to go  
while(true) // Repeat forever  
{  
 WaitForEvent(); // Sleep until an event occurs  
 if(Event(RequestToSend) AND canSend)  
 {  
 GetData();  
 MakeFrame(Sn); // The seqNo is Sn  
 StoreFrame(Sn); // Keep copy  
 SendFrame(Sn);  
 StartTimer();  
 Sn = Sn + 1;  
 canSend = false;  
 }  
 WaitForEvent(); // Sleep  
 if(Event(ArrivalNotification)) // An ACK has arrived  
 {  
 ReceiveFrame(ackNo); // Receive the ACK frame  
 if (!corrupted(frame) AND ackNo == Sn) // Valid ACK  
 {  
 StopTimer();  
 PurgeFrame(Sn - 1); // Copy is not needed  
 canSend = true;  
 }  
 }  
 if(Event(TimeOut)) // The timer expired  
 {  
 StartTimer();  
 ResendFrame(Sn - 1); // Resend a copy check  
 }  
}

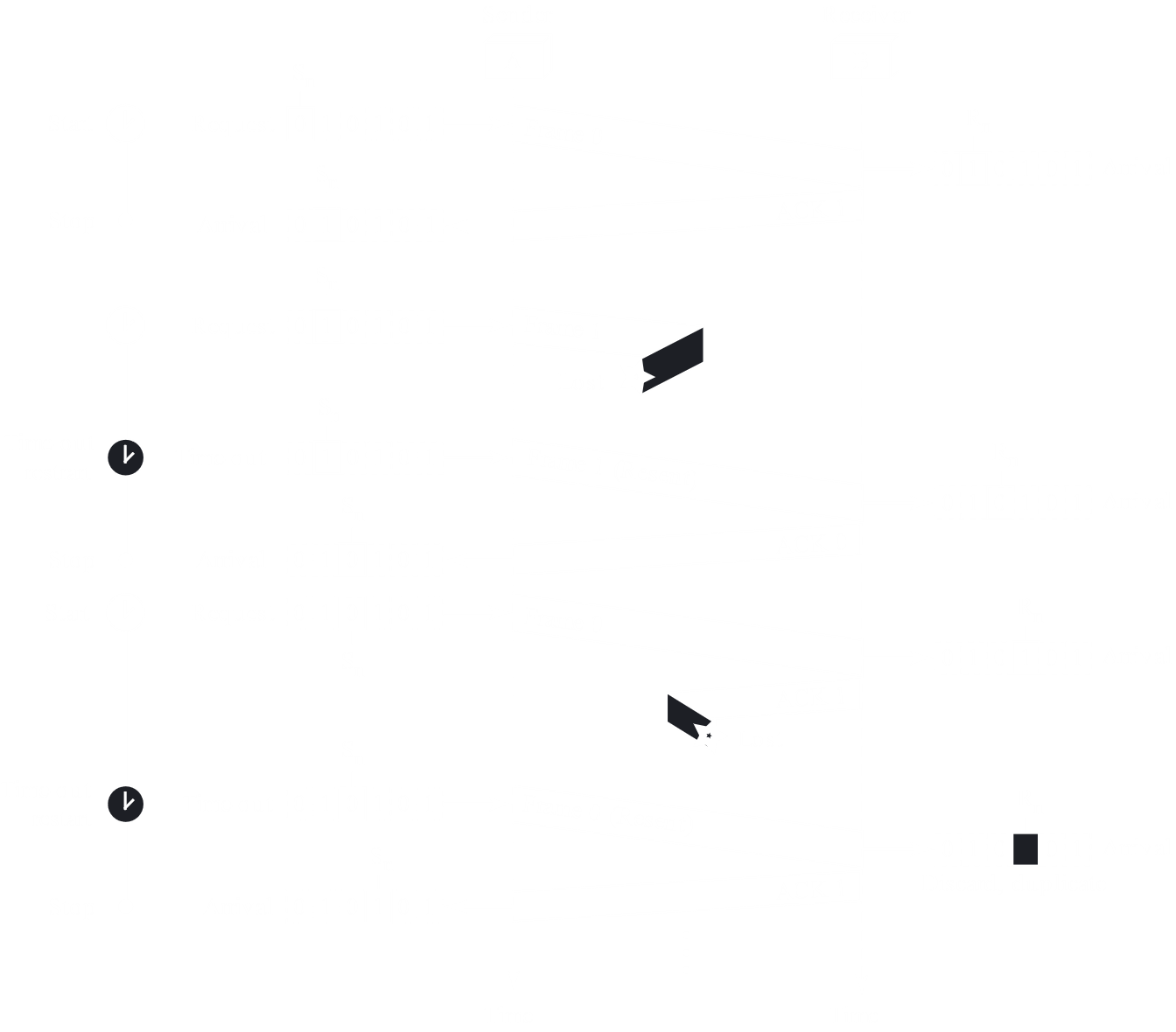
PSEUDOCODE

On the receiver’s end, we keep track of which frame we are expecting using another variable (). Once we get a frame, we check if it is corrupted. If it is, we sit tight. The sender will send the packet again. If it is uncorrupted, and this is the frame number we were expecting, we accept it, increment the value of and send an ACK with this value to tell the receiver to send the next frame.

Rn = 0; // Frame 0 expected to arrive first  
while(true)  
{  
 WaitForEvent(); // Sleep until an event occurs  
 if(Event(ArrivalNotification)) // Data frame arrives  
 {  
 ReceiveFrame();  
 if(corrupted(frame));  
 sleep();  
 if(seqNo == Rn) // Valid data frame  
 {  
 ExtractData();  
 DeliverData(); // Deliver data  
 Rn = Rn + 1;  
 }  
 sendFrame(Rn); // Send an ACK  
 }  
}

PSEUDOCODE

Example



Frame is sent first, and ACK is sent back. Frame is lost though, so after the timer runs out, it is resent. ACK is sent in return. Frame is sent successfully, but ACK is lost. Since the sender does not get the ACK, it sends Frame again after the timer runs out. Frame is not what the receiver was looking for, so it does not actually do anything with it. However, it does still send ACK to tell the sender that he needs to send Frame . This last bit is an inefficiency, since Frame did not need to be sent again, but there is nothing we can do about this. There is no way for the sender to know that Frame was delivered successfully, since the acknowledgement itself was lost. Thus, we need to live with this inefficiency.

### Go-Back-N ARQ

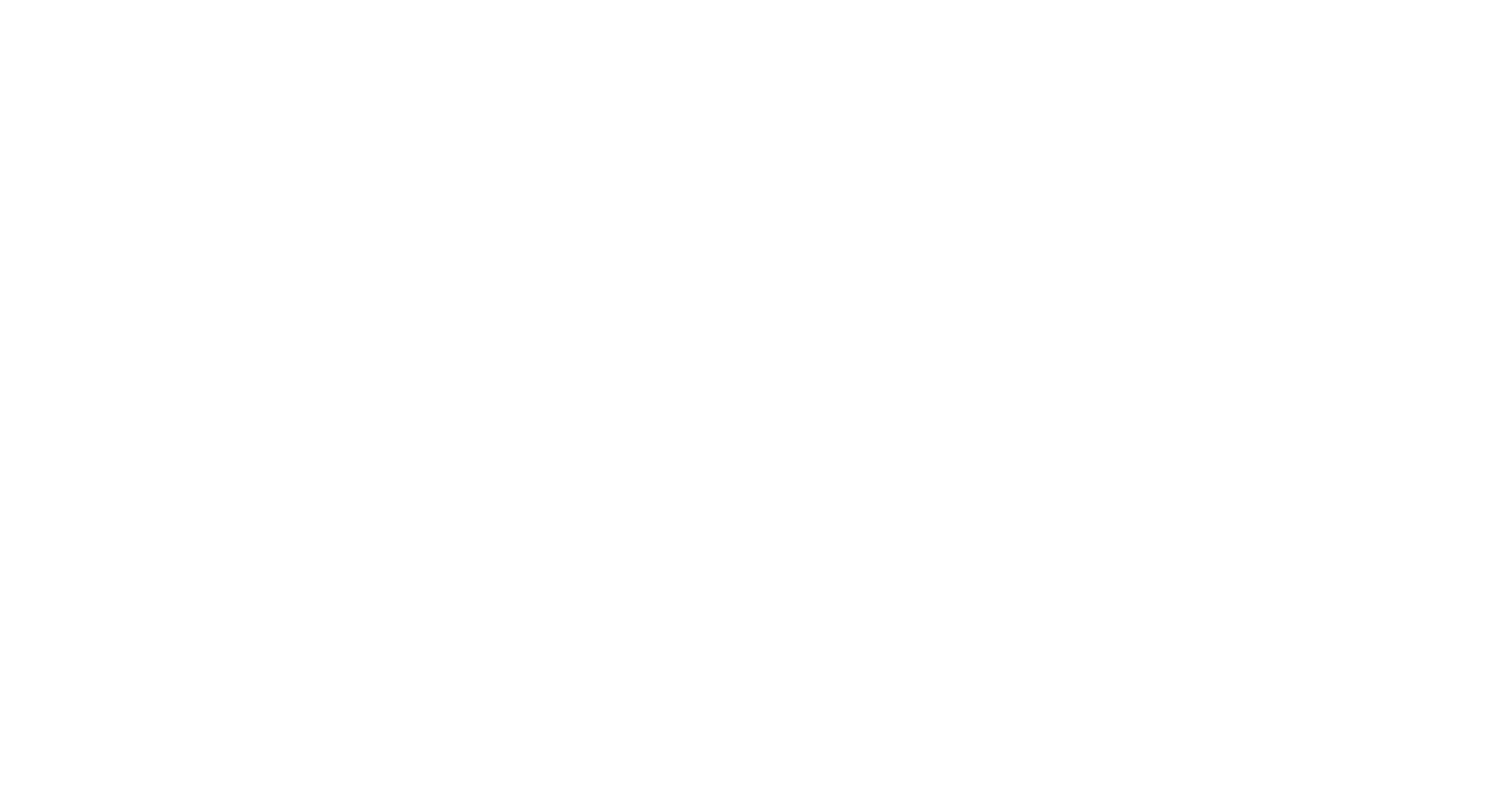
Consider that the distance between the sender and receiver is huge and the connection is slow. Thus, the bandwidth-delay product will be large. If we use the Stop-and-Wait ARQ protocol in this scenario, it will be extremely inefficient since we have to repeatedly send data frame by frame across this huge, slow connection.

Instead of sending data frame by frame, we could send several frames at once, so as to make the process more efficient. For each frame, we keep a copy until we receive an ACK for that particular frame. This is the concept of Go-Back-N ARQ.

The process of doing more tasks before the previous ones have finished is called pipelining. Pipelining was not available in Stop-and-Wait ARQ, but it is used in Go-Back-N ARQ and Selective Repeat ARQ.

#### Sliding Window Theory

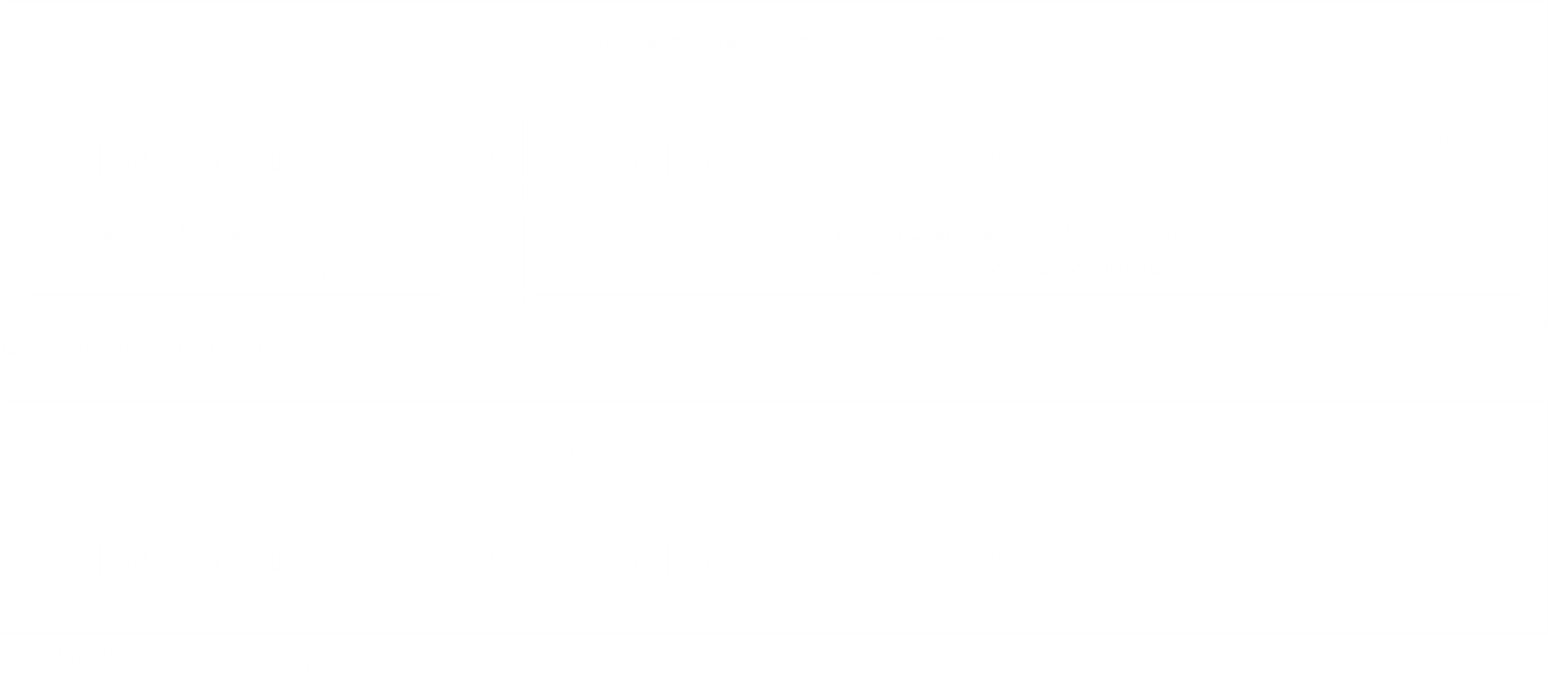
The concept of a sliding window is very important in the Go-Back-N ARQ protocol. Consider the image below:



The frames on the left of the orange box have already been received by the receiver and ACKs have been sent. Inside the orange box are the frames that have been sent but no ACKs have been received for them. marks the marks the left boundary of the window, and marks the next frame that will be sent once it is ready to be sent.

When ACKs are received, moves forwards. This process is called the closing of the window. When more frames are sent, moves forwards, increasing the window size. This is called opening of the window. There is another concept where moves backward. That is called shrinking of the window. However, for our purposes and applications, shrinking of the window is strictly forbidden. It simply cannot happen.

On the receiver’s end, the window is just one frame large. When the required frame is received, the window slides forward. If some other frame is received, it is discarded, even if the frame will be needed in the future.

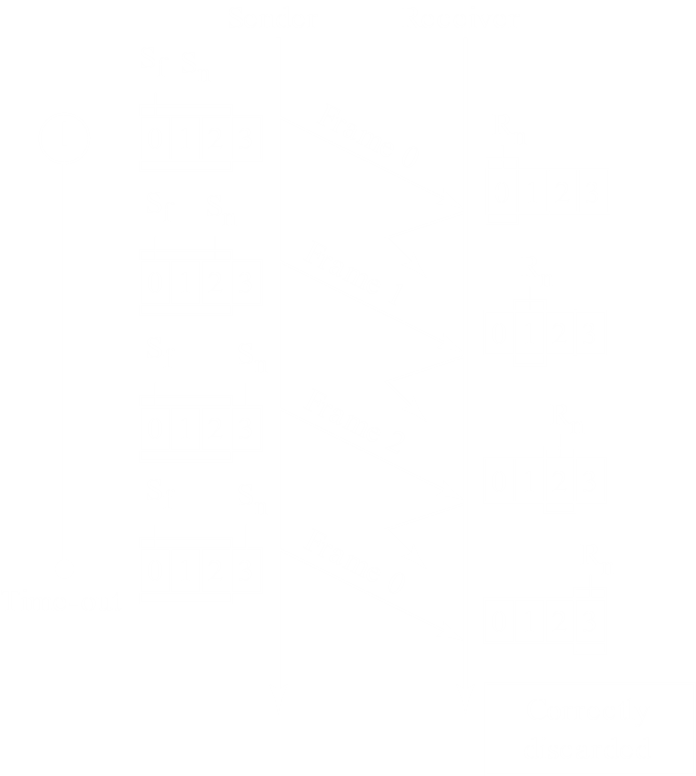


This behaviour on the receiver’s end leads to a problem. Say frame is lost for some reason, but frame to arrive properly. The receiver will discard all of them. Since frame was lost, no ACK was sent to the sender. Eventually the sender’s timer for frame will run out. At this point, the sender knows something went wrong and frame needs to be resent. It also knows that the fact that frame was not delivered properly means every frame that was sent after it must have been discarded and must also be resent. Thus, every frame in the window has to be resent. This is why the protocol is called Go-Back-N ARQ.

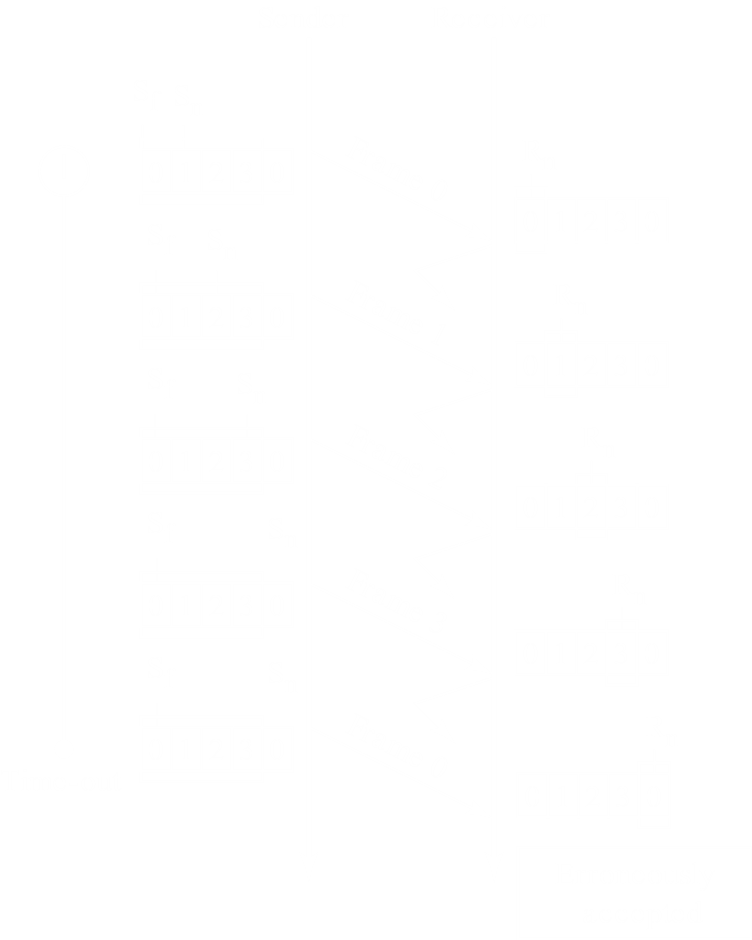
#### Window Size

An important note is that the maximum number of frames in the window cannot exceed , where is the number of bits required to represent the maximum frame number. Here, the maximum frame number is , so .

Consider why that is. Say we have four frames, numbered from to , so the maximum window size if . Thus, the first three frames, from to , are in the window. Say all three of these frames are successfully received, but all three ACKs are lost for some reason. This will prompt the sender to resend the three frames. However, the receiver is expecting frame now, so it will discard all three frames from frame to frame .

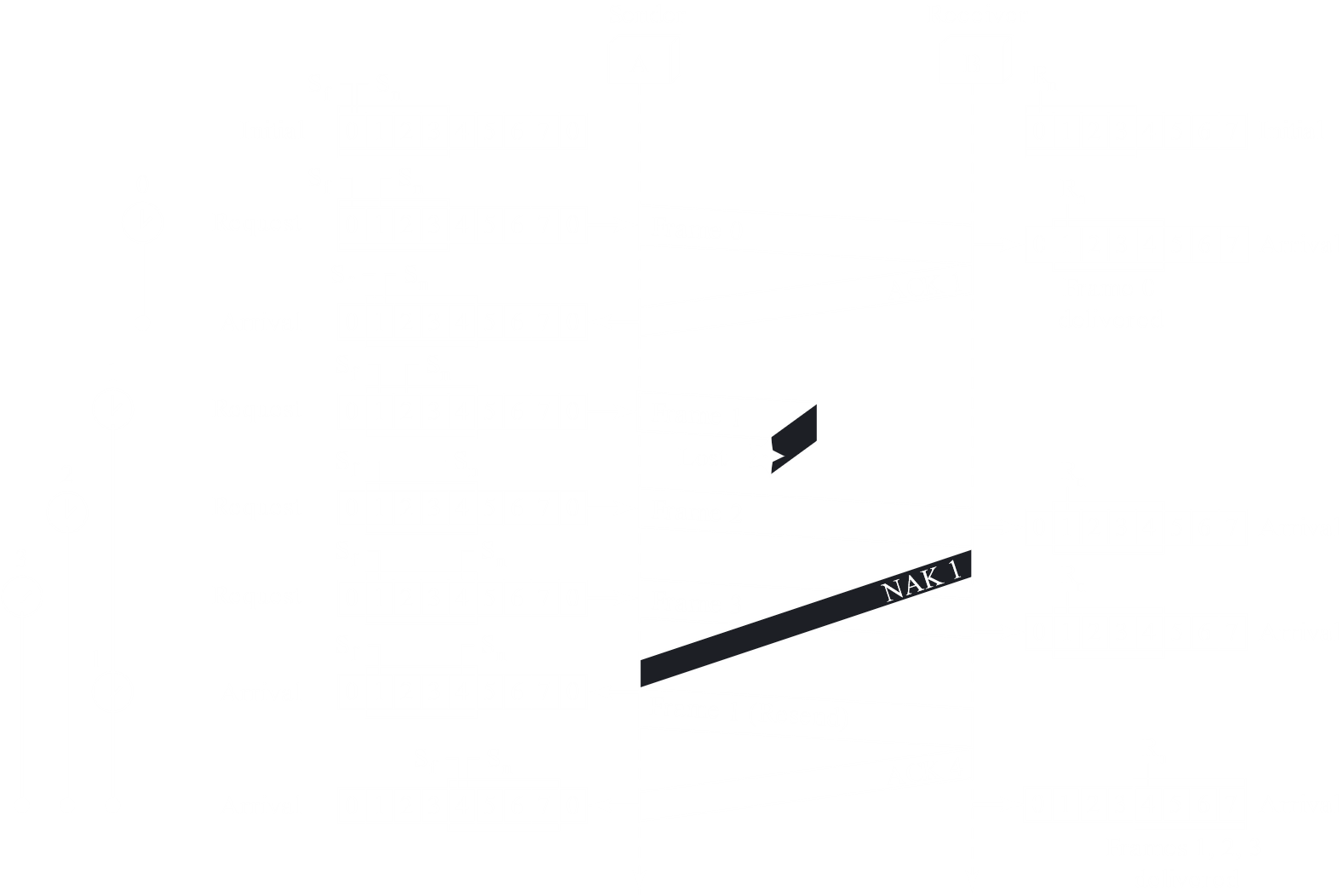


Now say the window size was increased by just . Thus, the window would hold frames through , all of which will be successfully delivered to the receiver. Thus, the receiver has the current ‘set’ of frames and is moving on to the next set, which means it requests frame from the next set. However, since none of the ACKs are coming through, the sender will think none of the frames from the previous set had been delivered, so it will send frame of the previous set again. This frame has the same number as the frame the receiver is expecting, so the receiver accepts it. Thus, we have an error.



### Selective Repeat ARQ

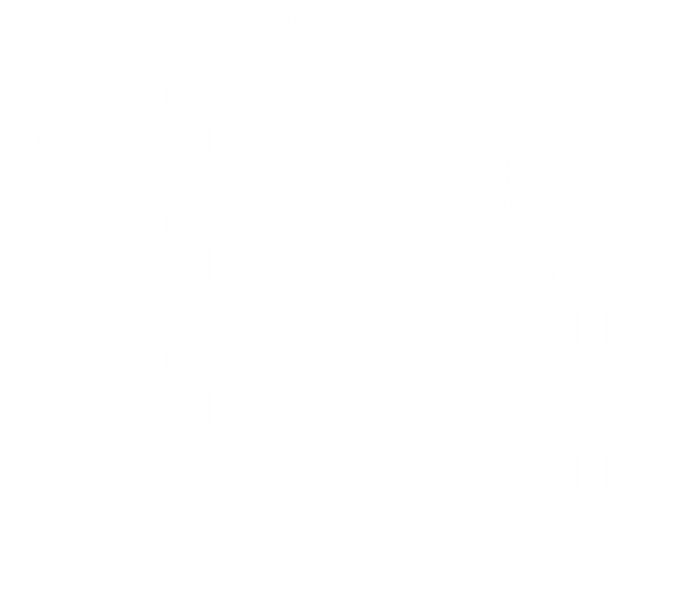
The main difference between Go-Back-N ARQ and Selective Repeat ARQ is that in Selective Repeat ARQ, the receiver window is the same size as the sender window. Thus, it does not discard any future frames when a frame in between is dropped or damaged. Instead, once the last frame for the window is received, the receiver sends negative acknowledgements (NAKs). One NAK is sent for each frame in the window that is missing. This NAK allows the sender to understand which frames were lost and thus retransmit just those frames. Additionally, ACKs are also sent for each frame that is received correctly.



#### Window Size

The window size for both the sender and receiver is the same in Selective Repeat ARQ, but it is much smaller. The window size is limited to , i.e. half the total number of frames.

Say we have four frames numbered to . Thus, the maximum window size is . If both the first frames are delivered successfully, but the ACKs for both are lost, the sender will resend frame . However, the window at the receiver’s end has moved and the receiver is expecting frame and now. As such, the receiver will discard the frames that the sender is sending. This will continue until the sender gets one of the ACKs informing it that the previous frames have been received already and thus sends the next set of frames.



Now consider what would happen if the window size was increased by just . We would have frames through in the window. All three would be received by the receiver, so the receiver’s window would now be waiting for frame from the current set and frames and from the next set. However, since all the ACKs are lost, the sender will think that the first three frames from the previous set were not delivered and will resend them. The receiver will mistake frames and from the previous set for the corresponding frames from the next set, since the numbers match. The receiver will thus accept those frames, causing an error.

