1 ARQ: Retransmission Strategies

We discussed hierarchical layering as a form of modularity that depicts the data network design in an abstract way. Each layer in the architecture regards the next lower layer a black box which provides a virtual communication channel: peer processes in a given layer exchange messages (Protocol Data Units) using the (communication) service provided by lower layer black box (see Figure 1).

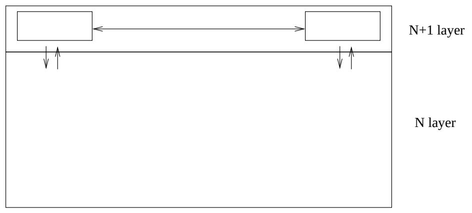


Figure 1: One peer module communicates with its peer at N+1 layer by placing a message into N layer black box communication system

In this section, we discuss how we can design a protocol that provides a reliable data transfer (service) to the next higher layer. We request that the protocol ensures that packets are delivered in order, and released once, and only once, and without errors to the next higher layer. What complicates the design of such a protocol is the fact that the lower layer may provide a unreliable service; that is packets maybe lost or delivered with errors. An example of such a situation is TCP/IP: TCP is implemented on the top of IP which provides a unreliable data transfer, but provides a reliable transfer to protocols in the application layers (such as SMTP) (see Figure 2). How to design such a protocol is the topic of this section.

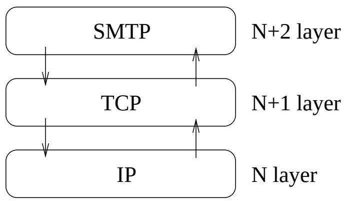


Figure 2: TCP is implemented on the top of IP which provides a unreliable data transfer, but provides a reliable transfer to protocols in the application layers (such as SMTP)

2 Stop-and-Wait ARQ Service

In an ideal world, packets (data units) exchanged between peer process would always be delivered without errors and within a bounded delay (see Figure 3).

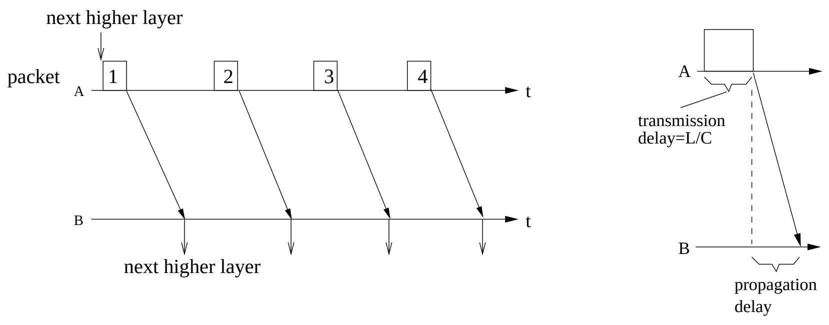


Figure 3: "Ideal world": Every packet is delivered without errors and within a bounded delay.

Unfortunately, when the lower layer provides a unreliable service, packets can be lost, arbitrarily delayed, or corrupted by bit errors (see Figure 4). Thus to obtain a reliable service, we have to design a protocol on the top of such an unreliable service that ensures that packets are delivered in order, and each packet is delivered once, and only once, and without errors. We make the following assumptions.

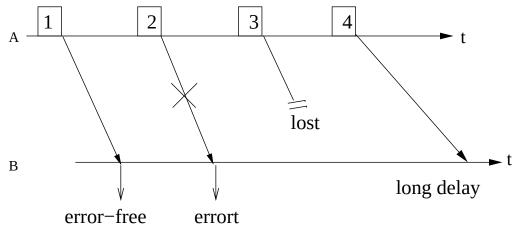


Figure 4: When the lower layer provides a unreliable service, then packets can be lost, arbitrarily delayed, or corrupted by bit errors

- **All packets with error are detected**
- **Delay can be arbitrarily long**
- **Some packets maybe lost**
- **Packets that arrive are in the same order**

Question to the reader: Are those assumptions always realistic?

In addition, we only consider unidirectional data transfer. The bidirectional case is conceptually not more difficult, but much more tedious to explain.

2.1 A First Approach

Consider data communication between two peer processes A (sender) and B (receiver). The following protocol seems to be provide a reliable service. B acknowledges received packet received from A: when B receives an error-free packet from A, then B sends an positive acknowledgement(called ACK) to A - when B receives a packet with an error, then it sends a negative acknowledge (called NAK) back to A. The sender A resends (retransmits) each packet until it receives a positive acknowledgment form B; only then A starts to send its next packet. In addition, A also resends a packet when it doesn't receive a ACK or NAK within a time-out period. This is to account for the fact that a packet might get lost (see Figure 5).

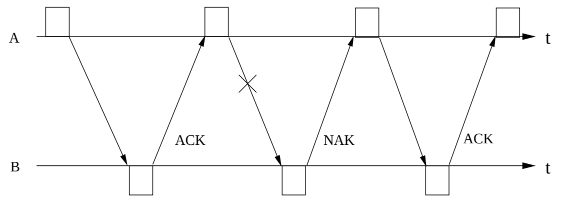


Figure 5: The sender A resends (retransmits) each packet until it receives a positive acknowledgment form B; only then A starts to send the next packet.

We illustrate this protocol using a few scenarios (see Figure 6).

1. "Ideal" Scenario: Assume that all packets, ACK's, and NAK's, are received without errors and within a bounded delay. When B receives an error-free packet, it sends a ACK to A and releases the packet to the next higher layer. Once A receives the ACK from B, it sends the next packet; and so on.
2. Error: Assume that a packet, ACK, or NAK, is corrupted by a bit error. In all these cases, A will detect the error and resends the packet.
3. Loss of a packet, ACK, or NAK: Assume that a packet, ACK, or NAK is lost. In this case, a time-out will prompt A to resends the old packet.

The scenarios seem to suggest that the protocol is correct; however, there is a problem with the protocol. Consider the situation where the ACK from the receiver is delayed causing the sender A to time-out and resend the same package. In this case, B can not distinguish whether the second

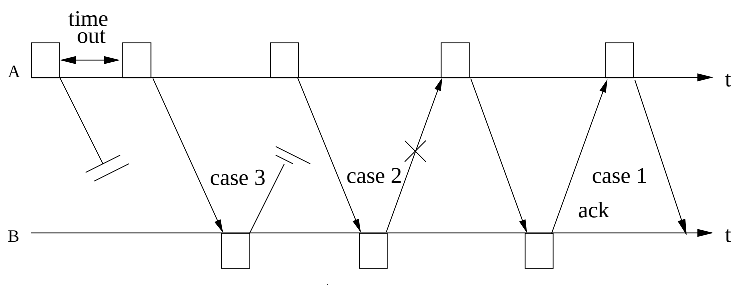


Figure 6:

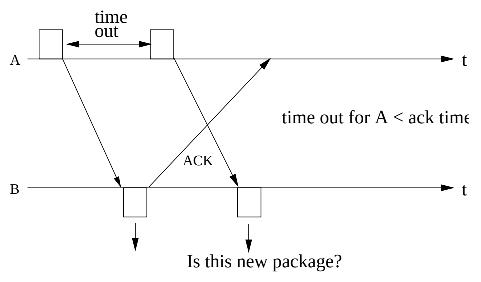


Figure 7:

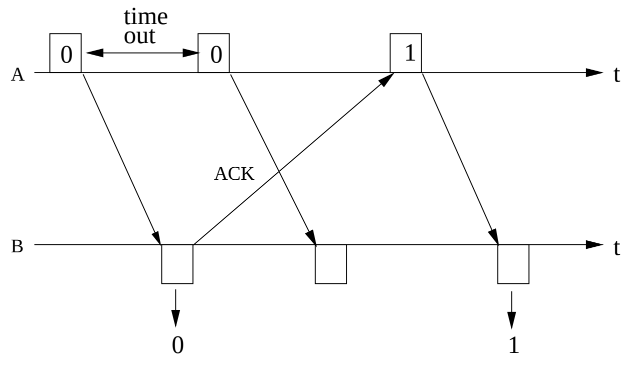


Figure 8: A sequence number SN is used to uniquely identify each packet.

transmission is a new package or not (see Figure 7). If B decides that it is a new packet and releases the packet to the next higher layer, then the constraint that each packet has to be delivered once and only once is violated.

2.2 The Stop-and-Wait Protocol

To avoid this, we can introduce sequence numbers (SN) which are included in the header of packets sent from A to B. Each packet has a unique sequence number and A increases the sequence number by one whenever a new packet is sent (see Figure 8).

However, even introducing sequence numbers does not entirely solve the problems of the above protocol. For instance, consider the situation where A sends packet twice due to a time-out. Assume that both packets arrive error-free and B sends twice a ACK. After receiving the first ACK, A sends the next packet in sequence, and then on receiving the second ACK, could interpret that as an ACK for the new packet, potentially causing a failure of the protocol (see Figure 9).

To overcome this problem, the receiver B, instead of returning a ACK or a NAK, should return the number (RN) of the next packet awaited (see Figure 10) . An equivalent convention would be to return the number of packet just accepted, but in practice this convention is not customary. B can request this next awaited packet upon the receipt of each packet, at periodic intervals, or at an arbitrary selection of times.

This last protocol is called Stop-and-Wait ARQ (Automatic Repeat reQuest). It is defined as follows.

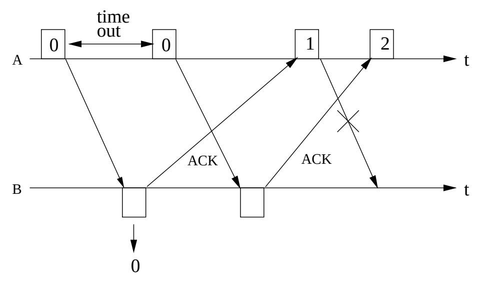


Figure 9: The second ACK received at A is interpreted as a ACK for packet 1; consequently, packet 1 will never be received at B

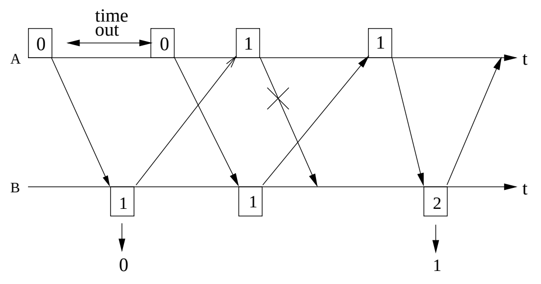


Figure 10: Instead of returning a ACK or a NAK, B returns the number (RN) of the next packet awaited

The algorithm at A for A-to-B transmission:

1. Set the integer variable SN to 0
2. Accept a packet from the next higher layer at A; if no packet is available, wait until it is; assign number SN to the new packet.
3. Transmit the SNth packet containing SN in the sequence number field.
4. If an error-free packet is received from B containing a RN greater that SN, increase SN to RN and go to step 2. If no such packet is received within some finite delay, go to step 3.

The algorithm at B for A-to-B transmission:

1. Set the integer variable RN to 0 and the repeat steps 2 and 3 forever.
2. Whenever an error-free packet is received from A containing a sequence number SN equal to RN, release the received packet to the higher layer and increment RN.
3. An arbitrary times, but within bounded delay after receiving any error-free data packet from A, transmit a packet to A containing RN in the request number field.

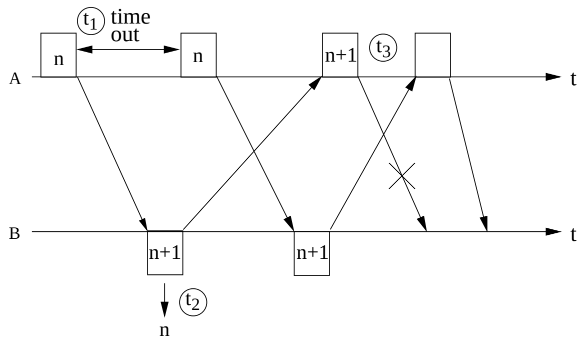


Figure 11: Proof of stop-and-wait ARQ

Can we verify that this protocol is indeed correct? Here, we give only a rough sketch of a the correctness proof for Stop-and-Wait ARQ. The proof involves two aspects: safety and liveness.

1. **Safety**: A protocol is safe if it never produces an incorrect result. In our context, this means it never releases a packet out of the correct order to the higher layer. To proof safety, it is sufficient to show that for the diagram in Figure 11 we have that $t_1 < t_2 < t_3$, where the packet n is transmitted at t_1, the packet n is received at t_2, and the sequence number is increased to n+1 at t_3. If $t_1 < t_2 < t_3$, the packet n+1 is transmitted only after packet n has been received; this guarantees safety.

2. **Liveness**: A protocol is live if it produces forever results (no deadlock). To proof liveness, one has to show that the time difference $t_3 - t_1$ is finite. This guarantees that each packet is eventually sent and received (within a finite delay). A necessary condition for liveness is that the probability p_0 that a packet, or ACK, is received without an error is positive ($p_0 > 0$). This excludes the case of a “broken pipe”.

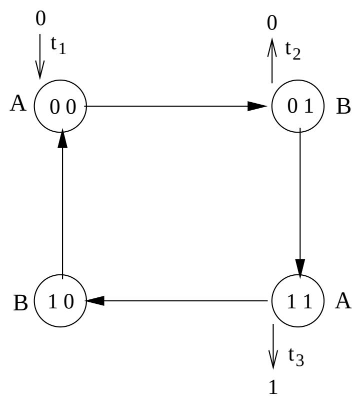


Figure 12: State transition diagram for modulo 2.

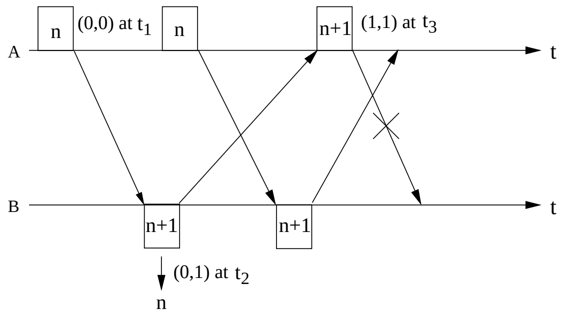


Figure 13: Trajectory of the state values for modulo 2.

3 Implementation Issues

In stop-and-wait ARQ, the sequence number can potentially become arbitrarily large. In order to avoid using a very large field (header), their values should be sent using a modulus m. In this case, each sequence and request number can be represented with a fixed number of bits, namely $log_2(m)$ bits. Given our assumption that packets travel in order, it turns out that modulus 2 is sufficient.

Figure 12 illustrates that modulo 2 is sufficient using a state diagram, where a state is identified by the corresponding sequence and request numbers (SN,RN). The possible states are (0,0), (0,1), (1,1), (1,0).

Initially, the combined state of A and B is (0,0). When the packet 0 is received without error, the RN at B changes to 1, yielding (0,1). When A receives this new RN value, the new state is (1,1). When the packet 1 is received at B, the RN at B changes to 0 and the new state is (1,0). Finally, when A receives the request RN=0, then the combined state is becomes again (0,0). From the state transition diagram, we can infer that the protocol is safe. A starts sending an packet with an even number only when the previous packet with an odd number has been received at B and released to the next higher layer. B releases each packet in correct order and once, and only once, to the higher layer (see Figure 13).

Note that the combined state is known to B only at the instant when the transition from (0,0) to (0,1), or the transition from (1,1) to (1,0), occurs. Similarly, the combined state is known to A only at the instant when the transition from (0,1) to (1,1), or from (1,0) to (0,0), occurs. This means

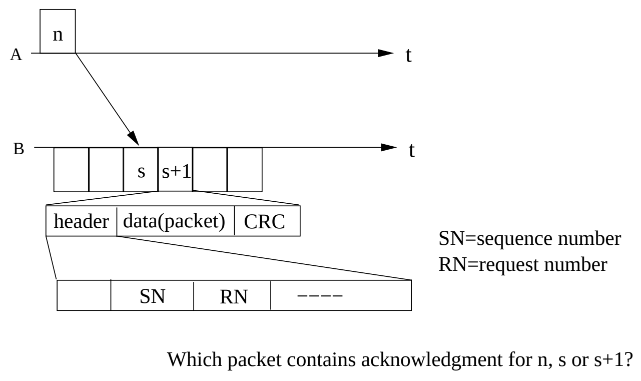


Figure 14: "Piggyback": Packets carry both a sequence SN and request number RN in the header.

that the combined state is never known to both A and B at the same time, and it is frequently unknown to both. But still, the protocol is correct, i.e. the sender A knows when it can start to send a new packet and B knows when a packet can be released to the next higher layer.

This situation here is very similar to that of three army problem. Here however, the protocol does not deadlock (is live) even though the combined state is never known to both processes at the same time instant. In the three army problem, it is impossible to coordinate the attack because the combined state has to be known jointly to both processes.

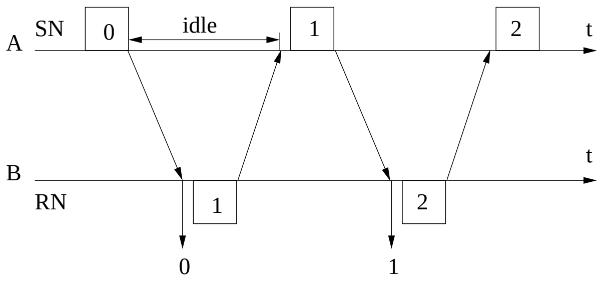
In the case of a bidirectional communication (B also send packets to A), packets sent from B to A can carry the request numbers "piggybacked" in the headers of the data to be sent (see Figure 14). Notice that when the packet is received at B, the acknowledgment for that packet is not sent back immediately, but only when B starts to transmit a new packet. In the Figure 14, when the packet n is received, then the acknowledgment is not sent back with packet s but until the next packet s+1. Why is this so?

There are a few additional issues that have to be addressed when implementing stop-and-wait ARQ, which we will cover when we discuss TCP.

- The two peer processes must agree to a starting "state" - in this case the first sequence number to be used - and must maintain that state as the protocol operates.
- Setting the timeout parameter is a trade-off between not requiring too many retransmissions and incurring long delays when packets are lost.

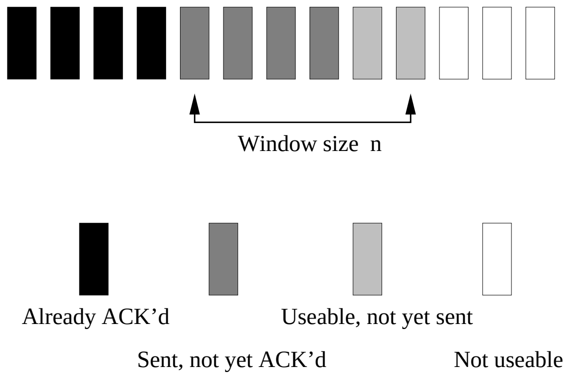
4 Go-Back n ARQ

Although Stop-and-Wait ARQ is simple, safe and live, it is not efficient.



During the time interval when the sender A waits for a ACK, A could send some subsequent packets in order to achieve higher throughput.

Go-Back n ARQ allows A to send n consecutive packets before hearing the first ACK from the receiver. As in Stop-and-Wait ARQ, sequence (SN) and request numbers (RN) are used. In addition, the sender maintains a window of size n indicating the range of packets that A is allowed to send. The window is initialized to $[0 \ldots n-1]$ and A can send packet 0 to $n-1$. As a acknowledgements received, this window slides upward; thus Go-Back n protocols are often called sliding windows ARQ protocols.



Go-Back n algorithm for sender A and receiver B:

Sender A:

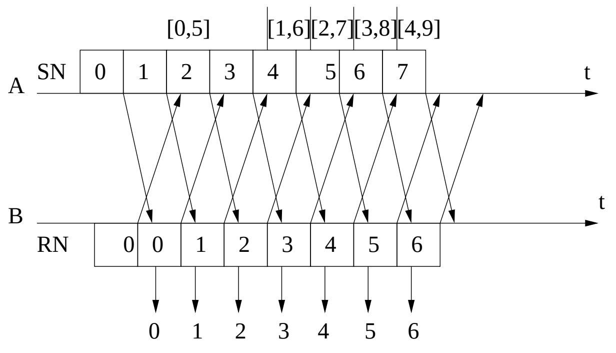
(1) Set SN_{min} and SN_{max} to 0,
(2) Repeat steps (3)-(5) forever, in any order.
(3) If $SN_{max} < SN_{min} + n$ and a packet is available from the higher layer, accept that packet, assign number SN_{max} to it, and increment SN_{max}.
(4) If a request number $RN > SN_{min}$ arrives from B, increase SN_{min} to RN.
(5) If $SN_{min} < SN_{max}$ and no packet is currently in transmission, transmit a packet SN, where $SN_{min} \leq SN < SN_{max}$. At most a bounded delay is allowed between successive transmissions of packet SN_{min} over intervals when SN_{min} does not change.

Receiver B:

(1) Set RN to 0, repeat steps (2) and (3) forever.
(2) When an error-free packet SN arrives from A, if $SN = RN$, release this packet to the higher layer and increment RN.
(3) Within bounded delay after an error-free packet is received, transmit request number RN to A.

Below we consider a few scenarios for the Go-Back n protocol.

Example 1. An ideal scenario of Go-Back n ARQ $(n = 6)$.

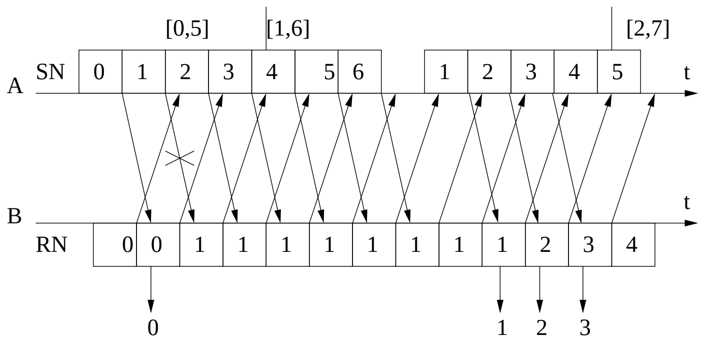


This scenario is "ideal" in the sense that:

- All packets are received error-free.
- ACKs always arrive back to sender "in time" — before sender's window is exhausted and resending is triggered.

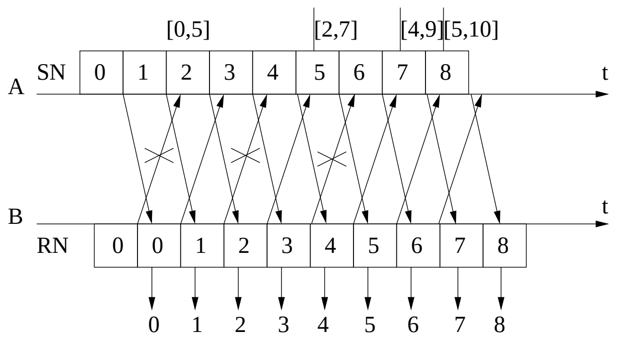
The sender A starts by transmitting packet 0. B receives packet 0 error-free, releases it to the higher layer and sends an acknowledgement back to A. After sending packet 0, A continues sending the successive packets. Before packet 5 is sent, the acknowledgement for packet 0 arrived from B error-free, so A moves its window from [0, 5] to [1, 6]. Then A starts sending packet 5, during which the acknowledgement for packet 1 arrives, and A adjusts its window to [2, 7]. As this scenario continues, A keeps sending new packets to B; each packet gets through the link correctly; B releases the packets in order and sends corresponding acknowledgements back to A, and A moves its window towards higher numbers upon receiving the acknowledgements. As a result, channel is fully utilized (Stop-and-Wait ARQ won't achieve this).
Note that packet 0 arrives at B when B is sending its second packet to A. The customary of piggybacking acknowledgements (RN numbers) is to frame them at the beginning of a packet. Hence it is too late to piggyback the ACK for packet 0 on the second packet from B to A; it will be piggybacked on the third packet.

Example 2. Go-Back n ARQ with data packets errors ($n = 6$).

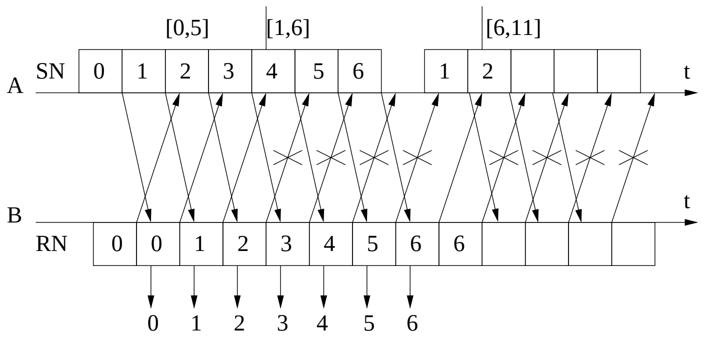


In this example, we will examine the behavior of Go-Back n when a data packet from the sender is not received correctly. As in Example 1, packet 0 arrives from A to B error-free, and B acknowledges this to A when it sends its third packet to A. However, packet 1 is corrupted/lost on its way to B. Hence in the following packets from B to A, B keeps asking for packet 1 (acknowledging packet 0). Again the ACK for packet 0 arrives at A when A is sending packet 4. A moves its window from [0, 5] to [1, 6] and continues sending packets 4, 5 and 6. During this period, A does not receive any new ACKs. Hence after packet 6 is sent, A's window is exhausted. A times out and starts resending from packet 1. This time packet 1 arrives at B error-free. The subsequent steps are similar to those in Example 1. Note that although packets 2-6 arrive at B error-free at the first time, they are still resent.

Example 3. Go-Back n ARQ with ACK errors ($n = 6$).

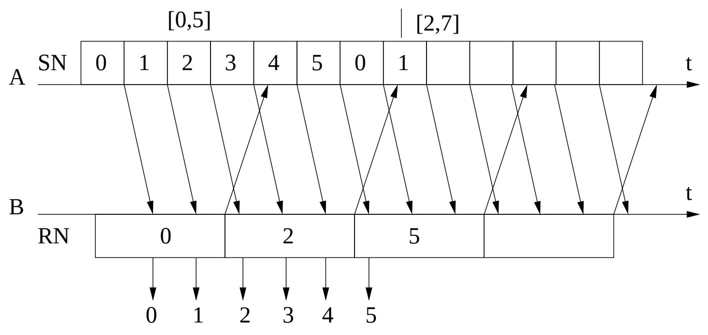


In the figure above, all data packets from A to B are error-free, but some ACKs from B to A are corrupted/lost. For instance, the ACK for packet 0 is lost. Therefore B does not receive this ACK during its transmission of packet 4, and its window remains as [0, 5], until the ACK for packet 1 arrives error-free when it is sending packet 5. In Go-Back n ARQ, the ACK for packet k effectively acknowledges all packets with sequence numbers less than or equal to k. Hence upon receiving the ACK for packet 1, A knows that both packet 0 and packet 1 have been successfully received, and adjusts its window from [0, 5] to [2, 7]. When A finishes sending packet 5, it is now two packets away from the window border, so it continues to send packet 6. If the ACK for packet 1 was also lost, A would have exhausted its window after it sends packet 5. Similarly, the loss of the ACK for packet 2 is remedied by the ACK for packet 3.
This shows that ACK errors are often a less severe problem than data error for Go-Back n ARQ. An ACK corruption/loss will not have any effect on channel utilization as long as a subsequent ACK arrives correctly and in time. Of course, if the corrupted/lost ACK message is piggybacked on a data packet from B to A, then there might be a data packet error at the same time for the other direction.



However, consecutive ACK errors may trigger unnecessary resendings. In the above figure, packets 0-5 all arrived at B error-free. Packet 0 is successfully acknowledged, while the ACKs for packets 1-4 are lost. The sender A receives the ACK for packet 0 when it is sending packet 4. It adjusts its window from [0, 5] to [1, 6] and continue to send packets 5 and 6. During this period, no ACKs arrive. Therefore when A finishes sending packet 6, its window is exhausted. A times out and starts resending from packet 1, which is the first unacknowledged packet.

Example 4. Go-Back n ARQ with uneven packet sizes ($n = 6$).

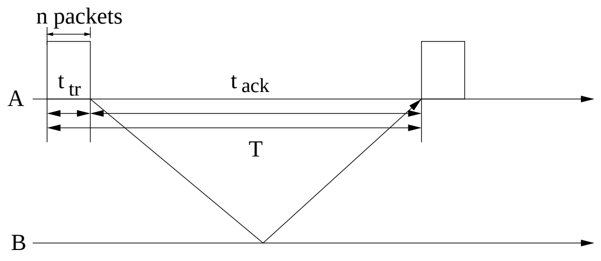


Assume ACKs are piggybacked. If we have a much larger packet size in one direction than in the other, problems may arise. In the above figure, no packet is corrupted or lost, but the ACKs from B to A are delayed by the long packets at B, and fail to arrive in time. For instance, packets 0 and 1 both arrive at B when B is sending its first packet. As explained earlier, it is too late for B to piggyback the ACK for these two packets on its first packet. So the first ACK from B to A isn't sent out until B finishes sending both its first and second packets, which are relatively long. By the time B's second packet arrives at A, A has exhausted its window and is in the process of retransmission.

How to choose the Go-Back number n?

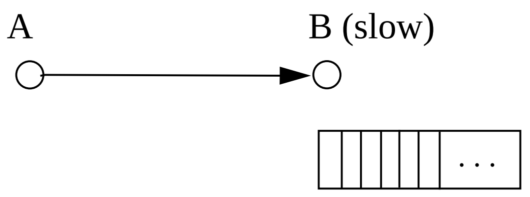
(1) On one hand, we would like n to be **as large as possible**. A large n can avoid retransmissions or delays due to large propagation delay and long frames or errors in the reverse direction.

(2) **Congestion control**. We discuss in the tutorial that the Go-Back number n (among some other factors) dictates the data rate at the sender and the number n can be used for congestion control.



Assume there is no packet error. As shown in the figure above, the sender sends out n consecutive packets, then waits for the ACK to come back before it starts sending the next n packets. Assume each packet has length 100 bits, the round-trip time is 2 seconds, and the capacity of the link is 100,000 bits/second. Therefore the data transmission rate at the sender will be $\frac{100n}{\frac{1}{100}+2}$ bits/second. If there are 200 senders sharing the same link and the link capacity is equally shared among senders, the following condition $\frac{100n}{\frac{1}{100}+2} \leq \frac{100{,}000}{200}$ has to hold in order to avoid link congestion; this implies that the Go-Back number n should not exceed 10.

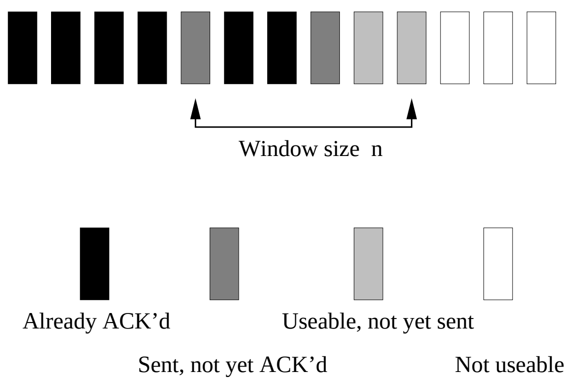
(3) **Flow control** In cases where the receiver B is slow (slow connection, slow processing speed), the sender A should refrain itself from overfeeding B's receiving buffer. This imposes another upper-bound on n.



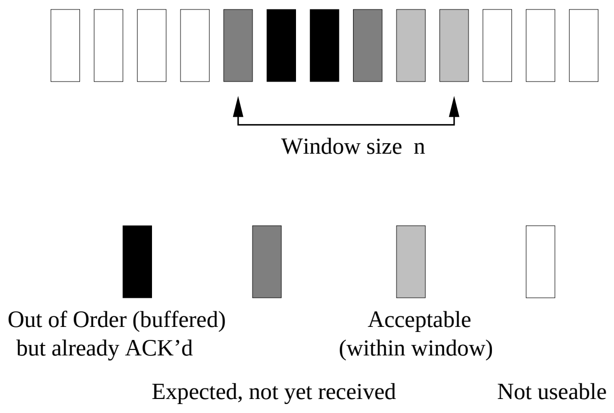
Assume that the buffer at B can hold at most 10 packets. Requiring that n does not exceed 10 will ensure that A never overfeeds that buffer at B.

5 Selective-Repeat ARQ

In Go-Back n ARQ, packets that arrive at the receiver error-free may not be accepted, due to corruption/loss of earlier packets, and hence have to be transmitted more than once. This motivates Selective-Repeat ARQ, where the out-of-order packets can be temporarily buffered to save further transmission, and only packets not correctly received are retransmitted.
In Selective-Repeat ARQ, both the sender and the receiver maintain a sliding window. As in Go-Back n ARQ, the sender side window dictates how far the sender can go from the first unacknowledged packet; the left end of the window is the first packet whose ACK has not been correctly received. The receiver side window dictates which packets may be accepted, namely packets with sequence numbers from RN to $(RN + n - 1)$; the left end of the window is the first packet that has not been correctly received.



Window at sender side


Window size n
Out of Order (buffered) but already ACK'd
Acceptable (within window)
Expected, not yet received
Not useable

Window at receiver side

Selective Repeat algorithm for sender A and receiver B:
In Selective Repeat, the sender keeps a buffer for storing ACK's from B that are received out-of-order (but are within the sender window), and the receiver keeps a buffer for storing error-free packets from A that are received out-of-order (but are with in the receiver window).

Sender A:
(1) Set SN_{min} and SN_{max} to 0,
(2) Repeat steps (3)-(6) forever, in any order.
(3) If $SN_{max} < SN_{min} + n$ and a packet is available from the higher layer, accept that packet, assign number SN_{max} to it, and increment SN_{max}.
(4) If a request number $RN = SN_{min}+1$ arrives from B, increase SN_{min} to RN and do the following:

(a) If the value $SN_{min} + 1$ is stored in the buffer, then go to step (b); otherwise quit Step (4).

(b) Delete the value $SN_{min} + 1$ from the buffer and increment SN_{min}. Go to step (a)

(5) If a request number RN, $SN_{min} + 1 < RN \leq SN_{max}$, arrives from B and RN is not stored in the buffer, then store RN in the buffer.
(6) If $SN_{min} < SN_{max}$ and no packet is currently in transmission, choose a packet number SN, $SN_{min} \leq SN < SN_{max}$, such that $SN + 1$ is not yet stored in the buffer, and transmit the SNth packet. At most a bounded delay is allowed between successive transmissions of packet SN_{min} over intervals when SN_{min} does not change.

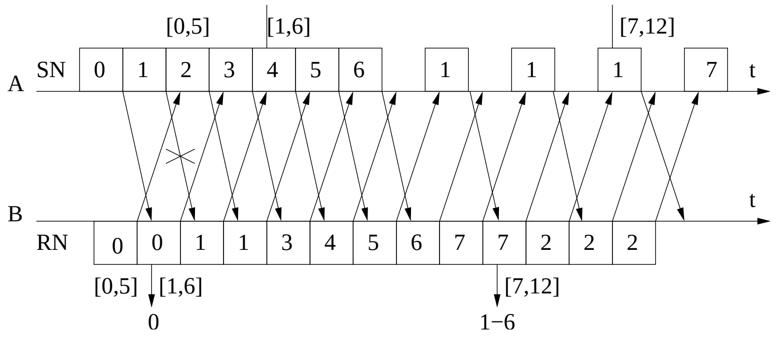
Receiver B:
(1) Set RN and RN_{min} to 0, repeat steps (2), (3) and (4) forever.
(2) When an error-free packet $SN = RN_{min}$ arrives from A, release this packet to the higher layer, increment RN_{min}, and send an ACK with $RN = RN_{min}$ to A. Then do the following:

(a) If the packet RN_{min} is stored in the buffer go to step (b); otherwise quit Step (2).

(b) Release the packet RN_{min} to the higher layer and increment RN_{min}. Go to step (a).

(3) When an error-free packet SN, $R_{min} < SN < RN_{min} + n$, arrives from A, then within bounded delay send an ACK with $RN = SN + 1$ to A. If the packet is not yet stored in the buffer, then store it in the buffer.
(4) When an error-free packet SN, $R_{min} - n \leq SN < RN_{min}$, arrives from A, then within bounded delay send an ACK with $RN = SN + 1$ to A.

Example 5. Selective-Repeat ARQ with data packet error



This example shows how Selective-ARQ deals with data packet errors. Similar to the scenario in Example 2, packet 1 is lost on its way to B, while the other packets and ACKs get through correctly. Since B does not receive packet 1, it will not acknowledge it. However, when the successive packets arrives at B correctly, B will buffer them and acknowledge them to A instead of discard them and keep asking for packet 1. When A finishes sending packet 6, it exhausts its window since the ACK for packet 1 is still not received. A times out and resends packet 1. During this period, ACKs for packets 4 and 5 arrive, but ACKs for packets 1 and 6 are still not received. Depending on the specific algorithm, A could now choose to resend either packet 1 or packet 6. In this example we assume the former. During the second retransmission of packet 1, the ACK for packet 6 arrives. Then A resends packet 1 for the third time. During the following timing out period, the ACK for packet 1 finally arrives. So A adjusts its window from [1, 6] to [7, 12] and starts sending packet 7. On the other side, when B receives a correct version of retransmitted packet 1, it accepts that packet, releases packets 1-6 to the higher layer in order, adjusts its window to [7, 12] and waits for packets within that range to come.

4. Choosing modulus m in ARQ

In practice, sequence numbers and request numbers cannot keep increasing without bound; rather, their modular versions with modulus m are used. This way each sequence number and request number can be represented within a fixed number of bits, namely $log_2(m)$ bits. The smaller m is, the less bits are required. However, we need to be careful about the choice of m in order to preserve the correctness of the ARQ algorithms. It turns out that for the three kinds of ARQ algorithms that we discussed, the following restrictions of m must be satisfied respectively:

- Stop-and-Wait ARQ: $m \geq 2$
- Go-Back n ARQ: $m > n$
- Selective-Repeat ARQ: $m \geq 2n$