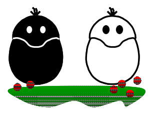
Computer Networks
Error correction with ARQ

Saad Mneimneh
Computer Science
Hunter College of CUNY
New York



- I am Ok.
- I am not Ok.

Ack and Nak

1 Introduction

We started our discussion of the DLC layer by saying that it is responsible for reliable transmission of **packets over a single link**, which means that every packet is delivered once, only once, without errors, and in order. To achieve this goal, the DLC had to detect errors using framing and parity check codes. Now assuming that the DLC can detect errors, the question becomes how to correct these errors and how to make sure that each packet is delivered once, only once, and in order?

The delivery once and ordering properties can be solved by including a sequence number in the packet header (in addition to possibly other framing information). The DLC can then determine whether a duplicate frame or an out of order frame is received, and ignores any such frame. So the essential problem appears to be error correction.

The last statement may give the wrong impression that the problem of reliable communication is one sided. Most of the times, errors cannot be corrected at the receiving side; so if the receiver declares an error upon receiving a frame, the sender must resend that frame sometime in the future. The receiver may have to request such retransmission. Moreover, if the receiver does not receive any frames, then nothing needs to be corrected. But for reliable communication to exist, the sender must guarantee the delivery of frames. While on a single link, lost generally means received with errors, if frames can be lost (for whatever reason), the receiver may have to acknowledge the receipt of frames in some way.

Therefore, the problem of reliable communication is a coordination problem among both the sender and the receiver DLCs. At any point in time, both DLCs must have a clear view of what has happened so far. But it appears from above that by simply acknowledging the receipt of frames (those that are not duplicates, are in order, and contain no errors), the receiver can instruct the sender of the next frame to be sent. The solution however is not that simple. To appreciate the difficulty of the problem, let us consider a classical problem known as the coordinated attack problem.

2 The coordinated attack problem

There are three armies, two colored blue, and one colored red. The red army separates the two blue armies. If the blue armies attack simultaneously, they win, but if they attack separately, the red army wins. The only communication between the blue armies is by sending messengers through the red army lines, but there is a possibility that any such messenger will be captured, causing the message to go undelivered. How should the blue armies coordinate their attack?

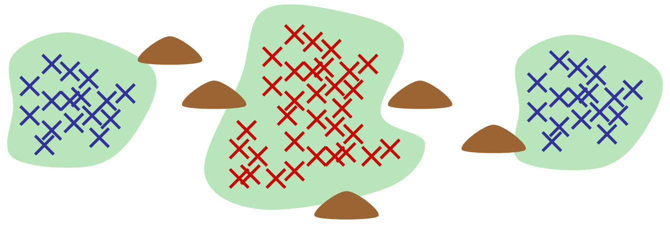


Figure 1: The coordinated attack problem

Let us state this problem formally. Denote the two armies by A and B. Both A and B make their own initial decisions, i.e. 1 (let's attack) or 0 (let's not attack). A and B can then exchange messages. Every message can be lost. A and B must finally agree on a decision, and the final agreement must be on one of the decisions made initially (why?). The goal is to find an algorithm that terminates after a finite number of messages and that solves the agreement problem.

What may be surprising is that no such algorithm exists! This fact is not hard to prove. The proof is by contradiction. Assume that such an algorithm exists. Execute this algorithm on an instance where both A and B initially decide 1. Then they must agree on 1 after a finite number of messages.

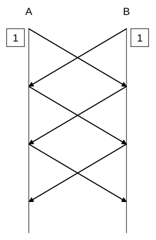


Figure 2: A and B agree on 1 after a finite number of messages

Now assume that the last message from A to B is lost. The execution of the algorithm looks the same to A. Therefore, A decides 1. B must also decide 1.

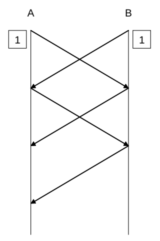


Figure 3: Last message from A to B is lost. A and B still agree on 1.

Now assume that the last message from B to A is lost. The execution of the algorithm looks the same to B. Therefore, B decides 1. A must also decide 1.

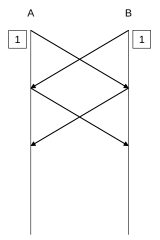


Figure 4: Last message from B to A is lost. A and B still agree on 1.

Repeat the argument until all messages are lost. Both A and B still decide 1.

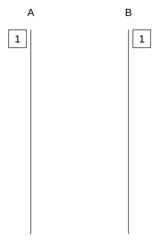


Figure 5: All messages are lost. A and B still agree on 1.

Now assume that B's original decision is changed to 0. The execution of the algorithm looks the same to A. Therefore, A decides 1. B must also decide 1.

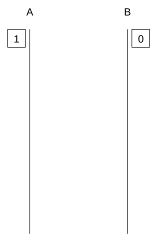


Figure 6: B's initial decision changes to 0. A and B still agree on 1.

Now assume that A's original decision is changed to 0. The execution of the algorithm looks the same to B. Therefore, B decides 1. A must also decide 1.

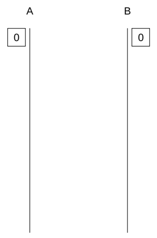


Figure 7: A's initial decision changes to 0. A and B still agree on 1.

We reach a contradiction! Therefore, such an algorithm does not exist.

So how can we agree on anything in the presence of message loss? The impossibility of coordinated attack is a result of the setting itself. For instance, if there is a non-zero probability that a messenger from the blue army will not be captured, a message will eventually get through the red army lines. We can then find an algorithm to ensure that both armies attack simultaneously with high probability (can you think of one?).

For most problems of communication, we assume that "eventually something good will happen". Therefore, we assume that there is a probability $p > 0$ that a frame will be received without errors. The sender might be required to wait for a confirmation from the receiver of this event. But there is no point in time where both the sender and the receiver have complete knowledge of the "state of the world". That would be equivalent to require both the sender and the receiver to know the following fact before proceeding further with any communication.

$$\text{``}I\ know\ that\ he\ knows\ that\ I\ know\ that\ he\ knows...$$
$$that\ the\ first\ packet\ was\ successfully\ communicated''$$

In other words, we do not require them to agree that the first packet was successfully delivered before starting on the next one. The following pattern of communication is very common: A sends a packet ... A does not know if B got the packet... A may resend... B sends an acknowledgment... B does not know that A got the ack... B may resend... A sends another packet upon receiving the ack... A does not know that

B got the packet... A may resend... B sends an acknowledgment... B does not know that A got the ack... B may resend...

This pattern of communication falls under the general strategy of *a*utomatic *r*epeat re*q*uest (ARQ), in which the receiving DLC detects frames with errors and then requests, in one way or another, the sender to repeat the information in those erroneous frames. That's how error correction is done. We will study different ARQ strategies.

3 Stop and wait ARQ

Let A be the sender and B be the receiver. The basic idea of Stop and wait ARQ is that A sends a frame to B and then waits for an acknowledgment (called ack) from B before sending another frame. When we start thinking about this strategy in detail, we realize that we have a problem. For instance, the frame or the ack may be lost. On a single link, this generally happens because of errors (so there is no physical loss here). While an ack with error may still be regarded by A as some kind of acknowledgment (though unlikely), if the frame from A to B is lost, B will never send the ack. Then A will wait forever. One possible remedy is to make B send a negative acknowledgment (called nak) upon receiving a frame with errors. In this case, A will never have to wait indefinitely. But regardless, A may receive the ack or the nak with errors (a frame from B to A), and hence cannot determine which is which. A solution to this problem is for A to use a *timeout*. If a nak is received, or an ack is not received within a timeout period, A resends the frame. Unfortunately, this does not correctly solve the problem as illustrated by the following figure.

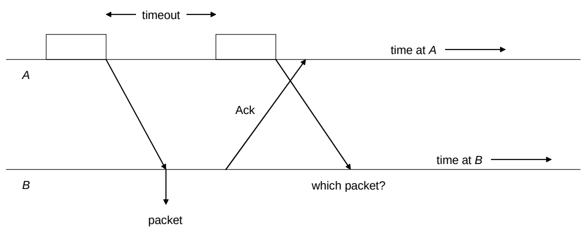


Figure 8: B cannot tell which packet is being received

Referring to Figure 8 above, one might think that B could simply compare the packets to resolve the issue of a duplicate packet. However, as far as the DLC is concerned, packets are arbitrary bit strings and the first and second packets could be identical.

A can add a sequence number SN to the frame header. B can use the sequence number to tell which packet is being received. Unfortunately, even the use of sequence numbers is not quite enough to ensure correct operation. The figure below illustrates the problem.

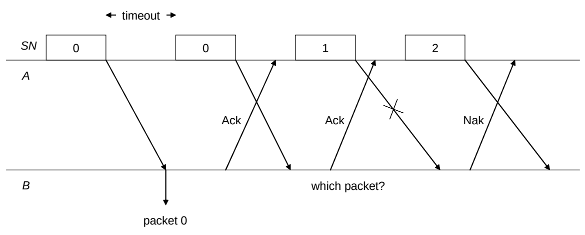


Figure 9: × means the frame arrives in error. A thinks that the second ack from B is for frame 1.

Therefore, acks and naks must also contain sequence numbers. This however makes naks redundant. For instance, B can send an ack with a request number RN of the frame to be received next in sequence. If B receives a frame with errors, then RN remains the same. There are many ways in which B can send an ack with RN:

- upon receipt of each frame
- in periodic intervals
- at arbitrary times
- piggyback RN in the frame header for packets going from B to A

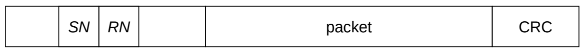


Figure 10: Piggyback RN

The final algorithm for Stop and wait ARQ is shown below for both the sender and the receiver:

```
A
$SN \leftarrow 0$
while (more packets)
      accept one packet from upper layer
      $ack \leftarrow$ false
      while (!$ack$)
            send packet in frame to $B$ with sequence number $SN$
            wait(timeout)
            if received frame from $B$ with $RN > SN$
               $SN \leftarrow RN$
               $ack \leftarrow$ true
```

```
B
$RN \leftarrow 0$
while (true)
    if frame with $SN = RN$ received
        release packet to upper layer
        $RN \leftarrow RN + 1$
    with probability $q > 0$ send frame to $A$ with piggybacked $RN$
```

The correctness of such a distributed algorithm is usually proved in two parts:

- safety: the algorithm never produces an incorrect result
- liveness: the algorithm continue forever to produce results (it never enters a deadlock in which no further progress is possible)

The safety property is self evident for this algorithm: B releases all packets in order (using a proof by induction if one wants to be formal). This however is guaranteed only because B can always detect errors. The liveness property is given by the fact that with a probability $p > 0$, a transmitted frame is received without errors, and with probability $q > 0$, B periodically sends its RN to A. Let us prove the liveness property.

- Assume that i is the value of SN at some time t_1
- Let t_2 be the time at which packet i is released to the upper layer ($t_2 = \infty$ if this event never occurs)
- let t_3 be the time at which SN is increased beyond i

We will show that $t_2 < t_3$ and that t_3 is finite given t_1 is finite. This is sufficient to demonstrate liveness using induction on i. Note that this does not prove safety, e.g. try to construct an example where $t_2 < t_3$ and t_3 is finite, and packets are delivered out of order.

Let $RN(t)$ and $SN(t)$ be the values of RN and SN at time t, respectively.

By definition of the algorithm, and the fact that both $RN(t)$ and $SN(t)$ are non-decreasing in t, we have

$$SN(t) \leq RN(t)$$

By definition of t_2 and t_3, $RN(t)$ is increased beyond i at t_2 and $SN(t)$ is increased beyond i at t_3. Since $SN(t) \leq RN(t)$, if follows that $t_2 < t_3$. Note that $t_1 < t_3$ by definition, but it is possible that $t_2 < t_1$ (why?).

If $t_1 < t_2$, then $RN(t_1) \leq i$ by the safety property (why?). Since $SN(t_1) = i$ and $SN(t) \leq RN(t)$, if follows that $SN(t_1) = RN(t_1) = i$. A continues to transmit frame i from t_1 until t_3. Therefore, the first error-free reception of frame i after t_1 will release the packet. Since $t_2 < t_3$ and $p > 0$, the time from t_1 to t_2 is finite. Therefore, either $t_2 < t_1$ or the time from t_1 to t_2 in finite. Now we show that the time from t_2 to t_3 is finite, which proves that t_3 is finite given t_1 is finite. With probability q, B (whether $t_1 < t_2$ or vice versa) continues to transmit frames carrying $RN \geq i + 1$ from time t_2 until some such frame is received error-free at A at time t_3. Since $p > 0$ and $q > 0$, the time from t_2 to t_3 is finite.

3.1 Bounded sequence numbers

One problem of the above algorithm lies in the fact that both SN and RN are unbounded. Since the size of the frame header must be finite, SN and RN will eventually not fit in the frame header. Therefore, it is practical to send these numbers modulo some integer. The question becomes what modulus is sufficient to preserve the correctness of the algorithm? Without a proper assumption on the communication, no modulus p will be sufficent. Consider the following scenario: A sends a frame with $SN = 0$ which is delayed. A times out and resends the frame with $SN = 0$. B receives the frame and sends $RN = 1$. A sends a frame with $SN = 1$. B receives the frame and sends $RN = 2$. This is repeated until A sends a frame with $SN = p - 1$. B receives the frame and sends $RN = 0$ ($p \bmod p$). B receives the initial frame with $SN = 0$ that has been delayed. Obviously, regardless of the value of p, B accepts the wrong frame. Therefore, while we may assume that frames can be lost, those that eventually arrive are assumed to arrive in the same order as transmitted. This assumption is not acceptable for general networks, but is is definitely true on a single link, and we may call it the FIFO (First In First Out) property of the link.

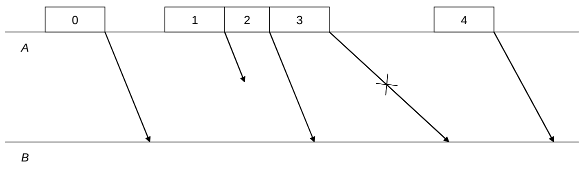


Figure 11: Ordered transmission of frames showing losses (frame never arrives denoted by short arrow) and errors (frame arrives with error denoted by $\times$)

Proposition (FIFO): If B receives a frame with sequence number SN at time t, then $RN(t) - 1 \leq SN \leq RN(t)$. Similarly, if A receives a frame with ack number RN at time t, then $SN(t) \leq RN \leq SN(t) + 1$.

We will prove a more general proposition in the next section. Based on this proposition, both the sender and the receiver need to distinguish between two values at any time. Therefore, sending SN and RN modulo 2 is sufficient to resolve any ambiguity. Here's the modified algorithm:

```
A
$SN \leftarrow 0$
while (more packets)
    accept one packet from upper layer
    $ack \leftarrow$ false
    while ($!ack$)
        send packet in frame to $B$ with sequence number $SN$
        wait(timeout)
        if received frame from $B$ with $RN \neq SN$
            $SN \leftarrow RN$
            $ack \leftarrow$ true
```

```
B
$RN \leftarrow 0$
while (true)
        if frame with $SN = RN$ received
            release packet to upper layer
            $RN \leftarrow (RN + 1) \bmod 2$
        with probability $q > 0$ send frame to $A$ with piggybacked $RN$
```

3.2 Throughput of stop and wait ARQ

With stop and wait ARQ, B must wait for a frame, and A must wait for an ack. Therefore, one frame is sent from A to B per approximately one RTT. Recall the definition of RTT to be the Round Trip Time for exchanging small messages, which is approximately twice the propagation delay (assuming no queuing delays). For instance, if the bandwidth of the link is 1.5 Mbps, RTT is 45 ms, and the frame size is 1 KB, then we send 1024×8 bits every $0.045 + (1024 \times 8)/(1.5 \times 10^6)$ seconds, i.e. ≈ 0.15 Mbps. This is only 10% of the link capacity. Therefore, we would like A to send up to 10 frames, or more generally up to $n \approx 1 + \frac{RTT \times Bandwidth}{frame\ size}$ frames [1], before having to wait for the first ack in order to fully utilize the link. The following figure compares the two scenarios.

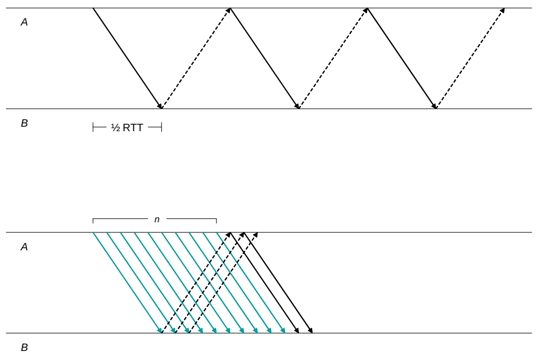


Figure 12: One frame (stop and wait) vs. n frames per RTT

4 Sliding window ARQ

Based on the previous section, we would like the sender to be ready to transmit the n^{th} frame at pretty much the same moment that the ack for the first frame arrives. Therefore, the sender keeps a *window* of frames of size n to hold frames with sequence numbers SN to $SN + n - 1$.

The sender can transmit any of these frames before receiving a frame with $RN > SN$. As before, when the sender receives a frame with $RN > SN$, it sets SN to the received RN by sliding the window, hence the name of this ARQ strategy.

[1] That's how many frames we can send during $RTT + \frac{frame\ size}{Bandwidth}$ time

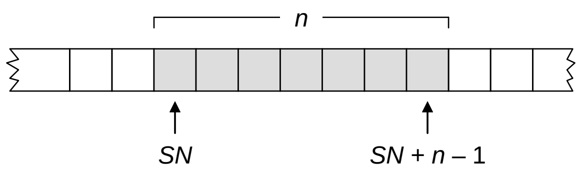


Figure 13: Sender window

Similarly, the receiver keeps a window of frames that is willing to accept (but not necessarily deliver the packet to the upper layer). Therefore, if the window size is m, the receiver can accept any frame with sequence number RN to $RN + m - 1$ before receiving a frame with $SN = RN$.

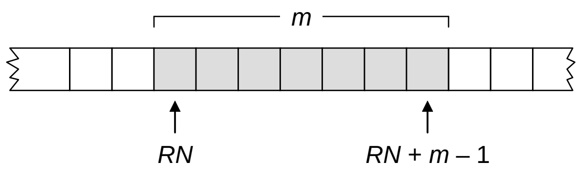


Figure 14: Receiver window

Upon receiving a frame with $SN = RN$, the receiver sets RN to $RN + r$ by sliding the window, such that all frames with sequence numbers RN to $RN + r - 1$ have been received, and delivers the corresponding packets to the upper layer. We will assume that $m \leq n$. Values of $m = 1$ (go back n) and $m = n$ (selective repeat) are most common in the literature.

Here's the algorithm for the sliding window ARQ for both the sender and the receiver:

```
$\underline{A}$
$SN \leftarrow 0$
**while** (more packets)
      accept packets from upper layer into window
      $ack \leftarrow$ false
      **while** (!$ack$)
            send packets in frames to $B$ with sequence numbers $SN$ to $SN + n - 1$
            wait(timeout)
            **if** received frame from $B$ with $RN > SN$
              $SN \leftarrow RN$ (slide the window)
              $ack \leftarrow$ true
```

```
$\underline{B}$
$RN \leftarrow 0$
**while** (true)
        **if** frame with $SN \in [RN, RN+m]$ received
           release packets $RN$ to $RN+r-1$ to upper layer such that
           all $r$ frames are received
           $RN \leftarrow RN + r$ (slide the window)
        with probability $q > 0$ send frame to $A$ with piggybacked $RN$
```

To prove the correctness of the sliding window algorithm, we have to prove both the safety and the liveness properties. The proof of safety is trivial since the receiver releases packets in order (assumeing it can always detect errors). The proof of liveness is exactly the same as for stop and wait ARQ (if fact, a slightly simpler proof could have been made for stop and wait ARQ).

4.1 Implementation using finite buffers

For simplicity of illustration, we considered a finite size window sliding on an infinite buffer of frames. At least that's how we pictured it in Figures 13 and 14 of the sliding window ARQ. In practice, however, only finite buffers are available. This sould be sufficient since at most n (or $m \leq n$) frames need to be stored at any time. Therefore, given a buffer buf of size n, the sender can store a frame with sequence number SN in $buf[SN \bmod n]$. Similarly, given a buffer buf of size m, the receiver can sotre a frame with sequence number SN in $buf[SN \bmod m]$. Note that given the rule by which the sender (receiver) accepts frames in its window, two frames with sequence numbers SN_1 and SN_2 cannot satisfy $SN_1 \bmod n = SN_2 \bmod n$ ($SN_1 \bmod m = SN_2 \bmod m$). The figure below shows some examples.

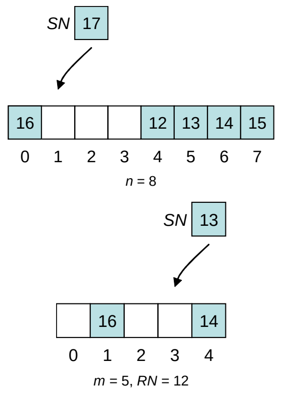


Figure 15: Buffer implementation of windows

Here's a more detailed implementation of the sliding window algorithm using this idea:

```
A
$SN \leftarrow 0$
while (more packets)
    if $buf$ not full
      accept a packet and store the new frame in the buffer
      $ack \leftarrow$ false
      while ($!ack$)
            send packets in frames to $B$ with sequence numbers $SN$ to $SN + n - 1$
            wait(timeout)
            if received a frame from $B$ with $RN > SN$
              free $buf[SN \bmod n] \ldots buf[(RN - 1) \bmod n]$
              $SN \leftarrow RN$ (slide the window)
              $ack \leftarrow$ true
```

```
B
$RN \leftarrow 0$
while (true)
    if frame with $SN \in [RN, RN + m - 1]$ received
      accept the frame and store it in $buf[SN \bmod m]$
      if $SN = RN$
        release packets $RN$ to $RN + r - 1$ to upper layer such that
        $buf[(SN + i) \bmod m]$ contains frame $SN + i$, $i = 0 \ldots r - 1$
        free $buf[SN \bmod m] \ldots buf[(SN + r) \bmod m]$
        $RN \leftarrow RN + r$ (slide the window)
    with probability $q > 0$ send a frame to $A$ with piggybacked $RN$
```

4.2 Bounded sequence numbers (again...)

Here again we suffer from unbounded sequence numbers for SN and RN. As before, assuming the FIFO property holds, we can send SN and RN modulo some integer p. For stop and wait ARQ, $p \geq 2$ was a sufficient modulus. For sliding window ARQ, we will se that p needs to be at least equal to $n + m$. Note that stop and wait ARQ is a special case of sliding window ARQ where $n = m = 1$. We start by generalizing the proposition in the previous section:

Proposition (FIFO): If B receives a frame with sequence number SN at time t, then $RN(t) - n \leq SN \leq RN(t) + n - 1$. Similarly, if A receives a frame with ack number RN at time t, then $SN(t) \leq RN \leq SN(t) + n$.

To prove the first part of the proposition, consider the time $t_0 < t$ at which the received frame SN was transmitted. We know from the algorithm that

$$SN(t_0) \leq SN \leq SN(t_0) + n - 1 \text{ (window)}$$

Furthermore,

$$SN(t_0) \leq SN(t) \leq RN(t) \text{ (non-decreasing)}$$

Finally, frame $SN(t_0)+n$ cannot have been sent before t_0, and so by the FIFO property it cannot not have been received before t:

$$RN(t) \leq SN(t_0) + n \text{ (FIFO)}$$

Putting these inequalities together, we obtain:

$$RN(t) - n \leq SN \leq RN(t) + n - 1$$

To prove the second part of the proposition, consider the time $t_0 < t$ at which the received ack RN was transmitted. Frame $SN(t) + n$ cannot have been sent before t, so it certainly cannot have been received before t_0. Therefore,

$$RN \leq SN(t) + n \text{ (window)}$$

Finaly, because of the FIFO property (think why):

$$SN(t) \leq RN \text{ (FIFO)}$$

Putting these inequalities together, we obtain:

$$SN(t) \leq RN \leq SN(t) + n$$

Based on the above proposition, the sender needs to distinguish between $n + 1$ values at any time. On the other hand, the receiver needs to dinstinguish between $2n$ values at any time. However, a closer look will reveal that the receiver will only need to distinguish between $n + m$ values at any time, because the receiver is only required to check if $SN \in [RN(t), RN(t) + m - 1]$ as illustrated in the figure below. Therefore, a modulus of $p \geq n + m$ is sufficient ($1 \leq m \leq n$).

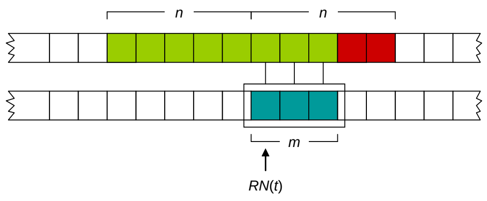


Figure 16: Ok for last $n - m$ SN numbers to wrap around modulo $p = m + n$

We will mention one last technicality about the choice of p in terms of the finite buffer implementation. Consider two sequence numbers SN_1 and SN_2. We have a problem if $(SN_1 \bmod p) \bmod m = (SN_2 \bmod p) \bmod m$, i.e. both frames should be stored in the same buffer location. Before adding the p modulus, this was not a problem because $SN_1 \bmod m = SN_2 \bmod m$ means that $|SN_1 - SN_2| \geq m$ and, therefore, one of the frames will not be accepted into the buffer. With $(SN_1 \bmod p) \bmod m = (SN_2 \bmod p) \bmod m$, it is possible that $|SN_1 - SN_2| < m$. One way to solve this problem is to choose for the value of p a multiple of m, because then $(SN \bmod p) \bmod m = SN \bmod m$. The same applies for the sender, so p should also be a multiple of n. Since we need $p \geq n + m$, we can choose $p = \max(2n, lcm(n, m))$. An alternative would be to use a circular queue implementation of the buffer and keep track of the head and tail (that's how it is practically done).

References

Dimitri Bertsekas and Robert Gallager, Data Networks
Larry Peterson and Bruce Davie, Computer Networks: A Systems Approach