# Sequence Numbers

* To distinguish between frames, we add a sequence number to each frame.
* Frame messages are of the form frame(i) where i is a sequence number.
  + frame(0) is the first data message, frame(1) is the next data message, frame(2), frame(3), frame(4), etc.
* The receiver sends acknowledgments to the sender, indicating which frames it has received.
* If the sender does not receive an acknowledgment for a message:
  + it retransmits the frame
  + uses the same sequence number it used the first time it sent this message (the sequence number of a message never changes)

# Transmission Window

* To prevent network congestion and not to over-run the receiver, there can be at most SWS outstanding messages (SWS = sender window size)
  + All frames in the range from frame(0) up to and including frame(i) must be acknowledged before the sender can send frame(i+SWS).
* E.g., if SWS = 3, and the sender sends frame(0), frame(1), and frame(2)
  + The sender waits until it receives an acknowledgment for frame(0) before sending frame(3).
    - Even if it receives an ack for frame(1), it still needs to receive an ack for frame(0) before sending frame(3).

# Acknowledgments

Keeping the Pipe Full

Receiver

Sender

Frame

Frame

Frame

Frame

Receiver

Sender

Frame

ACK

Time

Frame

ACK

Frame

ACK

Frame

ACK

**Advantages:**

* **More frames in pipe**
* **Less time overall**
* **Piggybacked ACKs**

Frame

Frame

Frame

Frame

Frame

Frame

Frame

Frame

ACK

ACK

Acknowledgments come in four "flavors"

* Cumulative Ack: ack(i) acknowledges all frame messages in the range frame(0) up to and including frame(i)
* Individual Ack: ack(i) acknowledges only frame(i)
* Negative Ack: ack(i) indicates that frame(i) was NOT received
* Block Ack: ack(i,j) acknowledges frame(i) up to and including frame(j)

We will see that each of these makes different assumptions about the behavior of the channels.

We will cover only cumulative and individual ack.

# Cumulative Acknowledgment

* Frame messages are of the form

frame(i)

where i is an integer (i.e. unbounded) sequence number.

* Acknowledgments are of the form

ack(i)

where i is an integer (i.e. unbounded) sequence number.

* ack(i) acknowledges from frame(0) up to and including frame(i).

# Sender Variables

* Two important variables in the sender process:

1. Largest Frame Sent (**LFS**):
   * sequence numbers in 0 .. LFS have been sent by sender,
2. Largest Acknowledgement Received (**LAR**)
   * sender has received an acknowledgement for frame messages 0 .. LAR.
   * Always: LAR ≤ LFS, why?

* Therefore, for an arbitrary seq. no. i:

1. if i ≤ LAR, then   
    an acknowledgment for frame(i) has been received by sender
2. if LAR < i ≤ LFS, then  
    frame(i) has been sent but, its ack has not been received
3. if i > LFS, then  
    frame(i) has not been sent.

# Receiver variables

* Receiver process has two variables/constants:

1. Next Frame Expected (**NFE**)
   * frame(0) up to frame(NFE - 1) have been received by receiver.
2. Receive Window Size (**RWS**) : out-of-order packets buffer size (often = SWS)
   * Last Frame Acceptable(**LFA**) : LFA **=** NFE + RWS - 1

* Note that frame(NFE) has not been received yet, since otherwise we would have incremented NFE.
* When frame(NFE) is received, it is given to the application, and NFE is increased.
* If frame(i), where i > NFE, is received, then store it only if i ≤ LFA.

# Receiver Variables (continued)

Therefore, for an arbitrary seq no i:

1. if i < NFE, then frame(i) has been received
2. if i = NFE, then frame(i) has not been received
3. if NFE < i ≤ LFA, then frame(i) *may have*  been received



SWS

LAR

LFS

…

…



RWS

NFE

LFA

…

…

# Variable Relationships



**Window Limits Transmission**

**Remarks**

* The whole point of using cumulative acknowledgments is that the receiver can wait to receive a few frame messages and then send one acknowledgment for all of them.
* This reduces the number of acknowledgments in the network and thus reduces congestion.
* Note that when sender sends frame(2), it can no longer send any more messages (until it receives the next ack) because at this point LFS = LAR + W.
* Notice that the receiver always sends ack(NFE-1) to the sender.
  + Sending ack(i) where i < NFE - 1 would not convey as much information as NFE - 1.

**Dealing with Packet Loss**

Assume SWS = 2 = RWS

# Remarks

What would happen if RWS = 1?

What is happening here?

**receiver**

**sender**

LAR=2

ack(2)

**LOST!**

ack(2)

LFS=2

frame(1)

LFS=1

LAR=0

ack(0)

LFS=0

NFE = 3

NFE = 1

frame(1)

frame(2)

frame(0)

* Here, ack(2) is lost.
* After a timeout period, frame(LAR+1), i.e., frame(1), is retransmitted.
* This causes the receiver to send another copy of the ack(NFE-1), i.e. ack(2) to the sender.
* The sender then can set LAR to 2.
* When timing out, should the sender retransmit everything from LAR+1 up to LFS, or just retransmit LAR+1?
  + Both things will work, but the general consensus is just to retransmit LAR+1.

**process** sender

**constant** SWS

**variables**

LAR, LFS : **integer {**initially -1}

i : **integer**

**begin**

**event:** LFS < LAR + SWS

**resp:** LFS := LFS + 1;

**send** frame(LFS) **to** receiver

**event:** **receive** ack(i) **from** receiver

**resp:** LAR := max(LAR, i)

**event:** LAR < LFS

**resp: send** frame(LAR + 1) **to receiver**

**end**

# Remarks

Obviously, you don't want to retransmit as often as you want (this would cause congestion).

However, for the correctness of the protocol, i.e., that data is delivered without errors, it does not require a timeout.

For efficiency, of course, we should retransmit only after we wait enough time to be quite certain (although not guaranteed) that the old copy of the message is dead.

If sequence numbers are **unbounded**, then the above works for any channel with any properties (such as reorder, duplication, and loss)

# Receiver Variables

* The receiver has the following additional variables

1. rcvd : **array**[0 .. RWS - 1] **of** **boolean** {buffer}
2. akn : **boolean** {must return an ack soon}

* rcvd is a buffer of size RWS containing messages received out of order.
* If received frame(i), where i ≥ NFE, it sets rcvd[i mod RWS] := **true**.
* If rcvd[NFE mod RWS] = **true**, then the receiver has received sequence number NFE.
  + It sets rcvd[NFE mod RWS] to false (i.e., removes message from buffer and gives it to the application).
  + It also increases NFE to the next sequence number

# Receiver Variables (continued)

* akn, indicates if it must send an ack to within the next few seconds.
* akn is set to true when either:

1. an old message is received again, since this means the old ack may have been lost
2. whenever we increase NFE, since now an ack with a larger sequence number than before can be sent

**process** receiver

**constant** RWS

**variables**

NFE : **integer** {initially 0} {NOTE: I am assuming LFA = NFE + RWS}

j : **integer**

rcvd : **array** [0 .. RWS-1] **of boolean**

akn : **boolean**

**begin**

**event: receive** frame(j) **from** sender

**resp:**

**if** j < NFE **then** akn **:= true** {old message}

**elseif** j ≥ NFE + RWS **then** **skip** {no buffer space}

**elseif** NFE ≤ j < NFE + RWS **then**

rcvd[j **mod** RWS] := **true;** {place data in buffer}

**while**  rcvd[NFE **mod** RWS] **do**

{deliver frame(NFE) }

rcvd[NFE **mod** RWS] := **false;**

NFE := NFE + 1;

akn := **true**

**end while**

**end if**

**event:** akn

**resp:**

**send** ack(NFE-1) **to** sender;

akn := **false**

**end**

# How often to execute the above action?

# In principle, for correctness it does not matter ☺

# For efficiency, the standard rule is

# send an ack for every two frames received

# if only one frame received, wait ½ sec. then send the ack Bounded Sequence Numbers problem

* It is important to note that the cumulative acknowledgment protocol does not work with bounded sequence numbers, i.e., the sequence numbers must be unbounded integers, if reorder is possible in the channel.
* Consider the following scenario, where the sequence numbers are in the range 0 .. r - 1 for some r.

|  |  |
| --- | --- |
| ack(2)  **ack(3)**  receiver  sender  **LOST!**  etc.,  etc.,  etc.  ack(4)  ack(5)  frame(3)  frame(3)  frame(2)  frame(5)  frame(4) | * Seq. no's. are in the range 0 .. r-1 * Assume we are already at seq. no. 3 and 4. * Since the sender receives ack(4), this acknowledges **both** frame(3) and frame(4). * As time progresses we run out of sequence numbers (they are in the rage 0 .. r-1 for some constant r) so we have to wrap the sequence numbers and continue with zero. * When we send frame(3) (which is NOT the same frame(3) as before) frame(3) gets lost. * However, an OLD ack(3) which was still in the channel shows up at this time (due to some heavy reorder in the channel), and sender thinks frame(3) was received when it was not!!! * Note that this happens even if the timeout is accurate (no unnecessary rexmissions) |

# Bounded Sequence numbers in practice

* In real life, sequence numbers are bounded, and channels (end-to-end channels) do reorder messages.
* Why then we don’t have the problem above?
  + You limit the speed of the sender
  + Sequence numbers should be very large so that by the time you reuse a sequence number, any other frame or ack with the same sequence number no longer exist in the channel (due to limited lifetime of messages).
  + Whether the timeout is accurate or not does not matter.

# Bounded Sequence Numbers

Sequence numbers are in the range 0 .. r - 1 for some r,

We assume a small r

In sender process:

LAR, LFS : 0 .. r - 1

In receiver process:

NFE : 0 .. r - 1

These variables are increased using mod r, i.e., LFS := LFS +r 1

Therefore, the relationship LAR+1 ≤ NFE ≤ LFS + 1 no longer holds.

This is because sequence numbers are bounded.

# Example

For example, assume LAR = r - 2, NFE = r - 1, LFS = r - 1.

Then, sender sends a new message.

New LFS value, LFS := (LFS + 1) mod r = 0

Thus, LFS = 0, NFE = r - 1, which violates NFE ≤ LFS + 1

Thus, we define a new relationship, within(a, b, c) where 0 ≤ a, b, c ≤ r - 1

Informally, within(a, b, c) = true, means, if we begin from a, and continue adding 1 (modulo r), and stop when we reach c, then along the way we obtained the value of b.

# Example

For example, if r = 20, a = 5, b = 8, and c = 3, then within(a,b,c) is true.

0 1 2 3 4 5 6 7 8 9 . . . 19

c a b

------------------

-------->

However, if r = 20, a = 8, b = 5, and c = 3, then within(a, b, c) is false.

0 1 2 3 4 5 6 7 8 9 . . . 19

c b a

------------

-------->

# Variable Relationship

At all times:

within(LAR+r1, NFE, LFS+r1) AND within(LAR, LFS, LAR+rSWS)

This is similar to what we had before, except that instead of using ≤ we use "within"

# Individual Ack

Individual ack with unbounded sequence numbers is quite easy, so I will not discuss it here.

We use bounded sequence numbers, in the range 0 . . r - 1 for some constant r. We assume r is small.

Channel Requirements:

* The channels should not duplicate messages
* The channels may reorder messages
* An accurate timeout is necessary. (you can eliminate the need for an accurate timeout is there is no reorder in the channel, or if r is very big)

By accurate timeout, we mean the following:

* There is a time to live for each message in the channel (messages have bounded lifetimes in a channel)
* Thus, you can determine how long to wait before you are sure that the message and its acknowledgment are lost (no longer on either channel).

# Variable Relationship

At all times:

within(LAR+r1, NFE, LFS+r1) AND within(LAR, LFS, LAR+rSWS)

(as mentioned above)

To ensure that we don't have "old" messages floating around in the channel, we must ensure that at all times, for all sequence numbers i,

frame(i)#ch.sender.receiver + ack(i)#ch.receiver.sender ≤ 1

I.e., either the frame, or its ack, can be in the channels, not both at the same time, and certainly no duplicates of any message.

W=3 and r = 6, and initially NFE=0 and LFS = LAR = r-1

|  |  |
| --- | --- |
|  | * Note that, for every frame, an ack is immediatelygenerated, as opposed to the cumulative ack, in which sending an ack can be delayed until receiving several frames. * Since ack(0) is lost, eventually sender retransmits data(0). * Also, note that LAR cannot be increased beyond r-1 until ack(0) is received. |

Below, recall that (a +r b) = (a + b) mod r

**process** sender

**constants**

SWS, r {2\*SWS ≤ r}

**var**

LAR, LFS, i : 0 .. r - 1,

ackd : **array** [0.. r - 1] **of boolean**

**param**

k : 0 .. r-1

**begin**

**event:**  LAR +r SWS ≠ LFS **then**

**resp:**

LFS := LFS +r 1;

**send** frame(LFS) **to** receiver

**event: receive** ack(i) **from** receiver

**resp:**

ackd[i] := **true**;

**while** ackd[LAR+r1] **do**

ackd[LAR+r1]:= **false**;

LAR := LAR+ r1

**end while**

**event:**

**timeout** LAR ≠ LFS ∧ within(LAR + r 1, k, LFS) ∧ ¬ackd[k] ∧

(frame(k)#ch.sender.receiver + ack(k)#ch.receiver.sender = 0)

**resp:**

**send** frame(k) **to** receiver

**end**

# Receiver

The receiver has a rcvd array where it stores which sequence numbers it has received out of order.

rcvd : **array** [ 0 . . RWS-1 ] **of boolean** {seq #'s received}

It is similar to the buffer we used in the cumulative ack protocol

We also need a pointer into rcvd indicating the position of frame(NFE) in this array

We will call this pointer p (range 0 .. RWS – 1)

1. **process** receiver
2. **constants** RWS, r
3. **variables**
4. NFE, j : 0 .. r - 1
5. rcvd : **array** [0 .. RWS - 1] **of boolean;** {initially all false}
6. p : 0 .. RWS -1 {pointer into rcvd indicating the location of frame(NFE)}
7. **begin**

**event: rcv** frame(j) **from** sender

**resp:**

**if** **within**(NFE -r  SWS, j, NFE -r 1) **then**

{ received a retransmitted message }

**send** ack(j) **to** sender

**elseif** **within**(NFE, j, (NFE +r (RWS-1)) **then**

{ received a new message }

**send** ack(j) **to** sender;

rcvd[p + RWS apart(j, NFE, r)] := **true**;

**while** rcvd[p] **do**

{deliver data(NFE) to application}

rcvd[p] := **false**; p := p +RWS 1; NFE := NFE r +1

**end while**

**else**

**skip** {not enough buffer space }

**endif**

**end**

We only have one action that receives data. In this action, q checks the sequence number.

If the sequence number is in the range within(NFE -r  SWS, j, NFE -r 1), then the message has been received before, but the ack for it got lost, so we send another acknowledgment for it.

If the sequence number is in the range within(NFE, j, (NFE +r (RWS-1)), then the message is new, and:

* We send an ack for this message.
* We mark it in the buffer as being received.
* NFE is moved forward if possible, and as far as possible.
* As we move NFE forward, we empty the buffer location and give the message to our application.
* apart(j, NFE, r) is simply the number of steps in the sequence number circle (ie. 0 .. r-1) from NFE to j, if j = NFE, then apart(j,NFE,r) = 0, and if j = NFE+ r 1, then apart(j,NFE,r) = 1

# Old vs New Messages

* Note that no message is “left behind”, that is, if there is a frame(i) in the channel from sender to receiver, it must be that

within(LFA+r1, i, LFS)

* Also, if there is an ack(i) in the channel from receiver to sender, it must be that

within(LFA+r1, i, LFS)

* Thus, ***all messages in either channel*** must fall within that range.
* Recall that within(LFA+r1, NFE, LFS+r1).
  + Thus, there is flexibility of where NFE could be in relation to LFA and LFS
  + We will look at both ends of the spectrum:
    - when LFA+ r1 = NFE, and
    - when NFE = LFS+r 1

# Old vs New Messages (continued …)

* Consider one end of the spectrum (LFA+ r1 = NFE), i.e.,

LFA …………………………………. LFS

NFE

In this case, the sender has sent frames LFA+1 … LFS and the receiver has not received any of them. All of these are “new” messages.

Recall also that LFS can be at most SWS away from LFA

* Therefore, if frame(i) arrives at the receiver and it is “new”, it will be in the range

within(NFE, i, NFE +r SWS-1)

(see picture in next slide …)



# Old vs New Messages (continued …)

* Consider the other end of the spectrum (NFE = LFS+ r1), i.e.,

LFA …………………………………. LFS

NFE

In this case, the sender has sent frames LFA+1 … LFS and the receiver has received ALL of them. If the receiver sees them again, then these are “old” messages.

Recall again that LFS can be at most SWS away from LFA

* Therefore, if frame(i) arrives at the receiver and it is “old”, it will be in the range

within(NFE- r SWS, i, NFE -1)

(see picture in next slide …)





* These two ranges cannot overlap, since otherwise receiver does not know if a message falling in the overlapping portion of the range is a new message or a retransmitted message.
* A message in the overlapping range will be considered old (first branch of “if” statement)
  + The receiver will throw it away,
  + and sends an ack for it
* Note that if the message is actually new, the sender receives an ack for a message that was thrown away.
* Hence, there is a deadlock:
  + The sender never sends that message again
  + NFE can never increase beyond this value.

# Minimum Value of r

How small can r be?

Hence, to distinguish between retransmitted messages and new messages, we must have:

r ≥ 2\*SWS

We did not consider different values of RWS, but this does not affect the argument above.

You could try modifying the “if” statement to first consider new messages (i.e., the overlapping ones would default to new not old).

I doubt it makes any difference. Most likely you end up delivering an old message as new.

Another issue: why must we have always that

frame(i)#ch.sender.receiver + ack(i)#ch.receiver.sender ≤ 1 ?

Consider the following scenario, which retransmits too soon.

|  |  |
| --- | --- |
|  | Data(3) is sent two times (retransmit too early)  One copy gets reordered in the channel.  Sequence numbers eventually wrap around, and a new data(3) (different from the old one) is sent.  The new data(3) is lost, and the old data(3) is received.  The receiver incorrectly delivers the old data(3) for the new data(3) |

The above problem does not occur if reorder is not allowed

* i.e. an accurate timeout is not needed if there is no message reorder, even though the sequence numbers are small (recall stop and wait, this is similar)

What if there is reorder in the channel, but the sequence numbers are huge????

* Then an accurate timeout is not needed

# Cum Ack with Bounded Seq Nos.

Sequence numbers are in the range 0 .. r - 1 for some r, we assume a small r

Thus, the channel **should not reorder messages**

* i.e., cum ack with small sequence numbers and message reorder DOES NOT work (we showed this earlier!)
* note: indiv. ack with small sequence numbers and message reorder DOES work.

At all times:

within(LAR+r1, NFE, LFS+r1) AND within(LAR, LFS, LAR+rSWS)

Again, we do not need an accurate timeout, **provided the channel does not reorder message**s.

# Cum Ack with Bounded Seq Nos.

# To prevent confusion at the receiver,

r ≥ SWS + **RWS**

# An accurate timeout is not necessary (as far as I can tell!)

# You can retransmit either LAR+1 or the entire window up to LFS, and it still works (again, as far as I can tell!)

# RWS ?

* If RWS ≥ SWS, then we have r ≥ SWS+RWS ≥ 2\*SWS, and the behavior is similar to before.
* If RWS < SWS, and r ≥ SWS + RWS, is this good enough? Yes!



A “new” message in the overlapping interval will be thrown away (not nice but ok), the ack sent, however, will be NFE -1, and thus the message is not acked (which is ok)

An “old” message in the overlapping interval will also be thrown away (which is ok) and the next ack to be sent is NFE – 1 (which is also ok)