



Computer Networking

Reliable Transport

Prof. Andrzej Duda
duda@imag.fr

<http://duda.imag.fr>

Reliable Transport

Reliable data transfer

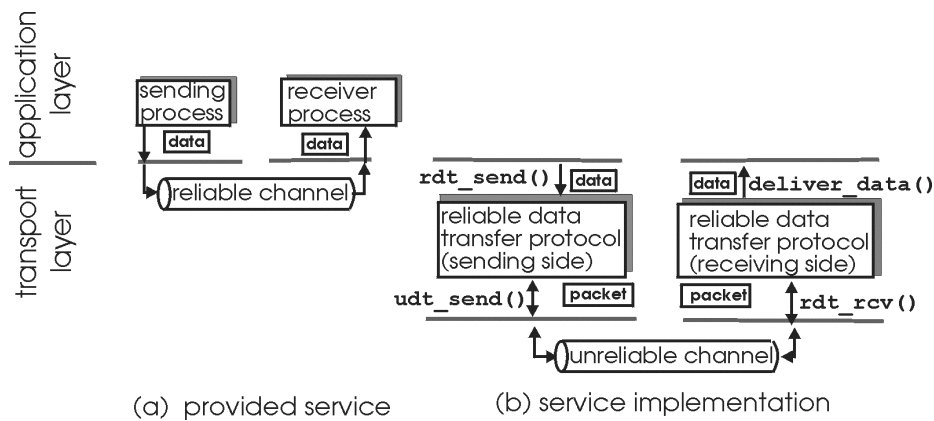
- Data are received ordered and error-free
- Elements of procedure usually means the set of following functions
 - Error detection and correction (e.g. ARQ)
 - Flow Control

Automatic Repeat reQuest

- Sliding window
- Error and Loss detection
- Acknowledgements: short control packets
- Retransmission Strategies
 - Stop & Go
 - Go Back N
 - Selective Repeat

Principles of reliable data transfer

- important in app., transport, link layers
- data are received ordered and error-free



- characteristics of unreliable channel will determine complexity of reliable data transfer protocol (rdt)

3

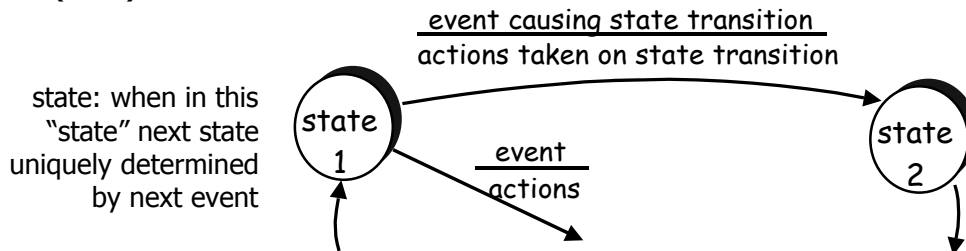
Reliable data transfer means that the data are received ordered and error-free. The problem of implementing reliable data transfer occurs not only at the transport layer, but also at the link layer and the application layer as well. The service abstraction provided to the upper layer entities is that of a reliable channel, where no transferred data bits are corrupted or lost, and all are delivered in the order in which they were sent. This is precisely the service model offered by TCP to the Internet applications that invoke it.

It is the responsibility of a **reliable data transfer protocol** to implement this service abstraction. This task is made difficult by the fact that the layer *below* the reliable data transfer protocol may be unreliable. For example, TCP is a reliable data transfer protocol that is implemented on top of an unreliable (IP) end-end network layer.

Reliable data transfer

Our approach:

- analyze reliable data transfer protocol (rdt)
- use finite state machines (FSM) to specify the model
- introduce gradually the Elements of Procedure (EoP)



4

We analyze the main elements of a reliable data transfer protocol, using finite state machine to specify the models of the sender and the receiver. The generic term "packet" is seen as more appropriate, and used here rather than "segment" for the protocol data unit, given that the concepts of a reliable data transfer protocol applies to computer networks in general. We talk about Elements of Procedure (EoP) as the elements for obtaining a reliable data transfer that are not limited to transport layer, but exist in general in Communication Networks.

Elements of Procedure

- Elements of Procedure transform a raw, unreliable frame transport mechanism into a reliable data pipe
 - ordered delivery of packets
 - loss-free as long as the connection is alive
 - no buffer overflow
- Elements of procedure usually means the set of following functions
 - Error detection and correction (e.g. ARQ - Automatic Repeat reQuest)
 - Flow Control
 - Connection Management
- Elements of procedure exist in
 - reliable transport (TCP for the TCP/IP architecture) (layer 4)
 - also: in reliable data link (ex: over radio channels, over modems -layer 2) as HDLC
- Congestion Control

5

The Elements of Procedure (EoP) transform a raw, unreliable frame transport mechanism into a reliable data pipe by delivering the packets in order and without losses, and avoiding buffer overflow. With EoP is usually addressed three following functions:

•**Error detection and correction** Various ways exists to carry out this process. However, techniques based on checks (parity) bits are not generally adopted for data networking, because they need too many bits to ensure the level of error correction required. Instead, **ARQ** (Automatic Repeat reQuest) **protocols** - reliable data transfer protocols based on retransmission – has been universally adopted.

•**Flow-Control** to eliminate the possibility of the sender overflowing the receiver's buffer. Flow control is thus a speed matching service--matching the rate at which the sender is sending to the rate at which the receiving application is reading.

•**Connection Management:** reliable data transfer must be connection oriented. In order to control the connection two phases are needed: connection setup and connection release.

Applications that use UDP must implement some form of equivalent control, if the application cares about not loosing data. UDP applications (like NFS, DNS) typically send data blocks and expect a response, which can be used as an application layer acknowledgement.

Finally, as is the case of TCP, we can also have congestion control to avoid network congestion.

ACK/NAK handling

- ACKs & NAKs: short control packets
 - cumulative versus selective
 - positive (ACK) versus negative (NAK)
- Stop and wait
 - The sender waits for an ACK/NAK
 - Only one packet at time can be transmitted
- Go back N
 - packets are transmitted without waiting for an ACK
 - All following packets are resent on receipt of NAK
- Selective repeat procedure
 - Only packets negatively acknowledged are resent

6

There are a number of ways of handling the response to ACK/NAK. The three most commonly identified procedures are the following:

- Stop and wait: The sender can send only one packet at time and have to wait for an ACK/NAK. If a NAK is received or a timeout expires the packet is retransmitted. If an ACK arrives the current packet is dropped by the sender buffer.
- Go back N: The packets are transmitted without waiting for an ACK. If a NAK is received or a timeout expires the packet in question and all the packets following are retransmitted.
- Selective repeat procedure Only packets negatively acknowledged, or for which the timeout has expired without receiving an ACK, are resent.

Retransmission Strategies

- underlying channel can garble and lose packets (data or ACKs)
 - checksum, seq. #, ACKs, retransmissions will be of help, but not enough
- to deal with loss & errors:
 - sender waits until certain time, then retransmits
 - duplicate
- Approach: sender waits "reasonable" amount of time for ACK
- retransmits if no ACK or NAK received in this time
- if pkt (or ACK) just delayed (not lost):
 - retransmission will be duplicate, but use of seq. #'s already handles this
 - receiver must specify seq # of pkt being ACKed
- requires countdown timer

7

If the underlying channel can *lose* packets as well, two additional concerns must now be addressed by the protocol: how to detect packet loss and what to do when packet loss occurs. This can be done with the use of checksumming, sequence numbers, ACK packets, and retransmissions.

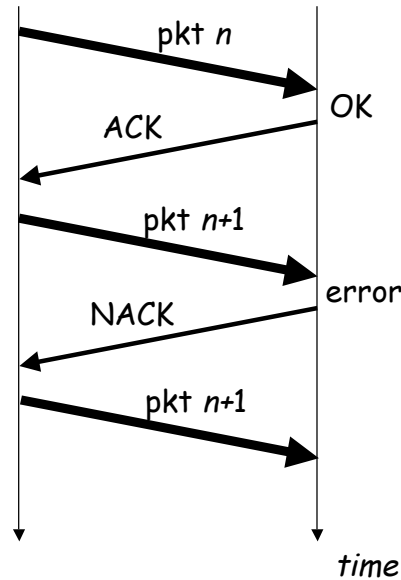
To deal with loss, the sender could wait long enough so that it is *certain* that a packet has been lost, and then it can simply retransmit the data packet. However, cannot wait forever.

The sender must clearly wait **at least as long as** a round-trip delay between the sender and receiver (which may include buffering at intermediate routers or gateways) plus whatever amount of time is needed to process a packet at the receiver. In many networks, this worst-case maximum delay is very difficult to even estimate, much less know with certainty. Moreover, the protocol should ideally recover from packet loss as soon as possible; waiting for a worst-case delay could mean a long wait until error recovery is initiated. The approach thus adopted in practice is for the sender to "judiciously" choose a time value such that packet loss is likely, although not guaranteed, to have happened. If an ACK is not received within this time, the packet is retransmitted. Note that if a packet experiences a particularly large delay, the sender may retransmit the packet even though neither the data packet nor its ACK have been lost. This introduces the possibility of duplicate data packets in the sender-to-receiver channel, that we already see how to handle.

From the sender's viewpoint, retransmission is a panacea. The sender does not know whether a data packet was lost, an ACK was lost, or if the packet or ACK was simply overly delayed. In all cases, the action is the same: retransmit. In order to implement a time-based retransmission mechanism, a **countdown timer** will be needed that can interrupt the sender after a given amount of timer has expired. The sender will thus need to be able to (1) start the timer each time a packet (either a first-time packet, or a retransmission) is sent, (2) respond to a timer interrupt (taking appropriate actions), and (3) stop the timer.

Send and Wait (simple)

- Reliable transmission
- Flow control
 - sender can send the next packet after receiving ACK
- No losses



8

The send side has two states. In one state, the send-side protocol is waiting for data to be passed down from the upper layer. In the other state, the sender protocol is waiting for an ACK or a NAK packet from the receiver.

It is important to note that when the receiver is in the wait-for-ACK-or-NAK state, it *cannot* get more data from the upper layer; that will only happen after the sender receives an ACK and leaves this state. Thus, the sender will not send a new piece of data until it is sure that the receiver has correctly received the current packet. Because of this behavior, protocols such as this one are known as **stop-and-wait** protocols.

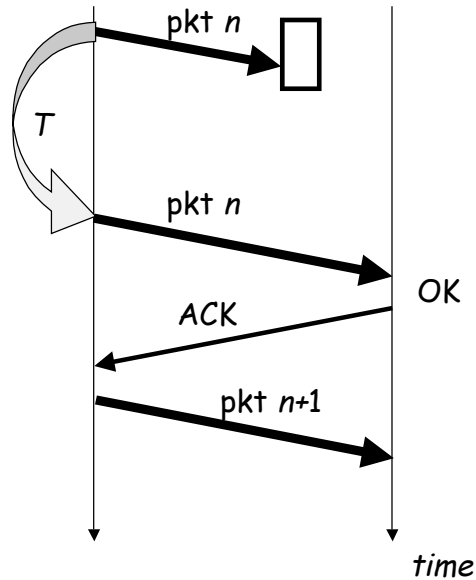
The receiver-side FSM has a single state. On packet arrival, the receiver replies with either an ACK or a NAK, depending on whether or not the received packet is corrupted.

There are two limitations with Stop and Wait:

- ACK and NAK can get corrupted. A solution is to introduce sequence numbers.
- a little efficiency because of idle periods, when the bandwidth delay product is not negligible. A solution is to allow multiple transmissions, which in turn requires a sliding window in order to avoid unlimited buffer at the destination.

Losses

- Timer
 - if no response within a time interval, retransmission
 - the time interval must be longer than RTT



9

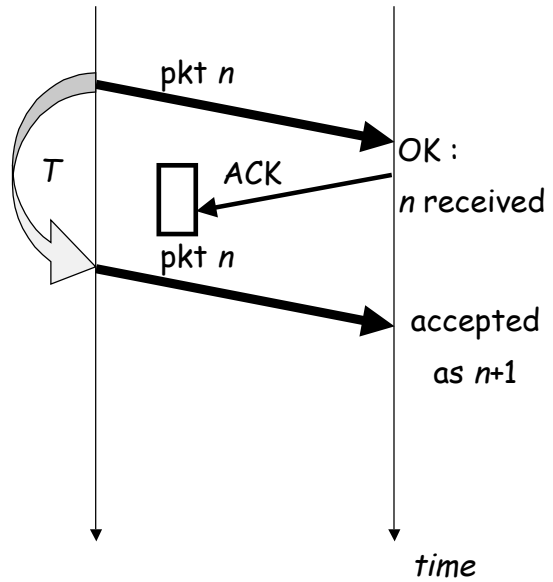
Minimally, we will need to add checksum bits to ACK/NAK packets in order to detect such errors. The more difficult question is how the protocol should recover from errors in ACK or NAK packets. The difficulty here is that if an ACK or NAK is corrupted, the sender has no way of knowing whether or not the receiver has correctly received the last piece of transmitted data. If we add a new message (e.g. "What did you say?") even this message can get lost, and we are at the same point. If we just retransmit we could introduce **duplicate packets** into the sender-to-receiver channel. The fundamental difficulty with duplicate packets is that the receiver doesn't know whether the ACK or NAK it last sent was received correctly at the sender. Thus, it cannot know a priori whether an arriving packet contains new data or is a retransmission!

A simple solution to this new problem (and one adopted in almost all existing data-transfer protocols including TCP) is to add a new field to the data packet and have the sender number its data packets by putting a **flag** (sometimes referred to as **sequence number**) into this field. The receiver then need only check this sequence number to determine whether or not the received packet is a retransmission.

The existence of sender-generated duplicate packets and packet (data, ACK) loss also complicates the sender's processing of any ACK packet it receives. If an ACK is received, how is the sender to know if it was sent by the receiver in response to its (sender's) own most recently transmitted packet, or is a delayed ACK sent in response to an earlier transmission of a different data packet? The solution to this dilemma is to augment the ACK packet with an **acknowledgment field**. When the receiver generates an ACK, it will copy the sequence number of the data packet being acknowledged into this acknowledgment field. By examining the contents of the acknowledgment field, the sender can determine the sequence number of the packet being positively acknowledged.

Problem of duplicates

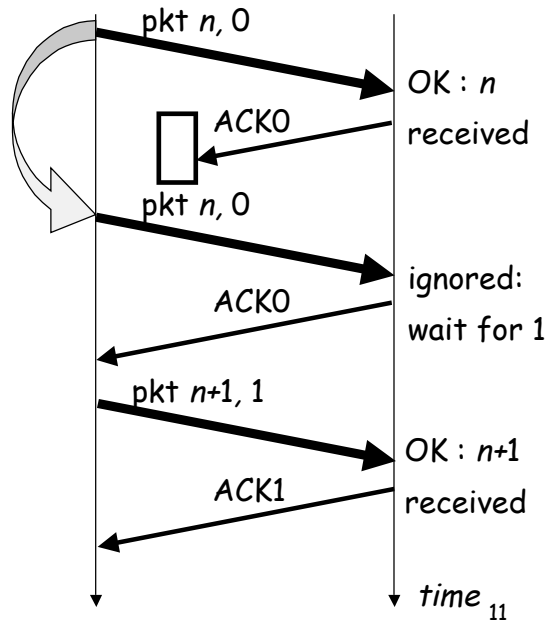
- Retransmission
 - next and retransmitted packets are confused
- Solution
 - sequence number



10

Sequence numbers

- Numbers of packets and ACKs
 - counter on k bits - mod 2^k
 - if block out of sequence, ignored
 - 1 bit is sufficient



Performance

- **Question:** What is the maximum throughput assuming that there are no losses?

notation:

- packet length = L , constant (in bits)
- acknowledgement length = l , constant
- channel bit rate = b
- propagation = D
- processing time = 0

12

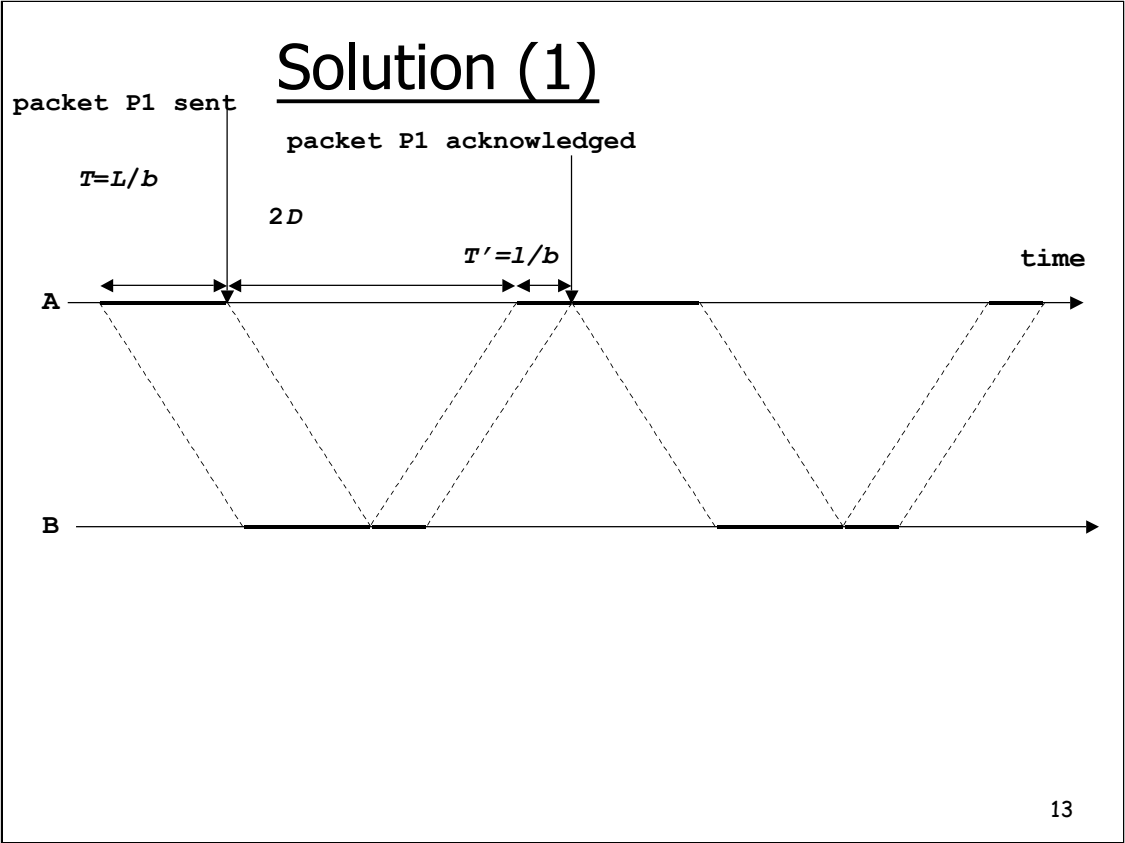
Stop and Wait protocol based are functionally correct protocols, but it is unlikely that anyone would be happy with its performance, particularly in today's high-speed networks.

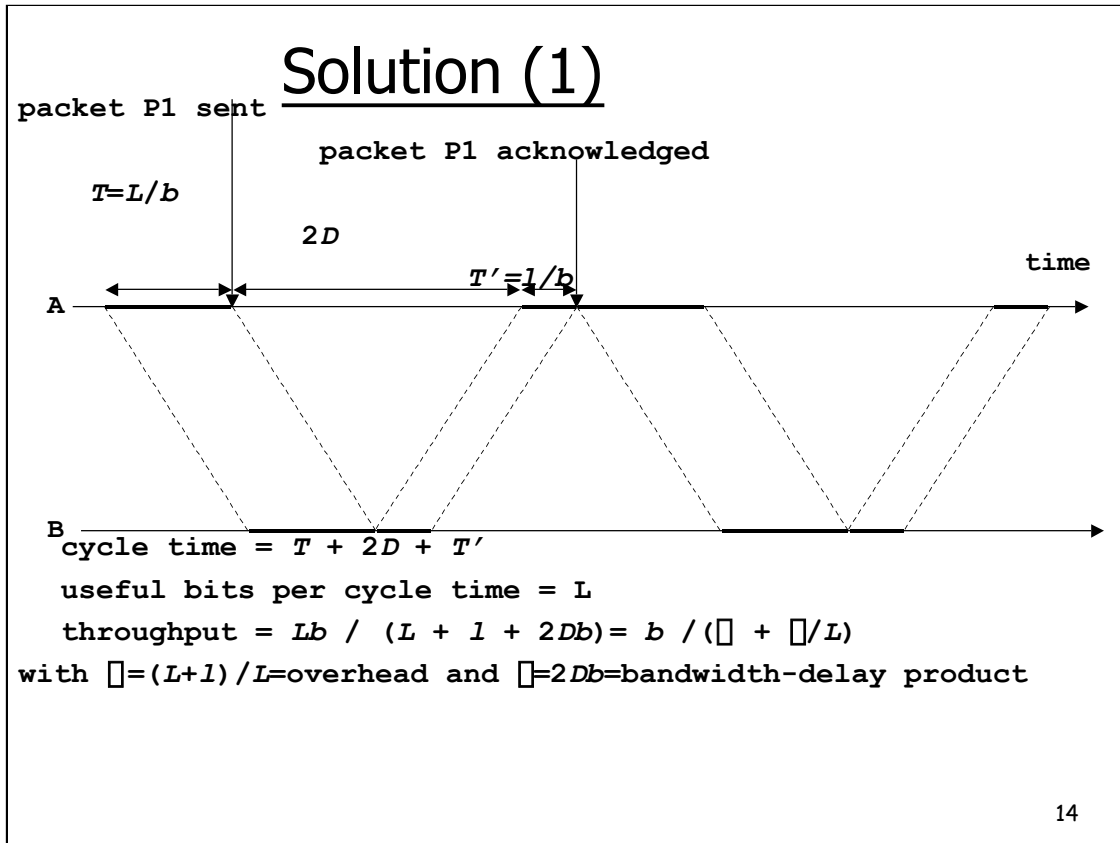
To appreciate the performance impact of this stop-and-wait behavior, consider an idealized case of two end hosts, one located on the West Coast of the United States and the other located on the East Coast. The speed-of-light propagation delay, T_{prop} , between these two end systems is approximately 15 milliseconds. Suppose that they are connected by a channel with a capacity, C , of 1 gigabit (10^9 bits) per second. With a packet size, SP , of 1 Kbytes per packet including both header fields and data, the time needed to actually transmit the packet into the 1Gbps link is

With our stop-and-wait protocol, if the sender begins sending the packet at $t = 0$, then at $t = 8$ microseconds, the last bit enters the channel at the sender side. The packet then makes its 15-msec cross-country journey, with the last bit of the packet emerging at the receiver at $t = 15.008$ msec.

Assuming for simplicity that ACK packets are the same size as data packets and that the receiver can begin sending an ACK packet as soon as the last bit of a data packet is received, the last bit of the ACK packet emerges back at the receiver at $t = 30.016$ msec. Thus, in 30.016 msec, the sender was busy (sending or receiving) for only 0.016 msec. If we define the **utilization** of the sender (or the channel) as the fraction of time the sender is actually busy sending bits via the channel, we have a rather dismal sender utilization, U_{sender} .

That is, the sender was busy only 1.5 hundredths of one percent of the time. Viewed another way, the sender was only able to send 1 kilobyte in 30.016 milliseconds, an effective throughput of only 33 kilobytes per second--even though a 1 gigabit per second link was available.

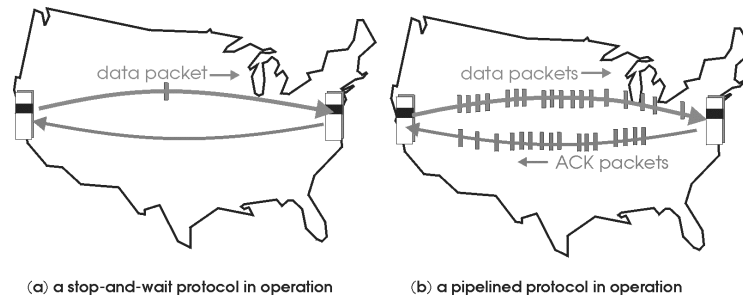




Pipelined protocols

Pipelining: sender allows multiple, "in-flight", yet-to-be-acknowledged pkts

- range of sequence numbers must be increased
- buffering at sender and/or receiver



- Two generic forms of pipelined protocols: *go-Back-N*, *selective repeat*

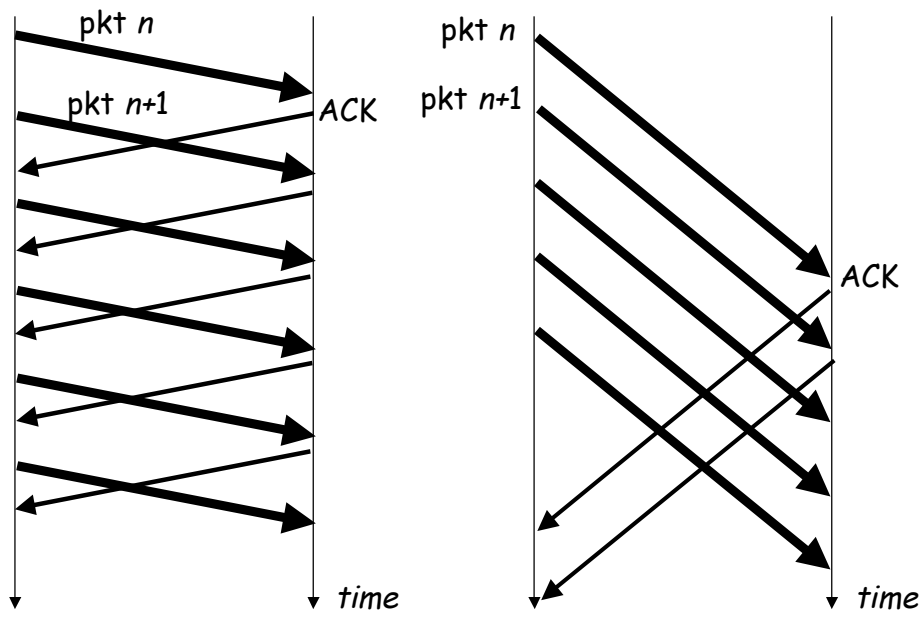
15

The solution to this particular performance problem is a simple one: rather than operate in a stop-and-wait manner, the sender is allowed to send multiple packets without waiting for acknowledgments, as shown in Figure 3.17(b). Since the many in-transit sender-to-receiver packets can be visualized as filling a pipeline, this technique is known as **pipelining**. Pipelining has several consequences for reliable data transfer protocols:

- The range of sequence numbers must be increased, since each in-transit packet (not counting retransmissions) must have a unique sequence number and there may be multiple, in-transit, unacknowledged packets.
- The sender and receiver sides of the protocols may have to buffer more than one packet. Minimally, the sender will have to buffer packets that have been transmitted, but not yet acknowledged. Buffering of correctly received packets may also be needed at the receiver, as discussed below.

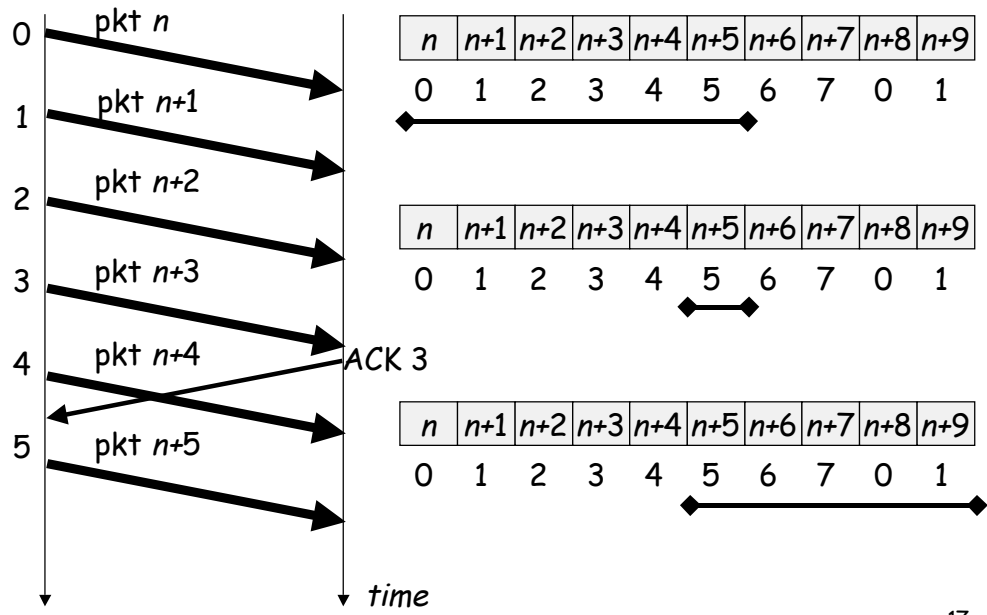
The range of sequence numbers needed and the buffering requirements will depend on the manner in which a data transfer protocol responds to lost, corrupted, and overly delayed packets. Two basic approaches toward pipelined error recovery can be identified: **Go-Back-N** and **selective repeat**. Both these protocols are based on the principle of **sliding window**.

Window size

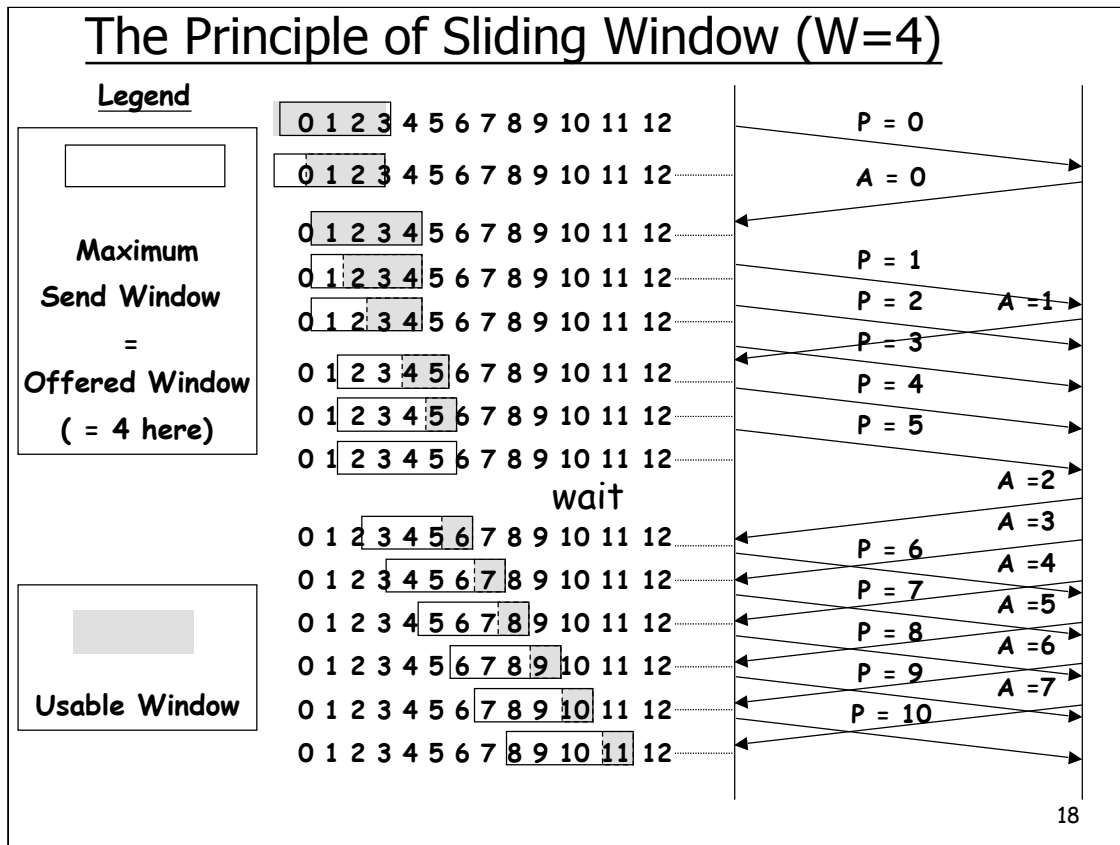


16

Sender window



17



The sliding window principle works as follows (on the example, packets are numbered 0, 1, 2, ..):

- a window size W is defined. On this example it is fixed. In general, it can vary based on messages sent by the receiver. The sliding window principles requires that, at any time: number of unacknowledged packets at the receiver $\leq W$
- the *maximum send window*, also called *offered window* is the set of packet numbers for packets that either have been sent but are not (yet) acknowledged or have not been sent but may be sent.
- the *usable window* is the set of packet numbers for packets that may be sent for the first time. The usable window is always contained in the maximum send window.
- the lower bound of the maximum send window is the smallest packet number that has been sent and not acknowledged
- the maximum window *slides* (moves to the right) if the acknowledgement for the packet with the lowest number in the window is received

A sliding window protocol is a protocol that uses the sliding window principle. With a sliding window protocol, W is the maximum number of packets that the receiver needs to buffer in the resequencing (= receive) buffer.

If there are no losses, a sliding window protocol can have a throughput of 100% of link rate (if overhead is not accounted for) if the window size satisfies: $W \geq b / L$, where b is the bandwidth delay product, and L the packet size. Counted in bytes, this means that **the minimum window size for 100% utilization is the bandwidth-delay product.**

Sliding window performance

- If there are no losses
 - if the window size satisfies:
$$W \geq b / L$$
where b is the bandwidth delay product, L the packet size.
 - sliding window protocol can have a throughput of 100% of link rate (if overhead is not accounted for)
 - counted in bytes, this means that **the minimum window size for 100% utilization is the bandwidth-delay product.**

Elements of ARQ

- The elements of an ARQ protocol are:
 - Sliding window:
 - used by all protocols
 - Error detection
 - at receiver on error detection (code)
 - Loss detection
 - at sender on timeout versus at receiver on gap detection
 - Acknowledgements: short control packets
 - cumulative versus selective
 - positive (ACK) versus negative (NAK)
 - Retransmission Strategy
 - Selective Repeat
 - Go Back n
 - Others

20

All ARQ protocols we will use are based on the principle of **sliding window**.

• **Loss detection** can be performed by a timer at the source (see the Stop and Wait example). Other methods are: gap detection at the destination (see the Go Back n example and TCP's "fast retransmit" heuristic).

• **Acknowledgements** can be **cumulative**: acknowledging "n" means all data numbered up to n has been received. **Selective** acknowledgements mean that only a given packet, or explicit list of packets is acknowledged. A **positive** acknowledgement indicates that the data has been received. A **negative** acknowledgement indicates that the data has not been received; it is a request for retransmission, issued by the destination.

• The **retransmission strategy** can be go back n, selective repeat, or any combination. Go back n, selective repeat are explained in the following slides.

Fundamentally, three additional protocol capabilities are required in ARQ protocols to handle the presence of bit errors:

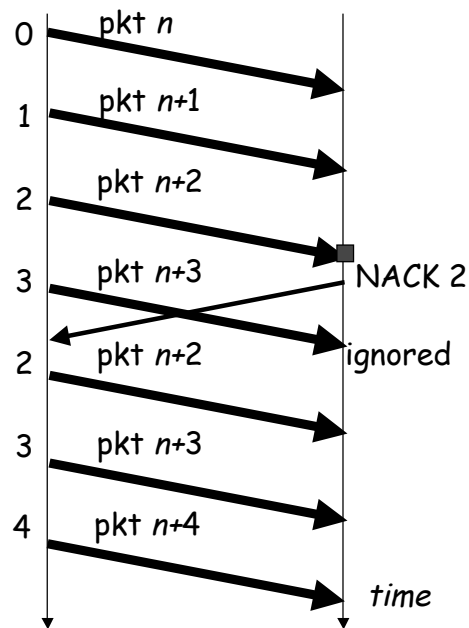
• **Error detection**. First, a mechanism is needed to allow the receiver to detect when bit errors have occurred. We saw that UDP uses the Internet checksum field for error detection, but we also need techniques to allow the receiver, once detected) to possibly correct packet bit errors. These techniques require that extra bits (beyond the bits of original data to be transferred) be sent from the sender to receiver; these bits will be gathered into the packet checksum field of the data packet.

• **Receiver feedback**. Since the sender and receiver are typically executing on different end systems, possibly separated by thousands of miles, the only way for the sender to learn whether or not a packet was received correctly by the receiver is for the receiver to provide explicit feedback to the sender. The positive (ACK) and negative (NAK) acknowledgment replies are examples of such feedback. In principle, these packets need only be one bit long; for example, a 0 value could indicate a NAK and a value of 1 could indicate an ACK.

• **Retransmission**. A packet that is received in error at the receiver will be retransmitted by the sender.

Go back N (GBN)

- Retransmission of all packets starting from the bad one



21

In a Go-Back-N (GBN) protocol, the sender is allowed to transmit multiple packets (when available) without waiting for an acknowledgment, but is constrained to have no more than some maximum allowable number, N , of unacknowledged packets in the pipeline. The figure shows the sender's view of the range of sequence numbers in a GBN protocol. If we define `base` to be the sequence number of the oldest unacknowledged packet and `nextseqnum` to be the smallest unused sequence number (that is, the sequence number of the next packet to be sent), then four intervals in the range of sequence numbers can be identified. Sequence numbers in the interval $[0, \text{base}-1]$ correspond to packets that have already been transmitted and acknowledged. The interval $[\text{base}, \text{nextseqnum}-1]$ corresponds to packets that have been sent but not yet acknowledged. Sequence numbers in the interval $[\text{nextseqnum}, \text{base}+N-1]$ can be used for packets that can be sent immediately, should data arrive from the upper layer. Finally, sequence numbers greater than or equal to $\text{base}+N$ cannot be used until an unacknowledged packet currently in the pipeline has been acknowledged. The range of permissible sequence numbers for transmitted but not-yet-acknowledged packets can be viewed as a "window" of size N . N is often referred to as the **window size** and the GBN protocol itself as a **sliding-window protocol**. In practice, a packet's sequence number is carried in a fixed-length field in the packet header. If k is the number of bits in the packet sequence number field, the range of sequence numbers is thus $[0, 2^k - 1]$. With a finite range of sequence numbers, all arithmetic involving sequence numbers must then be done using modulo 2^k arithmetic. In the GBN protocol, an acknowledgment for packet with sequence number n will be taken to be a **cumulative acknowledgment**, indicating that all packets with a sequence number up to and including n have been correctly received at the receiver. The protocol's name, "Go-Back-N," is derived from the sender's behavior in the

GBN: sender

- The GBN sender must respond to three types of events:
 - *packet to be sent.*
 - if the window is not full, send packet and variables are appropriately updated.
 - otherwise upper layer waits.
 - *Cumulative ACK.*
 - Ack for packet n (sequence number) = all packets up to and including n have been correctly received.
 - *A timeout.*
 - If a timeout occurs, resends *all* packets yet-to-be-acknowledged .
 - If ACK, timer is restarted for yet-to-be-acknowledged packets

22

The GBN sender must respond to three types of events:

• *packet to be sent.* The sender first checks to see if the window is full, that is, whether there are N outstanding, unacknowledged packets. If the window is not full, a packet is created and sent, and variables are appropriately updated. If the window is full, the sender simply returns the data back to the upper layer, an implicit indication that the window is full. The upper layer would presumably then have to try again later. In a real implementation, the sender would more likely have either buffered (but not immediately sent) this data, or would have a synchronization mechanism (for example, a semaphore or a flag) that would allow the upper layer to send packets only when the window is not full.

• *Cumulative ACK.* An acknowledgment for packet with sequence number n indicates that all packets with a sequence number up to and including n have been correctly received at the receiver.

• *A timeout event.* If a timeout occurs, the sender resends *all* packets that have been previously sent but that have not yet been acknowledged. If an ACK is received but there are still additional transmitted-but-yet-to-be-acknowledged packets, the timer is restarted. If there are no outstanding unacknowledged packets, the timer is stopped.

GBN: receiver

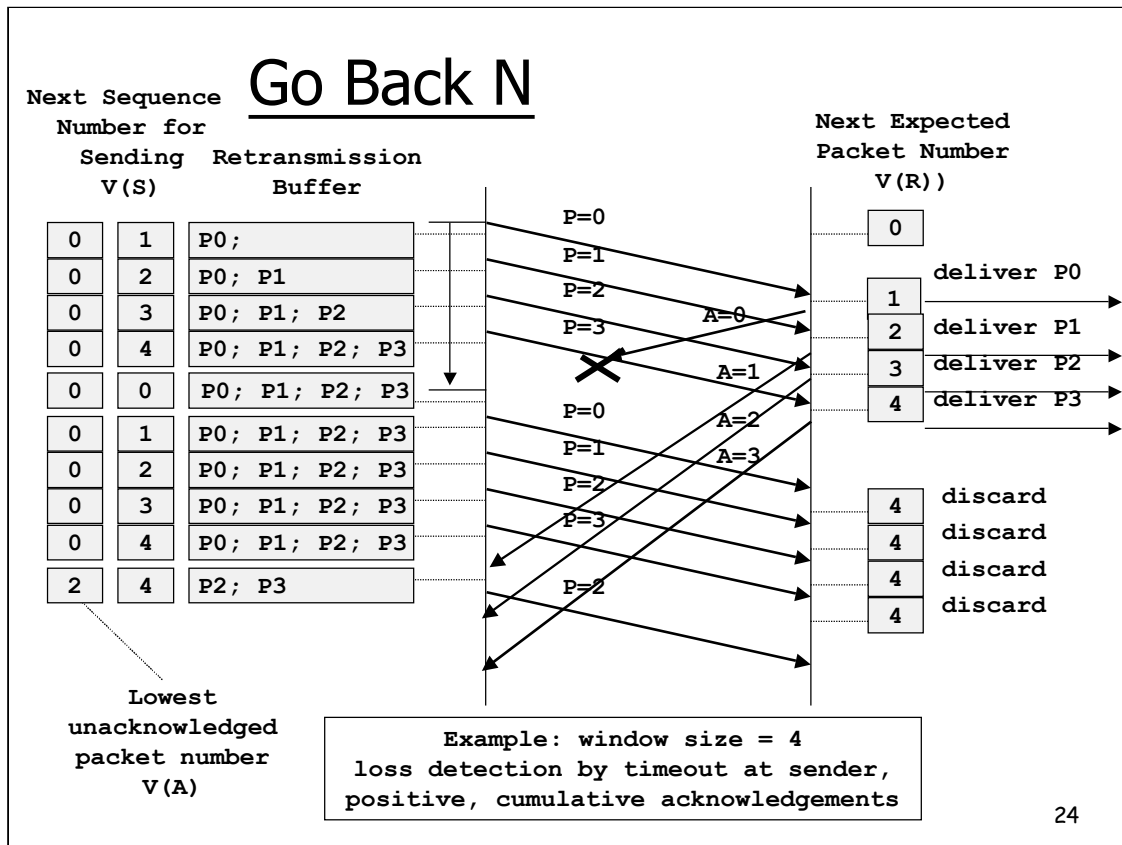
The receiver is simple:

- ACK-only: always send ACK for correctly-received pkt with highest *in-order* seq #
 - need only remember the expected seq #
 - cumulative ack
- out-of-order pkt:
 - discard (don't buffer) -> no receiver buffering! and no reordering
 - ACK pkt with highest in-order seq #

23

The receiver's actions in GBN are also simple. If a packet with sequence number n is received correctly and is in order (that is, the data last delivered to the upper layer came from a packet with sequence number $n - 1$), the receiver sends an ACK for packet n and delivers the data portion of the packet to the upper layer. In all other cases, the receiver discards the packet and resends an ACK for the most recently received in-order packet. Note that since packets are delivered one-at-a-time to the upper layer, if packet k has been received and delivered, then all packets with a sequence number lower than k have also been delivered. Thus, the use of cumulative acknowledgments is a natural choice for GBN.

The receiver discards out-of-order packets, for the simplicity of receiver buffering (not as TCP). Thus, while the sender must maintain the upper and lower bounds of its window and the position of `nextseqnum` within this window, the only piece of information the receiver need maintain is the sequence number of the next in-order packet.



Go Back N (GBN) is a family of sliding window protocols that is simpler to implement than SRP, and possibly requires less buffering at the receiver. The principle of GBN is:

- if packet numbered n is lost, then all packets starting from n are retransmitted;
- packets out of order need not be accepted by the receiver.

In the example, packets are numbered 0, 1, 2, ...

At the sender:

A copy of sent packets is kept in the send buffer until they are acknowledged

(R1) The sequence numbers of *unacknowledged* packets differ by less than W (window size)

The picture shows two variables: $V(S)$ ("Next Sequence Number for Sending") which is the number of the next packet that may be sent, and $V(A)$ ("Lowest Unacknowledged Number"), which is equal to the number of the last acknowledged packet + 1.

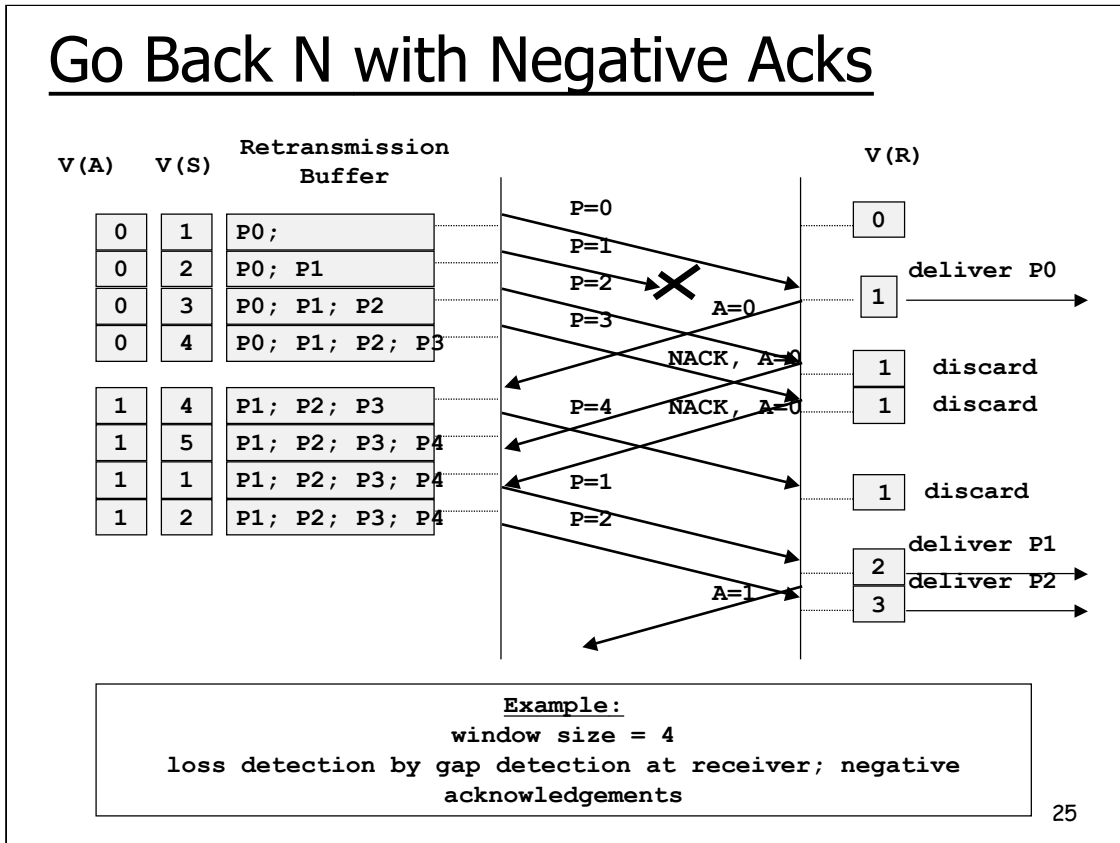
A packet may be sent only if: (1) its number is equal to $V(S)$ and (2) $V(S) \leq V(A) + W - 1$. The latter condition is the translation of rule **R1**.

$V(S)$ is incremented after every packet transmission. It is set (decreased) to $V(A)$ whenever a retransmission request is activated (here, by timeout at the sender).

$V(A)$ is increased whenever an acknowledgement is received. Acknowledgements are cumulative, namely, when acknowledgement number n is received, this means that all packets until n are acknowledged. Acknowledged packets are removed from the retransmission buffer.

At any point in time we have: $V(A) \leq V(S) \leq V(A) + W$

Go Back N with Negative Acks



Negative acknowledgements are an alternative to timeouts at sender for triggering retransmissions.

Negative acknowledgements are generated by the receiver, upon reception of an out-of-sequence packet.

NACK, A= n means: all packets until n have been received in sequence and delivered, and packets after n should be retransmitted

Negative acknowledgements are implemented mainly on sequence preserving channels(for example, with HDLC, LLC-2, etc). They are not defined with TCP.

Negative acknowledgements increase the retransmission efficiency since losses can be detected earlier.

Since NACKs may be lost, it is still necessary to implement timeouts, typically at the sender. However, the timeout value may be set with much less accuracy since it is used only in those cases where both the packet and the NACK are lost. In contrast, if there are no NACKs, it is necessary to have a timeout value exactly slightly larger than the maximum response time (round trip plus processing, plus ack withholding times).

GBN problems

- large window size and bandwidth-delay product = many packets can wait for ACK
 - a single packet error -> retransmit a large number of packets
 - this is not necessary!
- As the probability of channel errors increases, the pipeline can become filled with these unnecessary retransmissions.

26

With GBN, when the window size and bandwidth-delay product are both large, many packets can be in the pipeline. A single packet error can thus cause GBN to retransmit a large number of packets, many of which may be unnecessary. As the probability of channel errors increases, the pipeline can become filled with these unnecessary retransmissions.

Selective Repeat Protocol (SRP)

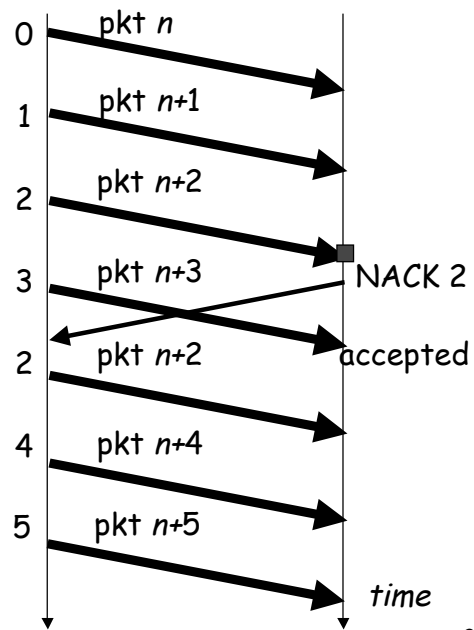
- receiver *individually* acknowledges all correctly received pkts
 - buffers pkts, as needed, for eventual in-order delivery to upper layer
- sender only resends pkts for which ACK not received
 - sender timer for each unACKed pkt
- sender window
 - N consecutive seq #'s
 - again limits seq #'s of sent, unACKed pkts

27

As the name suggests, **Selective-Repeat protocols (SRP)** avoid unnecessary retransmissions by having the sender retransmit only those packets that it suspects were received in error (that is, were lost or corrupted) at the receiver. This individual, as-needed, retransmission will require that the receiver *individually* acknowledge correctly received packets. A window size of N will again be used to limit the number of outstanding, unacknowledged packets in the pipeline. However, unlike GBN, the sender will have already received ACKs for some of the packets in the window. The SRP receiver will acknowledge a correctly received packet whether or not it is in-order. Out-of-order packets are buffered until any missing packets (that is, packets with lower sequence numbers) are received, at which point a batch of packets can be delivered in-order to the upper layer.

Selective Repeat

- Retransmission of the bad packet



28

Selective Repeat: sender

data to send from upper layer

- if next available seq # in window, send packet

timeout(n):

- resend packet n, restart timer

ACK(n):

- mark packet n as received
- if n smallest unACKed packet, advance window base to next unACKed seq #

29

1. *Data received from above.* When data is received from above, the SRP sender checks the next available sequence number for the packet. If the sequence number is within the sender's window, the data is packetized and sent; otherwise it is either buffered or returned to the upper layer for later transmission, as in GBN.
2. *Timeout.* Timers are again used to protect against lost packets. However, each packet must now have its own logical timer, since only a single packet will be transmitted on timeout. A single hardware timer can be used to mimic the operation of multiple logical timers.
3. *ACK received.* If an ACK is received, the SRP sender marks that packet as having been received, provided it is in the window. If the packet's sequence number is equal to `send-base`, the window base is moved forward to the unacknowledged packet with the smallest sequence number. If the window moves and there are untransmitted packets with sequence numbers that now fall within the window, these packets are transmitted.

Selective Repeat: receiver

packet n expected or higher:

- send ACK(n)
- out-of-order: buffer
- in-order: deliver up (also deliver buffered, in-order packets), advance window to next not-yet-received packet

packet n smaller less than N than expected:

- ACK(n)

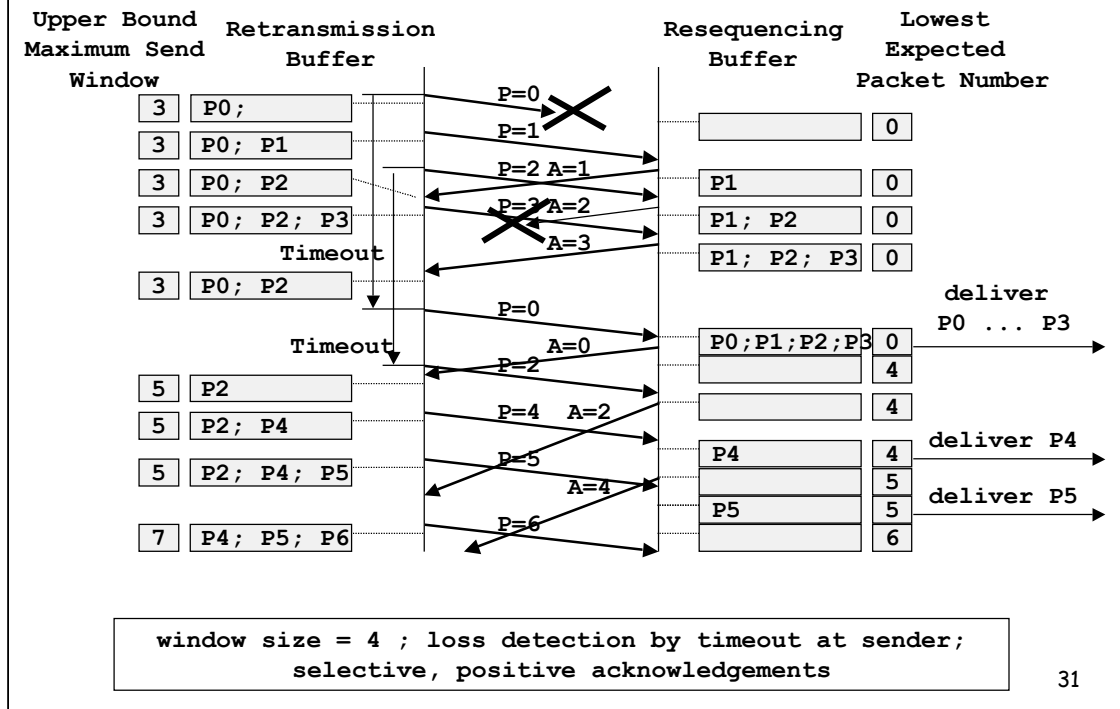
otherwise:

- ignore

30

1. *Packet with sequence number in $[rcv_base, rcv_base+N-1]$ is correctly received.* In this case, the received packet falls within the receiver's window and a selective ACK packet is returned to the sender. If the packet was not previously received, it is buffered. If this packet has a sequence number equal to the base of the receive window (rcv_base in Figure 3.22), then this packet, and any previously buffered and consecutively numbered (beginning with rcv_base) packets are delivered to the upper layer. The receive window is then moved forward by the number of packets delivered to the upper layer. As an example, consider Figure 3.25. When a packet with a sequence number of $rcv_base=2$ is received, it and packets rcv_base+1 and rcv_base+2 can be delivered to the upper layer.
2. *Packet with sequence number in $[rcv_base-N, rcv_base-1]$ is received.* In this case, an ACK must be generated, even though this is a packet that the receiver has previously acknowledged.
3. *Otherwise.* Ignore the packet.

Selective Repeat



31

Selective Repeat (SRP) is a family of protocols for which the only packets that are retransmitted are the packets assumed to be missing.

In the example, packets are numbered 0, 1, 2, ...

At the sender:

a copy of sent packets is kept in the send buffer until they are acknowledged

The sequence numbers of **unacknowledged** packets differ by less than W (window size)

A new packet may be sent only if the previous rule is satisfied. The picture shows a variable ("upper bound of maximum send window") which is equal to: the smallest unacknowledged packet number + $W - 1$. Only packets with numbers \leq this variable may be sent. The variable is updated when acknowledgements are received.

At the receiver:

received packets are stored until they can be delivered in order.

the variable "lowest expected packet number" is used to determine if received packets should be delivered or not. Packets are delivered when a packet with number equal to this variable is received. The variable is then increased to the number of the last packet delivered + 1.

Flow Control

- Purpose: prevent buffer overflow **at receiver**
 - receiver not ready (software not ready)
 - many senders to same receiver (overload focused on receiver)
 - receiver slower than sender
- Solutions: Backpressure, Sliding Window, Credit

- ***Flow Control*** is not the same as ***Congestion control*** (inside the network)

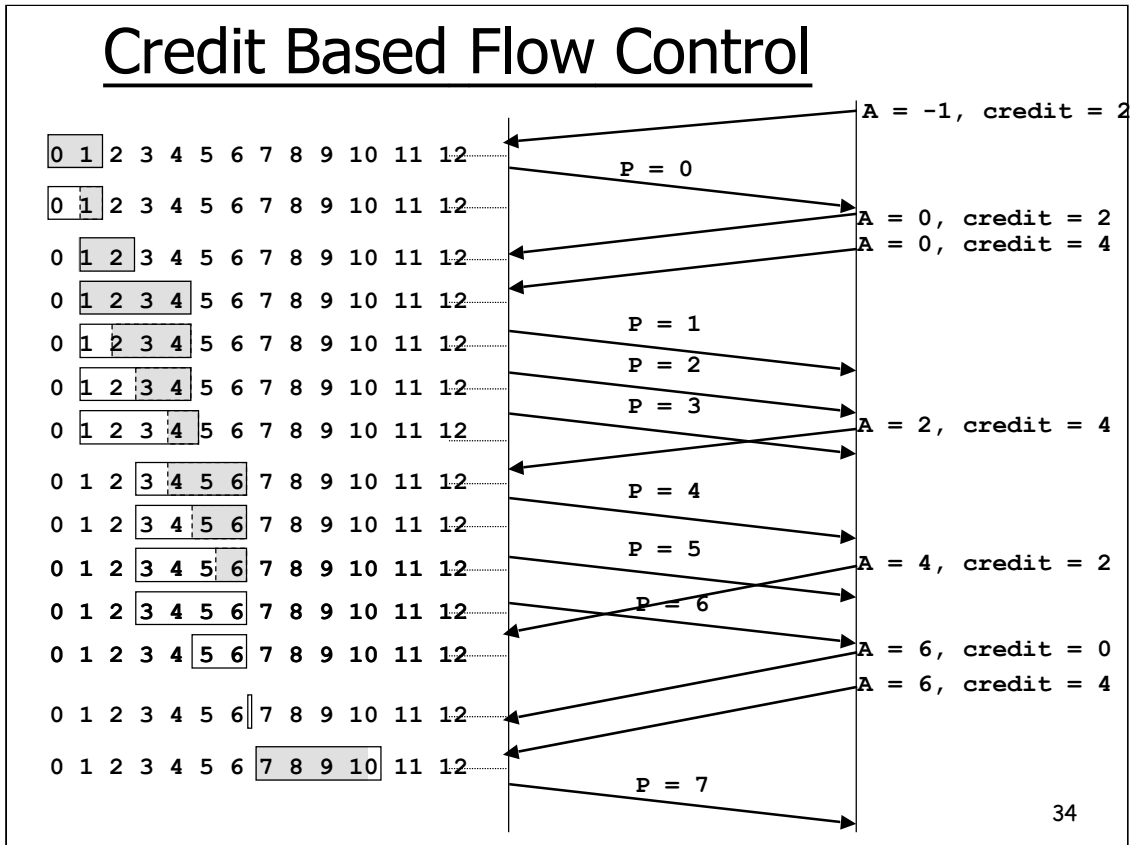
Sliding Window Flow Control

- Number of packets sent but unacknowledged $\leq W$
- Included in SRP and Go Back N protocols
 - assuming acknowledgements sent when receive buffer freed for packets received in order
- Receiver requires storage for at most W packets per sender

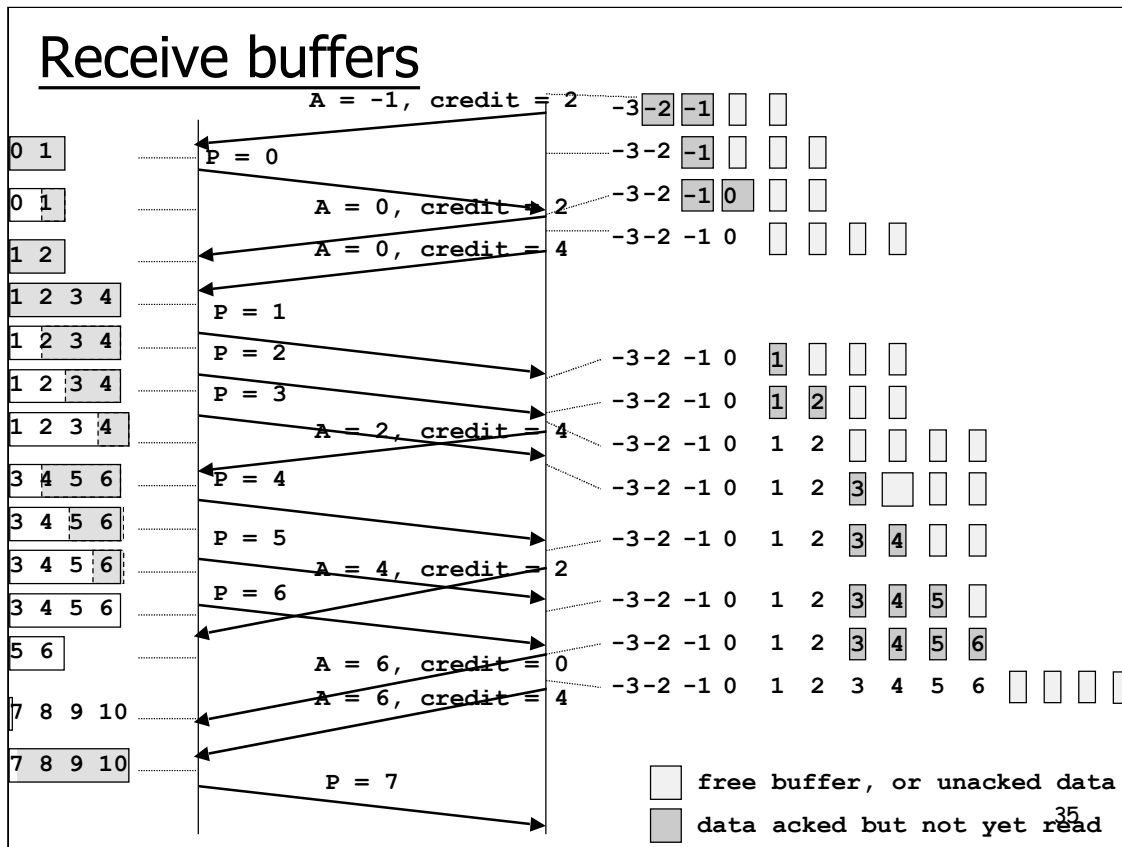
33

Sliding window protocols have an automatic flow control effect: the source can send new data if it has received enough acknowledgements. Thus, a destination can slow down a source by delaying, or withholding acknowledgements.

This can be used in simple devices; in complex systems (for example: a computer) this does not solve the problem of a destination where the data are not consumed by the application (because the application is slow). This is why such environments usually implement another form, called credit based flow control.



- With a credit scheme, the receiver informs the sender about how much data it is willing to receive (and have buffer for). Credits may be the basis for a stand-alone protocol (Gigaswitch protocol of DEC, similar in objective to ATM backpressure) or, as shown here, be a part of an ARQ protocol. Credits are used by TCP, under the name of “window advertisement”. Credit schemes allow a receiver to share buffer between several connections, and also to send acknowledgements before packets are consumed by the receiving upper layer (packets received in sequence may be ready to be delivered, but the application program may take some time to actually read them).
- The picture shows the maximum send window (called “offered window” in TCP) (red border) and the usable window (pink box). On the picture, like with TCP, credits (= window advertisements) are sent together with acknowledgements. The acknowledgements on the picture are cumulative.
- Credits are used to move the right edge of the maximum send window. (Remember that acknowledgements are used to move the left edge of the maximum send window).
- By acknowledging all packets up to number n and sending a credit of k , the receiver commits to have enough buffer to receive all packets from $n+1$ to $n+k$. In principle, the receiver (who sends acks and credits) should make sure that $n+k$ is non-decreasing, namely, that the right edge of the maximum send window does not move to the left (because packets may have been sent already by the time the sdr receives the credit).
- A receiver is blocked from sending if it receives $\text{credit} = 0$, or more generally, if the received credit is equal to the number of unacknowledged packets. By the rule above, the received credits should never be less than the number of unacknowledged packets.
- With TCP, a sender may always send one byte of data even if there is no credit (window probe, triggered by persistTimer) and test the receiver’s advertised window, in order to avoid deadlocks (lost credits).



The figure shows the relation between buffer occupancy and the credits sent to the source. This is an ideal representation. Typical TCP implementations differ because of misunderstandings by the implementers.

The picture shows how credits are triggered by the status of the receive buffer. The flows are the same as on the previous picture.

The receiver has a buffer space of 4 data packets (assumed here to be of constant size for simplicity). Data packets may be stored in the buffer either because they are received out of sequence (not shown here; some ARQ protocols such as LLC-2 simply reject packets received out of sequence), or because the receiving application, or upper layer, has not yet read them.

The receiver sends window updates (=credits) in every acknowledgement. The credit is equal to the available buffer space.

Loss conditions are not shown on the picture. If losses occur, there may be packets stored in the receive buffer that cannot be read by the application (received out of sequence). In all cases, the credit sent to the source is equal to the buffer size, minus the number of packets that have been received in sequence. This is because the sender is expected to move its window based only on the smallest ack number received. See also exercises.

Reliable Transport - summary

- Principles behind transport layer services - reliable data transfer:
 - Sliding window
 - Error and Loss detection: ARQ procedures
 - Retransmission Strategies
 - Stop & Go
 - Selective Repeat
 - Go Back n
 - Flow control