1 GSW… Error Control

Just about all communication systems attempt to ensure that the data gets to the other end of the link without errors. Since it's impossible to build an error-free physical layer (although some short links can get very close to error-free operation) this means there is a requirement for any packets[1] that do arrive with errors in them to be re-transmitted. How a receiver works out that there have been some errors in the packet is the subject of the "Error Detection" chapter; what the receiver and transmitter then do about it is the subject of this one.

Protocols that guarantee to get data to the destination correctly (or inform the layer above them that the attempt has failed) are known as *reliable protocols*. All reliable protocols need a bi-directional communication link, as the receiver has to be able to transmit short packets (called *acknowledgements*) back to the transmitter, informing the transmitter whether the information has arrived correctly, or not.

Since these acknowledgements (or *ACK*s) can be efficiently combined with those required for flow control, this function is often done at the same layers: usually the transport layer and/or the data link layer. If a packet fails to arrive, or arrives in error, the receiver can send back a negative acknowledgement (or *NAK*), which asks the transmitter to resend the information. This process is known as *ARQ* (Automatic Repeat reQuest).

While the term 'ARQ' is commonly used to refer to this form of error control, just automatically sending requests to repeat information isn't enough to make these schemes work. The receiver doesn't always send back a retransmission request for a packet that fails to arrive. After all, how could it? If the packet doesn't arrive, how would the receiver know the transmitter ever sent it in the first place?

That's not quite as stupid a question as it might seem. There is a way that a receiver can work out if a packet doesn't arrive, and that's by looking at the *sequence number* of the packet.

The idea goes like this: each new packet a reliable protocol layer sends contains a number in the header field, called the *sequence number*. This number increases by one for each new packet transmitted. Then if, for example, packets arrive at the receiver with sequence numbers 0,1,2,4,5 and 6, it's pretty obvious that a packet with sequence number 3 hasn't arrived. Only if the last packet the transmitter sends is lost does this scheme not work. In this case the receiver would never find out that a packet had gone missing, and the transmitter would need to detect the problem some other way: usually by using a *timeout timer*: a countdown that the transmitter sets going when it transmits a packet. If this timer counts down to zero before an acknowledgement is received, the transmitter assumes the packet has been lost, and sends another copy.

1.1 Issues with Reliable Protocols

Reliable protocols have to cope with three possible error situations:

---

[1] In TCP/IP networks, error control is most often implemented at the transport layer or above, so the use of the word packet, or perhaps message is appropriate. In some wireless networks and local-area networks, error-control is also done at the data-link layer, and in this case I should perhaps call them "frames". I'll try to keep this general, and stick to "packets" here.

- The packet gets lost on the way from the transmitter to the receiver, so that the receiver never finds out that the packet was sent at all. In this case, the receiver cannot send a negative acknowledgement (NAK).
- The packet arrives at the receiver with errors that the receiver can detect. In this case the receiver can send a negative acknowledgement (NAK), requesting the retransmission of the packet.
- The packet arrives at the receiver correctly, but the acknowledgement coming back from the receiver is corrupted, and either never arrives at the transmitter, or arrives with errors in it.

Problems due to the first situation can be solved by having a timeout timer at the transmitter, which automatically retransmits the packet in the cases where the receiver doesn't detect that the packet was sent, or doesn't tell the transmitter.

In the second case, the receiver can send back negative acknowledgements (NAKs) asking for a packet to be retransmitted. This is usually faster than waiting for the retransmission timer at the transmitter to time out. (Although in practice many protocols don't bother to send back NAKs for packets that arrive damaged, they just ignore them and let the timeout timer at the transmitter time out anyway. This is the case with, for example, Ethernet, which throws away any packet that arrives with an error, and never tells the reliable protocol layers above that anything has happened. If packet errors are rare, the slight loss in utilisation due to the slower retransmissions is not a big problem.)

The third situation introduces a new problem: the receiver will receive two copies of the same packet. To deal with this situation, some scheme must be provided whereby the receiver can tell the difference between a new packet and a retransmission of a packet it has already received, and this can also be done using the sequence numbers in the packet header. The complication is that there aren't an infinite number of sequence numbers: that would require an infinite length protocol header in the packet, and that's not efficient in terms of using most of the network capacity to transmit useful information. Ideally we're trying to keep the packet headers as short as possible.

So sequence numbers start from zero, and go up to some maximum sequence number, after which they loop back to zero, and start counting up again. For example, a seven-bit field for storing the sequence number would allow 128 different sequence numbers, from zero to 127. The 128th packet would then have a sequence number of zero, the 129th a sequence number of one, and so on. This means that in a long conversation, several packets will be transmitted with the same sequence number (in this example, one in every 128 packets will have the same sequence number). Reliable protocols must be carefully designed to make sure that this doesn't cause confusion at the receiver.

There's also an issue with reliable protocols that operate over the transport layer (i.e. above the network layer): packets can be re-ordered as they go across the network. Just receiving packets with sequence numbers 0,1,2,4,5 and 6 is no guarantee that packet number 3 has got lost, it might just have been delayed, and it'll turn up if the receiver waits long enough. This is another of those trade-off situations: how long should you wait before assuming the packet has been lost? Too long, and communications are held up for a long time before the packet is retransmitted; too short, and two copies of the packet will arrive at the receiver, which is a waste of the network capacity.

© 2007 Dave Pearce

There are several techniques used to implement error control in reliable protocols, the three most common are known as *stop-and-wait ARQ*, *go-back-N ARQ* and *selective-repeat ARQ*.[2] Just as with flow control, what we ideally want is a scheme that provides the maximum *utilisation*[3] with the minimum possible *overhead*[4].

1.2 Stop and Wait ARQ

Stop and wait is the simplest possible ARQ protocol: the transmitter expects a positive acknowledgement back from the receiver for every packet transmitted. If, after transmitting a packet, no acknowledgement arrives after a time *T* (the timeout period), the packet is assumed to be lost, and is resent. Provided this time period *T* is longer than the round-trip time for the link, this scheme is very simple, and very rugged. It's just not very efficient.

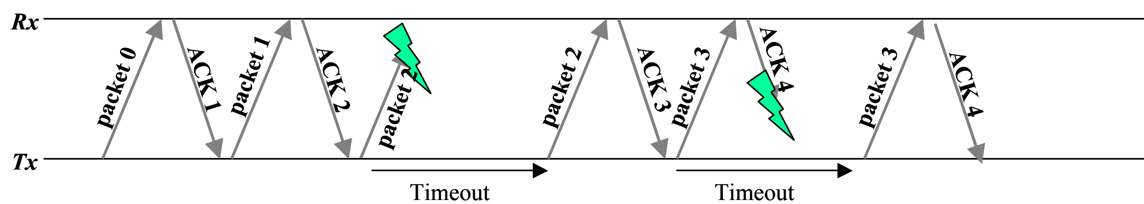

Figure 1-1 - Stop and Wait Error Control

The operation of the scheme is illustrated in the figure above. Note also that the receiver sends an ACK requesting the next packet, rather than saying it has received the last one. So, for example, receiving packet 0 causes an ACK 1, requesting a packet with a sequence number of 1. This is conventional; it is how most ARQ schemes work.

Note that it is possible for the receiver to receive two copies of the same packet (in this case packet 3).

The sequence number can be used to prevent the duplicate copy of packet 3 being passed on further up the protocol stack: the reliable protocol layer at the receiver just throws away any packet that arrives with the same sequence number as the last packet. In the case of stop-and-wait ARQ a one-bit sequence number is sufficient to avoid this problem, and this is a very low overhead.

1.2.1 *Utilisation of Stop-and-Wait ARQ*

The total time taken to transmit one packet on average is a function of the packet error rate on the link, the length of the packets and the acknowledgements, the timeout interval, and the

---

[2] Some common reliable protocols (for example TCP) use a combination of these schemes, but for clarity, I'll describe each one separately first.

[3] Utilisation being defined as the ratio of the throughput to the network capacity: in other words the proportion of time that the receiver is receiving useful information. Note that the receiver might receive the same packet more than once, but it's only the first time that the information is useful.

[4] The overhead is the additional amount of data that has to be sent to make the protocol reliable. This includes the sequence numbers and other fields in the packet headers, as well as the acknowledgements that come back from the receivers.

propagation time from the transmitter to the receiver. Just like stop-and-wait flow control, for short links and/or long packets with a low error rate this scheme works fine; but for longer links, or where the bit error rate is higher, the utilisation is poor.

I'll make the same assumptions as for the derivation of the utilisation of the stop-and-wait flow control scheme:

- The time taken for processing a packet at the receiver and the acknowledgement at the transmitter is negligible.
- The time required to transmit an acknowledgement is negligible compared to the time taken to transmit a packet (in other words, acknowledgements are short compared to packets).

And further, I'll assume that:

- The round-trip time is constant[5]
- The timeout period is the minimum possible (i.e. just the round-trip time)[6]
- The probability of error in an acknowledgement is small compared to the probability of error in a packet, and can be neglected.

For stop-and-wait flow control, we calculated a utilisation U of:

$$U = \frac{t_{packet}}{t_{packet} + 2t_{prop}} = \frac{1}{1+2a} \tag{0.1}$$

where a is the ratio of the propagation time across the network t_{prop} to the time required to transmit a packet, t_{packet}.

With these assumptions, the transmitter is transmitting packets every bit as often as it was in the ideal case: it's just that in some cases a negative acknowledgement comes back, or no acknowledgement comes back at all, and the packet transmitted is a retransmission, rather than a new packet. What we need to know is the proportion of these packets that are re-transmissions.

For a reliable protocol, the probability that any packet is a re-transmission (i.e. contains no new data) is just the probability that the previous packet was corrupted (or the previous acknowledgement was corrupted: here this is neglected).

Let the probability of a packet error be p. Then, the number of packets sent in one second is:

$$\text{Packets per second} = \frac{1}{t_{packet} + 2t_{prop}} \tag{0.2}$$

---

[5] This isn't usually true, the timeout period is usually set to a larger value than this, since the reliable protocols rarely know in advance exactly how long it will take a packet to get to the other end, and for the acknowledgement to get back. For example, an Ethernet frame might have to wait for some time before the network becomes quiet and the transmission can start, and this time is a random quantity; an Internet packet might have to wait for a random time in a queue in a router before the required output port becomes free.

[6] Usually impossible, since the round-trip time isn't known in advance, see the previous footnote.

hence the number of packets which have errors in them sent per second is:

$$\text{Errored packets per second} = \frac{p}{t_{packet} + 2t_{prop}} \tag{0.3}$$

therefore the number of packets transmitted per second that don't have errors in them:

$$\text{New packets per second} = \frac{1-p}{t_{packet} + 2t_{prop}} \tag{0.4}$$

and in the case where there is no flow control or errors, the network capacity is:

$$\text{Max packets per second} = \frac{1}{t_{packet}} \tag{0.5}$$

and since the number of packets transmitted per second with no errors must be equal to the number of packets received correctly[7], the utilisation U of the link is:

$$U = \frac{1-p}{t_{packet} + 2t_{prop}} \div \frac{1}{t_{packet}} = \frac{1-p}{1+2a} \tag{0.6}$$

There's an easier way to get to this result: since a proportion p of the packets are received in error, a proportion $(1 - p)$ must be received correctly. Since the total number of packets received correctly is by definition the throughput of the link, the utilisation is just $(1 - p)$ times the utilisation of a stop-and-wait flow control scheme. That's it.

1.3 Go-Back-N ARQ

At the expense of some additional complexity, we can do much better. Go-Back-N ARQ uses the sliding window technique (see the chapter on Flow Control) to allow the transmitter to send multiple packets before getting a response back from the receiver. Extending this sliding window flow control scheme to provide error control as well is rather more complicated than it might appear at first sight.

Firstly, some terminology: the number of packets that the transmitter can transmit before receiving an acknowledgement for any of them is known as the *window size* (just like in flow control). The packets that have been transmitted but have not yet been acknowledged are referred to as being *in flight*. (So, the maximum number of packets in flight is equal to the window size.) The maximum sequence number used to identify packets and keep them in the correct sequence is known as the *maximum sequence number*. Ideally, the maximum sequence number would be infinite: in practice, it is kept to a finite number to reduce the overheads in the packet headers.

The operation of go-back-N flow control can get quite involved: part of a representative communication is shown in the figure below.

---

[7] This is true here since we are neglecting the possibility of an acknowledgement being lost or corrupted, so no valid packet will be received twice.

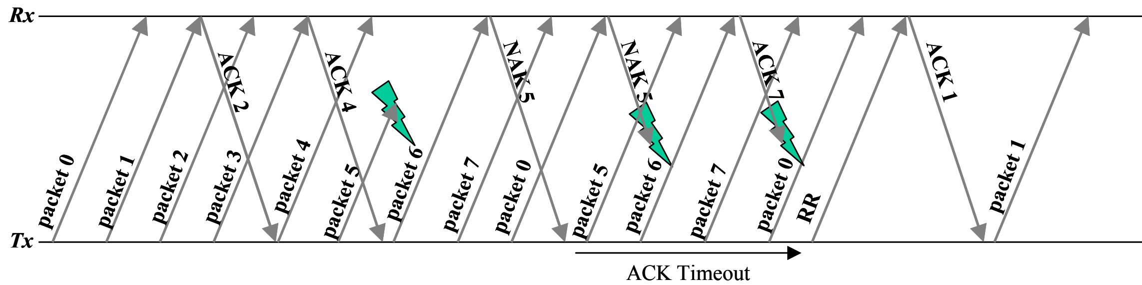

Figure 1-2 - Go-back-N Error Control

Note first that an acknowledgement is not sent back for every packet that arrives, in this case one acknowledgement is being sent for every two packets. This is a technique known as *delayed acknowledgements*. The advantage of this technique is that the number of acknowledgements required is reduced, by up to a factor of two. ("Up to", since in practice, you can't always send an acknowledgement every two packets: imagine what happens when there are only an odd number of packets to send, when would the last packet get acknowledged?)

Real receivers will often implement a strategy to prevent packets waiting without being acknowledged for too long. For example:

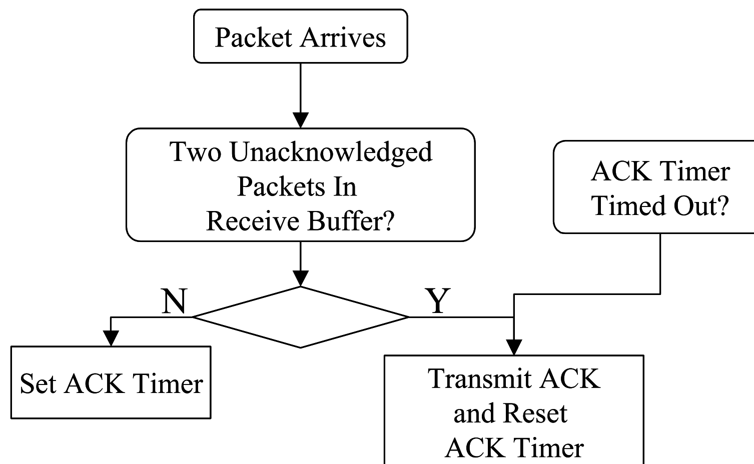

Figure 1-3 Example Delayed ACK Algorithm

Again, the length of time that the receiver should wait before sending an acknowledgement for a single received packet is a trade-off: too short, and too many acknowledgements will be sent using up a lot of network capacity; too long, and communications link can take a long time to recover from a lost packet (the timeout timer at the transmitter must be set to accommodate a maximum length wait by the receiver before it sends the acknowledgement).

Also note what happens after the ACK timeout. The transmitter sends an "RR" packet, which is a way of telling the receiver "I'm completely lost: which packet are you expecting?", and the receiver can then reply immediately (without waiting for any more packets to arrive), and the flow continues. It's often useful to have something in the protocol that allows the transmitter to call for help like this. (Note that it's the receiver that is controlling this flow. The transmitter asks the receiver what is happening, not the other way round.)

The figure also illustrates one of the key advantages of go-back-N error control: packets always arrive at the receiver in order. The receiver doesn't need any buffer memory at all, if a packet arrives out of order, it can be thrown away, in the sure knowledge that another copy of

the packet will come along later. This can greatly simplify the design (and reduce the cost) or receivers using this scheme. The disadvantage, of course, is that go-back-N requires several packets to be re-transmitted when only one is lost.

1.3.1 Window Sizes and Sequence Numbers for Go-Back-N

For stop-and-wait error control, a one-bit sequence number was all that was required. When using go-back-N error control, a larger maximum sequence number is required. The question is: how large? Or to put it another way, what's the maximum window size that can be used with a certain value of maximum sequence number?

Consider the following, where the maximum sequence number is seven:

- With a window size of nine, the transmitter transmits nine packets with sequence numbers 0,1,2,3,4,5,6,7 and 0. The last eight messages are lost in transit and never arrive. The receiver sends an ACK1, since the next packet it is expecting has this sequence number, and the transmitter carries on thinking everything is fine[8], and all nine packets have arrived.

Oh dear. Obviously a window size of nine is too much. What about eight?

- With a window size of eight, the transmitter transmits packets with sequence numbers 0,1,2,3,4,5,6 and 7. The receiver receives all of them, and returns some ACKs, all of which get lost in transit. The transmitter times out, and re-transmits all eight packets, which the receiver accepts as new packets, and returns another set of ACKs, which this time get through. The transmitter carries on happily, but the receiver has now accepted two copies of each of these packets.

Hmm. What about seven? Well, yes, in this case seven works.

These problems can be solved in general by ensuring that the window size is less than or equal to the maximum message number (provided you're counting from zero): that way only one packet with the same sequence number can be *in flight* (unacknowledged) at any time, and all the transmitted windows look different. With a maximum sequence number of seven (and therefore eight different sequence numbers: 0,1,2,3,4,5,6 and 7), we'll be fine as long as the maximum window size is seven or less.

Obviously go-back-N ARQ can be combined with sliding window flow control rather well, and both problems (error control and flow control) can be solved with a reasonably simple protocol that is simple to implement since it does not require the receiver to store packets. It's very popular.

1.3.2 Utilisation of Go-Back- N ARQ

An exact expression for the utilisation of go-back-N is difficult to derive, but we can calculate an approximate expression if we assume that the probability of losing two packets within the same window is negligible. This is a good assumption as long as the packet error rate is small.

[8] Well, probably not that everything is fine: it would have been expecting more ACKs to arrive that just one. However, it couldn't be certain whether it was the packets that had gone missing, or the ACKs. So it wouldn't know whether to re-transmit any packets or not.

We'll also make our usual assumptions[9] that the acknowledgements are short and are always received correctly, that the processing time at the transmitter and receiver is negligible, and the time-out period is equal to the round-trip time, which is constant.

Just as in sliding-window flow control, there are two cases to consider: when the window is big enough to keep the transmitter active all the time, and when it isn't, so that the transmitter has idle periods waiting for acknowledgements to arrive. Consider the former case first, since it’s easier:

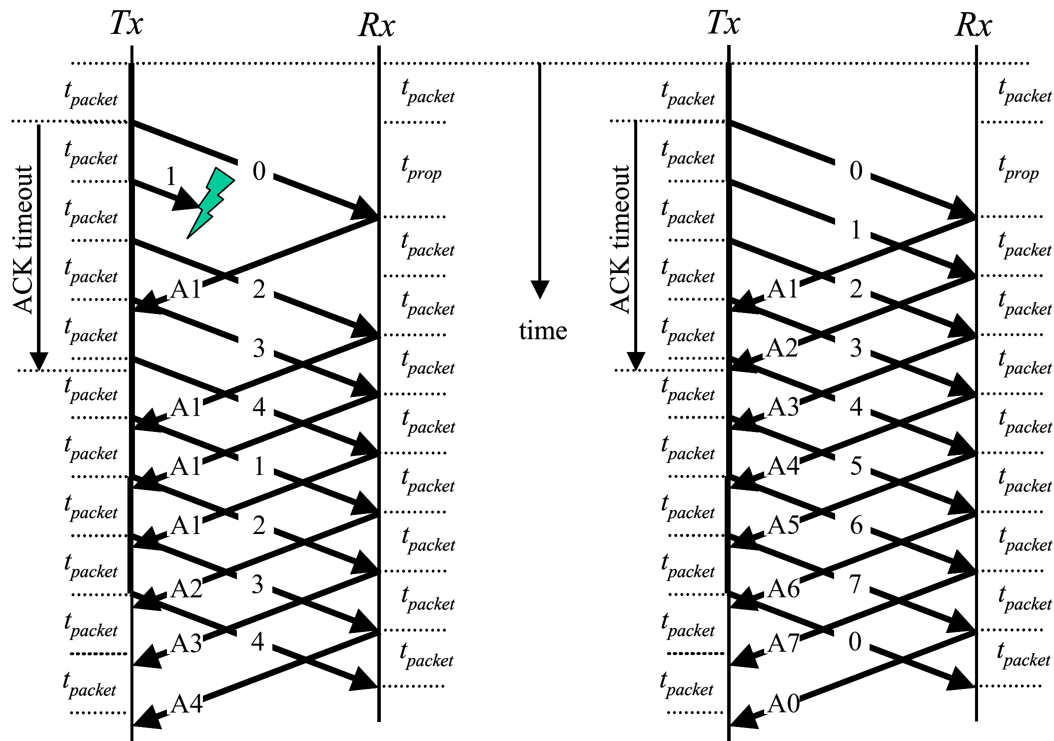

Figure 1-4 Go-Back-N with N > 1 + 2*a*

The figure above illustrates the case where a packet is lost (in this case packet with sequence number of one, on the left) and the corresponding case when no packet is lost (on the right). Here the window size *N* is large enough to keep the transmitter transmitting continuously.

Notice that the number of packets that require to be re-transmitted is the number of packets that the transmitter has sent in a time $t_{packet} + 2\ t_{prop}$, this being the time it would have taken the acknowledgement to get back to the transmitter if packet 1 had arrived, and therefore earliest time after sending the packet that the transmitted could know that anything had gone wrong.

Since the transmitter is transmitting continuously, the number of packets sent per second:

$$\text{Packets per second} = \frac{1}{t_{packet}} \qquad (0.7)$$

[9] See the chapter on “Flow Control” for more details about these assumptions.

and the number of packets which fail to arrive correctly per second is:

$$\text{Errored packets per second} = \frac{p}{t_{packet}} \tag{0.8}$$

where p is the probability of a packet error or loss. Now for each packet received with an error (or not received at all), the transmitter must ‘go back’ one round trip time, and start the transmission of all these packets again. The time ‘lost’ by the transmitter per lost packet is, as we’ve just seen, $t_{packet} + 2\ t_{prop}$. The amount of time spent doing these re-transmissions per second is then:

$$\text{Time spent retransmitting per second} = \frac{p}{t_{packet}}\left(t_{packet} + 2t_{prop}\right) \tag{0.9}$$

All the rest of the time, the transmitter is transmitting useful packets. Therefore, the utilisation of the link is:

$$\begin{aligned} U &= \frac{\text{Time receiving useful packets}}{\text{Total time}} \\ &= 1 - \frac{\text{Time spent re-transmitting packets}}{\text{Total time}} \\ &= 1 - \frac{p}{t_{packet}}\left(t_{packet} + 2t_{prop}\right) = 1 - p\left(1 + 2a\right) \end{aligned} \tag{0.10}$$

Now the other case: where the window is not big enough to keep the link full, and the transmitter has some idle time. In this case, an error in a packet will result in an entire window's worth of data being re-transmitted:

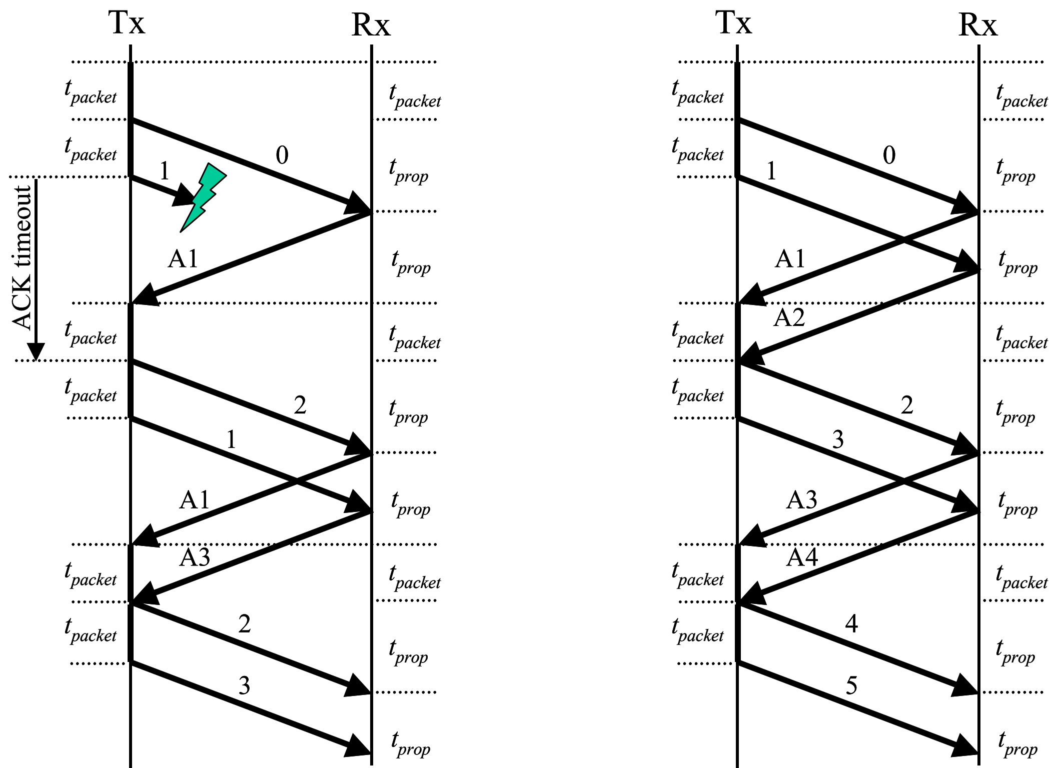

Figure 1-5 Go-Back-N with N < 1 + 2*a*

Note that at the end of the communication shown in the figure above, the case with an error (on the left) is exactly one window behind the case without the error (on the right). In this case, the window size is two.

More generally, if the window is *N* packets long, then:

$$\text{Packets per second} = \frac{N}{t_{packet} + 2t_{prop}} \tag{0.11}$$

and so the number of packets sent per second that don't arrive correctly is:

$$\text{Errored packets per second} = \frac{N\,p}{t_{packet} + 2t_{prop}} \tag{0.12}$$

For each of these packets, *N* packets must be resent, so the number of re-transmissions per second is:

$$\text{Retransmissions per second} = \frac{N^2\,p}{t_{packet} + 2t_{prop}} \tag{0.13}$$

and therefore the number of packets transmitted per second that are not re-transmissions:

$$\text{New packets per second} = \frac{N - N^2 p}{t_{packet} + 2t_{prop}} \tag{0.14}$$

and as before, in the case where there is no flow control or errors, the network capacity is:

$$\text{Max packets per second} = \frac{1}{t_{packet}} \tag{0.15}$$

therefore, the utilisation U of the link is:

$$U = \frac{N - N^2 p}{t_{packet} + 2t_{prop}} \div \frac{1}{t_{packet}} = \frac{N(1 - N\,p)}{1 + 2a} \tag{0.16}$$

You might like to confirm that in the case where the window size is only just big enough, that these two expressions give the same answer. ($N = 1 + 2a$ in this case.)

1.4 Selective Repeat ARQ

Selective Repeat ARQ is another method of combining sliding-window flow control with error control. It's more efficient, in that it only re-transmits the lost packets, not all the other ones that have been sent since. This requires a receiver that can tell the transmitter not only that some packets have been lost while others have arrived, but also which ones have been lost. (This information makes the acknowledgements bigger, but since the additional information in the acknowledgements are only required when a packet is lost, this doesn't have a large effect on the system efficiency.)

Compare the figure below with the figure for the corresponding case of go-back-N error control (figure 1-2). They're the same until the NAK arrives for frame five. At that point, a selective-repeat error-control scheme would re-transmit the lost frame five only, and then continue to transmit new frames. The result is more efficient transmission, and far fewer duplicate frames arriving at the receiver.

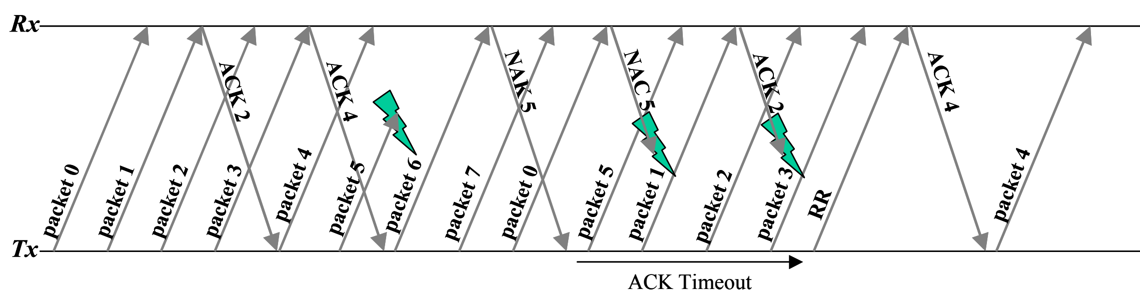

Figure 1-6 - Selective Repeat Error Control

There's a cost for this additional efficiency of course, and sometimes that cost is not worth paying. Firstly, this scheme requires a much more complex receiver, capable of storing packets in a local buffer when they arrive out-of-sequence, of inserting received packets into the correct place in this buffer, and of keeping a note of when all the packets in the buffer have arrived correctly, before passing them up to the higher layer in order, and of telling the transmitter exactly which packets have arrived and which have not.

There's another serious problem with selective-repeat error control as well: sooner or later, the simple scheme described above will fail. The problem occurs when the same packet gets lost several times in a row. For example, suppose the window size is three (so the transmitter can have three packets in flight at any one time), and the packet with sequence number two continually gets lost, every time it's sent.

Suppose the window size is three, and the propagation time is very long, so the transmitter has to wait after sending each burst of three packets before it can send the next one. The maximum sequence number is 7. Initially, the transmitter will send packets 0, 1 and 2. Packet 2 is lost, so it will next send packets 2, 3 and 4. Packet 2 gets lost again, so the transmitter will next send packets 2, 5 and 6. Packet 2 gets lost again, and the transmitter sends packets 2, 7 and 0. Once more packet 2 is lost, so the transmitter sends packets 2, 1 and 2. Oh dear. That's two packets with the same sequence number in the same window: that's not going to work, the receiver can't tell them apart.

The other problem with this is that the reliable protocol layer at the receiver is now storing packets 3,4,5,6,7,0 and 1 in its buffer. It can't send any of these up to the higher layers, since being a reliable protocol, it guarantees to send packets up in the right order. Sooner or later, even with a very large maximum sequence number, the receiver buffer is going to fill up.

Although in theory more efficient than go-back-N, due to these additional complexities, selective repeat is often not used. If the communications link is mostly error-free, then there aren't very many packet errors, and the gain in utilisation resulting from the additional complexity at the receiver of selective repeat is often not significant.

1.4.1 Window Sizes and Sequence Numbers for Selective Repeat

There's yet another reason why selective repeat isn't used more often. With go-back-N, everything worked fine provided the window size was no more than the maximum sequence number. With selective repeat, that doesn't work: you have to use a smaller window size.

If you want the transmitter to be able to transmit a full window at each round-trip time, then in theory at least, the maximum sequence number has to be infinite. Consider the case described above, with a window size of three, and the packet with sequence number two always getting lost. Eventually, no matter how big the maximum sequence number is, the transmitter will have to send two packets both with sequence number two[10].

Even if packet errors are so rare that we can safely assume that every lost packet gets through on the second attempt, there's still a disadvantage in using selective repeat. Consider the following, with a window size of six, and a maximum sequence number of seven (note that with go-back-N that would be fine):

1. The transmitter sends packets numbered 0,1,2,3,4 and 5.
2. The receiver receives all six packets, and acknowledges with ACK1, ACK2, …, ACK6.
3. There is a noise burst, and all the acknowledgements are lost.
4. The transmitter times out waiting for an acknowledgement, and re-transmit the packets.
5. There is another noise burst, and packets 4 and 5 are lost.

Compare with:

1. The transmitter sends packets numbered 0,1,2,3,4 and 5.
2. The receiver receives all six packets, and acknowledges with ACK1, ACK2, …, ACK6.
3. The ACKs arrive, and the transmitter sends packets with sequence numbers 6,7,0,1,2 and 3.
4. There is a noise burst, and packets 6 and 7 are lost in transit.

[10] In practice this problem is avoided using flow control techniques: if the receiver's buffer begins to fill up (as would be the case here), it signals back to the transmitter to stop any more packets arriving.

In both cases, after the error, the receiver receives packets with sequence numbers 0,1,2 and 3. However, in one case these are a re-transmission of the original packets, and in the other case they are part of a window of new packets. The receiver can't tell the difference.

The problem is that the transmitter has no idea whether the last packet to be received by the receiver is being placed at the end or at the beginning of the receive buffer. To make sure that there is no ambiguity, there must be a window's worth of packets before and after each packet, none of which shares the same sequence number. This means that the maximum window size must be limited to at most one half of the maximum sequence number.

On long links with few errors, where the size of the message number is already determined and cannot be changed (for example if there is a field of eight bits in the packet header for storing the sequence number), this limitation on the window size can result in selective-repeat ARQ having a lower utilisation than go-back-N ARQ.

1.4.2 Utilisation of Selective Repeat ARQ

It's comparatively easy to derive an approximate expression for this one. For the case where the window is not large enough to keep the transmitter transmitting continuously, the problem is identical to the case of go-back-N, except there is only one packet re-transmitted for each lost packet, not *N*. So the utilisation is just:

$$U = \frac{N - N\,p}{t_{packet} + 2t_{prop}} \div \frac{1}{t_{packet}} = \frac{N(1-p)}{1+2a} \qquad (0.17)$$

and similarly, for the case where the window is large enough, the transmitter can transmit packets continuously, it's just that a fraction p of them are re-transmissions, hence a fraction $(1 - p)$ of them are not re-transmissions, so the transmitter is transmitting useful data a fraction $(1 - p)$ of the time, and the utilisation is:

$$U = 1 - p \qquad (0.18)$$

but bear in mind that we've assumed that every packet gets through at worst at the second attempt, and the window size must now be at most half of the maximum sequence number.

1.5 Real Error Control Schemes

In real life, things are often a bit more complicated than this, since the most common reliable protocol on the Internet (TCP) has evolved error control schemes which interact with the flow control schemes, and attempt to prevent errors as well as requesting re-transmissions when they do happen.

Setting this aside for the moment (for more details see the chapter on "TCP Congestion Control"), TCP can be thought of as operating a hybrid go-back-N / selective-repeat scheme: the acknowledgements returned to the transmitter usually detail how many and which packets have been lost, allowing the transmitter to only resend the lost packets. However, there is no requirement on a receiver to store out-of-order packets. Most do, but a transmitter cannot assume that just because a receiver claims to have received an out of order packet, it will be storing it for future use. So the receiver is free to drop out-of-order packets if its buffer starts getting full.

TCP doesn't send negative acknowledgements (any packet that arrives with an error has probably already been thrown away by an unreliable protocol at a lower layer). The

transmitter has to work out from the acknowledgements that arrive which packets have got lost. In TCP it's very much the transmitter that is in control; it's the transmitter that works out which packets to retransmit and when.

Another common error-control scheme in real-life is LLC Type-2. This uses a go-back-N scheme with a window size of 127, and a seven-bit sequence number in the packet headers. Operation is straightforward, the receiver can send both NAKs and ACKs when packets arrive incorrectly and correctly respectively.

I could mention hybrid-ARQ schemes at this point as well, just for interest. These are more advanced schemes used at the data-link layer of some wireless protocols. The idea is that with advanced error-detection schemes, the receiver can sometimes tell which part of the packet has the errors in it. A re-transmission of the same packet might have errors in a different part of the frame. Despite the fact that both versions of the packets arrive with errors, it's sometimes possible to combine the information from both packets to produce an error-free packet. (The operation is rather more complicated than that description suggests.)

1.6 Key Points

- All reliable protocols require a receiver than can tell the transmitter which packets have arrived correctly by sending acknowledgements (and sometimes which ones have not arrived correctly as well, by sending negative acknowledgements).

- The three most common error control schemes, in order of complexity, are stop-and-wait, go-back-N and selective-repeat ARQ. For long networks / short packets, stop-and-wait is not very efficient. Whether go-back-N or selective-repeat is more efficient depends on the network length, packet size, window size and packet error rates.

- To identify the packets that have been lost, error control schemes use sequence numbers in the packet headers. The maximum sequence number (counting from zero) must be at least one for stop-and-wait, at least N for go-back-N and at least $2N$ for selective-repeat, where N is the window size.

1.7 Questions

1) True or false:

a) If acknowledgments never have errors (so only packets have errors), then stop-and-wait ARQ on a simple point-to-point link would not need sequence numbers.
b) If the maximum sequence number is 11, and the window size is 7, then go-back-N error control would always work.
c) If the maximum sequence number is 11, and the window size is 7, then selective-repeat error control would always work.
d) Go-back-N error control always has a greater utilisation than stop-and-wait error control.
e) Selective repeat always has a greater utilisation than go-back-N for the same window size.
f) Selective repeat always has a greater utilisation than go-back-N for the same maximum sequence number.

2) A radio link over 30 km uses stop-and-wait ARQ, with packets that take 1 ms to transmit, and a probability of packet error of 1%. The very short acknowledgement packets come back

over the phone network (at an average speed of 2/3 that of light), and can be assumed to be error free. What is the utilisation of this link?

3) The bit error rate in a link is 0.01%. If the packets transmitted are 128 bytes long, and the acknowledgements received are 16 bytes long, what is the probability of error for a packet, and for an acknowledgement?

What is the probability that no re-transmission is required for a packet in the case of a stop-and-wait ARQ scheme?

**4) Take those two additional assumptions we made in the derivation of the utilisation of stop-and-wait ARQ and relax them. Derive an expression for the utilisation of stop-and-wait assuming a timeout period of t_{out}, a probability of error in a packet of p_p, and a probability of error in an acknowledgement of p_a.

*5) I set up a go-back-N ARQ scheme on a link with a window size of four, and a maximum message number of three. Describe, using flow diagrams, an example of a situations for which this choice of window size and message number doesn't work.

**What is the minimum number of packets lost which can result in a problem?

6) What are the advantages of selective-repeat ARQ, and why is it sometimes not used?

*7) A link from London to Glasgow has a round-trip delay of four milliseconds, a bit rate of 1 Mbit/s, and a bit error rate of 1 in a million. If a stop-and-wait ARQ scheme is used, what is the utilisation of this link for packets of length 100 bits, and packets of length 1 million bits?

**Is there an optimum length of packet for maximum utilisation, and if so, what is it? (Assume that acknowledgement packets are so short that the probability of a lost acknowledgement is negligible.)

*8) A go-back-N error control scheme is operating over a link with a 10 ms propagation time, using 0.1 ms packets. The probability of packet error is 0.1%, the probability of errors in the acknowledgements can be neglected. There is a seven-bit field in the packet header that is used to store the message number). What is the utilisation of this link?

Hearing that there is a more advanced scheme called selective-repeat, the system is modified to use this protocol. What is the utilisation of the link now?

**9) An error control scheme is designed using go-back-N with a window size of 7 and a maximum sequence number of 7. Will this always work? Can anything go wrong? If so, how? How could you prevent this from causing a problem?