Point-to-Point and Multidrop Links

Norman Matloff

University of California at Davis

September 4, 2001

1 Overview

We will be concerned here with efficiencies of point-to-point and multidrop links. Directly or indirectly, these efficiencies translate to dollars. For instance, a point-to-point link often takes the form of a leased high-quality telephone line, which may be very expensive. In the case of a multidrop link, the line may be owned rather than leased, but if its efficiency is too small, its users will complain, and additional lines will need to be installed. Thus it is of high importance to have some idea of the size of the link's **utilization** u, which is the proportion of time a transmitter on the link is successfully sending data.

2 Latency and Bandwidth

Let us first define and explain the terms **latency** and **bandwidth**, which will be used in our analyses. Latency is defined to be the time a single bit takes to go from the transmitter to the receiver, while bandwidth is the number of bits we can put on the line per unit time, say per second.

These terms sound simple, and they are, but it is easy to get confused if you are not careful. An analogy I like to use involves the San Francisco Bay Bridge (westbound direction). The "latency" of the bridge is the time it takes for a single car to get from one end of the bridge to the other, while the "bandwidth" is the number of cars which can be loaded onto the bridge per unit time. Assuming cars travel a fixed speed, say 65 miles per hour, then the bridge's latency depends on its length. Assuming a fixed number of lanes on the bridge (the bridge is actually a "parallel" communications link, instead of a serial one), then the bandwidth depends on how fast the tolltakers at the eastern end of the bridge can collect tolls.

Getting back to computer networks, let α denote the ratio of a serial link's latency to the time needed to place one frame on the link. This latter quantity is equal to the frame length in bits divided by the bandwidth.[1] An

*Some of this material is adapted from *Data and Computer Communications*, by William Stallings, fourth edition, pub. by Macmillan, 1994.

[1]By the way, note that most books use the symbol **a** instead of α.

important interpretation of α is that it is the number of frames which can be on the link at any given time. As you will see, this will play a major role in the efficiency of the various protocols.

The purpose of this tutorial is to analyze these efficiencies. We will particularly be interested in u, the proportion of time the line is actually in use. If, say, u = 0.1 and the line's bit rate is 56K, then the effective bit rate is only 5.6K.

In all the analyses below, we will choose our time units so that the time to send one frame is 1.0. Note that this means that the latency will then be α.

3 Point-to-Point Links

3.1 Automatic Repeat Request Protocols

Suppose we have a serial point-to-point link. The concerns here are typically one or all of the following:

- **Flow control**: making sure we do not overwhelm a receiver's buffers.
- Preserving the sequential order of the parts of a message.
- Error checking and reporting.

A broad class of methods for dealing with these issues is that of **automatic repeat request protocols** (ARQ). These break a transmitted message down into smaller chunks, with the receiver occasionally sending a message saying, "OK, you can now send me some more chunks, starting with chunk i+1, as I have successfully received chunks up through number i." If the receiver finds that a chunk is in error (or so badly corrupted that it was never received at all), it will notify the transmitter, at which point the latter will have to retransmit one or more chunks.

3.2 Stop-and-Wait ARQ

The name of this protocol derives from the fact that after the transmitter sends out one frame, it stops and waits for a receipt acknowledgement (ACK) from the receiver.

It is easy to determine u for this protocol: Consider a point-to-point link, and suppose the transmitter has a continuing stream of frames available for sending, sending out the first frame at time 0. Call the sender and receiver S and R, respectively. Then here is the sequence of events, assuming no error (if, say, $\alpha < 1$):

- time 0: first bit of frame sent out by S
- time α: first bit of frame received by R
- time 1: last bit of frame sent out by S
- time $1 + \alpha$: last bit of frame received by R; ACK sent by R
- time $1 + 2\alpha$: ACK received by S; S starts sending next frame

(We are assuming that the ACK is very short compared to ordinary frames, so that its transmission time is negligible.)

So, processing of this frame occurs during the time

$$(0, 2\alpha + 1) \tag{1}$$

during which we will have been using the line for only 1 time unit of data transmission. Thus

$$u = \frac{1}{2\alpha + 1} \tag{2}$$

This derivation did not account for possible errors. Suppose there is a probability p that a frame will be received in error (and detected as such), thus probability 1-p of being sent error-free. Then a proportion p of the frame transmissions are wasted, so only a proportion 1-p of the utilization in (2) is really used. In other words, the equation for u after taking errors into account is

$$u = \frac{1 - p}{2\alpha + 1} \tag{3}$$

3.3 Sliding-Window Protocols

The Stop-and-Wait protocol is clearly quite wasteful if α is large. This can be seen not only in the mathematical sense—the value in (2) will be small if α is large, but also intuitively: If α is large, we could have many frames in transit on the link at once, but Stop-and-Wait only allows us to have one frame out there at a time. Our *effective* bit rate is much lower than what the link allows.

One alternative is to allow as many as N frames outstanding, i.e. pending ACKs, at once, instead of just one.[2] Suppose N = 8. Then we would have a 3-bit sequence number in each frame, with our frames being numbered 0,1,2,...,7,0,1,2,..., continuing indefinitely.[3] Both transmitter and receiver would maintain **windows** into this sequence, as follows.

Suppose the transmitter's window currently consists of 6,7,0,1,2,3,4. That means that at this point, it is allowed to send out frames in this range. It does not send frames earlier than 6, because it has already received ACKs for them, and it does not send out any later than 4 because the receiver hasn't said it is ready for them yet. If the transmitter now sends frame 6, the window shrinks to 7,0,1,2,3,4. Then suppose the receiver ACKs, say two frames the transmitter has sent out previously; the window can now expand by two frames, to 7,0,1,2,3,4,5,6. The window continues to shrink and expand in this manner.[4]

[2] In other words, Stop-and-Wait is really a sliding-window protocol with N = 1.

[3] There is a problem in distinguishing among multiple frames of the same number. Suppose the transmitter sends frames 2,...,7,0,1 and they are received correctly, but due to a burst of noise, all the ACKs are lost. The transmitter will **timeout**, i.e. a specified time period will lapse, and the transmitter will retransmit those frames—but the receiver will mistakenly think that these comprise the *next* set of frames of that number. Depending on the type of protocol used, the window size may have to be narrowed for this reason.

[4] This is then the origin of the term **sliding-window** protocol. Keep in mind, though, that this is actually a broad class of protocols, differing in various details.

Similarly, the receiver maintains its own window, starting with the number of the frame it expects to receive next, and ending with the last frame it is willing to receive at the moment (due to buffer space limitations).

Let us derive the value of u under error-free circumstances. Again assume the sender, S, has a large supply of frames it has ready to send. Suppose the first frame is sent out at time 0. Using the same reasoning which led to (2), we have that the ACK for this frame will be received by S at time $2\alpha + 1$. During the intervening time, S has been sending out subsequent frames as well. The central question at this point is, How many frames have we sent out so far? The answer is

$$min(N, 2\alpha + 1). \tag{4}$$

The reason for this is that in $2\alpha + 1$ time, we could send out $2\alpha + 1$ frames, but we are not allowed to have more than N frames outstanding at any time.

For concreteness, say N = 8 and again suppose S starts sending at time 0. At that time its window has the full 8 frames in it. Suppose $\alpha = 5$. Then S receives an ACK for the first frame it sent out—the one it started at time 0—at time $2 \times 5 + 1 = 11$. Now, during the time interval (0,11), S has the capacity to send out 11 frames, but since its maximum window size (say, due to limitations at R's buffer) is only 8, S will send out only 8 frames and then be idle during the time (8,11). This would be a utilization of 8/11.

On the other hand, if N = 16, then at time 11 S would have sent out 11 frames (and still have 5 frames left in its window). S would be busy 11/11 of the time, i.e. u = 1. (By the way, from time 11 on, S would always have its window size at 11, because at each time slot S would send out a new frame but would receive an ACK for an old frame.)

In other words, there are two cases in our analysis:

- $N < 2\alpha + 1$. Here we sent out N frames, but then had to wait $2\alpha + 1 - N$ time for our first ACK.
- $N > 2\alpha + 1$. That means that after sending out $2\alpha + 1$ frames, we still have frames left in this window which we can send out now, so we do not have to sit idle. In this case, we have solved the problem which arose with Stop-and-Wait, because here we never have to wait!

So, in the first case, we are always sending out frames, and u = 1. In the second case, we send out N frames in $2\alpha + 1$ time, so $u = N/(2\alpha + 1)$. In summary:

$$u = \begin{cases} 1, & \text{if } N > 2\alpha + 1 \\ \frac{N}{2\alpha+1}, & \text{if } N < 2\alpha + 1 \end{cases} \tag{5}$$

The term *sliding window* refers to a broad class of protocols. Within this class there is a wide variety of approaches to handling errors. In the next two subsections we will outline two prominent subclasses of **automatic repeat request** (ARQ) methods, Go-Back-N and Selective-Repeat. HDLC offers both, while newer Kermit versions use Selective-Repeat.

As mentioned earlier, TCP uses a kind of sliding-window protocol. It should be noted that in that setting, the value of α is typically quite large, due to queuing delays of a packet as it passes through various routers.

3.3.1 Go-Back-N ARQ

Suppose the transmitter currently has frames 3, 4 and 5 outstanding, and the receiver has successsfully received frames numbered up to and including 2. Suppose now frame 3 reaches the receiver. If the frame is received intact, the receiver will send an ACK(4), meaning that it has correctly received frames through number 3 and now is expecting frame 4. But if frame 3 is received in error, the receiver will send a NAK(3) frame, meaning that frame 3 is negatively acknowledged.[5] Under the Go-Back-N protocol, the sender would have to retransmit all previous frames, even though most were received correctly. (This way the receiver does not have to buffer so many frames.)

We will derive u for the case $N > 2\alpha + 1$. Here the transmitter sends continuously, but $u < 1$ because of retransmittals. We will also assume that α is an integer, and break time into slots of length 1. For concreteness, we will again take as our example N = 16, $\alpha = 5$.

Let us say that this system is in state i if there are i frames sent but not yet ACKed, i = 0,1,...,2α. There is no state $2\alpha + 1$, because the $2\alpha + 1$-th frame will be sent out just as one is ACKed, leaving only 2α still-outstanding frames.

How do we move from state to state? Well, consider our example N = 16, $\alpha = 5$. At time 0 S has sent nothing, so the state i = 0. At time 1, S finishes sending out its first frame, so now we are in state 1. At time 2, we reach state 2, and so on, through time 10 and state 10.

At time 11, though, things get more complex. Just as S sends out its 11th frame, it will receive an ACK or NAK in response to the first of the frames it has sent. If it is an ACK, then there now will be only 10 unacknowledged frames on the line—there momentarily had been 11, but one of them was just ACKed, so now it is only 10. In other words, if R received the first frame correctly, then we will stay in state 10.

On the other hand, if R found the first frame to be in error, then by the rules of Go-Back-N we must now send *all* of our unacknowledged frames again! In other words, if S receives a NAK at time 11, the new state will be 0.

Let π_i denote the long-run proportion of time we are in state i. Let us derive an equation first for $\pi_{2\alpha}$. By the above reasoning, if we are now in state 2α, either

- we were in state 2α in the last time slot and there was no error, or
- we were in state $2\alpha - 1$ in the last time slot

In other words, the probability of being in state 2α, that is $\pi_{2\alpha}$, can be expressed as

$$\pi_{2\alpha} = \pi_{2\alpha}(1 - p) + \pi_{2\alpha-1} \times 1 \tag{6}$$

so that

[5] In some versions of this protocol, the receiver might not send an ACK after every frame. For instance, if frame 3 is received correctly, the receiver may wait until it receives frame 4, and then send ACK(5) or NAK(4), each of which would implicitly be an ACK for frame 3. However, here we will assume a response to every frame. For simplicity, we are also ignoring issues such as corrupted ACKs and so on.

$$\pi_{2\alpha} = \frac{1}{p}\pi_{2\alpha-1} \tag{7}$$

On the other hand, if our current state is i, where $i < 2\alpha$, we will be in state $i + 1$ in the next time slot. So, using the same reasoning which led to (6), we have

$$\pi_i = \pi_{i-1} \tag{8}$$

for i = 1,2,...,$2\alpha - 1$. Among other things, this means we can rewrite (7) as

$$\pi_{2\alpha} = \frac{1}{p}\pi_0 \tag{9}$$

Note that

$$\pi_0 + \pi_1 + ... + \pi_{2\alpha} = 1 \tag{10}$$

Replacing π_1 through $\pi_{2\alpha}$ in this equation by π_0 (due to equation (8), we have

$$(2\alpha)\pi_0 + \frac{1}{p}\pi_0 = 1 \tag{11}$$

This gives us

$$\pi_0 = \frac{1}{2\alpha + \frac{1}{p}} \tag{12}$$

from which (9) implies that

$$\pi_{2\alpha} = \frac{1}{1 + 2\alpha p} \tag{13}$$

Now, only one observation remains to derive u: The quantity u measures the proportion of time slots in which S receives an ACK. This will only happen during time slots in which we are in state 2α, and in fact only during a proportion (1-p) of such time slots. So $u = (1 - p)\pi_{2\alpha}$, that is

$$u = \frac{1 - p}{1 + 2\alpha p} \tag{14}$$

3.3.2 Selective-Repeat ARQ

Here the receiver asks the transmitter to resend only those frames which the receiver found to be in error. This has the disadvantage that the receiver's design must be more complex, but a larger value of u should be obtainable.

Happily, this case is much easier to analyze than Go-Back-N. We simply mimic the reasoning which led to (3). Since each frame now acts independently of the others,[6] we know that each frame must be transmitted, on average, 1/(1-p) times. So, we simply divide (5) by 1/(1-p), yielding

$$u = \begin{cases} 1-p, & \text{if } N > 2\alpha + 1 \\ \frac{N(1-p)}{2\alpha+1}, & \text{if } N < 2\alpha + 1 \end{cases} \tag{15}$$

4 Multidrop Links

The above analyses concerned point-to-point links. Consider instead an multidrop link on which a primary, P, is at one end and s secondaries, $S_1, ..., S_s$, are situated uniformly spaced along the rest of the line.

The primary station controls the show; a secondary station "may not speak until spoken to."

Suppose for instance that the host computer is running a UNIX system, and that four terminals are connected to it. The host's transmitter will repeatedly send **poll** frames to the four terminals, one at a time and cyclically. A poll frame asks the terminal, "Do you have any characters to send?" Suppose that for a while none of the four users at the terminals are typing anything, but eventually the user at terminal 2 starts typing "ls", the UNIX "list files" command. After the user types the 'l', the next time this terminal is polled, it will respond that it does have something to send, namely the 'l'. Then the host will continue its cyclic polling of the four terminals, until the user at terminal 2 types the 's',[7] and so on.

We will define α in terms of the entire end-to-end length of the line, so that the latency for frames going between P and S_i is

$$\frac{i}{s} \cdot \alpha$$

Each of the secondaries is polled, in Round Robin fashion, first S_s, then S_{s-1}, and so on. Each secondary responds either by sending a frame or sending a very short message saying it has no frame to send. In a given Round Robin cycle, first P polls S_s. If S_s has a frame to send to P, it does so, after which P sends an ACK back. Then (whether or not S_s had a frame to send), S_s—not P, which would be slower—sends a poll to S_{s-1}. Then S_{s-1} sends a frame to P (and receives an ACK) if it has one, after which S_{s-1} sends a poll to S_{s-2}. The process continues in this manner until S_1's turn comes, after which a new cycle begins (P polls S_s, etc.).[8]

Consider a period of time during which all secondaries have something to send each time they are polled. Let us see how long one cycle will take, for large s, assuming no errors.

First, the polls: It takes α time for P's poll to reach S_s, and $\frac{1}{s}\alpha$ for each of the other s-1 polls, for a total of

$$\alpha + (s-1)\frac{1}{s}\alpha \approx 2\alpha$$

[6] This is in contrast to in Go-Back-N, in which a frame must be retransmitted if an "upstream" frame was in error, even if the subsequent frame was not in error.

[7] Or until a user at another terminal starts typing.

[8] This technique is called **hub polling**.

(We also have to account for the fact that S_1 will send a "poll" to P if it has nothing to send, to at least let P know its turn has come and gone. But the time for this would be $\frac{1}{s}\alpha$, which will disappear here if s is large.)

Next, the frame transmission times: $s \cdot 1$. Next, the frame latencies: $s \cdot \frac{\alpha}{2}$ (since the "average" secondary is at the halfway point on the line). Finally, the ACK latencies: $s \cdot \frac{\alpha}{2}$.

This all makes for a cycle time of $(2+s)\alpha + s$, during which time the line is used for $s \cdot 1$ actual data transmission time. Therefore,

$$u = \frac{s}{(2+s)\alpha + s} \approx \frac{1}{\alpha + 1} \tag{16}$$

5 Example Protocols

5.1 HDLC

Here we will look at the High-Level Data Link Control (HDLC) protocol. It can handle both point-to-point and multidrop links.

5.1.1 Station Relationships

Stations are connected on a serial line in an HDLC protocol in one of two relationship modes:

- Peer. All the stations play "equal" roles. Typically this is used in the context of just two stations, a point-to-point connection of one computer to another computer.
- Primary/secondary. The primary station is a host computer and the secondaries are terminals, say on a multidrop link.

5.1.2 Frame Format

An HDLC frame consists of:

- Flag: a pattern 01111110, indicating the start of the frame
- Adress: 8 or 16 bits of addressing information; in a primary/secondary setting, for instance, the address is that of the secondary which is either being sent to by the primary or sending to the primary
- Control: 8 or 16 bits of control information, such as frame type (e.g. poll; see below), window sequence number (using either the Go Back N or Selective Repeat protocol), and so on
- Data: a variable number of data bits
- CRC: a 16- or 32-bit error-checking field
- Flag: again in the pattern 01111110, indicating the end of the frame

A problem arises in that the data to be transmitted may contain the bit pattern 01111110. How is the receiver to know that this is part of the data, as opposed to the end of the frame? This problem is solved by the use of **bit stuffing**: When the transmitter notices that it is supposed to send five consecutive 1s in the data, it inserts an extra 0 after the fifth one, to indicate that these 1s really are data. The receiver is designed to remove any 0 it sees immediately following five consecutive 1s. This solution works, because when the transmitter really does send a flag, it will not stuff the extra 0, and thus the receiver will not remove it, and will receive the flag intact.

The flags are used to help the receiver keep its clock synchronized with that of the transmitter. For this reason, the transmitter will send flags even during idle periods.

5.2 PPP

5.2.1 Typical Contexts

You may already have heard of the Point-to-Point Protocol (PPP), which many people use to establish their home PC as a temporary Internet node via a phone-line connection to an Internet service provider (ISP).

PPP is also used extensively in the Internet itself. Recall that the Internet consists of tying together many different LANs (Ethernets, etc.). The ties take on two forms:

- Having one computer in common to two LANs.
- Having a point-to-point link, say a phone line, between a computer on one LAN and a computer on the other LAN.

In the latter case, the connection betwen the two computers is typically managed by PPP.

5.2.2 Operation

PPP works like HDLC's peer-to-peer mode, and even uses HDLC's frame format which we saw in Section 5.1.2:

- Flag.
- Address: Since there can be only one recipient of a frame, the field consists of 11111111.
- Control: 00000011, HDLC's code for "unnumbered frame."
- Data: Subdivided into two subfields, Protocol and Information (see below).
- CRC.
- Flag.

The Protocol field, one byte long, is used for multiplexing/demultiplexing. Sessions from several different protocols could be sharing this link. A common one is of course IP, coded 0x0021. All Interent packets sent through this link would have the Protocol field equal to 0x0021. This is a major reason underlying the formulation of PPP: HDLC did not allow for multiple protocols using the same link, so PPP was created.

The remainder of the Data field, called the Information subfield, is the actual data, e.g. the actual IP packet. This must consist of an integral number of bytes.

Like HDLC, PPP uses insertion/removal of fake data to solve the problem of the possibility that the data may contain a flag. However, PPP is based on bytes rather than bits, it uses byte stuffing instead of bit stuffing: If the data contains a flag, i.e. 0x7e, the flag is replaced by 0x7d5e. If the data contains 0x7d, that is replaced by 0x7d5d. On the receiving end, the opposite changes are made.

5.3 TCP

TCP uses a sliding-window protocol to send out the "chunks" of a message, called **segments**. Each byte in the entire message is given a **sequence number**. The receiver will tell the sender which byte number it expects next, and how many bytes it will accept.

Note that in the TCP context, the point-to-point-link nature of our discussion here is virtual, as a segment will typically traverse many networks in going from source to destination.