Fundamentals of Error and Flow Control Protocols

Introduction

As indicated earlier, we are going to use the IPC Model as our guide for digging into the details of networking. We are going to start with what is commonly thought to be the subject of networking courses, the design of data transfer protocols. Or, in our terms, the problem of what goes into error and flow control protocols and why. While the reading in Tanenbaum is the chapter on the Data Link Layer, in this class we are going to be considering the general case protocols independent of what layer they occur in. First of all, the point-to-point data link protocol is basically the case we first encountered in developing the IPC Model, and second, the protocols don't differ all that much. Tanenbaum's treatment of data link protocols is a bit historical, but we won't let that get in the way of considering more recent results. This then fits with the first two steps in the IPC Model as well. (In a later chapter, Tanenbaum considers Media-Access or Multi-Access Protocols that are Layer 2 and do require what are often called *addresses*.) Keep in mind that while the readings in Tanenbaum covers Layer 2, this lecture is covering all error and flow control protocols regardless of what layer they are used in.

In the development of the IPC Model, as soon as we went to two applications in two computers, we encountered the problem that bad things can happen to the data while it is in transit. Our messages, our PDUs, can be damaged and the data corrupted. In some cases, whole PDUs can be lost. Damaged or lost PDUs may require that the PDUs be sent again (*retransmission control*). The receiver will have to be able to tell the sender what PDUs could be used. In addition, the receiver has to be able to tell the sender to not send so fast, *flow control*.

Both retransmission and flow control are feedback mechanisms, which implies that there will need to be coordination between the sender and receiver, i.e., synchronization. In the initial case of IPC in a single computer, there were various shared-memory schemes and critical-section techniques that could be brought to bear, but between two computers, we don't have that capability. So, we will have to find another way.

Let's get started.

Phases of Communication

Consistency?

We didn't really cover enrollment in developing the IPC Model in Module 1. I could be accused of making the same mistake.

But I have an excuse! In that lecture, we assumed that the IPC Facility already existed and was part of the Operating System. We didn't go into much detail on how it was

done, because that is covered in a different course!

All communication goes through three phases: enrollment, allocation, and data transfer. First, we will have a somewhat formal definition followed by some explanation.

The Enrollment Phase is the phase that creates sufficient state within the network to allow an instance of communication to be created.

> The *enrollment phase* creates, maintains, distributes, and deletes the information within a layer that is necessary to create instances of communication. This phase makes an object and its capabilities known to the network, any addressing information is entered into the appropriate directories (and routing tables), certain parameters that characterize the communication this protocol can participate in are set, access-control rules are established, ranges of policy are fixed, and so on.
>
> The enrollment phase is used to create the necessary information for classes or types of communication. However, in some cases (for instance, multicast and some security services), enrollment specifies information for a particular instance (that is, flow). The enrollment phase has always been there but often ignored because it was part of the messy initial configuration and setup (which was often manual). Frankly, it just wasn't fun and had more in common with having to clean up your room than solid engineering! In general, enrollment is performed by ad hoc means (often manual) or by application protocols. A given enrollment protocol or procedure will generally be used for more than one data transfer or application protocol. The enrollment phase creates an information base of parameters and policies that will be used to instantiate particular PMs. When the PMs are created in the establishment phase, they will inherit the set of attributes associated with their protocol that were recorded during the enrollment phase. These attributes may be modified by the allocation phase parameters and subsequent operation of the PM during the data transfer phase. In practice, this phase is often characterized by two subphases: registration and activation.
>
> The *registration* operation makes the information necessary to create an instance available within the network (that is, distributed to directories in the network). The information is available only to systems within the scope of this protocol and its layer. *Deregistration* deletes the registration of the protocol from the network. In general, deregistration should be withheld until all existing instances have exited the allocation phase. There are no active instances of the protocol; that is, there are no active flows.
>
> In general, it is useful to separately control the registration and the actual availability of the protocol to participate in communication. *Activation/deactivation* is the traditional operation of taking a facility "offline" without deleting the system's knowledge that the facility exists. If a protocol has been registered but not activated, instances (PMs) cannot be created that can enter the allocation phase. Deactivation, in general, does not affect currently existing instances in the allocation or data transfer phases but does prevent new ones from being created.
>
> De-enrollment is synonymous with deregistration. Completion of deregistration completes de-enrollment. De-enrollment may not have any effect on PMs in the allocation or data transfer phases unless the PMs refer back to the original enrollment information, in which case they will abort.

To date, most architectures have relied on ad hoc procedures for enrollment, but that is beginning to change as the networks become more dynamic. For example, 802.11 has an enrollment operation that is very close to what is described here. The registration and activation operations and their inverses may be performed by network management, as in setting up permanent virtual circuits or setting up a connectionless flow. In some cases, the enrollment phase is performed when someone calls up someone else and says, “Initiate such and such so that we can communicate” or a standard that defines “well-known” sockets on which a listen is to be posted for communication with a particular application to take place. *Dynamic Host Configuration Protocol* (DHCP, RFC 1541), the assignment of MAC addresses, well-known sockets, key management, and such are all examples of aspects of enrollment. HDLC has included mechanisms for enrollment in the *Exchange Identification* (XID) frames used to select options to be made available, although this combines aspects of enrollment and allocation. With the advent and use of directory protocols and address-assignment protocols, the enrollment phase is becoming much less ad hoc and much more of a regular automated phase.

The Allocation Phase creates sufficient shared state among instances to support the functions of the data transfer phase.

This is the task of allocating resources for communication. This phase is sometimes called Connect, Open, or Establishment. *Allocate* is used here for two reasons: 1) it is a neutral term that does not get into what the task consists of and 2) it really is what it is all about: allocating resources for an instance of communication. (The other terms tend to bring up issues in the highly controversial connection/connectionless debate. *Allocate* avoids that controversy.) More on that later.

The primary purpose of this phase is to create the initial shared state in the communicating PMs to support the functions of the protocol. The allocation phase ensures that the PMs initially have consistent state information. (Although *consistent* does not necessarily imply the same state information.) The behavior associated with this phase can range from simply creating bindings between the (N+1)-PM and the (N)-PM (connectionless) to an explicit exchange of initial state information to support synchronization between two PMs (so-called connections) depending on the amount of shared state required to support the functions of the data transfer phase, primarily feedback functions. It is during this phase that the specific QoS requirements for data transfer that are acceptable to the user are made (or modified) if they were not fixed during the enrollment phase. While beyond the scope of this initial section, we will find later that for data transfer, allocation and synchronization are distinct. For data transfer protocols desynchronization will occur without explicit action, while DeAllocation will be explicitly terminated by the user of the layer. Application protocols may require an explicit desynchronization operation.

The mechanisms used for the synchronization phase depend on the mechanisms in the protocol that require synchronization. The stronger the coupling of the shared state, the more reliable the synchronization phase mechanisms must be. In general, protocols with feedback mechanisms require more robust synchronization procedures than those without.

And then of course, there is the **Data Transfer Phase,** which provides the actual transfer of data and functions to support it and where the real work is done. This is where the mechanisms of the protocol come into play. The

data transfer phase is entered when the actual transfer of data is affected according to the requested QoS among the addresses specified during either of the previous two phases. For application protocols, the Data Transfer Phase may be further subdivided into specialized subphases.

Layers and the Service They Provide

Every protocol design has two elements:

First is the service or interface definition. From the outside, the layer is seen as a black box. The black box hides the internal workings of the protocol. The services are those characteristics observable by the user of the black box, e.g., a protocol or a layer. It is those things that are externally visible. Not all of the functions that are used within the layer have a visible effect outside. This hides the complexity of the internal workings and presents a simpler service to the user of the layer, which may be another layer or an application. Consequently, we will distinguish the *service* a layer provides, i.e., those characteristics visible above the layer boundary, from the *functions* of the layer, which are performed by the layer, and the effect may not be overtly visible. For example, there are several functions that contribute to reliable data transfer. All the user of the layer sees is that the flow is reliable. It can't tell which functions did something or not. It could be that everything that was received from the layer below was perfect.

Second are the internal workings of the layer—what goes on inside the black box. A major function of the layer is the data transfer protocol definition. Some make the mistake of thinking that the data transfer protocol is the only function within the layer. As we have already seen, this is not the case. A layer is a distributed resource allocator, a distributed application that does InterProcess Communication (IPC). Later we will explore the other functions within the layer, such as the enrollment procedures, resource allocation, security, routing, etc.

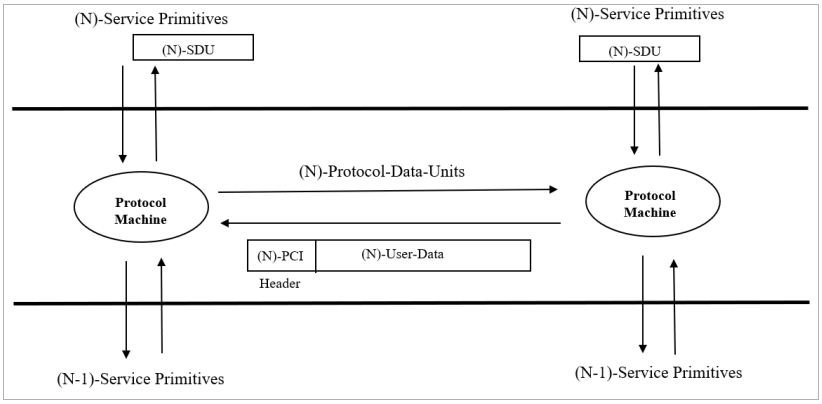
One aspect that we have already seen an example of and that is often overlooked is that the configuration of any layer assumes a minimal service from the layer below. If the layer below is unable to meet that minimal service, it must be augmented to bring it into the range expected by the layer above.

Services and Protocols

Every book talks about the service that a layer provides. What is just as important and few, if any, textbooks emphasize is the minimal service that an (N)-layer assumes from the (N-1)-layer. Any new lower layers must meet those requirements, otherwise there will be problems, ranging from loss of efficiency, greater delay, less throughput, to complete failure. If new (N-1)-layers don't inherently provide the minimal service, they will have to be augmented to bring the service presented to the (N)-layer up to the minimal level required. That usually will entail an overlay protocol.

The Nature of the Service (or Layer) Boundary

Earlier service was used as the abstraction of the interface, hence, the term *service-data-unit* (SDU) is used to refer to the unit of data provided to the PM by the (N+1)-PM across the service boundary (reserving the term *interface* for the implementation-specific case, such as a UNIX interface or a Windows interface). To a PM, an SDU consists entirely of user data but will have a size that is convenient to the (N+1)-PM. The service primitive invoked to pass the SDU to the PM will also pass other parameters to the PM for handling the SDU. Generally, one of these is a local “port-id” that identifies this end of the flow or the connection this SDU is to be sent on. The port-id is local to the system (that is, only known within the system and only unambiguous within it) and shared by the (N+1)- and (N)-PMs to refer to related communications. The PM may have to segment the SDU into several PDUs or may aggregate several SDUs into a single PDU so that the PDU is a size convenient to the requirements of the (N)-protocol. (This nomenclature follows the OSI Reference Model, not because I favor the OSI model, which I don’t, but because it is an existing nomenclature that tries to define common terms. I can find no good reason to invent new terms other than for the sake of generating new terms. For those who still have a visceral reaction to anything OSI, all I can say is, OSI is long dead, get over it.)



PDUs (Protocol-Data-Units) are also known as messages, packets, segments, frames, cells, etc.

While not rigid, the terms are generally used with specific layers.

PDU stresses that they are really all the same thing.

(SDU = Service Data Unit)

Why Service Primitives?

There is a bit of history on why the elements of an API are called *service primitives*. Why not API? That is what it is! The standards committee world said that APIs are specific to a language or operating system (which they are). Therefore, programming language committees define language-specific aspects. It is not the job of a networking committee. It was pointed out that a networking committee must have something, otherwise the protocol can’t be specified. The API is what drives the protocol. Consequently, service primitives were invented as the abstraction of an API. That said, in addition the abstraction is limited to just those primitives that are necessary to interact

with the internal operation of the black box. Consequently, a Service Definition does not define the whole API. Any API interactions that would only be local, e.g., Status, and not really cause the protocol to send a PDU or change state, are not part of the Service Definition.

The terms used here are from the OSI Reference Model. We have already introduced PDU. Now we will have service primitives as an abstract system-independent API, which pass Service-Data-Units (SDU) across the layer boundary. Once again, we have chosen to use these to emphasize the commonality across protocols.

The protocol state machine is going to operate on the SDU along with other parameters passed by the service primitives to formulate a PDU to send to its peer. We said that the PDU does not go directly from one protocol machine to its peer. The PDU is actually going to be passed down until it reaches the physical media and then across to another system and back up to its peer. However, when talking about the behavior of the protocol, we are going to talk as if PDUs do pass directly straight back and forth.

In some protocols, one of the questions in the design is the integrity: are the bounds of the SDU maintained end to end? In essence, are the 500 bytes passed to the (N)-layer the same 500 bytes delivered as a unit at the peer? Some protocols (layers) may ensure that whatever is delivered to the layer is what the layer delivers. Some protocols do not maintain this integrity. Five hundred bytes may be passed to the protocol, but how the 500 bytes are delivered is entirely arbitrary: 2 PDUs of 250 bytes; 2 PDUs of 346 and 154; 3 PDUs of 100, 200, and 300; or 10 PDUs of 50 bytes, etc. We will look at this in more detail later. An SDU is mapped to the User-Data field of one or more PDUs.

Nota Bene

There has been an alarming trend in some client/server applications to call the messages that pass back and forth an API. This misunderstanding arises because for some applications there is a one-to-one correspondence between the primitives at the boundary with the black box and the internal operation. This occurs with a simple subset of applications. It can be a major mistake assuming that all applications work this way.

What Are the Services That a Data Transfer Protocol Provides?

Allocation of communication resources: Listen, Connect, Disconnect.

Data Transfer: Send, Receive.

Is the Data Transfer Phase an undifferentiated stream, record, or idempotent?

- *Stream*: The data transfer is treated as an undifferentiated stream of bytes (see above). The sender may send 1000 bytes. When the application does a Receive, it may get whatever number of bytes is convenient to the protocol machine. If there is a structure to the data, i.e., some number of bytes are

significant to the application, the application has to determine that, which is not entirely unreasonable.

- *Record*: There were early protocols that did what was called *record* mode. The PDUs always had to have a fixed length regardless. It was very hard to deal with, and it disappeared fairly quickly.
- *Idempotent*: This is what the SDU is. If the sending application sends X bytes as a unit to the layer, the same X bytes are delivered to the receiving application when it does a Receive. Hence, an application can send a message that is meaningful to it and know that that message is what will be delivered.

Some of the protocols we will look at will do *stream* and some of them are *idempotent*.

Quality of Service parameters describing the characteristics of the flow, e.g., delay, jitter, loss rate, etc.

Connection/connectionless, contrary to what Tanenbaum says, is not a service of the layer. Whether either method is used should be invisible to the user of the layer. It's a function.

We will consider all of these in more detail as we progress.

Diving Into Stream vs. SDU (Idempotent)

One of the enduring debates in protocol design is, given that the (N+1)-PM delivered an (N)-SDU of a particular size to the (N)-PM and that under some conditions it may have been fragmented or concatenated *en route,* what does the (N)-PM deliver to the remote (N+1)-PM? What was sent or what was received?

Although it may often be the case that an SDU would be a single (N+1)-PDU, it might be more than one. Seldom would it only be part of a PDU. In any case, the (N)-SDU was a unit that the (N+1)-PM found to be significant for its processing. The early debate was between record and stream modes, derived from early operating system practices. The older mainframe systems tended to operate on fixed-length records, whereas more modern systems such as Sigma 7, Tenex, Multics, and its derivative UNIX communicated in terms of undifferentiated byte streams. Record mode was always considered as something that simply had to be lived with. There was general agreement that record mode was too inflexible and cumbersome.

Stream mode was considered a much more flexible, elegant approach that provided greater layer independence. A stream might deliver any combination from whole SDUs to pieces of an SDU to multiple SDUs or even part of two SDUs. Stream mode requires that the (N+1)-layer be able to recognize the beginning and end of its SDU/PDUs and be able to assemble them for processing. The (N+1)-protocol must have a delimiting mechanism and cannot rely on the layer below to tell it where the beginning and end of the PDU are.

CYCLADES came to a different solution, that the *identity* of SDUs was maintained between the sending and receiving users. CYCLADES called it “letter,” which isn’t that descriptive. We will call it *idempotent,* referring to its property of maintaining the identity of the SDU invariant. Because SDUs may be of any length, this differs significantly from traditional fixed-length record mode. This mode requires that the (N)-layer deliver SDUs in the

form it received them. If the (N)-protocol needs to fragment an SDU, it is (N)-protocol's responsibility to put things back the way it found them before delivering the SDU to the (N+1)-PM. There is something compelling about a "do anything you want but clean up your mess when you're done" approach!

This form is more consistent with good programming practice. Similarly, if the (N)-protocol combines several SDUs into a single PDU for its own reasons, it must deliver them as separate SDUs to the remote user. Consequently, the (N+1)-PM does not have to understand (or be modified for) every potential (N)-PM fragmenting or concatenation condition, nor make assumptions about what the (N)-PM will do. Maintaining the identity of SDUs maintains symmetry in an architecture. And symmetry is always good.[1] But, it does require the assumption that the layer below is well behaved. The essential difference between the two is that the idempotent mode is a *user's* point of view, whereas stream mode is more the *implementer's* point of view.

It makes no difference in the receiving system—the amount of work is the same: Either the receiving PM or the receiving user, the (N+1)-PM, must do the reassembly. In other words, the work is either done at the bottom of the (N+1)-layer (stream) or the top of the (N)-layer (idempotent). There are no strong logical or architectural arguments for one or the other. Although if it is done by the (N+1)-PM, it may have to be implemented several times (if there are many (N+1)-PMs i.e., applications). Then, good software engineering practice supports the (N)-PM performing the function.

That said, it will be easier for protocols with sequence space granularity of bytes to do stream mode (for instance, TCP), and it will be more work to keep track of where the SDU boundaries are. For protocols that do sequencing to the granularity of PDUs, the amount of work is the same if there is no concatenation. If the protocol concatenates, however, it must be able to find the boundaries between SDUs. In this case, stream mode will be less work for the (N)-protocol. Overall, the work as seen by the system is the same.[2]

Good Designs Are Never Obsolete!

This illustrates why it is important to study good designs. Here we have used the Telnet half-duplex solution (see Module 1) to solve what appears to be an either/or choice. Many students would complain about being taught the Telnet solution: "Why are we wasting time on this! Half-duplex terminals are a thing of the past! I will never need this!" Perhaps not, but as you have just seen, the form of the problem recurs, and so the same solution in a somewhat different guise can be applied.

But it does not have to be an either/or choice. It is possible to provide both. For the solution, we take a lesson from how Telnet modeled half and full duplex as degenerate cases of a single mechanism, and from the glib comment that "all data communications is a side effect" (see below). We just note that a stream is simply a very long SDU! If the protocol has the ability to indicate the boundaries of SDUs and negotiates whether it may deliver partial SDUs (in order) to the (N+1)-layer, the (N+1)-layer can have either interface discipline. Stream mode negotiates partial delivery and at a minimum indicates the end of an SDU only on the last PDU sent. Idempotent mode negotiates no partial delivery and indicates the end of SDUs at the appropriate times. Flags in the protocol might be defined as shown here:

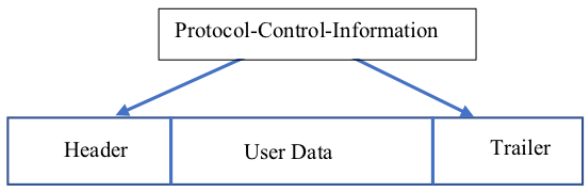
Supporting Both Stream and Idempotent

Delivery of Incomplete SDUs Allowed	More Data	Description
0	0	Self-contained PDU equivalent to Don't Fragment
0	1	Idempotent
0	1	Stream (with huge buffers)
1	1	Stream

Protocol Data Units

While we introduced PDUs in the first module, this is a good time to consider them in more detail. As noted earlier, to communicate from one place to another, finite quanta of information must be exchanged. Over the years, these "finite quanta" have been given a variety of names, such as frame, cell, packet, segment, message, and so on, depending on the inclination of the author and the kind of protocol. All of these are different terms for the same concept. To stress the commonality, we adopted the neutral term *protocol data unit* (PDU).

The Structure of a PDU



The structure of a PDU consists of three major elements: a header; less frequently a trailer, to carry the information necessary to coordinate the PMs; and the user data. Notice that the first two of these are labeled Protocol-Control-*Information* in the figure above. This is an important distinction: "Information" is what is understood by the PM, and "data" is what is not understood (and usually passed to the PM or application above). To remind us of this, we refer to the part that is understood by the (N)-PM (the header and trailer) as *Protocol-Control-Information* (PCI), and the user's data as *User-Data* because the (N)-PM does not understand it. This distinction is clearly relative. What is information (PCI) to the (N+1)-PM is merely part of the data to the (N)-PM. Similarly, the (N)-PCI is merely more user-data to the (N–1)-PM and so on. This distinction is crucial to much of what follows. It is important that one always be clear about what is information and what is data at any given point.

It's a Side Effect!

Thus, we see that information is what the PM understands, and data is what it doesn't. When a PM receives a PDU, it happily goes along processing each of the elements of PCI, updating its state and generating new PDUs until it reaches the stuff it doesn't understand. Then, it shrugs and throws this incomprehensible junk (user-data) over the

wall (to the (N+1)-PM) and happily goes back to processing the stuff it does understand.

Data transfer is a side effect!

PDUs are sometimes likened to processor instructions: Based on the parameters of the instruction (PCI) and the state of the processor (PM), the execution of PDUs performs operations on the state of the processor (PM). Unlike instructions, which carry an address to reference the data on which they operate, PDUs must carry the data themselves.

Various types of PDUs are used to transfer PCI among peer PMs. These PDUs may or may not contain user data. There is no architectural limit to the size of these PDUs. There might, however, be engineering considerations that impose size limitations on the PDUs in specific environments. For example, for a protocol operating in an error-prone environment, a smaller PDU size may increase the probability that a PDU will be received error free, or that the overhead for retransmission will be minimized. In a network of real-time sensors, the systems may have very limited buffer space, so smaller PDUs may be necessary.

Headers

Most PCI is contained in the header. Most fields in data transfer protocols are fixed length to simplify processing. Fixed-length fields generally precede any variable-length fields. A length field that gives the total length of the PDU is strongly recommended. The header of any protocol should have a protocol identifier to identify the type of protocol and a protocol version to identify the version of the protocol, as well as a field that indicates the function of the PDU. The PCI will also include a field that encodes the action associated with the PDU (for example, set, get, connect). Like instructions, this field may be either horizontally or vertically encoded; that is, it may consist of either a string of control bits, each indicating functions of the processor to be invoked, or an opcode or type field, which stands for the specific combination of functions. In general, horizontal encoding requires more space than vertical encoding because there are generally many combinations of bits that are not legal. Horizontal encoding is generally faster in hardware, whereas vertical is faster in software. TCP uses horizontal encoding. This was an experiment that has not stood the test of time. If the implementation treats the control bits as control bits, it leads to less-efficient implementation. Papers (Clark et al., 1989) have recommended treating them as a type, and this seems to be what most implementations do. Consequently, opcodes are generally recommended over control bits.

Trailers

The PDUs of some protocols have a trailer. The most common use is to carry an error detection code, such as a *cyclic redundancy code* (CRC). The advantage of the CRC in the trailer is that the CRC can be computed as each byte arrives without waiting for the whole PDU to be received. When the trailer arrives and is incorporated into the calculation, if the result is zero, then the PDU has no detected errors; otherwise the PDU is discarded. Generally, the use of a trailer is found in protocols operating near the physical media. For higher layers, where a

PDU is already in memory, there are fewer advantages of a trailer. Consequently, the use of trailers in protocols higher up is infrequent.

The general guidelines for the use of a trailer might be characterized as follows, but it is important to stress that it isn't so much the absolute characterization of the conditions as their relation to each other:

- The information in a trailer is such that it cannot be known at the time the header is created; that is, it is a function of the header and the user data.
- The processing time for the PDU is much less than the time required for the PDU to be sent or received, and the delay thus incurred would be a significant fraction of the delay quota for the traffic.

---

1. Yes, similar arguments can be made for stream. However, the argument that "I should get things back in the same form I gave them to you" is reasonable. Stream may impose additional overhead on the (N+1)-protocol that from its point of view is unnecessary (It knows what it is doing; why should it be penalized because the supporting protocol doesn't?)
2. Having been a strong proponent of stream-mode from the beginning of the Internet, I have spent considerable thought coming to this conclusion. *Fixed record* was clearly not a good idea; and although stream mode is elegant, it does ignore our responsibility to the "user" to clean up our mess.

Constructing Protocols

As it turns out, there are two kinds of protocols:

Application protocols modified the state external to the protocol,
Data transfer protocols modify the state internal to the protocol.

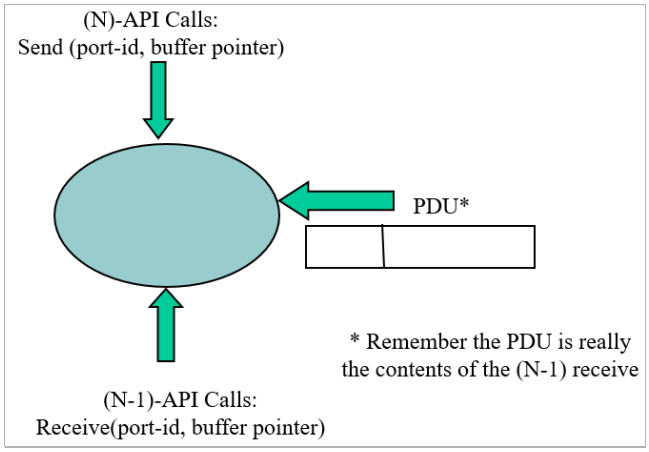
This is a very important distinction. We will cover application protocols later and we will find that not all application protocols are in the application layer. But for now, we will concentrate on data transfer protocols.

Implementing a Protocol

Finite State Machines vs. Threads

We need a model of the operation of a protocol and the means to precisely define it, so that others can implement it successfully. Generally, the most successful way we have found to define it is using a roughly finite state machine definition. There are other ways, but the finite state machine seems to be the one that most people are comfortable with.

What Does an Implementation Look Like?

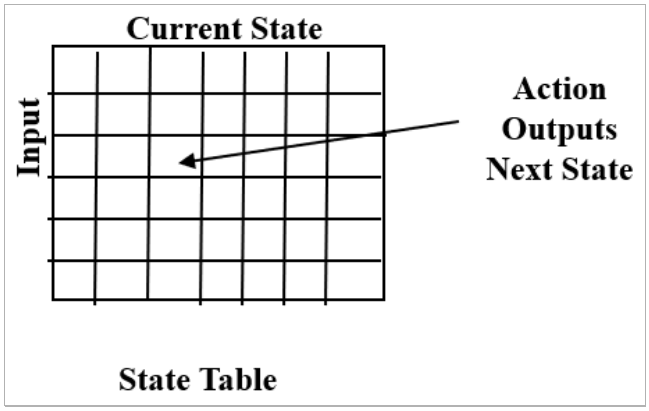


Remember the PDU is really the contents of the (N-1) Receive.

There are inputs from above, from below and from the side. Each function uses information in the PCI (the header), but this is overhead, so we want to keep it minimal. The PCI is used to update and maintain distributed shared state. The implementation is best represented as a finite state machine. It can't block on one interface or the other. There could be an input from above or below at any time.

As noted in the course on operating systems, one does a finite state machine design by starting with a state matrix. We list the inputs down one side of the matrix and list the states across the other side. Then for each cell of the matrix, we write what it does if it gets a certain input in this state, what the output is, and what the next state will be. This turns out to be a very nice way of implementing the protocol. It makes it very mechanical; the amount of code to be written at any one time is not very great. Consequently, there is less chance of introducing bugs (and if they are introduced, they are often the easy ones to find).

What Does an Implementation Look Like?



Remember, the PDU is really the contents of the (N-1) receive.

Some prefer using threads to implement something like this. (Tanenbaum among them.)

Threads and state machines are duals. In a sense, a thread is a path through the state machine. Think about

how a thread implementation would work: It gets an input from an interface. after some processing transitions to the next state, sends a PDU and blocks waiting for a response. When the response arrives, there is more processing; another PDU is sent and blocks waiting for a response and so on. Meanwhile other inputs may be arriving from the layer above or from the remote side.

In a threads implementation, because the inputs can occur at any time, it can't block waiting for a particular event. It can't be blocked waiting on the input that is expected based on what you the observer think will be next. The protocol machine doesn't know what may actually show up. There must be a separate thread for every source of inputs. Where the threads cross, there will be critical sections, creating more complexity.

I prefer state machines, as you probably already guessed. Whether using threads or state machines, it is roughly the same amount of code. But the big difference is that with threads *I* have to think of all the weird cases that might occur and make sure that there is a thread that handles it. With a state machine implementation, I just mechanically go through every cell in the matrix saying, "Okay, if this happens, what do I do?" If I go through the matrix and answer the question for each one of these cells, *nothing else can happen*. (It may turn out that a lot of them are, "if this happens, something really bad has happened! Pull the chain and get the heck out of here!" But you have answered the question.) You have covered all of the bases; there are no other bases to cover. The amount of code to write for each cell, even when there is something to do, is not that great, so the probability of errors is reduced.

A long time ago, we did an implementation of HDLC using this approach with a brand-new hire. It took six weeks to design and write the code. It took three days to debug it because there was basically nothing that could go wrong. Even the bugs that were there were all really small, silly mistakes. You have undoubtedly seen that: as soon as the bugs are fixed, lots of stuff falls into place. It means more time in design (which makes managers nervous). But spending more time on design than implementation pays off in the end. (As a coda to that story, when it came to doing the documentation, the programmer came to me and asked, "I could write a program that reads the state table and generates the documentation, right?" Yes, you could. And he did! Even better, there are several places in HDLC where it is optional what his done. When this was demonstrated to the customer, they said 'in that case we would like it to this (different thing). The programmer immediately made a small change to the state table and it was done. The customer was very impressed. They could see that maintaining or modifying the code would be easy.)

Structuring the Implementation

One of the prime rules we have come to learn over the years is: "Do What the Problem Says." It is smarter than we are. There is no point in fighting it. Sooner or later it will get its way. The longer we try to patch and avoid it, the more costly it will get and the more costly it will be to finally admit that we should have done what it said to begin with. Therefore, a protocol machine must interpret four inputs:

1. Interactions with the upper interface
2. PDUs from its corresponding PM(s)
3. Interactions with the local system

4. Interactions with the lower interface

All of these can be considered to be equivalent to procedure or system calls of the following form:

<procedure name>(<param 1>,<param i>*)[3]

The PDUs can be seen as procedure calls in that the PDU type is the name of the procedure and the elements of the PDU (that is, PCI and user-data) are the parameters:

<PDU type>(<PCI element><PCI element>*, user-data)

Associated with each of these are actions to be taken depending on the state of the PM (that is, the body of the procedure). The action taken by each procedure is to interpret the parameters and update the state vector associated with the PM and possibly cause other PDUs to be sent or interactions with the local system or the upper and lower interfaces to occur. Coordinating all of this is a control function or state machine that enforces the proper sequencing of these actions according to the state of the PM.

The local interactions access local resources, such as memory (buffer) or timer requests. Processing the PDUs invokes the mechanisms of the protocol. In some protocols, a distinct set of PDUs is associated with a particular mechanism, forming a module. Such a module is formed from a subset of PDUs and their procedures with the module's state machine. Strictly speaking, any module such as this is a protocol. These service primitives and PDUs invoke the mechanisms of the protocol. For data transfer protocols, the types of interface procedures are all the same: synchronize/finish and send/receive with appropriate parameter values.

The primary task of a PM is to maintain shared state. There are three loci of shared state in any PM: the upper- and lower-interface loci, or *bindings,* between an (N)-PM and an (N+1)-PM and between an (N)-PM and an (N–1)-PM; and the protocol locus, often called a *connection* or *flow,* between apposite (N)-PMs. The shared state is maintained by the exchange of PCI. The shared state between PMs in adjacent layers is maintained by the parameters exchanged in procedure or system calls between layered PMs in the same system, whereas the shared state between apposite PMs in different systems is maintained by the exchange of protocol PCI. The primary difference between the two kinds of shared state is that for a connection or flow, PCI may be lost. For an interface binding, it can be assumed that the exchange of PCI is reliable, and often, shared access to memory can be assumed.

State changes in a PM are caused by inputs from the upper interface (for example, an application or (N+1)-PM, or from PDUs from the apposite PMs, or from the local layer apparatus, for instance, timeouts). Whether inputs from the layer below cause changes in state is a matter of policy. In other words, if an (N–1)-flow should be deallocated or fail unexpectedly, it is a matter of policy whether this causes a subsequent state change in the (N)-flows multiplexed on the (N–1)-flow. It might not.

Separation of Mechanism and Policy

For the last 50 years we have been designing network protocols, and reading a *lot* of protocol specs. In that time, the set of functions has not changed much. There is a small set of functions that has been found to be very

useful. There hasn't been a new function appear in decades.

One of the things that is really curious is that every operating system textbook stresses the importance of separating mechanism policy. For example, the machinery for changing processes in an OS is the same, what differs (the policy) is deciding what process to run next: First In First Out, Round Robin, etc. Or the machinery for moving pages in and out of main memory is always the same. What differs (the policy) is deciding what pages to replace and what to bring in. Managing a disk is the same, ordering the disk requests is what differs, etc.

I have always believed that this was really important and useful. In the 1970s, Bill Wulf initially proposed this idea when they were doing research on multi-processor architectures at CMU. They were really onto something. However, I have never seen a networking textbook mention it. Not even once. Yet it applies even better to protocols in networking than it does in operating systems. We are seeing the same mechanisms with different policies, time after time after time. We are going to find that separating mechanism, and policy is going to be very important.

What do I mean by separating mechanism and policy in protocols? The classic example I always give is the bookkeeping to do the acknowledgments is mechanism, but when an acknowledgement is sent, is policy. There is always a field in the PCI for error detection. That is mechanism. Which error code is used is policy. Flow control is mechanism. The bookkeeping is the same, but how much can be sent is policy, etc.

A non-trivial example: We always thought we would do a different version of transport protocol for voice because you care about order, but you don't care about small gaps in the data stream. The listener won't hear them, so it isn't a big deal if they are missing. There isn't time to do a retransmission anyway. Then one day I realized that we don't need a new protocol, we just change the Ack policy.

We lie.

There is nothing in these protocols that says, you have to tell the truth. If you get data and there is a small gap, but it isn't big enough to worry about, send an Ack anyway. *Who is going to know!?* The protocol will work just fine. *But what that said was that we had the semantics of Ack wrong all these years*. Ack doesn't mean "I got it." Ack means "I'm not going to ask the sender to retransmit it!" That may seem like a small difference, but it can have major implications. We will be referring to this function as "retransmission and flow control", rather than Ack and flow control.

We will repeat what we said about the differences between mechanism and policy, using the example of acknowledgment. The bookkeeping necessary for acknowledgments: Keeping track of what has been received, what has not been received, and what has arrived in order, that's always the same. That is the mechanism. What varies in protocol is when an acknowledgement is sent. In every packet? In every third packet? After waiting so many milliseconds? Etc. That is the policy.

Every function can be separated into an invariant mechanism and one or more variable policies.

A protocol is composed of a set of functions that achieve the basic requirements of that protocol, whether that is error control, reading a file, flow control, two-phase commit, or so on. The choice of functions is made based on the operating region in which the protocol is intended to exist and the desired level of service that is to result from

its operation. Each function is divided into a mechanism and a policy.

Mechanisms are static and are not changed after a protocol is specified. The order of interpreting the fields of the (N)-PCI is determined by the PM (that is, defined by the protocol specification). In general, policy types occur in pairs: a sending policy and a receiving policy. For example, for the function detecting data corruption, a specific CRC polynomial is the policy. The sending policy computes the polynomial, and the mechanism inserts it into the PDU. The receiving policy computes the polynomial on an incoming PDU, and the mechanism compares the result with the field in the PCI. One might be tempted to call these inverses, but they really aren't; they are more complements of each other.

There are exceptions. For example, a policy to choose an initial sequence number would only occur in the sending PM. Hence, initialization policies or policies associated with timeouts may not occur in complementary pairs. The number of distinct types of policy associated with each mechanism depends on the mechanism but is generally only one. The number of policies of a specific type is theoretically unlimited, although in practice only a few are used. In general, there is typically a sending policy and a complementary receiving policy for the respective sending and receiving PMs. The coordination of the mechanisms in the sending and receiving PMs is accomplished by the exchange of specific fields of information in the (N)-PCI (see Figure 2-7). A single PDU may carry fields for multiple mechanisms in the (N)-PCI. A major consideration in the design of protocols is determining which fields are assigned to which PDU types.

For any one mechanism, a variety of policies may be applied to it. For example, consider the basic sliding-window flow-control mechanism used in many protocols. The sliding window is part of the protocol specification. When specified, this mechanism is not modified. However, there are a variety of policies for flow control: from simply extending new credit on receipt of a PDU, to periodically sending new credit, to high/low watermarks, and so on. Different policies might be used for different connections at the same time. Similarly, acknowledgment is a mechanism, but *when* to acknowledge is policy.

In the upper layers, OSI found that it was necessary to negotiate a "context." The presentation context selected the abstract and encoding rules of the application, whereas the application context was to "identify the shared conceptual schema between the applications." The concept of the presentation context was fairly well understood, but the application context never was.

Protocols should include a mechanism for specifying or negotiating policy for all mechanisms during allocation. (Many early protocols contained elaborate mechanisms for negotiating policy during connection establishment. However, it was soon learned that this was more effort than simply refusing the connection attempt with some indication of why and letting the initiator attempt to try again with a different request)

Policies chosen at the time that communication is initiated can be modified during data transfer. We will find below that changing the QoS on a flow can be quite straightforward, given the conditions for synchronization. Some policy changes may require that they be synchronized with the data stream to prevent pathological behavior. For example, changing the CRC polynomial for detecting corrupt data would require such coordination so that the receiver knew when to stop using the previous policy and begin to use the new one. It can be shown that this sort of strong synchronization is essentially equivalent to establishing a new flow. Hence, including this capability in the protocol would generally be deemed as simply adding unnecessary complexity. However, some

changes, such as changing the frequency of extending flow-control credit or the frequency of sending acknowledgments, would not require such synchronization and would not incur the same overhead. Although quite useful, it is less obvious that policy negotiation should be allowed during the data transfer phase. In general, changing policy during the data transfer phase requires synchronization that is essentially equivalent to establishing a new flow or connection.

Any ability to change policy on an existing connection or flow will have to be carefully handled to avoid aberrant behavior. The process, which determines which policies should be used or when they are changed, is outside the protocol. This may be requested by the layer above (or by the user), which knows its use of the protocol is changing. More likely, it will be effected by “layer management” to ensure that the parameters that agreed with the layer above are maintained or in response to changes observed in the characteristics of the layer below, and to ensure that the resource-allocation strategy of the layer is maintained.

By separating policy and mechanism, the operating range of a protocol is increased, and its ability to optimally serve a particular subset of an operating region is greatly enhanced. The choice of policy depends on the traffic characteristics of the (N–1)-flow and the *quality of service* (QoS) required by the user of the layer. The task of the (N)-layer is to translate these QoS characteristics as requested by the user of the (N)-layer into a particular choice of mechanisms and policies based on the service from the (N–1)-layer. As a rule of thumb, one would expect protocols nearer the media to have policies dominated by the characteristics of the media and consequently fewer policies would apply. For protocols further from the media, there would be a wider variety of policies that might apply. However, we will find that the real importance of the separation of mechanism and policy is the *invariance* or commonality it creates in a network architecture. This will prove to be very powerful and lead to a huge collapse in complexity.

Before we consider the mechanisms of data transfer protocols in detail, there are two other topics we need to touch on: One that sounds quite nebulous (Quality of Service) and one that sound quite mundane (the size of PDUs).

Quality of Service vs. Nature of Service

Quality of service (QoS) is a term that has been applied to the set of characteristics, such as data rate, delay, error rate, jitter, etc. that the user desires the communication to have. Proposals for QoS parameters (and sometimes rather extensive proposals) have been made many times over the past two or three decades, but few protocols have paid more than lip service to doing anything about it (to some extent with good reason). If you look carefully at these parameters and ask, “When this QoS parameter is changed, which policies of the protocol change and how?” you often find that the answer is “none.” What changes is outside the scope of a protocol specification.

There are two reasons for this. Any change in policy that could affect that parameter is a resource management issue, often a change in the buffering strategy: a topic generally not addressed by protocol specifications and normally considered the exclusive domain of the implementation. There is nothing that a protocol can do to affect the parameter.

Consider delay. Clearly, a protocol can minimize making delay worse, but it can do nothing to improve it. Parameters of this latter type are called *nature of service* (NoS). The distinction between QoS and NoS is essentially recognition of the old adage that “you can’t make a silk purse from a sow’s ear,” but perhaps we can make the sow’s ear a bit more acceptable. These are parameters largely determined by “nature.” We may be able to avoid making them worse, but there is little to nothing that can be done to make them better.

QoS represents a set of characteristics that the (N+1)-PM desires from the (N)-PM for a particular instance of communication (the silk purse). NoS represents the set of characteristics that an (N–1)-layer is actually providing and is likely to be able to provide in the future (the sow’s ear). The (N)-layer uses the difference between the QoS and NoS to select the protocol, mechanisms, or policies to match the desire with the reality. However, limits apply to what a particular protocol can do to improve a particular NoS to match the particular QoS that is requested. The nearer the (N)-layer operates to the physical media, the more constraining the NoS may be; that is, the technology dependencies limit the amount of improvement that can practically be accomplished by a single protocol. In some cases, some forms of error control may be more efficient or more effective if they are postponed to protocols operating further from the physical media. This and the fact that multiplexing at different layers allows for better strategies for aggregating PDUs are some of the reasons that there is more than one layer on top of the physical media. On the other hand, additional layers limit the achievable data-rate and delay characteristics, thus mitigating against too many layers. This is one of many trade-offs that are continually being balanced in the design of network architectures. We return to this topic later when we discuss particular QoS strategies.

Over the past 30 years, little progress has been made in providing QoS in networking systems. One of the reasons, frankly, is that no one has done the hard work of understanding how QoS parameters relate. If one looks at the proposed approaches and their proposed categories of QoS, they are all quite *qualitative* in terms of class of application and there is little agreement on what the definition of the parameters are that constitute the category. This isn’t helped by the fact that networking systems are not that sensitive. If we go back to our distinction between QoS and NoS, changes in the values of the parameters that affect QoS do not make fine grain changes in QoS. Part of this is because the QoS parameters are not orthogonal, or independent. The changes can be localized to a range, but seldom to precise values. Our goal should be to find such orthogonal parameters, but no one has been looking. To help the thinking along in this direction, we model QoS for a layer as a QoS-cube, an n-dimensional cube of the QoS parameters. We pretend that the parameters are orthogonal, and that the ranges that the changes in parameters make are the edges of the QoS-cube. If nothing else, this allows expressing a category of QoS quantitatively while we work to further our understanding of how to manipulate QoS effectively and either quantify the trade-offs or find parameters that are orthogonal or both.

Considerations for Choosing Policies

Policies will be chosen to optimize performance based on their position in the architecture. Protocols nearer the application will have policies suited to the requirements of the applications. Because many applications operate over these protocols, we can expect them to use a wider variety of policies. Protocols nearer to the media will be dominated by the characteristics of the media. Since these protocols have less scope and are specific to a

particular media, we can expect them to use a smaller range of policies. In other words, we can expect more QoS-cubes near the applications and fewer near the media. Protocols in between will be primarily concerned with resource-allocation issues.

The functions of multiplexing and relaying provide numerous opportunities for errors to occur. Analysis and experience have shown that any layer that relays can be subject to congestion and lost PDUs. Hence, an error and flow control protocol must operate over the relaying. This was the great X.25 versus transport protocol debate of the mid-1970s and early 1980s.

It is generally prudent (more efficient, less costly, and so on) to recover from any errors in a layer with less scope, an (N-1)-layer, rather than propagating the errors to an (N)-protocol with a wider scope. It is not only more costly to recover the errors in a wider scope, but there are multiple (N-1)-layers that could all be contributing errors to the wider scope of the (N)-layer. This does not mean it should never occur. This does not imply that the (N-1)-layers should be perfectly reliable. As we argued earlier, if the error-rate of the (N-1)-layer is well below the error rate expected by the (N)-layer, then it is fine. As was stressed before, every (N)-layer assumes and requires a minimal level of service from the layer(s) below. If one or more (N-1)-layers can't meet that requirement, then they must be enhanced so that they do meet the minimal requirement. This may require an error and flow control protocol operating over an error and flow control. This is most likely to occur in a legacy network, however, it might not be possible to do anything about the existence or absence of the first protocol, in which case the second protocol may be necessary to achieve the required QoS. This is probably the only instance in which one should find two adjoining error-control protocols. Another precaution one must be aware of is that if it is necessary to have an error and flow control protocol operating over another error and flow control protocol, one can have competing feedback loops that could adversely affect performance and stability.

The fundamental nature of relaying is to be always treading on the edge of congestion, and PDUs are lost when congestion cannot be avoided. Two adjoining relaying layers would tend to compound the errors, thereby decreasing the NoS of the second relaying protocol and thus impacting the QoS, and the performance, that an eventual single error-control protocol could achieve.

If the probability of the relaying introducing errors is low, an intervening error-control protocol may be omitted. For example, Ethernet LANs and related technologies are generally deemed sufficiently reliable that no "end-to-end" error control is necessary. For a wireless environment, however, the opposite could be the case. By the same token, the error-control protocol does not have to be perfect but only provide enough reliability to make end-to-end error control cost-effective. For example, the data link layer might tolerate an error rate somewhat less than the loss rate due to congestion at the layer above. However, these sorts of decisions should be based on measurements of the live network rather than on the hype of marketing or the conviction of the designers.

An application protocol can operate over any data transfer protocol that is within the scope of the next lower layer. However, an application protocol operating over a relay will be less reliable than one operating over an error-control protocol with the same scope (for example, mail).

Relaying and error-control protocols have been embedded into applications. The examples are legions of people arguing that the overhead of a transport protocol is just too great for their application. But then, they are inexorably drawn through successive revisions, fixing problems found with the actual operation of their

application until they have replicated all the functionality of a transport protocol but with none of its efficiency because their application was not designed for it from the beginning. We have already alluded to the relaying that is part of a mail protocol such as *Simple Mail Transfer Protocol* (SMTP). This application could easily be partitioned into a relay that did not differ much from those found below it and the actual mail application. The traditional checkpoint-recovery mechanism found in many file transfer protocols is an error-control protocol. Less obvious is the traditional *Online Transaction Processing* (OLTP) protocols, which involve a two-phase commit. OLTP can also be implemented as a form of data transfer protocol with different policies.

The Size of PDUs

Surface Area vs. Volume

The ratio of surface area to volume can be very important, especially as objects get smaller. Volume is decreasing by the cube, while surface area only by the square. This is why small objects cool much faster than large ones. As size decreases, the amount of surface area exposed relative to the volume means more of the volume is near the surface and heat dissipates faster.

What on earth does this have to do with PDU size?!

A similar thing happens with increasing bandwidth and shrinking PDU size. A now obsolete technology, Asynchronous Transfer Mode (ATM), had this problem. It had a very small PDU size (called a cell) of 53 bytes. Not only was the PCI 10% overhead, but the gaps between PDUs became a significant fraction of the bandwidth. At rates of up to 600 Mbps, even a small gap between cells was a significant fraction of the bandwidth, and it could be as much as 50%.

Determining the optimal size for PDUs is a normal engineering trade-off. In general, PDU processing overhead is proportional to PCI length, but independent of PDU length. Regardless, processing efficiency is maximized by making the PDU as long as possible. Similarly, the greater the amount of user data relative to the length of PCI, the greater the efficiency. However, other factors mitigate toward smaller PDUs, such as when the amount of data significant to the application is small, the buffering constraints in systems, fairness (that is, interleaving PDUs), the error characteristics of the media, and so on. Fragmentation (or segmenting) and concatenation may be used to match PDU sizes or improve efficiency between layers or between different subnets based on different media.

There is an optimal range for PDU size in each protocol, including applications, that will depend on where the protocol occurs in the architecture. For upper-layer protocols, this will be most strongly affected by the requirements of the application. Boundaries will tend to be created at points that have logical significance for the application. These sizes will give way to the requirements of the lower layers, while being moderated in the middle layers by system constraints (for example, the operating system and constraints on multiplexing). For lower-layer protocols, the size will be more determined by the characteristics of the subnetwork or the media. As noted previously, the PDU sizes for error-prone environments such as wireless will be smaller, thereby

decreasing the opportunity for errors. (The longer the PDU, the more likely bits were trashed.)

Don’t Get in a Rut

Wireless is a hostile environment! With the low data rates of wireless, small packets and good strong error detection codes are a must!

Why is wireless hostile? Lots of electrical noise, lightning, electric motors, etc. At low bandwidth, a 64-byte ARPANET PDU was 100s of miles long. It took a long time to move from IMP to IMP. Consequently, there are many chances for a string of bits to be trashed, burst errors. However, consider WiFi-10s of Mbps and increasing, and short distance. A 1500-byte PDU is 300 m, or less than 10μs. It is more likely that electrical noise will destroy an entire PDU than a string of bits in it.

One would expect larger PDU sizes in less error-prone media. For backbone networks where traffic density is the highest, one would expect media with very high data rates, very low error rates, and larger PDU sizes to take advantage of concatenation to increase efficiency. As data rate and traffic density increases, one wants to process fewer, bigger PDUs less often rather than more, smaller PDUs more often. Smaller PDUs will occur at lower data rates to minimize the time window for errors and increase the opportunity for interleaving other PDUs (that is, fairness). Smaller PDU sizes are more likely at the periphery, and size increases as one moves down in the layers and in toward the backbone. (Or as traffic density increases, for example toward the backbone, one wants to switch more stuff less often, not less stuff more often!) The ratio of PDU size to bandwidth and the ratio of PCI to PDU size should be relatively constant or decrease as one moves down in the architecture.

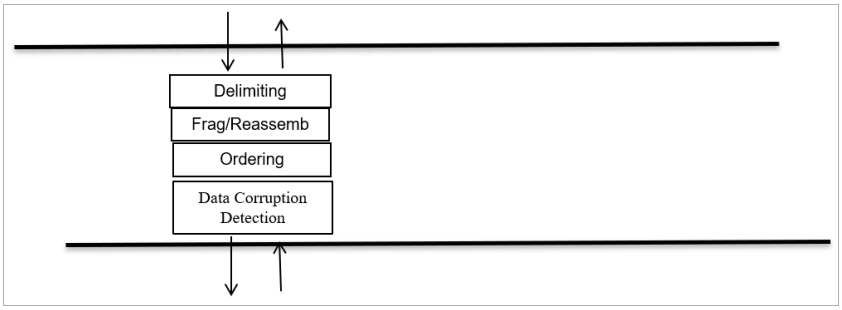
One of the factors in determining PDU size is to keep the ratio of the PCI size to the PDU size small. This ratio is an engineering choice, but, as a rule of thumb, below 5% is considered acceptable. Address fields are the greatest contributor to PCI size. Upper-layer protocols with wider scope will have longer addresses, and lower-layer protocols with less scope will have shorter addresses. Thus, we can expect some inefficiency in the upper layers as applications generate potentially shorter PDUs (but of a size useful to the application) with longer addresses, but we can expect increased efficiency at the lower layers as PDUs get longer, concatenation occurs, and addresses get shorter. Concatenation is not supported by the current Internet protocols. In the early years of networking, processors were much slower, and concatenation would have incurred too much delay. Later, networks were seeing congestion begin at about a 30% to 35% utilization. Consequently, the networks were operated at utilizations well below that. Concatenation would have required delaying PDUs to be concatenated, never good. With better congestion management and QoS support would allow higher utilizations lowering the time between PDUs so that concatenation would only have marginal impact on delay.

---

3. It pains me to have to do this, but the use of the * in the procedure or system calls is referred to as a *Kleene star* and means “zero or more” occurrences. There was a time in computer science when such explanations were unnecessary.

The Mechanisms of Data Transfer Protocols

Functions of a Protocol, Our Legos: I



Before we look in greater detail at each function and what kind of policies they might have, let's start by considering how the mechanisms of data transfer protocols relate to each other. These functions are our Legos, and there are a small number of them.

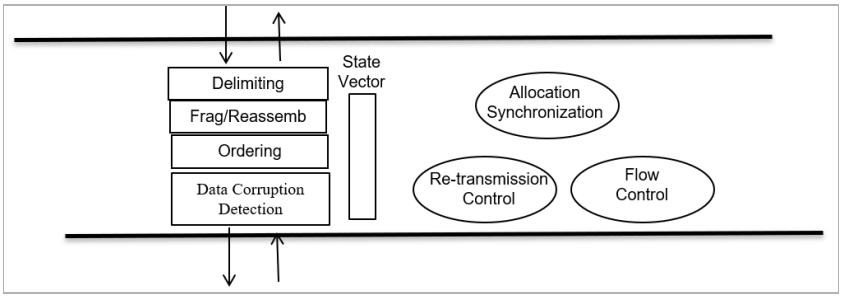
Data Corruption Detection/Correction, i.e., checking to see if the bits in the PDU have been damaged in transit. This is the first function that must be performed on any arriving PDU. Before anything can be done with a PDU, before we can process any of it, we have to know that the bits are good. The contents of a PDU cannot be trusted until we know the PDU has not been corrupted during transfer. Generally, the whole message is error checked.

Delimiting. Similarly, when data, an SDU, arrives from the layer above, the first step is to *delimit* it. We need to know where it starts and where it ends. We need to know if the SDU will fit in the user data field of one PDU. And we need to know if we are going to have to fragment it across multiple data fields in multiple PDUs. We may have to do *fragmentation/reassembly*. Or, if it is really small, we can pack more than one SDU in a single PDU, namely *concatenation/separation*.

Ordering. Of course, we want to make sure PDUs are delivered in order. In many kinds of networks, packets can arrive out of order. This will require sequence numbers to keep them in order. If this is a protocol over a point-to-point media, packets will arrive in order. But we still need to know that we got all of them, that none are missing.

Lost and Duplicate Detection will also use Sequence numbers which we'll talk about in a bit.

Functions of a Protocol, Our Legos: II



In addition, as we just described, there is *retransmission control* for recovering lost or damaged PDUs. *Flow control* to keep the sender from overrunning the receiver. For both of these we will use a sliding window or rate based, which we'll talk about later. These last two are feedback mechanisms that will require some kind of synchronization, i.e., make sure that the two stay coordinated. The conditions for synchronization will be created during the allocation phase. Deallocation will release the resources associated with this flow.

Those are the major functions that we're going to have to understand. Let's get started.

Error Detection and Correction

First, let us consider data corruption detection/correction. We will just be doing a brief overview of this topic. There are whole courses, whole careers, based on coding theory and information theory and the mathematics that goes with it, which is very esoteric. But we don't have time for that. We want to get a sense of the topic: the basics, how it is used, and what kinds of solutions apply when. We'll look at various things like hamming codes, binary convolution codes, etc.

During transmission, the contents of a PDU can be corrupted. There are two fundamental mechanisms for dealing with this problem:

- **The use of a checksum or CRC to detect the corruption.** The code is computed on the received PDU. If it fails, the PDU is discarded, and other mechanisms ensure its retransmission.
- **The use of forward error correcting code.** Forward error correcting code can detect and correct some number of errors, in which case the PDU may not have to be discarded.

The codes used must be chosen based on the nature of the error environment. For example, the traditional view has been that protocols closer to an electrical media (for instance, data link protocols such as HDLC or the various LAN protocols) are more subject to burst errors and thus require codes that can detect bursts of errors (for example, CRCs). However, optical media have different error characteristics and thus require a different kind of error code. And protocols more removed from the media (for instance, IP, TCP, X.25, or TP4) are more likely to encounter single-bit errors (memory faults) and therefore use error codes that detect single-bit errors. In addition, the error characteristics may interact adversely with other aspects of the protocol design, such as the delimiters. In general, a careful error analysis of both the protocol and the proposed operating environment must be done to determine the appropriate data-corruption detection strategy. In particular, the effect of PDU size on the strength of the polynomial must be considered. A particular polynomial will only achieve the advertised undetected bit error rate up to some maximum PDU length. Beyond that maximum, the undetected bit error rate goes up.

The Basics

Read

Read Tanenbaum Chapter 3, Sections 3.1.3–4, pp. 208–223.

The fundamentals of error coding are to detect the errors in the data. To do this, we take m bits of data and we introduce r check bits for a total of n bits. We make the r bits functions of the n bits. They introduce some redundancy to the data. Now, all 2^n combinations of bits are possible when the PDU is transmitted. (Any of them can be damaged, flipped from a 1 to 0 or from a 0 to a 1.) However, the functions used to generate the r bits is still true. If any of them don't produce the right answer, then the PDU has been modified in transit.

Two strings of bits are compared to determine their error distance. A minimum distance is defined as the number of bits that have to be changed to change one word into another. The number of bit flips it takes to change one codeword to another codeword is called the Hamming distance.

To detect D errors requires a Hamming distance of $D + 1$.

To correct D errors requires a Hamming distance of $2D + 1$.

To reiterate, the $M + r$ bits are a codeword. All 2^m combinations are possible, but not all 2^n combinations (because of how the r bits are calculated from the M bits). From this, the minimum Hamming distance can be calculated. Hamming distance is the number bits that differ, which can be found by XORing. To detect d errors require a Hamming distance of $d + 1$. Only $1/2^r$ messages are legal codewords. $(2^m/2^n = 2^{-r})$. It is this sparseness that allows us to detect and correct errors. To correct d errors need a distance of $2d + 1$.

Make powers of 2 check bits. The rest are the data. Use binary expansion to determine which bits contribute to parity of check bits: $11 = 1, 2, 8; 29 = 1, 4, 8, 16; 18 = 2, 16;$

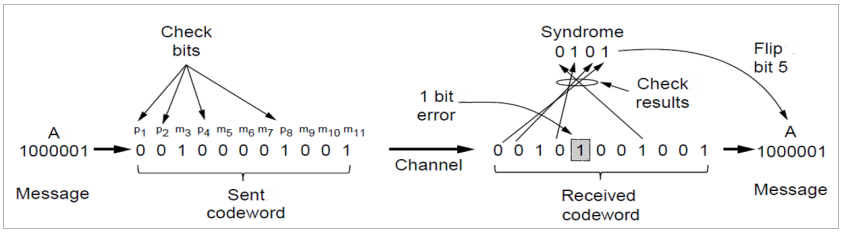
Why can a code with distance d+1 detect up to d errors? Because errors are detected by receiving invalid codewords (like 0101010101 for the example). If there are $d + 1$ or more errors, then one valid codeword may be turned into another valid codeword and there is no way to detect that an error has occurred.

Why can a code with distance $2d + 1$ detect up to d errors and correct 1 error? Because errors are corrected by mapping a received invalid codeword to the nearest valid codeword, i.e., the one that can be reached with the fewest bit flips. If there are more than d bit flips, then the received codeword may be closer to another valid codeword than the codeword that was sent. For example, sending 0000000000 with two flips might give 1100000000, which is closest to 0000000000, correcting the error. But with three flips, 1110000000 might be received, which is closest to 1111100000, which is still an error.

Read

Read Tanenbaum p. 214 for a full explanation of this example.

Error Detection Codes



Example of an (11, 7) Hamming code correcting a single-bit error.

Consider there are 11 total bits and 7 data bits. $2^r = 16$. Make the check bits the 2^n bit numbers, $1, 4, 8, 16$. The idea is for the check bits to be a parity check on certain data bits where data is 1000001. The check bits don't all sum to zero. So there is an error and their representation tells you which one. Here the check bits are the ones that are 2^x. Each data bit contributes to the check bits that make up its binary expansion. To see which ones, it contributes to rewrite the kth bit as its components: $11 = 1 + 2 + 8$ and $29 = 1 + 4 + 8 + 16$. contributes to check bits $1, 2, 8$. For even parity, if all bits are correct, the check bits will all sum to 0. If not, they won't and there is an error. The check bits that are not zero indicate which bit has changed. The check bits form an error syndrome. This locates the error. Here $0101 => 1 + 4 = 5$, so the 5th bit is in error.

Summarizing Error Detection and Correction

The important things to remember about data corruption detection/correction are:

1. The data corruption detection/correction policy is determined by the characteristics of the media or the lower layer. This will also determine whether detection and retransmission, detection but no retransmission, or error correction are preferred.
2. Different error codes are designed for different conditions. It is important to select the one suitable to the error characteristics of the media, or of the layer below.
3. This will be determined when the protocol or layer is defined. As outlined below, there may be different policies in use for different (N-1)-ports. SDU protection should be independent of the error and flow control protocol.
4. The fastest way to compute Cyclic Redundancy Codes (CRC) is by table lookup.
5. Beware of bit stuffing. We will learn about this next. Briefly, to keep the data transparent, bits must be inserted after the CRC is calculated.
6. The mathematics of error codes is very esoteric and very subtle. If you find yourself needing to develop a new protocol (or deciding what protocol to use in a new situation), this requires an error code. Usually one of the standard ones will be sufficient. But if the conditions are unusual, seek an expert for advice.
7. Regardless of what error policy is used, the same one will be required at the "other end" where the (N-1)-layer delivers the (N-1)-SDU. This implies that data corruption detection is really a function determined by the characteristics of each (N-1)-layer and not part of the (N)-protocol. It must be part of the (N)-layer to ensure that adequate protection is available. The (N)-layer should not trust the (N-1)-layer. There is potentially different SDU protection associated with each (N-1)-ports.

8. The error code will generally require 1, 2, or 4 bytes be added to the PCI. If the transfer rate is comparable to the processing rate of the error code, there may be an advantage to making the error code a trailer PCI. That would allow the error code to be calculated as the bytes of the PDU arrive. You would then apply the error code, and if the result is zero, the PDU is good. If not, the PDU is corrupted, and it is discarded. This may happen with media layers but is less likely with higher layers where the PDU has already been in memory for a "long time."
9. Beyond error detection and correction, SDU protection may also include any functions that must be done before the PDU is processed, such as compression and Time To Live.

These problems can be very real and go unnoticed for a very long time. In the late 20-teens, it was noticed that corrupted data was being delivered on TCP flows. These were very large gigabyte transfers between major physics labs at Gigabit rates. In 2021, this was mentioned in passing on an Internet history mailing list. Someone involved in finalizing the TCP specification in the late 70s noted that the TCP checksum was a placeholder until they could consult with an expert, and they never got around to doing it!

Delimiting and Packaging SDUs

There are two parts to delimiting:

First, is *bounding* the SDU. We have to know where the beginning and end is. There are two approaches to doing this that are *internal* and *external* to limiting. This is what Tanenbaum calls "framing," which is normally only applied to protocols operating directly over the media. Second, there is *packaging* the SDUs into the user data fields. There are two forms of packaging: *fragmentation/reassembly*, breaking SDUs across multiple PDUs, or *concatenation/separation*, combining many small SDUs into one PDU for better efficiency.

Read

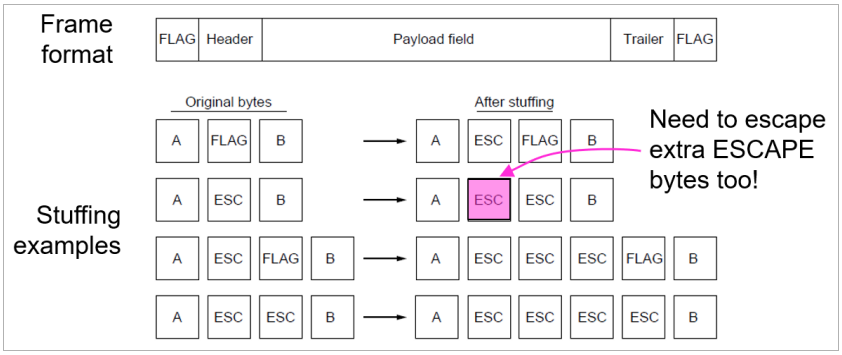
Read Tanenbaum Chapter 3, Section 3.1.2, Framing, pp. 205–208.

Delimiting

A *delimiter* is a mechanism used to indicate the beginning and end of a PDU. There are two basic methods for delimiting PDUs: external and internal delimiting. In external delimiting, a special bit pattern, usually called a *flag sequence,* is defined to denote the start and end of the PDU. The problem with this approach is that either data transparency is forfeited because the flag sequence cannot occur as a bit pattern in the PDU, or some "escape" mechanism is used to insert extra bits into the PDU to avoid the flag sequence, which are then removed by the receiver before any other PDU processing is done. Another common form of external delimiting is to use the lower layer to delimit the PDU. This may take the form of a length field in the (N–1)-PCI or in the physical layer in the bit encoding used (for instance, the use of Manchester encoding to delimit MAC frames in Ethernet). In internal delimiting, the PDU contains a length field as an element of PCI from which the number of bits or octets to the end of the PDU can be calculated. A degenerate form of internal delimiting is that the supporting service

provides only complete PDUs with a length field passed as a parameter as part of the interface. External delimiting is generally found in data link protocols, such as HDLC or the IEEE local-area network protocols. Network and transport protocols have generally used internal delimiting.

Bounding SDUs: External Delimiting



Special flag bytes delimit frames; occurrences of flags in the data must be stuffed (escaped).

- Longer, but easy to resynchronize after an error.

Internal delimiting – uses a count field that indicates the number of bytes in the PDU. The count indicates the number of bytes to the next count field. This tends to not be used for protocols operating directly over the media. If the count field is garbled, there is no way to find the next PDU in the byte stream. External delimiting is more common in data transfer protocols in higher layers, where there is much greater certainty the count is good, rather than directly over the media.

External delimiting. To prepare the reader for what is coming, let us first look at a very old protocol (that I hope is extinct) called *BiSync* that does external delimiting.

ASCII Table

Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

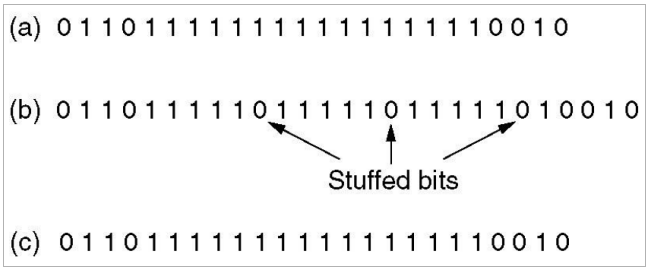
Have you have ever looked at an ASCII table and asked yourself what all this other stuff is for—all of these things that aren't characters or math or business symbols? Things like ESC, which you see on most keyboards, or STX, SOH, NUL, BEL, EOT, etc. All that stuff. They are for a character-oriented protocol called BiSync. They all stand for something, for example STX is Start of Text; SOH, Start of Header; and yes, BEL rang a bell, etc.

BiSync used a flag character to mark the beginning and end of a frame or PDU. Using a FLAG character, however, creates a problem. What if the FLAG character occurs in the data? One needs data- or bit-transparency; one needs to be able to send any bit combination. Otherwise, this approach is useless. BiSync solved this problem by what is known as byte-stuffing: anytime the flag character occurs in the data, insert an Escape character, followed by the FLAG character. That is what Escape is for. When the receiver sees the ESC, it deletes it and ignores the next character because it is not a FLAG.

What happens if a lone Escape character appears? Insert an Escape before it and the receiver will delete the first ESC and ignore the second. What about ESC FLAG? Just follow the rule and send: ESC ESC ESC FLAG, and so on. This Escape sequence will be used anytime one of the BiSync commands occurs as data. Clearly, while this is easy to understand, it is going to be somewhat inefficient, and few data transfer protocols are character oriented.

Byte-stuffing explains the concept. Now let's consider the more common method used in modern data-link protocols: bit-stuffing. This convention is used in HDLC (High-level Data Link Control).

Delimiting: External Delimiting



Bit stuffing

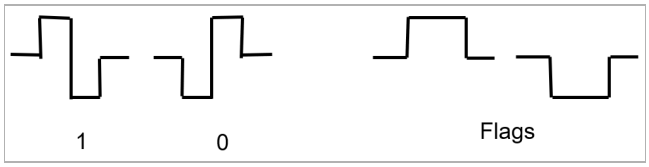
a. The original data.
b. The data as they appear on the line.
c. The data as they are stored in receiver's memory after unstuffing.

In this technique, the flag sequence is six consecutive 1 bits. The rule for sending transparent data is that when five 1s occur consecutively, a zero is inserted no matter what the next bit is. For the receiver, after the flag sequence, when five 1 bits are detected, if the next bit is a zero, it is deleted. (If it isn't a zero, then this is the Flag sequence ending the frame or PDU.)

In the last section, we noted that one had to be careful with the interaction between computing the error corruption detection code and delimiting. Here we see it. With bit-stuffing, the error code is calculated over the whole PDU, then the bit-stuffing is done. The zeros that were inserted to make the data transparent were not included in the error code calculation. If there are a lot of 1s in the data it can undermine the strength of the error code.

There is another way of doing external delimiting that's a little bit more like cheating. We will learn more about this later. Our goal is to be encyclopedic and cover all of the techniques for each function. This delimiting technique uses how the bits are encoded on the wire. One of the encoding schemes that is commonly used, especially in Ethernet, is called *Manchester Encoding*. Normally, you would think that the way bits are sent is that a high voltage is one, a low voltage is a zero. And that is used a lot. But there's another way where instead of being high and low, a transition from high to low is a one and a transition from low to high is a zero. Given the way signals deteriorate with distance and the way electronics are built, it's much easier to detect transitions than to detect levels. This is especially because, over distances, the signals are going to attenuate and be distorted. So, this is actually a fairly robust encoding scheme. Manchester encoded data always starts with a relatively long sequence of alternating 1s and 0s to train the electronics to the width of a bit.

Delimiting: External Delimiting



Bit encoding

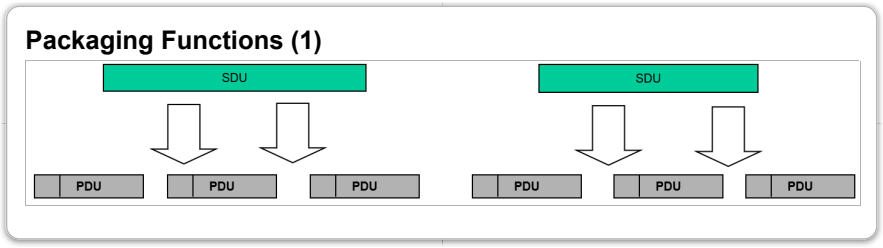
Manchester Encoding (LANs)

This convention can be used for flags. The flag sequence is defined as two highs or two lows, or a double-wide high or a double-wide low. Consequently, the flag sequence is outside of the ones and zeros. It is neither a 1 nor a 0. No escape mechanism, no bit-stuffing is required. It is naturally transparent. It's in a different “character” space. Two highs make no sense in Manchester and coding as a 1 or a 0. There is no conflict—it can’t be part of the data.

Packaging SDUs Into PDUs

The other aspect of delimiting is mapping the SDUs into the User-Data field of the PDUs, i.e., the packaging functions. In a layered model, the maximum PDU size is a property of the layer. In the beads-on-a-string or ITU model, there may be protocol translation at the boundaries between networks. In these networks, there has been fragmentation at these boundaries. However, as we will see, there can be problems with this.

Of course, the simplest approach to packaging SDUs into PDUs is to put one SDU in one PDU.



Fragmentation/reassembly is breaking up SDUs to put them into more than one PDU. This will occur when the maximum PDU size for a layer is smaller than the size of the SDUs submitted to the layer. This can happen fairly often. For example, in some wireless environments there will be small PDU sizes because wireless is a fairly hostile environment. The longer the PDU, the more likely it is to be trashed. There is a greater likelihood that some packets will get through without being damaged when sending lots of small packets than when sending a large packet. There will be fewer to retransmit.

Now, that said, if you calculate how long it takes to send, say, a 1500-byte PDU at the data rates today, it will be a relatively short period of time. Any electrical noise that could corrupt a PDU is generally going to be long enough to trash whole PDUs. It is much more likely that there will be lost PDUs than corrupted PDUs. But when wireless is much slower (a few thousand bits per second) with small PDU sizes, 100 bytes might be common. Corrupted PDUs would have been common. IoT could be a good example.

How is fragmentation done? There are a couple of ways that are typical. Keep in mind that we are fragmenting SDUs into PDUs. There will generally be fields in the PCI for fragmentation but not always:

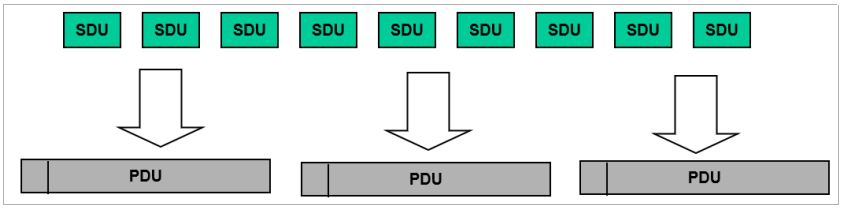
a. If the flow is stream, then the only requirement is to have a sequence number in the PCI to keep the data in order. There is no need to re-constitute the SDU. The application will have to determine any SDU

boundaries.

b. If the flow is idempotent, then the PCI will need a sequence number to keep the SDUs in order: an offset in bytes from the beginning of the SDU to indicate where this fragment belongs and usually either a More or Last bit, to indicate that this is or isn't the last fragment (a More bit, 0 when the last, otherwise 1 or the reverse if a Last Bit is used. Both are not required. The disadvantage of this is that if the PDUs are not fragmented, the offset field is wasted space.
c. Another technique where there can be fragmentation at the boundaries between networks has sequence numbers in units of bytes not PDUs. The advantage is that the offset field is not required—the sequence number, SEQNUM0, of a PDU is its number in the byte stream. When it is fragmented, the sequence number of the first fragment, SEQNUM1, is the SEQNUM0 + length of the data + 1, and so on. No offset is required. It would be elegant, if it weren't for the disadvantages. Counting bytes means the sequence number will roll over very quickly, which can limit throughput if there are no mechanisms to extend it. Because the boundaries of retransmissions won't be the same, the reassembly code is horrific. We will have more to say about this when we get to TCP.

If we have fragmentation/reassembly, then we should have the opposite, *concatenation/separation*, which is mapping many small SDUs into a single PDU.

Packaging Functions (2)



Combining/Separation – Little SDUs into a PDU.

To do concatenation/separation also requires adding PCI to the PDU. There are a couple of ways to do this:

- Have a sequence of <length field><data>, where each length field points to the next length field, and the PDU sequence number maintains the order of PDUs.
- At the beginning of the user data, have a "catalog" of lengths in bytes that points at the beginning of each SDU. This might be represented as

  <count 1><count 2> … <count n> <SDU 1><SDU 2> … <SDU n>,

  where <count i> points at <SDU i>.

The interesting thing to note here is that both aspects of delimiting are, strictly speaking, outside the protocol. Packaging is purely about mapping the SDU to the user data field.

You will not see this in IP networks. There is no example of this being done in IP networks, where one upper layer protocol PDU always goes in one lower layer PDU.

That is what is found frequently in the Internet today. To some extent, this is historical. In the 1970s, processors

were sufficiently slow that it took too long to get too fancy doing either fragmentation or concatenation. Today, the rationale for continuing this practice is that it would incur too much delay to try to combine SDUs into PDUs. (The reason for this is that most networks operate at around a 30% loading. Consequently, it can be a long time between PDUs.) They kept it really simple.

Today, neither of these arguments holds much weight. Processors are more than fast enough to do the concatenation without incurring much delay and the network can be operated at loads near congestion, so there are PDUs available to be concatenated.

The bright student will say, “but all those PDUs aren’t going to the same place! There is no point to concatenating them!” Perhaps not at the edge of the network, but as traffic moves through the subnets, there is common traffic between intermediate points in the network. As I like to say, there may not be constant traffic between Lake Forest, Illinois, and Lexington, MA, but there may well be nearly constant traffic between the Boston metro area and the Chicago metro area. (Remember these intermediate flows are not the same flows as the flows from source to destination. They are flows in lower intermediate layers.) On those parts of the network, concatenation makes sense. Those flows satisfy two principles of network design: as one goes down in the layers and in toward the backbone, *MTU should increase, and it should be switching more stuff less often, rather than less stuff more often*.

Summary of Delimiting and Packaging SDUs

The delimiting policies are:

1. Determining the Max Transmission Unit (MTU) for this layer.
2. At design time choose internal or external delimiting for a media layer, otherwise use a length field.
3. 3. Determine what typical SDU sizes will be. If the user of the layer is an (N+1)-layer, then it will be the (N+1)-MTU; otherwise it will be a range.
4. What are the criteria for fragmentation? Is it ever less than an MTU? What is the criteria for concatenation? What is the maximum number of SDUs that will be concatenated, if it isn’t “all that will fit”? Will it break SDUs across PDUs?
5. Again, there's no state to record here, so this is all self correcting.

Note that the encoding strategy supports idempotent SDUs. We can maintain the integrity of SDUs from source to destination within a layer.

Tightly Bound Data Transfer Mechanisms

Let us pause here and take stock of what we have covered. In some sense, the two mechanisms we have covered so far are not, strictly speaking, part of a data transfer protocol. As we noted, data corruption detection or correction policy is mainly determined by the characteristics of the specific (N-1)-layer and may even be affected by the characteristics (QoS) of an (N-1)-port at that layer. We have subsumed this under a function referred to as SDU Protection. SDU Protection consists of all those functions that are independent of the PDU

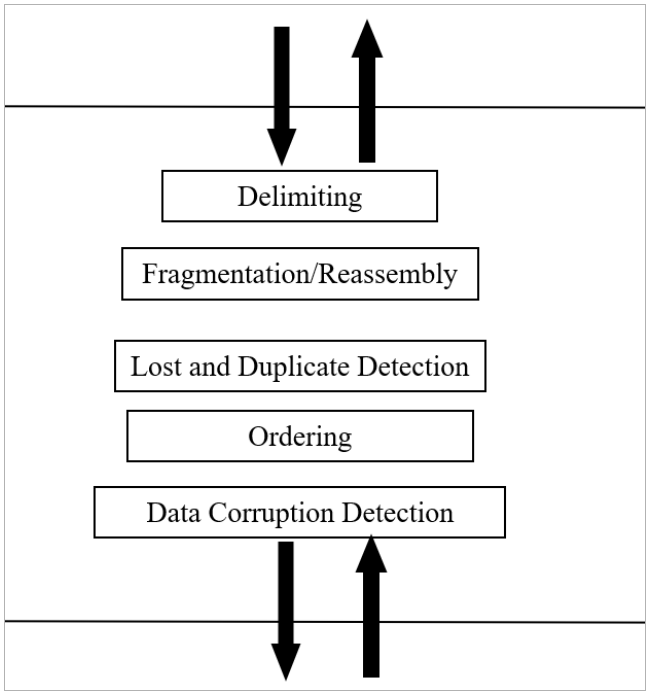
and usually have to be done before the PDU can be processed, such as data corruption detection, compression, time-to-live (hop count), etc.

Delimiting and packaging is not, strictly speaking, part of the data transfer protocol either, but it is about marking the bounds of the incoming SDU and packing it into the User-Data field.

As for a data transfer protocol, we will want to add a sequence number to enumerate each PDU. The sending protocol machine will record (in a state vector) the next sequence number to be sent, while the receiving protocol machine will generally record (in a state vector) the sequence number of the last PDU received in order. This will allow the order of PDUs to be maintained and to do lost and duplicate detection. More on that in a bit.

A critical design choice in a protocol is the size of the sequence number. Clearly, the field must have a finite size and avoid allowing PDUs with the same sequence number to be outstanding in the layer at the same time. Of course, it is likely that regardless of the size of the sequence number field, more than 2^n PDUs may be sent causing the sequence number to rollover. As we will soon see, there will be mechanisms performing modulus arithmetic on them. This will lead to situations where 6 is less than 1. We will have more to say about this shortly.

Data Transfer Protocol



The only state we need to keep track of is the sequence number: last sent, last received.

Note: Notice that all of these must be associated with the data.

What do we have so far? At the top of the layer, we have *delimiting*, the first thing done when an SDU is passed down from the layer above.

At the bottom of the layer, we have *data corruption detection*, which is the first thing done when an (N-1)-SDU is

delivered from the layer below. We have to make sure that (N-1)-SDUs are good before we look at the bits and determine what to do with them.

When (N)-SDUs arrive from the (N+1)-layer, delimiting is done to *fragment* (N)-SDUs or concatenate them, if necessary. In the case that a PDU is delivered from the (N-1)-layer, then we need to use the information in the (N)-PCI to reassemble PDUs that arrived before or others arriving into whole (N)-SDUs that can be delivered to the (N+1)-layer before, and then we need to put the SDUs *in order*. And we may need to do *lost and duplicate detection*.

CRC	Type	Ver	Length	Flags	Sequence Num	Offset	User Data

What will the PDU look like so far? Given those functions (which can be separated into a mechanism and multiple alternative policies), this is what is the PDU will look like. The *Error Check* (CRC) is in a fixed position from the beginning or end of the PDU, so it can be found and computed first. One always has to have a *Version* field and a *Length* field. A field for *Flags* like the More flag, a *Sequence Number*, and *Offset*. And of course, the *user data*.

The only state we really need to keep so far is to keep track of the sequence numbers that were *the last one sent* and the last one received *in order*. As we will see, this allows Lost and Duplicate Detection. Keep in mind that we are doing modulo arithmetic on these sequence numbers, so the sequence numbers will wrap back to zero and start over again. (Later we will look at making sure two different PDUs with the same sequence number are not in the network at the same time. Generally, we will do this by enforcing a Maximum Packet Lifetime and only allowing half the sequence numbers to be in the network at once. But there are other ways to do it.) Notice, that none of these mechanisms are computationally complex. They will be very simple to code, and the code will be quite fast.

The title of this section is *Tightly Bound Data Transfer Mechanisms*. The PCI associated with these mechanisms must be with PDUs that carry the data. Next, we will discuss the *Loosely Bound Data Transfer Mechanisms*, which may be with the data but don't have to be.

Loosely Bound Data Transfer Mechanisms

Now we come to the core of all data transfer protocols: the feedback functions of retransmission and flow control. This is where things can get complicated and very subtle. The asynchrony inherent in the operation of a protocol makes the problems that can occur subtle and, for most students, outside their experience.

We will look at two feedback mechanism: flow control and retransmission control (or Acknowledgements, Acks). The purpose of flow control is to keep the sender from overrunning the receiver—sending data faster than the receiver can handle it. There are three basic approaches: the Fixed-Width Window, the Dynamic-Width Window, and Rate-based. We will cover the simple Stop-and-Wait as the degenerate case of a fixed-width window of 1. Retransmission control determines when PDUs can be deleted from the retransmission queue, i.e., when the

sender receives an Ack indicating the receiver will not request retransmission ofxs PDUs with this sequence number or less. As we saw, this does not necessarily mean the receiver received the PDU. Then there are the optimizations for this function of Selective Ack and Selective Nack (Negative Acknowledgement). As we will see, the sliding window links these two feedback mechanisms and does it in subtle ways.

This is because these feedback mechanisms both require some degree of synchronization, which we will consider in the next section.

Read

Read Tanenbaum, Chapter 3, 3.3, pp. 224–234.

In the reading, Tanenbaum embarks on developing a sequence of data link protocols ideally introduce the concepts necessary for the protocol to work and replicating the sequence in which the protocols were developed historically in the data comm world. Consequently, some of the aspects he emphasizes were dependent on the conditions at the time. Also, these protocols come from the beads-on-a-string tradition in the commercial world of data comm of the 1970s that hewed closely to deterministic solutions and special cases rather than a simpler design that folded requirements into the model as degenerate cases.

Meta-note: While we often say that good design will alternate between top-down and bottom-up, with top-down, one is proceeding from the more abstract to the less abstract. At each step, one can observe the remaining functions and determine what level abstraction they belong to, which to include, and which to postpone. If one makes mistake, it is relatively easy (and inexpensive) to back up and explore a different path. With bottom-up, one starts with the less abstract and often the concrete. Building up the design from the bottom is making a commitment to a design at an early stage. A mistake here can be much more costly to recover from and tends to lead to generating special cases. In other words, inserting kludges in an attempt to avoid throwing anything away and being forced to go back to the beginning and starting over. Once special cases are introduced, they tend to proliferate (because it is the wrong model). The bigger the project, the more important this is and the more costly it can become.

Fixed-Width Sliding Window Flow Control

Read

Read Tanenbaum, Chapter 3, Section Bidirectional Transmission: Piggybacking, pp. 234–236

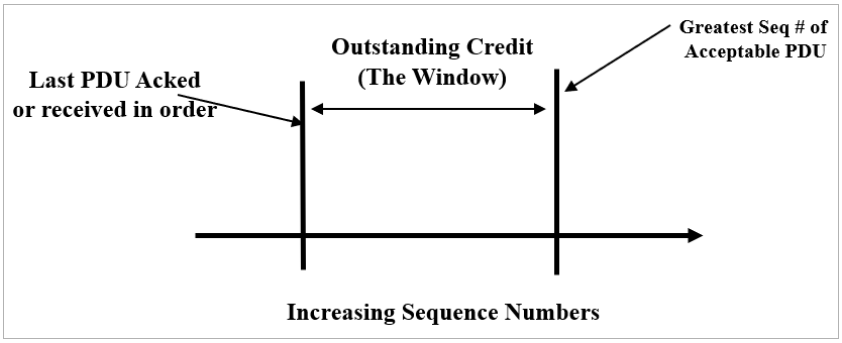
As you will notice, Tanenbaum has emphasized that control and data are sharing the same link. This is an echo of what was pointed out in the first lecture about whether packet switching was a major change in thinking. For those with a telephony background, this is significant, because in telephony they didn't (and, to some extent, they still don't) implement the control "signals" on a different physical link. Consequently, in-band signaling was a

significant departure. For those with a computing background, this is almost insignificant. It seems obvious that control and data would be on the same media, while processing them would be distinct. More recently, there has been experimentation with assigning control PDUs to a different QoS class, but still on the same media.

Tanenbaum describes all three of the protocols here: the one-bit sliding window, go-back-N, and Selective-Reject are *fixed-window flow control* protocols.

What window?

The Sliding Window



Imagine a line with points labeled for each sequence number assigned to a PDU. Now let's assume that there is a sliding window on this line. The left window edge (LWE) is the sequence number of the last PDU Ack'ed. (This means that all PDUs up to this number will not need to be retransmitted.) These protocols have a fixed window width, n. The right window edge (RWE) is $LWE + n$. The RWE is the sequence number of the last PDU we can send without getting an Ack. (The receiver believes it has enough buffer space for n PDUs.) Every time the sender gets an Ack for PDU x, the window moves LWE by x, where $x \leq n$, which also moves RWE, so that RWE is still $LWE + n$.

This means that the width of the window is tightly linked to the acknowledgement. Sending an Ack to the sender not only acknowledges all PDUs up to that sequence number but also advances the window by the same amount indicating that the receiver has the buffer space to receive more PDUs.

It is interesting that Tanenbaum is silent on an important case that is the "fly in the ointment." Consider what will happen if the receiver receives a PDU correctly, in order, but the receiver's user has not read any data to deliver an SDU and there is no buffer space at the receiver:

- The receiver should send an Ack, but if it does it will open the window and the sender will send more PDUs.
- However, the receiver has no buffers for more PDUs.
- On the other hand, if the receiver does not send an Ack, the sender will assume the PDUs were discarded, timeout and retransmit all un-Acked PDUs.

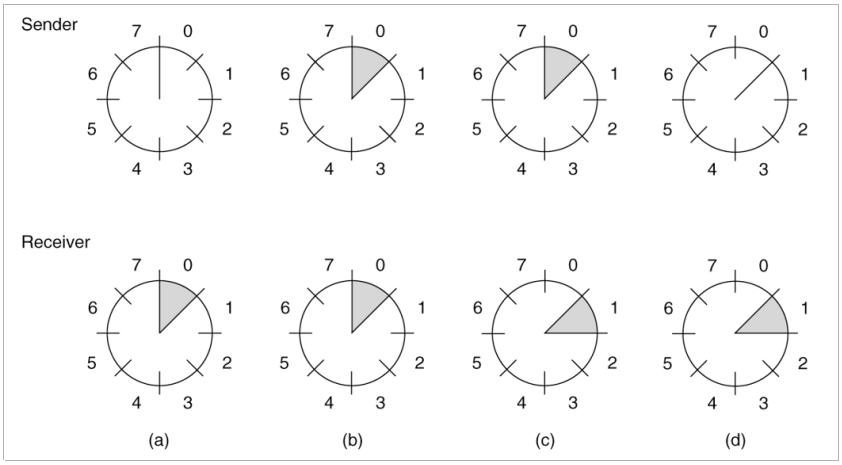
This is not a good outcome either way. This class of protocols solved this problem by adding two additional PDU types. (Notice we are creating a special case. This is basically a patch.) The receiver sends a Receiver Not

Ready (RNR) to Ack that the PDU was received. The sender can stop the timer and delete the PDU(s) from the retransmission queue, but not advance the window. Then, when the receiver's user reads data and frees up buffer space, the receiver can send Receiver Ready (RR). After covering Tanenbaum's description of this, we will look at a design from a different model that avoids the special case.

Example of a Fixed-Window Flow Control

The sender maintains a set of sequence numbers corresponding to the frames it is permitted to send. The frames are said to fall within the sending window. The receiver maintains a receiving window corresponding to the frames it is permissible to accept. (See the discussion above). And they differ among themselves in terms of the efficiency, complexity, and buffer requirements. For these data link protocols with small windows, the window is often represented as a wheel to illustrate the rollover of the sequence number. The sender and receiver each have a window to represent the state of the transfer. (Actually, there will be two of these, one pair for each direction.)

Data Communication View with a Fixed Window



Source: Tanenbaum, A., Wetherall, D., & Feamster, N. (2020). *Computer Networks* (6th ed.).

The figure illustrates the sliding window with 3-bit sequence numbers, e.g., 0–7, and a window only allows one PDU to be outstanding at a time. Following the events as they occur:

- a) This is the initial situation. The Sender has not sent anything, and the shaded portion indicates that the Receiver is willing to accept one PDU.
- b) The Sender has sent a PDU, but it has not yet arrived at the Receiver. The shading at the Sender indicates that it has one PDU outstanding and cannot send more until it gets an Ack.

Tanenbaum skips a step: between b) and c), the state should be shown where the PDU has arrived at the Receiver, but the Ack has not yet been sent, and the Sender is still blocked from sending another PDU.

- c) indicates that the Receiver has received the PDU, sent an Ack, and advanced its window to indicate it is willing to receive another PDU, but the Ack has not yet reached the Sender.

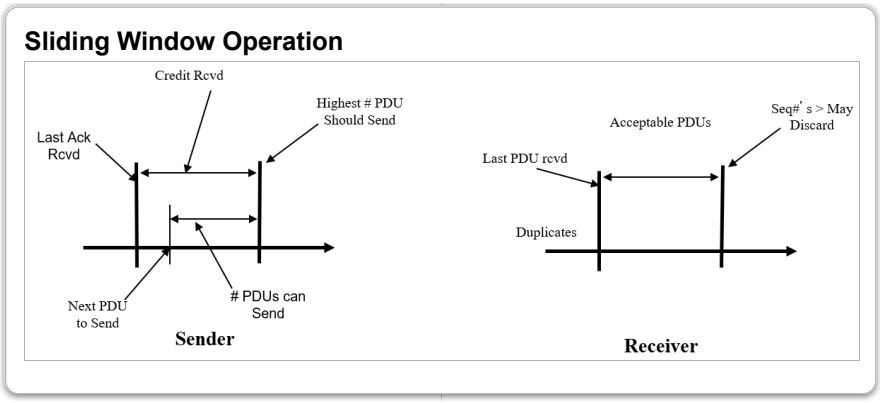
- d) indicates that the Ack has reached the Sender and the Sender could now send another PDU.

Tanenbaum has the code for the three protocols mentioned here in Chapter 3 of the book. The slides for the three protocols in this sixth edition and from the fifth edition are posted on Blackboard. Please take a look and tell me which one you think is the easiest to understand and what it would take to make them more understandable. Notice that he does not cover the case discussed above that requires RNR and RR.

Dynamic Window Flow Control

There is another way of looking at this that is simpler, more flexible, and more general. As near as we have been able to determine, this solution was developed before Go-Back-N and represents a different mode of thinking. In fact, it appears it was rethought entirely.

The generalization of the sliding window looks like the figure above (Figure: The Sliding Window) Again, we assume that the sequence number space is increasing from left to right and sequence numbers are being assigned to each PDU as it is sent. The Sender's Left Window Edge (LWE) is the last PDU Acked by the Receiver or the Receiver's last PDU received in order. The number of PDUs that can be sent is the width of the window. The Sender cannot send anything beyond the Right Window Edge (RWE).



Now let us look at how this works: Both sides keep a window.

On the sending side, the LWE is the last packet acknowledged. That means that all PDUs earlier than this have been acknowledged. Any PDU with a sequence number less than this number can be deleted from the Retransmission Queue. The next line is the sequence number of the next PDU to be sent. Dynamic window differs from the fixed window in that a new field has been added to the PCI called the Credit field. The width of the window is how many PDUs can be sent without receiving more credit. The RWE is the largest sequence number that can be sent until there is more credit, where RWE = LWE + Credit. (It is worth taking a moment to understand why. The Receiver is using flow control to keep the Sender from sending more data than the Receiver can handle. Thus, Credit is linked to the amount of buffer space the Receiver has. The Receiver knows the Sender can't send beyond the RWE, but does not know how much has been sent. Therefore, when it sends an Ack, it takes into account its previous Credit beyond that Ack number, the amount of buffer space available, and how much of the buffer space can be expected to be there; computes a new Credit value; and includes it in

the Ack PDU. Notice that the Receiver can shrink the window in this scheme.)

> Generally, the way dynamic window works is the Receiver will send a PDU with an acknowledgement and a credit in terms of number of PDUs. The Credit is added to the LWE to get the RWE. The last sequence number sent better be somewhere in between. The Sender can send PDUs from the Next Sequence Number to Send until the RWE.
>
> As the Sender sends PDUs, it puts a copy in the retransmission queue and sets a timer to a value that is the estimated Round-Trip Time (RTT), plus some extra to allow for the processing and/or waiting for gaps (caused by out of order PDUs) to fill at the Receiver. If the Sender does not receive an Ack for the PDU, then everything from the sequence number of the Ack to the Last PDU Sent is retransmitted.
>
> Now, at the receiving side, the LWE is the last PDU received in order and acknowledged. This means that anytime a PDU with a sequence number less than the LWE is received, it is a duplicate and can be discarded.
>
> Anything received within the window can be kept. There is some discretion here depending on buffer space and the policies that are in place. If buffers are tight, we may want to only accept PDUs that are in order (and force them to be re-transmitted). Or, if there is plenty of buffer space, keep out-of-order PDUs (potentially avoiding retransmissions), and wait for the late PDUs to fill in the gaps and then Ack all of them at once.

This asynchronous and concurrent sending, receiving, and ack'ing is going on all the time. PDUs and Acks can be lost or arrive out of order. The LWE can only move when a contiguous sequence of PDUs have arrived. (As we have seen, there is a caveat to this: If there is a gap and the retransmission policy tolerates gaps, an Ack may be sent to delete PDUs from the retransmission queue and move the LWE.) The Sender has a timer set for the retransmission queue. The time-out is the time for a round trip to the Receiver plus a little more to allow for the variations in the delay generating the Ack and the delay it encounters in transit. When the retransmission timer expires, everything from the LWE to the last PDU sent is retransmitted. Clearly, determining the retransmission timer to allow for all of this is critical.

Notice that the width of the window is no longer tightly tied to the Ack and no longer monotonically increasing. The window can expand and contract in response to the state of the buffers available. The RR and RNR commands are not necessary. If the window is closed, and the Sender can't send any more PDUs, there is no problem. The Receiver may still need to send Acks to empty the retransmission queue, but it can keep the RWE in the same place.

However, there is another problem. Suppose the window has been closed, and the buffer constraints at the Receiver get better, so the Receiver sends an Ack PDU with credit that opens the window, but it is lost. There is a potential deadlock. The Sender is waiting for an Ack PDU to open the window, while the receiver has sent an Ack PDU that opens the window and is wondering why no data is being received. Both sides should take action: 1) every so often, the Sender should send a short PDU to the Receiver to see if the window is open; and 2) the Receiver should retransmit the last Ack and Credit. Note this is also the correct response to a probe from the Sender when the Window is still closed.

There is one more thing we need to discuss that is a problem for both the fixed and variable windows. The sequence number space is going to roll over. The sequence number is of finite size and all arithmetic on it is modulus the size of the sequence number field. There's going to be a place where things get a little dicey where the window is straddling the rollover point. There will be a situation with 3-bit sequence numbers where the RWE is 3 and the LWE is 7. In other words, 3 is greater than 7!

With this example, one might be wondering, if the window is too large, isn't there a danger of wrapping around and having two PDUs in the network with the same sequence number at the same time?

The answer is *yes*. There are also other reasons that a PDU could be wandering around the network for a long time, which we will get to later. It turns out that preventing this is fundamental to networking. Precautions need to be taken to prevent that. Traditionally the solution has been to make the Credit field in a protocol half the width of the sequence number field. For example, if there are 8-bit sequence numbers, the Credit field will be 4 bits. That implies that there are 255 PDUs that can be sent before there is wraparound, but the sender must get an Ack before 16 PDUs have been sent. (The Ack acks all sequence numbers less than it, so one knows all of the PDUs are no longer in the Internet, unless they are retransmissions. When they arrive (if they do), they will be discarded as duplicates.) But this solution can also have its problems.

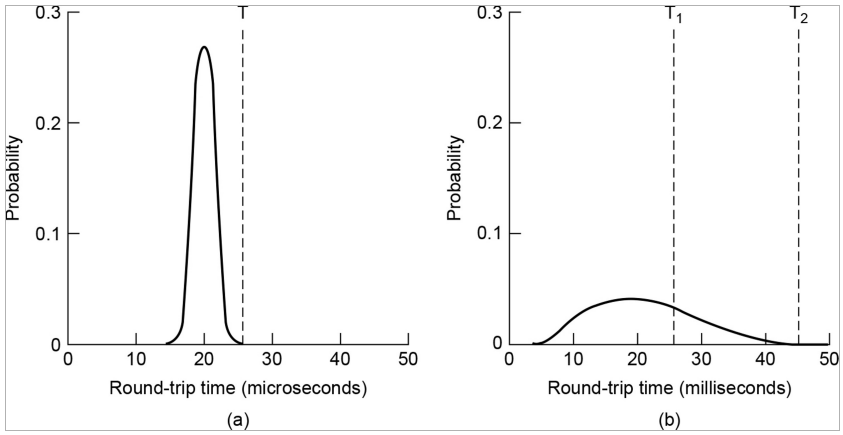
Notice in our example that 16 is much less than 255. The sender could send 16 PDUs fairly quickly and have to wait for an Ack, while there is little chance that it could get as far as 75 or 100 without getting an Ack. This could be a major limit on performance depending on the rate at which PDUs can be sent and the roundtrip time; a wider window might be worthwhile.

Later we will find there are more explicit mechanisms that ensure the maximum lifetime of a PDU can be enforced. It turns out that a smaller credit field can also limit the performance of the protocol. The solution to this is beyond the scope of this course.

Retransmission Control

For most protocols, when a PDU has been corrupted or lost, it is recovered by having the PDU retransmitted. Clearly, we want to minimize retransmissions. They waste network capacity. The receiver must let the sender know what it has received in order and no longer will need to have it retransmitted. The receiver does this by sending an Acknowledgement (Ack) that is the sequence number of the last PDU received in order. If the receiver does not get a PDU for a while or if it gets PDUs out of order, it can't send a new Ack. When the sender sets a timer for PDUs it sends, if the timer expires and there has been no Ack for that sequence number, the sender retransmits all PDUs from that sequence number to the last sequence number sent! (When PDUs are Acked, the timers are canceled, so unnecessary timer events don't have to be handled.)

TCP Timer Management



The question is how long should the retransmission timer be? We estimate the roundtrip time (RTT) by measuring how long it takes to get an Ack plus a little more to account for variation in sending the Acks and the delay the Ack incurs enroute to the sender. Tanenbaum quite correctly makes a point that the variation in RRT for point-to-point protocols is very small. Most of the time is propagation delay, which is constant. There is only a small amount of variation to account for the system at the other end being busy when it needs to send an Ack. For protocols operating over relays, the amount of variability will be much greater. The propagation delay for each hop between relays is constant, but the queueing delay at each relay can be quite variable, plus there is variability in the end system. We will look at this case in greater detail later.

For flow control, our primary concern is to keep the pipe full. To do this we will need an estimate of the "bandwidth-delay" product, then determine the buffering strategy and our policy for moving the right window edge.

Piggybacking Acks

Read

Read Tanenbaum, Chapter 3, Section 3.4.1, pp. 234–236.

As Tanenbaum describes, piggybacking Acks means sending the acknowledgement with the data in the return traffic supposedly for greater efficiency. For example, request/response traffic would be a good place to take advantage of piggybacking. This sounds reasonable and students will hear this given as a great advantage. The purpose of this section is to put this misconception to rest. The use cases where piggybacking Acks is advantageous are very limited to the point of not being worth the effort.

There are basically two ways to do this: either by concatenating the Transfer PDU and the Ack Control PDU, or by expanding the Transfer PDU PCI to carry the Ack number. (This may require a bit in the PCI to indicate when the Ack sequence field is valid and will still result in Acks being sent with no data. The best way to avoid this complication is to define the protocol so that the receiver always sends what it believes the left window edge to be. It is no more overhead and the field is going to be there anyway, it can be used to confirm whether the sender

is in the correct state.

Obviously, the reason for piggybacking is to be more efficient. Is it? It *can* be for a small set of applications. Taking the two ways of doing piggybacking described above, let's look at it.

- Let us say that the normalTransfer PDU PCI consists of two port-ids, a version, type, control bits, CRC, sequence number of NT bytes, and user data, m. A normal Transfer PDU is $NT + M$ bytes. A Control Ack PDU consists of the same things as NT plus an Ack field of n bytes.
- With the concatenation approach (CT), the total length is $2NT + m + n$. The overhead due to PCI alone is $2NT + n$.
- If we use the integrated approach (IT) putting the Ack field in the original PCI, the overhead for the PDU is $NT + n + m$. The overhead of the PCI is merely $NT + n$.

Of course, the traffic has to be more or less symmetrical (sending one Transfer PDU generates one Transfer PDU from the receiver); otherwise, there is no point to piggybacking Acks. The most likely symmetrical traffic will generate a request/response application. However, even there, it should be "balanced." There should be roughly the same amount of traffic (PDUs) going in both directions. If the application is "unbalanced" (lopsided), then there will need to be more Acks going in one direction than the other and nothing to piggyback on.

As we see from the estimates above, there is savings as long as the amount of data is less than the difference $CT – IT$, or $m \leq (2NT + n) – (NT + n) = NT$. It is only more efficient if the amount of data being exchanged is less than the length of the PCI. That isn't much!

Okay, it doesn't have to be strictly less than or equal to be worthwhile, but in that vicinity. That isn't going to be much! Essentially, one header is probably on the order of 20 bytes.

More than that won't be fewer bytes sent, but it will still be more efficient, right? Yes, but remember that as the amount of data, m, increases, the PCI becomes a smaller and smaller fraction of the PDU, and the savings is less and less significant. Remember you are only saving one PCI per PDU. Let's try some numbers:

- Let NT = 20 bytes, n = 4, and M = 50, then the two methods would generate CT = 44+ 50 = 94, IT = 24 + 50 = 64 or about 30%. That is significant.
- Now what about 100 bytes?
  CT =144, IT = 124 or 16%.
- At 200 bytes, it is 9%.

A few hundred bytes is not much for today's applications and PDUs are more often much longer. The advantage quickly becomes insignificant.

Coming Clean

As we will see later, the example here is using numbers from TCP, where piggybacking is often pointed to as a major advantage of TCP. Histograms of TCP traffic indicate a mode at 20 bytes, 576 bytes, and 1500 bytes Ethernet PDUs. The maximum size PDU that every device has to be able to handle is 576, and 20 bytes are Acks and SYNs (to

initiate a connection) with no data. There are many more Acks than SYNs. It appears that piggybacking is not significant in the Internet.

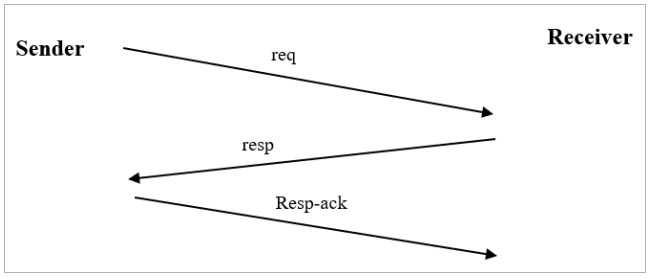
There is another complication. Piggybacking implies withholding sending an Ack until the application has responded. The application must be relatively tightly coupled to the protocol. However, the protocol is executing in the OS and getting higher priority than just an ordinary application in user space. Delaying sending the Ack means longer round-trip times, longer retransmission time outs, and longer retransmission queues at the sender, as well as potentially poorer response time.

And with asymmetric traffic streaming video and audio, piggybacking is not useful at all, or even for most web traffic where the request may be one PDU, the response is going to be many PDUs. Piggybacking becomes useful only for a small subset of a small subset of applications, such as character-at-a-time echoing of terminal traffic for Tenex systems on the ARPANET. Or to be blunt about it, Tanenbaum is wrong— piggybacking seldom improves efficiency.

Synchronization

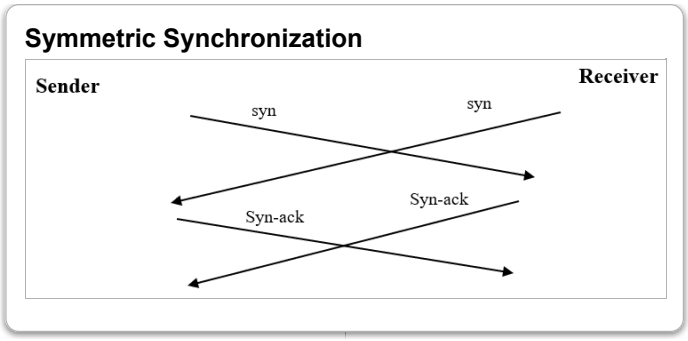
These are feedback mechanisms. They are going to require synchronization. We have to figure out how to do that. When we did the IPC Model in a single computer, we could use shared memory techniques to do synchronization, but we don't have that. We will need to have loose synchronization, to coordinate the communicating IPC Processes, as well as initializing some parameters, such as initial sequence number, size of a maximum transmission unit, etc.

The Three-Way Handshake



The three-way handshake is used when there is feedback and the potential for lost messages.

In the book you will read about two-way and three-way handshakes as the means to establish synchronization. The idea is that for a two-way handshake, one sends a request, followed by a response, but this has failure modes that can leave the state of one end unknown. Every textbook will emphasize the three-way handshake consisting of a request, a response, and a response-ack to synchronize and initialize the protocol state machines.



What happens if they cross in the mail? That depends. There are two different views of this. Some said that it created one connection; others said it created two separate connections. In the early days this created quite a bit of discussion because they were not being precise about the model of the connection. The difference depends on the point of view, whether one is looking at the problem from the point of view of the protocol state machines or as a user of the layer.

From the point of view of the protocol state machines, if both sides choose the same port-ids and send the initial request, they will be seen as requesting the same connection and each request will be seen as the response to the other. If a request is made by the user of the layer, the sender will assign a local port-id and send a PDU connection request. The sender doesn't know the port-id that will be assigned by the receiver. The receiver will receive it, notify the destination user, and send a response with its own port-id. (Note that the port-ids are only locally unambiguous.) The sender and receiver port-ids are concatenated to form the connection-id.

Clearly, if two users make requests at the same time at the layer boundary, these are two unrelated events. Each protocol state machine will do what was just described, each choosing local port-ids that are not in use, and two connections will result. The only way that two requests can cross in the mail and create one connection is if the both port-ids are known to both parties before the requests are sent, which is never the case.

Now there is just one other thing:

The three-way handshake is a myth!

While it fits our intuition, it is totally irrelevant.

It has nothing to do with establishing synchronization.

It has as much to do with creating synchronization as saying, "Abracadabra ala kazam! You're synchronized!!"

And it's actually more complexity than the problem requires.

This is a subtle point. We are not saying the three-way handshake is wrong. We are saying the three-way handshake has nothing to do with why synchronization is established. However, every textbook in networking will tell you that it is how it is done.

Watson's Theorem

In the late 1970s, Richard Watson made a remarkable discovery. He proved that

The *necessary and sufficient conditions* for synchronization

for reliable data transfer is to enforce an upper bound on three times:
Maximum packet lifetime, MPL
Maximum time to wait before sending an Ack, A
Maximum time to exhaust retransmissions, R

(I have always liked the way he describes it.) He assumes:

All connections exist, all the time.

The state vector for the protocol state machine is merely a cache of information on connections that there has been activity "recently", where "recent" is defined as

no traffic for $2 * (MPL + A + R)$

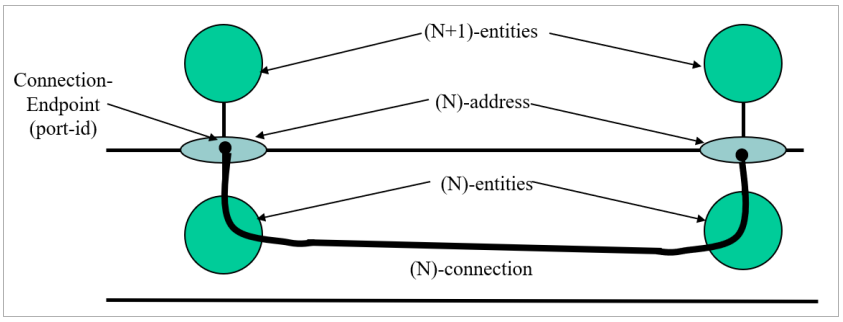
Then the state vector can be discarded because it's irrelevant. After this length of time, there are no PDUs in the network that are relevant to the state of the connection.

If there is more traffic to send, we merely start sending again and the state will be restored.

This is an incredible result. It is very counter-intuitive. What it means is the bounds on the times are the most important property. It works, because the times are bounded. We will talk more about this when we get to TCP. This result is one of the most important and least-known principles in networking, as important as the invention of packet switching and the invention of datagrams.

To further support the result, Watson and his team at Lawrence Livermore Lab designed and implemented a protocol that embodied this result, called delta-t. It exposed a flaw in the Internet and the OSI Reference Model where a connection started at the sender's port goes to the destination port.

Delta-t Exposes a Flaw



The Internet exposes the addresses to the application. This is what connect and disconnect do. This combines port allocation and synchronization.

What Watson is saying when he assumes all connections exist all the time is that port allocation and synchronization are distinct.

Ports are allocated. And when there's data to send, then the connection exists.

Delta-t Says Separate the Two
met_cs535_mod2_lec2_slide47_animation video cannot be displayed at the printable page. Go to the module page to watch.

The state is created, and the Connection-EndPoints (CEP) are bound to the port-ids. In a good design, the Connection-EndPoint-ids (CEP-ids) and the port-ids are distinct.

Port-ids are never carried in protocol, only CEP-ids.

That means that if there is no traffic on this connection for some period, the state associated with the connection can disappear because we know there are no PDUs in the network that relate to that state information. This is because we have enforced an upper bound on those three times.

The connection state is gone, but the port-ids still exist. In fact, the connection can still use the connection-endpoint-ids. If there is more data to send, just start sending.

This is a much more secure design than designs that conflate port-ids and connection-endpoint-ids. Furthermore, multiple connections can be bound to the same port-ids. We will see how that can be used in a moment.

Port allocation is a layer-management exchange to allocate ports at each end. Synchronization is a data transfer exchange. This is a good point to introduce some terminology to make all of this easier to understand. We will refer to the relation between port-ids as a *flow*, while we will refer to the synchronization relation between protocol state machines as a *connection*.

Since synchronization is based only on bounding three times, once ports are allocated, one can simply start sending. The first PDU in each new sequence sets a flag that delta-t calls the Data-Run-Flag. When that flag is set, it means this is a new regime. There are no packets with sequence numbers in the network less than this. Because we've waited $2 * \Delta t$ where $\Delta t = MPL + A + R$

When all is completed, a manager operation deletes the ports, and everything is done. Notice that the state of the connection goes away on its own after $2 * \Delta t$.

Next to the packet switching and datagrams, this is probably the next most important result in all of networking.

Basically, this says we don't need the three-way handshake. The distinction between port-ids and connection-endpoint-ids has major implications.

- It makes the protocol far more secure.
- It greatly simplifies the equivalent of IPsec.
- It enables the ability to change QoS without tearing down the connection.
- It enables having different QoS incoming and outgoing.

And probably other capabilities.

Watson's Result Implication

Watson's result implies an even more fundamental result:

It defines the bounds of networking.
If maximum packet lifetime can be bounded, it is IPC.
If MPL can't be bounded, it's remote storage; it's a file transfer.

A Discussion on IPC

Student: What is IPC?

Professor: IPC means that the message goes in and comes out within an upper bound or it doesn't come out at all ever.

Student: The Internet that we have today doesn't use this delta-t idea, right?

Professor: Correct.

Student: It uses a three-way handshake.

Professor: We will get to that when we get to TCP, but I will explain the answer now.

The Internet explicitly bounds maximum packet lifetime (MPL), but the other two bounds are implicitly bounded by our assumptions about the performance of TCP. So, yes, TCP uses this but not carefully. Consequently, it is not as robust as a protocol that actually incorporated the result. As I said before, the three-way handshake has absolutely nothing to do with it.

The processes are exchanging messages to do all that you are seeing. In fact, one of the earlier slides said it is exchanging three messages and a lot more than that. But the messages are not why it works.

Let me explain it by going back to the beginning of all this. In the early days, they were trying to figure out what was the minimum needed to do synchronization. Dag Belnes, who was a visiting professor at Stanford, wrote a paper called “Single Message Communication,” in which he analyzed how many messages it takes to reliably deliver one message.

Belnes went through all the cases: 1 message, obviously we can lose a message and not know it; 2 messages, send it and get back an Ack but can still lose a message; 3 messages, still conditions where it can be lost; 4 messages, the same; 5 messages and it is reliable, except for a fairly rare case that can be detected if there is a timeout. The goal was to find a way with no timeouts. Timeouts seemed like a kludge.

Separately, another PhD student, Phil Merlin, who was under Dave Farber, was doing an analysis of the same kind of problem from a different point of view. He was able to prove that this could not be done without at least one timeout.

Then Ray Tomlinson at BBN proposed that there should be a three-way handshake at the beginning and then the data transfer and Acks would provide the necessary PDUs for the five-way exchange until the end and then two PDUs (disconnect) to finish it. (Remember Belnes had shown that to reliably deliver one PDU required a five-way exchange. Tomlinson’s solution was that result applied to exchanging more than 1 PDU.) And that is, more or less, where it stood. Then Watson came up with this.

Not long ago, I asked Dick, “How did you do this?” (I suspected that when he saw the previous results where a timeout was always needed, he thought we should take it from the other end. Suppose we have time bounded, what else do we need? And the answer came back, nothing.)

But that wasn’t Dick’s answer, though I still wonder if that wasn't part of it, because his answer was, “I just didn’t feel like we'd gotten to the bottom of it.” He kept thinking about it, rolling it around in his head thinking about what was really going on. Eventually, he saw the solution and was able to construct a proof that the necessary and sufficient condition is to impose an upper bound on those three times.

One other note: This is something we don’t see very much anymore. Finding this solution wasn’t Dick’s job. His group’s job was keeping the computers and networks running for Lawrence Livermore Lab, so that Livermore could continue their nuclear physics experiments. But he couldn’t let go of the idea that there was more to it. He didn’t know there was more to it when he started. But he had to find out one way or another. He is a great computer scientist.

Optimizing Retransmission Control

We have already learned that the semantics of Ack is “I am not going to ask you to retransmit any PDUs with a sequence number less than or equal to this.” The main purpose of an Ack is to clean out the retransmission queue.

We have touched on Selective Ack, which is used to reduce retransmissions with large bandwidth-delay products. With today’s data rates, it easy to have a megabyte or more in flight. If one packet is lost (they are usually about 1500 bytes), it would be very wasteful to retransmit whole megabytes or even a large part of it. Selective Ack and Selective Nack (negative acknowledgement) can be used to force an early retransmission of just the missing PDUs, which can be very useful. With a connection with a large bandwidth-delay product, a Selective Ack or Nack can be sent as soon as a missing PDU is detected to force an early retransmission. However, an Ack will still have to be sent when the missing PDU is received. This has implications for the time A in Watson’s Theorem: “How long to wait before sending an Ack?” One of the times, which has to have an upper bound. This would mean that A would have to be more than 2 RTT. This is significantly longer than immediately sending Acks as PDUs arrive. It is clear that Selective Reject (as it is called by Tanenbaum) is not as simple as it first appears and distinctly beyond the scope of this course.

Constructing Data Transfer Protocols

Now that we have all the pieces, let’s see what goes into creating a protocol. One of the first questions is going to be:

How Many Kinds of PDUs Do We Need?

One of the major (and much argued about) decisions to be made in the design of a protocol is the number and format of the PDUs. We know that there must be at least one PDU to carry the user’s data, which we will call the *Transfer PDU*, but how many others should there be? (The names of PDUs are generally verbs indicating the action associated with them, such as Connect, Ack, Set, Get, etc. Keeping with this convention, we will use *Transfer PDU*, completely aware that in many protocols it is called the *Data PDU*.) Beyond the considerations discussed in the preceding section, there would seem to be no architectural requirements that would require multiple types of PDUs. There are engineering constraints that would argue in specific environments for or against one or more PDU types to minimize the bandwidth overhead or processing.

An oft-quoted design principle recommends that control and data be separated. The natural bifurcation noted previously would seem to reinforce this design principle. Because a PDU is equivalent to a procedure call or an operator on an object, associating more and more functionality with a single PDU (and the Transfer PDU is generally the target because it is the only one that has to be there) is “overloading the operator.” TCP is a good example of this approach. Minimizing the functionality of the Transfer PDU also minimizes overhead when some functionality is not used. However, there is nothing in the structure of protocols that would seem to require this to be the case.

For data transfer protocols, the minimum number of PDU types is one, and the maximum number would seem to be on the O(m+1) PDU types—that is, one Transfer PDU plus m PDU types, one for each loosely bound mechanism—although undoubtedly, some standards committee could define a protocol that exceeds all of these bounds. For most asymmetric protocols, the maximum may be on the O(2m), because most functions will consist of a request and a response PDU. In symmetric protocols, the “request” often either does not have an explicit response or is its own response. Hence, there are O(m) PDU types.

Good design principles favor not overloading operators. It would follow then that there should be a PDU type per loosely coupled mechanism. Separate PDU types simplify and facilitate asynchronous and parallel processing of the protocol. (This was not a consideration with TCP, which in 1974 assumed that serial processing was the only possibility.) Multiple PDU types also provide greater flexibility in the use of the protocol. Loosely coupled mechanisms can be added to a protocol so that they are backward compatible. (For a protocol with one PDU type, adding a mechanism will generally require changing the one PDU format.) Similarly, a mechanism can be made optional by simply using a policy that never causes the PDUs to be generated. (For a protocol with a single PDU type, the PCI elements must be sent whether or not they are used, or a more complex encoding is required to indicate whether the elements are present.) With multiple PDU types, no overhead is necessary to indicate a PDU’s absence. From this it would seem that more, rather than fewer, PDU types would generally be preferred.

Do this and our earlier discussion of piggybacking Acks imply that protocols should always have more than one PDU type? Not necessarily. This is not a case of right or wrong, true or false. The choices made are based on the requirements of the operating environment. As we have seen, TCP was designed for an environment with a very large proportion of very small PDUs. Clearly, the choice was correct; it saved 30% to 40% in bandwidth. In an environment that does not have a lot of small PDUs, and especially one with a large range of traffic characteristics, optimization for small PDU size loses its weight while the need for greater flexibility gains weight, indicating that under different conditions, a different solution might be more appropriate.

Contrary to the often-knee-jerk response to these decisions, the choice of a mechanism is not right or wrong but a case of appropriate boundary conditions. It is important to remember the conditions under which these sorts of choices are appropriate. Although it might seem unlikely that network requirements will ever return to the conditions of remote character echoing, history does have a way or repeating itself, although usually in a somewhat different form, which is what makes the topologist’s vision defect so useful. These vision-impaired architects are able to recognize when this new doughnut looks like that old coffee cup!

Let’s take a look at the mechanisms. Each of the mechanisms uses information in the PCI.

- What information has to be with the data?
- What information doesn’t have to be with the data? It could be but doesn’t have to be.
- Is there any reason for it not to be?

The Types of Mechanisms

A protocol mechanism is a function of specific elements of the PCI (fields) and the state variables of the PM that yields changes to the state variables and may generate one or more PDUs. These elements of PCI are conveyed

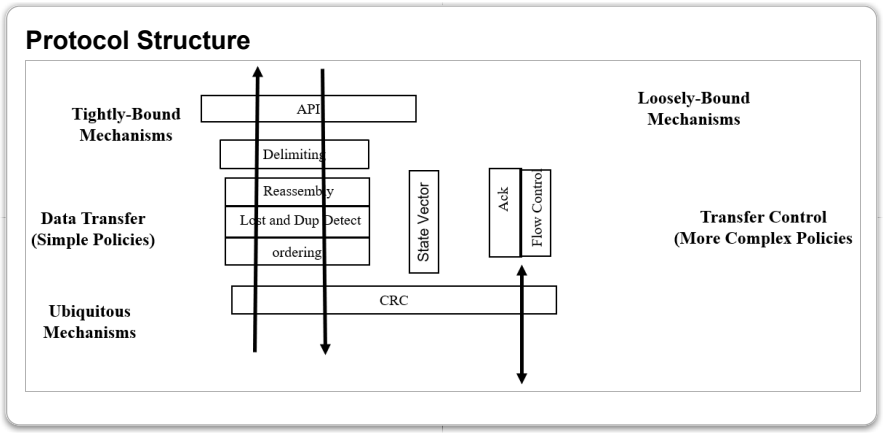
to the peer PM(s) by one or more PDUs to maintain the consistency of the shared state for that mechanism. As we have seen, some mechanisms may use the same elements. (For example, sequence numbers are used for lost and duplicate detection, retransmission control, and flow control.) As we have seen there are three kinds of mechanisms:

Tightly-bound mechanisms – those that use PCI that must be with the Transfer PDU. This includes mechanisms like sequencing, fragmentation, etc. where policy is imposed by the sender.

Loosely-bound mechanisms, – those that use PCI but that doesn't have to be with Transfer PDU. This includes mechanisms like retransmission and flow control, where the policy is imposed by the receiver. It is the receiver that decides to Ack. It is the receiver that decides when to extend more credit so that the sender can send more data.

Ubiquitous mechanisms – those that use PCI that must be with all PDUs, such as error checking and delimiting. (We might call them *very tightly bound*.)

A Clean Protocol Design

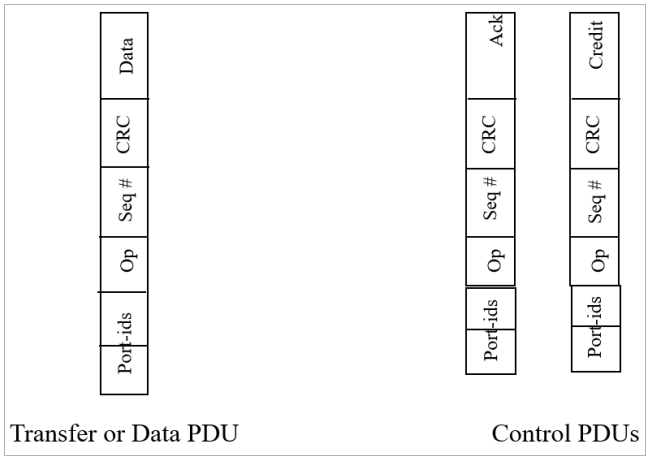


Notice how nicely they separate into the two kinds. Tightly-bound mechanisms write to the state vector. Loosely-bound mechanisms read the state vector. The computation for the tightly-bound mechanisms is relatively simple and fast. The computation for the loosely-bound mechanisms is more complex. The two kinds of functions barely interact. If all goes well, occasionally the loosely-bound side may shut down a queue from sending PDUs, but that is it. The tightly-bound side operates at the rate of the data transfer, while the loosely-bound side needs to generate feedback based on the retransmission and flow control policies.

That leads to a protocol structure where we have the ubiquitous mechanisms, and then the tightly-bound mechanisms, the sequencing putting PDUs in order, lost and duplicate detection, fragmentation/reassembling, and delimiting. Then there are the loosely-bound mechanisms of retransmission and flow control, which coordinate through a State Vector.

A Transfer PDU

A Transfer PDU and at most one PDU for each loosely coupled mechanism.



Notice with this structure the only difference between a Transfer PDU for a reliable protocol and a connectionless unreliable protocol is whether a sequence number is used for ordering, i.e., whether the ordering policy is null or not and whether loosely bound mechanisms are used. If they are not used, they are simply never sent.

That argues very hard for these PDUs to be generated separately. Consider the data transfer functions; they are all relatively simple, bookkeeping, putting sequence numbers on outgoing PDUs, putting incoming PDUs in order in a queue, and then merging or breaking apart PDUs that are in the user data field. Whereas in data transfer control, we are making estimates of buffer usage, what has been received in order and out of order, what we should be sending X, and how long we have to wait before sending an Ack. This side is computationally more complex than the data transfer side.

This kind of clean separation is *exactly* what you want to see in a system design. When you have functions of different time cycles, you would like to see them relatively decoupled from each other. And that's what's happening here.

So, basically, we have a Transfer PDU with port-ids, a Type field that says this is a Transfer PDU, a sequence number, CRC, perhaps fields for fragmentation/reassembly, and data. And then there are the control PDUs for retransmission and flow control. These control PDUs are very short. It is often the case that the Right Window Edge is calculated by adding a Credit field to the latest Ack value, hence it would make sense to combine them.

We arrive at two PDU types: One for data and one for control. Next there is syntax.

Syntax

Designing the syntax of a protocol can have a major effect on its efficiency. For some reason, people seem fascinated by the process of designing message headers. I have never quite understood the fascination. Notice

that, in general, the syntax of data transfer protocols has several considerations, among them the desire for low overhead processing PCI to keep it fast. We are trying to move data as fast as possible. The syntax of lower layers tends to avoid variable length fields. If variable length fields are required, put them at the end of the PCI, so it's faster to access the other fields. Avoid complex and variable structures in IPC protocols. Try to align fields on byte boundaries, which also makes it easy to access the fields. If we look at how these fields are used, there are only two kinds of fields. Although they are all integers, some of them are just labels that we use to index or hash into a table. Others, like the sequence or numbers or credits, are fields on which we do simple arithmetic modulo the width of the field. The procedures of the protocol are the same.

Armed with this, we can apply what we learned about the Presentation Layer and use the concepts of abstract syntax with encoding rules to make protocols invariant with respect to syntax.

We can define an abstract syntax for data transfer protocols with different encoding rules for different environments.

The Final Result

If we take abstract syntax with encoding rules, separation of mechanism and policy, and such together, we realize

There's only one data transfer protocol
with different policies and encoding rules

By separating mechanism and policy, by making protocols invariant with respect to syntax, by decoupling data transfer and data transfer control through state vector, we have made the argument that there is one simple implementation for a single data transfer protocol that can be configured to cover the entire range of protocols.

We have actually done this and we've tested it against probably 50 or more protocols. We have yet to find an exception.

Summarizing What We Know About Layers and Protocols

What We Know about Protocols and Layers:
met_cs535_mod2_lec2_slide57_animation video cannot be displayed at the printable page. Go to the module page to watch.

The necessary and sufficient condition for a layer is a locus of distributed shared state with bounded maximum packet lifetime.

Port-ids are necessary and sufficient to ensure layer separation and independence. The port-id is the only identifier shared between an (N+1)-layer and an (N)-layer, and it is never carried in protocol. This achieves the isolation of the layers.

There may be multiple layers of the same scope, but the functions assigned to the layers must be independent.

What Is This "Loci" Stuff?

The word *locus* (or loci in the plural) is probably not familiar. Let me explain. They used to teach this is geometry, and they don't teach it anymore.

I have a good example of *locus*. Martin Gardner used to write the Mathematical Games column of *Scientific American*. They were fun and interesting. We all learned about Flatland from him. Once he had an article on circles. The article started with a quotation like this:

"Mommy!! Mommy!! Why do I always go around in circles?"
"Shut up or I will nail your other foot to the floor!"
– Sick joke circa 1956.

Then the article starts: "Thus, we see that a circle is the locus of all points equidistant from a given point."

Suppose I take a piece of string and hold one end against the whiteboard and pull the string taut at a fixed distance. Then with a marker attached to the string at the pulled end, I draw on the board keeping the string taut. I will get a circle: The locus of all points equidistant from a given point.

Suppose I fix the string loosely at two points on the whiteboard and use a marker to stretch the string taut between them. If I then draw around that, what do I get?

An ellipse!

An ellipse is the set of points for each of which the sum of the distances to two given foci is a constant. (Wikipedia)

And hyperbole is the set of points for each of which the absolute value of the difference between the distances to two given foci is a constant. (Wikipedia)

A locus is a set of all points whose location satisfies or is determined by one or more specified conditions. (Wikipedia)

For us, we are saying that functions with particular properties are located together.

There is, at most, one SDU Protection Module where we do data corruption detection and one multiplexing task per (N-1)-port. Every (N-1)-port could be to a different layer (or different ports of the same layer), with different technology that can have different error characteristics, so we can have a different SDU Protection Module for each one. Because multiple connections may be mapped to an (N-1)-port, there is a multiplexing function for each one. Hence there can be multiple instances of the data transfer protocol multiplexed onto each of the different out-going ports.

There is one Relay Task per IPC process. Because PDUs on any incoming port can be routed to any outgoing port, there is only one relaying task.

What We Know About Protocols and Layers
met_cs535_mod2_lec2_slide58_animation video cannot be displayed at the printable page. Go to the module page to watch.

There is one error flow control state machine per flow allocated.

Those with feedback requirements required data transfer control bounding the three timers. Watson implies decoupling ports and connection end-point identifiers.

While the model does not impose the restriction, the conclusion we have reached is that there should never be flows without flow control. They is a potential denial-of-service attack vector. Retransmission control is not required, but flow control is. (Data Transfer is still connectionless in the sense that resource allocation is dynamic and each PDU is forwarded independent of other PDUs. This just says there must be a way to control the flow.)

Data transfer functions are decoupled from data transfer control through the data structure or state vector.

These are similarly decoupled from layer management or the resource allocation or resource management.

What we see is not so much layering within the same scope as orthogonal separation into three loci of processing. Layer management is where routing is done, including a database, in a sense a large state vector. (For some reason we call these things state vectors when they're small and Management Information Bases when they are large. But they are the same thing.) This is the same thing we saw before, because routing and resource allocation are computationally a lot more complex and have a longer cycle time or “duty cycle.” Hence, they are decoupled from the high-performance functions through the state vector.

What we see in the layer is that the functionality in each system decomposes nicely into three loci of processing. Moving left to right, we have very fast, computationally simple data transfer, very short duty cycle; followed by feedback mechanisms decoupled through a state vector with a more complex and longer duty cycle; followed by layer management functions decoupled through this state vector-ish Resource Information Base even more complex and longer duty cycle.

Again, this is exactly what you want to see when you're doing a design like this.

Specifying a Protocol

Once we have designed a protocol, we have to be able to tell people precisely what they have to do to implement it so that it will talk to other implementations. This turns out to be more difficult than you might think.

Back in the ARPANET days, they revised the Telnet Specification and a new document was produced (RFC 495). Everyone thought it was a very good specification, clear, concise, easy to understand. That is, until a new site totally new to the ARPANET, who hadn't been involved in the discussions joined the ‘Net and wrote an implementation that bore no resemblance to anything anyone else had done. It was realized that specifying a protocol was not an easy problem.

First of all, the conditions in different systems are not the same. The best way to implement in one system may be the worst in another, or at least not great. There is also the concern that using the implementation as the specification will give one group an advantage in the marketplace. So, we want an implementation-independent specification methodology.

There was a lot of work done on how to do formal specifications of protocols. They can be quite useful, especially for complex protocols. With a formal specification, it is possible to prove that various properties like progress (the protocol will always make progress); that there are no deadlocks; etc. It is possible to generate the implementation from the formal specification.

We have tried many different models. Even by the late '70s, there were over 20 different approaches to formal description of protocols. The main key categories of techniques are the finite state machine models, language models, and temporal ordering models. With temporal ordering, the specification is written in terms of conditions and when they have to exist. The idea is that the implementation has to have reached some condition before some event.

It is desirable that the specification to say as little about the implementation as possible in order to give the implementor as much freedom as possible. Temporal ordering is probably the one that does that best. The trouble is I have never found anyone who could design in temporal ordering. Most everyone designs in terms of finite state machines and then translates into a temporal ordering scheme.

One caution, especially with some of the predicate logic techniques, is that the formal method becomes more complex than the implementation. If the formal specification is more complex than the code, there is a higher probability of bugs in the formal specification than in the code. (It was also discovered that “commenting” a formal specification was as important as “comments” in code!)

Generally, the formal specification was included with a more traditional prose specification. Consequently, one of the questions that arose was: If there are bugs, which one takes precedent?

My answer was *none of them*.

The only definitive definition is in the heads of the designers. One has to consider all specifications of the same protocol and figure out what they say, what is the correct behavior, and how or if they have to be modified. It is necessary to determine what was the intent. Because they are all done by humans, one can't be sure that any of them are more definitive than any other.

Why We Can’t Design the Perfect Transport Protocol

Over the years, there has been much discussion and many proposals for new and “better” transport protocols to replace TCP—none with much success. There was always some major category of traffic that the proposal did not handle well. On the other hand, we have been fairly successful with new data link protocols.

The separation of mechanism and policy makes the reason for this clear. Transport protocols are intended to support the requirements of their applications. There are roughly six or eight mechanisms in a transport protocol. By not separating mechanism and policy, we have (unintentionally) been saying that we expect to find a single point in an eight-dimensional space that satisfies all the requirements. When put this way, this is clearly absurd! There are no panaceas, but there is a lot of commonality! If we separate mechanism from policy, the problem can be solved: a common protocol with different policies for specific situations.

Why was there more success with data link protocols? Data link protocols are tailored to address the requirements of a particular physical medium. Hence, their performance range is narrower. Thus, a single set of policies will satisfy the problem.

Design Issues

There are a number of design issues that must be considered in designing a protocol and figuring out what the policies should be.

- We need to determine the range of operation of the protocol. If it's near the media, the properties of the media will dominate.
- If it's near the applications, it is the range of requirements of the applications that dominate.
- Determine the service expected from the layer below. What is the minimum quality of service that can be expected?
- How much shared state is required? While the model allows the data transfer side to operate alone, for implementation we have adopted the view that all flows must be flow controlled whether retransmission control is used or not. Data transfer without flow control is a denial-of-service threat.
- Determine whether the data transfer is going to be a stream or SDUs.
- What is the maximum PDU size? We would like to be able to send large PDUs, but if the media is hostile, that might lead to more retransmissions. For example, in wireless, smaller PDUs may be needed rather than larger ones to reduce retransmissions. When sending large PDUs over a hostile environment, there is a higher probability that they will be damaged than if sending more, smaller PDUs. It may be necessary to retransmit a few of them but less data overall. That needs to be taken into account. That said, you also have to take into account how long a PDU is relative to a burst of noise. For example, with wireless, if something like lightning or an electric motor causes noise, how long do PDUs last? If the network is operating at 50 Kbps, those PDUs are going to occupy the media a lot longer than if they are operating at 100 Mps. At 100 megabits per second, it is more likely that a burst of electrical noise will destroy an entire Ethernet PDU of 1500 Bytes.
- How many PDU types? We have argued here that you probably want to have at least two: one for data and one for control.
- How large should the sequence numbers field be? The requirement for sequence numbers is to be able to keep the pipe full until an Ack can be returned. The sequence number field has to be a power of 2 larger than the maximum number of PDUs that can be sent, given the data rate in more than a round-trip time. (Some extra space is prudent.) At the same time, we don’t want the sequence number field to be any larger than necessary. What units is the sequence number incremented in? PDUs, bytes, bits. I have seen all three used in different designs. How is the initial sequence number chosen? Always zero? Random?
- What is the retransmission strategy? Piggybacking? Use Ack and/or Nack?
- Estimating the time-outs needed for the layer and individual flows? Estimate the roundtrip time? Etc.
- What is the flow control policy?

Implementation Issues

Here are the implementation issues to resolve:

- You need to minimize context switches and data copies. (Both of these are killers in an operating system. They really soak up time.)
    - If one is not careful, passing PDUs from layer to layer can involve both context switches and data copies. Avoid copying data all the time. Leave the data in one place and manipulate it there.
- Is the protocol to be implemented as a process to thread or a library?
- There is likely to be more than one process accessing the same queue, so there will need to be some way

of doing a critical section. Will the implementation buzz, use semaphores, or monitors? How will critical sections be handled?

- We have discussed flow control between the sender and receiver, which implies a need for flow control across the interface to tell the process above to slow down. How should it be done?

Buffer Management

This is a very important topic. It is always a critical problem. This brings up one of my favorite papers. The question is whether it is better to allocate buffers to connections, or to take buffers from a pool for all the connections. It turns out, this is one of the few no-brainers in computer science. You want to pool the buffers. Never statically allocate buffers to what uses them.

In a paper by Peter Denning, whom you may remember from operating systems as the person who invented the working set paging algorithms, Denning develops a statistical model for console behavior, i.e., terminal behavior in a timesharing computer. This is essentially the same problem as buffer management for protocol flows and it applies equally well. The paper was written *in 1968*.

The reader is likely to laugh at the parameters on these numbers, but this holds equally well, even today. Denning did this very nice queuing model and he works out all the details and plugs in some numbers. Referring to the figure below, Denning finds the following:

Pooled vs. Static Buffers

Total Buffer Size (in bytes)	Probability of Overflow Pooled	Probability of Overflow Static
750	.006	.90
1000	10^{-6}	.76
1500		.44
2000		.22
2500		.10
3000		.05
3500		.02
4000		.01
4500		.004

Blank because the numbers were too small to be represented by the computer.

It takes to here with static to get to where we started with pooled.

If there are 50 users, the ratio of characters/interrupt is 10 characters per interrupt.

Let's start with 750 bytes of buffer space. (That is not kilobytes. But bytes!) If buffers are statically allocated, the probability of running out is .9. It's virtually assured. With pooled buffers, i.e., all 50 users take buffers from one buffer pool, the probability of that running out is .006.

If the number of buffers is increased to 1000, the probability with static allocation drops to .76, about 3/4ths—not much change. However, with pooled buffers, the probability drops to 10-6. The chances are very SLIM that you're going to run out.

After this, the reason the column of the pooled case is empty is that the probabilities become so small that it couldn't be represented in the computer.

Notice, the buffer size has to get all the way down to 4500 bytes to get the static allocated probability, where we

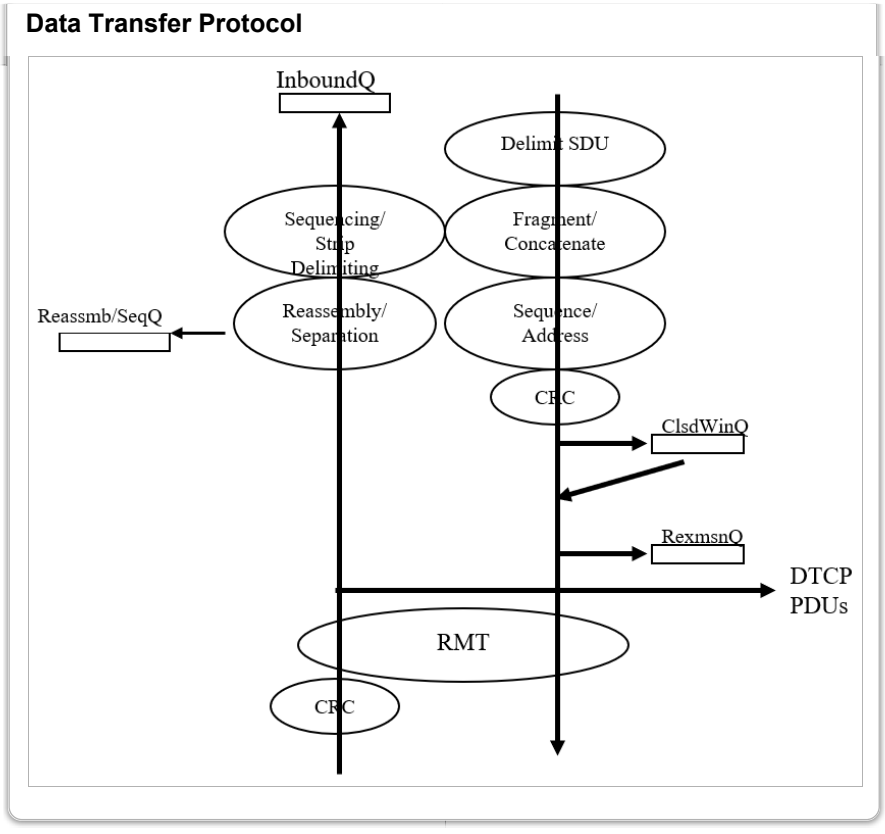
started with pooled allocation at 750 bytes. What does this mean in terms of the allocations? For the static case, at 750 bytes over 50 terminals, that implies there are 15 bytes allocated to each terminal. With an average of 10 characters/interrupt, that is about 1.5 interrupts. In other words, the OS needs to empty the buffer relatively quickly because it may not have room for the next interrupt. For the pooled case, there are enough buffers to handle 75 interrupts per second. This would mean that all of the 50 users were typing at least 10 characters per second and half of them twice that rate. Highly unlikely considering that most 10 character interrupts would have the user waiting for a response and not typing.

To get to the pooled case requires 4500 bytes or 90 bytes per terminal, the equivalent of 9 interrupts before running out. This provides much more leeway.

This is one of the few no-brainers I have ever come across. There is no weighing the trade-offs. Ninety-nine percent of the time you never want to statically allocate buffers. But as we noted at the beginning of the course, there are always exceptions. Later on, we will encounter a case where running out of buffers might be desirable.

In networking, we have seen manufacturers do this, time and again. They usually do it out of ignorance, but it does allow them to offer bigger and bigger boxes with higher and higher margins. This result wasn't rediscovered for routers until 2004 when they found that 90% of the buffers on high-end routers were not necessary. This negated the need for most high-end routers. (As an indication of the importance of this result, in a footnote in a follow-up paper, the authors noted that a VP of a major router company offered to fund their research, if they would discredit their original findings.) Even though terminal and router traffic are very different, given the huge differences in Denning's results, it may be the case that even fewer buffers are required.

Reviewing the Engineering of Data Transfer

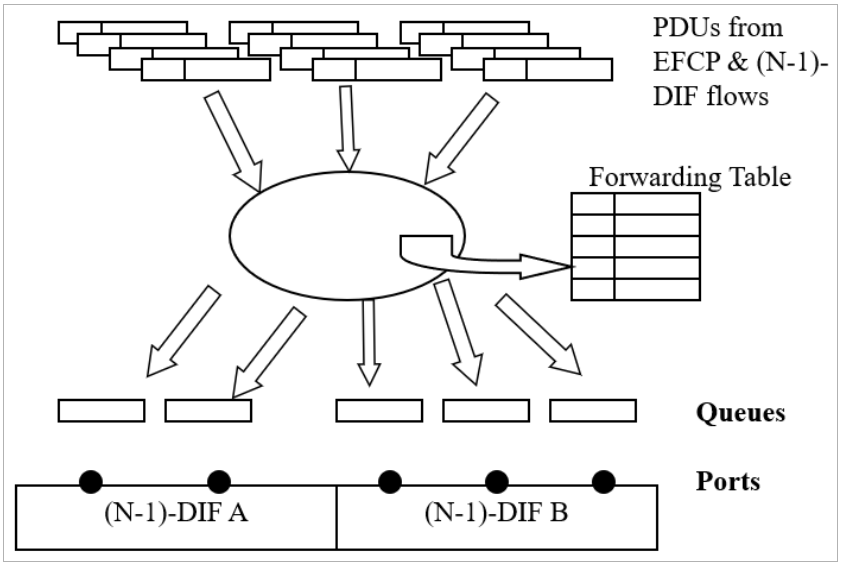


We have done a thorough job investigating the logical structure of data transfer protocols and found some interesting insights that were not initially obvious. For the most part, this is the implementation structure. However, one should never consider a protocol specification as an implementation design document.

As I always say, *anything you can get away with that isn't notice able from the outside is legal.* In this figure, we have a small example of that. Notice that for incoming PDUs, the first operation, even before being demultiplexed and delivering a PDU to the appropriate instance of the protocol, is to determine if the PDU has been corrupted. This must be first, otherwise we can't trust any of the values in the PCI. However, notice that for outgoing PDUs, this checking can be done much earlier before flow control is applied, before multiplexing.

Why? By this time, the PDU is fully formed. There will be no more changes to it. Why wait until just before the PDU is handed to the layer below? Are there advantages to doing it earlier? Definitely. Notice that if it is done early and the flow control window is closed, the PDU is put on the ClosedWindow queue and is ready to go. When the window opens, the PDUs can be moved to a queue for sending. Nothing else is needed to be done. Once it is sent, it is placed on the retransmission queue and, here again, if it needs to be retransmitted, it is ready to be retransmitted. No further processing is required.

The Relaying and Multiplexing Task

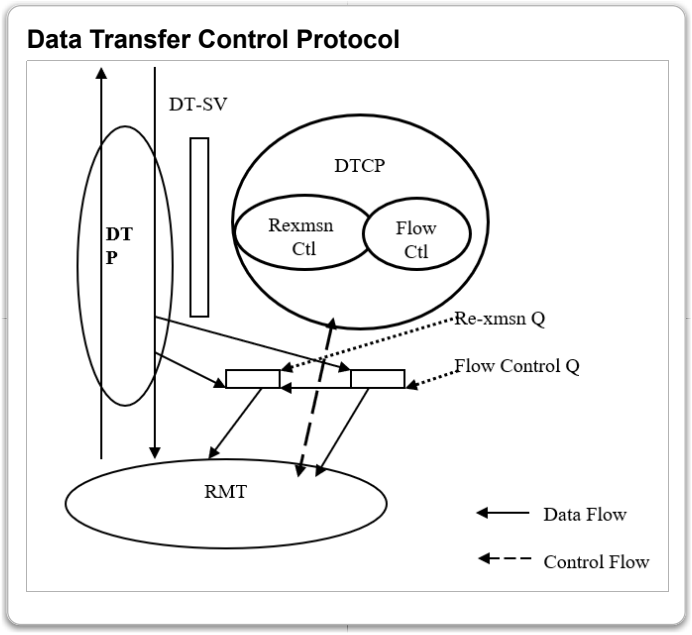


The relaying task has two sources of inputs: PDUs generated locally and those that are incoming. Locally generated PDUs are outgoing.

1) For outgoing PDUs, the relaying task will consult the forwarding table and send the PDU to the appropriate multiplexing task for the appropriate (N-1)-port it is being routed to.
2) For incoming PDUs, the relaying task will inspect the address and if it is the address of this IPC Process, it will be delivered to the appropriate data transfer instance. If not, it will be handled by 1).

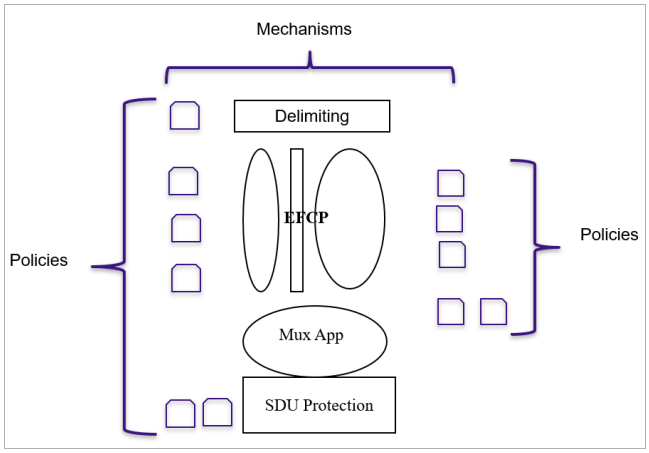
The multiplexing task will use the QoS-cube-identifier to assign the outgoing PDU to the appropriate queue of outgoing PDUs. For incoming PDUs, the PDU is handled according to 1) above.

Finally, the feedback functions are brought in, and we see how they relate to the other two. Note that there is an opportunity here to assign the Control (feedback) PDUs to a different (N-1)-flow than normal data.



Leveraging the Discovery That There Is a Single Data Transfer Protocol

The Structure of Data Transfer Protocols



We have found that this structure repeats in layers of different scope, but what is the way to implement it in a system?

Of course, the big result of this chapter is that there is one data transfer protocol with different policies and different encoding rules. We have applied this to over 50 protocols and found no contradictions. We have been able to identify mistakes in the structure of other protocols. This creates a large amount of commonality. Is this just an ivory tower exercise or does it really have some advantage?

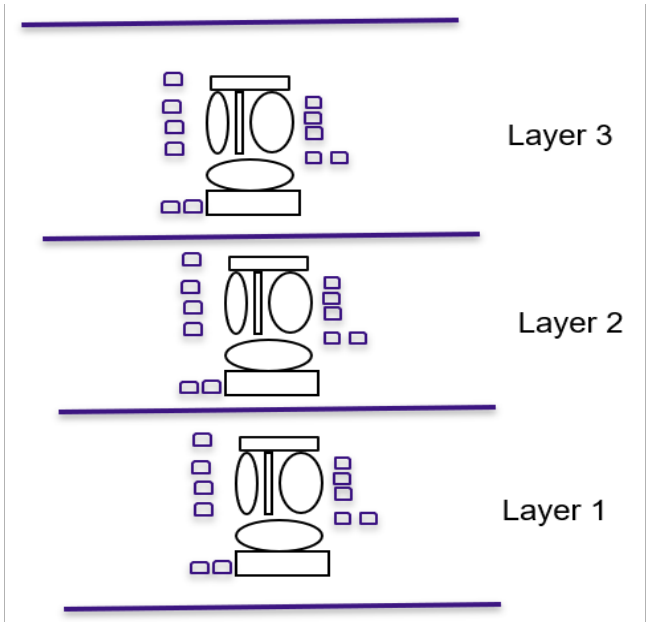
There are huge advantages. First of all, given this level of commonality, we can create policies that allow protocols to behave in far more complementary ways. This will greatly improve our ability to manage these networks and avoid side effects and unexpected consequences due to unforeseen dependencies. Far more complex management changes can be made with confidence.

But the really big impact is the complexity collapse. Having the same implementation for the protocols at all layers means that once the code is wrung out for one, it is wrung out for all. If the data transfer protocol is secure at one layer, it is secure at all layers. There is also a massive collapse of complexity across the repeating layers.

At first blush, one might think that this would imply doing something like this:

Repeating Layers Would Seem to Imply

A pipeline like this would be the way to go.



Repeating layers would seem to imply this structure.

But we can do far better.

How much better? Orders of magnitude better. Here is how.

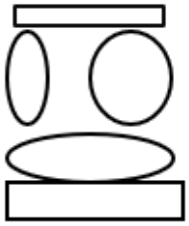
Remember automata theory? (Some readers are probably thinking, he has to be kidding! But no, it is really applicable.)

This relies on using those models. To refresh:

- A **Turing Machine (TM)** is a mathematical model of computation that defines an abstract machine that *implements a specific algorithm* that is executed by manipulating symbols on a semi-infinite tape.
- A **Universal Turing Machine (UTM)** is a *Turing Machine* that simulates an arbitrary *Turing Machine* on arbitrary input. The *universal machine* essentially achieves this by reading both the description of the *machine* to be simulated as well as the input to that *machine* from its own tape.

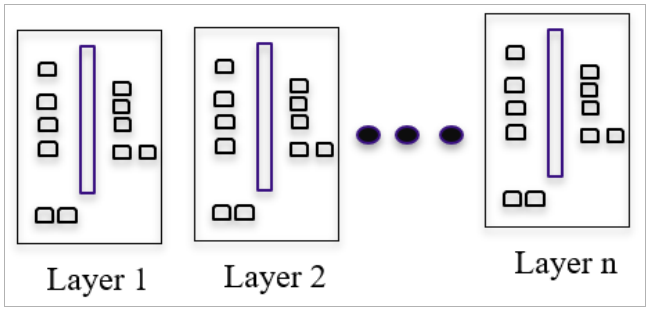
So a Turing Machine executes a single algorithm, while a Universal Turning Machine can execute any Turing Machine and its algorithm. In other words, a UTM is a computer and a TM is a program executing on a computer.

Then suppose:

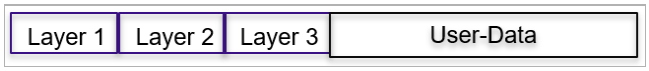
The protocol mechanisms are analogous to a Universal Turing Machine  .

Each set of policies, syntax definition, and state vector for a layer is analogous to a Turning Machine. Let's call it a layer vector.

The policies define a Turning machine (TM) and there is one for each layer.



And of course, the PDU is analogous to the tape.



When a PDU arrives, the mechanism implementation loads the layer vector for layer 1; processes the PCI for layer 1, updating its state vector, etc.; moves the tape, err, PDU down; loads the layer vector for layer 2; processes the PCI for layer 2, updating its state vector; moves the PDU down, and so on. This can be done in both directions.

Furthermore, suppose the mechanism implementation was in silicon, and the layer vectors were a combination of configuration data, state vector, and small modules of software. We just reduced the layers to a single implementation. A huge complexity collapse. Power requirements just went through the floor, and performance went way up.

Can this really be done?

Been there done that. We did an operating system based on this concept 40 years ago that was proven secure. There is already a preliminary implementation. It is perfectly feasible.

Nota Bene

It is necessary to point out that every insight we have come across so far (and there are more to come) have come from the theory, not the implementation.

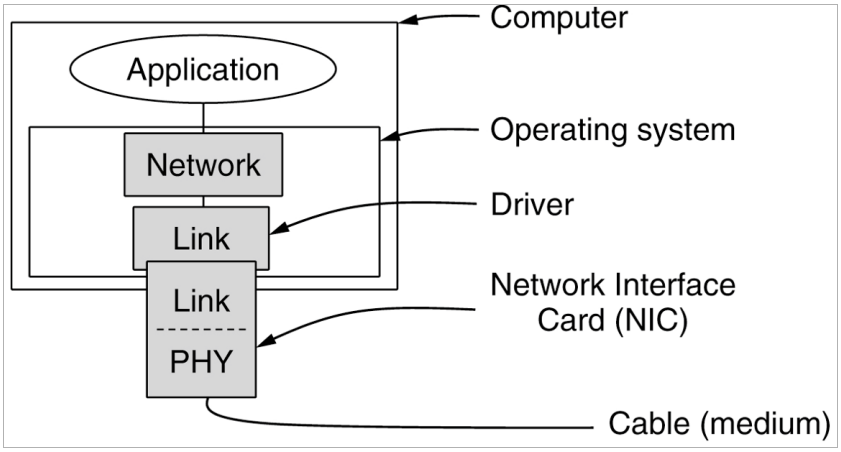
An Exercise in Separating Mechanism and Policy

Tanenbaum developed six data link protocols of increasing complexity. While these are error and flow control protocols as this lecture has been developing, they are distinctly data communication protocols, rather than networking protocols. He started with an idealistic protocol to make the point and then, for the most part, followed the history of their development. Be aware that while this was going on from probably the late '60s to the mid-'70s, the modern form of error and flow control protocol was developed in 1971–72 and already existed when the later versions of these were being developed.

For exposition in a textbook, Tanenbaum does make some simplifying assumptions:

- **Independent processes** are assumed. The physical, data link, and network layers are assumed to be independent. Communication between them is by passing messages back and forth.

Initial Simplifying Assumptions



Implementation of the physical, data link, and network layers.

- **Unidirectional communication.** It is assumed that a reliable connection-oriented service is used between A and B, and A has an infinite supply of data to send to B. Later, he considered simultaneous traffic from B to A.
- **Reliable machines** and processes. It is assumed that the machines do not crash.

There is an unstated assumption that the upper layer will take any data as soon as it arrives. Tanenbaum doesn't state that assumption, but it is there, in the way the protocols are defined.

As we saw earlier this avoids needing to add two additional commands, Receiver Not Ready and Receiver Ready. He also assumes that any error checking on the data has been done.

This is a good place to apply the separation of mechanism and policy to these protocols. In class, we might divide up into groups to do the last three, but that isn’t possible here.

The list of mechanisms we have to choose from are:

- Delimiting
- Packing SDUs into PDUs

- Ordering/Sequencing
- Lost and Duplicate Detection
- Data Corruption
- Flow Control
- Retransmission Control (Ack)

So here are the solutions for the first three protocols:

Protocol 1: Utopian Simplex Protocol

Delimiting:	None specified
Packing SDUs into PDUs:	1:1
Ordering/Sequencing:	None specified, assuming point to point
Lost and Duplicate Detection:	None specified
Data Corruption Detection:	None specified
Transfer:	If there is data, send it
Flow Control:	None specified
Retransmission Control:	None specified
Number of PDU Types:	1, Transfer

Notes: This is pretty simple and really utopian!

Protocol 2: Simplex Stop-and-Wait Protocol

Delimiting:	None specified
Packing SDUs into PDUs:	1:1
Ordering/Sequencing:	None, Stop-and-Wait doesn't need sequence numbers.
Lost and Duplicate Detection:	None specified
Data Corruption Detection:	None specified
Transfer:	• If receiving Data and not waiting for Ack, then send. • If receiving Data and waiting for Ack, then send when Ack is received.
Flow Control:	Stop-and-Wait
Retransmission Control:	None
Number of PDU Types:	2, Transfer, Ack

Notes: Here Ack is used for flow control.

Protocol 3: Simplex Protocol for a Noisy Channel

Delimiting:	None specified
Packing SDUs into PDUs:	1:1
Ordering/Sequencing:	Sequence number
Lost and Duplicate Detection:	Use the sequence number
Data Corruption Detection:	Checksum
Transfer:	If there is data, send it
Flow Control:	None specified
Retransmission Control:	*Sender*: When Transfer PDU sent, set timer for RTT+e: if timer expires, retransmit all unacked PDUs *Receiver*: When User takes data, send an Ack
Number of PDU Types:	2, Transfer, Ack

Notes: This protocol has a bug. See the text for why sequence numbers are required to avoid duplicates. If the user is delayed in taking the data, retransmission will time out and retransmit unnecessarily, and it will continue to do so. To fix this, you need two commands: RNR, Receiver Not Ready; RR, Receiver Ready.

Protocol 4: One-Bit Sliding Window Protocol

Delimiting:	None specified
Packing SDUs into PDUs:	1:1
Ordering/Sequencing:	None, Implicit in the one-bit window
Lost and Duplicate Detection:	Implicit
Data Corruption Detection:	Assumed but not specified
Transfer:	If there is data, send it.
Flow Control:	Fixed 1-bit Window makes it Stop-and-Wait
Retransmission Control:	*Sender*: When Transfer PDU sent, set timer for RTT+e: If timer expires, retransmit unacked PDU *Receiver*: Only Ack if data is received correctly and in order.
Number of PDU Types:	1, Transfer PDU and piggyback the Acks

Notes: One-bit sliding window, introduces the sliding window for both flow and retransmission control, where sending Ack implies advancing the window, but the one-bit window effectively makes it stop-and-wait.

Piggybacking Acks creates a problem if there is no data going the other way.

Protocol 5: Go-Back-N Protocol

Delimiting:	None specified
Packing SDUs into PDUs:	1:1
Ordering/Sequencing:	Sequence Numbers, if max window is N, then sequence number field is $2 * log_2(N)$.
Lost and Duplicate Detection:	Using sequence numbers
Data Corruption Detection:	Assumed but not specified
Transfer:	If there is data, send it
Flow Control:	Sliding window
Retransmission Control:	*Sender*: When Transfer PDU sent, set timer for RTT+e: if timer expires, retransmit all unacked PDUs *Receiver*: Only Ack if data is received correctly in order.
Number of PDU Types:	1 or 2, Transfer, Ack

Notes: The Go-Back-N requires correct packets to arrive in order. Otherwise, it withholds the Ack and lets the sender retransmit.

Protocol 6: Selective Repeat Protocol

Delimiting:	None specified
Packing SDUs into PDUs:	1:1
Ordering/Sequencing:	Sequence numbers, if max window is N, then sequence number field is $2 * \log_2(N)$.
Lost and Duplicate Detection:	Using sequence numbers
Data Corruption Detection:	Assumed but not specified
Transfer:	If there is data, send it
Flow Control:	Sliding window
Retransmission Control:	*Sender*: When Transfer PDU sent, set timer for RTT+e: if timer expires, retransmit just unacked PDU *Receiver*: When user takes data, send an Ack, piggybacked on Transfer PDU
Number of PDU Types:	2 or 3, Transfer, Ack

Notes: Given the result from Watson, this requires a very large value for the bound on the A time, how long to wait before Ack. The sender's retransmission timer is going to be RTT+A+e. Given the scenario in the T, if a PDU is corrupted but others are received, then the retransmission timer must be much larger than RTT+e. Otherwise, the subsequent retransmission timeouts on subsequent packets will also time out, forcing their retransmission and defeating the purpose of Selective Repeat. It would seem that the retransmission timer has to be at least 2.5 RTT+e, implying that A should not be e but something greater than .5RTT.

There is more to think about here. Given that the bandwidth/delay product 2.5RTT prevents the next PDU from being retransmitted, there can be more to think about, such as deriving an expression for what it should be. There is a project here, especially if one considers Selective Ack or Selective Nack.

Data Link Layer Protocols in Practice

Read

Read Tanenbaum Section 3.5, pp. 252–261.