Stalking the Upper Layer Architecture

Introduction

Before we launch into the details of Error and Flow Control Protocols, we are going on a little trip through some of the early application protocols. This material is not covered in Tanenbaum, but it is covered in Chapter 4 of Patterns. The reason for doing this is not because you should know how these particular applications work, but because there are some very interesting insights, design patterns, and lessons to be learned. It is doubtful that you will ever see these exact same problems again, but you may well see the same pattern again. Or it might give you the insight to think about a problem differently. It is a window into a different way of thinking.

We start with a couple of quotes here.

> "You have Telnet and FTP. What else you do you need?"
> —Alex McKenzie, 1975

One is from my now good friend Alex Mckenzie, who was in charge of host software development for BBN systems on the ARPANET and wrote the Telnet RFC. When I was just a wee graduate student, I attended one of the early INWG meetings on the West Coast. As the meeting broke up that evening, a tall Alex turned to me and said in his deep bass voice, "So why are you here, kid!?" A little intimidated, I replied, "I'm interested in the upper layers, sir," to which he responded, "You have Telnet and FTP. What else do you need?" I was dumbfounded. I could just see all sorts of applications. Turns out he wasn't far off.

> "If the virtual terminal is in the presentation layer, where is the power supply?"
> —Al Reska, 1979

The other quote comes from an early (1979) OSI meeting in London, from the U.S. delegate from the Teletype Corporation. Many of you have probably never heard of the company or seen one of their products. A Teletype was an electro-mechanical terminal. Basically, a mechanical typewriter that could decode ASCII text and cause a character to be printed on paper. They were noisy. Some referred to it as "your basic steam terminal." During a meeting on upper-layer architecture, the delegate said, "If the virtual terminal is in the presentation layer, where's the power supply?"

I literally fell out of my chair! I couldn't believe the remark!

Although there has been a strong interest in “the upper layers” over the years, as these quotes indicate, there have been differing misconceptions about them. Since the earliest days of the ‘Net, the upper layers have been a bit mysterious and unclear. The early network software removed concerns of reliability and routing from the applications. And with only a couple of dozen hosts connected by what seemed like high-speed lines, one had the illusion of having an environment for distributed computing. All of this sparked the imaginations of those involved to the possibilities, and it continues to do so today. It seemed that rather than there being merely an amorphous collection of applications, there ought to be some general structure for organizing upper-layer functions, as there was for the lower layers.

The lower layers had succumbed to organization much more quickly, at least on the surface. By 1975, it was fairly common to hear people talk about transport, network, data link, and physical layers. It wasn’t until later that things got sufficiently complex that the rough edges of *that* model began to show. Even then, however, those four layers seemed to capture the idea that the lower two layers were media dependent, and the upper (or middle) two were media independent and concerned with resource allocation.

The upper layers (above transport) were a different story. No obvious structuring or decomposition seemed to apply. Part of this was due to both the lack of applications and, in some sense, too many applications. For some time, the network seemed to be able to get by with just Telnet and FTP (mail was originally part of FTP). To find applications on a host, “well-known sockets” were used as a stopgap measure. (There were only three or four applications, so it wasn’t really a big deal). It was recognized early on that there was a need for a directory, and there were some proposals for one (Birrell et al., 1982). But beyond that, application protocols were very much “point solutions.” Protocols were unique to the application. There did not seem to be much commonality from which one could create a general structure that was as effective as the one for the lower layers, or that easily accommodated the variety of applications and, at the same time, provided sufficient advantage to make it worthwhile. Unfortunately, early in the life of the internet, upper-layer development was squelched before the promise could be explored. As we will see, it is this event that most likely contributed the most to the arrested development of the Internet and left it the stunted, unfinished demo we have today.

The OSI work made a stab at the problem and for a while, looked like it was making progress. But although they were able to uncover elements of a general structure, that architecture suffered from early architectural missteps that made application designs cumbersome and from overgeneralization that required complex implementations for even simple applications. But it was mainly the internal divisions that killed OSI.

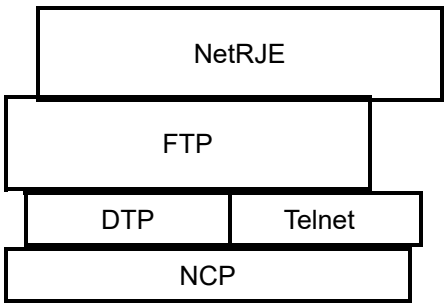
With time, there has been an opportunity to consider what was learned from these experiences and what they can tell us about the nature of the upper layers. Consequently, a much better understanding has surfaced based on a broad experience not only with a variety of applications, but also recognition of similarities between the upper and lower layers. It is now much clearer how the lower layers differ from the upper layers, what the uppers layers do and do not do, how many layers there are, and what goes where. This chapter attempts to bring together these disparate results and put them in a consistent framework. Are all the problems solved? Far from it. But having such a framework will provide a much clearer picture of how

we can go forward, and it will allow new results to be put into a context that makes them more useful. Upper-layer naming and addressing will be considered later; here we are concerned with working out the structure. What we are interested in is understanding how of “networking” relates to “applications” or distributed applications. We need to understand what the difference really is. We are not so much interested in being encyclopedic as considering what we have uncovered about “the upper layers” that is both right and wrong, informative, and something we should learn from.

The Upper Layers of the ARPANET

Once the idea of a packet-switched data network had been proposed and built, it was necessary to show that it was good for something. It is not hard to imagine what happened. ARPA had put considerable money into building the network. It was supposed to be a resource sharing network, so show us some resource sharing! The immediate answer was to implement all the things one did with a computer, only do them over the ‘Net. We had to have remote terminal access, Telnet; file transfer with FTP; and remote job entry, RJE.

The early idea was to mimic operating system functions in a network. Operating systems were often our guide for all of this. Actually, these were pretty good building blocks for other things that might follow. This led to an early upper-layer architecture with some innovations and lessons. Using NCP as the IPC facility, there was Telnet and DTP (the data transfer protocol for FTP); FTP was built on top of those two; and then RJE was built on top of that.



It would be difficult to claim that the members of the early Network Working Group started with an idea of upper-layer architecture. Their focus was on building a network. Applications were primarily there to show that it worked! The group had its hands full with much more basic problems: How do you do anything useful with a widely different set of computer architectures? How do you connect systems to something that they were never intended to be connected to? (You make it look like a tape drive.) How do you get fairly complex network software into systems that are already resource tight? Just implementing the protocols was a major effort for any group. The early ARPANET had much more diversity in the systems connected to it than we see today. Just in the hardware there were systems with all sorts of word lengths (16, 18, 24, 32, 48, 64, etc.) and at least two varieties of 36-bit words. There were at least a dozen different operating systems with widely disparate models for I/O, processes, file systems, protection, and so forth. Today, all these systems

would be classed as “mainframes.” But as one would expect, there were different distinctions then: A few machines were mainframes—number crunchers, i.e., systems to submit big batch computation jobs to. Some were timesharing systems whose users specialized in an area of research, in essence precursors to networked workstations. Some were access systems, dedicated to supporting users but provided no computation services themselves; they were early minicomputers. All of them would now fit in a corner of a cell phone.

But at that time, they seemed to be very different. If there was an architectural direction, it was to provide network access to each of these systems as if you were a local user (or as close as 56 Kbps lines would allow. And 56 Kbps seemed vast when most remote access was 110 or 300 bps!) Furthermore, some of these systems were very tightly constrained on resources. The big number cruncher on the ‘Net was an IBM 360/91, with an outrageous amount of memory: 4MB! (And the operating system was huge! It occupied half of that memory!) The general attitude was that an operating system ought to fit in a system with 16 KB and still have plenty of room to do useful work. The primary purpose of these early applications was to make the hosts on the Internet available for remote use. Thus, the first applications were fairly obvious: terminal access, file transfer, and submitting jobs for execution. But once it worked (and it worked well), our imaginations ran the gamut of what could be done with a resource sharing network. But, from these problems and these initial applications came some very important architectural results in the design of application protocols that still serve us well today, and, as one would expect, there were also a few missteps along the way.

Telnet

The first problem was supporting terminals over the network. A simple basic capability quickly became not so simple. In the early 1970s, the computer industry was in a state of transition moving from batch mainframe systems[1] to timesharing systems, where they have remained pretty much ever since. With mainframes, a user would submit programs (jobs) were submitted on punched cards, the user would then go away and wait for printer output (132 characters wide) from their program. There were many terminals connected to these computers. However, the terminals were generally dedicated to a specific application or could only access a specific application. In some cases, specific terminals would only access specific applications.

The timesharing systems, on the other hand, moved entirely away from a batch model to the use of terminals connected to a single “application” that provided a command-line interface to do everything including running other applications. (The internals of the OS were designed to be more interactive, rather than bulk multiprogramming.) This begins to be something closer to what most readers will be familiar with.

Because this was a period of transition both were still being used and some systems were a bit of both. The existing mainframe OSs (IBM, Honeywell, etc.) often had a timesharing application grafted on to them. (These tended to not work very well and had slow response time for the terminal users.) In the late 60s,

minicomputers had appeared. Often as application specific systems. While timesharing systems were first built by universities, it was Digital Equipment Corporation (DEC) that pioneered this market along with Data General and others. By 1970, minicomputers were not so mini and supported full powerful timesharing, although many early products had problems. All of these systems could have 10s to 100s of terminals connected to them, consisting of a keyboard and some means of displaying the output, either a screen or paper. The terminals were electronic (with a screen) or electro-mechanical devices (with paper) but few if any, had what could be called a CPU.

One might conclude that these terminals were pretty simple. Nothing could be further from the case. Vendors found all sorts of ways to distinguish their terminal products, including the keyboard layout. (Alpha-numerics were the standard QWERTY keyboard layout but there the commonality ended. They often placed the “return” key off the home row and near keys you didn’t want to hit accidentally!) The task of the OS to manage them while ensuring fast response time could be fairly complicated with many special cases and strange behaviors with a strong demand. Most of which were undocumented, which made writing device drivers for them…interesting. Supporting terminals dominated much of the work on OSs, especially trying to maintain good response time. Some systems were quite practical and interacted with the computer line-at-a-time. Thus, maximizing the amount of data to be sent and minimizing the interrupts.[2] Other systems trying to be fancier or more “modern.” interacted character-at-a-time.

With these, when a user typed a character, the character would not be displayed locally, but immediately sent to the computer and echoed from the application or the OS.[3] This of course means that many messages between the application and the terminal were a single character. This generated an interrupt, and single character in response, or once in a while a few characters. With the speed of the processors of the day and the overhead of an interrupt, this could have a major impact on performance, even though computers were much faster than humans typing. Inserting a network and another computer between the terminal and the application only made this problem worse.

Generally, a vendor would offer a line of terminals to a particular computer. One could have multiple types of terminals for each type of computer. The variation in the terminals could quite wide. So much so that an Operating System might have as many as one device drivers for each type of terminal that it had to support. This created the first problem for supporting terminals over a network. This was the problem Telnet had to solve: how to handle all of the types of terminals on all of the different hosts.

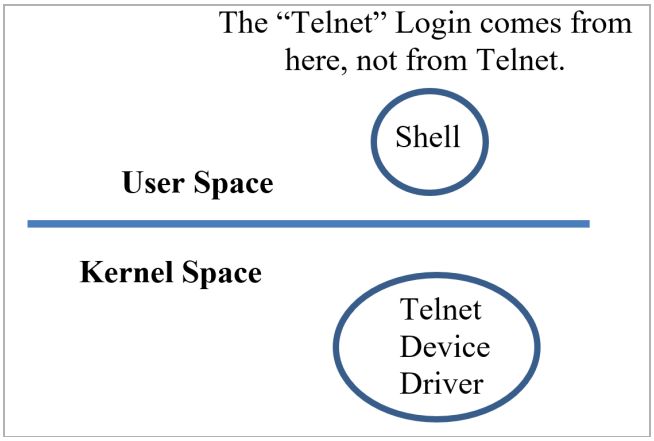
The first version of Telnet was sufficient to demonstrate that the network was usable; but it reflected so many of the terminal characteristics to the user, it was less than a satisfactory solution. The NWG met in late 1972 and drafted the “new” Telnet.[4] The experience with Telnet not being very satisfactory the first time paid off. With the experience gained from the first attempt, the NWG had a better understanding of what a terminal protocol needed to do and the problems Telnet needed to solve and how to do it effectively.

The second attempt at Telnet is based on a very innovative model worked out by Bernie Cosell of BBN (on the flight to the NWG meeting and was shocked when the group immediately saw its elegance and adopted

it). This is a really elegant solution. Everyone else saw remote terminal as dumb terminals talking to a smart host that allowed one to login to a timesharing system, an asymmetrical protocol. A number of experts and textbooks say that Telnet is a remote login protocol. It is not, never was and has nothing to do with remote log in. Cosell's insight was much more subtle and powerful. He reasoned that while this was a protocol between a very limited terminal and a host, it wasn't the terminal that was interpreting the protocol, but the computer the terminal was connected to and taking it one more level, that was the device driver in the host communicating with the device driver in the computer connected to the terminal. If the application was utilizing properties of the terminal, it was doing it *through* the OS's device driver. Hence, Telnet is a terminal device-driver protocol, one level below remote login. And it is symmetrical. It is hard to build services on top of an asymmetrical service, but easy on top of a symmetrical service.

As was just noted the primary goal of the ARPANET was to be heterogeneous resource sharing network and a wide variety of terminals on a wide variety of systems. In 1972 there were at least a dozen different OSs in common use. Each having its own types of terminals. This meant that there were potentially $n \times m$ types of terminals to support. Not good. The Telnet solution to this was to not try to support them but define a new type: the Network Virtual Terminal (NVT). Define a canonical terminal so that each host joining the network only had to support one new terminal type, the NVT (See section 2.1.2 below for more on this.) Telnet was the first virtual terminal protocol.

The Relation of Telnet to the Shell or Command Interpreter



Make it look as if one was a local terminal on the host. The way to do that was to make a connection appear as a terminal to the host OS, i.e., a terminal device driver that created what was called a pseudo-terminal. Making a device driver, it could work equally well with the new timesharing systems where the application was the command interpreter as well as other applications the way the older mainframes worked.

In the early ARPANET, it was used in just that way to support FTP, RJE, and the command interpreter or shell. Each one was given a well-known port. A Telnet connection to that port provided an instance of that application and its login. (Yes, FTP and RJE had logins as well.) Because Telnet was a device driver, it appeared to the command-interpreter application just like any terminal directly connected to the system.

What would happen when one those terminals was turned on or when any key on the keyboard was pressed? It would wake up, the command interpreter application would find out it was active and send a "herald message" (usually a line saying what host and OS version it was, followed by a request to login. When a Telnet connection is made to socket 23, the login message one sees is not generated by Telnet but by the command-interpreter or shell. One could argue that the service on socket 23 should not be called Telnet, but something like Remote Shell.

Why haven't we called it the shell in this explanation? Because in 1972, the only OS to have a shell was Multics.

Today we continue to use timesharing systems. Every major OS in use today was originally a timesharing OS. What major breakthroughs in computing were made to solve the response time problem of handling all of those terminals? Easy, the number of terminals was reduced to 1 and it was called a laptop!

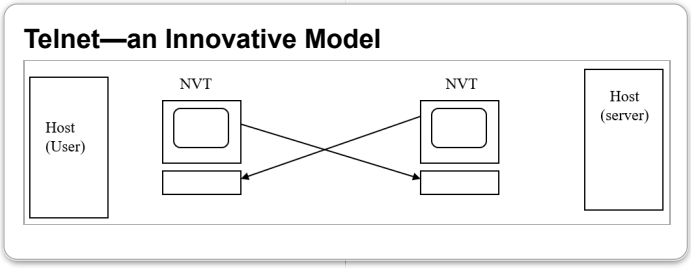
A Symmetrical Design

As we shall see, there were two parts to Bernie's insight: First, that this was a computer network. (Yes, sometimes the obvious is an insight. It is getting out of the box you were in. Most others were still thinking of this as a datacom network of terminals connecting to a host. One may "know" something but not yet be thinking in those terms. It is a bit like what happens with a foreign language when you quit translating in your head and simply understand it.) The ARPANET was a network of peer computers, not a network of terminals connected to a host. (Terminals did not have a processor, and sometimes it's questionable whether they had logic.[5]) The second insight follows from the first: If this was terminal traffic between two computers, then the dialog between the computers was between the terminal device drivers. Making it a device driver meant that it just looked like any other terminal to the host's operating system. It was not a special case. This made an incoming Telnet connection look like any other terminal connected to that computer that had just been powered up or dialed-in locally. The first thing it would receive was the herald message and the login request from the host operating system's command interpreter. The login is part of the normal operating system behavior, not part of the protocol-device driver. Notice this makes Telnet a degenerate case, not a special case. The "new" Telnet protocol had several attributes that are unfortunately still rare in the design of protocols.

The designers of Telnet realized that it was not simply a protocol for connecting hosts to terminals, but it could also be used as a character-oriented IPC mechanism between distributed processes: in essence, middleware. In particular, it could be (and was) used to have one character-oriented program talk to another one without needing any changes, as when, for a major conference demonstration of the ARPANET, the Psychologist Doctor program at Stanford used Telnet to talk to the Paranoid Patient program at MIT!

Another problem that had to be addressed was the incredibly wide variety of terminals (probably more than 50) on the market at the time, running the gamut from electric typewriters printing on paper, to displays that mimicked the paper printers, to fairly complex storage terminals that handled forms and could display text

with bold or reverse video. They were all different: different keyboard layouts, different control characters that did different things, etc. No two were ever the same. It was a mess. This was a potential n by m problem since every site on the 'Net was different, potentially with a different operating system with its own terminals, of which there could be several variations. We could end up with N terminals times m different hosts for an almost n^2 problem, an "m by n problem."



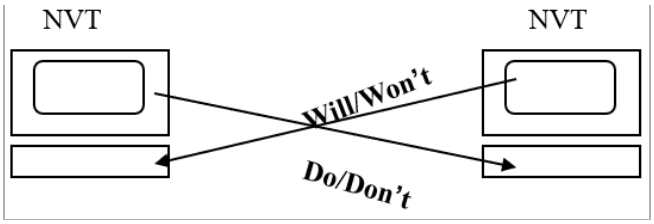
The Network Virtual Terminal

To solve the $n \times m$ problem, Bernie said no, define the Network Virtual Terminal (NVT), a canonical model of two terminals connected back to back, the keyboard of one side connected to the screen of the other and vice versa, making it symmetrical. The NVT did only the simplest and most common of functions, the scroll-mode terminal. After this, options were created for more complex ones. (It is important to note that this was a canonical model, not a "least-common-denominator" model, i.e., there were elements of the NVT that some real terminals might not have and that would have to be simulated.) This solved the "m by n problem," because now when a new site joined the 'Net, there was only one new terminal type to support: the NVT, not n^2 terminals. All that was required was to translate your system's terminal conventions to and from the NVT. It was a very nice, very elegant solution.

The only mistake in this design was making Telnet stream-oriented. As we just saw, there were control characters that could be among the data bytes. There was much more data than there were control sequences. The control sequences were rare. But being stream-oriented required every byte to be tested to see if it was a control character. For processors of the time, this was a lot of overhead. It would have been good to have had an indication in each PDU as to whether it was control or data. The main reason for doing this was that BBN TENEX systems insisted on character-at-a-time echoing over the 'Net. This was not very efficient and often resulted in not very good performance: You could type a line or more before the echoes started to arrive. The limit was only because you would lose track of where you were and would have to wait for it to catch up to see what you had typed.

Symmetrical Negotiation

Telnet: Symmetrical Negotiation



There were many different functions that the terminals did. To handle this, Telnet defined an innovative symmetrical negotiation mechanism that allowed a request by one to be the response to the other. The mechanism was used by the two users to select and enhance the characteristics of the NVT, such as character-echoing, turning off half duplex, message size, line width, tab stops, and so on. The negotiation was structured so that when the connection was established, each side announced what it intended to do or not to do (by sending the commands WILL/WONT followed by the appropriate Telnet options) and what it intended the other side to do or not to do (DO/DONT). The options were encoded such that an implementation that did not understand an option could refuse it without having to "understand" it; that is, just send a WONT x. Each side's announcement became a response to the other side. If there was no conflict, one side would announce DO x, and the other would announce WILL x (the WILL becoming a response to the DO and vice versa). If a conflict occurred, each option defined a scheme for resolution. Telnet is one of the very few symmetrical application protocols. Notice that although the protocol was symmetrical, what was being negotiated was asymmetrical. Different systems did or required different functions. The Telnet negotiation gave them an elegant means to attempt to offload some functions, if possible, and still do what needed to be done.

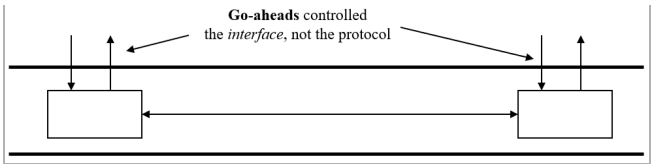
This led to one of my favorite error messages. Earlier we mentioned that there was a version of the IMPs called a TIP (Terminal Interface Processor) that had a very rudimentary command language. In fact, for most of these protocols, *you* were the user protocol machine driving the server. The user actually typed the protocol commands. It was very rudimentary. To help with that and make it easier for the user, BBN implemented a service they called TIPServ that, when a terminal came up, would automatically connect to the TIP service and provide a somewhat more elaborate command-line interpreter to use. But it had the best error message I have ever seen. When you typed something it couldn't do, it said:

WONT.

Not can't. *Won't*. That was it!

Half-Duplex Is an Interface Issue

Telnet: Half/Full Duplex



While of little interest today, the handling of half-duplex terminals shows the subtlety and elegance of Telnet. At the time, terminals that could not send and receive at the same time (that is, half duplex) were still common, in particular IBM terminals such as the 2741, a computer-driven Selectrix typewriter. Consequently, they had to be accommodated, even though most people understood that full-duplex operation was displacing them and was much simpler to handle. The problem was how to support both kinds. Most everyone saw this as an “oil-and-water” problem, two irreconcilable special cases. Either the connection between the host and the terminal could be used in both directions or only one way at a time.

Most protocol designers took the idea of “turning the line around” literally and assumed that the protocol had to be half duplex. However, Bernie recognized that this was a computer network to which terminals were attached. The protocol had to manage only the interface between the device-driver (the protocol) and the remote user as half duplex; the protocol could operate full duplex. Hence, the solution was that Telnet sent indications (a Telnet command called the Go-Ahead) so that the receiver knew when it could “turn the line around” (that is, tell the half-duplex terminal it could send). This allowed Telnet to simply send the indication regardless of whether the other side was full or half duplex, and the receiver either used it or ignored it.

The default case was that a terminal would come up using Go-Ahead. A full-duplex terminal would send a Go-Ahead whenever it wanted. A half-duplex terminal would use the Go-Aheadwhen it needed to tell the remote interface to “Go Ahead.” A full-duplex terminal would come up and start sending Go-Aheads, perhaps with every PDU.

Consider what we know about this:

A Telnet protocol machine (PM) interfacing to a terminal knows that it is likely to get small amounts of data from it. If the terminal is full duplex, it may be receiving one or two characters at a time. If the terminal is half duplex, it will receive several characters all at once, usually a line-at-a-time. On the host side, several characters all at once will be more common for either mode. The host side will send a Go-Ahead with each PDU. If the terminal side Telnet PM is receiving several characters at a time, it will send a Go-Ahead in case the host is half duplex. If it is getting characters a few at a time, it may send a Go-Ahead all the time. If the PM is waiting to hear from its peer and gets a Go-Ahead, it cues its interface to go ahead. If the PM gets a Go-Ahead and it has already done that, it just ignores it.

They might determine early in the dialog that this is a full-duplex terminal talking to a full-duplex host and one side would send a DONT Go-Ahead and the other would respo

Telnet: Half/Full Duplex

-Ahead, which

cross in the mail and acknowledge each other to quit sending Go-Aheads that aren't needed.

Why Telnet Is Important?

Most textbooks no longer cover Telnet (undoubtedly because they deem remote terminal support a thing of the past). This is precisely what is wrong with today's networking textbooks. The reason for covering Telnet is not because it provides remote terminal support, but because it teaches lessons in understanding networking problems.

It is unlikely anyone will see these precise problems again. But the patterns may occur again, and even if not, knowing about them may provide (dare I say?) inspiration to look for similar solutions. This is an opportunity to see how someone took what looked like special cases and found an elegant solution. That may be sufficient for some good designer to look deeper into a problem to find a simple solution. Experience and theory have shown that simple elegant solutions tend to make solving future capabilities simpler. One doesn't find devils in the details but angels.

We consider Telnet because these concepts are important in the education of network designers (the goal of a university education).

Half-duplex was subsumed as a *degenerate* case and did not greatly distort the structure of the protocol (as it did with many others). Take, for example, the OSI session protocol, which made the protocol half-duplex and made full-duplex an extended service. Consequently, the minimal session protocol requires more functionality than one that uses the full-duplex *option*. Half-duplex terminals could use full-duplex hosts and vice versa. Neither really had to be aware of the other, and the application did not have to be aware of which was being used. As the use of half-duplex terminals declined, the use of the Go-Ahead has quietly disappeared.

Full-duplex terminals could talk to half-duplex terminals or systems and vice versa, half-duplex terminals could talk to half-duplex terminals, and full-duplex terminals could talk to full-duplex terminals. They took what in most systems was considered an oil-and-water problem and found a synthesis that made them degenerate cases of the same thing. This is an elegant solution. This is an example of good design. Special cases should be avoided wherever possible.

What Did We Learn from Telnet?

First of all, none of us will ever see precisely these problems again, but we can guarantee that one will see problems reminiscent of these. The experience of the

Telnet design should remind us to consider whether a simplifying design is possible.

However, most proposals for doing what Telnet did, did not take this approach. They saw it as an asymmetric protocol for a very limited terminal talking to a much more capable host. Was that wrong?

No, but it was superficial.

It ignored what was really going on. As we have seen, the designer(s) of Telnet recognized that the terminal wasn't directly talking to the host. There was another OS in between. Telnet talked to it, which managed the terminal. Be careful to do what the problem says. From the beginning of this course, we have seen examples of how "skipping steps" or thinking, "Oh, I know what is going on here. I don't need to look at it in detail." Of course, considering too much detail can also be a mistake.

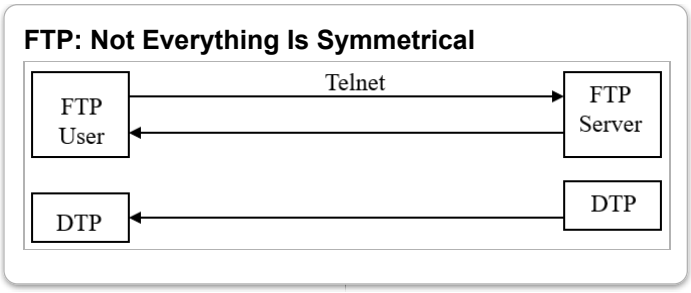
The designers also recognized the advantage of making the solution fit within the structure of the OS and not making it a special case that could later add to the complexity. This made Telnet more than a one-off or point solution. It could easily be used for other things, thus reducing complexity. A solution that fits nicely into the structure is far better than a special case. Avoiding special cases makes a much more robust design.

And in a stroke of brilliance, Bernie Cosell saw that it could be even simpler by making the solution symmetrical and assuming that there were two NVTs connected back-to-back and a very simple negotiation scheme would make it easy to determine who did what.

Being precise about what things are and where they go, we saw how Telnet recognizes that it wasn't the protocol that had to support half-duplex, but local device driver directly controlling the terminal. Again, this reduces complexity and supports a robust system.

File Transfer Protocol (FTP)

Then there was File Transfer Protocol (FTP). There is no way to make FTP symmetrical. A requestor is moving a file from one place to another. FTP was built using Telnet, partly for architectural reasons and partly for short-term practical reasons. On the one hand, there were good design reasons to separate control and data. One didn't want important commands to get delayed behind a large file transfer. On the other hand, the limitations of the TIP required the TIP user to be the FTP user process, i.e., type the protocol commands to be sent. The FTP sent commands by opening a Telnet connection and sending the commands to the server FTP. These commands were used to initiate a data-transfer process to actually effect the file transfer on a separate connection. This had an advantage because file transfers could be slow, and the user needed the ability to query the server for the status of the transfer or even to cancel the transfer.

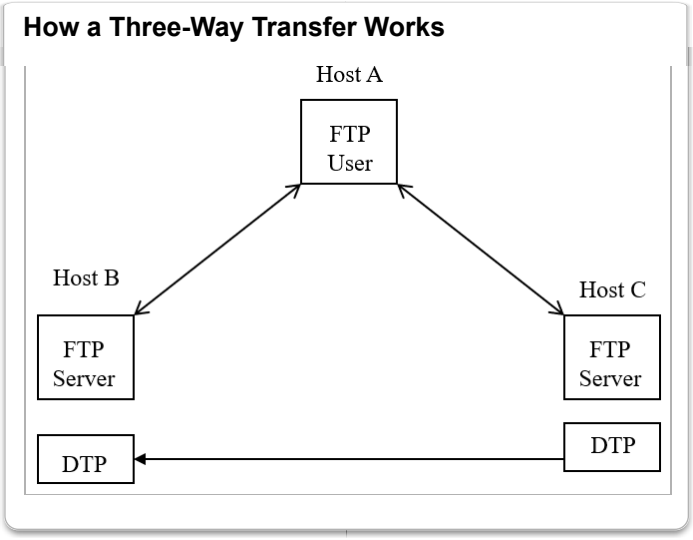


FTP also defined the Network Virtual File System (NVFS), the basic commands used to carry out file transfers and to interrogate a foreign file system. There was such a wide variety in file systems that the NVFS (like the NVT) was restricted to the bare minimum. The main emphasis was on the attributes of the file, while saying as little about the nature of the contents as possible. There are basic conventions for the file format (characters, binary, and so on), the structure of the file (record or stream), and the BYTE command to indicate the word size for a binary file (as noted, earlier computers had a wide range of word sizes). This presented a problem. At an FTP meeting, someone asked, “what happens if I store a file with a BYTE size of 23 and try to retrieve it with a BYTE size of 17? (Both are numbers no reasonable person would use.) Mike Padlipsky, sitting in the back corner of the room, piped up with one his typically sage solutions: “Sometimes when changing apples into oranges, one gets lemons!” And we all agreed that was the right answer. You get what you deserve! The protocol allowed for checkpoint recovery and third-party transfers.

FTP supports checkpoint recovery, now called restart. Checkpoint markers would be inserted into the data stream at regular intervals. When all data up to a checkpoint was written to disk (non-volatile store), the checkpoint marker would be sent back to the FTP user. If there was a failure, then the transfer could be restarted from the last checkpoint received rather than at the beginning.

There were some File Transfer Protocols designed that used a sliding window much like what we will see in Error and Flow Control Protocols. In these FTPs, the transfer would stop if a fixed number of checkpoints had been sent without getting an acknowledgement for the earliest one. This mimics how the sliding window is used in the lower-layer protocols. FTP could be considered an Error and Flow Control Protocol with “greater scope,” i.e., from disk to disk rather than process to process.

Mail starts out as two FTP commands. It is kind of amusing that there was an FTP meeting in 1973. At the end of a two-day FTP meeting, Steve Crocker walked in and said, “guys, you have to put mail in.” In the last half hour, the meeting added two commands for mail. There really wasn’t that much interest in it. FTP was much more important than just doing mail. Of course, it turned out that mail was much more important, probably the most useful thing we ever did.



Another unique capability of FTP was third-party transfers. One could set it up, so that **A** could Telnet to host **B** and to host **C**, then transfer a file from **B** to **C** or **C** to **B**. It would work as follows:

> After telnetting to **B** and to **C** and logging in to both,
> **A** sends a SOCK command to **B** to begin setting up the data connection, and
> **A** sends a SOCK command to **C** to finish setting up the data connection.
> **A** tells **B** to Listen and tells **B** to passive with a PASV command.
> Then **A** tells **C** to RETRieve a file and tells **B** to STORe the file, and
> The transfer begins on the data connection sending the file from **B** to **C**.

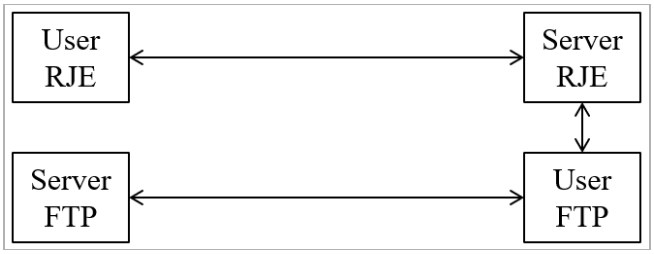
Rather than attempt to impose common file system semantics (which would have greatly increased the amount of effort required and the degree of difficulty in implementing it in the existing operating systems), FTP transparently passes the specifics of the host file system to the FTP user. An intelligent method for encoding responses to FTP commands was developed that would allow a program to do an FTP but at the same time provide the ability to give the human user more specific information that might be beneficial to determining what was wrong.

It is hard to say there is anything wrong with FTP per se. One can always suggest things it doesn't do and could. Over the years, as the commonality of file systems has increased, new commands have been added to FTP. For what it does, it does it about as well as one could expect. None of these really break any new architectural ground. More would have been done at the time if it had not been for the constraints on the TIP (there were complaints at the time that the tail was wagging the dog) and schedule pressures to have something to use. There were small things wrong that illustrate how a temporary kludge can come back to haunt you. For example, as noted previously, FTP was kludged to allow TIPs to transfer directly to a specific socket given by the SOCK command. Later, this became a Best Common Practice (!) and a major problem with NATs. The SOCK command was bad architecture. As we have seen, addresses should not be used outside of their layer. Only application names should be used in the application layer.

Remote Job Entry

The ARPANET developed a Remote Job Entry (RJE) protocol, which went a step too far. In the early 70s, RJE stations were fairly common in business. They were a small station with a terminal for commands, a card reader, and a line printer. (While we were totally off punch cards by the late 1960s, they were not uncommon in business into the 1980s.) The RJE station could be used to submit jobs to a remote host and retrieve output to a printer. The station didn't have a computer exactly, just enough logic to use a phone line to submit card decks to a mainframe and retrieve output to the printer. This was obsolete in the ARPANET by 1980. Terminals and moving files were supported, so moving programs was the next logical step. The idea was to replicate that functionality over the 'Net between hosts as an early example of resource sharing. However, the official network RJE was a step too far.

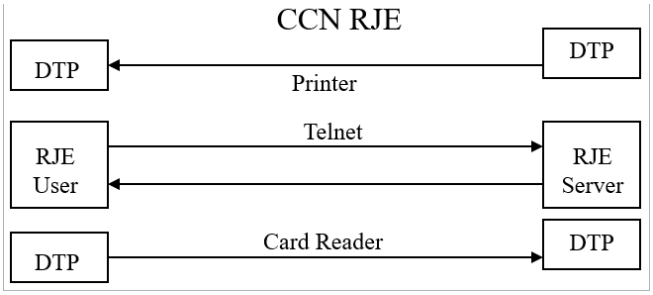
RJE (A Step Too Far)



The design was all perfectly reasonable. However, the designers started paying too much attention to architectural elegance and not enough to the users' pragmatic requirements and constraints. It was not hard to see that the job input and the printer output were files. It was very neat to build FTP on top of Telnet. NETRJE required the ability to send RJE commands and move files. What could be simpler than to use a Telnet connection for RJE commands and then use FTP to move things around? It was simple to describe and easy to implement (if you don't need it). NETRJE put the greatest resource usage where it was least expected to be: the RJE client. The User RJE would log into the Server RJE and tell it what the user wanted to do. The Server RJE would use its User FTP to retrieve the input file from the User's host and then store the output file when it was done.

This was a bad idea. It was elegant. It used the facilities already created, reused modules, and built one application on top of another. All good things. But it lost sight of the original purpose. The user was submitting a job, using RJE because it doesn't have the compute resources to begin with. It may not even have a file system. It could have just a card reader and a line printer. This puts the greatest burden, handling the file system, on the user side rather than on the server side. It is not clear that the official NETRJE was ever used, although it was likely implemented.

RJE (A More Pragmatic Solution)



What did get used was CCNRJE for IBM360/91, the largest machine on the Internet at the time, at UCLA's Campus Computing Network. Their design had User RJE opening a Telnet connection to a Server RJE to submit the job, and then it opened a transfer DTP connection [to?] a fixed offset from the Telnet connection to pull in the input file and a different DTP connection to a different fixed offset to store the output file. This keeps the bulk of the heavyweight computation on the server side, not on the client side.

What Was Learned

First and foremost, the upper-layer development of the ARPANET, as rudimentary as it was, proved that applications could be built and could be useful. Very useful. Technically, we gained valuable experience with distributed systems. We learned the difficulty of dealing with the subtle differences in the semantics that different systems had for what appeared to be very similar concepts. We even found some elegant solutions to some sticky problems that could serve as examples going forward. We learned the necessity of striking a balance between over specifying and keeping the result useful. And we learned not to get carried away with elegance. Our triangulation of Clauswitz and Mao struck home. But we also realized that we had chosen to solve specific problems. Although we had a good start, we did not yet understand the fundamental nature of the upper layers. What structure did a resource sharing network require?

With these accomplishments came much enthusiasm among the people developing and using the early 'Net. In addition to these specific protocols that everyone was using, there were numerous projects that went far beyond these three applications. To conclude that this was the extent of the use of the early 'Net would be grossly inaccurate. Distributed collaboration, hypertext systems, and production distributed database systems were all built during this period.

Rerunning History

Of course, you can't, but it is interesting to conjecture what might have happened had ARPA pursued development of the upper layers. Clearly, the direction USING had laid out to explore new applications should have been pursued. This was a major area, which had hardly been considered; there seemed to be many application protocols that could be useful. Given the capabilities of the systems, that direction most likely would have led toward something like a Novell NetOS

environment. But the advent of the Web raises questions about whether that path would have created or inhibited an environment conducive to the explosive growth of the Web and the immense benefits it has brought. (Although, a less explosive spread might have had some benefits, too, a little less exuberance might have been wise.) It is easy to see how a NetOS environment could lead to self-contained islands. On the other hand, by not pursuing upper-layer development, ARPA removed the driver that would have required richer lower-layer facilities, greater security, finishing naming and addressing, a much earlier impetus to provide QoS, and so on. This might have avoided many of the problems of spam, virus attacks, mobility, scaling, etc. that plague today's Internet). Without new applications as a driver, the only impetus the Internet had was to do what was necessary to stay ahead of growth, which through the remaining 1970s and early 1980s was relatively moderate. And most of that pressure was relieved by Moore's Law. As discussed later, this turns out to be the critical juncture in the development of the Internet that allowed it to remain an unfinished demo for 25 years and is now the crux of our current crisis.

As implementations of Telnet and FTP came into use, people became excited at the possibilities. And it was clear that much more would be needed to make the 'Net a viable network utility. A group of interested parties formed a Users Interest Group (USING) to develop the necessary protocols. The group began to look at a common command language, a network editor, common charging protocols (not for the network but for using the hosts), an enhanced FTP, a graphics protocol, and so on. This group had an initial meeting in mid-1973 to get organized, and a first major meeting in early 1974. However, ARPA became alarmed that this group would essentially wrest control of the network from them and terminated all funding for the work (Hafner et al., 1996). ARPA would have benefited much more by harnessing that energy. Actually, ARPA had already, in a sense, lost control of the direction of the 'Net (short of shutting it down). The wide use of the 'Net might have come much sooner had ARPA encouraged, not squelched, the initiative that users showed.

There were no new application protocols developed in the DARPA effort for the next 20 years! All subsequent developments have addressed specific protocols and have not considered how they relate to each other, what common elements there might be, or how it could be a distributed resource sharing utility as early papers described it. The early ARPANET upper-layer protocols made a greater contribution to our understanding of the design of protocols than to the architecture of the upper layers. But that was to be expected, given that it was a first attempt. There was only so much that could be done. Developing distributed applications was not the primary rationale of the project anyway. Unfortunately, the innovations in these early protocols were often promptly forgotten by future work: the importance of separating control and data, or the fact that terminal protocols could be symmetrical and more useful, was never utilized by any other protocol, and I suspect never realized by their designers. For example, the early OSI VTP was symmetrical because of Telnet, but the U.S. delegates from DEC pushed it into a much more cumbersome asymmetrical design in line with their product.

The development of the ARPANET upper-layer protocols uncovered several interesting aspects of application protocols:

- It was important to separate control and data, at least on different application flows.
- The tendency of application functions to partition vertically not horizontally. In applications, progress tended to consist of PDUs for one phase with different PDUs for the next phase and so on, while in data transfer protocols, different functions tended to be applied serially in a given order on each PDU.
- The limitations of the TIP created less-than-proper conventions that we still live with FTP and SMTP, 4-character commands generally with a single parameter and replies with a 3-digit code followed by text. With the TIP, the human was the user process. This allowed the person to type commands and see their results while also making it easily processed by a program. This limitation disappeared quite quickly but left us with an albatross of character-oriented protocols.[6] They make some sense for SMTP, but less sense for anything else.
- The port command in FTP allowed passing IP addresses between the FTP user and server. Of course, this was a violation not only of layers but levels of abstraction. It would be like passing pointers in a Python program. Again, this was because of the TIP, where peripherals (usually printers) had to be configured to a particular address and port. (It is painful to see kludges we put into FTP to accommodate constraints at the time now touted as “best practice.”)
- And, of course, the biggest issue was well-known sockets, not so much for doing them to start with, as not getting rid of them as soon as possible. As a reminder, this is the network analog of what was done in early operating systems, where applications were invoked by “jump points” in low memory.

But, of course, the major development was the canonical model.

The Concept of a Canonical Model

The use of the canonical form (for example, the NVT) was a major innovation both theoretically and practically. It is the tip of the iceberg to understanding the key elements of the theory of application protocols. This was the so-called “n^2 problem.” Potentially, hosts would have to support $O(n^2)$ translations from each kind of system to every other kind of system. On the other hand, to define a model that was the least common denominator would have been so limiting as to be essentially useless. A middle ground was taken by defining a canonical (abstract) model of the elements that were to be transferred or remotely manipulated (in this case, terminal or file system elements). For Telnet, this was the NVT, and for FTP, an NVFS. The definition of Telnet is strongly tied to the behavior of the NVT. Although the definition of FTP refers to its canonical model less frequently, it is no less strongly tied to the model of a logical file system. The concept was that each system would translate operations from its local terminal or file system in its local representation into the canonical model for transfer over the network while the receiving system would translate the protocol operations from the canonical model into operations on the local representation in its system. This reduced an $O(n^2)$ problem to an $O(n)$ problem. Of course, it also has the advantage that each system only has to implement one transformation from its internal form to the canonical form. It also

has the benefit that new systems with a different architecture don't impact existing systems.

There are two unique characteristics to this approach that differed from other attempts. First, the model was taken to be a composite of the capabilities, not the least common denominator. Although there was no attempt to replicate every capability of the terminal or file systems represented in the network, useful capabilities that either were native capabilities or capabilities that could be reasonably simulated were included. Even with this approach, the wide variation in operating systems made it difficult to define every nuance of each operation to ensure proper translation.

A Rose Is a Rose Is a Rose

This was further complicated by the fact that, at that point, everyone knew their system very well, but knew little of the others, and the systems often used the same terms for very different concepts. This led to considerable confusion and many debates (sometimes heated) for which the ARPANET and the IETF are now famous. This also made writing clear and unambiguous specifications for application protocols difficult. Even when we thought we had written clear specifications (for example, Telnet, 1973), we were often brought up short when a new group would join the 'Net and come up with an entirely different view of what the specification said. OSI tried to solve this with Formal Description Techniques (FDTs), which are daunting to many developers; the Internet, by requiring two implementations, which tends to inhibit innovation.

Translation is the operative word here. Contrary to many approaches, the implementation strategy was not to implement, for example, the NVFS on the host as a distinct subsystem and move files between the local file system and the NVFS (such approaches were tried and found cumbersome and inefficient) but to translate the protocol commands into operations on the local file system and the files from the canonical file format to the local file format. The least-common-denominator approach was avoided by the simple recognition that there did not have to be a 1:1 mapping between operations on the logical object and the local representation, but that the operation in the world of the model might translate into multiple operations in the local environment. In addition, and perhaps most significantly, it was also found that the process of creating the abstract model for the canonical form uncovered new understandings of the concepts involved.

This did expose semantic invariances that translations had to be aware of and not violate. For example, one time BBN loaded ASCII-to-EBCDIC translation tables into the TIPs that couldn't generate the line-delete or character-delete characters for Multics. This is what can happen when mapping between a seven-bit character set (ASCII) and an eight-bit character set (EBCDIC). But more important was an experimental File Access Protocol that would allow moving parts of a file. The thought was that it would be simple to use a byte pointer into the file and simply express transfers as <byte-pointer><length>, until it was realized that the "end of record" indicator for the NVFS was two characters—CRLF—and for Multics, one character—an EBCDIC NL.[7] That made translating into and out of the canonical form much more complicated.

The ARPANET application protocols required the use of a single canonical form. One of the widespread complaints about this approach was requiring like systems to do two translations they didn't need, along with the assumption that it is more likely that like systems would be doing more exchanges with each other than unlike systems. Accommodating this requirement, along with a desire to regularize the use of the canonical form, led directly to the syntax concepts incorporated into the OSI presentation layer. However, by the time OSI began to attack the problem, the problem had changed.

In the early 70s, computers seldom "talked" to each other, and when they began to, they talked only to their own kind. So, when the ARPANET began making different kinds of computers talk to each other, many kinds of computers had to be accommodated. As one would expect, over time the amount of variability decreased; not only were there fewer kinds, but also systems tended to incorporate the canonical model as a subset of their system. Also, new network applications were created that had not existed on systems, so their form became the local form. Consequently, the emphasis shifted from canonical models to specifying syntax.

Does this mean that the canonical model is no longer needed? We can expect other situations to arise where applications are developed either to be vendor specific or perhaps to be industry specific (groups of users in the same industry) in relative isolation but will later find a need to exchange information. The canonical form can be used to solve this problem. Today, for example, the canonical model is used to create the *Management Information Bases* (MIBs) or object models for these applications or for interworking instant messaging models.

On Getting It Right. The ARPANET experience showed that there is some advantage to getting it wrong the first time. The first Telnet protocol was not at all satisfactory, and everyone believed it had to be replaced. But the experience led to a much better design where the conflicting mechanisms were accommodated not by simply putting in both (as standards committees are wont to do) but by creating a synthesis that allowed both to meet their needs without interfering with the capability of the other.

Sometimes, you have to get it very wrong. When FTP was completed in 1973, there was a general feeling that the problem was much better understood and now it would be possible to "get it right." However, FTP wasn't wrong enough, and it never went through a major revision, although some minor revisions added commands to manipulate directories, and so on.

While this is an example of the tried–and-true rule of thumb, that you "often have to throw the first one away," this may also be a consequence of "we build what we measure." Unlike other disciplines where the engineering starts with a scientific basis, we have to "build one" in order to have something to measure so that we can do the science to determine how we should have built it. No wonder we throw the first one away so often! It wasn't a failure. It was just on the path to getting it right. (I know, management doesn't want to hear that.)

With the early termination of research on applications in 1974, the early developments were limited to the bare-minimum applications. With the impetus removed to develop new applications that would push the bounds of the network, new insights were few and far between. Perhaps one of the strongest negative

lessons from this early upper-layers work was that elegance can be carried too far. RJE using FTP and FTP and RJE using Telnet led to an impractical solution.

For now, we will defer our consideration of applications in the Internet (more or less keeping to the chronology) and instead shift our attention to OSI to see what it learned about the upper layers. We will then return to the Internet to pick up later developments there and see what all of this tells us about the fundamental structure of networks.

Speeding Up Standards

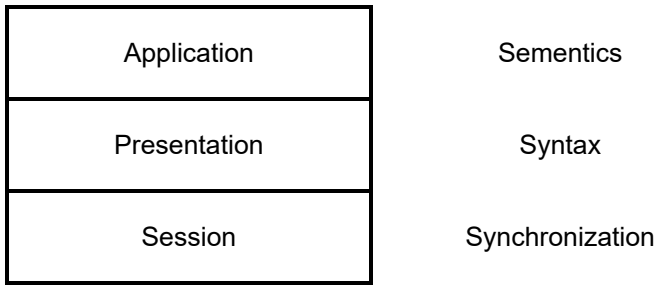
OSI was intent on getting things done quickly! I know, many will find this hard to believe, but it is the case. There has been a lot of talk about speeding up the standards process. From years of participating and watching the standards process, it is clear that the vast majority of the time is consumed by people becoming familiar and comfortable with unfamiliar ideas. There is little, if anything, that can speed up such a process. Building consensus simply takes time, and the greater the diversity of interests and the more people involved, the longer it takes. Rushing a consensus or trying to force a particular answer usually destroys the result (for example, SNMPv2). There are no shortcuts. When IEEE 802 started, they were going to produce Ethernet standards in six months; it took three years. The IETF is a classic example. When it was reasonably small and its population fairly homogeneous, it developed a reputation for doing things quickly. But did they? It took nine years to develop the final version of TCP, while it only took OSI five years to do the same with TP4. OSI had a far more diverse participation, but the Internet took longer than any other standards group has ever taken. (For example, IPv6 has taken 20+ years for a small change.) The IETF's participation is still less diverse than OSI's was. It appears that other factors are contributing to the lengthy development time.

---

1. Mainframes were considered the "big machines" of the time although by today's standards there are stuffed toys with more compute power.
2. It should be noted here that this is when systems were also transitioning from polled to interrupt-driven. Polled had been simple for batch systems and much of the response time issues on existing systems to which timesharing had been added was because the hardware polled. The more terminals, the longer it took to poll them in between the higher-priority of running programs.
3. Fancy as in when the user had typed a few characters that disambiguated the command, the command interpreter would echo the entire command name.
4. At this point, calling it "new Telnet" is a bit like calling the Pont Neuf the New Bridge.
5. This is one of the great incongruities of design: On the one hand, the designer must not have been that smart or the design would be simpler. On the other hand, they couldn't be that dumb or they would never have gotten it to work!
6. The only rationale I have ever been given for why character-oriented protocols have an advantage is that they are easier to debug, which implies more time will be spent debugging them than using them.
7. CRLF = Carriage Return, Line Feed; NL = New Line.

The Upper Layers in OSI

By the late 1970s, upper-layer development shifts to OSI. After consideration of the upper layers in the ARPANET was halted, the only major investigation was going to occur in the OSI debates. Beginning in 1978, OSI was the first of the standards groups intent on getting something out quickly, and it was the first to learn that, with a widely diverse set of interests, it could be difficult, if not impossible, to achieve agreement.



At the first meeting in March 1978, the group adopted an architecture developed by Charles Bachman, then of Honeywell Bull, that had seven layers. At that time, the characteristics of the protocols for the lower four layers were well established. Although there was uncertainty about what went into the Session, Presentation, and Application, the “official” reason for adopting the seven layers in the Honeywell model was that it seemed to make a reasonable working model and had enough flexibility to accommodate whatever might be discovered. *And unofficially*, it wasn’t the same number of layers as IBM’s SNA. The pressure of IBM’s market share was always strong, although it have some disturbing asymmetries built into it. It was clear there would be many application protocols. But for the time being, the terminal, file, mail, and RJE protocols formed the basis of the work. What parts of these protocols, if any, went into the session and presentation layers? Or did they all belong in the application layer? And, did other functions belong in session and presentation? In the official collaboration with the PTTs, OSI became an even more highly charged political environment, now a three-way debate among IBM, the other computer companies, and the phone companies. OSI had no new applications beyond more elaborate versions of the common three but then neither did anyone else. (Keep in mind that for the vast majority of the OSI participants and the commercial world in general, they had no idea what was going on in the DARPA world.) To some extent, this was because the 80s were a period of technical consolidation as the newness of the ideas and the hardware limitations conspired so that no new applications were proposed anywhere. For approximately the next three years, considerable debate continued, attempting to work out the upper-layer architecture. The upper layers were pretty much a clean slate. The phone companies quickly stole the session layer for Videotex.

Deft Political Maneuver

As we have discussed, CYCLADES worked out the lower four layers. The CYCLADES Transport Protocol was done by Hubert Zimmermann of CYCLADES and Michel Elie of Honeywell Bull in France. Charlie Bachman worked for Honeywell-Bull in the United States. The CYCLADES work got incorporated into the Honeywell-Bull Distributed Systems Architecture, to which Charlie added the upper three layers. When OSI was created, the United States held the Secretariat and therefore appointed the chair, which was Bachman. Bachman's status as a Turing Award honoree and his reputation as a fair moderator gave him great influence in those early meetings. The French knew that if they introduced a proposed model, it would be one of many along with those from other countries. But if the Americans introduced a model, especially by the Chair, it would be more than likely that others would accept what the Americans proposed. So rather than pushing something of theirs, the French supported the American model, which was actually their model with the addition of the upper three layers. It was already determined that the French would chair the OSI Architecture Working Group, which gave added support to the U.S. proposal. Zimmermann was to be the Chair. This was a very deft bit of political gamesmanship by which the French got their proposal adopted without it being their proposal.

This was essentially confirmed by 1983 when we found that the upper three layers were really one layer. Zimmermann and I were on our way to the meeting where the protocols for the upper three layers were to be "adjusted" so that they could be implemented as a single layer. Zim muttered to me, "Those aren't really layers. There's no multiplexing in them." I thought, this makes a lot of sense. He knows more about the nature of layers than we do. He has been further down this path than we have. But I am getting ahead of myself.

The Theft of the Session Layer

As with everything else in OSI, there was no consensus on the upper layers. The disagreements fell along the same battle lines as in the lower layers: the PTTs versus the computer industry. The issues were different but the sides were the same. The European PTTs had two point-products under development that they wanted operating under the OSI name. It didn't matter to them whether accommodating them left a path open for future applications. They were a monopoly. If they built it, customers had to buy it. The computer industry, on the other hand, realized that the upper layers had to lay a foundation for everything to come. So as the computer industry faction began to try to make sense of the upper layers, the European PTTs inserted themselves into defining the Session Layer. They had been developing protocols for two new services to run over X.25: Teletex and Videotex. Teletex was billed as email. It was actually Telex with some memory and rudimentary editing capability, a far cry from the email protocols that had been ubiquitous in the ARPANET and other research networks for almost a decade.[8] Videotex was more sophisticated: a terminal-based information system with a rudimentary graphics capability. Although hypertext had been around for over ten years at that time, it was not widely available. Videotex was targeted at what could be done with the

technology of the early 1980s. The best example of Videotex was the French Minitel system. The intent of the system was to replace the phone book and Yellow Pages and allow people to make hotel and train reservations, etc. as described earlier.

If We Build It, They Must Come

This is the fundamental market strategy of a PTT. But in this case, they didn't. People will not pay to get advertising. The primary, and perhaps only, successful example is the French Minitel. France Telecom justified giving away the terminals with phone service with the argument that between the costs saved in printing and distributing phone directories annually and the revenue from advertisers and information services, such as booking airlines or trains, it more than covered the cost. As with the Web, pornography was the first to make money from Videotex. However, giving away the terminals would make it hard to justify upgrading the hardware. The French PTT failed to see that technology would change much faster than phone companies were accustomed to. Still, the design was resurrected in the 1990s and called WAP (Wireless Access Protocol), which only lasted a few years before it was overtaken by technology yet again.

Jumping on the OSI bandwagon of the early 1980s, the PTTs wanted the Teletex and Videotex protocols (which were already designed and being built) to be OSI. OSI was just beginning to determine what the upper layers should be. The PTTs basically laid out their protocols and drew lines at various places: This small sliver is the Transport Layer; here is the Session Layer; there is really no Presentation Layer; and the rest is the Application Layer. These were intended to run over X.25. Throughout the late 1970s and early 1980s, the PTTs argued in every forum they could find that transport protocols were unnecessary. However, OSI was coming down hard on the side of the debate that said X.25 was not end-to-end reliable and a transport protocol was necessary. The PTTs insisted X.25 was reliable. So, Class 0 Transport was proposed by the PTTs so that they would have a transport layer that didn't do anything. And then, when they got to the application layer and thought no one was looking, they stuck in RTSE (Reliable Transfer Service Element) to provide end-to-end reliability and said they were doing checkpointing for mail. However, the parameters for the protocol were in the range used for a transport protocol, rather than the range checkpoint recovery. It was really quite amusing how many pundits and experts bought the argument. (Another game seen often.)

The functions proposed for the Session Layer that fell out of this exercise were various dialog control and synchronization primitives. There was a strong debate against this. Most people had assumed that the Session Layer would establish sessions and have something to do with login, security, and associated functions. This came soon after the competing efforts CCITT effort were proposed to merge into a joint ISO/CCITT project. There was considerable pressure to demonstrate cooperation with the PTTs, even if it was wrong. So, the European block voted for the PTT proposal. Interestingly, AT&T did not side with the other PTTs and argued that the functions being proposed for the Session Layer did not belong there but belonged higher up. As we will see AT&T was correct. (This merger is what really killed OSI.)

There are a lot of 80s mainframe ideas here. What were these upper layers?

To support Videotex, the session layer was supposed to be a toolkit of primitives for structuring the dialogue between the user and the server applications. The primitives, called Session Functional Units, were:

- Major/Minor Synch – This puts synch marks in the data stream and may be coordinated with other flows. Major Synch stops sending data, synchs, and then resumes. Minor synch does not stop the data flow.
- Resynch – This is used to back the flow up to a previous synch mark.
- Activity Management – This is a set of primitives for “multiplexing” application dialogs (see below).
- Half/Full Duplex – This is exactly what it says it is. (Clearly, the PTTs were not as smart as the ARPANET solution.)

The only one of these that requires more explanation is Activity. Suppose two parts of an application are using the same connection: One is doing a long file transfer, and the other is doing a quick status request. Rather than make the status request wait until the file transfer completes, an Activity-Interrupt is sent, halting the current Activity; then an Activity-Start is sent, followed by the status-request and whatever is related to it; and then an Activity-Resume is sent to continue the file transfer. There is also an Activity-Discard to roll back whatever was done.

These are functions used by the Application, not by the Presesntation Layer. So, shouldn’t they be closer to the Application? One would think so. The excuse given at the time was that they were ‘pass-through’ functions. That will only work if they don’t interact with what the Presentation Layer does and for the simple applications the PTTs were doing that wasn’t a problem…yet.

As the reader can see, the OSI session layer really has nothing to do with creating sessions—something that took many textbook authors a long time to figure out. It doesn't create sessions. Sessions were really established at the application layer. (This is going to cause trouble later. They are not doing what the problem says. You can pay the piper now or you can pay him a lot more later.)

Sorting Out the Upper Layers

Meanwhile, the Upper-Layer Architecture (ULA) group continued to try to sort out what the upper layers were all about. It was fairly stoic about what had happened with the Session Layer. Most believed that even if the functions were not in the *best* place, they were close enough for engineering purposes (an argument we hear often in the IETF these days). The Session functions were needed, and when Presentation and Application were better understood, a way could be found to make the outcome of the Session Layer less egregious. (As it would turn out, the problems it caused were too fundamental, but this does serve a valuable lesson about compromising technical veracity to commercial and political interests.)

Early on (and almost jokingly), it had been noted that in the applications (or in the application layer), PDUs

would have no user data. They would have all PCI, all header. In other words, this was where the buck stopped. There was nowhere else to forward user data. Or more formally, the PDUs contained only information (that is understood by the protocol interpreting the PDU) and had no user data (that which is not understood by the protocol interpreting the PDU) for a higher layer. We will also find that this distinction still matters even in the application.

Because the upper-layer problem was less constrained than the lower layers and clearly part of the more esoteric world of distributed computing, a more formal theoretical framework was necessary to understand the relations among the elements of the application layer. For this, they borrowed the idea of conceptual schema from the database world.

Database Schemas

In the database world, the conceptual schema was the semantic definition of the information, which might be represented by different structures or syntaxes for storage (called the internal schema) and for presentation to a user or program (called the external schema).

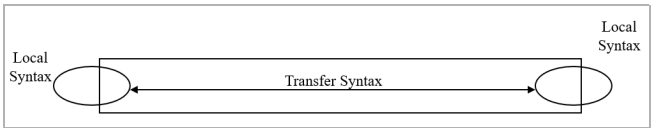
From this, it was clear that for two applications to exchange information, the applications needed “a shared conceptual schema in their universe of discourse.” If the other person doesn’t have a concept of a “chair,” it is impossible to talk about “chairs” regardless of what language is used. As it turned out, this was a generalization of the concept of the canonical form developed for the ARPANET protocols, but now with greater formalism. The concept comes directly from Wittgenstein’s *Tractatus*. The conceptual schema defines the invariant semantics that must be maintained when translating between systems with different local schemas.

Clearly, if the applications had shared conceptual schemas (that is, semantics), the application layer must provide the functions to support the manage- ment and manipulation of these semantics. And wherever there are semantics, there must be syntax. So, if the application layer handles the semantics, then clearly the presentation layer must handle the syntax! Now they seemed to be getting somewhere; maybe there was something to these upper three layers! On the other hand, one cannot manipulate semantics without the syntax. Consequently, one finds that the Presentation Layer can only negotiate the syntax used by the application. Any actual syntax conversions must be done by the application.

So, the common part of the Application Layer provides addressing and negotiates the semantics to be used, which were identified by the Application Context, and the Presentation Layer negotiates the syntax identified by the Presentation Context.

The Presentation Layer

The Presentation Layer (Negotiates the Transfer Syntax)



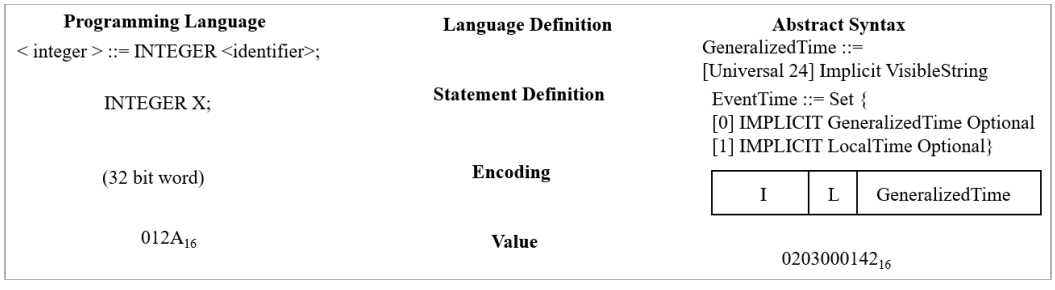
As we just saw, the Presentation Layer negotiates the transfer syntax. This solves the complaint from the ARPANET of having a single canonical form, but goes much further and makes application protocols invariant with respect to syntax. Contrary to what some textbooks say, the presentation layer does not do encryption or compression and never did.

The Presentation Layer provides the negotiation of the abstract syntax and the particular encoding rules. The translation is actually done in the Application Layer, so this isn't much of a layer. The Presentation Layer negotiates what the bits look like that the application is exchanging. As noted above, this satisfies the requirement that the ARPANET had run into that led to using the canonical form but was extra overhead transferring between like systems: The protocol is defined in terms of the abstract syntax, then negotiates encoding rules that are the same as the internal local syntax between like machines.

The primary purpose of the Presentation Layer is to negotiate the syntax to be used on the connection. A syntax language was defined as an abstract syntax that could have multiple encoding rules. The abstract syntax of a protocol refers to the data structure definitions specifying the syntax of the PDUs in a particular syntax language, whereas the encoding rules refer to a particular bit representation to be generated by that abstract language and sent on the connection.

This may sound a little esoteric, but it really isn't. The abstract syntax is like writing down data structure definitions in a programming language. This is then compiled like any other program. As with any compiler, one can have different code generators. In this case, the "code generators" or encoding rules define the format for the bits that are actually generated.

Abstract and Concrete Syntax



The Presentation Protocol provided the means to negotiate the abstract and encoding rules used by an application. Note that Presentation only *negotiates* the syntax. It does not do the translation between local and transfer syntax (what the ARPANET called the *local and canonical form*). OSI realized that such a clean

separation between syntax and semantics is not possible. The translation must ensure that the semantics of the canonical model are preserved in the translation. This was referred to as the *Presentation Context*. Because an application defined its PDU formats in the abstract syntax, an application could use any new encoding rules just by negotiating it during connection establishment. OSI defined *Abstract Syntax Notation 1* (ASN.1) as the first (and it would appear to be the last) example of such a language. Then it defined the *Basic Encoding Rules* (BER) (ISO 8825-1, 1990) as a concrete encoding. BER was a fully specified encoding of the (type, length, value) form and allowed the flexibility for very general encodings. BER proved to be inefficient in its use of capacity and a little too flexible for some applications. With impetus from the International Civil Aviation Organization (ICAO), OSI embarked on two other sets of encoding rules: one to be bandwidth efficient, Packed Encoding Rules (PER), and one to be more processing efficient, Light-Weight Encoding Rules (LWER). PER was done first and achieved both goals. It was 40% to 60% more capacity efficient, but surprisingly was roughly 70% more efficient in processing. No need was seen in pursuing LWER, so it was abandoned. Experimentation has indicated that PER is about as good in encoding efficiency as is likely, indicated by the fact that data compression has little or no effect on it.

Today, ASN.1 is in wide use. There are 8–10 or more different encoding rules for the same abstract syntax, including for XML, UML, JSON, etc. The real benefit of this is that it makes a protocol specification that is invariant with respect to syntax. We will use this later in the course.

The Application Layer

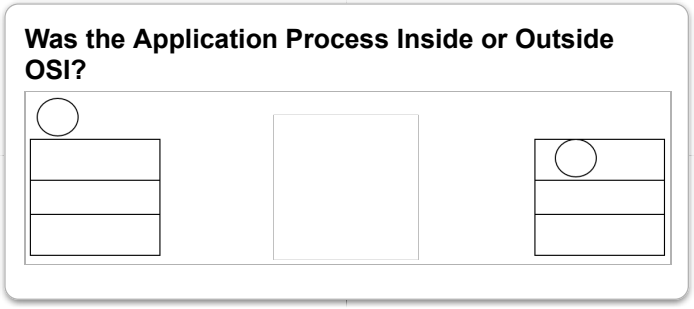
The OSI Application Protocols

Initially, OSI embarked on developing a number of application protocols, some of which have been alluded to here: OSI variations on Telnet, FTP, and RJE (Job Transfer and Manipulation, JTM). File Transfer Access Method (FTAM) and JTM were based on British proposals, and Virtual Transfer Protocol (VTP) was based on a combination of proposals by DEC and European researchers. While focused groups concentrated on the core of each of these, there was a bigger issue to be solved: What was the structure of the Application Layer? It turned out to consist of two parts: a theoretical issue that crossed the line into politics, specifically, determining the bounds of the OSI Model and the nature of the top of the Application Layer, and finding a rational structure for the application protocols that encouraged good design.

The Bounds of the Application Layer and the OSI Model

A problem arose when other standards committees that developed application standards came to OSI and wanted answers to questions Where is the top of the model? What is the nature of the top of the model? Is the application in or out of the OSI-Environment? These were major theoretical questions, that were both technical and political.

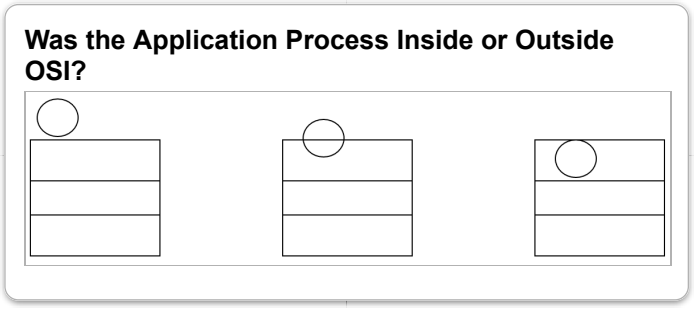
In the early ARPANET, this question did not come up. There was only one Telnet, only one FTP, only one RJE per system, etc. They were all part of the operating system. They weren't domain-specific applications. There was no need for the distinction. But when we started talking about other applications, like banking, airlines, and everything else, it certainly did. Around 1985, the question came up as to where were the application standards that dealt with those domains. Were they inside or outside the OSI environment?



This was fundamentally a political turf question. It was asking the question, is a banking application protocol inside or outside the OSI-Environment?

If the application is inside the OSI-Environment, then the banking standards committee is part of the OSI committee. They don't like that answer (and frankly, neither did OSI). If the application is outside of the OSI-Environment, OSI still has to say something about the application, but what? OSI was developing capabilities that were not specific to a domain. This was fundamentally a question about the nature of distributed applications. This was debated in the ULA group for close to 18 months. As Rapporteur of the OSI Model, I would check in every so often to see how the discussion was going. It was a very abstract discussion, often very esoteric, on the nature of distributed applications. After all, they were trying to predict how sophisticated distributed applications could get. Often it sounded like they were worried about how many angels could dance on the head of a pin. It was getting so esoteric, it made my head hurt. (If someone ever wants to delve into this, it may be hard to find because the only name for this topic was obscene. I know, I know. But sometimes a phrase comes up and is used informally in conversation, but when it comes time to write something down, everything else is long and complicated, while the informal phrase is very appropriate. There was nothing else. Because of the drawing, it was called the Balls In/Balls Out Question. So BIBO was all we could put in the official minutes.)

They finally came up with the answer! And the answer was: The Balls are on the line! I thought it was another Standards Committee wimp out! Good grief! Couldn't we have a real definitive answer to this?



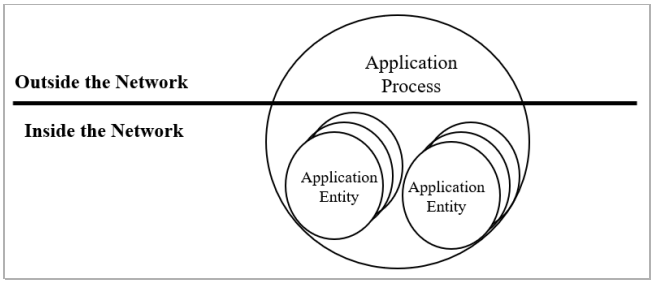
It turns out, this was exactly the right answer! It wasn't until the Web a few years later that we actually had an example of what was going on here. This is a good example of where the theory was actually worked out before we had a real-world example of what was going on. Not only that, but this was the only time I have ever seen a major technical insight come out of a political turf battle. *That alone should make this interesting!*

What was this telling us that was so important? The result said that the application process has two parts: the Application-Entity (AE) that is in the OSI-Environment and the rest of the Application Process (AP), which is outside the OSI-Environment. (For our purposes, replace OSI with IPC.)

That Which Must Not Be Named

Why is there no name for this “other part,” the part of the AP not in any AE? Good question. On the one hand, some member bodies insisted that we not describe or name anything outside the OSIE. This was the turf of other committees, and we shouldn’t say anything about it. On the other hand, to not do so was just asking for trouble. To draw the distinction too closely would have some pedants insisting on some formal boundary, whereas good implementation would probably dictate a less-distinct boundary between the AE aspects and the AP aspects or even interactions of AEs. Later we will see that this was well founded.

Applications and Communication: II



Let’s suppose that I have a hotel reservation application. It uses HTTP to talk to the user to make a reservation. It also uses a remote database protocol to record the updates to the reservations database. These are both application-entities. The application-entity is the part of the application concerned with communication. Neither of these protocols have anything to do with hotel reservations. The “hotel-ness” of the application is what the rest of the application process is concerned with—the reason for the application in the first place. This is the thing that makes it specific as a particular problem.

An application process can have multiple AEs that are different application protocols. The application entity is inside the IPC environment; the application itself is outside. Why isn't this just another layer? If it were then there would have to be something in the AP to interface to the entity in the layer below and so on. There would be an infinite regress. Hence the distinction between the application-entity and the application

process is the correct relation.

This turns out to be a very important result.

The Structure of Application Protocols

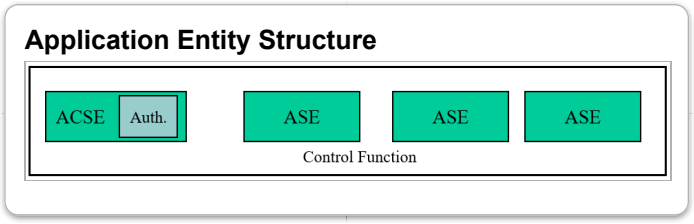
There were a couple of goals for the application layer that the ULA group believed were desirable: to avoid the well-known port kludge and to make the development of application protocols modular and allow for re-usable components. To avoid the well-known port kludge required two aspects: application-layer naming and a common connection-establishment mechanism to be used for all applications.

Believe it or not, we have already seen the application-naming structure. It follows from the application process/application entity result. Clearly, application-process names were global in scope. Because more than one copy of an application process could be executing at the same time, application-process-instance-ids unambiguous within the application process are required. Application entity names were unambiguous with the scope of the application process, and AE-instance-ids were as well.

Without such a common application-connection mechanism, a host would have to be able to interpret all the initial PDUs of all the applications it supported to determine what application to deliver the PDU to. Clearly not practical. A common connection-establishment procedure fit well with the modularity concept.

Hence, OSI specified the common connection-establishment mechanism, *Association Control Service Element* (ACSE), to provide mechanisms for application layer addressing, application context negotiation, and authentication. Thus, the applications defined by OSI (CCR, TP, file transfer, and so on) only define the behavior of the data transfer phase. ACSE provides the establishment phase. Of course, the first thing that should be done any time an application connection is created is to authenticate who it is talking to. There was a plug-in in ACSE to do authentication. This is just the kind of structure one wants to see for developing application protocols. The Internet has yet to explore any of this.

This modularity concept isn't so much new as it is just good software engineering. The OSI approach to application protocols allowed applications to be constructed from reusable modules called Application Service Elements (ASEs). It was clear that some aspects of application protocols could be reused (for example, a checkpoint-recovery mechanism or a two-phase commit scheme).



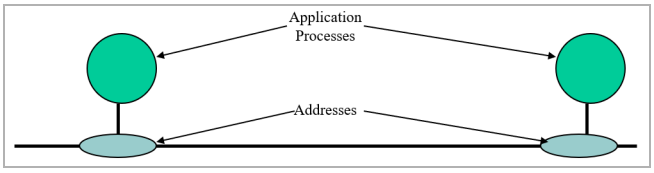
Clearly, some part of the application protocol would be specific to the task at hand. As we have seen, a common ASE was required by all applications to create the application connection, ACSE. There was also a

connectionless form of ACSE called A-unit-data that complemented the unit-data standards in the other layers. In OSI, an application protocol consisted of ACSE and one or more ASEs bound together by a Control Function (CF), which moderated the interactions among the ASEs. Because ASEs had inputs and outputs like any other protocol state machine, the CF was just a state machine that controlled the sequencing of their interactions among the ASEs. There is nothing unique about this approach. But it is unfortunate that this approach was not pursued; it might have facilitated uncovering some interesting structures in application protocols.

For example, one could take the file transfer protocol ASE, combine it with a checkpoint-recovery protocol, and have a file transfer application that does checkpoint recovery. But then one could take that checkpoint-recovery module and use it with a transaction-processing module as a one-phase commit, and so on.

ACSE Exposes a Flaw

Why ACSE was Necessary (a Flaw in the Architecture)



This exposed a flaw in the architecture: why was ACSE required? In OSI, and, as a matter of fact, the Internet as well, addresses were exposed at the layer boundary and, to avoid the well-known port kludge, ACSE was needed to carry the name of the application.

Why is this a flaw? Go back to the IPC model—it doesn't expose addresses to the application. Can you find the memory address of a variable in a Java program? No. The Internet has the same problem. It is trying to use addresses where they don’t belong. We still need something like ACSE, but not for the reason OSI needed it. The source and destination application-naming information must be communicated to facilitate the authentication of the connection.

Why is this called Association Control, rather than Connection Control? It is another terminology subterfuge forced by the traditionalists. A connection should be the shared state between the protocol state machines or, in OSI parlance, (N)-entities. But very early on, they had defined an (N)-connection as “the relation between (N+1)-entities” (!) and exposed addresses at the layer boundary. We knew it was wrong but didn’t have the arguments for why until later. This ended up causing several convoluted workarounds in the model. At the Application Layer, there was no (N+1)-entity. Consequently, a term was needed for the relation between (N)-entities, not (N+1). The term “*association*” was adopted.

The OSI Application Protocols

As we noted at the beginning of this section, OSI started out defining versions of the same three application protocols that the ARPANET did: a Virtual Terminal Protocol (VTP) that was based on a traditionalist proposal by DEC; File Transfer Access and Management (FTAM); and Job Transfer and Manipulation (JTM) based on a British proposal. Each had more functionality (to be expected since they came ten years later), but none showed any new architectural innovations, including those incorporated in the early ARPANET/Internet protocols, nor were they designed so that initial implementations were inexpensive. The minimal implementation was always fairly large.

Later, OSI developed a number of important areas that would have been too early for the ARPANET, such as Commitment, Concurrency, and Recovery (CCR), a two-phase commit protocol intended as a reusable component; Transaction Processing (TP), which made apparent the limitations of the upper-layer architecture structure; Remote Database Access (RDA); Remote Procedure Call (RPC), how to standardize 1 bit; a directory facility, X.500; and management protocol, Common Management Information Protocol (CMIP), which is discussed later.

CCITT (ITU-T) took the lead in developing an email protocol, X.400. The original ARPANET email had been part of FTP and was later extracted into SMTP with enhancements. Mail was directly exchanged between computers (most were timesharing systems). By the time of the discussions leading up to X.400, most systems were workstations or PCs. Hence, this led to different roles with the concepts of message transfer agents and user agents. Initially, X.400 used only the session layer, but as the Presentation Layer was incorporated, unforeseen problems arose. (More on that later.) Detailed descriptions of these exist elsewhere. Let us turn our attention to the major new insights made by the Upper Layer Architecture group.

Summary of the OSI Upper-Layer Results

The upper-layer architecture work had made some interesting progress in furthering our understanding of distributed applications, such as:

- The capability to negotiate syntax of the application.
- The distinction between abstract syntax and encoding rules, which enables making protocols invariant with respect to syntax.
- A common mechanism for establishing application connections and introducing authentication.
- That application protocols should be composed of components, some of which would be reusable.
- Determining the nature of the boundary at the top of the architecture is very important. In fact, just understanding the distinction between the AE and the AP is important.

Often, developing standards is likened to “herding cats.” This is especially true when some or all of the participants already have plans for products. As we have seen, this was especially true with OSI, where the computer companies were aiming at distributed applications in the future, while the phone companies and IBM were intent on codifying products of the past. One of the problems with doing standards is that many participants only have experience with point products, not creating a structure for a whole new environment

of products. Consequently, functionality that has short-term use is included, ignoring the structure implied by the problem. As we said above, ignoring what the problem says means taking a chance. One can pay the piper now or ignore him and pay a lot more later when he can't be ignored. To add to the insult, the short-term solution that ignores the problem is generally obsolete by the time the standards are approved. The reason the ULA group was able to make some progress was that the PTTs and IBM were not terribly interested in the future. By the early 80s, the piper's bill was coming due.

Problems Arise in the Upper Layers

It didn't take long for problems to begin to arise. Before OSI, no one had dug deeply into the nature of the upper layers. There were the three applications and some others being proposed. But the sense was that there should be a general structure. Bachman's seven-layer model had seemed reasonable and met the initial political requirements. There was general consensus that there should be something called a Session Layer, but it wasn't what ITU had turned it into. Bachman had understood that Presentation was for reconciling differences in format. There was an asymmetry here that was troubling. As this picture of the upper layers came together in the early 1980s, cracks began to appear in the structure, and as the decade wore on and the applications became more ambitious, the cracks became more severe.

Were There Really Seven Layers, or Only Five?

By 1983, it had become clear that each Session and Presentation connection supported a single application connection. There was no multiplexing above transport and no need for addressing in the Session and Presentation Layers. Consequently, there was no need for the session and presentation layers to set up connections serially. That would incur considerable overhead if done that way. To some extent, OSI was even further down the path of having too many layers that caused too much overhead, which had undermined the early ARPANET's attempt. But there was a way out.

A Note on Committee Realities

Some readers will think the OSI people were pretty stupid for not seeing these problems coming. They did. Or at least some did. However, if you have ever participated in a standards committee, you know that taking the "right" technical approach seldom counts for much. What does count are implications for products and market position. At the time the structure was being set in the early 1980s, these objections were theoretical, of the form "this could happen." It was not possible to point to specific protocols where it happened. This, of course, allowed so-called pragmatists, those with a vested interest, and the just plain dull to counter these arguments as just speculative prattling. (Some participants were so "forward looking" that even in the late 1980s, there were some who claimed that the amount of data traffic would never exceed the amount of voice traffic!) And there were

arguments about whether we should bother going to the trouble of doing it right if we don't know it will really be needed. (Sound familiar?) It was not until the late 1980s that work had progressed sufficiently that there was hard evidence. This is often the case in all efforts, in standards and elsewhere. One does not have the hard data to back up real concerns. The demand to accommodate immediate needs, regardless of how bad a position it leaves for the future, is often overwhelming and much more damaging. Basically, don't violate rules of good design. It might not be clear what will benefit from it, but one can be assured that something will.

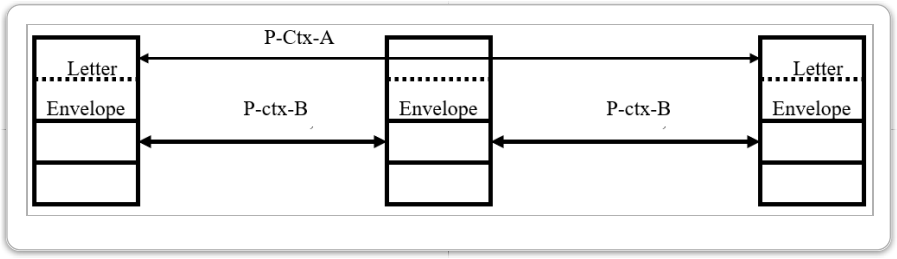
It was becoming more and more clear that the functionality of the upper layers decomposed not so much "horizontally" (each function applied in the same order to all PDUs) as the lower layers did, but more "vertically" into modules (the dialog progressed in phases with different PDUs). (In the mid 1970s, I had made this observation while investigating how to decompose protocols to improve their processing (others remarked on it as well), but there wasn't enough data to discern a pattern). This idea was opposed by many who were intent on adhering to the seven-layer structure regardless. (It was amazing how quickly for some the seven-layer model went from a working hypothesis to a religion.) It was clear that the implementation of the establishment phase of the Session, Presentation, and Application Layers should be considered one-state machine and the Session Functional Units (that is, the dialog control and synchronization primitives) should be viewed as libraries to be included if required.

In 1983, steps were taken to "adjust" the protocol specifications to allow the connection establishment of all three upper layers to be implemented as a single-state machine. This meant that the preferred implementation was to merge ACSE, Presentation, and Session into a single-state machine. This created a much smaller, more efficient implementation, if the implementer was smart enough to see it. (Most weren't.) This became known as the "OSI clueless test" for implementers. (A layer-by-layer implementation of the upper three layers indicated the implementor was clueless.)

This clearly says that the layer structure might need modification. But in 1983, it was too early to really be sure what it all looked like. Just because the connection establishment of the upper three layers could be merged was not proof that the data transfer phases should be. And remember, the CCITT wanted the data transfer phase of the session layer just as it was for Teletex and Videotex. So, it was not going to support any radical reworking of the upper layers. It also opposed the use of the presentation layer (Teletex and Videotex were defined directly on top of the Session Layer) and would not have agreed to a solution that made Presentation a fait accompli. So, in the end, it was felt that this was the best that could be achieved at the time. It was a small modification that made a move in the right direction and allowed much more efficient implementations to be done. It was hoped that it would be possible to work around what was in place after there was a better understanding. Meanwhile, the upper-layer group continued to work out the details.

This would not have been too bad if it were the only problem, but other problems arose that indicated that there were more fundamental problems with the upper-layer structure.

Problems With Mail: The Presentation Context Was Too Broad

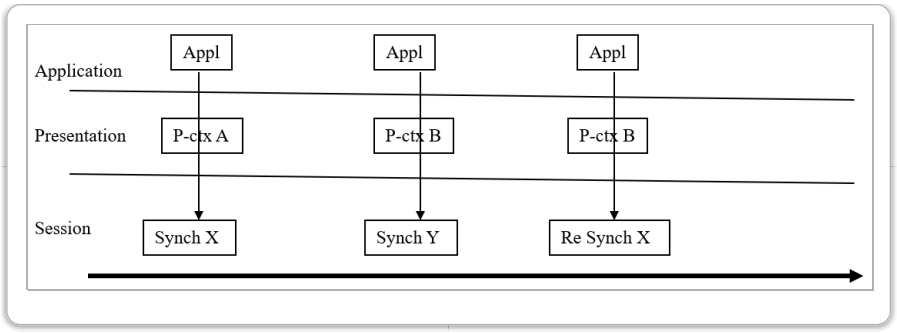


There were problems caused by relaying email in X.400. Connections between applications were the ultimate source and destination of data. However, the OSI Reference Model explicitly allowed relaying in the application layer. X.400 (or any mail protocol) may require such relaying. This implies that while the syntax of the "envelope" has to be understood by all the intermediate application layer relays, the syntax of the "letter" needs only to be understood by the original sender and the ultimate receiver of the letter, not all the intermediate relays. Because there are far more syntaxes that might be used in a letter, this is not only reasonable, but also highly desirable. However, presentation connections can only have the scope of a point-to-point connection *under* the application layer relay.

It was not possible for the Presentation Layer to negotiate syntax in the wider scope of the source and destination of the letter, independent of the series of point-to-point connections of the relays. The "letter" could be relayed beyond the scope of the Presentation Layer on another Presentation Connection to its final destination. The architecture required the relay to support all the syntaxes required for the envelope *and the letter*, even though only the sender and the receiver of the letter needed to be able to interpret its syntax. SMTP avoids this by an accident of history. When mail was first done, there was only one syntax for the letter, ASCII. By the time there were more, Multipurpose Internet Mail Extensions (MIME) could simply be added for the letter, with ASCII required for the envelope.

Not only that, but there was a problem with Presentation with the syntax of mail because no one had foreseen that one would build applications on top of applications with different scope, although in hindsight, relaying in the Application Layer implies that. Because the Presentation Layer defined the context for the whole Application Layer, a single syntax was required for the envelope and the letter. This meant that every mail relay had to be able to understand every syntax that a letter might use. Why? A mail relay only needed to understand the envelope. It didn't need to understand the syntax of the letter. Clearly this was a nonstarter. What was needed was to be able to have a Presentation Context that for the envelope and a separate Presentation Context for the letter. This could be solved simply by a sequence of application connections for relaying and a higher application connection for the letter.

Transaction Processing Exposes Wishful Thinking



According to the theory, if an application needed to change the presentation context during the lifetime of the connection, it informed the Presentation Layer, which made the change by renegotiating the Presentation Context. Different parts of an application might require a different syntax to be used at different times on a connection. However, it quickly became apparent that because the application protocol could invoke the Session synchronization primitives to roll back the data stream, the Presentation Protocol would have to track what the application protocol requested of the Session Layer so that it could ensure that the correct Presentation Context (syntax) was in use for the data stream at the point in the data when it was rolled back by the Session Layer. This added unnecessary complexity to the implementation and broke every rule about good design. It was a strong indication that the Session Synchronization Functions belonged above presentation, not under it. Confirming our original fears. (It is hard to describe just how intense all of these discussions were. It was very difficult to make a logical argument, that wasn't going to contradict poor design of existing development.

As more complicated applications began to be developed, conflicts in the use of the Session Layer developed. For example, CCR was defined to provide a generic two-phase commit facility that used session functions to build the commit mechanism. Later, the transaction processing protocol (TP) used CCR for two-phase commits but also made its own use of session primitives necessarily on the *same* connection. Session had no means to distinguish these two uses, and there was no guarantee that they would be noninterfering. TP would have to make sure it stayed out of the way of CCR, but that violates the concept of CCR (and all protocols) as a black box. In essence, Session Functional Units were not re-entrant and really in the wrong place. While there was a straightforward, elegant, and powerful solution, the participants preferred a quick fix.

Figuring Out What Was Wrong or "Green Side Up"[9]

All of these indicated severe problems with the upper-layer architecture, but the problems were also an indication of what the solution was. The whole upper-layer architecture had been badly distorted when the PTTs stole the session layer.

The session layer should have been a Session Layer: the functionality to create the context for an application connection. The intuition from the computing side that something like that was needed was correct. In other words, ACSE was directly on top of the Transport Layer. It wasn't really a layer by itself but

a common module of the Application Layer for creating application connections and authenticating that the applications were talking to who they thought they should be. Session Functional Units were useful but worked better as ASEs. The Presentation Layer collapses into defining the encoding rules for the Data Transfer Phase, including the ability to modify the Presentation Context. This might happen if the encoding rules needed to change in conjunction with a major synch or activity. The modular structure defined by the ULA group with the encoding rules created a rich fundamental structure for distributed applications. The view at the beginning that Bachman's seven-layer model was flexible enough to accommodate what we would discover would have been true had the PTTs not messed up the Session Layer.

There was just one thing that was needed, and it was "easy" to do: make ACSE recursive so that applications could be built on applications. This would not only solve the mail problem described above, but provide a framework for more sophisticated distributed applications. ACSE was revised to be recursive, so we could build applications on top of applications. I can confirm that specifying a protocol where the protocol it interfaces to is the protocol being defined is a very mind-bending intellectual exercise.

As mentioned earlier, this confirmed the early insight that the upper three layers were one layer. Later, the simplified implementation strategy was made more explicit within the bounds of what was possible with the specification of the Fast Byte implementation strategy where the Session and Presentation Layers were essentially nullified with one-byte placeholders, which made the upper layers look the way they should have. (It wasn't perfect, but it was within what was possible.

The lesson from this is clear: It is better to do what the model says and subset it for the current market, and then enhance it as there is new demand.

But the Conclusion is Inescapable.

There is no upper-layer architecture. Not as we thought of it. There is an Application Layer on top of the Transport Layer. *There is an application-layer structure or architecture*.

What Was Learned

OSI made major progress in furthering our understanding of the upper layers but ran into problems caused by the conflicting economic interests unwilling to compromise. Supposed short term gains outweighed long-term profits. Authentication, addressing the desired application, and specifying some initial parameters were generally the concepts associated with session functions. These were embodied in ACSE in the Application Layer. The theft of the Session Layer for the dead-end PTT Teletex and Videotex protocols turned the OSI upper-layer architecture upside down. The excuse for the problems, given as the use of session being a "pass-through" function, merely delayed the inevitable and cost more, not only monetarily but also in credibility. It is hard to argue that there is never a situation where a pass-through function may be the correct

solution. However, pass-through functions must be limited to those that do not cause a state change in an intervening layer. This severely limits the possibilities. Furthermore, in general one would want functions located as close to their use as possible. There would have to be a strongly overriding reason for locating another function between an application and a function it uses. Fundamentally, it appears that pass-through functions should never be necessary.

The fact that there is no multiplexing in the Session and Presentation Layers is a strong indication that although there may be session and presentation functions, they are not distinct layers. The presentation negotiation of syntax is best associated as a function of the application connection-establishment mechanism, which, practically speaking, it is. The session functions are actually common modules for building mechanisms used by applications that should be considered, in essence, libraries for the Application Layer. This also satisfies the previous result and is consistent with the common implementation strategy. Assuming that there are common building blocks for applications other than those found in the session protocol, this would seem to imply that one needs an Application-Layer Architecture that supports the combining of modules into protocols.

So, if the architecture is rearranged to fit the implementation strategy, all the earlier problems are solved… except one: the syntax relay. The flow for relaying is independent of the flow of the data for what is relayed. To take the email example, and apply what was developed in the earlier chapter, the letter from an email is a connectionless higher-level flow that specifies the syntax of the data it carries and is encapsulated and passed transparently by the relaying protocol of its layer (with its syntax) via a series of relays. Essentially, we need two layers of application connections to separate the “end-to-end” syntax of the letter and the “hop-by-hop” syntax of the envelope. This is an interesting result but not all that surprising. We should expect that applications might be used as the basis for building more complex applications.

One might want to build application protocols from modules but also want to build application protocols from other application protocols. To do this in about 1990, I proposed to revise ACSE, to make it recursive. To do this required ACSE to also negotiate the syntax of these protocols within protocols. As it turned out, the design of the OSI transaction-processing protocol using Session, Presentation, and Application Layers in the traditional manner was cumbersome and complicated. However, with this extended Application-Layer Structure, the design was straightforward and closely reflected the implementation approach. This model could also solve the mail relay problem simply by making the letter an encapsulated connectionless PDU with its own syntax being sent among application relays with their own application connections and syntax for the envelope.

OSI improved our understanding of the nature of the upper layers. The recognition of the roles of syntax and semantics in application protocols was crucial. The distinction between abstract syntax and various encoding rules is equally important and allows protocols to be designed such that they are invariant with respect to their encoding. This means the encoding can be changed without rewriting the entire protocol in much the same way that a compiler can change code generators without changing the language.

The recognition that Application and Presentation Context were specific cases of a general property of

protocols (that is, negotiating policy) was also important. The realization that the field in lower-layer protocols used to identify the protocol in the layer above was really a degenerate form of Presentation Context (that is, it identified the syntax of the data-transfer phase), not an element of addressing, contributed to our general understanding of protocols and answered a question that had been outstanding for decades.

In the same way that the Presentation Layer proved to be a false layer, this too proved to be a false and unnecessary distinction. However, OSI became locked into a particular structure of layers too early. If ACSE had been the Session Layer, OSI would have had five layers but a very different five layers than SNA.

The Main Lesson of OSI

The main lesson of OSI is to never include the legacy in a new effort. We can see how OSI's attempt to accommodate ITU caused many bad compromises to be made in the upper layers and bifurcated the lower layers. ITU hijacked the Session Layer for services that were a dead end. This so contorted the structure of the upper layers that there were no means to correct the problems without considerable modification of the architecture and the protocols. The ITU insistence on X.25 in the lower layers split them in two. (How X.25 was supposed to accommodate applications of the 1990s is still a mystery.) OSI was two incompatible architectures. All these conflicts undermined the effort and confused its adopters. Every word and punctuation mark in every document was a point of contention. Every minor point was a compromise with the old model, consistently weakening any potential of success. There was never a consensus on what OSI was, making it two architectures for the price of three, when it should have been one for half the price.

On the surface, including the legacy seems reasonable and cooperative. After all, one must transition from the old to the new. It is reasonable, but it is wrong. Contrary to what they say, the legacy doesn't really believe it is the legacy. They will point to successes in the marketplace. If people buy it, how can it be wrong? There will be considerable emotional attachment to the old ways, not to mention the vested interest in the many products and services to be made obsolete by the new effort. The legacy will continually be a distraction. Their reticence to change will cause the new effort to fall short of its potential and jeopardize its success. Death by a thousand cuts.

No matter how great the temptation to be reasonable and accommodating, this is one point that cannot be compromised.

While none of the above helped, none of it is why the Internet ended up dominating. There are three reasons why the Internet became dominant: 1) DARPA was spending orders of magnitude more money than everyone else combined; 2) Moore's Law masked the flaws; and 3) computer companies were targeting the market for corporate networks or internets. The market for an internet utility was coming but had not arrived yet. Corporate demand was moving toward it, but the Web brought it about much sooner.

---

8. The PTTs have consistently proposed initiatives [Videotex, Teletex, ISDN, ATM, WAP, and so on] whose market window, if it exists at all, passes before they can create the standard, build the equipment, and deploy it. The sale of these is a credit to their marketing departments or the desperation of their customers.
9. The punch line of an old joke from at least the 1970s, possibly before. The joke is about guys laying sod and the foreman having to constantly remind them, "green side up." The reader can fill in the rest. (The name "Green Side Up" is used fairly commonly in the lawn and garden industry.)

Is the Application-Entity the Application Protocol?

Of course, that is what we said it was above. It was the example we used to explain what the application-entity was. So, at first blush, yes. But consider the following:

> There are only six operations that can be done remotely—*read/write*, *create/delete*, and *start/stop*—on different object models.

That implies that *there's only one application protocol*. The objects being manipulated are always part of the application. For example, with file transfer, the files are not part of the File Transfer Protocol, they are outside the protocol. Read and Write are operations on the file or file attributes.

So, is there's no AE or only one AE?

No, there can be multiple AEs. The AE defines the collection of objects available to a given application connection, for this communication. The objects available are likely a subset of the entire model or object models in different domains.

Thus, we transition from an IPC model to a programming model. Distributed applications manipulate objects in a programming language that is translated into sequences of the six operations on the objects.

What does the one application protocol look like?

Connect and *disconnect* would be part of the common ACSE-like function and then an operation with an invoke ID so that requests can be asynchronous. Scope and *filter* enable the same operation on multiple objects in one operation. Typically, scope designates part of the object model, and filter specifies a relational expression to which, if satisfied, the operation is applied.

The transition from an IPC model to a programming-language model greatly simplifies the development and deployment of distributed applications.

What Did We Learn?

- Collect models so that you don't fall victim to “when all you have is a hammer everything looks like your thumb!”
- Look for solutions that are symmetrical, while keeping in mind that not all of them are.
- Sometimes what looks technically logical isn’t right.

- In general, follow good design principles, even when there's no technical reason to do so. There's a very good chance, the problem will bite you later if you don't.
- Always look for common structures.
- The importance the Application-Entity/Application-Process distinction.
- Making protocols independent of syntax.
- Applications are all about the object models, not the protocols.

And a bit of doubt is always healthy.

But if there are two important take-aways from this, they would be:

Be clear-eyed about what the problem really is. Remember Bernie Cosell's insight that Telnet was a computer-to-computer protocol, not a computer-to-terminal protocol. Take that extra time to consider if the first take on the problem is the right one or if there is something else more fundamental in the solution.

And second, remember the lesson from the OSI upper layers: do what the problem requires. If a stopgap measure is necessary, replace it as soon as possible. Don't wait for conditions to require replacing it. Often by that time, others will have rationalized the stopgap as reasonable and used its unique properties for subsequent development, which will make replacing it difficult.