Networking Is IPC and Only IPC

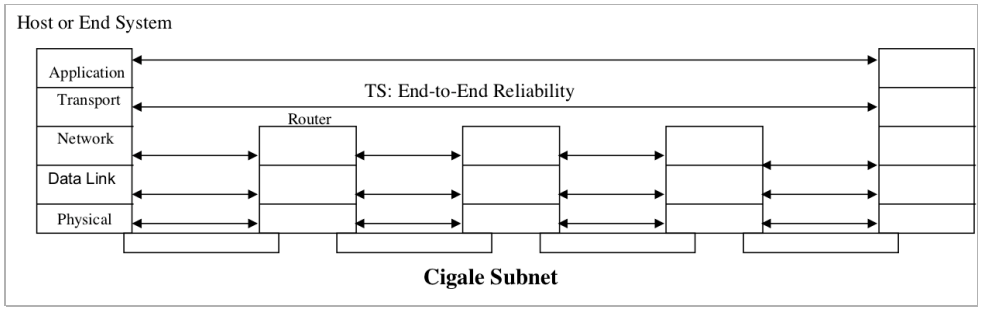
Introduction

> "There is something fascinating about science. One gets such wholesale returns on conjecture out of such a trifling investment of fact."
> —Mark Twain, "Life on the Mississippi"

If you're not reading Mark Twain, you should be, especially the essays. They are great! But all Twain is great! I was Skyping with one of our grad students in Europe from Greece one afternoon, and the conversation turned to Twain. The student had read *Huckleberry Finn*. I asked if it was in English or Greek, and the reply was in Greek. My response was, "Then you haven't read *Huckleberry Finn*." I went to my bookshelf, pulled out my copy, and opened it at random. On the two facing pages, I read three different American dialects. This quote from Twain's book *on the Mississippi* takes place in the same region. It is in two parts: The first part has stories of Sam Clemens' days as a pilot on Mississippi River steamboats. The second part is 50 years later, when the great American author, **Mark Twain**, returns to take a vacation down the Mississippi on a steamboat and reminisces about his days on the river. At one point, he was up in the pilot house talking to the captain and the pilots. One of them pointed out that the distance from Cairo, Illinois, where the Ohio and Mississippi meet (Now, you may have thought Cairo is pronounced Kai-row, but we know how to pronounce things in Illinois: it is pronounced KAY-row) to New Orleans was 500 miles shorter than when Twain had been on the river. As you probably know, rivers naturally wind back and forth a bit to take the easy way. Then, once in a while, there will be a flood that cuts through and shortens the river, leaving an oxbow lake. Twain said that, given that thousands of years ago the mouth of the Mississippi was 500 miles out in the Gulf of Mexico, then thousands of years in the future, the distance from New Orleans to Cairo was going to be 50 miles! And then he said this quote: "There is something fascinating about science. One gets such wholesale returns on conjecture out of such a trifling investment of fact!" He knows his conjecture is foolish. Let us see if our facts can get a better return by investment.

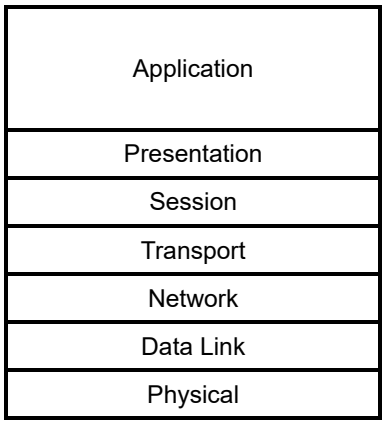
Looking at What We Had

The Cyclades Architecture (1972)

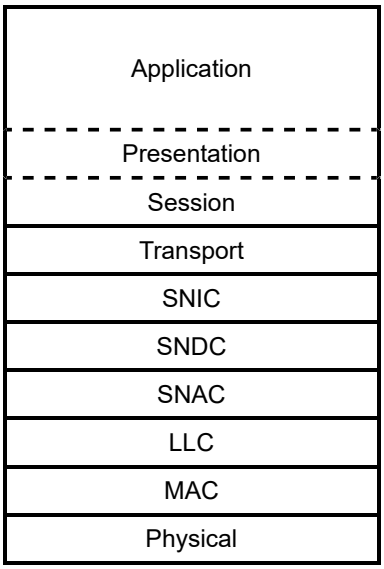


In the last lecture, we talked about how we started out with the CYCLADES architecture. The Physical Layer and the Data Link Layer detect corrupted packets. The Network Layer does relaying and multiplexing of datagrams, where the primary loss is due to congestion and rare memory errors incurred during relaying. The transport layer recovers from those errors. The Data Link Layer doesn't have to be perfect. (It was in those early networks, this was a HDLC-like protocol.) The Data Link Layer just has to keep the error rate on the point-to-point links well below the rate of loss due to congestion and memory errors seen by the Transport Layer.

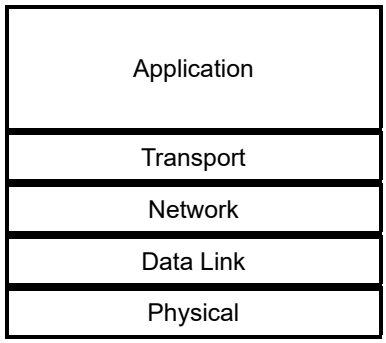
Based on this, OSI proposed that there were seven layers.



But as they dug into it, things became more complicated, or they seemed to. By 1983, OSI found that that the upper three layers didn't exist. They were actually one layer. The network layer turned out to be three sub-layers and the link layer was divided into two parts.



Something was on going on. This reflected what we were seeing. But it was clearly not a clean design. Meanwhile, the Internet just stayed with what they always had. TCP, IP, whatever was above that, and whatever was below that.



We were missing something. There was something we weren't seeing.

Going Back to Early Ideas

It has always been my position that *networking is interprocess communication and only interprocess communication.* I am not alone in this belief. This was common knowledge in the early days of networking. This is how it was thought about it. We saw networking as more related to operating systems than to telecom. You can see that from the early applications we did: Telnet (terminal support), File Transfer Protocol (FTP), and Remote Job Entry. We were replicating the functions of an operating system on the network.

In 1970, Dave Walden wrote RFC 61, "A Note on Inter-Process Communication in a Resource Sharing Network." In a 1972 paper on the issues of writing an NCP1 implementation for the ARPANET, Bob Metcalfe (inventor of Ethernet) mentions in passing, as if it is a simply common knowledge, that "networking is interprocess communication." In 1982 in RFC 871, Mike Padlipsky calls interprocess communication axiomatic to networking.

With things getting messy and complicated, I realized I needed to understand what I really knew about networking independent of politics, hardware constraints, installed base, etc. Also, a friend had asked a question about TCP and I didn't like the answer I gave. The more I thought about it, the more I realized I could write a little exercise that would explain why all of this stuff was necessary. So, I did.

To understand why networks are the way they are, I had to go back and consider the problem from the very beginning—not historically, *logically*. I had strayed away from the idea that Networking was IPC and only IPC.

This lecture is a roadmap of where this course going. You are not expected to understand everything in this lecture now. By the end of the course, that will hopefully be different. But that doesn't mean you shouldn't raise the questions. We will either answer them or ask you to hang on to that question until a bit more has been covered. But ask anyway! We will go through all of this in a lot more detail as the course progresses. This is an overview of where the course is going.

---

1. An implementation of the ARPANET Host-to-Host protocol was called the Network Control Program (NCP) as was the protocol itself.

Interprocess Communication

Let's start with the basics and build it up. This is going to be very simple. Readers with some exposure to networking will have a tendency to react, thinking, "O, *I* already know this. I can skip this." Patience. Pay close attention anyway. Think carefully about what is being said. I thought I knew all this too! (And I had a lot more reasons to believe that!) But I learned things too. Think carefully as we go through each step, think about its implications, and I will try to point some of them out. This will be at a higher level of abstraction. One place we often go wrong is getting detailed too soon. You will see some things and your reaction will be, "O, yeah, that is what that is *really* about!" (If you don't, I can almost guarantee that you missed something.)

If you don't know anything at all about networks, don't worry about it. This will be a nice introduction at the 50,000-foot level to how things are supposed to work; why networks do the things they do. Later we will go back and look at all of this in detail. But bring your questions to class. We can either answer them or we may have to wait until we have more details under our belt. So sit back and enjoy the story. Along the way we will undoubtedly have to throw away a ladder or two.

An Aside on Thinking: Top-Down vs. Bottom-Up

We know the importance of getting a good design. We have also heard that purely top-down or purely bottom-up thinking is not good—a combination is necessary. But how they are used is different. We just cautioned about getting too detailed too soon. That is part of it. We start top-down thinking at a high-level of abstraction. Getting this abstraction right is very important, because it affects everything that derives from it, as do the initial assumptions. It is important to not let short-term immediate requirements influence the assumptions. It will constrain the solution too early and likely cause one to miss degenerate cases, e.g., including capabilities at no cost. The best approach is to let the problem tell you what is important. It is smarter than we are. If there is an existing model that seems to work, carefully consider how it differs from the current problem. Very subtle differences can have a profound effect. (We have seen very small subtle assumptions, like the distance between computers, have a profound effect on an architecture.) Keep an eye on these differences as areas where there might be trouble. If it starts to look cumbersome, we should consider a new fundamental abstraction. But don't be too quick to abandon a model; sometimes there is "hump" to get over and then it is much simpler. One progresses in rough (seldom formal) levels of abstraction. The advantage of top-down is that the higher levels of abstraction leave more options open as to where the solution can go. Bottom-up imposes constraints early, which then limits the solution to areas of the problem space in the hope that they are not at odds with the solution as a whole. Avoid special cases; look for degenerate cases. (As we will see, sometimes the most different, the most oil-and-water problems turn out to be degenerate cases of a more general view and a much simpler solution.) We may see abstractions that group similar low-level pieces together. This can be good. It provides a trial to see how it fits into the solution as a whole. The choices can also be wrong. But even then, they provide insight into what may be right. We come down from the top including elements in the solution, at each stage (successive levels of abstraction) we survey what are pieces at the bottom that have not been included that are candidates for inclusion at the next lower level of abstraction. If we are careful and willing to make mistakes and back up, generally we will find that excellent design will result. To do this well is a talent. As with all

talents, some have it, some don't. It can be improved but not created. There is nothing bad about this. Not everyone has every talent! I certainly will never be a virtuoso violinist or even decent violinist, an Olympic gymnast, etc.

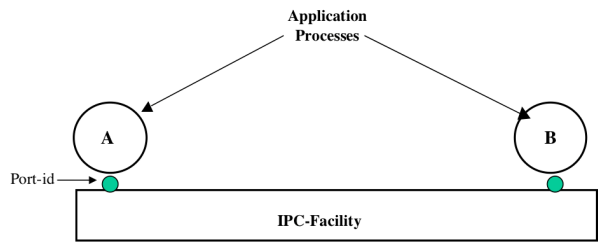
Not long ago, a group who had conducted over 50 oral history interviews with important people from the Internet was asked what major insights or principles had the interviewees noted? Thinking that some general principles must have come out of the last 50 years that could avoid future mistakes, point at simplifications, etc. The response was that several had remarked that the Internet showed that bottom-up evolution worked. That seemed to beg the question, what major insights had come from that? The interviewers couldn't think of any. Upon reflection about the observation, one realizes that what the success of bottom-up evolution implies is that there is always something that can be added to deal with a problem. In other words, there is always a patch or kludge that can be made. This would imply constantly increasing complexity, which is indeed what we are seeing.

A rather long digression, back to our problem.

So, what do we need for communication? First of all, you need some minimal shared understanding. When Columbus first came ashore in the New World to meet the local people and said pointing to his chest, "me Columbus." They didn't think that he meant his breastplate. They knew that he meant himself and they could build on a shared but limited understanding. We also need a medium to carry the messages, and in the operating systems, the mechanism to do this is **interprocess communication (IPC)**.

Two Application Processes in One Computer Communicating Through an IPC Facility

Let's start with the basics:



Two applications communicate in the same system. The IPC Facility is a black box that provides a service to the application that makes it appear there is a channel between them.

How does this work? First, we need system calls so that the processes can invoke IPC. I posit that these are the ones we want:

- port-id := Allocate (dest-appl-name, ipc characteristics)
    - to allocate an IPC channel, which returns a handle like a file descriptor, which we will call a *port-id*. The Allocate includes the name of the destination application to be communicated with and maybe some characteristics about the communication. This tells IPC the kind of resources that are going to be required.[2]
- success := Read (port-id, length, buffer-ptr)
- success := Write (port-id, length, buffer-ptr)
    - *Read* and *Write (*or *Send* and *Receive)* to the port-id, the number of bytes pointed at by buffer-ptr.
- success := Deallocate (port-id, reason)
    - *Deallocate* is invoked when the dialog is complete to release the resources associated with the IPC.

Notice the similarity with the operations on a file. We will want to preserve that if at all possible. IPC is going to create a channel between the two applications identified by the port-ids, which have local scope (only unambiguous between the application and the IPC facility). Port-ids are not only a short identifier for the end of the channel but also allow an application to have multiple IPC channels at the same time.

An IPC Facility like this is generally constructed with some sort of shared memory mechanism and usually requires the instructions to create a critical section to handle concurrency and keep everybody out of each other's way.

So, how does it work?

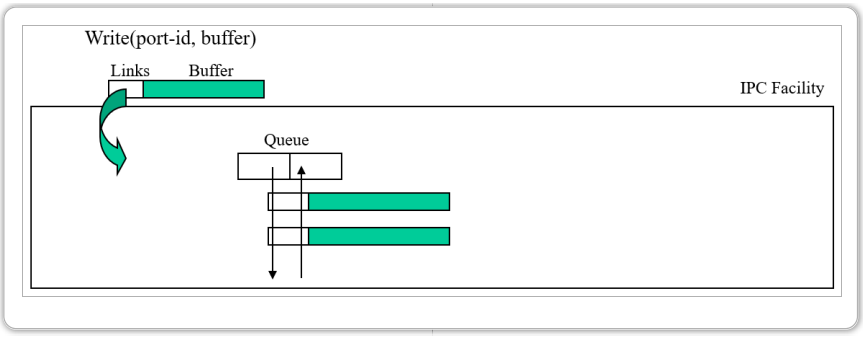
1. **A** invokes the IPC facility to allocate a channel to **B** by calling an *Allocate*.
2. IPC determines whether it has the resources to honor that request. If so, IPC uses its *search rules* to find **B**.
3. When it finds **B**, it determines whether **A** has access to **B**. IPC may cause **B** to be instantiated.
4. **B** is notified of the IPC request from **A** and given port-id **b**.
5. If **B** responds positively, IPC notifies **A** using port-id **a.**
6. to N. They do their *reads* and *writes* back and forth exchanging information for a while and when they›re done…
7. They do a *Deallocate* on the port-id to release the resources.

Taking the Cover Off the Black Box

What goes on inside the IPC facility to make all of this happen? There will be two FIFO queue heads allocated for each port-id pair, one queue for each direction: **A** to **B** and **B** to **A**. The sender invokes Write to deliver a message to the IPC facility, which posts it at the end of the queue and the destination invokes Read to cause the IPC facility to remove messages from the head of the queue to deliver them to the destination.

Click the tabs inside the menu to see the details of what happens inside.

Steps 1	Step 2	Step 3	



A FIFO queue head is allocated for each port-id pair.

- Memory allocation keeps links (user can't see) so buffers can be passed from queue to queue.
- Lots of ways to implement this.
- But manipulating the queue requires a critical section, i.e. synchronization.

The OS memory allocator keeps links as part of the buffer, which the user can't see, so that buffers can be passed among the various queues the OS keeps[3] and the free-space queues. Depending on the implementation, IPC may have its own links in the buffer for IPC queues to use. But manipulating the queue requires a critical section and synchronization. Each write causes a buffer to be linked to the end of the queue. Each read causes a buffer at the head of the queue to be dequeued and passed to the application. These operations cannot be done at the same time, so a critical section is necessary to keep queuing and dequeueing from interfering with each other.

Notice that we are assuming that if **A** and **B** want to exchange messages, they know what the messages mean and what they should do with them.

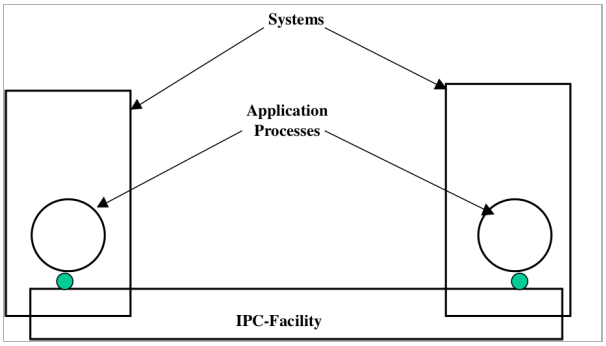
---

2. In mathematics, "=" is a Boolean operator, not an assignment operation. That should be its meaning in computing. Its use as an assignment operator is sloppy thinking. There was an actual "left arrow" in ASCII to be the assignment operator. Some started using ":=" instead. (Clearly "<=" had to be used for "less than or equals.")
3. The IPC facility looks like just another device to the OS.

Two Applications Communicating in Two Systems

What happens if we have two applications communicating in two systems?

Two Application Communicating in Distinct Systems



How does it work? We start out the same:

1. **A** invokes the IPC facility to allocate a channel to **B**.
2. IPC determines whether it has the resources to honor that request. If so, IPC uses its *search rules* to find **B.**

Finding the Other Application

We have a problem. **B** may not be in the same system as **A.** Management of the name spaces is no longer under the control of a single system. We can't have different applications called **B** in two systems. The names must be different. We didn't get very far before we ran into major problems. Each system no longer knows all the available applications. We need a way to ask the other system if it has a **B**.

We need to be able to send a message to the other system asking if it has a **B**. To do that, we will need to tell the other system what to do with the message asking about **B**, and the other system will have to implement code to handle the message. The other system has to be told what it means to get that message and to do something with it. Also, local access control can no longer be relied on to provide adequate authorization, so we'll have to do something about that.

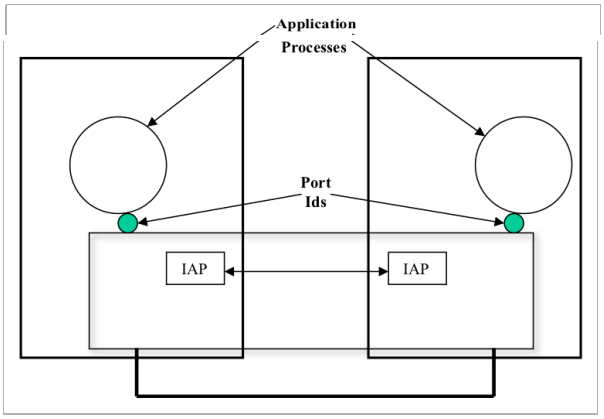
We need some kind of protocol to carry application names and access control information to the other system, such as asking it "Have you seen **B**? I don't have **B**! Where is B?!"

Let's call this an *IPC Access Protocol (IAP)*, just to have a name for it. What is it for?

IPC Access Protocol (IAP)

- To find out if the other system has an application with this name.
- To tell the other system something about the nature of the communication being proposed. (This is so that the other system can determine if it has the resources to support the communication.)
- To send the other system access control information to determine whether the requesting application is allowed to access the application it is requesting.

How Does It Work Now?



How do we know when to use this IAP? Well, if the requested application isn't here, it might be over there! We need to use it! Remember, what did we said about PDUs?

Protocol Data Unit (PDU)

PCI	User Data

The PDU must tell the protocol machine what to do with the information in the PDU. Protocol-Control Information (PCI) or the “header” is what the protocol understands while user data is what the protocol doesn’t understand.

What do these PDUs look like?

Type	Dest. Appl. Name	Src. Appl. Name	QoS & Policies	Access Control

This is a simple request-response protocol. It will have a `type` field that tells whether it’s a request or response, a `destination application name` or the `source application name`, so that you can use it with the `access control information`, plus some information about the nature of the communication.

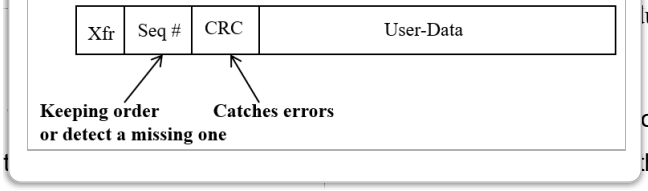
Notice, given what we saw in the last lecture about PDUs, this one’s a little odd. There is no user data. The buck stops here. This is what characterizes application protocols; they don't have any user data. They are the data.

Getting Data There Intact

But we have a more basic problem here. How do we get the PDU there?

We know that in getting PDUs from **A** to **B** in different systems…*bad things can happen!*

We know that PDUs can be corrupted while they're being transferred. We need to ensure that the bits we send get to

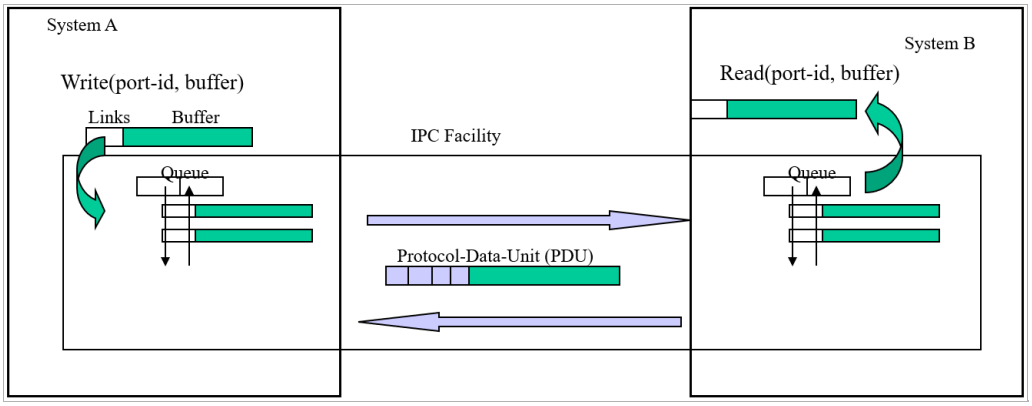


the other system unaltered. ore detail later). We can use a relatively simple error code t he sender to retransmit it, or we can use more complex error *correcting* techniques, but the PDU may be so badly mangled, it still may have to be retransmitted. We will need Flow Control to keep the sender from sending faster than the receiver can process the PDUs.

Both of these, retransmission and flow control, are feedback mechanisms. Feedback requires coordination and some synchronization. We have lost the ability to use shared memory for synchronization. We must create some shared state between the two systems and some sort of explicit mechanism to ensure synchronization and maintain performance.

We need some kind of protocol to do error recovery and flow control. Just to have a name for it, we will call that an *Error and Flow Control Protocol* (EFCP).

What Happens Inside: EFCP



For the other system's IPC facility to use the EFCP PDU, the protocol must only process the PCI. Now we're going to need a FIFO queue on both sides for sending and receiving PDUs. (Actually, we will need a pair of them for each direction, but it makes the figure too busy.)

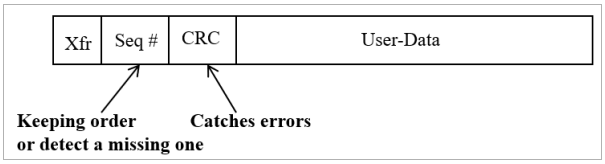
What needs to be in the PCI of the PDU?

We need to detect lost or corrupted PDUs, delimit the PDU so we know where the PDUs begin and end, and pack the SDUs (Service Data Units from the user) into the User-Data field. We need synchronization locally for the queues, but we also need synchronization between the systems for flow and retransmission control.

What does this protocol look like?

It is going to be a symmetric protocol because we're going to send data in both directions. Obviously, we're going to need a Transfer PDU that carries the data. We will need to know that it is a Transfer PDU that tells us it is for moving data, i.e., a type field.

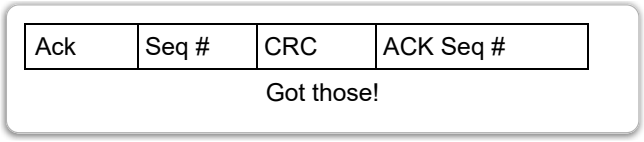
Transfer PDU



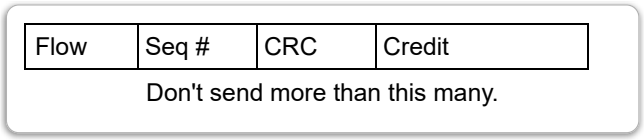
In this case, the first user data we want to send will be the IAP request/responses, but EFCP won't know that that is what it is. Once the destination application has been found and source application notified, then the protocol will be used to carry the data exchanged by the applications.

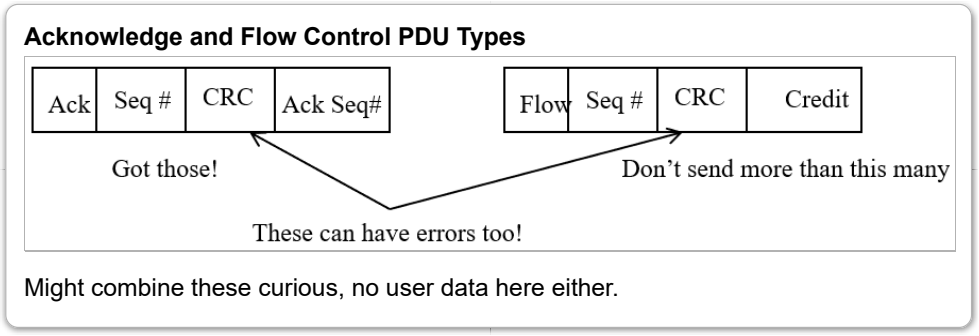
We will have a CRC. Some kind of error code that can be used to determine whether the PDU has been damaged. ("CRC" is used here because "Error Code" wouldn't fit. We will explain later what a CRC is.) We will want to have a sequence number, so we can keep PDUs in order and also detect whether any PDUs are missing. Then there is the User-Data field to carry what this is all about.

But that doesn't help us with our feedback mechanisms. We are going to need a couple other PDUs: one, an Ack (Acknowledge), which says we have received all PDUs up to a certain sequence number. We will have to protect it from having errors and maybe give it its own sequence number.



We need a PDU (credit) to tell us how many PDUs we can send before we have to stop. These can have errors, too, so we will need the CRC and sequence number fields. These are small. When we send one, we nearly always send the other one, so we might actually end up combining them for efficiency purposes.





Now It All Works

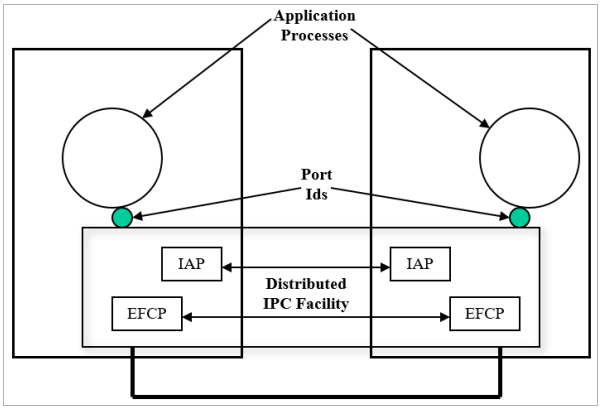
So now what happens? As we said before, we need a queue at both ends.

Click the tabs inside the menu to see the details of what happens inside.

Steps 1	Step 2	Step 3	

We are going to exchange packets back and forth with the PCI, which will allow us to take PDUs from the sending queue, send them across the wire, process the PCI, and put them at the end in the destination queue, where the destination application can read data. The PCI tells the destination what to do with the data and the protocol state machine will handle the Ack and Flow Control or Credit PDUs, to make sure that everything stays synchronized. So there are PDUs going back and forth.

How Does It Work Now?



So, how does it work now? Let's start over:

1. **A** invokes the IPC facility to allocate a channel to **B** by calling an *Allocate*.
2. IPC determines whether it has the resources to honor that request. If so, IPC uses its *search rules* to find **B.** Not finding it locally, it sends an IAP request to **B**, creating an EFCP connection to use for sending the IAP request.
3. When it finds **B,** it determines whether **A** has access to **B**. IPC may cause **B** to be instantiated.
4. **B** is notified of the IPC request from **A** and given port-id **b**.
5. If **B** responds positively, IPC sends an IAP response back. IPC notifies **A** using port-id **a.**
6. to N. Creating the EFCP connection, **A** and **B** do their *reads* and *writes,* sending PDUs back and forth with data for a while and when they're done…
7. One or both of them do a *Deallocate* on the port-id to release the resources.

From the users' perspective, it's just like before. Nothing is changed.

Assumptions Invalidated by Moving to Two Systems

To be explicit, let us summarize the assumptions that no longer hold:

- Management of the namespace was no longer under the control of a single entity; we had to deal with that.
- We must tell the other system what to do with the PDUs we are sending it.
- We lost local access control; we need a scheme for doing that over two systems.
- All resources are no longer under the control of a single system.
- Bad things can happen to corrupt messages in transit. We have some things we can do about that, including retransmission control, which we'll talk about in more detail later.
- The receiver must be able to tell the sender it is sending too fast and to slow down. We need flow control.
- The feedback mechanisms will require synchronization. With loss of shared memory, some form of explicit synchronization is required.

Otherwise, not much has changed, at least not for the application (and we should try to keep it that way).

What New Concepts Were Needed?

To handle these assumptions, we added three new concepts:

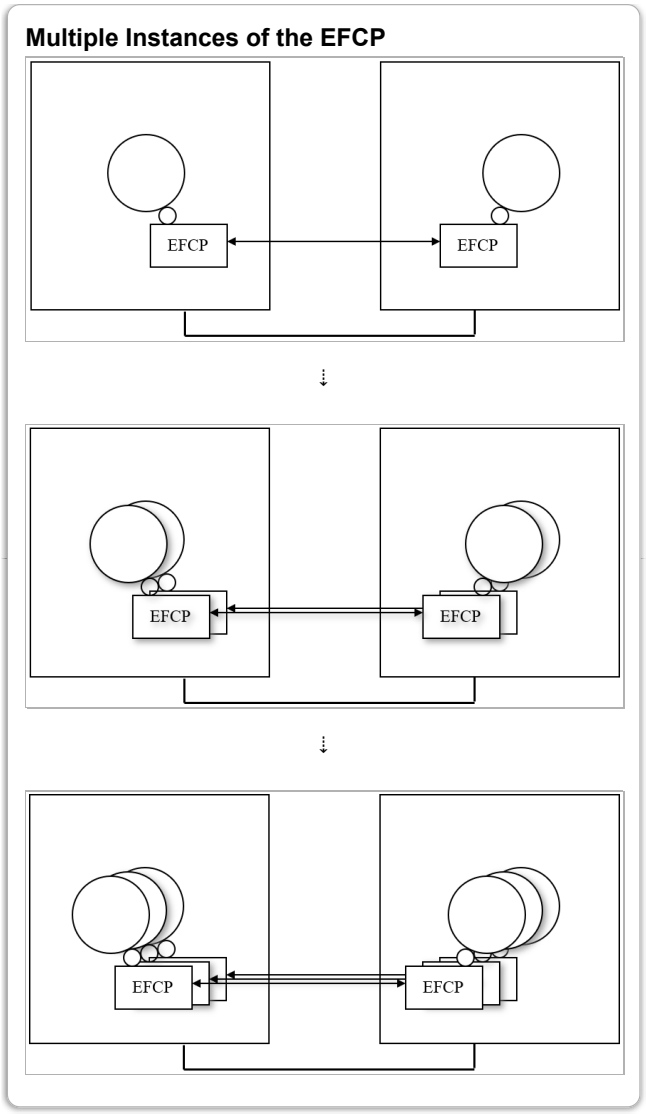
- An application namespace that spans both systems.
- A protocol to carry application names and access control information.
- The protocol that provides the IPC mechanism itself and does Error and Flow Control maintains a shared state, detects errors, ensures order, and provides flow control.

For the time being, resource allocation can to be handled by either side refusing the request.

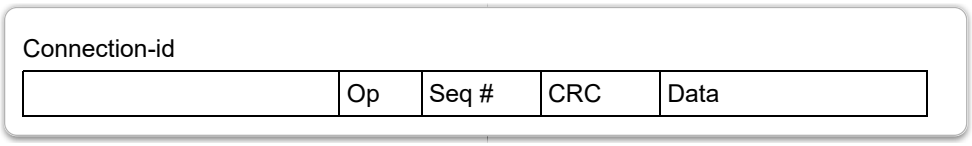
What Happens if There Are More Than Two Applications Between Two Systems?

What happens if there are more than two applications between two systems? How does this work?

Need to Distinguish Multiple Flows



The first thing that we need is the ability to distinguish which PDUs belong to which EFCP connections. We need to add some sort of connection-id to the front of all PDUs. Knowing which flow this PDU belongs to means we know which instance of the protocol state machine to update. Note: In designing the format of the PCI, there is a tendency to put the fields in the order in which they are used.



So how do we do that? Because either side could initiate a connection, the two sides can't just assign an identifier independently. They might assign the same one. In the early days, some people solved this by saying: One side will start allocating ids from the left end of the field and the other end will assign from the right end of the field, and we hope they don't meet in the middle. It works, but it is pretty kludge, not a great idea. It will only work if there are precisely two systems in the network and no more.

But wait a minute! This problem doesn't occur at the API. There the port-ids distinguish different instances of IPC, but they are assigned independently. There could be a conflict. The two ends could assign the same value, but not if the

two port-ids are simply concatenated to form a connection-id. In fact, that gives us distinct port-ids for each direction of the flow. The sender concatenates <Destination port-id, source-port-id>.

Connection-id

Dest-port	Src-port	Op	Seq #	CRC	Data

This is commonly what is done. That is what is commonly done.

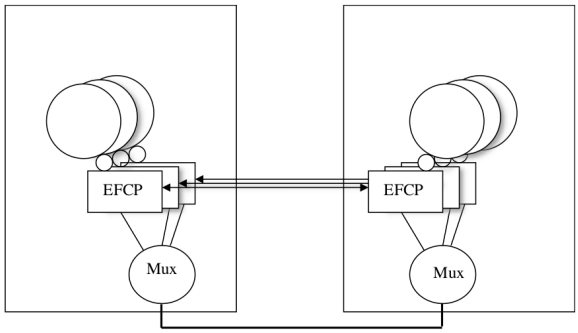
This allows more than two flows between the same two systems.

Deciding Which PDUs Are Sent

Now there's another problem.

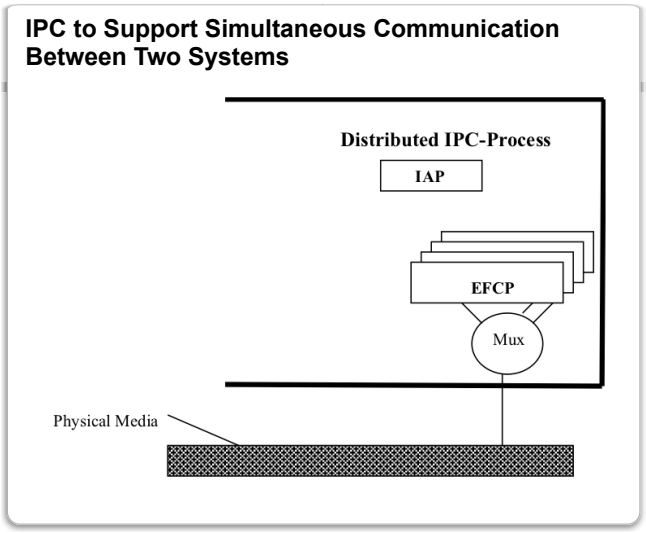
We have multiple applications all trying to send PDUs over the same wire at the same time. We need a task to *manage multiple users of a single resource*, the wire. (You should recognize that phrase. It is the definition of what an operating system does.) There is going to be contention for the wire.

Simultaneous Communication Between Two Systems, i.e., Multiple Applications at the Same Time



We need something that handles that. We add a multiplexing task. This task just services the queues and posts PDUs to be sent. It implements the policy determined by a separate task that is more complex to determine how the resources are shared.

Keeping Track of Everything



For N applications between two systems, we have added new structures/concepts:

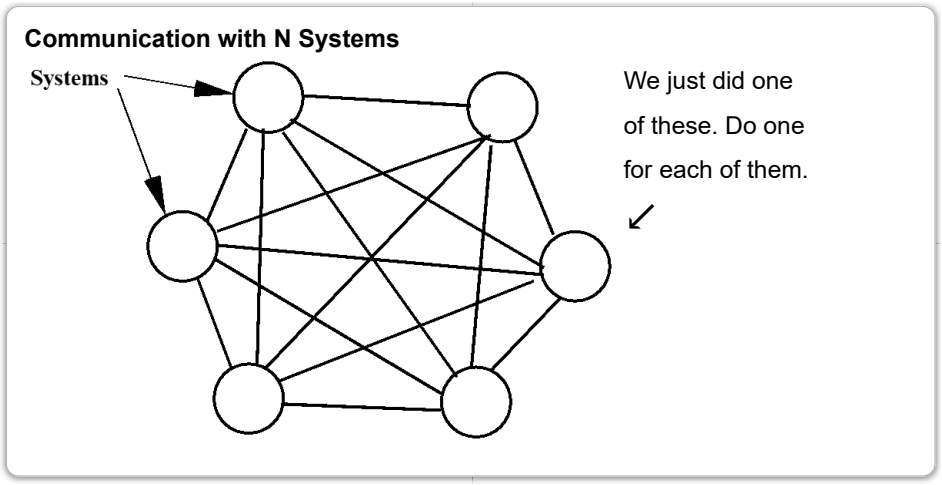
1. With more than one connection between two systems, a connection-id must be associated with every PDU to distinguish what data belongs to which connection. This is commonly done by concatenating the port-ids.
2. A multiplexing task to manage a single resource, the physical media. The multiplexing app will have to be fast and have low overhead. It should not get in the way and slow things down. It decides what data is sent and when.

Application naming could be a bit more complicated with multiple instances of the same application, but we will leave that until later.

Now we are able to support multiple simultaneous communications between two systems.

N Systems…Directly Connected

Now let's go to N systems…directly connected. (You thought we were going somewhere else but not yet. We are going to take this one small step at a time.)



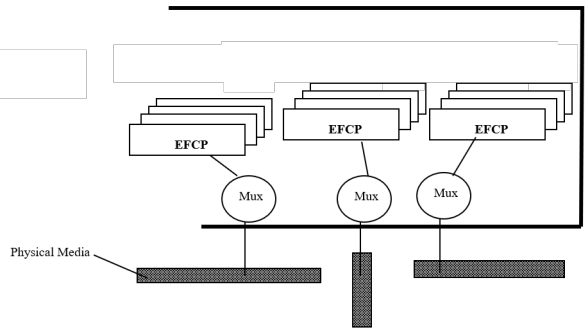
What do we do for this configuration? We have N systems all directly connected. This is simple. We just did one of

these for two systems. Now each of the N systems needs (N-1) instances of what we did for two systems. For that, we replicate one of them for each of the wires that comes out of a box.

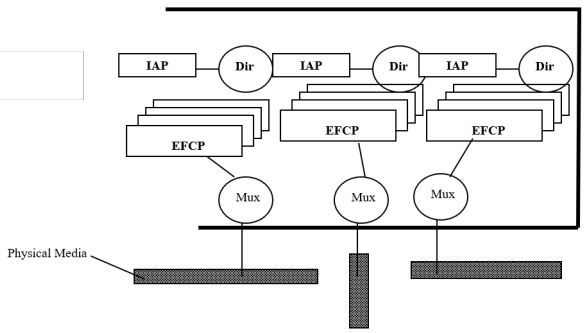
Creating the First Layers

A Separate Multiplexing Application Is Needed for Each Physical Media Interface

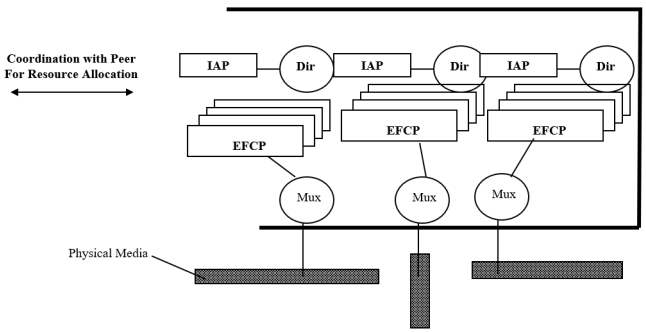
There is a common structure here that can be factored out.



We will need to use IAP to find out which applications are on the other end of which wires.

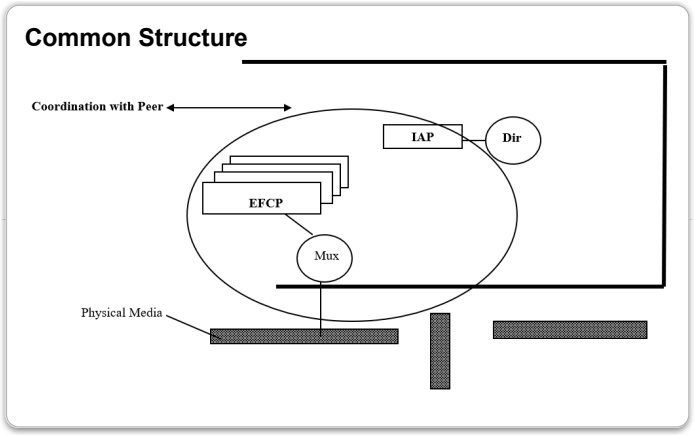


And with more systems, better coordination of resources will be required.

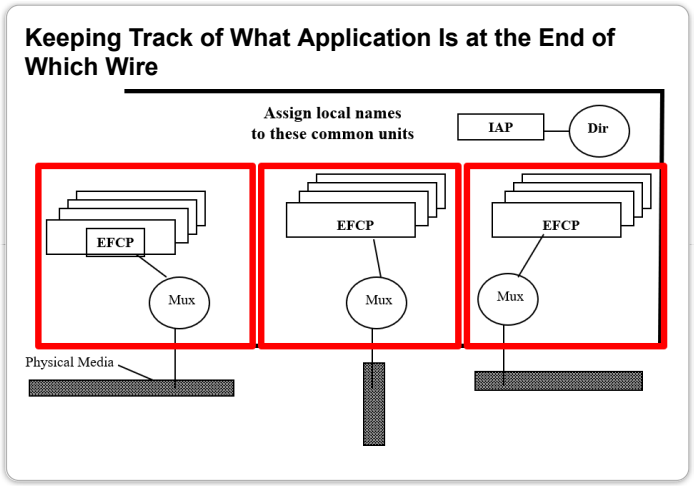


We can organize these IPC facilities that interface to a wire into separate modules of similar elements. Each one

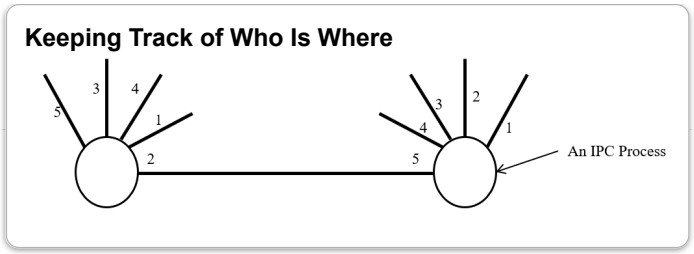
constitutes a “Distributed IPC Facility” (DIF) of its own, which consists of: IAP, Error and Flow Control Protocol, multiplexing task, Directory, Per-Interface Flow Management, etc. Then, we need a management task to manage their use and moderate user requests.



We can factor these out and make each one of them into independent black boxes all by themselves. (*We just created a layer*.) In fact, we have created an IPC facility with multiple layers of the same rank, one for each wire. We want it to look like one facility to the user though, pulling out the IAP/Directory function to be common to these layers.

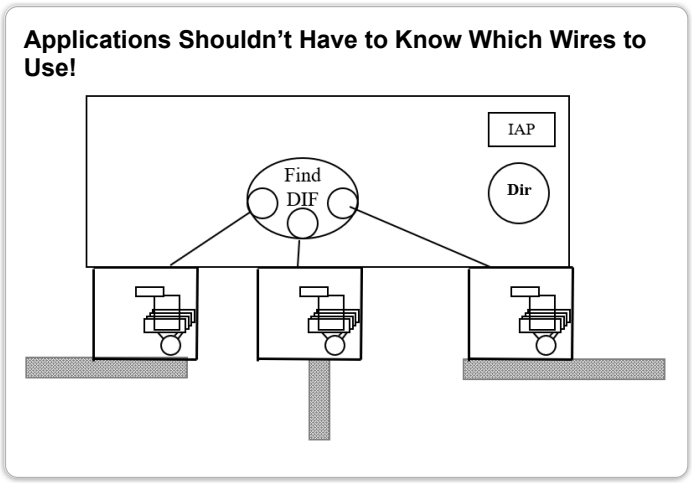


IPC can find the destination by choosing the appropriate interface. Local names of multiplexing applications that are managing interfaces suffice to label interfaces. (With point-to-point wire, PDU that goes in one end *has to come out* the other!) Each one can number them however they want. They are only used locally. A local table keeps track of which applications are reachable by which interface. The same local names can be used to keep track of which EFCP-instances (port-ids) are bound to which multiplexing task. *The important point here is that, so far, all IPC has needed is local identifiers*. (Addresses have not been needed.)



Replication Entails More Management

Application names become more elaborate. We want to be able to have multiple instantiations of the same application on the same system. We will want application names to be location independent as in operating systems. We want to be able to move an application without changing its name. Once in a while, there may be a need for them to have a location-dependent name, like 2nd-Floor Printer. But for the most part, these are rare. Most of the time, we will want location-independent names. We will want to create local databases to remember what's where. We will likely use a combination of cache and query: start by querying the first time, and when it is found, record it in the table/database. Time can be saved when there is a request for the same application, unless it moves.



Because we really don't want the user applications to have to keep querying all the different wires, we need a Find function that figures out what applications are at the end of which wires. The function will maintain a table so that it doesn't have to ask every time. The application requests an Allocate, and the function consults its table. If the name is found, it reports back. If not, it queries the wires looking for the application. This preserves the API that the application sees to make it location-independent and look as much like a normal file operation as possible. Also, so that IPC with an application in the same system is no different than IPC with an application in a remote system.

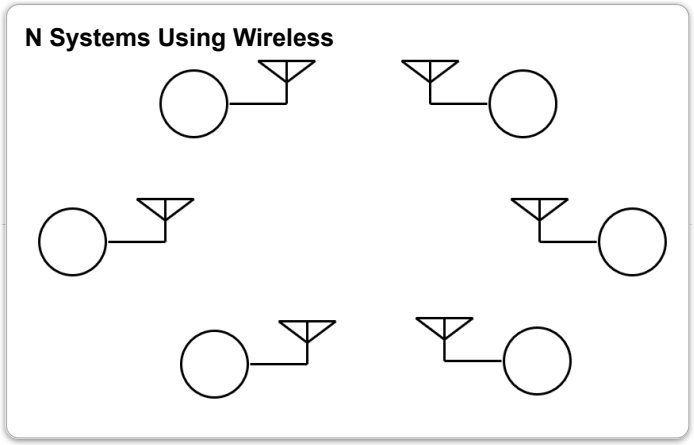
With (N-1) destinations, we may need to coordinate resource allocation a little bit so we might need some sort of flow manager.

But we know this fully connected graph stuff isn't going to scale very well. We need something that's better and less expensive.

Hey, this is the 21st century! Everything is wireless! That would be much cheaper!

Well, perhaps remember there is much more sand in the world than spectrum. But let's look at a wireless solution.

N Systems Using Wireless



This is certainly cheaper. Instead of each node having (N-1)-interfaces, now each node has one interface.

In the previous step, we had (N-1)-layers of each with two members. Now we have one layer with N members. One nice property of this is that to the users of the layer, each node appears to be one hop away. If, internally, the layer is working over shared media, they are only one hop. If it is relaying, it could be more. In essence, relaying is simulating a shared medium.

Introduction

Whenever there is a layer with more than two members, because the layer operates either over shared media or over relays, we need some sort of “label” on the PDUs so that the members of the layer recognize that a PDU is for them. With shared media, every member will see every PDU sent. If the layer relays, the “label” indicates which node it is for. The protocol will carry the source and destination “labels.”

Dest Addr	Src Addr	Dest-port	Src-port	Op	Seq #	CRC	Data

The Costs of Wireless

Wireless comes with a cost, almost all of which derives from it being a shared medium.

Each frequency band has different propagation characteristics with regard to the distance, the material it will pass through, etc.

Wireless is a notoriously hostile environment for corrupting PDUs. We will need to ensure that most PDUs that

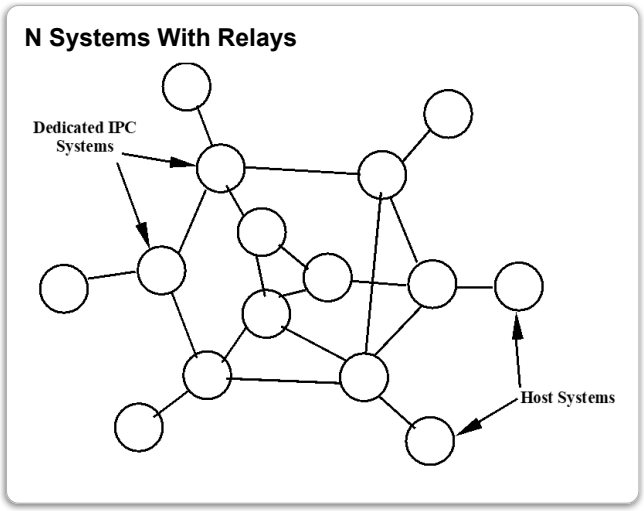
are passed up are not corrupt.

Most nodes can hear every transmission, and a collision will trash both PDUs. Only one node can transmit at a time, which greatly limits the capacity of medium. The more members of the layer, the more frequently there will be more contention. There are methods that get utilization beyond the basic 36%, but contention is still an issue and greatly reduces the capacity of the layer.

Wireless isn't going to scale well to large networks. Consequently, wireless tends to be used at the periphery of a wired network that relays. But we would like to keep the property that makes all members of a layer appear directly connected (one hop away).

So, what about N systems connected with relays? Hopefully, it is cheaper.

Communicating on the Cheap. N Systems Connected With Relays



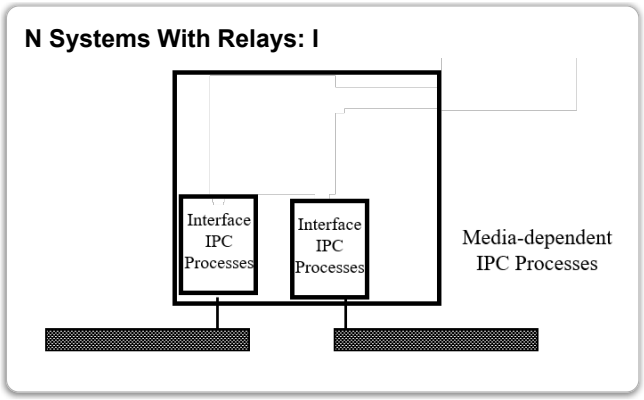
Computers have been falling in price and are now pretty cheap. Let's dedicate some machines to relaying so that we don't need quite so many wires, and maybe we can create more resilency at the same time. For greater efficiency, this can also take advantage of the fact that not every node talks to everyone else the same amount, which can either mean less expensive lines of lower capacity, or excess capacity that can be used by others. And there is still good resiliency to failure.

If we are going to have these relays, we don't have to have a wire from every system to every other system. By dedicating systems to relaying, we can reduce the number of lines required. For example, this has 6 hosts, 9 relays, and 19 lines (which are shorter), whereas the N systems directly connected case in the previous section would need 30 lines for 6 hosts. A more realistic example might be Tanenbaum's Figure 1-14 c) of the ARPANET in March 1971, where there were 19 lines, 15 hosts, and 15 switches. The directly connected approach would require 210 lines for 15 hosts.

Building Relays

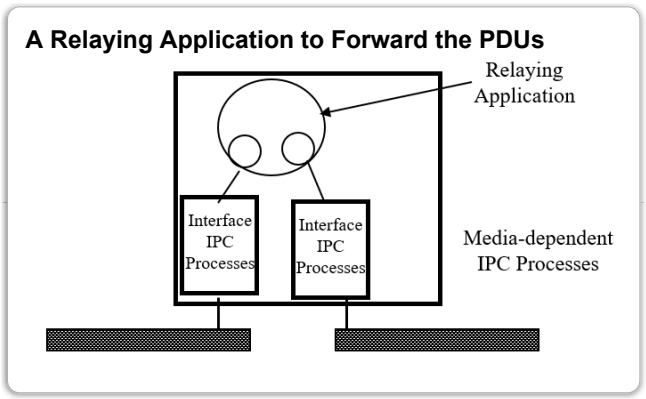
To do this, we will need systems dedicated to relaying and multiplexing. We will have one of our IPC layers for each wire connected to our relay system, and a relaying application that will take incoming PDUs and determine what interface each PDU should go out on.

A relay will consist of one of our layers for each interface (“wire”) that is connected to it. The media may be a variety of technologies with different characteristics.



Each interface layer will use a protocol suitable for that media. If the media is a point-to-point wire, labels won’t be necessary; local ids will suffice as we have used them before. If the media is a shared media, e.g., wireless, labels will be needed that are unique among the members of the layer.

Each “wire” may be a different technology and will have different error characteristics. We want to limit media errors to the smallest scope reasonable. We should keep errors from the media as low as practical.



Names!? What’s with Names!?

Yes, I have been purposely obtuse. These are commonly called addresses. However, as we will see later, their use caused the term to lose its meaning. There are three different senses these “labels” have been used in networks:

1. In shared media like wireless, each node simply has to recognize its label.
2. In small networks, labels are simply assigned by enumeration or seemingly randomly.

3. In large networks, labels are assigned with information embedded in them that gives an indication of which ones are near each other, for some sense of “near”, commonly relative to the graph of the layer.

The first are simply *identifiers*. We might call the second, “*forwarding-ids*," and only the third are really *addresses*. Addresses are location-dependent and route-independent. Again, where “location” is relative to the graph of the layer.

Forwarding

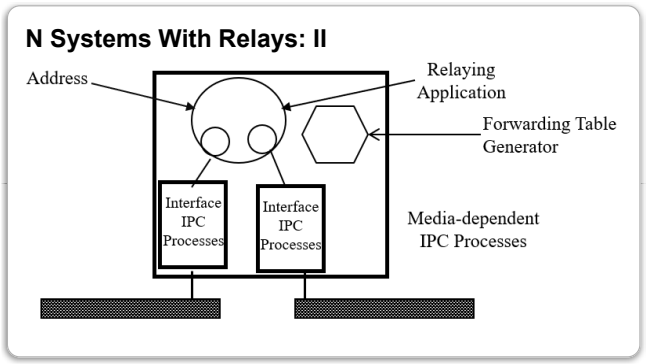
But how does it know which way to send the PDU? Good question.

A wire from a host no longer goes to just one destination but is on the path to many destinations.

We are creating a layer with more than two members that simulates a shared media. We must assign labels to all end systems and to the relays. The protocol for this layer will carry the source and destination “labels.” The relaying application will use the “labels” to determine:

- is this PDU’s destination here or
- does it have to be forwarded and if so, where?

It would be inefficient to just forward every PDU to every node to see who it is for. Traditionally this has been called routing, which exchanges information among the members of the layer on connectivity. There are algorithms for computing efficient paths through a graph. We can use these to generate a forwarding table for the relaying application to use. If the incoming PDU is not for it, it will look up the destination address in the forwarding table to find what interface to forward it on.



New Problems

But relays create new problems. (There is always something!)

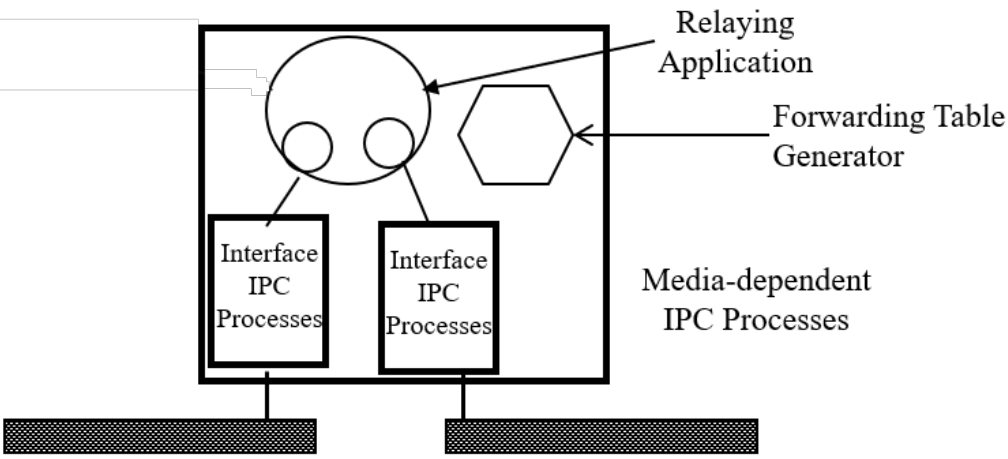
Relaying systems create some problems we can’t avoid. Congestion can occur from time to time when too many PDUs arrive at the same time. Often, it will resolve on its own, but sometimes action is required, and if the congestion gets very bad, PDUs will be discarded. And, very rarely, there are annoying bit flips in memories while PDUs are being

relayed. The PDU is protected in transit to the router when it is sent, but while it is in memory, errors may occur. To catch those errors, we will need an Error and Flow Control Protocol over the top of the whole thing.

N Systems With Relays: III

Dest Addr	Src Addr	Dest-port	Src-port	Op	Seq #	CRC	Data

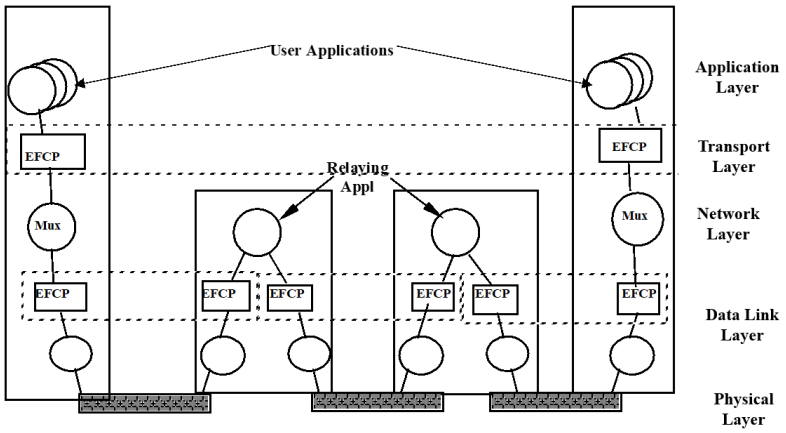
Typical PCI for an Error and Flow Control Protocol With Relays



What Have We Constructed?

For the most part, this lays out logically what is needed for a network. Now, if we step back and look at what we have constructed, you are going to see something that looks familiar.

The Big Picture



However, this is halfway between a bead-on-a-string model and a layered model.

Many of you are thinking: “Come on! You just wasted a lot of time re-deriving the same old five-layer model!”

No, we didn’t.

What we have come to here is something very different.

The IPC Model (A Purely CS View)
met_cs535_mod1_lec2_slide42_animation video cannot be displayed here. Go to the module page to watch.

What we’ve found is that there is one type of layer and it repeats.

All the layers do the same functions but for a different range of the problem. **Networking is IPC and only IPC.**

The only functions that go in a layer are those that are required for IPC.

Layers are not divisions of functionality but divisions of resource allocation. All layers have the same functions, they just operate over a different range of the problem with parameters specific to that range. Layers are distributed resource allocators, a distributed applications that do IPC.

A distributed IPC facility consists of multiplexing, Error and flow control and layer management, where the routing and resource allocation is done. Layers are configured to operate over a given range of data rate (capacity), quality of service, and scope.

Application names and port-ids are the only externally visible identifiers that the application should ever see. Port-ids have only local significance. The only information an application must, and should, know to establish communication is the destination application name and the port-id returned by IPC.

We have uncovered two principles and a couple of observations:

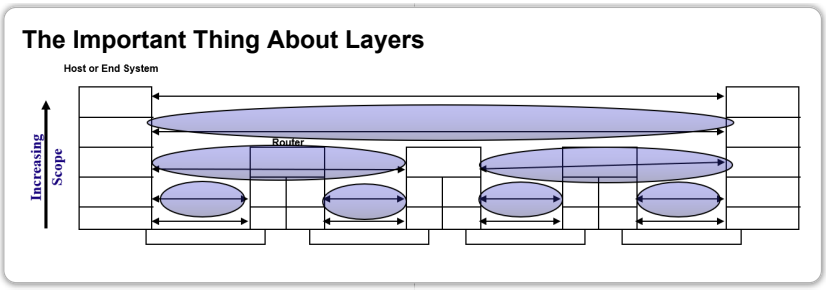
- When more than one flow between two systems is possible a connection-id is required.
- When a layer has more than two members, PDUs require names for the IPC processes, and their scope is the entire layer.

We also found that:

- The first data to send was not the application's data but data required for internal coordination of the layer.
- It is interesting how far into the construction we had to get before there was a requirement for another identifier with greater scope than a single system.

Networks are required to operate over a range of five or six orders of magnitude. There is not one set of policies that can cover that range. There is no "one size that fits all" solution. Layers are distributed resource allocators. Different layers are configured to best serve a given range. (The reason you may have been told that layers do different functions is because Dijkstra said that about operating systems in 1968. In the very early days of networking when things were simple, it looked like that was also true of networks. Dijkstra also said that functions don't repeat. At first, it looked like the layers had different functions, but that didn't hold up as we explored further. We kept finding that there were cases where more than one layer did fragmentation/reassembly, error detection, flow control, etc. It turns out that functions repeat in operating systems too. An OS has scheduling, memory management, and IPC at the kernel level, the supervisor level, and the user level (threads and heap). Was Dijkstra wrong? Not really. Functions don't repeat *over the same scope*. Systems were so constrained in 1968 (and in the early 70s) that he couldn't see that there would be different scopes. But in the early days of CYCLADES, Hubert Zimmermann did see that scope was the important property of layers in networks.

This is why the scope of the layers is important.



There might be a protocol doing lost PDU detection and error detection, and then we find it again at an upper layer.

Why?

Because those functions *have not been done over that scope* and there are new sources of the errors. That isn't redundant at all.

In fact, we should have seen this before, because if you really look at what the OSI seven-layer model looked like, it was saying there was a repeating structure that was right there in front of us. But we were so focused on the

differences, we didn't see the similarities.

The Repeating Structure

met_cs535_mod1_lec2_slide45_animation video cannot be displayed at the printable page. Go to the module page to watch.

From this, we can arrive at some rules for layering.

- The locus of shared state of a given scope constitutes necessary and sufficient condition for a layer.
- There may be multiple layers with the same scope, but if there are, functions within the layers must be independent. (If this occurs, it usually turns out that it is not a requirement of the problem, but something we are imposing.)
- Functions do not repeat in the same scope.

We are going to use this IPC model as a framework for what we do in this course. Everything we see will follow from this: the principles of protocols, enrollment, security, mobility, multihoming, etc. As we consider each topic, we will start by asking, "what does the model tell us?"

The model and what we learn from it will serve as a basis for analyzing existing solutions. It will allow us to step outside and see better what is going on. But there are three other results that we need:

1. The nature of applications and their relation to IPC
2. The separation of mechanism and policy
3. Richard Watson's fundamental results on synchronization

We will pick these up as we go through the course.

Practicing What We Preach

For a subject we thought we knew, this is not a bad result. In fact, it's a nice little return on investment.

However, the real lesson here is that we should practice what we preach.

This is an example of the advice we give every freshman engineer: When you do your work, write down every step. Don’t skip steps. If you skip steps, you are going to make mistakes. I know you think you know what you are doing. You don’t! Write the steps down!” We all had to learn that the hard way, right?!

There is nothing magical about what we just did. It’s not rocket science. There are no new blinding insights. Any of us could have done that exercise anytime over the last 40 years. There were many times that others (me, included) wrote about how making a phone call or sending a letter was like going through the layers. But we never did it for computers with the same care.

Why? Because we knew what we were doing. We didn’t.

The lesson is that, as we all become more experienced, we should never forget that “habit of mind,” that disciplined thinking. None of us are that smart.

Boston University Metropolitan College