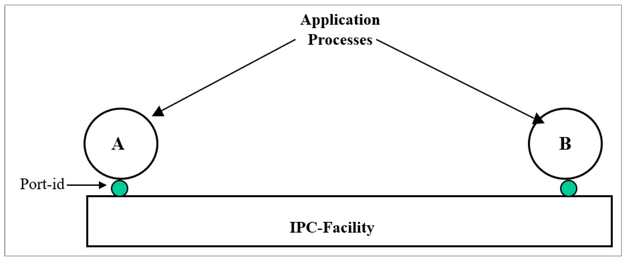
Assembling What We Have

Introduction

So far, we have investigated the physical characteristics of the media, the elements of data transfer protocols between two computers, and the requirements for when there are more than two applications communicating between two computers. We have climbed a ladder and reached a plateau. Now is a good time to stop and look at what we have found so far. What we learn next may enable us to throw away a ladder or two.

We have been going through the elements of networking from the point of view that Networking is IPC and only IPC. This is where we started in the second lecture of Module 1.

When I first presented this, I told you to sit back and take it in. You weren't expected to understand everything. That was what the rest of the course was about. This would give you an overview of where the course was going. We have now reached a juncture in that journey. We have covered the aspects of reliable data transfer between two systems with no relays, although we began to touch on it in the last lecture. In the next part of the course, we will get into the impact of relaying and the nature of naming and addressing in a big way. The problems will become much more subtle and likely more outside your experience. So let us recap what we have found so far.



Two applications communicating in the same system.

We talked about what *InterProcess Communication was in a single system*.

Sloppy Notation

It is painful to write "=" here. This invites sloppy thinking. Terminology should facilitate clear thinking. Thinking is hard enough without notation encouraging sloppiness. "=" is a Boolean operator. Assignment is not equality. There was a time when the ASCII character set contained a left arrow, which is much better. In lieu of that, this should be ":=" or "<-", but not "=".

We looked in greater detail at how that worked:

i. We introduced the concepts of a port-ids (of local scope) as handles (like a filed descriptor) for referring to an allocated IPC and allowing the applications to have more than one IPC at a time.

ii. The API to interact with the IPC black box,

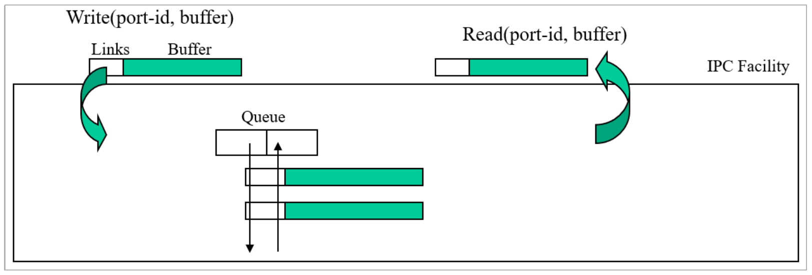
<reason> = Allocate (dest-appl-name, port-id, QoS)

<reason> = Read (port-id, buffer-ptr)

<reason> = Write (port-id, buffer-ptr)

<reason> = Deallocate (port-id, reason)

iii. We considered what went on inside the black box and found we would need some critical sections to coordinate the passing of buffers between the two processes.

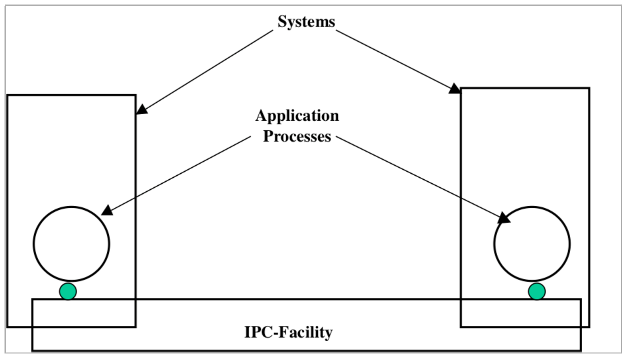


- A FIFO queue head is allocated for each port-id pair.
  - Memory allocation keeps links (which the user can't see) so buffers can be passed from queue to queue.
  - Lots of ways to Implement this.
  - But manipulating the queue requires a critical section, i.e., synchronization.
- Each Write causes 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.

All of this was more operating systems than telecom, which is as it should be. And then we saw how it worked:

1. **A** invokes the IPC Facility to allocate a channel to **B**; **a** <- Allocate(**B**, x);
2. IPC determines whether it has the resources to honor the request.
3. If so, IPC uses "search rules" to find **B** and determine whether **A** has access to **B**.
4. IPC may cause **B** to be instantiated. **B** is notified of the IPC request from **A** and given a port-id, **b**.
5. If **B** responds positively, IPC notifies **A** using port-id, **a**.
6. Thru n) Then using system calls, **A** may send PDUs to **B** by calling **Write(a, buf),** which B receives by invoking **Read(b, read_buffer)**
7. When they are done one or both invoke Deallocate with the appropriate parameters.

Two Applications Communicating Between Two Computers



Two applications communicating in the distinct system.

Then we took our first small step toward networking by making a small incremental change to consider what happens with *“two applications in two systems,”* and to do that, we had to develop. But that small step invalidated a lot of our assumptions:

1. The application name space was no longer under the management of a single system.
2. The requesting application had to be identified to the requested application.
3. All resources were no longer under the control of a single system, which we chose to ignore for the time being.
4. Bad things could happen to data being exchanged between the two systems,
5. We had to keep the sender from sending too fast for the receiver.
6. We had lost our means of synchronization. We needed a new means and it had to work in a hostile environment.

To compensate for these invalidated assumptions, we developed two protocols. In the introduction, we outline the basics of protocols. Here we dove deeper into them.

a. We needed an application name space that spans both systems, and the means to manage it. We saw that the names should be able to be location-independent, so the applications can be moved to a different system and still have the same name.
b. We defined an application protocol we called *IPC Access Protocol* (IAP) to request an instance of IPC with another application in a different system.
c. We went into much more detail about what an error and flow control protocol looked like. We applied a separating mechanism and policy to the following mechanisms:
   - Delimiting, including packing the user data field
   - Data corruption detection
   - Sequencing
   - Flow control
   - Lost and duplicate detection
   - Multiplexing
   - Retransmission control
d. And we applied Watson’s theorem as a robust means to create synchronization.
e. And while we were at it, we considered some of the practicalities involved in designing and building this class of protocols.

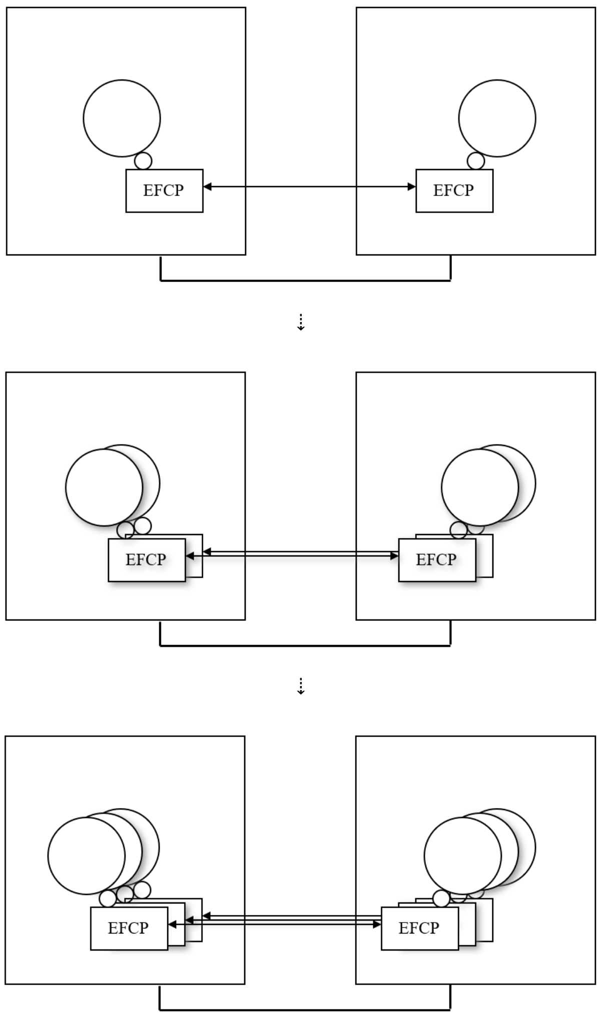
We looked at the lessons learned in the early development of application protocols. We saw some stellar successes and avoidable failures. We found the fundamental distinction between application protocols (modify state external to the protocol) and data transfer protocols (modify state internal to the protocol). We also found some important insights into the nature of the Application Layer and the importance of the application process/application entity distinction.

Watson's theorem stressed the importance of decoupling port allocation and synchronization, which parallels our distinction of IAP and EFCP. Given that, let us give IAP a name closer to what it is actually doing; we will call it the Flow Allocator and it will manage the flows in coordination with its apposite in the other system. We will consider the Flow Allocator in greater detail later in the course.

A major result here was the discovery that there is only one error and flow control protocol, which can be used to create a major collapse in complexity and create protocols that are defined to work together. Then we moved on to consider more than two applications between two systems.

More than Two Applications between Two Systems

Multiple Instances of the EFCP



Will have to add the ability in EFCP to distinguish one flow from another.

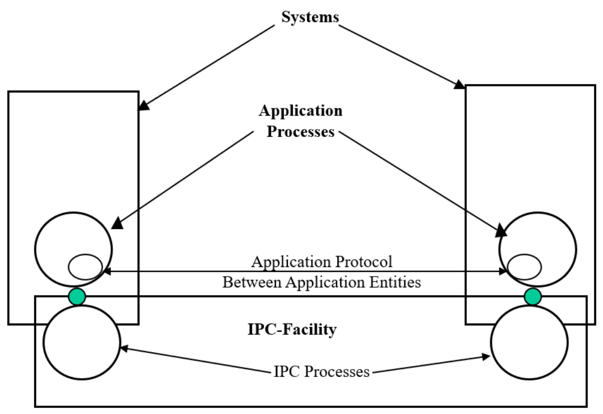
We immediately discovered that we needed to distinguish what data transfer PDUs belong to which connections. We needed some sort of connection-id. Because IPC could be initiated by either system, we needed to avoid the two systems picking the same value for the identifier. We solved this by simply concatenating the port-ids as a connection-id which is added to the PCI of the data transfer protocol.

Connection-id

Dest-port	Src-port	Op	Seq #	CRC	Data

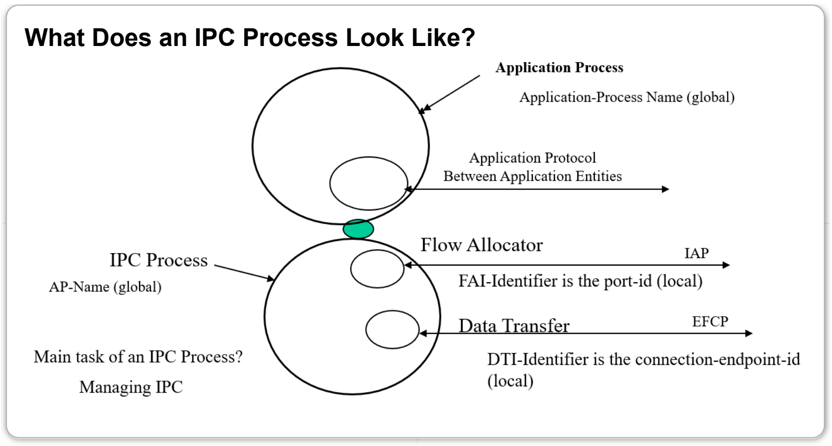
We have processes accessing IPC through system calls. What is going on inside IPC? There must be a process of some sort handling all of the stuff we have constructed so far. Just so it is easy to know which one we are talking about, the user of IPC, which we have called Application Process so far, or what is doing IPC, we will call these IPC Processes.

Back in Lecture 3, we discovered the importance of distinguishing the Application-Entity from the Application Process. It is an Application-Entity-Instance that is interacting with IPC. What is then going on inside the black box? Inside the IPC Process?



IPC Processes are simply Application Processes that do IPC. (There has been a habit in networking to consider the entities in the layer as somehow different than the entities using the layer. In reality, there really isn't any significant difference so we don't make one here.)

The system calls by the Application-Entity-Instance are delivered to the Flow-Allocator-AE-Instance, or FAI for short. Thus, we see that the FAI-identifier is the port-id, and that the Data-Transfer-AE-Instance-Identifier, or DTI-identifier, is the connection-endpoint-id. It is very nice how these seemingly disparate results all mesh so well. Nothing is contradicting our model. But we really haven't said much about the details of the Flow Allocator.


What Does an IPC Process Look Like?
Application Process
Application-Process Name (global)
Application Protocol
Between Application Entities
Flow Allocator
IAP
FAI-Identifier is the port-id (local)
IPC Process
AP-Name (global)
Data Transfer
EFCP
DTI-Identifier is the connection-endpoint-id (local)
Main task of an IPC Process?
Managing IPC

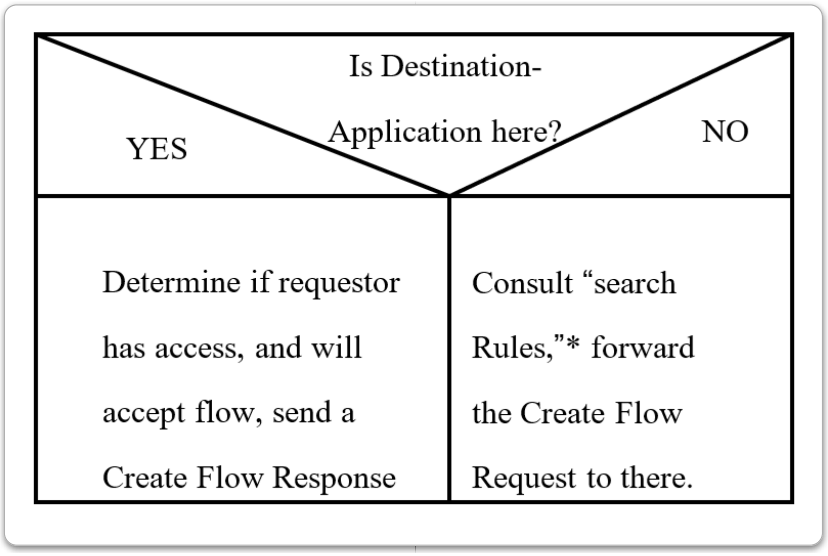
As we have seen, a flow is created when an Application-Entity invokes an Allocate service primitive. This is input to the Flow Allocator (FA), which, just as in IPC, in an OS will consult its search rules to find the requested application. If the application is local, then it will use the OS's IPC to determine if the requesting application has access to it and, if so, notify the requested application.

However, if the search rules determine that the requested application is not local, the search rules will suggest the next place to look for the requested application. This will require generating a "Create Flow" request and sending it to the FA indicated by the search rules.

Hence in general, an FA has two inputs: Allocate requests and Create Flow PDUs. In both cases, the FA consults its "search rules" and finds it is either local and further processing is required, or the "Create Flow" is forwarded to the next place to look. In this case, it is the Flow Allocator in the other system.

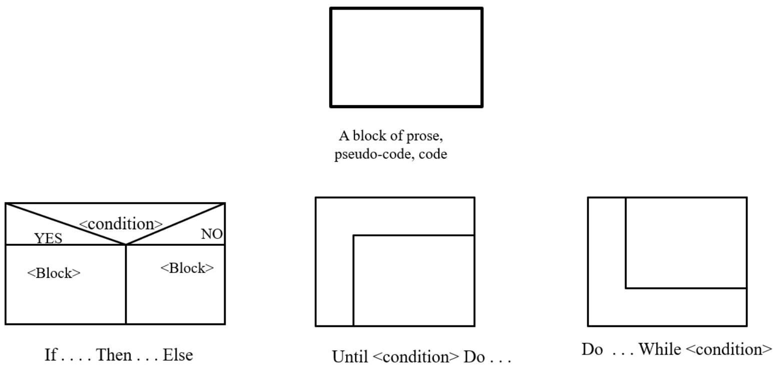
> Create_Request (flow object, destination-application-name, source-application-name, source-CEP-id, QoS parameters, policy list, access control)

The Flow Allocator basically has two inputs: local Allocate API requests and incoming Create_Flow Requests. It does the following:



Is Destination-Application here? YES	NO
Determine if requestor has access, and will accept flow, send a Create Flow Response	Consult "search Rules,"* forward the Create Flow Request to there.

One of Day's Asides (Nassi-Schneiderman Diagrams, 1972)


A block of prose, pseudo-code, code
<condition>
YES
NO
<Block>
<Block>
If . . . . Then . . . Else
Until <condition> Do . . .
Do . . . While <condition>

We have another one of Day's Asides. The diagram above illustrating the choice the FA needs to make was not idly chosen.

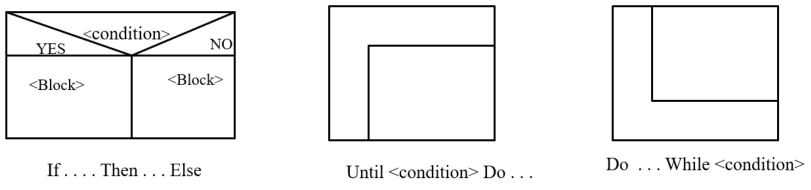
Have you ever done a flow chart?

I despise flow charts!!!

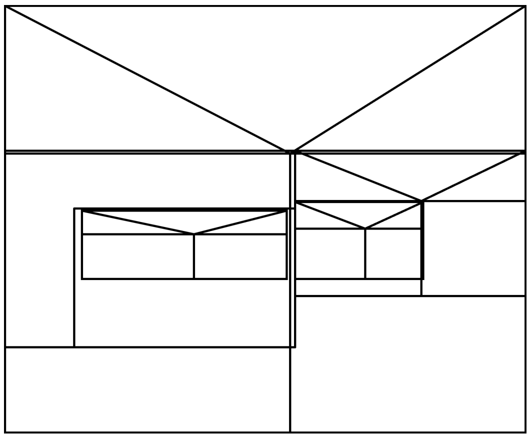
Flow charts are basically a graphical form of assembly language! I am not programming in assembler! I haven't used assembler since I was an undergrad! If I am going to use a graphical design notation/tool, I want something working at the same level of abstraction that I am programming!

Enter Nassi-Schneiderman diagrams from a short note in SIGPLAN in 1972. It uses a box. Write what is supposed to take place in the box. Use prose, pseudo-code, code, whatever. There are a few forms of these. The one I just used is, "if…then…else…" As you can see, the condition goes in the triangle and true and false actions are on the left and right, respectively. There are also forms for both kinds of loops, depending on whether the termination condition is first or last: the "For…Do Until" loop condition goes at the top, the body of the loop in the box, the "Do…While…" loop condition goes at the end, and the body of the loop goes in the box.

One of Day's Asides (Nassi-Schneiderman Diagrams, 1972)


<condition>
YES
NO
<Block>
<Block>
If . . . . Then . . . Else
Until <condition> Do . . .
Do . . . While <condition>

Now these can be nested to any level.



They nest.

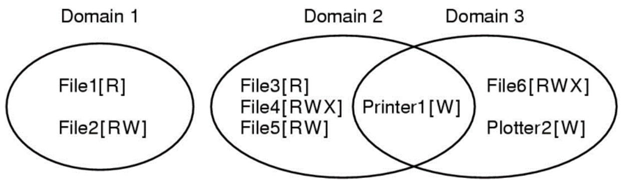
The idea is that one starts at a high level of abstraction with prose and then refines it perhaps with pseudo-code through a few iterations adding more and more detail until one is writing code. Notice there are no off-page connectors as with flow charts. Code developed this way is well-structured.

One thing I like is that one does a procedure per sheet of paper. Some young coder might say, “my procedure won't fit on a piece of paper!” To which the proper response is, “It is too big, kid, you can’t understand it. Decompose it!” The notation doesn’t let you write poorly structured code. The notation helps you do it right. It won’t let you do anything else.

OK, enough fun! Back to work!

When the flow allocator condition is true, we have found the IPC Process that has access to the requested application. The question now is, does the requesting application have access to it? A little refresher from operating systems on access control is warranted.

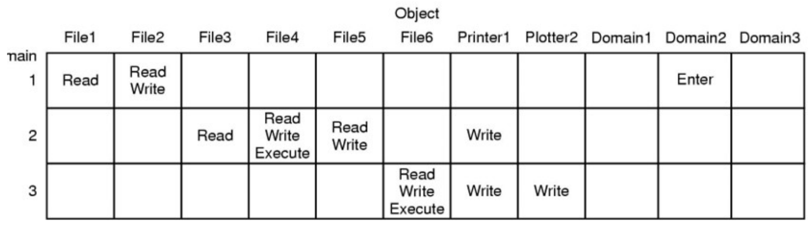
Protection Mechanisms: Protection Domains



Examples of three protection domains (objects, rights).

You know that we have protection domains and a protection matrix. In operating systems, the common permissions are read, write, and execute, and sometimes append. For us, we will want something like “accept IPC,” perhaps with a list of domains from whom IPC can be accepted. From this we can build a protection matrix, and we can even add the domains to the matrix.

Protection Mechanisms: Protection Domains

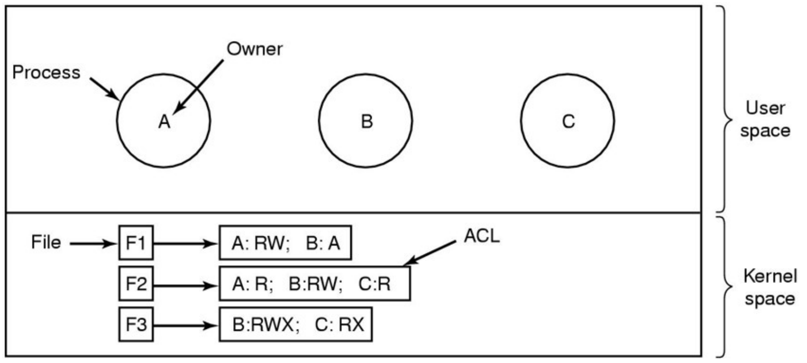


						Object					
main	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

A protection matrix with domains as objects.

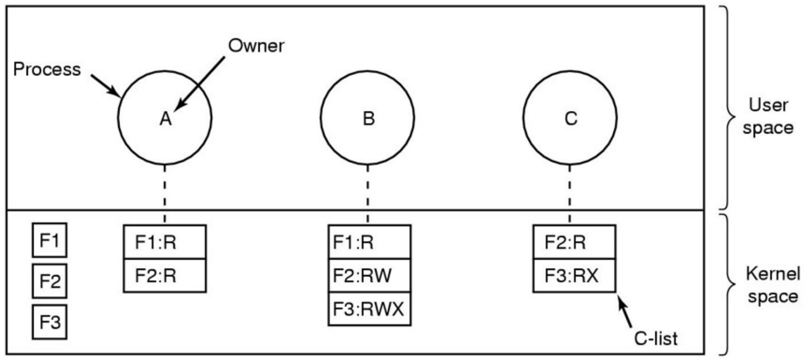
If we use Access Control Lists (ACLs), then we're reading the protection matrix by columns. For each object, who can use it and how.

Access Control Lists



If we use Capabilities, then we are reading the protection matrix by rows. For each user, this shows what objects they have access to and what sort of access.

Capabilities



Each process has a capability list; Protection matrix by row.

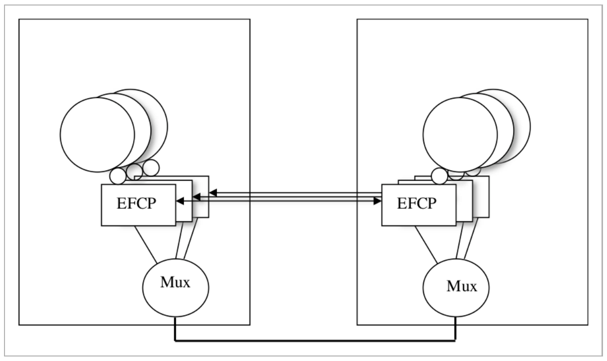
Access Control Lists are normally found in OSs, while Capabilities tend to be used in distributed systems. Or, Access Control Lists for fewer users with more objects; Capabilities for fewer objects per many users. (Domains are used for sets of users as a short cut.)

Just access control? Yes, all IPC can do is determine whether the requesting application has access to an application.

IPC cannot be sure the application requested is really that application. Only the source application can determine that. Authentication is between applications. The first operation an application should perform after successfully allocating a flow to the requested application is to authenticate that it is who it says it is. The authentication policy rests with the applications.

The other issue that this step creates is that with multiple applications all trying to send PDUs over the single wire connecting the two systems, there are *multiple users of a single resource*, the wire. This is the definition of an operating system function, i.e., a resource allocation function. We need to create a multiplexing task to ensure fair use of the resource.

Simultaneous Communication Between Two Systems, i.e., multiple applications at the same time

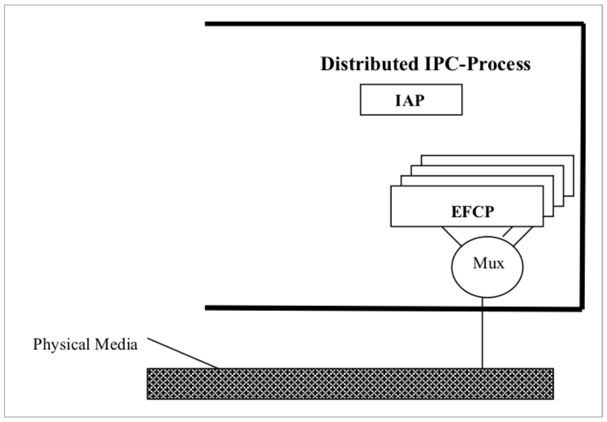


We will also need an application to manage multiple users of a single resource.

This is the other new concept needed at this step. The multiplexing task is like the scheduler for a disk drive or for the processor in an OS. It determines which of the PDUs queued to be sent will be sent next. The multiplexing task only handles the sending of PDUs as quickly as possible. The policy it uses to determine that requires much greater considerations and is handled by the Resource Allocator function of Layer Management.

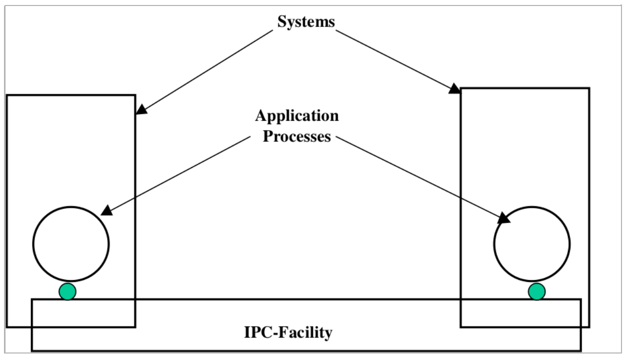
Then we found that we could group the functionality together as elements that are associated with one interface.

IPC to Support Simultaneous Communication Between Two Systems



Now this is how it works. From the user's perspective, nothing has really changed. But there is a lot going on inside the black box.

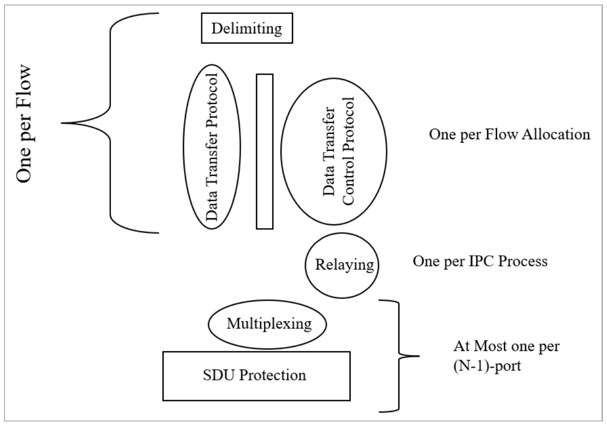
Two Application Communicating in Distinct Systems



Taking Stock of Where We Are

When we analyzed error and flow control protocols, we found that they naturally cleaved into two kinds of mechanisms: tightly bound mechanisms that had to be with the Transfer PDU (Data Transfer) and loosely bound mechanisms (Data Transfer Control) that didn't have to be with the Transfer PDU and that were coordinated through a state vector. There is one of these pairs for each flow that is allocated.

Data Transfer

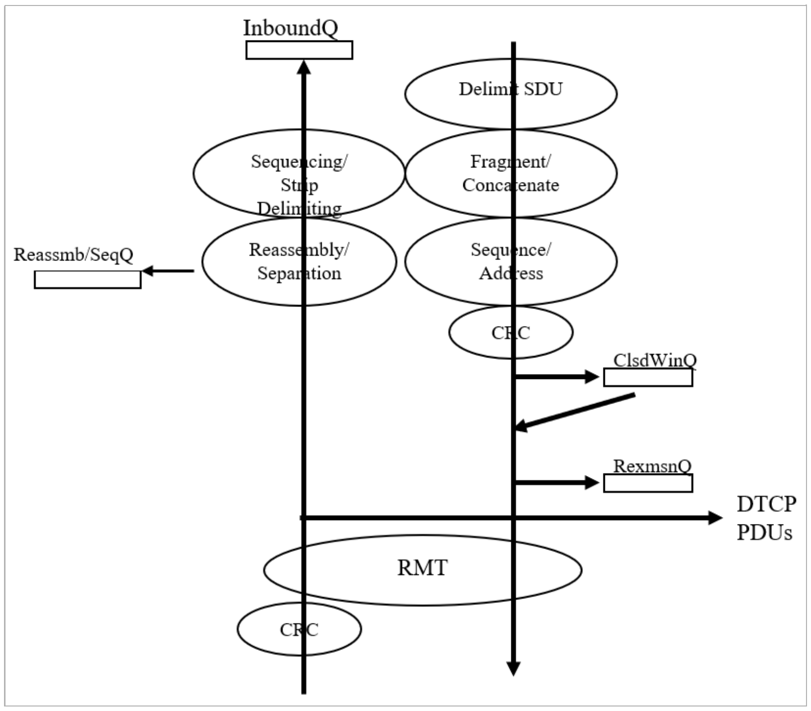


- More flows coming in the top than going out the bottom.
- Tightly coupled mechanisms in the Data Transfer Protocol.
- Loosely coupled mechanisms in the Data Transfer Control Protocol.
- SDU Protection must be common to the layer.

Furthermore, we found that the two parts barely interacted. Data Transfer wrote the state vector. Data Transfer Control read the state vector and generated Ack and flow control feedback. Once in a while, Data Transfer Control might tell Data Transfer to hold up sending (no flow control credit), but that is the limit of their interaction. In addition, the Data Transfer functions were simple and efficient to implement, which is consistent with fast duty cycle, while Data Transfer Control were computationally more complex and had a longer duty cycle. This is precisely what one wants to see in any system design. One does not want longer duty cycle functions intertwined with short duty cycle functions. We also found that there is one relaying or forwarding task per IPC Process, and one multiplexing task per (N-1)-port.

We noticed that while we laid this out logically, that doesn't mean we have to implement it this way. The architecture may describe it one way, but the implementation can do things in any order, as long as it isn’t visible outside the black box. For example, when a PDU comes in, the CRC has to be done immediately, otherwise we can't trust the bits. If the PDU passes the CRC test, the other functions can be performed. But when sending a PDU, we first delimit the SDU, perhaps fragment or concatenate, and add sequence numbers and addresses to the PDU. Once the PDU is completely assembled, we do the CRC. We don’t wait until just before it is sent.

Data Transfer Protocol



If the flow control window is closed, hang it in the closed window queue. When the window is open, the PDUs can be sent with no further processing. When the PDU is sent, put it in the retransmission queue. In either case, the PDU is ready to go; there is nothing left to do to it. If we have to resend it later, all we have to do is take it off the queue and send it.

When it comes to implementation, anything you can get away with is legal. The architecture, even the protocol specification, is not a design document.

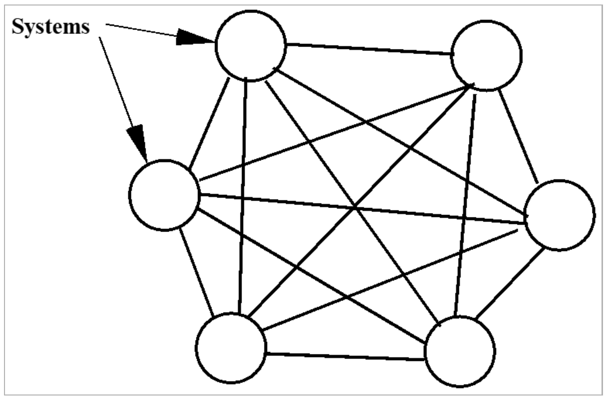
That's the basics of data communication. Now on to networking.

(It will turn out that DataCom is a degenerate case of Networking.)

Let us finish going through the IPC model. This is a preview of what' to come.

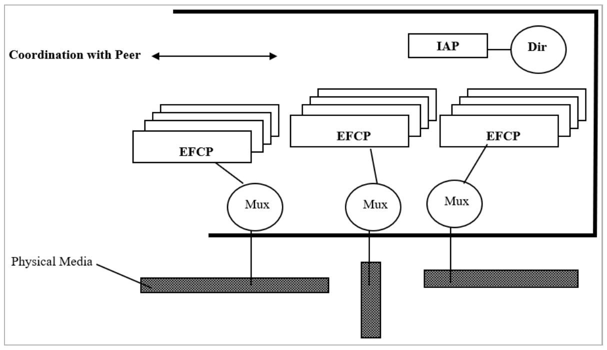
N Systems with M Applications Directly Connected

Communication with N Systems



This step is straightforward. We just take the unit we created in the last step, let's call it a *Distributed IPC Facility (DIF)* or *layer*, and replicate it for each wire terminating on a system. The main problem here is managing the proliferation of point-to-point layers.

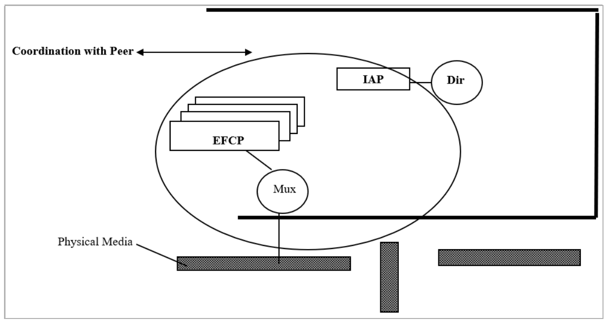
With Multiple Interfaces It Gets Messy



But we also need a task to manage the use of the interfaces and mask any differences. A little organizing will help.

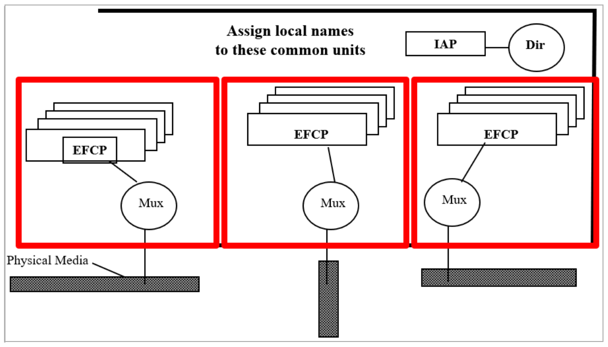
IPC can find the destination application by finding out what interface it is on (using the Flow Allocator to ask by sending a Create Flow Request). (The destination system is at the other end of the wire. There is nowhere else for it to be!) We might assign a local identifier for each interface to distinguish them. We also want to keep a database of what applications we found and where we found them, a proto-directory, and a variation on what we saw with learning bridges, only for applications. We can also use these local identifiers to keep track of which EFCP instances are bound to which multiplexing task.

Common Structure



We can do some organizing and re-factoring.

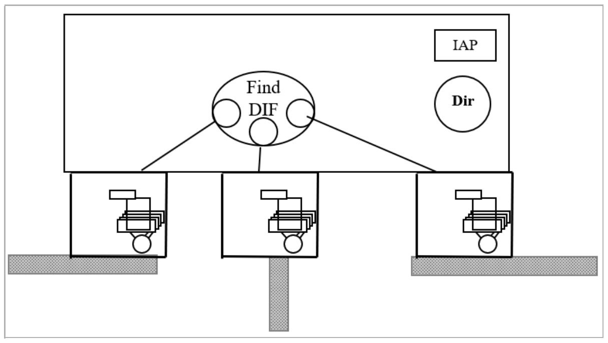
Keeping Track of What Application Is at the End of Which Wire



To maintain the API so that the application doesn't have to figure out which wire a requested application is on. (After all, in good OSs, one doesn’t have to know what disk drive files are on.) We posit a new Finder function that keeps track of what has been found where. Allocate requests come to it, and either already knows or sends Create-Flow requests on different interfaces looking for the requested application.

A Little Re-Organizing

A Virtual Facility?



So we have a Distributed IPC Facility for each interface and an application over all of them to manage their use and to give the user a common interface, a Virtual IPC Facility?

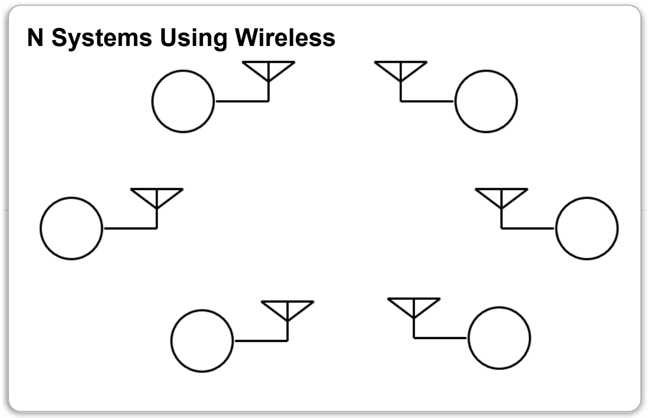
BUT

This fully connected graph is going to be pretty expensive, and it isn't going to be to scale. Everyone doesn't need to talk to everyone else or at least not the same amount. We need to do something a little cheaper.

Hey, it's the 21st century, what about wireless? Everything is wireless. Let's take a look. (But there is more sand in the world than spectrum.)

N Systems Using Wireless

Wireless is certainly cheaper. Instead of each node having (N-1)-interfaces, each one has 1interface. However, … (why is there always a 'however.')



Wireless does provide a marked decrease in complexity. Before each node had (N-1)-layers each with 2 members, now each node has 1 layer with N members. Now whenever there is a layer with more than two members, we need to put an identifier on each PDU to indicate which member the PDU is for, (sometimes mistakenly called an address). If the layer is a shared media (as this is), then every node will see every PDU. Hence each node will only need to recognize its identifier. If the layer relays, the identifier may be assigned such that it indicates where in the graph of the network it is without indicating how to get there. (This is the true definition of an *address*.) These identifiers will have to be unambiguous within the layer, which for wireless could be indeterminate.) Each PDU will have to carry the source and destination identifier.

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

The PDU must be expanded one more time.

Wireless introduces some new considerations. Wireless comes with a cost, almost all of which derives from it being a shared media:

It will have limited range.

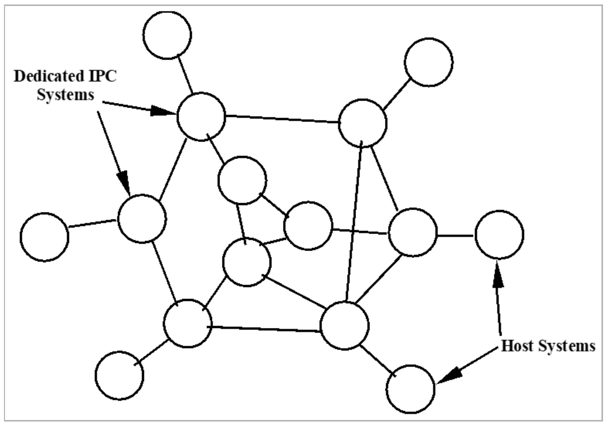
It is a notoriously hostile environment for causing errors in PDU or destroying them altogether.

There is contention for the media to send PDUs. There are methods that get utilization above the basic 36% but contention is still an issue and greatly reduces capacity.

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, i.e., one hop away.

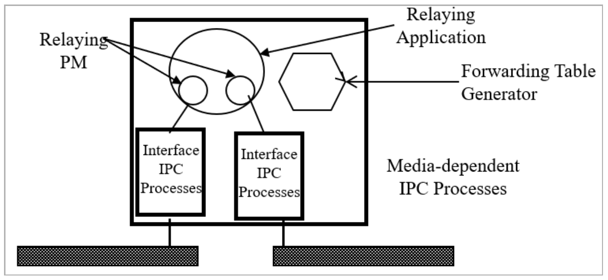
N Systems Connected by Relays (On the Cheap)

N Systems with Relays (On the Chaep)



Recognizing that computers are getting cheaper, we can dedicate systems to just doing relaying.
That means that will have a box that is just used for relaying. It can use what we've already developed. We will need one of our point-to-point layers for each wire terminating on our Relay System.

Common Relaying and Multiplexing Application Header



Introducing Relaying Systems

In addition, we will need some new elements: such as globally accepted identifiers for the source and destination systems and, also for the relays. The path to the destination is no longer point-to-point, not one hop. The relays will have to be told where the PDU is going. We will need to add more information to our PDUs in the form of the kind of identifiers we required for the wireless step we just did. If the layer is large enough, we can assign these identifiers to act like addresses. So that our PDU looks like this:

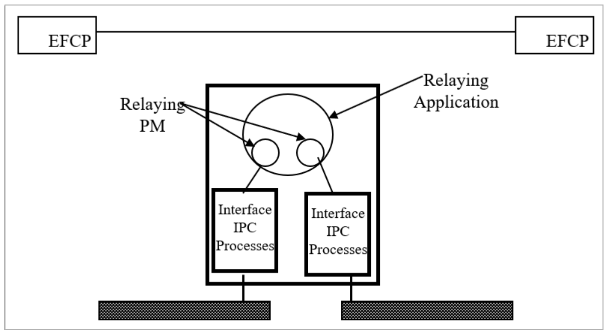
Dest Addr	Src Addr	Dest-port	Src-port	Op	Seq #	CRC	Data

This will require a relaying task that inspects the destination address, consults a forwarding table, and sends the PDU on the correct interface toward its destination. The relays will need some way to figure out given the address, where does it forward the PDU toward its destination. We will need a forwarding table generator. We will need to figure out how to route PDUs through a network.

Dealing with the Problems of Relaying Systems

But Relay systems can create problems too. We can't avoid transient congestion, from time to time with the loss of PDUs, and annoying bit errors can occur in memories during relaying. Therefore, an Error and Flow Control Protocol will be required operating over the relays between the source and destination.

Communications on the Cheap



Concluding Remarks

Now things will start to get complicated and over the next few modules. In the next module, we'll cover routing, and the traditional Network Layer.

Then we will discuss naming and addressing, which is probably the most important topic in the entire course. This topic is not covered by Tanenbaum or for that matter any other textbook in the field. These three topics are very tightly related, and it is impossible to do them without forward references. So, hang on!!