IDN-3

Process Organization

Part 1

Eric Rosen

August 1981

The following note is an attempt to begin to block out the process organization of the gateway, and to look at some of the problems of inter-process communications and resource management that arise. These considerations are only preliminary, but unless someone strongly objects that I am on the wrong track, I will use them as a guide for further thinking.

Let's look a little at the software organization of the gateway. Each network interface or access line will have to have a process associated with it (call this a "Pathway Access Process", or PAP), or maybe two, PAP Input and PAP Output. These processes will implement the access protocols of the individual networks to which the gateways are connected. In fact, maybe we should think of four processes per network interface: PAP Input Level 2, PAP Input Level 3, PAP Output Level 2, and PAP Output

Level 3. I leave it open for the present as to what the word "process" will actually mean in the implementation (though I am somewhat predisposed to think that what I am calling a "process" should be the same kind of thing that gets associated with a PCB in the NMFS system). The point is that our modules and interfaces and stuff should be clean enough so that we can freely intermix the possibly different Level 2 and Level 3 options of a given access protocol.

The PAP Input Level 3 module will pass each packet to a "Gateway Input" (GI) process for the particular access line (port) over which the packet was received. The GI process will have to decide, of each packet, which of four categories it falls into, and dispose of the packet on that basis. The four categories are:

a) Transit packets. These are packets for which this gateway is neither the source gateway nor the destination gateway. Such packets will be passed to a process which we might call "Dispatch" (see below.)

b) Internet control packets. These will be things like routing updates which need special processing by the gateway system. Such packets should be passed directly to whatever

process or processes handle them. Thus routing updates might be passed to a process which manages the topological database and performs the SPF computation, while being simultaneously passed to all GO processes (see below) for flooding to other gateways. IN some cases, where control packets need especially rapid processing (think of line up/down), GI might process the packets directly.

c) Internet Entry packets. These are packets from Hosts which are entering the internet system at this point. (Or more accurately, are entering OUR internet system at this point.) These should be passed to a process which we might call "Internet Arrival Process" (IAP).

d) Internet Departure packets. These are packets from Hosts which will depart our internet system at this point. These should be passed to a process which we might call "Internet Departure Process" (IDP).

You might wonder why we need a GI process, over and above the PAP Input Level 3 process. This process will tend to be responsible for stuff at the level of the gateway-gateway packet headers, such as delay measurements, processing received piggybacked acknowledgments, etc. Similarly, there needs to a a

Gateway Output process (GO) "above" every PAP Level 3 Output process, to set bits in the gateway-gateway header, etc. This way of looking at thing allows the PAP modules to avoid doing things which are specific to gateway protocols, and hence preserves protocol layering. Of course, as I said above, this sort of mental organization of the gateway software might or might not map directly to an actual implementation.

Internal Pathway up/down considerations (i.e., which neighboring gateways are reachable over which access lines) will be handled by an interaction between GO and GI. Access Pathway up/down considerations (i.e., which Hosts are reachable directly over which access lines) will be handled by IDP.

If a gateway is multi-homed to some network so that it has several access lines to that network, there are some subtleties. Each access line needs its own PAP Input and Output processes. We should probably take separate measures of delay to each neighboring gateway over each access line, as well as running a separate instance of the internal Pathway up/down protocol for each neighboring gateway over each access line. However, maybe we should have only one GO process and one GI process for each individual network to which the gateway is connected. The

processes above GO and GI would not know whether a particular Pathway consists of several access lines to the same network or not. The delay, congestion, and up/down information used by the routing algorithm would be reported by the GO or GI process for the composite set of access lines, and would be computed by these processes based on information passed up from the PAP processes. Dispatch (in the case of packets going further in the internet) or IDP (in the case of packets leaving the internet) would route packets to a particular GO process, leaving it up to the GO process to use its local knowledge to further route the packet to the most appropriate PAP Output process. The way in which GO computes routing information based on delays for each access line, and the way in which GO chooses from among several access lines to the same network is something I will think about further.

There may be may be cases in which there are Pathways to a neighboring gateway through several distinct networks. This seems like a bit of a problem for the process structure I am proposing. I'm gong to have to think some more about this.

The IAP process will look at the internet leader supplied by the source Host, and use it to construct the internet header

(remember my distinction between external leaders, used for protocol between Host and gateway, and internet headers, used for protocol among gateways). IAP will also handle the flow control and connection block stuff that I have discussed in previous messages. IAP may decide not to accept a packet, in which case it may create an error message or NAK and pass it to the GO process for the network from which the rejected packet was received. (Note: such messages need not go out the same port or access line from which the rejected message was received.) Such messages should probably be returned by using the addressing information which can be gathered from the Pathway Access Protocol envelope, rather than by using the addressing information which can be gathered from the Internet Protocol envelope; hence the former information must be passed to IAP from PAP Input Level 3 (via the GI process).

If IAP accepts the packet, it should be passed to the dispatch process.

The purpose of the Dispatch process is to decide what the packet's next hop is. That is, Dispatch looks up the next hop for this packet in the routing tables, which will be indexed by destination gateway and type of service, both of which can be

found in the packet header. When the next hop is chosen, Dispatch must pass the packet to PAP Output Level 3 for transmission to the next gateway.

It is possible that Dispatch will find that the destination gateway is not reachable. (This would be indicated in the routing tables.) IN this case, Dispatch should attempt a retranslation of the destination Host address, to see if there is another destination gateway to which this packet can be sent. (This is possible if the destination Host is multi-homed to a number of gateways.) If so, the header must be altered, and the packet passed to the appropriate PAP Output Level 3. If not, the packet can be discarded. I don't think it is necessary to return any error message or NAK to the source host in this case.

I assume we will also have some number of "internal hosts" ("fake hosts" in IMP parlance). These internal hosts will submit and receive packets to the gateway proper using the usual internet access protocol, but no additional Pathway Access Protocol is needed. Hence the internal hosts can submit stuff directly to IAP and receive stuff directly from IDP. At least some of the internal hosts will implement higher level protocols, for monitoring and control purposes. Otherwise may deal in "raw

messages," and not implement any higher level protocol. We might want to give certain internal hosts particular options that are not generally available to users, such as freedom from any congestion control constraints.

We will probably want to consider designing some of our internal hosts as "test nodes." (This idea stems from a suggestion of Paul Santos.) That is, one way to test out our implementation of particular Pathway Access protocols would be to implement both the host side (the PAP processes) and the network side (which would be in the test node). At its simplest, the test host could serve as a sort of software loopback facility. (Many network access protocols, unlike 1822, are non-symmetric, so they can't be tested by simple hardware loopbacks.) If we are dealing in a connection-oriented network access protocol, the test host could participate in setting up connections. We could program the test host to send the gateway all of the strange control messages that networks are likely to send hosts during general operation, but which they never send when one is trying to test out host software.

Of course, in addition to these "event-oriented" processes, there might be any number of timer-based processes to handle

various periodic or housekeeping functions, as well as lower priority background processes. The processes mentioned so afar are not meant to be an exhaustive list.

Needless to say, this model of the gateway software architecture is based on the way the IMP is organized. If anybody has a fundamentally different picture, please speak up.

I have spoken repeatedly of one process "passing" a packet to another. What I have in mind is the former process executing a system call which (a) placed the packet on a queue for the latter process, and (b) causes the latter process to be scheduled (according to its assigned priority). This call should accommodate priority queuing with no hassles. Sometimes, processes may need to pass information to each other. This can either be done through commonly accessible data locations, or through the sending of messages. CMOS tends to encourage the latter, I think. I tend to prefer the former, since once the locking problems are worked out (which may not be that difficult in a properly structured system) we don't need to use any dynamically allocated resources like message blocks to do inter-process communication. That means that we don't have to worry about crashing the system because we have run out of such

blocks. We'll have to consider this very carefully when we get down to a "lower level" in the design.

Incidentally, to make some of this "level 2" and "level 3" talk more concrete, here are some examples of stuff that we might have to handle:

ARPANET access

Level 3 -- 1822 Protocol NAAP (New ARPANET Access Protocol), which is basically 1822 but has logical addressing and some non-blocking features.

Level 2 -- 1822 (local or distant host) VDH HDLC

SATNET access

Level 3 -- Host-SATNET protocol

Level 2 -- Same choices as for ARPANET

Actually, I don't really know how many combinations of level 2 and level 3 possibilities we will actually have to handle, but I think our software architecture had better make it simple to add in new combinations.

Of course, the IAP and IDP processes will no doubt have to handle several versions of Internet Protocol simultaneously.

With data moving among all these processes, we have a number of resource management problems, particularly buffer management.

It is possible to provide some general principles of buffer management, but each new combination of protocols that we might have to deal with has to be considered individually, and its buffer management implications understood. In cases where we try to borrow canned software from other projects, we will have to look carefully at the software to see if it fits properly (or can be made to do so) with our buffer management system.

I can't produce a detailed buffer management spec at this time, but I would expect our buffer management system to be very similar to that I spec'd out for the IMP in the message I sent a few days ago. That is, we might have a transit pool, to hold those packets which move directly from GI to GO, and an end-end pool to hold those packets which move from GI to IAP or IDP. The buffers should also be organized into sub-pools to assure that each output device (GO process) can always get hold of a certain amount of buffers. Each input device (GI process) should also have a certain number of buffers assigned to it, to allow input with minimum latency. (We have to remember to save some buffers for internal host input, which bypasses the GI processes.) Since host input and output is much less tightly coupled in the gateway than in the IMP, we might want to separate the IAP and IDP pools

entirely; I'm not certain yet of all the implications of this.

As packets move from process to process, we will want to make sure that no receiving process ever "blocks" the sending process. That is, all packet movement should be non-blocking. I don't mean by this merely that system calls which queue packets for the next process should never hang indefinitely, but also that the receiving process should be ready to accept responsibility for a packet as soon as the sending process is ready to give it up. The receiving process can discard the packet, or queue it, or process it, or transmit it, as it sees fit, as long as this doesn't cause the sending process to back up. This sort of scheme is necessary to minimize unpleasant interactions among processes. By extension, the gateways also ought not to block the individual networks by refusing to take packets from those networks.

Of course, any rule may have exceptions, but I think there should be a presumption in favor of the above principles. Such principles are difficult to adhere to unless each process that needs a pool of buffers is capable of getting a large enough pool to suit its needs most of the time; I am assuming that the gateway will not be chronically buffer-short.

It is important to realize that all buffer management information should be "globally" available. Buffer management should not be done invisibly by the system calls that move buffers from queue to queue; rather, the application needs full access to all information about the status of the buffer system. In particular, any process should be able to "charge" a buffer to any pool. For example, suppose a particular packet must move from process A to process B to process C, and that at the time process A sees the packet, it is already known that it will eventually get to process C. Then process A should make sure that the buffer gets charged immediately to whatever buffer pool is associated with process C. And when process C finally receives the packet, it must have a way of knowing that the packet is already charged to its buffer pool, so that it doesn't get charged a second time (as in the IMP's "double counting" bugs). The need to make all this information accessible to all processes may have implications for the design of the system calls or subroutines that we use to move packets around; it certainly will have implications for the design of the buffer header formats.

Later on in our design effort I'll spec this out much more

precisely. Now I'd like to give a couple of concrete examples of the applications of these rules. I said above that the gateways should never block the networks by refusing to take packets from them. Let's see how this applies in the case of the ARPANET. Each 1822 interface should have a buffer dedicated to it which is long enough to take an 8-packet message from the IMP. When a message is received over the 1822 interface, the PAP Input process should try to get a free buffer (of maximum message size) to put up immediately for the next input. If such a buffer cannot be obtained, then we should process the message ONLY if it is an internet control message (routing, up/down). If there are any piggybacked acks, or anything of that nature, we should process them. Then we should discard the message, and reuse its buffer for input. While this causes some loss of data, it is necessary so that we can still receive and process internet control messages, even when we are out of buffers for handling data messages. It is also necessary to prevent being declared tardy.

Since we do process internet control messages, there may be times when we can't get enough buffer to read in a full 8-packet message, but we can gets enough buffer to read in smaller

messages. I think that in this case we should still accept input from the ARPANET. If the input is too long, we should pull it through and throw it on the floor. This will at least enable us to receive control messages (which are likely to be relatively short). If this situation persists for any amount of time, the gateway which is sending to us over the ARPANET will end up thinking that the delays to us over the ARPANET are very long, and also that the Pathway to us is congested. This allows the routing and congestion control procedures to take effect and reduce the amount of traffic sent to us.

Another example: suppose that IAP sees a packet, and decides, by looking into the appropriate connection block, that it cannot send that packet yet because of congestion control restrictions. It makes sense for IAP to maintain a pool of buffers to use for holding such packets internally, until the congestion control procedure allows the packets to be sent through the internet. However, when this IAP pool is filled, IAP must discard any further packets which it cannot send immediately. Any such packets must not be allowed to back up into the buffering needed to handle 1822 input.

Another example: we will want to limit the number of packets

that can be queued to a single access line, so that one slow network does not eat all the gateway's buffers, thereby preventing I/O from/to other networks. Further packets that arrive at the GO process will have to be thrown on the floor. If the packets are transit packets (from a neighboring gateway), this should cause the neighboring gateway to see a high delay to this gateway. This gateway should also reflect these discards in its determination of the congestion status of this access line. In the case where the discarded packet is an IAP packet, we should probably send a NAK to the host, IF we can get a buffer to do so.

Note that a number of neighboring gateways may be reachable over a single network access line. That is, a gateway on the ARPANET may reach each of five neighbors over a single 1822 access line. In this case, we may want to reserve a certain amount of the buffering reserved for that access line for each neighboring gateway. Since the number of neighboring gateways may vary dynamically, this might be a little tricky.

It is interesting to consider the extent to which this nodal software architecture can be considered to be "layered." I have tried to set it up so that it is possible to "mix and match"

protocols at different layers more or less arbitrarily. That is, a process that handles a particular protocol should not need to know what protocols are in effect either above or below it. However, all processes must be cognizant of certain global or system-wide requirements and goals. In particular, every process may have buffer management responsibilities which accrue to it through its role in the system, and which therefore cannot be specified for that process when considered in isolation from other processes. Further, higher level protocol processes may need to know information (such as the source address of a packet in the local network) which is contained in the envelopes of lower level protocols. If we can produce an implementation following these guidelines, we have a chance of producing a system which preserves protocol layering but which does not introduce the inefficiencies and insularities that many excessively layered systems seem to have. However, producing this sort of implementation will be quite tricky, and to do it we will have to have rather extensive design review to a quite low level. Also, we will have difficulty trying to incorporate "off-the-shelf"software modules for various protocols (e.g., someone else's HDLC code) without modification, since an

off-the-shelf package wouldn't be likely to take account of system-wide resource management t issues.