IDN-16

Inter-process Communication Involving Routing

Eric Rosen

March 1984

In this note I want to discuss the issues involving inter-process communication between the routing process and the other processes in the gateway. There are several instances in which routing updates, or at least the information therein, needs to be passed from process to process:

- Routing updates (or update acknowledgments) arrive at the gateway, having been flooded from neighboring gateways. These updates must be delivered to the ROUTING process, to be incorporated in the topological data base and to initiate a recomputation, if necessary, of the shortest path tree(s).

- Once it is determined that an incoming update needs forwarding to neighboring gateways, it must be suitably addressed and delivered to an appropriate output process or (more generally) processes.

- Updates generated by the gateway itself (say, by the neighbor up/down process) need the same sort of treatment as incoming updates.

- Routing updates received from neighbors need to be acknowledged.

- After the routing process computes its routing tables (i.e., the tables which map exit gateways to neighboring "next hop" gateways, and which map destination nets to exit gateways), these tables must be made available to other processes. In general, there may be several sets of tables, corresponding, for example, to different types of service.

Let's look at the most straightforward possible ways of handling these things, and consider whether they are suitable, or whether they require further refinement.

INCOMING UPDATES

The simplest way of handling incoming updates is to have all the input processes treat them the same as any other ordinary packet, running it through all the input protocol processing, and

then queueing it for DISPATCH. When the packet comes to the head of the DISPATCH queue, DISPATCH then queues it for ROUTING by placing it on the ROUTING Queue (RQ).

Unfortunately, if the DISPATCH queue is very long (perhaps due to poor routing caused by slow updating), or if the input protocol processing imposes delays, this can cause unnecessary delay in the processing of the update.

Input processing delays can be largely sidestepped if the input driver process can recognize that a certain packet is a routing update, and can place it directly on RQ. Of course, the packet would also have to be queued for the LEVEL2 protocol process. However, when the input protocol processing is finally finished, the packet can be discarded, instead of being queued to DISPATCH. (We know that the use count can be made to ensure that the buffer does not really get freed until ROUTING has also disposed of it.) Update acknowledgments would be treated in the same way.

One doesn't want to miss updates or acknowledgments because of buffer management considerations. That is, one doesn't want

to discard updates or acks before processing them. So the buffers on the ROUTING queue should remain charged to input, just as if they had been placed on the DISPATCH queue.

This procedure introduces an apparent sub-optimality, stemming from the fact that not all of the received updates actually require processing. Some may be duplicates of updates already received (a very common occurrence, due to the flooding algorithm), and some may be out-of-date. If a neighboring gateway is running wild somehow, it may fill all one's buffers with duplicates of the same update. If we could devise a scheme to ensure that the ROUTING queue never contained more than one update from each gateway, we wouldn't have to worry about wasting a lot of time and space with duplicates. Such a scheme would ensure that duplicate or obsolete updates never actually get placed on the ROUTING queue.

To tell whether a given routing update needs processing, or can just be discarded as a duplicate, one needs to compare the sequence number and age of the given update with the sequence number and age of the last update from the same source gateway which was actually placed on the ROUTING queue.

Unfortunately, with several updates from the same source gateway arriving over various interfaces, making sure that the dual queue consumed by ROUTING always contains only the most recent update from each gateway requires some sort of inter-process locking, with consequent problems of lock contention. It seems as if this is best avoided.

It is probably better just to put incoming updates on the ROUTING queue as they arrive, in the order in which they arrive. The ROUTING process can be required to "immediately" (i.e., as its highest priority) remove all the updates from RQ and store them (before processing the information in them) in a way that saves only the most recent one from each gateway. For instance, it can have a table with one entry for each gateway which contains the age and sequence number of the most recent received update from that gateway. Each entry would also have a place for a buffer id. If the buffer id is zero, then the indicated update has already been processed. Otherwise, it is the buffer id of the as yet unprocessed update.

(Perhaps we want to put a limit on the number of updates that can be removed from the dual queue before processing, just

to make sure some processing eventually gets done. Or we might want to cause a "crash" if ROUTING just can't keep up.)

Now, what happens when an update is removed from the head of the dual queue? First, a determination is made as to whether it is more recent than the update whose sequence number and age appears in the table. If it is not more recent, it is discarded. Otherwise, if the buffer id area in the table is non-zero, that buffer is discarded, and the buffer id of the buffer being considered is placed in the table. If the buffer id area in the table is zero, then the buffer id of the buffer being considered is placed in the table, and the number of the source gateway is placed in a ring buffer.

After the RQ is emptied, the ring buffer will contain the number of every gateway from which there is an unprocessed routing update. The ROUTING process can treat this ring buffer as a queue, removing gateway numbers and going to the above-mentioned table to find the buffer id of the update to process. When a gateway number is removed from the ring buffer, the corresponding buffer id area in the table must be zeroed. (This ring buffer can be implemented as a dual queue whose maximum

length is the number of gateways. Events should never get placed on this dual queue, since it is used internally only, rather than for inter-process communication; only the Poll operation should be used for consuming this queue. In fact, this ring buffer could be disposed of entirely if one were willing to do a search of the table; it is just an optimization.)

Essentially, this procedure just introduces some pre-processing of the queue of received updates, to avoid the (presumably) more expensive processing involved if one processes update n from gateway g while update n+1 from gateway g is right behind it on the queue.

The pre-processing procedure assumes that it is being done by a single process, using data structures that are unavailable to any other process. There might be some application for multiple routing processes running simultaneously on different processors, however. If we need to compute N different routing trees for some reason, and if we have N processors, it could be advantageous to have each tree computed by a separate routing process. Nothing in this scheme, however, prevents there from being N computations of a routing tree, which can all proceed in

parallel. However, it is required that there either be a single process which handles all the pre-processing of the updates, or else that the pre-processing be replicated independently in all of the routing processes.

In the latter case, all the updates would have to be queued to all the routing processes. While this may seem inefficient, it does provide a simple and flexible way to test out and even to switch dynamically between alternative routing strategies. The former procedure is probably more efficient, assuming that the pre-processing of updates is a small part of the total routing computation. If that procedure were to be adopted, however, additional inter-process communication mechanisms would have to be developed for use between the process which pre-processes the updates, and the processes which compute the paths.

Updates created by this gateway (i.e., which report about the connectivity between this gateway and its next-door neighbors) should be handled as incoming updates. Such updates might be created by the neighbor up/down process, for example, and then placed on the ROUTING queue.

The pre-processing procedure as so far specified does not accommodate the case in which multiple updates (presumably from different source gateways) have been combined into a single message by the neighbor which transmitted them. Yet in our functional spec for the routing procedure we required this feature. To accommodate this case, it is necessary only to increase the use count of the buffer by one less than the total number of updates contained in the message. This will prevent the buffer from being flushed until all its updates are processed.

OUTGOING UPDATES

In order to implement the flooding protocol, it is necessary to keep track of which neighbors have seen which updates. That is, given a neighbor N and an arbitrary source gateway G, there needs to be a mapping of the ordered pair (N, G) to a sequence number, which is the sequence number of the most recent update from G known to have been seen by N. We shall call this table the "flooding table". Any updates we receive whose source gateway is G and which are more recent need to be forwarded to N; updates whose source is G but which are no more recent should not

be forwarded to N.

If a particular update is received from a particular neighbor, then it is a sure bet that that neighbor has seen that update. The packet headers we use on our routing updates will identify the neighbor that transmitted the update. Thus we can easily keep track of the most recent update seen by each neighbor. We also intend to have explicit acknowledgments for routing updates; of course the reception of one of these from a particular neighbor tells us that certain updates have been seen by that neighbor. These acknowledgments will be queued to the ROUTING process for the purpose of maintaining the flooding table, but require no further processing or forwarding themselves.

Whenever ROUTING processes a particular update which has NOT been seen by some neighbor, it is the responsibility of ROUTING to cause that update to be forwarded to that neighbor. ROUTING should first determine which output interface or interfaces can be used to reach that neighbor. Then it should signal the output process for each such interface (this will be the LEVEL3 process if there are two protocol processes for the interface) to

transmit that update to that neighbor.

Periodically, ROUTING should check to see if there are updates which have not been seen (or rather, are not known to have been seen) by a given neighbor. If so, and if that update has not been transmitted to that neighbor very recently, ROUTING should signal the output process(es) to transmit to that neighbor ALL the updates which it has not yet seen. (This implies that ROUTING must keep track of the time it transmits a given update to a given neighbor). In this case, the updates should be batched into a single multi-update message, to the extent possible. This is the procedure that will cause retransmission of routing updates until it is determined that the neighbor has received them. The output process must know that it is being requested to do a retransmission, so it can mark the update messages as being retransmissions. Note that not in a multi-update message, some of the updates may be retransmissions, and some not.

Although ROUTING can tell from the flooding table whether any particular update needs to be forwarded to any particular neighbor, making this determination efficient may be a matter of

devising a suitably clever data structure; I am not proposing any particular data structure here.

When ROUTING processes an update which hasn't yet been seen by us, an explicit acknowledgment should be returned to the neighbor which forwarded that update to us. It is probably best to do this at the end of a round of pre-processing, so that multiple acknowledgments can be batched in a single message.

Explicit acknowledgments should also be sent when we receive an update which, although either a duplicate or obsolete, is marked explicitly as being a retransmission.

Now the hard part: how does ROUTING "signal" an output process to transmit some routing updates to a neighbor?

The simplest scheme is for ROUTING itself to get the necessary buffers and to create the message, and then to enqueue it for the appropriate output protocol process. One does not want the routing message to get queued behind lots of data; it should rather be placed on a high priority queue so it precedes any queued data.

However, putting a routing message on a priority queue does not necessarily ensure that it will be transmitted very shortly. The output device may be blocking, or one of the protocols may be blocking. There can still be a significant delay. We want to have some scheme to ensure that ROUTING does not create and enqueue a retransmission of update u until the previous incarnation of update u has actually left the gateway. This requires some acknowledgment scheme between ROUTING and the output processes.

We can use the inter-process acknowledgment scheme we have already discussed: after a packet has been disposed of, it gets queued back to some process which has seen it previously. When a routing message has been transmitted (i.e., when the driver process gets it back from the interrupt handler after transmission has been completed), the message can be placed on "routing message disposition" queue for the ROUTING process. Once ROUTING has enqueued a message for transmission, it will not generate any retransmissions until it has thus determined that the previous transmission has been completed. (Note however that this must be timed out, for diagnostic purposes.) This is the

only way to prevent all the buffers from filling up with retransmissions of updates that haven't yet left the gateway. This problem has been seen in the IMP from time to time, when a trunk or modem is experiencing certain peculiar kinds of failures. The time stamp kept by ROUTING to mark the time of the last retransmission of update u to neighbor n should really be the time at which the transmission actually took place, rather than the time at which ROUTING put the transmission on the DISPATCH queue. (This implies that part of the disposition information passed back to ROUTING is the time of transmission.)

Another possible problem: by the time a routing update from source gateway G is actually about to be transmitted, more recent routing information from G may have been received. Since a given update obsoletes all prior ones from the same source gateway, one would like to ensure that only the most recent information gets transmitted. This could be implemented by having ROUTING enqueue the output process not with a completed routing message, but rather with a sort of "skeleton message". Basically, this would just constitute a command to the output process to send routing information to a particular neighbor. The information itself,

however, would not be placed in the message by ROUTING. Rather, the output driver process would obtain this information later and place it in the message at the last possible moment, thereby ensuring the sending of the most recent information possible.

The only real problem with this procedure is the need for having inter-process communication between the output driver process and the ROUTING process. The information the output process would need is available in the tables of the ROUTING process, but this information cannot just be read out by the output process without the benefit of some sort of locking mechanism. An alternative would be to gather the information through some sort of inter-process message passing, but once a packet has actually made it down to the driver process, we don't want to add further delay by waiting for an inter-process message exchange.

Unless we get some evidence that this is going to be a real problem, then the cost of doing it probably exceeds the benefit.

Another problem: if a gateway has a large number of neighbors over some interface (e.g., the gateway is on the

ARPANET or the MILNET), do we really want ROUTING to place a large number of messages on the output queues, each of which has exactly the same data? It might be better to have ROUTING construct the message along with a command (placed in the buffer) for the output process to send that same message to each of a list of destinations. It can be left to the output process to determine just how this "multicast" is to be implemented. If the output device really supports multicast addressing, then that facility should certainly be used. Otherwise, the output process may decide just to transmit from the same buffer N times, changing the destination address each time, until a transmission has been made to each destination. Or the output process could decide to send N separate messages.

In general, one doesn't want to lose routing messages because of buffer management considerations. ROUTING should be treated as an input pseudo-device from the perspective of creating outgoing messages. When ROUTING needs to get a buffer, it should first try to get it from the general free queue of the processor which is connected to the output device. If this is empty, the general free queues of other processors should be

considered. If these are all empty, ROUTING will use its subsidiary free queue. When ROUTING attempts to enqueue a message for output, it should first attempt to charge the buffers to the output interface. If that fails, though, the message should be enqueued anyway, not discarded, and will remain charged to input, with ROUTING as the "input device".

ROUTING TABLES

The ROUTING process needs to write several tables to be read by other processes. There will be a set of tables (indexed possibly by type of service) for mapping destination network into exit gateway, and a set of tables for mapping exit gateway into neighboring gateway. Only the ROUTING process will write into these tables, but other processes will read them. Clearly there is a potential problem if a process reads the table while it is being written.

We need to make sure that no process reads one of the routing table while it is being modified, but we also want to allow any number of processes to be able to read each table simultaneously. We don't ever want to block a process from

reading a table, if we can avoid it, since reading the tables is an essential part of the high throughput path through the gateway. And we don't want to impose much of a delay on letting ROUTING modify the tables, since that just forces the tables to be out-of-date longer. Note also that what we need here is not just a simple dual queue lock. That might work if we wanted to allow access to just one process at a time, but we want to allow multiple simultaneous access for reading. What we really need is a read-lock and a write-lock which are distinct from each other.

One simple way to implement the read-lock is by "double buffering" each "public" routing table. ROUTING can supply other processes (through the common segment) with a pointer each table. This points to the "public" table. Modifications are actually made to a "private" copy of this table. When the modifications are completed, the two tables are logically "switched" by overwriting the pointer with the address of the newly modified table. This ensures that a table is never read while in the process of being modified.

Implementing the write-lock is more difficult. Once the tables are logically switched, all new reads will reference the

new table. However, there may be old reads in progress, which are still referencing the old table. ROUTING must not be allowed to modify the old table until it is certain that no more reads of that table are in progress.

This can be implemented by beginning each table with a "use count" field. Each process increments this field when it begins a read and decrements it when it finishes a read. Once the pointer to an old table is overwritten with the pointer to a new table, this use count can only decrease. When it reaches zero, no read is in progress, and no further reads are possible, so it is safe to modify the table.

However, suppose ROUTING is ready to modify the table, but discovers that a read is still in progress (non-zero use count). We certainly don't want ROUTING to go into a busy wait until the count becomes zero. Rather, we would like to cause an event to be sent when the count becomes zero. This event would have to be sent by the process which is the last to complete reading the table after the switch is made. Unfortunately, it is difficult for a process to determine whether the switch was made before it completed its read. To avoid race conditions, some indivisible

means of making this test is needed.

The following procedure should work. When ROUTING is ready to modify a table (after performing the logical switch), it indivisibly decrements the use count. If the old value was zero, then no process is in the midst of a read, and the table may be modified. If the old value was non-zero, ROUTING will not modify the table until the posting of a specified event. Each process, when it completes a read, will indivisibly decrement the use count and check its previous value. If the old value was already zero, then the specified event will be posted.

Note that a use count whose value is already zero will never be decremented unless ROUTING is waiting to modify the table. This can only happen after the tables are logically switched, so the process that decrements a zero value must be the last one which will read that table. Furthermore, unless a reading process decrements a use count whose value is already zero, it can be sure that ROUTING is not waiting to modify the table.

If ROUTING needs to wait before modifying the table, it need not sleep while awaiting the posting of the event. It can

continue doing other things, such as serving its input dual queue, causing transmission of routing updates, etc.

If ROUTING has several updates to process, it need not publicize a new table after each update is processed; it may elect to process several updates before publicizing a new table. This may delay the effect of the first update, but may also speed the effect of the later updates, a trade-off which needs to be further examined.

We wouldn't want a hanging DISPATCH process to totally disable ROUTING by leaving the write-lock hanging for an arbitrary period of time. Thus this lock needs to be carefully timed out. If ROUTING cannot modify a table for a period of several seconds, we should probably crash the gateway. This prevents the gateway from continuing to run while it is unable to process routing updates.