JET PROPULSION LABORATORY INTEROFFICE MEMORANDUM
SFB: 363-87-001

JULY 22, 1987

# NOTES ON IMPLEMENTING AND DEBUGGING MESSAGE SENDBACK

Steve Bellenot

These notes document the problems that were noticed while getting message sendback to work. Of course, from the begining there was code which attempted to handle reverse messages, and many people helped with the debugging.

1. The GVT window:

Perhaps the biggest surprize was that while GVT is being calculated, there is a period of time that reverse messages must have sendtimes greater than LVT. Indeed, between the time gvtlvt is computed and gvtupdate updates the GVT, each node requests memory at least once. Suppose memory is low and a message to sendback is chosen which has sendtime greater than the old GVT, but not greater than the new soon-to-be-GVT. The message could arrive on another node after that node has done gvtupdate. Thus we could have a rollback to a time before the new GVT. A solution is not to send messages with sendtimes less than the LVT (determined at gvtlvt) between the times of gvtlvt and gvtupdate.

2. Pointers dangle everywhere:

There are (at least) two new ways a pointer can dangle with message sentback. Now messages in the output queues can be annihilated and messages in the input queues can be reversed, dequeued and sentback.

First it was co, the "current output pointer" of each ocb. If the message pointed to by co was annihilated, co couldn't just be made NULL like the way ci (current input) becomes NULL. Rollback makes no assumptions on ci, but assumes that if there is output at this "old time" then co points to it. Because of queries (at least), co could be pointing anywhere in the current output bundle. Thus undangling co requires you to "look both ways".

Next it was a local variable f = find (...), followed by a m_create, and concluding with a l_insert( f, ...). The call to m_create could pick f to sendback and f would no longer be in any queue. (Mike found this one. By never picking anything but an event message to sendback with sendtime > the timestamp of the memory request, I didn't see this error.)

3. New infinite loops:

I assumed that negative messages were never sentback. Thus if an arriving reverse positive message didn't annihilate, it was unreversed and sent forward. However since anti-messages were sent via the MI call sndmsg, on-node reverse messages could bounce back and forth forever. (It was repeatedly picked as the best message to sendback. It was dequeued and given to deliver, but since it didn't annihilate with anything in the output queue, we re-enqueued it back it the input queue. Hence a long no-op.) But there still wasn't enough memory so we found the same message to sent back again.

This problem got me to uncouple enqueuing with rollingback. A new runstat called RDY_2_ROLL was created and which allowed cancel_omsgs to once again call deliver. This ended up being a much bigger job than it first seemed. The debugging routines check_ocb, and the dangle checks for ci and co date from this period.

Interestingly enough, queries caused a problem here, a special case was required when an input bundle consisted entirely of query reply messages.

4. Stack garbage:

When running low on memory, states are not always allocated. When there isn't a saved state at each event, the number svt < lvt is possible and hence stack garbage becomes a problem. An attempt was made to rid ourselves of this problem once and for all. No output messages were made or cancelled for times when svt < lvt. (After all we did it right the last time or lvt would be equal to svt.) However queries required more work since co points to the current query, so we can figure out which query reply to read. Finding the correct query reply seems impossible in general(see "the never ending stack garbage problem"), but sender alone is enough for Commo*.