4GL's, COBOL and Data Communications

by
John D. Alleyn-Day
A H Computers
8210 Terrace Drive,
El Cerrito, CA 94530-3059
415-486-8202 or 415-525-5070

Introduction

One of our clients, Underground Service Alert, operates an extensive system for sending messages to hundreds of printer terminals located in the states of California and Nevada. Although not a true real-time system, it must neverless transmit messages in a timely fashion with insignificant delays. They were using on old system that had been around for about ten years and that had come to the limit of its performance because of various problems with the software. It was our task to change this software structure to allow it to handle their increased business.

The Database System

The major problem was that the old system revolved around an IMAGE database and, in particular, a detail dataset that was used to control the transmission process. This was a very complex set with six paths. These paths were present to allow access by a variety of routes but they had a very deleterious effect on performance. One of the critical keys was the "status" of the record, indicating whether or not the message was "Sent", "Waiting" or "Failed". The sending program read down this path to find messages to send. Unfortunately, changing the status, as one must for each message transmitted, involved a delete and an add.

This process was causing a major I/O bottleneck in the system and it was therefore necessary for us to find another method of controlling the transmission without such an excessive overhead. Furthermore, we wanted to introduce an additional feature, namely making the order of tranmission dependent on a "priority" field that indicated the importance of the message.

The "status" had to be a key, so that the messages waiting to be sent could be quickly found. At the same time, it had to be capable of fast updating. Several different methods were considered, including MPE flat files or MPE message files, from which the records would be deleted as they were sent. However, incorporating the priority scheme made a FIFO queue such as a message file inapproriate.

Our final solution was to use a KSAM file for our transmission queue. Making one of our keys "status" followed by "priority" allows us to find messages waiting to be sent in priority order, and to insert new messages into the queue at the point corresponding to the priority that we assigned. Furthermore, KSAM allows us to change secondary key values with comparatively little overhead, without deleting and adding the record. There is a further performance advantage of using KSAM. We usually write more than one record at a time to the file (because any one message is usually sent to several different destinations) and, if we lock around this group of updates, then the buffers are most likely to be written out to disc only at the end of all the updating. This can also save enormously on disc I/O.

The Structure of the Programs

Our main requirement is to control approximately ten outgoing modems, to have each dial a specific number, prepare a message from data on our database and then transmit it. The database then needs to be updated with the information that the message was sent.

There are several subsidiary functions that must also be performed. We have to schedule the transmissions appropriately, ensuring that the most critical messages go first and that two different modems don't try to send to the same terminal. We also need to keep track of failing terminals and not attempt to send to a terminal that is down. Furthermore we need a mechanism for changing the operating parameters "on the fly" and methods for inquiring into the performance and controlling it.

In general, the activities associated with each modem are asynchronous, in the sense that what happens on one modem has little or no connection with what is happening on another. There are also long I/O waits associated with each modem, while it is dialing or transmitting data. We cannot have a program wait for one modem to complete dialing before it continues to another, without serious impact on the total throughput of the system.

We could have written a program with privileged mode no-wait no-buf I/O. This has been done by many people in the past for special-purpose terminal-handlers. However, it would have been very complex and it would still not avoid the problem that there would be no modem port I/O while the program was updating the database. However, this situation, with multiple simultaneous, asynchronous processes is what time-sharing is all about, and is precisely what the HP3000 was built to handle. So we make use of process handling and allow the MPE operating system to handle the complex scheduling of the various processes, rather than trying to do it ourselves.

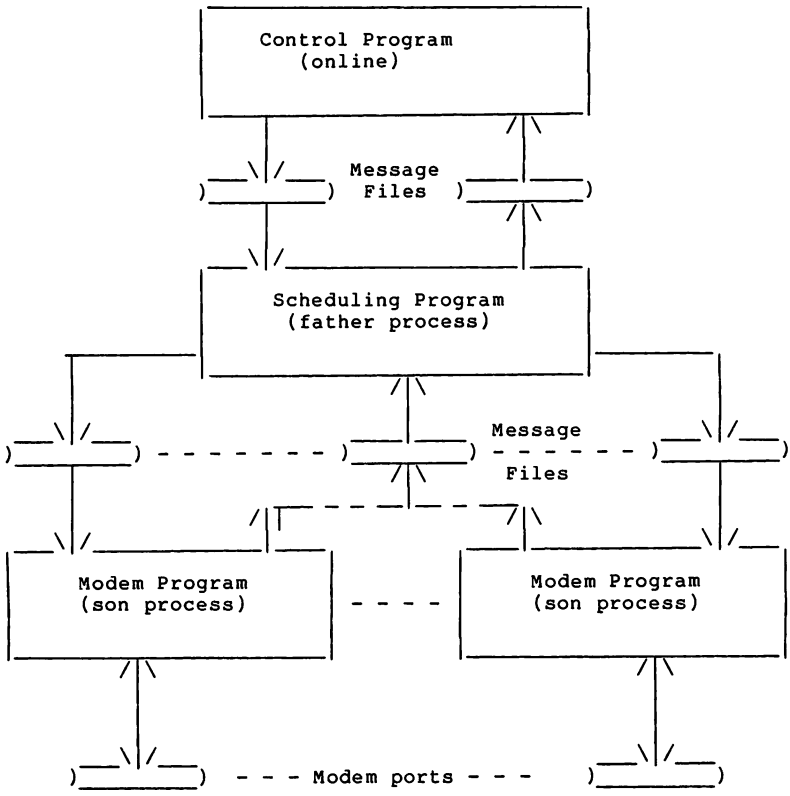

Fig. 1. Interconection of Programs and Message Files.

The programs are set up as shown in Fig. 1. The main scheduling program is the father process,and keeps track of the messages to be sent and the availability of each line. It also updates the queue file to reflect the status of the messages. Each dial-out modem has its own son process. This process opens the port, sets up the modem, prepares messages, dials the modem, and transmits the messages. The way in which the modems are controlled was discussed in a talk that I gave last year at the Interex conference in las Vegas, called "Dialing out from the HP3000".

There is also a "control" program, run in session mode that communicates with the scheduling program via two permanent message files. This allows the supervisor to see the internal tables, to alter internal parameters and to fail and restore lines and terminals.

We make use of MPE to do our scheduling for us. The son processes spend a great deal of their time waiting on I/O, either from the port or from the message file in which commands from the control program are placed. If there is nothing to be sent, the son process is suspended on a read of the message file and hence uses virtually no computer resources at all.

The father process, on the other hand, cannot suspend in this way. Not only must it respond to replies from the sons, but also to commands from the control program. Furthermore, it has other functions that must be performed at regular intervals.

The Operation of the Programs

The scheduling program has several internal tables that are used to store vital information. The largest table is the "internal queue", which is a miniature version of the "queue" file and is replenished from this file at regular intervals. It is there for performance reasons to save on disc I/O.

The father process runs through a polling loop. It first checks the time, and if five minutes (or whatever interval has been set by the supervisor) has passed, the program reads the "queue" file to determine the messages waiting to be sent, using the key by status, priority, etc. These messages are put into the internal queue table. The program checks each modem and if it is not busy, finds the terminal with the next highest priority message and instructs the modem to send the next four messages waiting for that terminal. The program also checks the return from the son processes. Depending on the result, various actions are taken, such as updating the queue file to indicate that the message has been sent and checking the error tables to update hte error counts.

There is an internal table that keeps track of failing terminals, counting how many times a call failed to go through, or how many

times the line was busy. After some number of failures, the terminal is considered to be down and all subsequent messages to that terminal are failed immediately. A terminal can be restored by the supervisor via the control program. A terminal may also be manually failed by the same mechanism.

Another internal table keeps track of the outgoing lines. In this table are stored the last command sent and the terminal and messages currently being handled by the corresponding modem as well as counts of failures on the line and flags to indicate a failed line or a busy line.

The control program is a very simple program and interacts with the transmission process by putting commands in the sending message file. These are retrieved by the scheduling program which either changes parameter data within its stack or responds by dumping the contents of one of its tables into the reply message file. The control program then formats this data and displays it for the supervisor.

Telecommunication Problems

The major difficulty in this scheme is recognising and handling telecommunication problems. Ideally, all problems would be recognized and handled automatically by the program, and in most cases we have achieved that. Busy signals, "no carrier" and disconnects are interpreted and handled by the scheduling program.

Sometimes, it is a little tricky to assign a problem to a terminal or a line. No dial-tone, of course, is a line failure, and a busy signal is a terminal problem. But a disconnect could be a problem associated with a specific terminal and the line connecting to it or it could be associated with an outgoing line and the modem attached to it. However, when we are updating the failure counts, we set the count back to zero as soon as a successful transmission is made. So we count this type of failure both as a line failure and a terminal failure, and it soon becomes apparent into which category it really falls. Even this can get a little difficult, as happened once when a major line out of the local MCI switch was damaged, and a very large number of calls failed to go through. This looked to the program as if both terminals and lines were failing. However, to the supervisor, it was very apparent that something dreadful was going on, and a call by him to MCI soon determined the nature of the problem.

There is one problem that we have not yet completely resolved. For some reason, the modems will, from time to time hang up when dialing. The program thinks that all is well, and keeps trying to dial, but the modem doesn't respond. It doesn't occur very frequently, and, of course, rarely when I am around to see what has happened. It can easily be cleared by hanging up the modem,

so the scheduling program contains code that checks each dialing modem, and warns the supervisor if a modem takes longer than a certain time to complete a dialing command. This invariably means that the modem has got into this peculiar state, and the supervisor is able to handle it. It would be preferable to correct this problem automatically without human intervention but, in the meantime, this is an effective strategy.

Problems of Operating Strategy

Structured programming is particularly important in an application such as this. Firstly, great use is made of HP intrinsics, and it is most convenient to have these in sections of their own so that they can be executed as simple functions. Furthermore, in operating the program, it becomes important to be able to change the strategies used in polling the various functions. This is made particularly easy if the program is well structured.

As an example of the strategy changes that we made, we started with a strategy that had the program go through the following sequence repeatedly. First the time was checked to see if the internal queue should be refreshed. Then the program checked all the replies waiting from the modem programs and carried out any updating needed as a result. Then the program looped through each modem, and, if it were possible to send, the appropriate commands were prepared and sent. Then the supervisor's message file was checked for a command.

Although this looks like a perfectly reasonable way to poll, it turned out to have two major drawbacks. First of all, there was a tendency for there to be a lot of completed messages at one time and this would lead to heavy updating while the son processes sat idle, waiting for their next command. Then, transmit commands were sent all at once to the son programs, tending to keep them in sync. We wanted to keep them out of sync, so that message preparation would not occur at the same time on different modems, as this process puts a significant demand on the database. Furthermore, because the updating was all in a chunk, it often took a long time to acknowledge the message from the supervisor, which translated into a poor response time for that command.

We fixed these problems by changing our strategy. Now, after each reply from a son process, we check to see if we can start another message on that line, and do so right away if possible. In addition, we check the supervisor's message file after each update, so that we never have to wait more than a few seconds for a reply. It was particularly easy to do this because the structured programming allowed us to shuffle the processing around just as we wished.

However, this is still not the optimum strategy. The supervisor's

message file is read with a timed read, and it takes a second to time out, so the more frequent reads add additional delay time into the loop and thus make the program operate at something less than optimum speed. Its speed, however, is better than with the original strategy.

Useful Programming Techniques

Writing and debugging multiple process programs of this kind can get to be quite complex, and I developed one or two techniques that may be useful to other people attempting the same thing.

As a matter of course, I always program trace messages into my COBOL programs, usually for each entry into a section. In the past I used the "parm" to turn on and off a flag for this purpose. With son processes it gets a little complex doing it this way, so for these programs, I used a different technique, namely a JCW that is set before the programs starts. There is a different JCW for each program, so that the messages from each program can be controlled independently.

With several programs running in the same session, sorting out the messages can get quite complex. Each message, of course, identifies itself in the message with its name, and, in the case of a son, with a number to identify which one it is. This number is passed by the father in the "parm" and corresponds to that program's position in the line table. It is passed back in any reply to identify which son process the reply is coming from.

As an additional debugging aid, there is a special command in the control program that will turn on the debug flag in the father program. This can be invaluable when a problem arises in the live system after the program has run for several hours. The debug messages can be turned on "on the fly" and they prints out on the STDLIST. As an addition to this, turning on the debug flag in this way also sets the debug JCW which is checked by the jobstream. This is important as we normally use "set stdlist=delete" to get rid of the listings if all is well, but in this case we don't want that to happen.

In the tuning of the program there were changes that had to be made to several parameters such as the maximum size of various tables, that necessitated recompiling the program. This is conveniently achieved by making use of macros in the HP COBOL to assign specific values to the parameters and have them substituted in the code as it is compiled. It avoids searching through the code for each place that the number needs to be changed.

The same facility is used for error handling on file intrinsics. A standard routine is laid out at the start of the program, with the appropriate error messages substituted for each use of the routine. This saves a great deal of debugging time for error

routines.

Using the 4GL system with Cobol

All the screens in this system have been implemented in a 4GL. The decision to use a 4GL was based on improved development times and the availabilty of the language for future development. Unfortunately things didn't work out quite the way we expected. Our system is somewhat out of the ordinary and many 4GL's work best on very ordinary systems. If the 4GL methods don't fit your development, the 4GL can cause more trouble than it is worth. We found that we didn't save very much development time by using a 4GL.

We carefully avoided using features in COBOL that might cause problems, such as item locking in IMAGE, and we ran into no significant problems with mismatches between COBOL and the 4GL. From this point of view, our melding of COBOL and the 4GL was very successful.

However, it appeared that the 4GL was running particularly slowly and also seriously slowing down the COBOL program. Although we were able to make some marginal improvements we finally discovered that this was because the 4GL in question has an unexpected and undocumented "feature" whereby each write is forced to disc, rather than waiting until the file is unlocked at the end of a group of writes. There is no way of overriding this "feature", and the vendor is not about to change it in any way. Combined with other inefficient ways of handling the data that the 4GL forces us into, this completely eliminates one of the major advantages of KSAM stated earlier. We are looking into methods to overcome this problem.

Summary

Mixing a 4GL with COBOL can be done without any major programming difficulties. However, inefficiencies in the 4GL will necessarily be present and can impinge on the efficient operation of the COBOL programs.

The handling of multiple outgoing telecommunication channels using father and son processes is a very effective method of control. Most of the programming is fairly straightforward, if one is familiar with using intrinsics within COBOL program programs, but tuning the program for optimum performance requires a considerable amount of experience and trial and error.

```
005900*    Define the maximum number of line-groups to be handled
006200*    Define the maximum number of lines to be handled
006300$define %lines=10#
006400
006500*    Define the maximum number of messages to be batched
006600*    into one call.
006700$define %batch=4#

007300*define the standard error routine for intrinsic calls
007400$define %filecheck=
007500      if c-c less than zero
007600        call intrinsic "FCHECK" using
007700                              \!1\,
007800                              file-err-code
007900        move low-values to file-err-msg
008000        call intrinsic "FERRMSG" using file-err-code,
008100                               file-err-msg,
008200                               file-err-length
008300        move "!2" to quit-msg
008400        perform pquit#

 .....................

027700 01  line-data.
027800      02  line-table occurs %lines times.
027800      02  line-table occurs 10 times.
027900        03 hold-index occurs %batch times    index.
027900        03 hold-index occurs 4 times    index.
028000        03  batch-pointer          pic 99.
028100        03  mt-file-num            pic s9(4) comp.

 ...................

107100      move +1 to dummy.
107200      call intrinsic "FCONTROL" using
107300                           ctlt-file-num,
107400                           \4\,
107500                           dummy.
107600      %filecheck(ctlt-file-num#,Fcontrol Error Controlt File#).
007500      if c-c less than zero
007600        call intrinsic "FCHECK" using
007700                             \ctlt-file-num\,
007800                             file-err-code
007900        move low-values to file-err-msg
008000        call intrinsic "FERRMSG" using file-err-code,
008100                               file-err-msg,
008200                               file-err-length
008300        move "Fcontrol Error Controlt File" to quit-msg
008400        perform pquit.
```

Fig 2. Example of using macros in COBOL