BUILDING AN ESM WITH CMS PIPELINES

Melinda Varian

Office of Computing and Information Technology
Princeton University
87 Prospect Avenue
Princeton, NJ 08544 USA
—.—
Email: Melinda@princeton.edu
Web: http://pucc.princeton.edu/~Melinda/
Telephone: 1-609-258-6016

SHARE 91
Session 9147
August 1998

ENHANCEMENT TO "STARSYS"

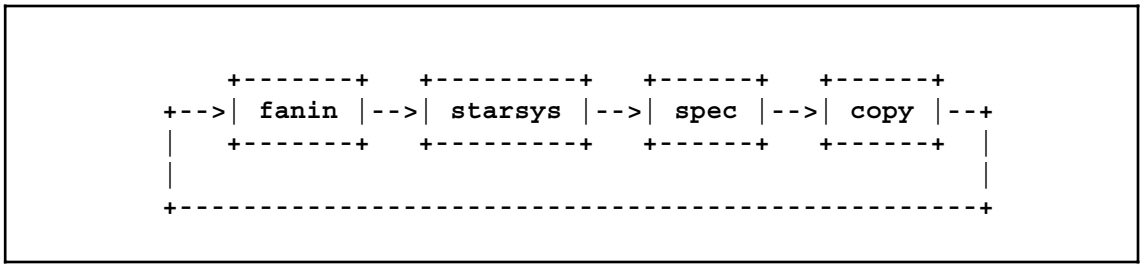
It recently became necessary for me to rewrite the External Security Manager we have used for many years to authorize users' CP LINK commands. The old ESM supported only two-digit years and three-digit virtual device addresses and was our last CMS 5 application, so it was definitely due for replacement.

CMS Pipelines is the obvious choice for writing such an application nowadays, of course, but its starsys stage did not work with the *RPI system service. starsys communicates with a two-way system service (for example, we use it with *ACCOUNT), but it returns a zero-length answer buffer to the system service. That is fine in the case of *ACCOUNT, as that system service uses the receipt of the answer buffer only as a signal that CP can discard the accounting record, that it has been fully processed by the application. The *RPI system service requires more information in the answer buffer, however. As we have *RPI implemented at Princeton, the answer buffer contains either the string "OK" or the text of a message to be sent to the user explaining why the LINK request is being denied.

Because IBM is no longer funding *CMS Pipelines* development, it was clear that we could not use *CMS Pipelines* for this project unless we were prepared to enhance the starsys stage ourselves. However, that seemed likely to be less work than writing the IUCV code we would need to do the job without *CMS Pipelines*, so I did the necessary modification to FPLSMG, which you can find in the Appendix. With this modification applied, starsys allows a primary input stream to be connected. If it has a connected primary input stream, it reads a record from that stream each time it is ready to send an answer buffer to the system service and uses the contents of that record as the message to be returned to the system service.

I tested this modification by building a minimal ESM:

```
pipe f: fanin | starsys *RPI | spec /OK/ 1.80 | copy | f:
```



This is an ESM that just can't say "no". Each time the starsys stage receives a message from the *RPI system service, it writes the message as a record on its output stream, where the spec stage converts it to a record that say "OK". The copy stage consumes the record written by starsys, freeing starsys to read the "OK" record from its input stream and send it back to the *RPI system service, thus authorizing CP to allow the LINK to complete.

Once this trivial ESM was working, the hard part of the project was done. All that remained was to teach the ESM to say "no" now and then, as appropriate.

While I was working on doing that, however, Rob van der Heij heard about my starsys enhancement and asked me to send it to him. A couple of days later, he sent this note:

```
Guess who is starting up so quick on my test system ;-)

09:57:43 GRAF  0040 LOGON  AS  RACFVM   USERS = 2
09:57:44 HCPRPI035I CP/RACF communication path established to RACFVM
09:57:44 HCPXPI035I CP/HACS communication path established to RACFVM
```

He had been annoyed by how long it took his RACF/VM server to initialize when he was doing second-level testing, so he had replaced it with this pipe:

```
pipe f: fanin | starsys *RPI | spec 1-* 1 x00 1 | copy | f:
```

That gave him a "RACF/VM" that couldn't say "no" but that initialized very quickly—just what he needed in his test environment.

USING "LOOKUP" AND "DEAL" TO PARSE COMMANDS

The rest of the project was just normal plumbing, but I will mention a couple of techniques that turned out to be useful.

The main pipeline for the ESM has two segments, one that runs starsys to communicate with CP and another that runs starmsg to communicate with end users, who can SMSG commands to do such things as add or query or delete authorization rules for their minidisks.

For parsing the user commands, I used a technique suggested by Rob. The commands are fed through a lookup stage that has a master file listing all legal commands and all legal abbreviations of those commands. The master file records also have a field that is not part of the key, a two-character code:

```
commands.1  = 'ASSIGN  AS'
commands.2  = 'ASSIG   AS'
commands.3  = 'ASSI    AS'
commands.4  = 'ASS     AS'
commands.5  = 'AS      AS'
commands.6  = 'AUTH    AU'
commands.7  = 'AUT     AU'
commands.8  = 'AU      AU'
commands.9  = 'DELETE  DE'
commands.10 = 'DELET   DE'
commands.11 = 'DELE    DE'
commands.12 = 'DEL     DE'
. . .
```

Once those records have been read into lookup's reference, lookup starts reading the commands from users on its primary input stream:

```
'PIPE (listerr endchar \ name AUTOLINK)',
       . . .
'\',
       'starmsg |',                 /* Cmds SMSGed from users.   */
       'not chop 8 |',              /* Discard message type.     */
       . . .
    'L: lookup substr w1 of 17-* w1 detail master |',  /* Legal? */
       'spec 1-* 1 read w2 nw |',   /* Command and routing code. */
 'deal: deal streamid w -1 strip',  /* Route commands to process.*/
       . . .
```

Commands that don't match any of the keys in the master file are written to the secondary output of lookup, where there is code to reject the request.

Commands that do match one of the keys in the master file are written to the primary output of lookup, where the spec stage appends the second word of the master record to the command record, thus putting that two-character code at the end of each legal command. The command records are then read into a deal streamid stage, which routes each record to one of its output streams, depending on the contents of the streamid field, which in this case is the last word of the record, the two-character code from the master file. Because the strip keyword is specified, the streamid field is removed from the end of each command record while the record is being routed to the appropriate output stream of the deal stage. The streams of deal are defined like this:

```
'PIPE (listerr endchar \ name AUTOLINK)',
    . . .
   'deal: deal streamid w -1 strip',  /* Route cmds to process.  */
'\',
   'deal.AS: |',                      /* Here if ASSIGN command. */
    . . .
'\',
   'deal.AU: |',                      /* Here if AUTH or EXCLUDE.*/
    . . .
'\',
   'deal.DE: |',                      /* Here if DELETE command. */
    . . .
'\',
   'deal.DR: |',                      /* Here if DROP command.   */
    . . .
```

That is, each of deal's output streams is defined to have a two-character streamid that matches one of the two-character codes from the master file. Thus, all the parsing and routing of the command verbs is handled in just three stages, making the pipeline quite easy to follow.

USING "LOOKUP" TO TEST RULES

Testing the database of minidisk access rules set by the users was an obvious application for lookup. And this project was an obvious place to take advantage of the new capabilities of lookup that allow one to update the master file while lookup is running:

```
'ADDPIPE (listerr endchar ? name ALPermission01)',
      . . .
  'l: lookup 1.26 detail |',     /* Match tests to rules.      */
      . . .
'?',
     'command LISTFILE * * B ( DATE NOH |',     /* Build list. */
      . . .
     'makemast |',               /* Format as master records.  */
  'l:',                          /* Master file for LOOKUP.    */
'?',
     '*.input.1: |',             /* From AUTH command.         */
     'makemast |',               /* Format as master records.  */
  'l:',                          /* Rules to be added.         */
'?',
     '*.input.2: |',             /* From DELETE command.       */
     'makemast |',               /* Format as master records.  */
  'l:'                           /* Rules to be deleted.       */
```

The authorization rules are kept in minidisk files, one file per user minidisk. When the ESM starts up, this pipeline reads all of those minidisk files into the lookup reference (after having reformatted them a bit). The lookup is then ready to receive LINK request records on its primary input and test them against the rules in its reference.

So far, of course, this is something that could have been done with a lookup all along. What's new here is that the lookup has tertiary and quaternary input streams defined.

Elsewhere in the ESM pipeline, commands SMSGed from end users are being received and processed at the same time that CP LINK requests are being received by the starsys stage and are being fed into this pipeline to be matched against the rules database. Among the commands those end users might be SMSGing to the ESM are AUTH commands (which say to add rules authorizing other users to LINK to a minidisk) and DELETE commands (which say to delete rules authorizing other users to LINK to a minidisk). The portions of the pipeline that process the AUTH and DELETE commands aren't shown here. They are very straightforward in that they just update the appropriate rules file on disk. But after they have rewritten the rules file, they also feed a record to this lookup to force it to update its master file, thus changing the rules in real time.

The pipeline segment that processes the AUTH command writes a record containing the new rule to an input connector (*.input.1:) that is connected to the tertiary input of this lookup. That causes lookup to add the new rule to its reference immediately. Similarly, the pipeline segment that processes the DELETE command writes a record containing the deleted rule to an input connector (*.input.2:) that is connected to the quaternary input of this lookup. That causes lookup to remove the corresponding rule from its reference immediately.

Thus, we can leave the ESM running for weeks at a time without the rules in memory getting out of sync with the rules in the disk files.

USING AN AFFIXED "LOOKUP"

The lookup we've just been talking about is obviously relatively expensive to start up, as it reads a large number of files into its reference. Therefore, one wants to leave it running, rather than, say, starting it over each time a CP LINK request is received. On the other hand, the code that processes a CP LINK request turned out to be much more straightforward if it were written as a "sipping subroutine pipeline", *i.e.*, a Do Forever group in which a CALLPIPE is issued each time a LINK request is received.

One can have it both ways by putting the expensive-to-initialize lookup stage into an "affixed pipeline" and letting it stay permanently running, attached to the side of the stage that needs it. (An "affixed" pipeline is one that has an input connector attached to the output of the stage that issues the ADDPIPE and an output connector attached to the input of that stage.) Then one can use a CALLPIPE in a loop to create a subroutine pipeline for each incoming request. That inexpensive-to-initialize subroutine pipeline can connect temporarily to the permanently affixed pipeline to give it some records to process and to get the results back. It can then terminate when it has finished processing one request.

Here is another view of the pipeline we saw earlier that uses lookup to test whether a LINK request matches any of the rules in the database:

```
'ADDSTREAM BOTH'                    /* Define quaternary streams.  */

'ADDPIPE (listerr endchar ? name ALPermission01)',
      '*.output.3: |',              /* Tests from 4ry output.      */
   'n: nfind !|',                   /* EOT marker; not a test.     */
   'l: lookup 1.26 detail |',       /* Match tests to rules.       */
   'f: faninany |',                 /* Merge output streams.       */
      '*.input.3:',                 /* Output to quaternary input. */
'?',
       . . .
   'l:',                            /* Master file for LOOKUP.     */
'?',
   'n: |',                          /* End of transaction here.    */
   'f:',                            /* EOT into quaternary.        */
'?',
       . . .
```

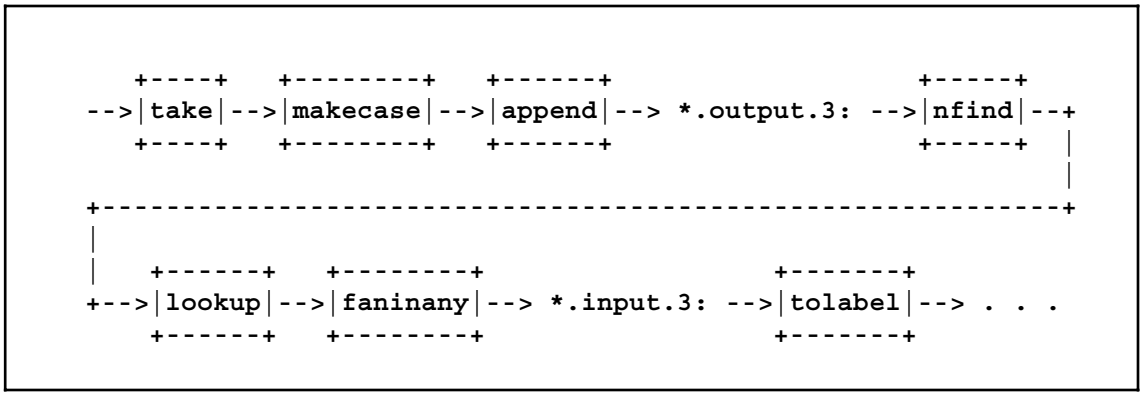
The ADDSTREAM command creates quaternary input and output streams for the stage that issues the ADDSTREAM and ADDPIPE commands. The pipeline created by the ADDPIPE reads input from the quaternary output (*.output.3:) and writes output to the quaternary input (*.input.3:). Records starting with an exclamation mark are diverted around the lookup stage, going straight from the input to the output. All other records are put through the lookup to test whether they match rules in the rules database. If they do, they are also written to the output (which goes, of course, into the quaternary input of the stage that issued this ADDPIPE command).

Once it has issued those ADDSTREAM and ADDPIPE commands, the stage waits for a LINK request record on its primary input. When one arrives, it issues a CALLPIPE command to process the request:

```
'PEEKTO'                            /* Wait for request to arrive. */

'CALLPIPE (listerr endchar ? name ALPermission02)',
      '*.input.0: |',               /* LINK requests from HCPRPI.  */
      'take 1 |',                   /* Sip one request at a time.  */
      'makecase |',                 /* Convert to cases to test.   */
      'append literal !|',          /* Add transaction end marker. */
      '*.output.3:',                /* Into the affixed pipeline.  */
'?',
      '*.input.3: |',               /* From the affixed pipeline.  */
   't: tolabel !|',                 /* Up to the EOT marker.       */
       . . .                        /* Process matching rules.     */
'?',
   't: |',
      'take 1 |',                   /* EOT marker to here.         */
      'hole'                        /* Consume it.                 */
```

When the CALLPIPE command is issued, its connectors connect to the connectors of the ADDPIPE, in effect inserting the ADDPIPE into the middle of the CALLPIPE, as shown in this diagram:



The makecase stage converts a single LINK request into some number of records, each defining an hypothesized rule that would either allow or deny this particular LINK. (For example, a request for a write LINK would be allowed by a rule that allows a write LINK or by a rule that allows a multi-write LINK, but it would be excluded by a rule that disallows a write LINK or by a rule that disallows a read LINK.)

A record containing an exclamation mark is written after the records containing the rules to be matched, and all of the records are fed into the affixed pipeline (through the output connector).

The lookup stage in the affixed pipeline writes any matching rules to its output stream, which is connected to the quaternary input here. The exclamation mark record, which is the last record into the affixed pipeline for a given transaction, is also the last record out. So, this CALLPIPE command completes when that record is detected by its tolabel stage. By that time, any of the hypothesized rules that were matched by actual rules will have been processed (the processing isn't shown here), so this transaction is complete.

This sequence can happen over and over, with a new CALLPIPE being run each time a LINK request record is received. The only tricky part is that the CALLPIPE must be able to detect the end of a transaction so that it can detach itself from the affixed pipeline and leave the affixed pipeline with no records still flowing through it. If that rule is met, new instantiations of the CALLPIPE can continue connecting to the affixed pipeline and using its services, leaving it running "forever".

Although this usage is somewhat arcane, it is a technique well worth mastering once you begin writing servers with *CMS Pipelines*.

Appendix A

ENHANCEMENT TO “STARSYS” TO SUPPORT AN INPUT STREAM

This file is called FPLSMG ANSWER. You are welcome to use it at your own risk.

```
./ I 01622001          $ 1632001 10000               06/19/98 15:22:42
* 19 Jun 1998  +++ Read answer for two-way system service.     ANSWER
./ R 08535001          $ 8538991 3990                06/19/98 15:22:42
         FLAGTM TWO_WAY       Not first, but is it two-way?    ANSWER
         IF    ONE            Yes, gets answers, not cmds.     ANSWER
           PCALL PROCESS      So call it to get answers.       ANSWER
         ELSE  ,              Not two-way, so input is cmds.   ANSWER
           PCALL READCOMMANDS Read commands to send.           ANSWER
         FI    ,                                             ANSWER
./ R 09248001          $ 9257991 9990                06/19/98 15:22:42
&MODULE.2W FPEP FP=NO,ARGS=YES,                              ANSWER  *
./ R 19602001          $ 19604991 2990               06/19/98 15:22:42
   ZIP   R0                   Zero answer length.            ANSWER
   ZIP   R1                   Zero answer address.           ANSWER
   #LT   R14,POSITION         Where are we?                  ANSWER
   IF    POSITIVE             Not first, so read answer.     ANSWER
     PIPLOCAT EXIT=MINUS      Get record with answer.        ANSWER
     LEAVE COND=NOTZERO       Exit if there is none.         ANSWER
   FI    ,                                                  ANSWER
   IUCV  REPLY,ANSBUF=(R1),ANSLEN=(R0),PRMLIST=(R6)         ANSWER
./ I 19912001          $ 19918001 6000               06/19/98 15:22:42
   #LT   R14,POSITION         Where are we?                  ANSWER
   IF    POSITIVE             Not first, so have read record.  ANSWER
     PIPINPUT (,0)            Consume that record.           ANSWER
   FI    ,                                                  ANSWER
```