TECHNIQUES FOR BUILDING SERVICE MACHINES WITH *CMS PIPELINES*

Melinda Varian

Office of Computing and Information Technology
Princeton University
87 Prospect Avenue
Princeton, NJ 08544 USA
—.—
BITNET: MAINT@PUCC
Internet: maint@pucc.princeton.edu
Telephone: (609) 258-6016

SHARE 83, Session 2607
August, 1994

I. AN EXAMPLE (STARMSG, IMMCMD, AND PIPESTOP)

CMS Pipelines is a very powerful tool for building service machines. If you use *Pipes* for your personal computing, you have already learned most of what you need to know to use it for writing servers. However, there are a few *CMS Pipelines* stages that are particularly useful in servers that you may not yet have experience with. Those are what I will be discussing today. Let's jump right in and look at a service machine that we have running at Princeton. Although this is a very small example, it is a real one and illustrates some useful techniques:

```
/* DUMYSMTP EXEC:  Divert SMTP's spool files to BABYBEAR's SMTP  */

'PIPE (endchar ? name DummySMTP)',
  '  starmsg CP SET IMSG IUCV',    /* Listen for file arrivals.  */
  '| find 00000007'Translate(Left(userid(),8),'_',' ')'RDR FILE'||,
  '| b: nlocate 41.8 /VMBITNET/',  /* (Don't want war with RSCS.)*/
  '| spec 26.4 1',                 /* Pick out just the spoolid. */
  '| f: faninany',                 /* Merge in other stream, too.*/
  '| spec',                        /* Format CP commands:        */
     '/TAG FILE/ 1',               /*   TAG FILE                 */
     'word 1 nextword',            /*   nnnn                     */
     '/BABYBEAR SMTPB/ nextword', /*   BABYBEAR SMTPB           */
     'write',                      /*   -----------------------  */
     '/SLEEP 1 SEC/ 1',            /*   SLEEP 1 SEC              */
     'write',                      /*   -----------------------  */
     '/TRANSFER/ 1',               /*   TRANSFER                 */
     'word 1 nextword',            /*   nnnn                     */
     '/VMBITNET/ nextword',        /*   VMBITNET.                */
                                                    (continued)
```

```
  '| cp',                          /* Issue CP commands.         */
  '? b:',                          /* File came from RSCS; bad!  */
  '| y: faninany',                 /* Merge in other stream, too.*/
  '| spec',                        /* Build TRANSFER command.    */
     '/TRANSFER/ 1  word 2 nextword  /MAINT/ nextword',
  '| cp',                          /* Issue TRANSFER command.    */
  '? cp query rdr all *',          /* Prime the pipeline.        */
  '| drop 1',                      /* Drop header or only line.  */
  '| n: nfind VMBITNET' ||,        /* (Don't want war with RSCS.)*/
  '| spec word 2 1',               /* Pick out just the spoolid. */
  '| f:',                         /* Go transfer the file(s).   */
  '? n: | y:',                     /* Send bad file to MWV.      */
  '? immcmd CP',                   /* CP immediate command.      */
  '| cp',                         /* Pass command to CP.        */
  '? immcmd STOP',                 /* STOP was typed on console. */
  '| pipestop'                     /* So stop the pipeline.      */

'CP SET IMSG ON'                   /* Restore normalcy.          */
```

Our mailer virtual machines have algorithms for distributing Internet traffic among our SMTP service machines. (We require multiple SMTP servers to handle the load.) When we decided to move one of the SMTPs to our P/370, BABYBEAR, we didn't want to modify the mailers to know how to send SMTP traffic to another RSCS node, so we wrote this little server to run in one of the SMTP machines and forward any files it receives to SMTPB on BABYBEAR.

The starmsg pipeline stage connects to the CP *MSG system service and routes any messages received from that service into the pipeline. When starmsg starts up, it issues its argument string as a command. In this case, it issues a CP SET IMSG IUCV command to tell CP to route any information messages for this virtual machine *via* the *MSG service. So, the starmsg sits there waiting for CP to give it IMSGes. When a message is received, starmsg prefixes it with a 16-byte header consisting of the message class and the origin userid and then writes it to its output. find discards messages we are not interested in, keeping only those that contain IMSGes "from" this virtual machine that begin with the text "RDR FILE". These will be messages about the arrival of reader files, *e.g.*:

RDR FILE 5677 SENT FROM VMMAIL . . .

The nlocate stage confirms that the file was not transferred from our RSCS virtual machine, VMBITNET. If it was, then the message record is diverted to the second segment in the pipeline, where it is transformed into a CP TRANSFER command that will transfer the file to me to check. (That little test is there to prevent this server and RSCS from getting into a loop transferring the same file back and forth for some reason.) If the file is one we want to process, a spec stage is used to pick out the spoolid and another spec stage converts each spoolid record into three CP commands, a TAG, a SLEEP, and a TRANSFER. The cp stage issues those commands, and all is quiet until another message arrives.

You will note that there are three other pipeline segments that we haven't yet discussed. The third segment of the pipeline runs once as soon as the pipeline is initiated. It issues a CP QUERY

RDR command in case there are already some files in the virtual machine's reader. If there are, the response from QUERY RDR is transformed into spoolid records and those records are fed into the secondary input of the faninany stage and then processed in the same way as the spoolid records extracted from the file arrival messages.

The last two segments of the pipeline process commands typed on the virtual machine console. The two immcmd stages are also started as soon as the pipeline starts up. Their arguments specify that they wish to see any console commands that begin with "CP" and "STOP", respectively. If a command starting with "CP" is typed on the console, the first immcmd stage receives it as a record (with the string "CP" removed). It passes that record to a cp stage, which executes it as a CP command. The second immcmd stage is given control when the string "STOP" is typed on the console. It simply copies its input record to its output, which is connected to the input of pipestop.

As soon as pipestop receives an input record, it posts any stage in the pipeline specification that is waiting on an ECB for an external event. The only such stages in this pipeline are starmsg and the two immcmd stages; they all terminate when they are posted. When starmsg terminates, the input stream for find is severed, which causes it also to terminate, thus severing the input for nlocate and causing it to terminate. This termination process cascades through the pipeline until all the stages have terminated, at which time the PIPE command completes. (If you are using a version of *CMS Pipelines* that doesn't support pipestop, the same effect can be achieved by converting a record into a PIPMOD STOP command and passing that to a command stage.)

In this example, if the user had wished to terminate the processing of messages, but to leave the processing of immediate commands running, he could instead have typed "hmsg" on the console to terminate only the starmsg stage. *CMS Pipelines* sets up the "hmsg" immediate command automatically when it sets up a starmsg stage for the *MSG system service.

starmsg can be used to connect to system services other than *MSG. To use it to connect to some other CP service, simply specify the service as the first argument, *e.g.*, starmsg *msgall.

II. STARSYS

Although starmsg can be used to connect to both one-way and two-way system services, starsys is preferred for connecting to the two-way services, such as *ACCOUNT. This fragment is from the pipeline that runs in OPERACCT on our VM/ESA system:

```
'CP RECORDING ACCOUNT ON LIMIT 100'

'PIPE',
   'starsys *account |',        /* Connect to *ACCOUNT (2-way).*/
   'nfind   'type5'|',          /* Discard Type 5 records.     */
   'nfind   'type8'|',          /* Discard Type 8 records.     */
   'nlocate 13.8 =V/SIE   = |', /* Discard V/SIE records.      */
    . . .
   'pipchgrt |',                /* Post to $File and Tranfile. */
    . . .
```

starmsg accepts input from CP as quickly as CP can deliver it, even if that means that starmsg must buffer a large number of records. starsys accepts only one record at a time and does not acknowledge receipt until its own output record has been consumed. If you use starsys *account and make sure that all of the stages between starsys and the stage that completes your accounting process do not delay the record, you can be sure that if your pipeline fails or your system crashes, no unprocessed accounting records will be lost. CP will not discard a record until its receipt has been acknowledged, and starsys will not acknowledge receipt until your pipeline has processed the record.

Incidentally, if you aren't familiar with the concept of delaying the record, that is something you should learn about before implementing a server of any complexity. There are several places to go for explanations of this concept; the one I most recommend is in the paper *CMS Pipelines Explained* by the author of *CMS Pipelines*, John Hartmann. I have also included a discussion of record delay in Appendix C in my handout.

III. PROCESSING SERVER PARAMETERS

When writing a server, one often needs to deal with parsing a parameter file and with making the parameters available throughout the server. The *CMS Pipelines* REXX device drivers can make that very easy to do. This is a fragment from a subroutine called by one of my servers during its initialization:

```
/*-------------------------------------------------------------*/
/*  Set up array of local parameters for use by caller and by  */
/*  its subroutines.  First read the BITFTP PARMS file and     */
/*  convert each entry into an item in the "PARM." array.      */
/*-------------------------------------------------------------*/

'PIPE (name ParmSet1)',          /* Set local parameters:      */
  '< bitftp parms |',            /* Read parameter file.       */
  'chop string =/*= |',          /* Chop off comments.         */
  'locate /=/ |',                /* Require "assignment".      */
  'xlate fs = field 1 upper |',  /* Upper-case the name.       */
  'spec fieldseparator =',       /* Format to be VARLOADable:  */
    '/=PARM./ 1',
    'field 1 strip next',        /*     =PARM.name=value       */
    '/=/ next',
    'fields 2-* strip next |',
  'varload'                      /* Load into my environment.  */
```

That pipeline parses a parameter file that has REXX-like syntax and converts each entry from the file into a variable whose name is in the form "PARM.entryname" and whose value is the value specified in the file. It formats the names and values of these variables into records in the format required by the varload stage in order to load the variables into its REXX environment. Then, after the parameter file has been processed, the subroutine pokes about to determine some other values that will be needed by the server:

```
/*-------------------------------------------------------------*/
/*  Issue a CMS IDENTIFY command to get more items for the     */
/*  "PARM." array.                                             */
/*-------------------------------------------------------------*/

'IDENTIFY ( STACK LIFO'
Parse Pull parm.myself . parm.sysname . parm.rscsname . . ,
   parm.timezone .

/*-------------------------------------------------------------*/
/*  Use RXSOCKET to determine my host's Internet name.         */
/*-------------------------------------------------------------*/

Parse Value Socket('Initialize','Get') With RC .
Do While RC ¬= 0
   Say 'Waiting to initialize socket'
   'CP SLEEP 1 MIN'
   Parse Value Socket('Initialize','Get') With RC .
End
Parse Value Socket('GetHostName') With RC hostname
Parse Value Socket('GetDomainName') With RC domainname
Parse Value Socket('Terminate','Get') With RC .

parm.hostnode = hostname'.'domainname /* My Internet host name.*/
```

When it has finished building the PARM. variables, it extracts them from its REXX environment and stores them in its caller's REXX environment:

```
/*-------------------------------------------------------------*/
/*  Load the "PARM." array into the caller's REXX environment. */
/*-------------------------------------------------------------*/

'PIPE (name ParmSet3)',          /* "PARM." stem into caller:   */
   'rexxvars toload |',          /* Load all my variables.      */
   'find  PARM.|',               /* Select PARM. entries.       */
   'varload 1 |',                /* Load into caller.           */
   'stem parmsave. 1'            /* And save VARLOADable, too.  */
```

rexxvars toload loads all exposed REXX variables into this pipeline and writes records in the format expected by varload. The find stage selects the records that are of interest. varload 1 stores them in the REXX environment one level back; that is, it stores them in the EXEC that called this EXEC, so that it can use them in its processing. stem parmsave. 1 also stores the PARM. values in the caller, but instead of storing them as individual variables, it stores them as the stemmed array PARMSAVE., in the format expected by the varload pipeline stage. Then,

when the caller runs the main pipeline for this application, any REXX stage can load the PARM. variables into its own REXX environment by using this subroutine pipeline:

```
'CALLPIPE (name ParmLoad)',
   'stem parmsave. main |',     /* Load parms from main EXEC.  */
   'varload'                    /* Load into my environment.   */
```

The main option on the stem stage says to reference the REXX environment that issued the PIPE command. If you are running a version of *CMS Pipelines* that doesn't support the main option, you can specify the number of environments to go back. If you are running a version that doesn't support the toload option, it isn't difficult to achieve the same effect with a few pipeline stages.

IV. DELAY

In writing servers, one often needs to perform actions at specific times on certain days or after a specific interval. Both are easy to do in *CMS Pipelines* by using the delay stage. In this example, any Class C reader files are taken out of hold once every 45 minutes, and the calling pipeline is then prodded to examine them:

```
'CALLPIPE (name DelayCmd)',
  '  literal +45:00',           /* Once every 45 minutes.      */
  '| duplicate *',              /* Forever.                    */
  '| delay',                    /* Wake up.                    */
  '| spec /CHANGE RDR CLASS C NOHOLD/',  /* Make a CHANGE cmd. */
  '| cp',                       /* Release Class C files.      */
  '| nlocate /NO/',             /* If no files, don't prime.   */
  '| spec /00000004*       RDR FILE/',   /* Fake file arrival. */
  '| *:'                       /* Prime the pipeline.         */
```

The literal creates a record that says "+45:00", and that record is duplicated repeatedly. delay reads a record, interprets it as a request to wait for 45 minutes and does that. When it wakes up, it writes the record to its output and reads another from its input, which causes it to wait for another 45 minutes. The record delay wrote to its output is converted into a CP CHANGE RDR command, which is then executed. If the response from the CHANGE command is not "NO FILES CHANGED", the response is changed into a record that simulates a reader file arrival message. That is fed into the calling pipeline to prompt it to take some action.

Waking up at a specific time is equally easy. This fragment is from a piece of code that sets the variable "delay" to the next time the CP monitor should be started, 9:45 or 13:45 on a weekday:

```
Select
  When today = 'Friday' & ,          /* After 15:45 Friday?     */
    minutes >= 945
      Then delay = '81:45'           /* Wait til 09:45 Monday.  */
  When today = 'Saturday'            /* Saturday?               */
    Then delay = '57:45'             /* Wait til 09:45 Monday.  */
  When today = 'Sunday'              /* Sunday?                 */
    Then delay = '33:45'             /* Wait til 09:45 Monday.  */
  . . .
```

(The time of day is in terms of a 24-hour clock, but the value can be specified non-traditionally.) Once the variable “delay” has been set, this pipeline is invoked to wait:

```
'PIPE (endchar ?)',
     'literal' delay '|',            /* Time we need to wake up.*/
     'delay |',                      /* Wait until then.        */
     'specs /Delay expired./ 1 |',   /* Remember what happened. */
  'fin: faninany |',                 /* One of two reasons.     */
     'take 1 |',                     /* Stop on first one.      */
     'var reason',                   /* Save reason.            */
'?',
     'immcmd STOP|',                 /* Stop if we're asked to. */
     'specs /Stopped./ 1 |',         /* Remember what happened. */
  'fin:'                             /* Feed to FANINANY.       */
```

This pipeline terminates as soon as a record reaches the take 1 stage. That may be the result of the expiration of the timer or of a “STOP” command being typed on the console. The reason for stopping is stored in the variable “reason”, to allow the EXEC to decide what to do next.

Similarly, this subroutine pipeline passes a record to its caller at midnight every night:

```
'CALLPIPE',
  'literal +24:00:00 |',             /* Once a day.             */
  'duplicate * |',                   /* Forever.                */
  'literal 23:59:59 |',              /* At midnight.            */
  'delay |',                         /* Get a timer pop.        */
  'spec /STATS: QUOTA CLEANUP/ |',   /* Build a CLEANUP cmd.    */
  '*:'                               /* Write into pipeline.    */
```

V. MANAGING A LOG FILE

In most cases, it is best to make your server one big pipeline with one or more streams running from end-to-end. I have found it convenient to have a stream running the length of my servers for sending records into a stage that manages the server's log file. This example is from a stage that is used by a server that needs to preserve its log in case of a system crash:

```
/*-------------------------------------------------------------*/
/*  Connect to the main pipeline and read in records to be     */
/*  written to a log file.  End the subroutine pipeline when   */
/*  4000 records have been written and then send the file to   */
/*  the specified recipient, erase it, and start over again.   */
/*  If the input stream is severed (mostly likely because the  */
/*  SHUTDOWN EXEC on OPERATOR has sent us a SHUTDOWN command), */
/*  write and send whatever we have received; then terminate.  */
/*-------------------------------------------------------------*/

Do Forever                        /* Do until end-of-file.       */

  'PEEKTO'                        /* Notice end-of-file.         */

  'CALLPIPE (name ManageLog)',    /* Print log every 4000 lines: */
    '*: |',                       /* Input from pipeline.        */
    'take 4000 |',                /* Get a file's worth (or eof).*/
    'pad 1 |',                    /* Retain null lines.          */
    'chop 132 |',                 /* Make sure printable.        */
    'diskslow' file '|',          /* Don't block writes.         */
    'spec',                       /* Convert every fifth...      */
      'read read read read',      /* ...record that is written...*/
      '/FINIS * * A/ 1 |',        /* ...into a FINIS command.    */
    'command'                     /* Issue the FINIS commands.   */

  Address Command 'CP SPOOL PRT TO' parm.loguser
  Address Command 'PRINT' file    /* Print the log file.         */
  Address Command 'ERASE' file    /* Erase the log file.         */

End
```

diskslow is used in this case because it does not block its output, but instead writes out every record immediately. Every fifth record passed to diskslow here is subsequently turned into a FINIS command that is issued by the command stage, thus checkpointing the log file frequently. (Obviously, one should avoid random FINISes if the server has other files open, as this stage runs concurrently with the rest of the pipeline.)

VI. TRIGGERING MULTIPLE EVENTS

In a complex pipeline, you may sometimes find it useful to let a single record trigger multiple events. In this example, which carries that idea to the extreme, when the user types a “STOP” command on the virtual machine console, it triggers events all over the pipeline:

```
'CALLPIPE',
     'immcmd STOP |',           /* Have we got a STOP command? */
  'o: fanout |',                /* Yes, make multiple copies.  */
     'spec',                    /* Make this one pretty.       */
       '/ Console command is:/ 1',
       '1-* nextword |',
     'timestamp 16 |'           /* Prefix date and time.       */
     '*.output.1:',             /* Send it to the log file.    */
'?',
  'o: |',                       /* Second copy to here.        */
     'pipestop',                /* Use it to stop externals.   */
'?',
  'o: |',                       /* Third copy to here.         */
  'l: find RELOAD|',            /* Is it STOP RELOAD?          */
     'spec /1/ |',              /* Yes, make Boolean.          */
  'f: faninany |',              /* Collect Boolean value.      */
     'var reload 1 tracking |', /* Store in caller.           */
     'find 1|',                 /* Was it CMD STOP RELOAD?     */
     'spec /EXEC PROFILE/ |',   /* Yes, make it "EXEC PROFILE".*/
     'stack',                   /* Stack that command.         */
'?',
  'l: |',                       /* Not STOP RELOAD.            */
     'spec /0/ |',              /* Make Boolean.               */
  'f:',                         /* Send it into FANINANY.      */
'?',
  'o: |',                       /* Fourth copy to here.        */
     '*.output.0:'              /* Feed into the GATE stage.   */
```

When the immcmd stage receives a record, fanout makes four copies. The first is garnished and time-stamped and written to the secondary output stream of the stage that issued the callpipe; that stream is connected to the input of a logging stage much like the one in the previous example. The second copy of the record is written to a pipestop stage, thus terminating all stages throughout the pipeline complex that are waiting on external events. The third copy is examined to see whether the user specified the RELOAD option. If he did, that record is transformed into the numeral “1”; otherwise, it is transformed into a “0”. The numeral record is passed to a var tracking stage, which stores it as a Boolean variable in the caller’s REXX environment. If the value is “1”, it is then turned into an EXEC PROFILE command, which is stacked in the CMS stack to start the server up afresh after it has stopped. The fourth copy is written to this stage’s primary output stream, which is connected to the primary input of a gate stage; that causes gate to terminate, which will terminate other portions of the pipeline. (For a discussion of gate and other aspects of pipeline termination, see Appendix D in my handout.)

VII. SYNCHRONISE

synchronise is used to force records on parallel streams of a pipeline to march through that pipeline in unison. synchronise waits until it has a record on each of its input streams and then copies those records to the corresponding output streams. It is especially useful for throttling back a stage, such as duplicate *, that can produce an infinite number of records. In this example, it is used to synchronise the processing of records with external events; one record is read from the calling pipeline and written back to it each time an SMSG is received:

```
/* PACER REXX:   Use external events to pace record processing */

'CALLPIPE (endchar ?)',
         '*.input: |',              /* Input from caller.      */
   'sync: synchronise |',           /* Correlate with SMSGes.  */
         '*.output:',               /* Output to caller.       */
'?',
         'starmsg CP SET SMSG IUCV |',      /* Capture SMSGes. */
   'sync: |',                       /* Synchronize with input. */
         'hole'                     /* Into the bit bucket.    */

saverc = RC
Address Command 'CP SET SMSG OFF'   /* Back to normal.         */
Exit saverc*(saverc<>12)            /* RC = 0 if end-of-file.  */
```

Only when the starmsg stage has captured an SMSG and has made it available on the secondary input stream of the synchronise stage does synchronise process the next record from its primary input stream. Then one record is read from each stream and copied to the corresponding output stream. Since synchronise's primary output stream is connected to the calling pipeline and its secondary output stream is connected to a hole stage, the record from the calling pipeline is passed back to the calling pipeline and the SMSGed record is discarded. Then no further records are processed until the next SMSG is received. This subroutine pipeline runs until synchronise encounters an end-of-file on any input or output stream. (That is the reason for the hole stage. The records on synchronise's secondary output stream are not needed by this application, but that stream must remain connected to prevent synchronise from terminating too soon.)

VIII. CONTROLLING A RUN-AWAY PROCESS

If your service machine must invoke unreliable programs, you may need a way of controlling those programs when they get into trouble. I have a set of service machines that perform FTPs by AUTOLOGing "robot" virtual machines and using the Single Console Image Facility (SCIF) to control them. A REXX filter sends FTP commands and subcommands to a robot:

```
Address Command 'CP SEND' parm.myrobot Arg(1)
```

Responses are captured by a starmsg stage and passed to the REXX filter on its secondary input stream, where they are examined for too many Pascal error messages:

```
/*-------------------------------------------------------------*/
/* Process the SCIF messages until a CMS prompt is received;   */
/* then disconnect from pipeline with messages stored and      */
/* "ftprc" set.                                                */
/*                                                             */
/* The CMS prompt is a message such as:                        */
/*      User BITFTP5R has issued a VM read                     */
/*                                                             */
/* The "console" stage here must come after the reference to   */
/* the "gate" stage, because "console" does not propagate end- */
/* of-file backwards.  Selection stages that divert records    */
/* to the "gate" stage must come before it to avoid a stall.   */
/*-------------------------------------------------------------*/

'CALLPIPE (endchar ? listerr name SendRobot)',
  'y: faninany |',              /* Get primary input for GATE. */
     'console |',               /* Display response.           */
     'drop 9 |',                /* Ignore until get 10th AMPX. */
     'take 1 |',                /* That one is enough.         */
     'count lines |',           /* Make it Boolean.            */
     'var gotampx |',           /* And set logical variable.   */
     'find 1|',                 /* Drive GATE only if true.    */
  'g: gate strict'              /* Force EOF on secondary.     */
  '?',
     '*.input.1: |',            /* Connect to secondary input. */
     'tolabel' has_issued'|',   /* Continue until CMS prompt.  */
     'find 00000008|',          /* Select only SCIF output.    */
     'spec 17-* 1 |',           /* Discard message header.     */
     'spec w2-* 1 |',           /* Discard robot's name.       */
     'nfind Command:|',         /* Discard FTP prompt.         */
     'stem result. |',          /* Save FTP responses.         */
  'x: nfind AMPX|',             /* Divert Pascal error message.*/
  'g: |',                       /* Run through gateway.        */
     'console |',               /* Display response.           */
      findrc,                   /* Set the FTP return code.    */
'? x: | y:'                     /* AMPX message to GATE pipe.  */

/*-------------------------------------------------------------*/
/*  If the number of AMPXnnn messages for this job is greater  */
/*  than 10, we are probably going to get a zillion of them,   */
/*  so bail out as quickly as we can.                          */
/*-------------------------------------------------------------*/

If gotampx  Then Do             /* If more than 10, serious.   */
  Address Command 'CP CHANGE * RDR' spoolid 'CLASS X'
  Call ToMWV                    /* Transfer the file away.     */
  Address Command 'CP SEND CP' parm.myrobot 'IPL CMS PARM AUTOCR'
  Signal Restart                /* Restart robot and self.     */
End
```

The SCIF messages enter the second pipeline segment. Under normal circumstances, the subroutine pipeline accepts all the SCIF messages until the tolabel stage encounters a CMS prompt, which indicates completion of the command that was sent to the robot. The response lines are winnowed down to the useful bits and stored in the RESULT. stem before this callpipe command completes.

Any Pascal error messages encountered among the SCIF responses are diverted to the first pipeline segment, where they are counted. When the tenth one arrives, the Boolean variable "gotampx" is set to "true", and a record is written to the primary input of gate, causing it to terminate. When gate terminates, end-of-file propagates forward from its secondary output and backward from its secondary input, causing the entire subroutine pipeline to terminate immediately. The variable "gotampx" can then be examined and appropriate actions taken.

IX. ADDPIPE

addpipe can make your complex servers much less complex. Appendix E in my handout includes a general introduction to addpipe, so I will just mention briefly some of the advantages of learning to use addpipe.

One good thing you can do with addpipe is prefix or suffix sections of pipeline to a REXX filter. The reason for doing this is to hide complexity. Here is an addpipe issued by a REXX filter in a screen-scraper application, a service machine that intervenes between MVS/ESA and VM Programmable Operator:

```
'ADDPIPE (endchar ?)',           /* Suffixed pipeline.         */
     '*.output: |',              /* Get output from this stage.*/
  'l: locate 1600 |',            /* MVS init writes 1600-byte..*/
     'nlocate 1601 |',           /* ..blocks to screen.        */
     'deblock 80 |',             /* Deblock to lines.          */
  'f: faninany |',
     'xlate 2.2 0-9 40 |',       /* Blank outstanding reply.   */
     'change 3-4 /||/ / |',      /* Remove replied signal.     */
     'nfind _IEE163I_MODE=_R |', /* Discard noise from bottom. */
     'strip trailing |',         /* Remove blank lines.        */
     'locate 1 |',
     'pad 80 |',                 /* Make all lines same size.  */
     '*.output:',                /* Output to stage's output.  */
'?',
  'l: |',
     'locate 79 |',              /* Rest of MVS writes...      */
     'nlocate 80 |',             /* ..79-byte blocks.          */
  'f:'
```

The REXX filter that issues this addpipe interprets the orders in a 3270 datastream and writes reformatted data to its output. Because this addpipe is suffixed to its output stream, additional cleanup is done as the records leave the REXX filter. Full screens are deblocked to lines,

miscellaneous squiggles are erased, and uninteresting lines are discarded. The same operations could have been done in REXX or by a callpipe in the filter itself, but either of those would have been more expensive. The same stages could have been inserted into the main pipeline after this REXX filter, but that would have increased the apparent complexity of the main pipeline. The function these stages perform is logically part of this REXX filter, so it is better to hide them in the filter.

Another good reason to use addpipe in a server is that it can allow you to spawn a separate operation for each request that the server receives. addpipe runs asynchronously with the stage that issues it, so control is returned as soon as the new pipeline has been created. If your pipeline does nothing that causes *Pipes* to lose control for long periods, you can get the appearance of multi-tasking very easily and allow short requests to be processed while longer tasks are still running. The paper by Rob van der Heij that is included among our handouts gives an elegant example of doing just that.

X. REXX PIPELINE COMMAND

If you find yourself writing very large REXX filters for your server and want a way to make your code more modular, you should be aware that you can use the REXX pipeline command to invoke a REXX program from your REXX filter. Using the pipeline command REXX in a REXX filter is equivalent to calling an external function in a REXX command procedure. It can also be a handy way of invoking user exits:

```
Address Command 'STATE BFUSER1 REXX *'  /* User exit provided? */
If RC = 0  Then                        /* Yes, invoke user exit. */
   'REXX bfuser1' Uid Unode spoolid
```

The called program has access to the streams belonging to the filter that invoked it. It can also use the REXX device drivers in subroutine pipelines to manipulate REXX variables in the filter that invoked it.

XI. PERFORMANCE ANALYSIS

After I had written a few servers that made heavy use of *CMS Pipelines*, I found it necessary to write a pipeline execution profiler, so that I could improve the performance of those servers. This program, which is named Rita, is available on VMSHARE and on the 1994 VM Workshop Tools Tape. It is quite easy to use; you simply change a PIPE command to RITA and run your program. When your program terminates, Rita will produce a report showing the CPU utilization of every pipeline stage. That should allow you to spot the areas that will most benefit from attention.

Appendix A

ANOTHER SMALL EXAMPLE

This is a small service machine implemented in *Pipes* that illustrates the use of delay, starmsg, immcmd, and PIPMOD STOP. The guts of the server are hidden in a single subroutine called getcmd:

```
/*  GETCMD REXX:  Feed requests to higher level of the server    */

primer = '00000004*       RDR FILE **** FROM'    /* Constant.    */

'CALLPIPE (endchar ?)',
  '| immcmd CMD',                        /* Immediate commands.   */
  '| spec /00000004*/ 1.16 1-* next',    /* As if SMSG from self. */
  '| f: faninany',                       /* Join all commands.    */
  '| *:',                                /* Pass to caller.       */
  '? starmsg',                           /* Listen for SMSGes.    */
  '| f:',                                /* Pass to caller.       */
  '? literal' primer 'startup',          /* Prime the pipeline.   */
  '| f:',                                /* Pass to caller.       */
  '? delaycmd',                          /* Timer-driven retries. */
  '| f:',                                /* Pass to caller.       */
  '? immcmd CP',                         /* CP immediate command. */
  '| cp',                                /* Pass to CP.           */
  '? immcmd STOP',                       /* STOP command.         */
  '| spec /PIPMOD STOP/',                /* Build PIPMOD STOP.    */
  '| subcom cms'                         /* Pass to CMS.          */
```

getcmd receives all kinds of input:

- Commands typed on the virtual machine console are trapped by the immcmd stages. If a STOP command is typed, then a PIPMOD STOP command is given to CMS to stop the stages in the pipeline that are waiting on external events and timers. If a CP command is typed, then it is given to CP. If a command intended for the server (prefaced by "CMD") is typed, then it is sent through the faninany stage and the connector into the calling pipeline.

- Commands SMSGed from other virtual machines are received by the starmsg stage and are also fanned in and sent through the connector to the calling pipeline.

- Messages generated by the arrival of spool files are trapped by the same starmsg (all classes of messages have been set to IUCV), so they also go to the calling pipeline.

- A literal stage fires once when the pipeline is first invoked. This sends a "primer" line into the calling pipeline to get it to start processing.

- Timer interrupts are generated by a subroutine called delaycmd, which feeds a line into getcmd each time it has done something for which the main (calling) pipeline should be wakened. (getcmd fans those lines in and sends them through the connector to the caller, just as it does with everything else.)

Here's a portion of delaycmd:

```
'CALLPIPE (endchar ?)',
  '  literal 47:00:00',                  /* 11pm tomorrow.        */
  '| duplicate *',                       /* Forever.              */
  '| literal 23:00:00',                  /* 11pm today (or now).  */
  '| delay',
  '| spec /CHANGE RDR CLASS K NOHOLD/',
  '| cp',                                /* Release Class K files.*/
  '| nlocate /NO/',                      /* Forget it if no files.*/
  '| spec /'primer'/ 1 / class k delay/ next'   /* Fake arrival. */
  '| f: faninany',                       /* Join all messages.    */
  '| *:',                                /* Pass to GETCMD.       */
  '? literal +240:00',                   /* Four-hour interval.   */
  '| duplicate *',
  '| delay',
  '| spec /CHANGE RDR CLASS J NOHOLD/',
  '| cp',                                /* Release Class J files.*/
  '| nlocate /NO/',                      /* Forget it if no files.*/
  '| spec /'primer'/ 1 / class j delay/ next',  /* Fake arrival. */
  '| f:'                                 /* Pass to GETCMD.       */
   . . .
```

Thus, the main pipeline starts out by invoking getcmd:

```
'PIPE (endchar ?)',
   'getcmd |',                      /* Wait for something to do. */
    . . .                           /* Process requests.         */
```

All its input comes from getcmd, arriving in whatever order the various events (reader files, console commands, timer-driven events, SMSGed commands) occur and whenever they occur. The rest of the pipeline is simply an extensive decoding network that looks at the single record received from getcmd for each event and decides what action to take in that case.

Appendix B

REQUESTING PERMISSION OR AUTHENTICATION

One of our service machines is actually a complex of several virtual machines. A central driver dispatches work to a group of identical servers. There is a limit on the amount of data this complex will send to a given user each day, so the driver machine keeps track of the total number of bytes sent to each user by the servers. When a server is ready to begin processing a request for a user, it asks permission to proceed by sending a CP message to the driver:

```
beginmsg = ' Begin job for' mailto        /* Message for driver. */

'CALLPIPE (name SendBeg)',         /* Tell driver beginning:      */
  'var beginmsg |',                /* Load in BEGINMSG.           */
  'timestamp 16 |',                /* Time-stamp it.              */
  'change //MSG' parm.driver 'STATS: / |',    /* Format command. */
  'cp'                             /* Issue MSG command.          */
```

It then invokes another subroutine pipeline to wait for the response from the driver. A starmsg stage elsewhere in the pipeline captures the response from the server and feeds it into the stage's tertiary input. If the driver doesn't reply within 30 seconds, the server proceeds without waiting further. If the driver does respond, its response is split into three words, which are stored in a stem for further examination:

```
'CALLPIPE (endchar ? name Proceed)',  /* Get driver's response:  */
     '*.input.2: |',               /* Will come on tertiary input.*/
  'f: faninany |',                 /* Either that or a timer pop. */
     'take 1 |',                   /* Need only one record.       */
     'locate 1 |',                 /* Discard if it's null.       */
     'split |',                    /* Split to get three values.  */
     'stem proceed.',              /* Store in stem.              */
'?',
     'literal +30|',               /* Thirty seconds.             */
     'delay |',                    /* Wait that long.             */
     'change /+30// |',            /* Make record null.           */
  'f:'                             /* Send to faninany.           */
```

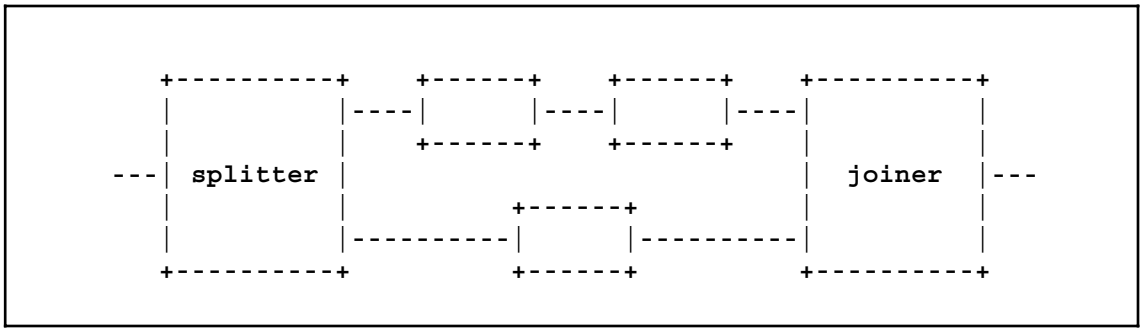
After the server has completed a request, it sends a similar message to the driver to tell it how many bytes of data were sent to the user. starmsg captures the messages from the servers, and at midnight, a subroutine in the driver (shown at the bottom of page 7) simulates a message to cause the usage counts to be reset. The driver's code to process these messages is quite simple; the messages are fed into a REXX filter, the central portion of which is shown here:

```
/*-----------------------------------------------------------------*/
/* Loop forever processing server messages as they arrive.         */
/*-----------------------------------------------------------------*/

count. = 0                              /* Initialize usage.       */

Do Forever
   'PEEKTO message'                     /* Wait for a message.     */
   Parse Upper Var message caller . . function .

   Select
      When function == 'CLEANUP'        /* It's midnight.          */
      Then Do                           /* All is forgiven.        */
         Drop count.                    /* Drop the stem.          */
         count. = 0                     /* Reinitialize it.        */
      End
      When function == 'BEGIN'          /* Starting a request.     */
      Then Do
         Parse Upper Var message . . . . . . mailto
         bytes = 0                      /* No data sent yet.       */
         Call Permission                /* Send STATRESP to caller.*/
      End
      Otherwise                         /* Completing a request.   */
         Parse Upper Var message . . . bytes . . . . mailto
         Call Permission                /* Send STATRESP to caller.*/
   End

   'OUTPUT' message                     /* Write message to log.   */
   'READTO'                             /* Unblock producer.       */
End

Error:  Exit RC*(RC<>12)                /* RC = 0 if end-of-file.  */

/*-----------------------------------------------------------------*/
/*                     Permission Subroutine                       */
/*-----------------------------------------------------------------*/

Permission:                             /* Record usage and reply. */

count.mailto = count.mailto + bytes     /* Increment counter.      */

response = (count.mailto < parm.rscsdaily),     /* Set values.     */
           (count.mailto < parm.maildaily),
           (count.mailto < parm.uucpdaily)

Address Command 'CP MSG' caller 'STATRESP' response

Return
```

Appendix C

ON RECORD DELAY AND PIPELINE STALLS

When writing service machines, it can be very useful to know how to reason about the flow of records passing through multi-stream pipelines, such as the one below. A "splitter" stage writes records to two or more output streams; later in the pipeline, a "joiner" stage reads records from those same streams:



When any stage writes an output record, it remains "blocked" until that record has been "consumed" by the stage connected to its output stream. That is, it does not regain control until after the record has been read with the equivalent of the pipeline command readto. However, most pipeline stages do not consume a record when they first read it. They first read the record with the equivalent of a peekto command, which does not consume it. They do not consume the record until after they have written it to their own output stream and it has been consumed by the stage to which they wrote it. (Stages that process records in this way are said not to "delay the record".) Thus, when splitter stages, such as locate, drop, fanout, and others, write a record on one output stream, they must then wait until that record has been consumed before they can write another record to any of their output streams.

If all the stages in the multi-stream portions of a pipeline like the one shown above are of the sort that do not delay the record, each of them passes the record along without consuming it until after each of the subsequent stages has consumed it. Ultimately, then, this splitter stage must wait for the joiner stage to consume each record before the splitter can write the next one. If the joiner stage cannot consume a record that the splitter stage has written, the pipeline stalls.

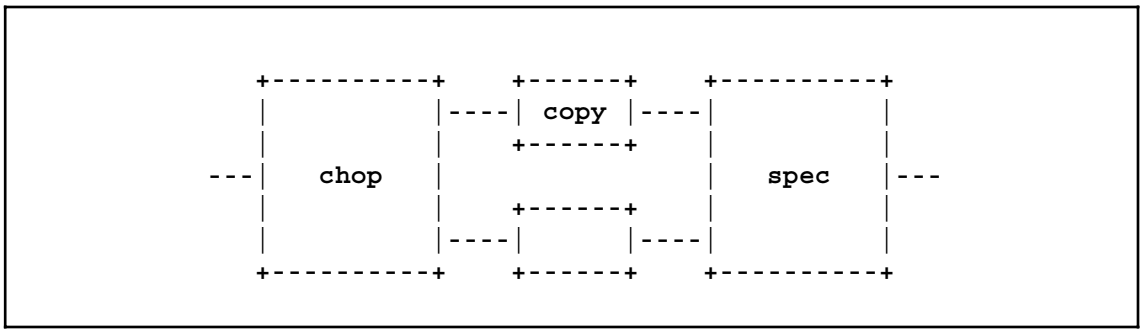
If the joiner stage is faninany, then this configuration will never stall, because faninany always reads any record that is available on any of its input streams. Other joiner stages are more exacting, however. Some, such as spec and synchronise, wait until they have a record available on each of their input streams before they consume any of them and then consume them in stream-number order. Others, such as collate and merge, wait until they have a record available on each of their input streams and then choose which one to consume based on the contents of the records. fanin is the most extreme; it consumes all the records from one input stream before it will read any records from any other input stream.

A further complication is that a stage in the multi-stream portion of a pipeline may "buffer" the records; that is, some stages, such as sort and instore, consume all their input records before writing any output records and, thus, may keep the joiner stage waiting for records on one of its input streams.

So, in a pipeline where there is a stage that reunites streams that originated in a single stage earlier in the pipeline, there is a potential for pipeline stalls. The splitter stage may try to write onto one stream while the joiner stage is trying to read from another stream. When that happens, the pipeline stalls.

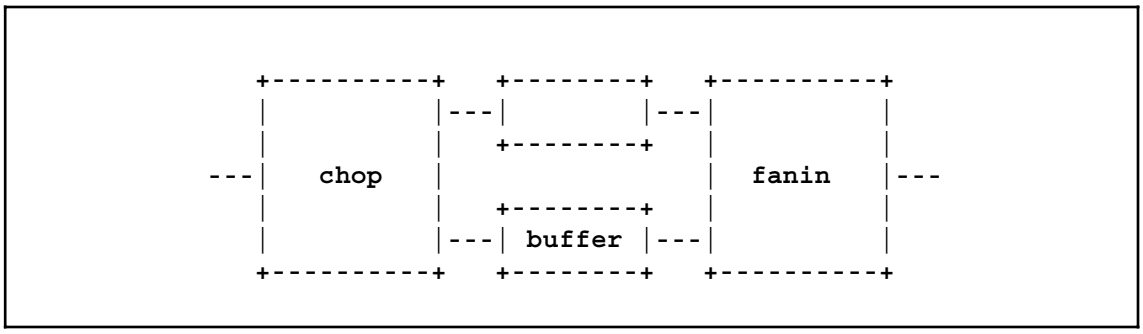
To prevent stalls, one inserts between the splitter and joiner stages (on one or more of the streams) a pipeline stage that will unblock the splitter stage by consuming the necessary number of records and holding them until the joiner stage is ready to read them.

The number of records that need to be "buffered" in this way varies. In the case with the least requirement for such buffering, the splitter stage writes a record to each of its output streams in rotation; the joiner stage reads records in stream-number order; and none of the intermediate stages delays the records. In this case, the flow of the records is not data-dependent, and a stall can be prevented simply by introducing a quantum delay on the low-numbered stream(s), using a copy stage:



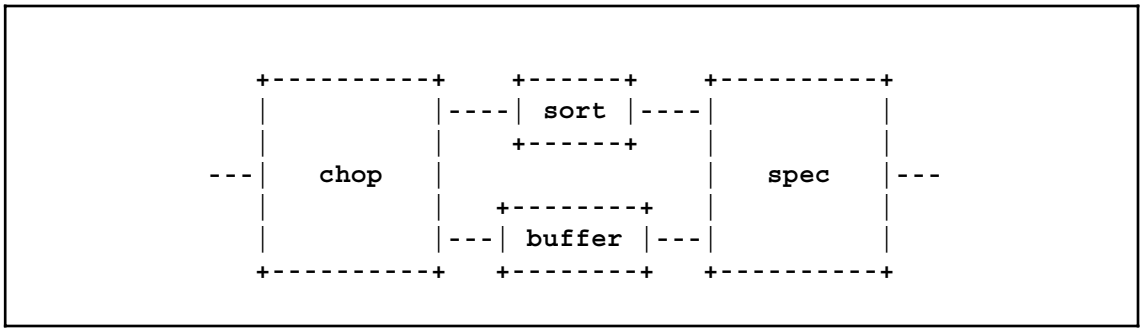
copy is a very simple stage consisting of a loop containing readto and output commands. It does a consuming read to get a record and then copies that record to its output stream. So, the copy stage here consumes the record that chop writes to its primary output stream, freeing chop to write a record to its secondary output stream. Meanwhile, copy writes the first record to its output stream. As a result, spec then finds records available on both of its input streams and consumes them both, freeing chop to write to its primary output stream again and freeing copy to read from that stream again.

At the other extreme in the amount of buffering required is the case where the joiner stage is fanin. Because fanin will read no records from its secondary stream until its has read all the records from its primary stream, one must introduce a stage such as buffer to consume all records on the secondary stream and hold them until fanin is ready for them:

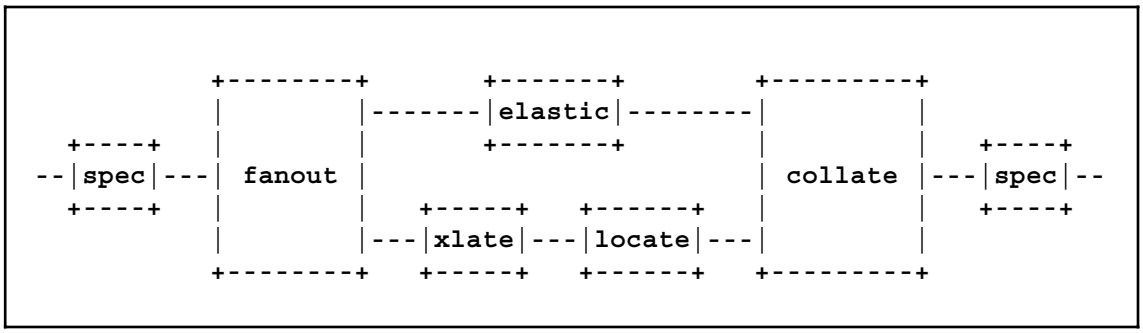


Otherwise, the splitter stage would become blocked trying to write the first record to fanin's secondary input.

The same requirement to insert a stage to buffer all the records on a stream may arise because one of the other streams contains a stage, such as sort, that buffers all the records:



In the intermediate case, one or more of the streams may need some buffering of records, depending on the order in which the splitter stage decides to write and the order in which the joiner stage decides to read. This is a job for elastic:



When elastic has only one input stream, as in this case, it copies its input records to its output, buffering as many as may be necessary to prevent a pipeline stall. It reads input records whenever they become available and writes output records as they are consumed, while attempting to minimize the number of records in its buffer.

In general, one can use elastic wherever buffering is needed to prevent a stall. However, if one knows that only a quantum delay is needed, then copy is more efficient than elastic, while if one knows that the entire file must be buffered, then buffer is more efficient than elastic. If the file being processed is large, it may be worth doing the analysis to distinguish the cases.

Appendix D

ON PIPELINE TERMINATION

In the simplest case, a pipeline consists of an input device driver, a few stages to manipulate the data, and an output device driver, and termination of such a pipeline is usually very straightforward. The input device driver reaches end-of-file on the device it is reading from and terminates. Its termination severs its output stream, causing the stage that was waiting to read from that stream to get an end-of-file on its input. That stage has nothing more to do, so it also terminates, causing its output stream to be severed and the next stage to receive end-of-file on its input. Very quickly, the entire pipeline collapses in domino fashion, with each stage getting a chance to do any end-of-file processing it may need to do before it terminates.

The first bit of complication that may arise is that one of the stages in the middle of the pipeline may decide to terminate before the input device driver has reached end-of-file. This might be, say, a take 5 stage. Once that stage has read five records and has copied them to its output stream, it terminates, causing both its input stream and its output stream to be severed. That causes the following stage to receive end-of-file on its input and to start the same domino termination effect as before. But it also causes the preceding stage to receive end-of-file the next time it tries to write an output record.

Many stages terminate "prematurely" when they receive end-of-file trying to write output. For example, if the stage is an input device driver, there is no point in its continuing to read from its device, if it has nowhere to write the records to, so it terminates. If the stage is a selection stage, such as find or locate, and it had only the one output stream connected, then it, too, will terminate when its output stream is severed. Again, it makes little sense to keep selecting records when no other stage is available to read the selected records.

So, when our take 5 terminates, end-of-file may propagate all the way back to the beginning of the pipeline, causing it to collapse as before, but with the dominoes falling from the middle toward both ends, rather than from beginning to end.

On the other hand, some stages do not terminate when they receive end-of-file on their output stream. For example, the host command processors, such as cp and cms, keep on issuing commands, and the output device drivers continue writing to their devices, even though they no longer have a connected output stream. They can still do useful work without being able to write to the pipeline, so there is no reason for them to stop. By the same logic, selection stages do not terminate when one output stream is severed, if the other output stream is still connected.

Thus, end-of-file may not propagate backwards in the pipeline through a branch or through certain stages. And this is certainly the way one would want things to work—segments of the pipeline that still have useful work to do keep on going, even though other segments may have completed. When their work is done, they, too, terminate.

However, this may sometimes produce results that are at first glance surprising. In the pipeline in the middle of page 7, for example, it is safe to insert a console stage after the take 1 stage, but not before it. If console comes before take 1, end-of-file will not propagate backwards through the console stage when take terminates, so something more drastic, such as a PIPMOD STOP command, will be required to terminate the delay and immcmd stages, which otherwise terminate only if their output streams are severed.

How a stage behaves when its output is severed is well documented in the author's help files; each of the relevant stages has a section entitled "Premature Termination". A stage's behavior in this respect is generally the same as that of all the other stages in its class and is generally what one would expect.

There are, not surprisingly, a few cases in which what one expects may vary with the circumstances. That is the reason the new stop option was added to fanout to allow the user to specify the number of output streams that must reach end-of-file before fanout will terminate. The default is for it to continue running until all of its output streams have reached end-of-file (unless, of course, its input stream is severed), but now one can force it to terminate when, say, only one of its output streams has reached end-of-file.

Typically, the built-in stages terminate when their input streams are severed, so in general end-of-file easily propagates forward as the input stream of each stage is severed, even though the pipeline may branch. However, stages that have multiple input streams may or may not terminate when just one of their input streams is severed. synchronise, for example, terminates when any one of its streams is severed, but spec, on the other hand, continues running until all of its input streams are severed. For the most part, these behaviors are what one would wish. However, spec, too, has been enhanced to have a stop option to specify how many of its input streams must be at end-of-file before it will stop.

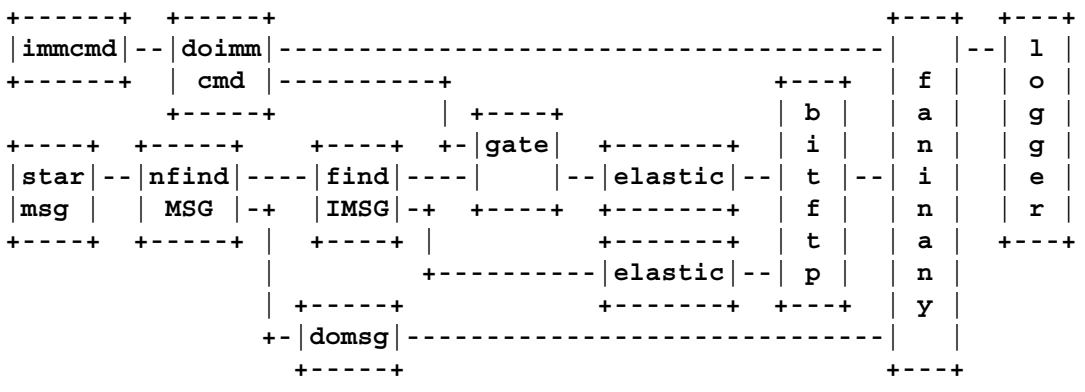
There is a set of input device drivers that never receive an end-of-file indication. These are the stages that take their input from external events or processes for which no end is defined. For example, immcmd accepts console commands and starmsg receives messages via IUCV. The delay stage, which waits for a timer expiration, is similar, although not an input device driver. All of the stages that wait on external events can be terminated by use of the PIPMOD STOP immediate command. PIPMOD STOP is not selective; it terminates every delay, immcmd, starmon, starmsg, starsys, and udp stage in the entire pipeline set. This is reasonable, as often one wishes everything to terminate once any one of several possible events occurs.

Another of the recent enhancements to *Pipes* is the addition of the pipestop stage. As soon as pipestop receives an input record, it performs the same function as PIPMOD STOP; that is, it posts all of the ECBs that are being waited on by other stages.

Some of the stages that can be stopped by PIPMOD STOP and pipestop can also be stopped by their own private immediate commands, which stop only the stage in question. For example, the HACCOUNT immediate command stops only a starsys *account stage. Using the immcmd stage, one has the ability to create other immediate commands to signal a single user-written stage to terminate. Armed with the appropriate immediate commands and the new gate stage, one can disassemble a complex pipeline much more gracefully than by using PIPMOD STOP.

In general, terminating a pipeline in an orderly fashion gets to be something one needs to think about only when the pipeline becomes rather complex and has more than one stage waiting on external events. Then it may become a satisfyingly intricate process, somewhat akin to designing fjords. For this reason, I was grateful for the addition of gate (which is included in CMS 10 but not documented until CMS 11).

```
'PIPE (endchar ?)',              /* Using GATE to terminate:    */
      'immcmd CMD|',             /* Capture immediate commands. */
   'd: doimmcmd |',              /* Analyze and process them.   */
   'f: faninany |',              /* Gather records for log file.*/
      'logger',                  /* Manage the log files.       */
'?',
   'd: |',                       /* DOIMMCMD's STOP signal.     */
   'g: gate strict',             /* Terminate when get signal.  */
'?',
      'starmsg |',               /* Connect to CP *MSG service. */
   'u: nfind 00000001|',         /* Divert user messages.       */
   's: find 00000007|',          /* Divert SCIF; keep IMSGes.   */
      'doimsg |',                /* Select interesting IMSGes.  */
   'g: |',                       /* Run through gateway.        */
      'elastic |',               /* Buffer a few IMSG records.  */
   'b: bitftp |',                /* Run the FTP requests.       */
   'f:',                         /* Write them to the log file. */
'?',
   's: |',                       /* SCIF messages to here.      */
      'elastic |',               /* Buffering needed here, too. */
   'b:',                         /* Into BITFTP's 2ry input.    */
'?',
   'u: |',                       /* User messages to here.      */
      'domsg |',                 /* Respond to user queries.    */
   'f:'                          /* Write to the log file.      */
```



The purpose of gate is to spread end-of-file in a delicate manner. gate terminates when it reads a record on its primary input stream; until then, records it reads on other streams are simply copied to the corresponding output stream. When the option strict is specified, gate checks the state of its primary input stream each time it is ready to copy a record to another output stream; when strict is omitted, gate stops the next time its primary input stream is selected.

The example shown on the preceding page illustrates the use of gate. This is the central routine of a service machine. It has two input stages, immcmd and starmsg. starmsg receives three kinds of messages: user messages, which are processed by the domsg stage, CP information messages, which form the primary input of the bitftp stage, and SCIF (secondary console interface) messages, which form the secondary input of the bitftp stage. One of the immediate commands that this pipeline accepts is STOP, but the familiar practice of turning that immediate command into a PIPMOD STOP command and issuing that to terminate the pipeline will not do in this case. The bitftp stage may be running a transaction that should be allowed to complete before the pipeline terminates. If a PIPMOD STOP command were issued, the starmsg stage would be terminated. That would mean that the bitftp stage could receive no more SCIF messages. It would, therefore, be unable to complete its transaction, which involves driving a process in another virtual machine. In fact, the bitftp stage would be unable to complete its transaction if starmsg, nfind, find, or the second elastic stage were terminated, because they form the path through which it receives the SCIF messages. Furthermore, it would be unable to log its transaction if the faninany or logger stages were terminated too soon.

However, by using gate, one can arrange things so that pulling the plug on this pipeline is no problem. When the doimmcmd stage receives a STOP command from the immcmd stage, it first writes a message to the logger stage (via the faninany stage), then sends a record to the gate stage on gate's primary input stream, and finally terminates. Its termination causes the immcmd stage to terminate, because (like most other input device drivers) immcmd terminates when its output stream has been severed. When gate receives the record from doimmcmd on its primary input stream, it also terminates, which severs its other input stream and its output stream. The severing of its output stream causes the elastic stage that had been connected to that stream also to terminate, severing its connection to the primary input of the bitftp stage. The fact that gate's secondary input stream has been severed does not affect the find stage, whose primary output was connected to that stream; find still has its secondary output connected, so it continues to run. Similarly, faninany continues to run despite the severing of its primary input stream when the doimmcmd stage terminated.

The remaining stages continue to run while bitftp finishes its transaction. It then reads from its primary input stream to get an IMSG describing its next transaction. When it discovers that its primary input has been severed, it issues a PIPMOD STOP command, which terminates the starmsg stage, starting a domino effect that results in the termination of nfind, find, elastic, and domsg, but leaves faninany running, because it still has an input stream connected. bitftp then terminates itself, resulting in the termination of faninany and finally logger (but not until after logger has logged the final messages from all the other stages).

Appendix E

ON ADDPIPE

The addpipe command is used to add one or more pipelines to the set of running pipelines and allow the stage that issued the addpipe to continue to execute in parallel with the newly added pipelines.[1]

When the addpipe command is invoked to add a new pipeline, the resulting changes in pipeline topology may include the breaking of connections between the invoking stage and other stages. A stream connected to a stage that issues an addpipe will be disconnected from that stage if it is referenced by a connector in the new pipeline.

What happens to the disconnected stream depends on the configuration of the connector that references it:

- It may become permanently connected to the new pipeline and have no further connection to the stage that invoked addpipe.
- It may be suspended while another stream temporarily occupies its connection to the stage that invoked addpipe.
- It may be connected to the new pipeline, which is in turn connected to the stage that invoked addpipe.

Thus, connectors may be specified in three possible configurations:

- A "redefine connector" detaches a stream from the stage that invoked addpipe and permanently attaches it to the new pipeline. A connector is a redefine connector when it is either:

  — An input connector at the beginning of a pipeline, or
  — An output connector at the end of a pipeline.

  This is an example of a redefine connector that transfers the invoking stage's input stream to an added pipeline:

  ```
  addpipe *.input: | xlate upper | > output file a
  ```

  After this command is issued, the stage that issued it will get an end-of-file indication if it tries to read its input stream, because its input stream is no longer connected. The diverted input stream is processed by the new pipeline, which upper-cases the records as they pass through and then writes them to a file.

  The following pipeline, which has redefine connectors at both ends, is identical to a pipeline short operation:

---

[1] This section represents an attempt to elaborate on the discussion of addpipe in the *CMS Pipelines User's Guide and Filter Reference* (SL26-0018). It is heavily indebted to both that manual and the *Toolsmith's Guide* (SL26-0020).

```
addpipe *.input: | *.output:
```

The stage that issues this addpipe command transfers both its input stream and its output stream to the added pipeline (which connects them to one another).

- A "prefix connector" suspends a connection between the stage that invokes addpipe and another stage and replaces that connection with one between the new pipeline and the invoking stage. If it is an input connector, records flow from the added pipeline through the connector into the invoking stage's input stream. If it is an output connector, records flow from the invoking stage's output stream through the connector into the added pipeline. A connector is a prefix connector when it is either:

  — An input connector at the end of a pipeline, or
  — An output connector at the beginning of a pipeline.

  When a stream is connected to a new pipeline with a prefix connector, the old connection is saved on a stack. End-of-file on the new connection sets return code 12 for a readto, peekto, or output command issued by the invoking stage. A sever command can then be used to restore the stacked connection.

  Here is an example of a prefix connector that temporarily connects an added pipeline to the input stream of the invoking stage (for the purpose of allowing that stage to read a parameter file before beginning its main work):

```
"ADDPIPE < parm file| *.input:" /* Connect input to parm file. */
"NOCOMMIT"                      /* Disable automatic commit.   */
"READTO line"                   /* Read first line of file.    */
do while RC=0                   /* Keep reading until EOF.     */
   "READTO line"
end
"SEVER input"                   /* Re-instate input stream.    */
"COMMIT 0"                      /* See if other stages are OK. */
if RC<>0 then exit 0            /* Exit quietly if not.        */
```

  The addpipe command takes over the stage's input stream and begins reading the parameter file into it. The readto commands that the stage subsequently issues get the parameter records that were put into its input stream by the added pipeline. After the stage has read the last of those records, its next readto gets a return code 12 (end-of-file), which causes it to exit from the Do While group. It then issues a sever command, which re-instates its original input stream, and a commit 0 command, which waits for the other stages to be ready for data to flow through the pipeline.

- "Hybrid connectors" cause a stream to flow through both the added pipeline and the stage that added the pipeline. A pipeline has hybrid connectors when it has either:

  — Input connectors at both ends, or
  — Output connectors at both ends.

When the added pipeline has input connectors at both ends, the input stream for the stage flows through the added pipeline before it becomes available to the stage. For example, this addpipe command would cause the input records for the stage to be deblocked before being read by any readto commands in the stage:

```
addpipe *.input: | deblock net | *.input:
```

When the added pipeline has output connectors at both ends, the output stream from the stage flows through the added pipeline before flowing into another stage. For example, this addpipe command would cause any records produced by output commands in the invoking stage to be upper-cased before being passed to the next stage:

```
addpipe *.output: | xlate upper | *.output:
```

A single stage may invoke the addpipe command more than once. When the added pipelines use prefix or hybrid connectors, a given stream may be referenced repeatedly, resulting in stacking of its connections.

Examples of using Addpipe

Addpipe with no connectors: A pipeline created by addpipe need have no connections to any of the other pipelines in the pipeline set. It may simply run in parallel with the other pipelines without data flowing between them. One way such an unconnected pipeline can be useful is in monitoring long-running pipelines, such as service machines. For example, the following addpipe might be used in a service machine that receives requests in the form of reader files:

```
'ADDPIPE (name THI)',              /* Timer-driven displays.   */
   '  literal +1:00',              /* Once a minute.           */
   '| duplicate *',                /* Forever.                 */
   '| delay',                      /* Wait for timer pop.      */
   '| specs /QUERY FILES/ 1',      /* Format Q FILES command.  */
   '| cp',                         /* Give it to CP.           */
   '| nlocate /NO RDR/',           /* Be quiet if caught up.   */
   '| specs',                      /* Format the display.      */
        '/Backlog is:/ 1',
        '7.5           nextword',
        '/files./      nextword',
   '| console'                     /* Put it on the console.   */
```

The added pipeline displays the backlog of requests once per minute, operating independently of the pipelines that are processing the requests.

Addpipe with hybrid input connectors: This example of using addpipe with hybrid input connectors is from Chuck Boeheim. It is a variant of the readlist stage discussed in the body of the paper. The pipeline added by the addpipe command processes the input stream for this stage before it is read by the readto command, converting each record into a traditional CMS ":READ" card, which is later written to the output stream by the output command before the subroutine

pipeline created by the callpipe writes the contents of the specified file to the output stream. Note the use of lookup and the read keyword of spec:

```
/* GETFDATE REXX:     Send contents of files into the pipe       */
/*                    preceded by a :READ record bearing the     */
/*                    filename, disk label, and timestamp.       */

/* Input: filename;  Output: ":READ" card followed by the file.  */

Signal On Error

/* Add a subroutine pipe before this stage to put the filenames  */
/* into the format we need.                                      */

'AddPipe (endchar ?)'           ,  /* Pre-process input stream.. */
  '  *.input:'                  ,  /* ..for this stage.          */
  '| nfind *'                 || ,  /* Remove comments.           */
  '| nfind &TRACE'            || ,  /* And any control statement. */
  '| change /&1 &2 //'          ,  /* Get rid of exec args.      */
  '| change /&3 //'             ,  /* Some lists have three.     */
  '| state'                     ,  /* Ask CMS for the FST info.  */
  '| disk: lookup 19.1 15.1'    ,  /* Look up the disk info.     */
  '        detail master'      ,  /*                            */
  '| spec /:READ / 1'           ,  /* Put :READ word in,         */
  '       1.20     8'          ,  /*   then the fileid,         */
  '       57.17   36'          ,  /*   then the timestamp; and  */
  '       read'                ,  /*   from the next record,    */
  '       1.6     29'          ,  /*   get the disk label.      */
  '| *.input:'                 ,  /* Connect to ourselves.      */
'?'                             ,
  '  cms query search'          ,  /* Generate the disk table.   */
  '| disk:'                    ,  /* and feed to lookup.        */

Do Forever                         /* Do until get EOF.          */
   'READTO record'                 /* Get next input record.     */
   'OUTPUT' record                 /* Write back to stream.      */
   Parse Var record . fn ft fm .   /* Break out file name.       */

   'CALLPIPE',                     /* Invoke pipeline.           */
      '<' fn ft fm '|',            /* Put file into stream.      */
      '*:'                         /* Connect into main pipe.    */
End

Error: Exit RC*(RC<>12)            /* RC = 0 if EOF              */
```

Note also that the addpipe command is issued before the stage issues any input commands to read from its input stream. This is required to establish the connection into the input stream before data begin flowing on it. The same consideration applies to the example below of using hybrid

output connectors; the addpipe command must be issued before the stage issues any output commands, if the added pipeline is to process all the output records.

Addpipe with hybrid output connectors: The following example of using addpipe with hybrid output connectors is from Gregory DuBois, of SLAC. This addpipe command would be used in a stage that produces very large binary records as its output. The user wishes to save the data in a format that will later allow another pipeline to upload it with varload:

```
'ADDPIPE (name GenVar long endchar ?)', /* Post-process output.. */
   '   *.output: |',              /* ..stream from this stage.    */
   '   fblock 64000 |',           /* Form into 64,000-byte recs.  */
   '   specs',                    /* Start making VARLOADable:    */
   '      number 1',              /*    subscript in 1-10;        */
   '      /:/ next',              /*    then delimiter (:);       */
   '      1-* next |',            /*    then the record.          */
   '   strip leading |',          /* Discard blanks in subscript. */
   '   change //:BINARY./ |',     /* Prefix delimited stem name.  */
   'c: count lines |',            /* Record count to secondary.   */
   'f: faninany |',               /* All data recs plus BINARY.0. */
   '   *.output:',                /* Output to next stage.        */
'?',                              /* End of first pipeline.       */
   'c: |' ,                       /* Record count to here.        */
   '   strip |' ,                 /* Deblank count for BINARY.0.  */
   '   specs',                    /* Make it VARLOADable:         */
   '      /:BINARY.0:/ 1',        /*    delimited name;           */
   '      1-* next |',            /*    then value.               */
   'f:'                           /* Merge into output stream.    */
```

A stage invokes this addpipe command to insert the GENVAR pipeline into its output stream with hybrid connectors. Thus, the records that flow out of the invoking stage flow through GENVAR before they get to the next stage. The output produced by the invoking stage is split into 64,000-byte records, and each record is prefixed by the string :BINARY.n:, where n is its record number. The record count is put into another record of the form :BINARY.0:count, and all the records are written to the output stream. Further on in the calling pipeline, these records might be written to disk. Later, they could be read from disk and put through a varload stage, which would store them as a stemmed array, making it convenient to use them for further computations.

Addpipe with prefix input connectors: The following example is from John Hartmann, the author of *CMS Pipelines*. It is a REXX stage that uses a stack of addpipe commands with prefix connections, restoring the stacked connections as its recursion winds down. The inclpack stage processes an input stream containing a PACKAGE file, which lists the files to be distributed as part of a software package. The files in the list may themselves be PACKAGE files, nested to any depth. The lists are processed recursively to produce output records for all the files required for the package. The PACKAGE file records have " &1 &2 " in columns 1-7 and a filename, filetype, and filemode in the next 20 columns.

```
/* INCLPACK REXX:              Include PACKAGE files recursively */
Signal On Novalue

Call Dofile                         /* Begin the recursion.        */
Exit                                /* Exit when done.             */

Dofile: Procedure
Parse Arg stack                     /* Packages being done now.    */

Do Forever
   'READTO in'                      /* Next record from input.     */
   If RC <> 0  Then Leave           /* Leave if no more.           */
   If Left(in,7) ¬== ' &1 &2 '      /* Comment record?             */
      Then Iterate                  /* Yes, ignore.                */
   'OUTPUT' in                      /* File record: to output.     */
   Parse Var in . . fn ft fm .
   If ft <> 'PACKAGE'               /* Iterate if not name of...   */
      Then Iterate                  /* ...another PACKAGE file.    */
   fid = fn'.'Left(fm,1)            /* Iterate if this package...  */
   If Find(stack, fid) > 0          /* ...already being processed. */
      Then Iterate
   'ADDPIPE <' fn ft fm '| *.input:' /* Add a pipe to put this...*/
   If RC <> 0  Then Exit RC         /* ...PACKAGE file into input. */
   Call Dofile stack fid            /* Process recursively.        */
   'SEVER input'                    /* Restore previous stream.    */
End

If RC = 12  Then Return             /* Return on end-of-file.      */
Exit RC                             /* Otherwise exit.             */
```

The Dofile procedure processes a PACKAGE file recursively. The argument to the procedure is a list of the PACKAGE files already being processed, which is used to prevent a loop caused by a package including itself (or including another package that in turn includes it).

The Do Forever loop reads a record and checks whether it names a file. If it does, the record is copied to the output. If the record names a PACKAGE file, a check is made to determine whether that PACKAGE file is among those currently being processed.

If the package is not being processed, then an addpipe command is used to inject the contents of the PACKAGE file into the pipeline. As the added pipeline has a prefix connector that references the stage's input stream, the current input stream is saved on a stack of dormant input streams, and the input stream for the stage is connected to the new pipeline, allowing its < stage to read the PACKAGE file into the pipeline.

The Dofile procedure is called to process the new package. When it is done, the input stream (which is now at end-of-file) is severed. That re-instates the stream on top of the dormant stack to continue reading the most recently interrupted file. This process continues until all the input streams have been processed and an output record has been written for every file required for the

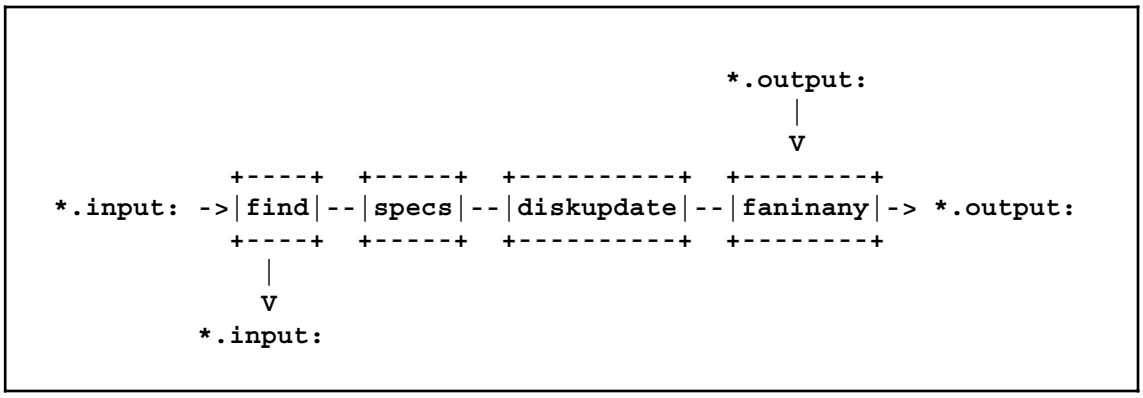
package. (As there is nothing here to prevent a file from being named twice, there would typically be a sort unique stage further on in the pipeline to discard duplicate records.)

Addpipe with hybrid input and output connectors: The following example is from Glenn Knickerbocker, of IBM. It is a stage that is used to apply updates to several different files. The input stream contains records suitable for use by a diskupdate stage, *i.e.*, each record is preceded by a 10-character field giving the number of the record it is to replace. There is an added wrinkle here, however: in front of the record number in each input record is a 20-character field containing the blank-delimited filename, filetype, and filemode of the file to be updated. This stage uses addpipe to create a new pipeline for each file that is to be updated:[2]

```
/* PUTFILES REXX */               /* Update specified files.     */
Signal On Error

Do Forever                        /* Do until end-of-file.       */

   'PEEKTO line'                  /* Examine next record.        */
   Parse Var line fn ft fm .      /* Extract file identifier.    */
   findname = Translate(Left(line,20),'_',' ') /* Fill with _'s. */

   'ADDPIPE (end / name PutFiles)', /* Add a pipe for this file. */
      '|    *.input:',            /* Input from calling pipeline */
                  ,               /*   or other ADDPIPEs.        */
      '| a: find' findname,       /* Record for this file?       */
      '|    specs 21-* 1',        /* Yes, remove name field.     */
      '|    diskupdate' fn ft fm, /* Update specified file.      */
      '| b: faninany',            /* Merge the two streams.      */
      '|    *.output:',           /* Output to calling pipeline  */
                 ,                /*   or other ADDPIPEs.        */
   '/',
      '  a:',                    /* Updates for other files.    */
      '|    *.input:',           /* Input to other ADDPIPEs in  */
                ,                 /*   this stage (or PEEKTO).   */
   '/',
      '     *.output:',          /* Output from other ADDPIPEs  */
                ,                 /*   in this stage.            */
      '| b:'                     /* Go send to calling pipeline.*/

End

Error: Exit RC*(RC<>12)           /* RC = 0 if end-of-file.      */
```

[2] A pipeline specification may have a stage separator between the global options and the first stage. This may be necessary to distinguish between global options and options that apply only to the first stage.

The pipelines added by the addpipe command in this stage have the following topology:



A single pipeline of this topology can be very powerful. Because this pipeline begins and ends with input connectors, it has a hybrid connection into the input stream for the stage that invoked addpipe. This causes input records to flow through this pipeline before they can be seen by the stage. Because this pipeline also begins and ends with output connectors, it also has a hybrid connection into the output stream for the stage. This causes it to receive any output records from the stage before they flow into the next stage. Having hybrid connectors at both ends gives this pipeline the additional characteristic of being able to shunt records from the input for the stage to the output for the stage without their being seen by the stage.

In this example, this concept is carried even further, because multiple pipelines of this topology are created, one for each file to be updated. The result is “nested” hybrid connections on both the input stream and the output stream.

When the stage starts, the peekto command examines the first record, and the addpipe creates a pipeline for processing the file named in the first record. After that, the first added pipeline will divert any records for its file before they can be seen by the peekto. The next record that gets through to the peekto will be for another file, so addpipe will be used again to create a second pipeline for processing records for that file. This process will continue until a pipeline has been added for each of the files named in the input stream, after which the peekto will see no more input records. Each pipeline will read its input stream and divert records for its file to its output stream, while sending records for other files along its hybrid input connection, so that they can be passed to the other added pipelines until eventually reaching the right one. Output from the multiple added pipelines is similarly cascaded.

Addpipe with prefix input and output connectors: In the following example (from John Hartmann), a stage that already has primary and secondary streams defined issues an addstream pipeline command to establish tertiary input and output streams for itself:

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

It then issues an addpipe command to create a pipeline that will connect to those tertiary streams using prefix connectors, with the stage’s output connected to the input of the added pipeline and *vice versa:*

```
'ADDPIPE (endchar ? name NodeSyn)',
      '*.output.2: |',            /* Input from tertiary output. */
      'xlate |',                  /* Upper-case domain name.     */
      'pad 49 |',                 /* Pad to full key.            */
   'l: lookup 1.49 master |',     /* Find matching RSCS nodename.*/
      'spec /+/ 1 50-* next |',   /* Remember we got a match.    */
   'f: faninany |',               /* Join output streams.        */
      '*.input.2:',               /* Output to tertiary input.   */
'?',
      '< bitftp nodesyn |',       /* Read in nodename synonyms.  */
   'l: |',                        /* Non-matches come to here.   */
      'change //-/ |',            /* Remember didn't get match.  */
   'f:'                           /* Route to FANINANY.          */
```

This creates a permanent subroutine into which the stage can now and then throw a record and get back a response. The advantage of this arrangement over repeatedly creating a similar pipeline with callpipe is that the added pipeline is initiated only once; this might be a significant savings if, for example, the master file being read by the lookup stage is large.

Once the added pipeline has been affixed to the invoking stage, the stage can send records into the pipeline by writing on its tertiary output stream and can receive records from the pipeline by reading from its tertiary input stream. A convenient way to do this might be to use a subroutine pipeline such as the following:

```
'CALLPIPE (endchar ? name GetSyn)',
   'var usernode |',              /* Domain-style nodename.      */
   '*.output.2:',                 /* Into the added pipeline.    */
'?',
   '*.input.2: |',                /* From the added pipeline.    */
   'take 1 |',                    /* Stop when get one record.   */
   'find +|',                     /* Select for matches.         */
   'change /+// |',               /* Remove marker.              */
   'var nodename'                 /* Set user's nodename.        */
```

This inexpensive callpipe, which might be invoked repeatedly, writes one record to the added pipeline, receives one record back, and then terminates, returning control to the stage that issued both the addpipe and the callpipe.

For this scheme to work, the added pipeline must produce a known number of records from each input record and must not delay the record. If the stage uses output to write to the added pipeline and readto to read from it, then the added pipeline must have a one-record “elastic” (*e.g.,* a copy stage) to consume the record and thus allow the output to complete, so that the readto can be issued. If the stage uses a subroutine pipeline to connect to the added pipeline, then the subroutine pipeline must send an end-of-file out through both of its (redefine) connectors to the

invoking stage, so that the stage's connections to the added pipeline are restored when the callpipe command completes.

How Addpipe Differs from Callpipe

addpipe is similar to the callpipe command in that they both add pipelines to the running set. However, addpipe and callpipe differ from one another in several important ways:

- When callpipe is invoked, the stage that invokes it is suspended until the new pipeline has run to completion. When the callpipe command completes, its return code is the return code resulting from running the added pipeline.

  When addpipe is invoked, the stage that invokes it regains control as soon as the new pipeline has been created. The added pipeline runs in parallel with the stage that created it (and, in fact, the added pipeline can continue to run after the invoking stage has ended). Neither is able to examine the other's return code. The return code from the addpipe command itself indicates only whether its pipeline specification was syntactically correct.

- callpipe can use only redefine connectors. (This follows from the fact that the stage is blocked, so no data could flow on a prefix-style connection.) When a subroutine pipeline created by callpipe decides that it will process no more records and sets end-of-file to flow out through a connector to the invoking stage, the invoking stage's original connection is automatically re-instated.

  addpipe can use redefine connectors, but callpipe is more suitable for most cases where redefine connectors are required. When a redefine connector is used with addpipe, the original connection cannot be restored. When a prefix connector is used with addpipe, the restoration of the stage's original connection is not automatic, but requires an explicit sever of the added stream.

- A pipeline added by callpipe can run only at the same commit level as the invoking stage (or a lower one). If a subroutine pipeline created by callpipe attempts to commit to a higher level than the invoking pipeline, it is suspended until the invoking pipeline reaches that commit level.

  The commit level of a pipeline added by addpipe is independent of the commit level of the invoking pipeline. Data can flow on a connection established by addpipe even if the pipelines on the two sides of the connection are not at the same commit level.