PLUMBING THE INTERNET

CMS/TSO Pipelines Support for TCP/IP

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

VM & VSE Technical Conference
Session 23L
May 1998

CMS/TSO PIPELINES SUPPORT FOR TCP/IP

CMS 12 introduced *CMS/TSO Pipelines* support for TCP/IP. The *Pipes* TCP support is very powerful, very robust, and extraordinarily easy to use. It is built around three new stages, tcpclient, tcpdata, and tcplisten. The *Pipes* TCP support was enhanced extensively for CMS 14. These enhancements are available through the *CMS Pipelines* "Runtime Library Distribution"[1] and, thus, can be used in software products that redistribute the Runtime Library (including shareware).

Support for TCP-based clients

Pipelines makes writing TCP-based clients extremely easy. If you know basic "plumbing", you can send or receive data across an IP network as easily as you can read or write a disk file.

tcpclient: The tcpclient stage connects to a TCP/IP server, transmits its input records to that server, and writes the data it receives from the server onto its output stream. If tcpclient has a secondary output stream, any "ERRNO" received from TCP/IP is written to that stream; otherwise, the ERRNO is displayed in a message. tcpclient's syntax is:

---

[1] The Runtime Library Distribution makes a recent version of *Pipelines* available royalty-free, so that software developers can package it with their products. The distribution is available *via* the World Wide Web (http://pucc.princeton.edu/~pipeline/) or by anonymous FTP from pucc.princeton.edu (cd anonymous.376). The examples in this paper use the level of *CMS Pipelines* from CMS 14 or the Runtime Library Distribution.

```
     TCPCLIENT    hostname|IPaddress  port  [USERid vmid]
                  [LOCALport port]  [LINGER seconds]
                  [LOCALIPaddress [ANY|HOSTID|<IPaddress>]]
                  [ [SF|SF4] [ONERESPonse [TIMEOUT seconds]]]
                  [DEBLOCK [ [SF|SF4|<pcstyle>] [GROUP stage]]]
                  [ONERESPonse] [STATistics]
                  [TIMEOUT seconds] [GREETING]
                  [GETSOCKName] [KEEPALIVe] [OOBINLINE] [REUSEADDR]

     DEBLOCK      Specify blocking to be performed on output data.
     GETSOCKName  Write socket address structure as first output.
     GREETING     Expect a connection-opening response from server.
     GROUP        Group output records with specified stage.
     KEEPALIVe    Enable KEEPALIVE socket option.
     LINGER       Wait after end-of-file on input.
     LOCALIPaddr  Specify the local IP address to be used.
     LOCALport    Specify the local port to be bound to the client.
     ONERESPonse  Expect one response to each transmission.
     OOBINLINE    Enable OOBINLINE socket option.
     REUSEADDR    Enable REUSEADDR socket option.
     SF           Block and deblock with halfword length fields.
     SF4          Block and deblock with fullword length fields.
     STATistics   Report statistics.
     TIMEOUT      Time out after specified period.
     USERid       TCP/IP server virtual machine name.
```

A simple TCP client: tcpclient is so simple to use that even though I know nothing at all about socket programming, it took me less than half an hour to write a pipeline to query and update a CSO database on one of our UNIX systems the first time I tried using tcpclient. This is a fragment from a pipeline that does a CSO query:

```
'PIPE (endchar ? listerr name GetId)',   /* Query CSO database:  */
     'var query |',                 /* Load CSO QUERY command.    */
     'append literal quit|',        /* Follow by a CSO QUIT cmd.  */
     'insert x0D25 after |',        /* Append CRLF for UNIX.      */
     'xlate 1-* from 1047 to 819 |', /* Translate query to ASCII.*/
  't: tcpclient' host port 'linger 30 |',   /* Send cmds to CSO. */
     'xlate 1-* from 819 to 1047 |',/* Translate reply to EBCDIC.*/
     'deblock C terminate |',       /* Deblock CSO response.      */
      . . .                         /* Further processing.       */
'?',
  't: |',                           /* 2ndry output of TCPCLIENT. */
     'nfind 0_OK|',                 /* Normal socket-close ERRNO. */
     'append literal |',            /* (In case no TCPIP ERRNO.)  */
     'var errno'                    /* Save any non-0 ERRNO.      */
```

Two CSO commands, a QUERY and a QUIT, are loaded into the pipeline, delimited with "carriage return" and "line feed" characters, translated from EBCDIC to ASCII, and read into the tcpclient stage, which transmits them to the specified host and port and "lingers" for 30 seconds or until the connection is closed, whichever happens first. When tcpclient receives a response from the server, it writes it into the pipeline, where it is translated to EBCDIC and deblocked in preparation for further processing. If an error is reflected by TCP/IP, a record containing the ERRNO is written to tcpclient's secondary output stream, and tcpclient terminates. When the socket closes normally, an ERRNO record containing the string "0 OKSocketClosed" is written to the secondary output. Since this particular application expects the socket to be closed, that ERRNO record is discarded, leaving the REXX variable "errno" set to null in the normal case.

A more sophisticated TCP client: CALLSMTP REXX (a fragment of which is shown here) is the lowest-level routine in a client SMTP implementation done with *CMS Pipelines*. (The entire implementation can be found in Appendix A.) CALLSMTP uses some of the newer options on tcpclient to make the rest of the pipeline simpler:

```
'CALLPIPE (endchar ? listerr name CallSMTP)',
    '*.input.0: |',               /* Input for SMTP from caller. */
    'literal HELO' hostnode || crlf'|'  /* First cmd of session. */
    'xlate 1-* from 1047 to 819 |', /* Translate input to ASCII. */
  't:tcpclient' host port,        /* Send cmds to SMTP server.   */
      'timeout' timeout,          /* Exit if no reply.           */
      'oneresponse',              /* Synchronize cmds/replies.   */
      'deblock string x0D0A',     /* A reply ends with CRLF.     */
      'greeting',                 /* Don't count connected msg.  */
      'group /nlocate 4.1 x2D/ |',/* Don't count if continued.   */
    'xlate 1-* from 819 to 1047 |', /* Xlate replies to EBCDIC.  */
    '*.output.0:',                /* Output from SMTP to caller. */
'?',
  't: |',                         /* TCPCLIENT's secondary output*/
    '*.output.1:'                 /* Return ERRNO to caller.     */
```

Byte stream parsing is usually one of the nastier bits of writing a TCP application. This is just another symptom of the UNIX fallacy of believing data to be unstructured. What is transmitted between the client and the server is a byte stream, but the server needs to get commands from the client and the client needs to get responses from the server. So, until they have determined the record boundaries in their incoming byte streams, they have no information on which to act.

As the design of the *Pipelines* support for TCP evolved, the author progressively lifted the burden of byte stream parsing from the programmer, thus making the writing of TCP applications less tedious and complex. With the current version, the programmer can specify that the TCP stages do all of the byte stream parsing and the synchronization of responses with commands and that they do these things in such a way as not to "delay the record", thus making the flow of data through a multistream pipeline completely predictable.

In the CALLSMTP example, the tcpclient timeout option says to time out if no response is received from the server within the specified interval. The oneresponse option says to expect one response for each command. When that option is specified, tcpclient consumes an input

record containing a command only after it has written the corresponding output record containing the response to that command; that is, it does not delay the record. The deblock option tells tcpclient how to recognize the boundaries of responses it receives from the server, which may, of course, be spread across multiple network packets. The greeting option says to expect a message from the server when the connection is opened, to time out if this message is not received, and not to count this message as the response to the first transmission. The group option has as its argument a delimited string naming a stage to be executed after the records from the server have been deblocked, in order to determine what constitutes one whole response. The point here is that many TCP servers give one response to each command but may continue that response across multiple CRLF-delimited lines. In the case of SMTP, all lines of a response except the last one have a hyphen in column 4. Thus, the group option in this example says to discard records with an ASCII hyphen in column 4, rather than counting them as a response. (Alternatively, one could specify a stage in the group option that would join continued records into a single output record that would be counted as one response.)

The result of using these options is that tcpclient insures that one complete response is received for each input command. It breaks its output records at response boundaries and times out if either no response or an incomplete response is received. We will discuss the signifiance of all this in more detail later.

Support for TCP-based servers

The new *Pipes* support makes writing TCP-based servers almost as easy as writing clients. No greater knowledge of network protocols is required, but your plumbing may need to be slightly more sophisticated for writing servers than for writing clients.

tcplisten: The tcplisten stage is used in TCP-based servers to wait for connection requests from clients. Its syntax is:

```
TCPLISTEN    port  [BACKLOG number] [USERid vmid]
             [LOCALIPaddress [ANY|HOSTID|<IPaddress>]]
             [GETSOCKName] [STATistics] [REUSEADDR]

BACKLOG      Maximum number of pending requests.
GETSOCKName  Write socket address structure as first output.
LOCALIPaddr  Specify the local IP address to be used.
REUSEADDR    Enable REUSEADDR socket option.
STATistics   Report statistics.
USERid       TCP/IP server virtual machine name.
```

tcplisten listens for and accepts connection requests on a TCP port using the socket interface to TCP/IP. Once a client connects, the socket representing the connection should be transferred to a tcpdata stage for further processing. When tcplisten is ready to give the socket for a new request to a tcpdata stage, it writes a record describing that socket to its primary output stream. As soon as that record has been consumed, tcpclient resumes listening for further connection requests.

tcpdata: The tcpdata stage is used in a server to receive data from a client and transmit data to the client. Its syntax is:

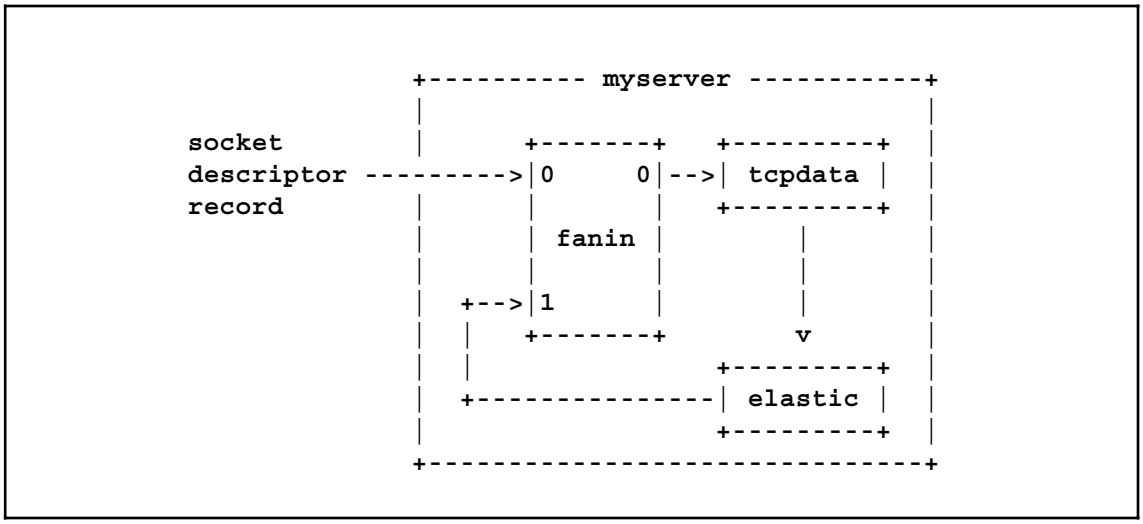
```
TCPDATA       [LINGER seconds] [SF|SF4]
              [DEBLOCK [ [SF|SF4|<pcstyle>] [GROUP stage]]]
              [GETSOCKName] [KEEPALIVe] [OOBINLINE]
              [ONERESPonse] [STATistics]

DEBLOCK       Specify blocking to be performed on output data.
GETSOCKName   Write socket address structure as first output.
GROUP         Group output records with specified stage.
KEEPALIVe     Enable KEEPALIVE socket option.
LINGER        Wait after end-of-file on input.
ONERESPonse   Expect one response to each transmission.
OOBINLINE     Enable OOBINLINE socket option.
SF            Block and deblock with halfword length fields.
SF4           Block and deblock with fullword length fields.
STATistics    Report statistics.
```

tcpdata determines the socket to use by examining its first input record, which should be one written by a tcplisten stage. Once tcpdata gets the initial record describing its socket, it takes that socket and begins passing data to and from the client across the network.

Data received from the client are written to tcpdata's primary output stream, and data read from tcpdata's primary input stream are sent to the client. Thus, the code implementing the server must be able to read from tcpdata's output stream and write to its input stream. You will find the elastic stage that was added to *Pipes* in CMS 10 to be very useful for implementing such a looping pipeline. (For a discussion of using elastic, see the section entitled "On Record Delay and Preventing Pipeline Stalls" in my *Pipe Dreams* paper.)

A simple TCP server: To illustrate this sort of feedback pipeline, let's look at the example from the author's help file for tcpdata; it shows what is probably the simplest possible server, one that just echoes the input from the client back to the client:

```
'CALLPIPE (listerr name myserver)',  /* Simplistic ECHO server:  */
     '*: |',                 /* Get socket record from TCPLISTEN.  */
     'take 1 |',             /* Take just one socket descriptor.   */
  'i: fanin |',              /* Then client output from secondary. */
     'tcpdata |',            /* Send to client & get from client.  */
     'elastic |',            /* Buffer to avoid stall.             */
  'i:'                       /* Feed client output to FANIN.       */
```

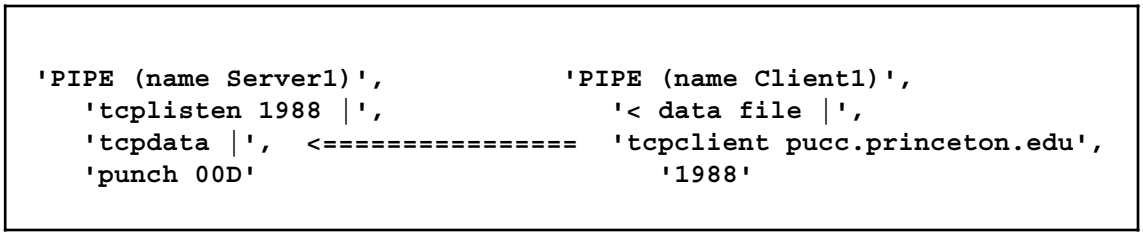


This subroutine pipeline (called “myserver”) would be launched by a server when a client connection request is received. The subroutine receives one record through its input connector, the record written by tcplisten to give it the socket for the request. fanin passes that record to tcpdata, which takes the socket and begins receiving data from the client. Each time it receives a packet from the client, tcpdata writes a record on its output stream, where elastic reads it and then writes it to fanin’s secondary input. Since fanin has already received end-of-file on its primary input, it reads the client records from its secondary input and writes them to its primary output, where tcpdata reads them and transmits them across the network to the client, thus echoing the data received from the client back to the client.

I am sure you can see that it would be straightforward to build a more useful server by inserting additional processing into the loop in myserver, between the elastic and the fanin. Unlike this echo server, your server will likely need to reblock its input into proper records before it can make sense of them. Also unlike the echo server, your server will likely need to detect the end of the transaction and terminate gracefully, after formatting a suitable response to be sent through fanin and tcpdata to the client.

Building your own client and server

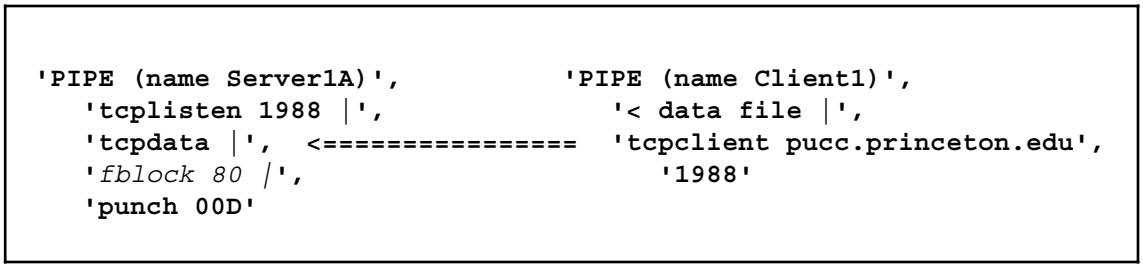
Let’s start from scratch and build up a simple client and a simple server, both running under VM. For the sake of simplicity, we will define our own “Internet protocol” for this example, but once you understand the basic process of building a client or a server, you will find that building one that adheres to a standard Internet protocol is simply a matter of reading the relevant RFCs and doing some plumbing. We will start with a server that is even simpler than the echo server and with an equally simple client, each running in its own virtual machine:



The tcpclient stage in the client requests a connection on port 1988 of the server. tcplisten in the server is waiting for a client connection request on that port. When it gets one, it writes the socket descriptor record to its output. tcpdata reads that record and takes the socket that it describes, thus opening communication with the client. The client pipeline reads DATA FILE and transmits it through the tcpclient stage to the server, which punches it. (To keep the example simple, we are assuming that DATA FILE consists of 80-byte records.) If you run this example yourself, you will see that it really does transmit data across the network. It's as easy as that!

Although Server1 *does* punch a file, I must confess that the output is not what one might hope. For those of us who are accustomed to dealing with S/390 architecture, it is often difficult to remember that TCP/IP does not understand record boundaries. Even though it is connecting VM to VM here, TCP/IP transmits a bytestream that is split into packets that have no relationship to the records that were read from the minidisk file. So, what tcpdata receives in Server1 is not necessarily the 80-byte records that were fed into the tcpclient stage in Client1. When I ran this example several times, the packets that tcpdata received were of varying sizes between 80 and 320 bytes, in a pattern that was not reproducible.

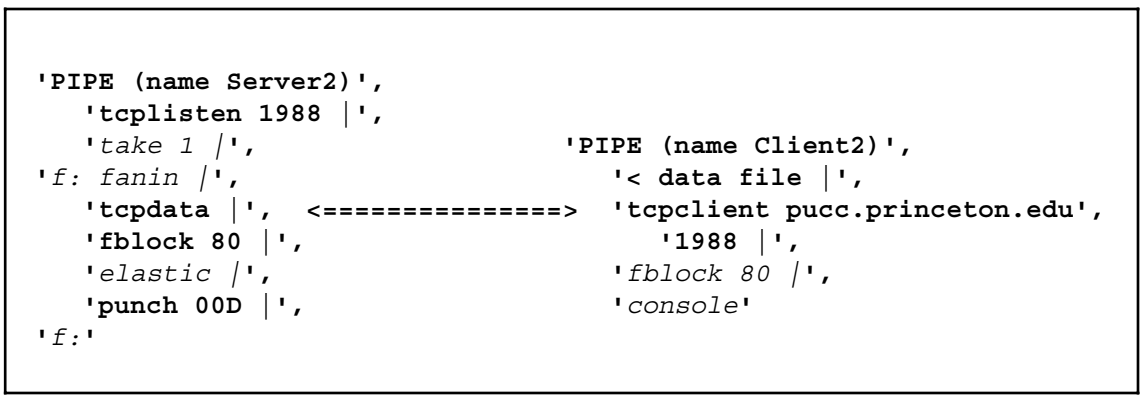
Unless you give it a deblocking option, tcpdata simply writes each packet it receives from the network to its output stream as a record. And when it writes one of those 320-byte records to the input of punch, punch will happily punch the first 80 bytes and discard the rest. So, our Server1 produces an output file that is missing some records. Fortunately, we can fix that problem very easily by using fblock 80 to reblock the output of tcpdata into the original 80-byte records:



```
'PIPE (name Server1A)',          'PIPE (name Client1)',
   'tcplisten 1988 |',              '< data file |',
   'tcpdata |',  <================  'tcpclient pucc.princeton.edu',
   'fblock 80 |',                     '1988'
   'punch 00D'
```

Client1 and Server1A work correctly. The client sends a file to the server and terminates. The server waits for a client connection, receives the file from the client, punches it, and terminates.

Although that protocol may be robust enough for some purposes, it is more likely that you will want the client to know that the server has successfully processed the file. If you do, we need to enhance our server to give the client some feedback:



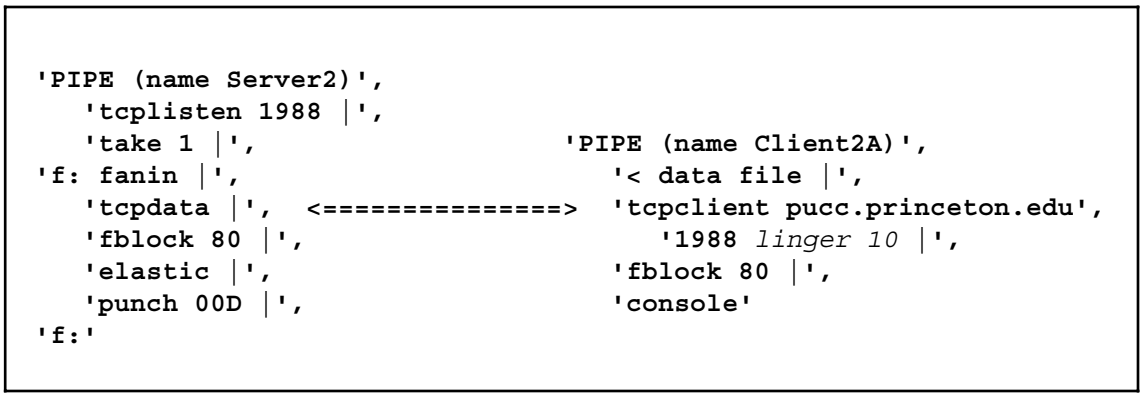
```
'PIPE (name Server2)',
   'tcplisten 1988 |',
   'take 1 |',                   'PIPE (name Client2)',
'f: fanin |',                       '< data file |',
   'tcpdata |',  <===============>  'tcpclient pucc.princeton.edu',
   'fblock 80 |',                     '1988 |',
   'elastic |',                    'fblock 80 |',
   'punch 00D |',                  'console'
'f:'
```

We have made several changes here. The server now looks more like myserver, the echo server we were discussing earlier. The output of punch is now fed through a fanin to tcpdata to be sent back to the client. Because we are now building a feedback loop, we must insert an elastic stage to consume the output of tcpdata quickly, to unblock it as soon as it writes a record, so that it can then read any records waiting to be sent back to the client. And we must insert a take 1 to sever fanin's primary input stream, so that it can read those feedback records from its secondary input. (You will note that this is exactly the scheme we used in the echo server earlier.)

We have also changed the client to do something with the records it gets back. It now displays them on its console. Note that the client must also reblock the packets it receives from the server into 80-byte records. The record boundaries will have been lost again when the records were transmitted back across the network.

That all sounds good, but in real life the client doesn't actually receive any of the feedback records. The problem is that tcpclient terminates as soon as it has sent its last input record across the network. So, the client has gone away by the time the server has punched the records and is ready to send them back across the network. This causes the server to terminate with ERRNO 32 EPIPE when tcpdata tries to write to the socket that is no longer connected.

We can enable the client to see the feedback records by making a small change to the client:



The linger 10 option on tcpclient causes it to wait up to 10 seconds after it sends its last packet before it terminates. (It will terminate sooner than that if it sees the socket close.)

Server2 and Client2A now behave correctly, in that the file from the client is punched by the server and then goes back across the network to the client, where it is displayed on the client's console. However, termination of the two pipelines is still a bit untidy. The client displays the file on the console and then sits there waiting until the end of the 10 seconds, after which it terminates, causing the socket to close. tcpdata, which is still trying to read from the socket, then sees the socket go away. That results in the server terminating with ERRNO 54 ECONNRESET.

Certainly, it would be better if both the client and the server would terminate with no errors as soon as the transaction completes. With a little more effort, we can get them to do that. The first thing we must do is define an end-of-transaction indicator. I will be thoroughly old-fashioned and declare that the end of a transaction in this protocol is signalled by an 80-byte record that begins "//EOJ". Let's see how to make that work:

```
'PIPE (name Server3)',           'PIPE (name Client3)',
   'tcplisten 1988 |',              '< data file |',
   'take 1 |',                      'append literal',
'f: fanin |',                          Left('//EOJ',80)'|',
   'tcpdata |',  <================>  'tcpclient pucc.princeton.edu',
   'fblock 80 |',                      '1988 linger 10 |',
   'elastic |',                     'fblock 80 |',
   'tolabel //EOJ|',                'console'
   'punch 00D |',
'f:'
```

The client is enhanced to insert an 80-byte //EOJ record at the end of the file it transmits to the server. A tolabel //EOJ stage is added to the server to make the server pipeline terminate when the transaction is complete. When it sees the //EOJ record, tolabel terminates, and end-of-file propagates backward and forward from tolabel to the output and input streams of tcpdata. tcpdata sees the end-of-file and terminates, causing the socket to close. Once tcpclient had sent the //EOJ record, it began "lingering", so it is now happy to see the socket close. tcpclient terminates, shutting down the client pipeline with a return code of 0.

We have now solved the problem of terminating promptly and in an orderly fashion, but we probably don't really want to echo all of the input records back to the client, so we make another small enhancement to the server:

```
'PIPE (end ? name Server4)',     'PIPE (name Client3)',
      'tcplisten 1988 |',           '< data file |',
      'take 1 |',                   'append literal',
   'f: fanin |',                       Left('//EOJ',80)'|',
      'tcpdata |',  <=============>  'tcpclient pucc.princeton.edu',
      'fblock 80 |',                   '1988 linger 10 |',
      'elastic |',                  'fblock 80 |',
   't: tolabel //EOJ|',             'console'
      'punch 00D',
'?',
   't: | take 1 | f:'
```

The client is unchanged, but the server now sends only the //EOJ record back to the client at the end of the transaction.

This is now reasonably satisfactory, except for the fact that the server terminates after it has processed a single client request. We are more likely to want it to sit around all day responding to client requests and not to terminate until we stop it with a PIPMOD STOP command. To accomplish that, we build a main pipeline that listens for client requests and passes them to a sipping subroutine pipeline. The subroutine pipeline looks much like our Server4, but with an input connector:

```
          'PIPE (name Server5) tcplisten 1988 | server5'


/* SERVER5 REXX */

Do Forever                        /* Continue until PIPMOD STOP.*/

  'PEEKTO'                        /* Await connection request.  */
  If RC <> 0  Then Exit RC*(RC<>12)

  'CALLPIPE (endchar ?)',         /* Handle one client request: */
        '*: |',                   /* Socket rec from TCPLISTEN. */
        'take 1 |',               /* Sip one socket descriptor. */
     'f: fanin |',                /* Pass records to TCPDATA.   */
        'tcpdata |',              /* Talk to the client.        */
        'fblock 80 |',            /* Reblock the records.       */
        'elastic |',              /* Free TCPDATA to read input.*/
     't: tolabel //EOJ|',         /* Stop when get to //EOJ.    */
        'punch 00D',              /* Punch the data records.    */
  '?',
     't: | take 1 | f:'           /* Return //EOJ to client.    */

  Address Command 'CP CLOSE PUN' /* Close the punch.           */

End
```

Each time tcplisten in the main pipeline receives a client connection request, it writes the socket descriptor to SERVER5 REXX, which sips one descriptor at a time, continuing forever to launch a subroutine pipeline for each client connection request. The only thing left to wish for is that the server handle multiple clients simultaneously. We will discuss ways of achieving that later.

A real client and server (IDENT)

Before getting to more complicated matters, however, let's look at another simple client and server, but real ones this time that implement a standard Internet protocol, IDENT:

```
/*--------------------------------------------------------------------*/
/*                 IDENTify the client per RFC 1413                   */
/*                                                                    */
/*  Sample IDENT query: "49201 , 80"                                  */
/*            <port-on-server> , <port-on-client>                     */
/*                                                                    */
/*  Sample IDENT reply: "49201 , 80 : USERID : VM/CMS : maint"        */
/*   <port-on-server> , <port-on-client> : <resp-type> : <add-info>   */
/*--------------------------------------------------------------------*/
```

Writing an Internet server, you are likely to need the IDENT protocol defined by RFC 1413. When you receive a client connection request, you can find out who the user is by sending a

query to port 113 on the remote system. Normally, there will be an IDENT daemon listening on that port, and it will reply with either an error message or a message containing the userid of the client. Here's how to send an IDENT query:

```
query = remote_port ',' my_port  /* Build IDENT query message. */

'CALLPIPE (endchar ? listerr name IdentClient)',
     'var query |',              /* Load IDENT query message.  */
     'insert x0D25 after |',     /* Garnish with CR and LF.    */
     'xlate from 1047 to 819 |', /* Make the message ASCII.    */
  't: tcpclient' remote_ip '113',/* Send message to IDENT port.*/
       'deblock string x0D0A',   /* Deblock the response.      */
       'oneresponse',            /* Enforce 1 output per input.*/
       'timeout 10 |',           /* Time out after 10 seconds. */
     'xlate from 819 to 1047 |', /* Reply from ASCII to EBCDIC.*/
     'append literal |',         /* In case get no response.   */
     'var remote_ident',         /* Store remote_ident.        */
'?',
  't: |',                        /* Any TCP/IP ERRNO to here.  */
     'append literal |',         /* In case no ERRNO received. */
     'var errno'                 /* Store the ERRNO.           */
```

This IDENT client is not basically different from the very simple client we have just been building. It loads a record into the pipeline and uses a tcpclient stage to transmit that to the server and to put the response from the server into the pipeline. The primary difference here is that we are using a standard protocol, so the data are transmitted in ASCII, with individual lines delimited by carriage return and line feed. Therefore, we use an insert and an xlate to massage the query before it is sent to the client, and we use an xlate and the deblock option on tcpclient to produce proper EBCDIC records from the packets sent in response to our query.

One enhancement here over the client we built earlier is that we use the secondary output of tcpclient to capture any TCP/IP error message and put it into a REXX variable for easier processing. The main enhancement, however, is that we have taken advantage of some of the options on tcpclient:

1. We have told tcpclient how to reblock the packets it receives into records. So, if the IDENT server sends its response spread across multiple packets, tcpclient will wait until it has a complete CRLF-delimited output line before writing anything to its output.
2. We have told tcpclient to time out if it does not receive a response within 10 seconds. If that happens, it will produce an appropriate error message on its secondary output, so that our program can decide what action to take.
3. We have told tcpclient to expect one complete output line in response to each input record. As we are giving it only one input record and we have told it how to recognize when it has received a complete response, it has the information to decide to terminate when the transaction is complete or to time out if the transaction has not completed in 10 seconds.

Thus, we don't need to worry about any of the byte-stream parsing or about making the connection close gracefully. tcpclient does all of that automatically.

Now, let's look at the other side of the transaction, at the IDENT server. You will note that it is very similar to the server we built up earlier:

```
'CALLPIPE (endchar ? listerr name IdentDaemon)',  /* Do IDENT: */
     '*: |',                      /* Process client request:    */
     'take 1 |',                  /* Get one socket descriptor. */
  'f: fanin |',                   /* Make a feedback loop.      */
  't: tcpdata',                   /* Talk to the client.        */
       'deblock string x0D0A |', /* Get a full request line.   */
     'xlate from 819 to 1047 |', /* Translate it to EBCDIC.    */
     'elastic |',                 /* Unblock TCPDATA.           */
     'doident' rmport rmaddr hostname '|',  /* Build response. */
     'insert x0D25 after |',      /* Delimit response lines.    */
     'xlate 1-* from 1047 to 819 |',  /* Make response ASCII.  */
  'f:',                           /* Feed response to client.   */
'?',
  't: |',                         /* Any TCP/IP ERRNO to here.  */
     'nfind 0_OK|',               /* Connection close is normal.*/
     'append literal |',          /* In case no other error.    */
     'var errno'                  /* Store the ERRNO.           */
```

The enhancements here are very similar to the ones in the client. The packets from the client must be reblocked and translated to EBCDIC before we can process the request; our response must be delimited with carriage return and line feed and translated to ASCII before it is sent to the client. And any TCP/IP error message is written to the secondary output of tcpdata and stored in a REXX variable for later analysis. (We expect the client to close the connection, so we discard any "0 OKSocketClosed" message rather than storing it in the ERRNO variable.)

TCPSNIFF: A combination server and client

Now let me show you a more sophisticated example of using the TCP stages. (This example is the work of John Hartmann, the author of *CMS Pipelines*.) TCPSNIFF is a "sniffer" for TCP. TCPSNIFF can be inserted between a client and a server to examine the traffic passing between them. Assume, for example, that you wish to look at the traffic between a TELNET server and one of its clients. You would invoke TCPSNIFF EXEC with arguments telling it the local port on which it should listen and the TELNET server's host name and port. You would then invoke TELNET on the client, giving it the host name of the machine on which TCPSNIFF is running and the port number TCPSNIFF has been set up to listen on. TCPSNIFF would record all traffic between the client and server, passing packets between them transparently.

TCPSNIFF EXEC first sets up a STOP immediate command, as the pipeline would otherwise be never-ending. Then the tcplisten stage begins listening for client connection requests. When a client connects, tcplisten writes a record describing the socket to its output stream. As soon as that record is consumed, tcplisten begins listening for another connection request. fanout writes one copy of the socket descriptor record to its primary output, where it is reformatted for display on the console. fanout writes another copy of the socket descriptor record to its secondary output, which is connected to the input of the tcpsniff stage, a REXX pipeline filter (see page 14).

Each time a client connects, a record is written to that REXX filter. This process continues until a STOP command is typed on the virtual machine console. The immcmd stage captures the STOP command and writes it to its output, which is connected to the primary input of a gate stage. When gate sees a record on its primary input, it terminates, thus severing all of its input and output streams. Since the output of tcplisten is connected to the secondary input of gate, the termination of gate causes tcplisten to receive end-of-file on its output, so it terminates, and the entire pipeline closes down as soon as the REXX filter finishes handling any active request.

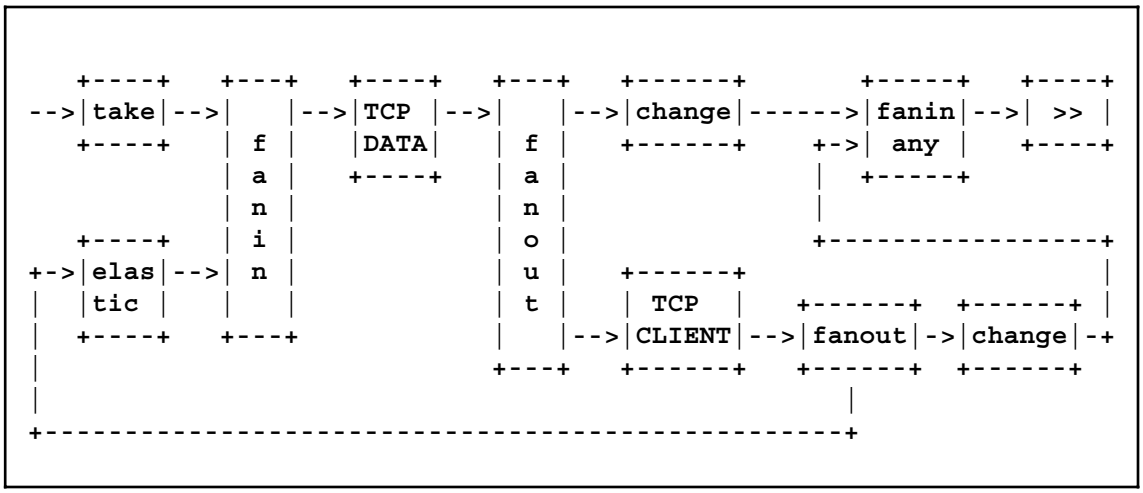
```
/* TCPSNIFF EXEC: TCP port sniffer.                                  */
/*                            John Hartmann  7 Feb 1996 12:38:57 */
/*********************************************************************/
/*  Send TCP socket data to a remote host and pass the result    */
/*  back.  To sniff a telnet session, this program would be the  */
/*  server as seen by the telnet client, and it would be the     */
/*  client as seen by the real telnet server.  The real server   */
/*  is specified as the second and third operands.               */
/*                                                               */
/*  Specify "( ascii" if the data stream is in ASCII and you     */
/*  want the log file to be in EBCDIC (from 819 to 1047).        */
/*********************************************************************/

Signal On Novalue
Address Command

Parse Arg port remote_sys remote_port fn ft fm '(' options

'PIPE (endchar ? listerr name TCPSniff)',
  '?   immcmd stop',              /* Set up immediate command.   */
  '| g:gate',                     /* Terminate when get STOP.    */
  '?   tcplisten' port,           /* Write rec when get socket.  */
  '| g:',                         /* (For stopping TCPLISTEN.)   */
  '| o:fanout',                   /* Divert copy of socket rec.  */
  '|   spec 65.16 1',             /* Just sockaddr_in structure. */
  '|   socka2ip',                 /* Make human-readable.        */
  '|   spec',                     /* Garnish.                    */
        '/Request from port/ 1  w2 nw  /on/ nw  w3 nw',
  '|   console',                  /* Display address and port.   */
  '? o:',
  '|   tcpsniff' Arg(1)          /* Feed socket rec to sniffer. */

Exit RC
```

The tcpsniff filter first sets the value of the REXX variable “xlate”, which is to be inserted into a subroutine pipeline. “xlate” is the null string if no translation of the packets is desired; otherwise, it is a pipeline stage to translate from ASCII to EBCDIC (from codepage 819 to codepage 1047). The filter then determines the name of its log file based on defaults and the user’s arguments.

```
/* TCPSNIFF REXX: Sniffer to process one connection request      */
/*                            John Hartmann  7 Feb 1996 12:49:10 */
Signal On Novalue
Signal On Error

Parse Arg . remote_sys remote_port fn ft fm '(' options
Upper options

xlate=''                          /* No translation of log file.   */
If Wordpos('ASCII', options) > 0   /* Unless option specified.   */
   Then xlate = '| xlate from 819 to 1047'

logfile = Word(fn 'TCP', 1) Word(ft 'TRACE', 1) Word(fm 'A', 1)

Do Forever                        /* Until STOP immediate command. */

  'PEEKTO'                        /* Wait for client connection.   */

  'CALLPIPE (endchar ? listerr name Sniffer)',
    '?   *.input:',               /* Socket record from TCPLISTEN. */
    '|   take 1',                 /* Sip one request at a time.    */
    '| i:fanin',                  /* Socket rec, then servr output.*/
    '|   tcpdata',                /* Play server to the user.      */
    '| f:fanout',                 /* Divert copy of user input.    */
        xlate,                    /* Possibly translate for log.   */
    '|   change //I /',           /* Mark as user's input.         */
    '| y:faninany',               /* Gather records for log file.  */
    '|   >>' logfile,             /* Append to the log file.       */
    '? f:',                       /* User input for real server.   */
    '|   tcpclient' remote_sys remote_port,    /* Play client.   */
    '| o:fanout',                 /* Divert copy of server output. */
        xlate,                    /* Possibly translate for log.   */
    '|   change //O /',           /* Mark as server's output.      */
    '| y:',                       /* Merge with user's input.      */
    '? o:',                       /* Server output to here.        */
    '|   elastic',                /* (To prevent a stall.)         */
    '| i:'                        /* Feed server output to user.   */

  Say 'Connection to' remote_sys remote_port 'closed.'

End

Error: Exit RC*(RC<>12)
```

The sipping subroutine pipeline in the tcpsniff filter is invoked once per connection request. It receives one socket descriptor record from the calling pipeline and passes that to the fanin stage. fanin passes that record to tcpdata and then sees end-of-file on its primary input, so it switches to reading from its secondary input. tcpdata takes the socket described by that first record and

establishes communication with the client, playing the role of a server. When tcpdata receives a packet from the client, it writes it on its output stream, where the fanout with label "f:" writes copies to both of its output streams. The first copy from fanout is for the log file. It is translated to EBCDIC (if that was requested) and then flagged as being an input record and appended to the log file. The second copy from fanout goes to the second pipeline segment, where it is fed into the tcpclient stage, which sends it to the real server.



The output of tcpclient consists of packets from the real server intended for the real client. Another fanout is used to make two copies of those packets. The first copy is translated, marked as output, and sent to the log file. The second copy is diverted to the last segment of the pipeline, where it may be buffered by elastic before being fed into the secondary input of the fanin stage in the first pipeline segment. fanin feeds those packets into tcpdata, thus sending the output from the real server to the real client.

When the client ends the TELNET session, the tcpdata stage sees the connection close, so it terminates, and end-of-file propagates through the subroutine pipeline, causing the CALLPIPE command to complete and leaving this REXX filter waiting for another record describing the socket for another client connection request. Ultimately, a STOP immediate command will cause end-of-file to propagate through the calling pipeline, so that the PEEKTO in this REXX filter will receive RC 12, and the filter will terminate.

Building a multi-tasking client

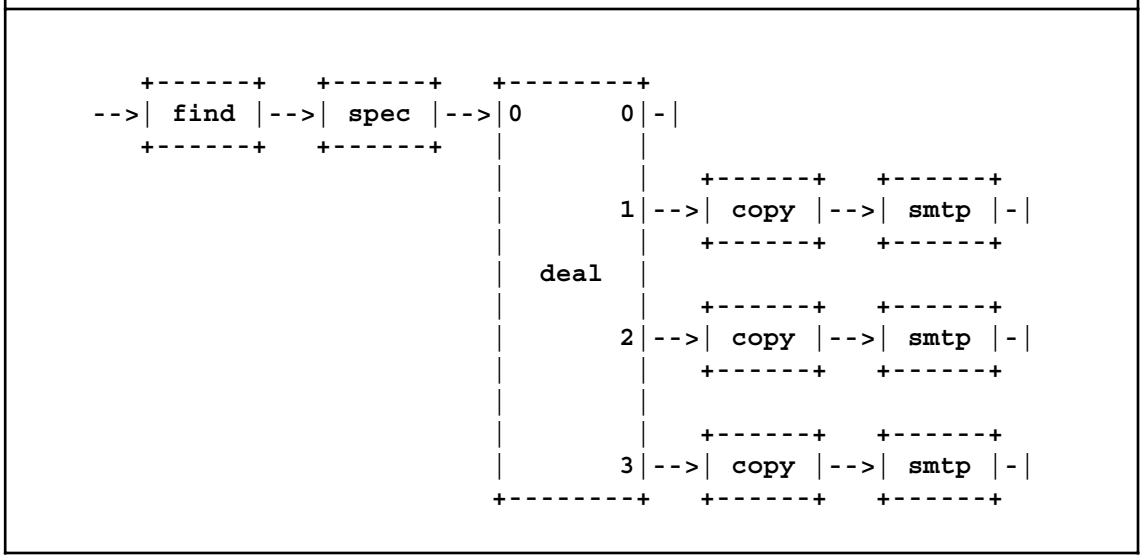
There are situations in which greater total throughput can be attained if a client has multiple connections to its server. There are several ways to achieve the effect of multi-tasking in a pipelined client. Let's start with a simple one-connection case, an SMTP client:

```
PIPE  starmsg  |  find ... |  spec  26.4  1  |  smtp
```

A starmsg stage captures CP information messages, and a find (which isn't completely shown) selects the messages describing arriving reader files. spec isolates the spoolid number, which is passed to the smtp stage, signalling it to process the corresponding spool file. When the smtp stage has finished processing the spool file, it will read another spoolid record from its input and then process that file, continuing to process one file at a time forever.

An easy way to add more connections to this scheme is to build a pipeline containing multiple smtp stages and to use a deal stage to round-robin the files amongst the smtp stages:

```
'PIPE (endchar ?)',                 /* Dispatch by round-robin:    */
      'starmsg |',                  /* Capture spool arrivals.     */
      'find' arrival_msg'|',        /* Select file arrival message.*/
      'spec 26.4 1 |',              /* Isolate spoolids.           */
   'd: deal',                       /* Dole them out to SMTPs.     */
'?',
   'd: | copy | smtp',              /* DEAL's output stream 1.     */
'?',
   'd: | copy | smtp',              /* DEAL's output stream 2.     */
'?',
   'd: | copy | smtp'               /* DEAL's output stream 3.     */
```



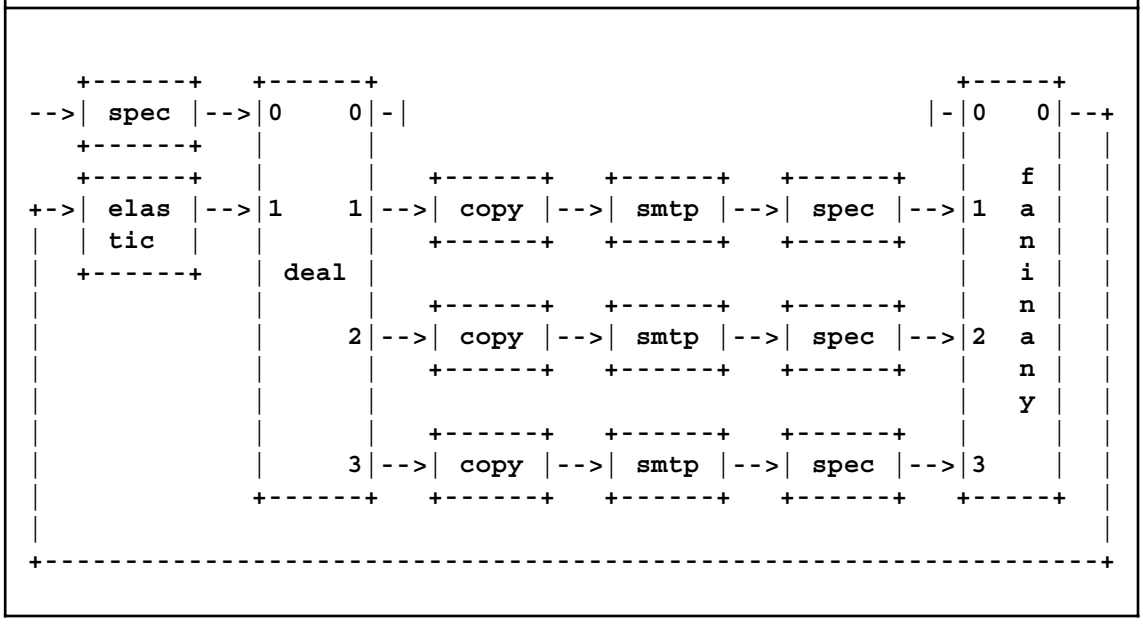
The deal stage was added in CMS 12. It works like a card player dealing cards, passing records to its output streams in turn. In this example, deal has no primary output stream, so it passes the first record it receives to its secondary output stream (stream 1), the next record to its tertiary (stream 2), the next to it quaternary (stream 3), and then the next to its secondary again, continuing in sequence to “round-robin” its output.

If you want these smtp stages to run in parallel, though, you must be careful that the output of deal is consumed “quickly”, so that deal can continue passing out spoolid records to other streams. The copy stages consume the output of deal to unblock it immediately. (The function of copy is to consume a record before passing it on. It can be thought of as a one-record buffer.)

This scheme will allow the smtp stages to run in parallel. However, one of the smtp stages might be sending a large file on a slow link, so it might not be ready to consume a spoolid record the next time through the round-robin process. That would block deal and cause the other smtp stages to go idle, because deal could not give them any work to do. One could change each copy stage to elastic, of course. That would prevent the blocking of deal, but would still leave files queued waiting for busy smtp stages to complete.

A better scheme would be not to dispatch the SMTP processes in round-robin fashion at all, but instead to let each process queue for a file when it is ready for one. This pipeline does that:[2]

```
'PIPE (endchar ?)',                 /* Dispatch on request:        */
      'starmsg |',                  /* Capture spool arrivals.     */
      'find' arrival_msg'|',        /* Select file arrival message.*/
      'spec 26.4 1 |',              /* Isolate spoolids.           */
   'd: deal secondary',             /* Dole them out to SMTPs.     */
'?',
   'y: faninany | elastic |',       /* Secondary input of DEAL.    */
   'd: | copy | smtp | spec /1/ 1 | y:', /* DEAL's output strm 1.*/
'?',
   'd: | copy | smtp | spec /2/ 1 | y:', /* DEAL's output strm 2.*/
'?',
   'd: | copy | smtp | spec /3/ 1 | y:'  /* DEAL's output strm 3.*/
```



As before, each smtp stage gets its input from an output stream of deal, but now each smtp stage also has an output stream of its own that feeds ultimately into the secondary input of deal (through the spec, faninany, and elastic). deal is invoked with the secondary option, which says to read a stream of stream numbers from its secondary input and write the records from its primary input to the specified output streams. So, when an SMTP process is ready for another input file, it simply produces a record containing its stream number, thus queuing for output from deal secondary. That is, when a process is ready for work, it writes an output record containing its stream number to an input stream of faninany, and faninany feeds those stream number records to elastic, which buffers them for feeding into the secondary input of deal secondary (the second occurrence in the pipeline of the label “d:”), in the order in which they were received.

---

[2] For a more bullet-proof implementation of this scheme, see Appendix A.

Letting the processes queue for work significantly improves throughput. The elapsed time to deliver 1,000 identical files using 5 processes was reduced by nine percent as the result of this change. The multi-tasking scheme used here is quite effective. In the production version of this SMTP client, the total throughput with 4 processes is twice as great as with 1. (It increases little with additional processes, presumably as the result of the synchronization imposed by the DIAGNOSE used to read from the spool.)

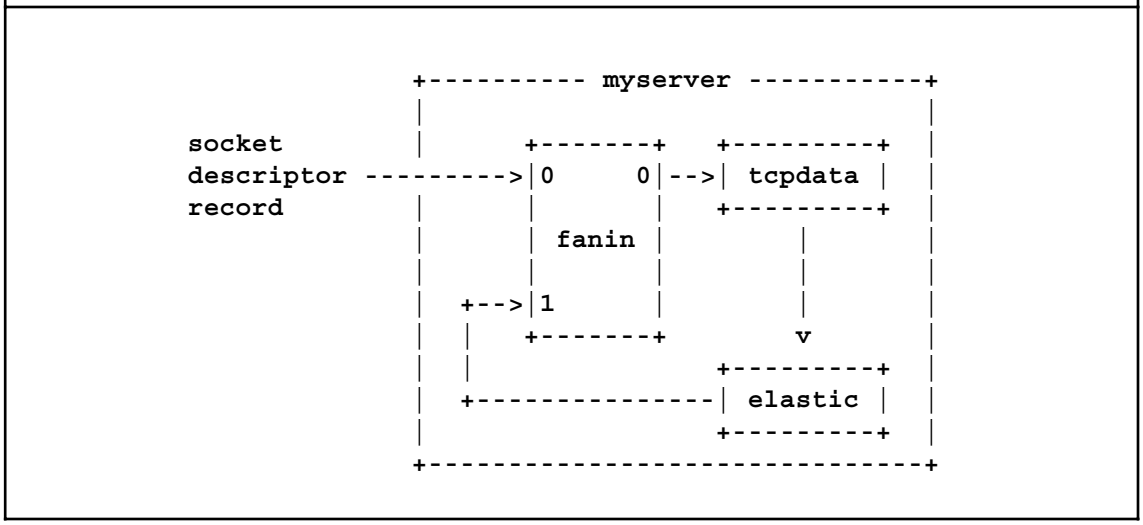
Building a multi-tasking server

Obviously, when one is building a server, being able to handle multiple simultaneous connections becomes even more important. Although one could build a server in which a processing stage, such as myserver, immediately followed the tcplisten stage:

```
PIPE tcplisten 260 | myserver
```

that would not be very useful as a server, because it would be able to handle only one connection at a time. Fortunately, it is not at all difficult to build a piped server that can handle many simultaneous connections. Let's start by reviewing myserver:

```
'CALLPIPE (listerr name myserver)',  /* Simplistic ECHO server:  */
     '*: |',                 /* Get socket record from TCPLISTEN.  */
     'take 1 |',             /* Take just one socket descriptor.   */
  'i: fanin |',              /* Then client output from secondary. */
     'tcpdata |',            /* Send to client & get from client.  */
     'elastic |',            /* Buffer to avoid stall.             */
  'i:'                       /* Feed client output to FANIN.       */
```



You will recall that myserver is the skeleton for a server process. It receives a record describing the socket for a client on its primary input and passes that to tcpdata, which communicates with the client across the network. tcpdata writes packets received from the client *via* TCP to its output stream, where elastic buffers them and feeds them to fanin, which then feeds them back to tcpdata, which sends them back across the network to the client.

In order to build a server that can communicate with multiple clients simultaneously, all we need do is build a pipeline set with multiple tcpdata stages and let the *Pipelines* dispatcher take care of the multi-tasking for us. The scheme used earlier for building a multi-tasking client can also be used very successfully to build a multi-tasking server. (See Appendix B for an example.) However, I will also describe another approach that makes interesting use of ADDPIPE.

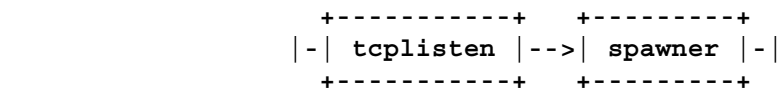
This example (from the author's help file for tcplisten) starts with a simple pipeline of the form:

```
PIPE tcplisten 260 | spawner
```

where spawner is a simple subtasking stage that issues an ADDPIPE to spin off a new server process every time it receives a socket descriptor record from tcplisten:

```
/* SPAWNER REXX:        Launch server process for every request  */

Signal On Error

Do Forever
  'PEEKTO'                          /* Wait for connection request.*/
  'ADDPIPE *.output: | myserver'    /* Launch a server process.    */
  'CALLPIPE *: | take 1 | *:'       /* Pass descriptor to process. */
  'SEVER OUTPUT'                    /* Let process run unconnected.*/
End

Error: Exit RC*(RC<>12)
```

We start with the simple two-stage pipeline:

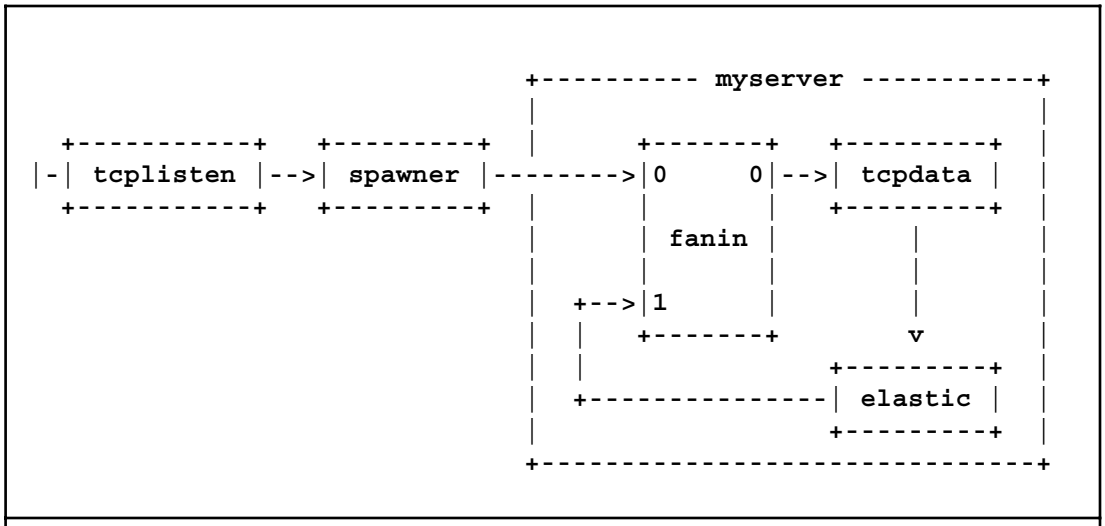


The input of spawner is connected to the output of tcplisten. tcplisten's input and spawner's output are not connected to anything.

But, the four *Pipelines* commands in the Do Forever group in SPAWNER REXX implement a loop that spawns a separate pipeline for each client connection request:

1. spawner issues the PEEKTO command and sits in that PEEKTO waiting for input from the tcplisten stage. When tcplisten receives a connection request from a client, it writes the record containing the description of the socket to its output stream, thus allowing the PEEKTO to see an input record and complete. Note, however, that a PEEKTO does not consume its input record, so the record written by tcplisten remains in the pipeline.

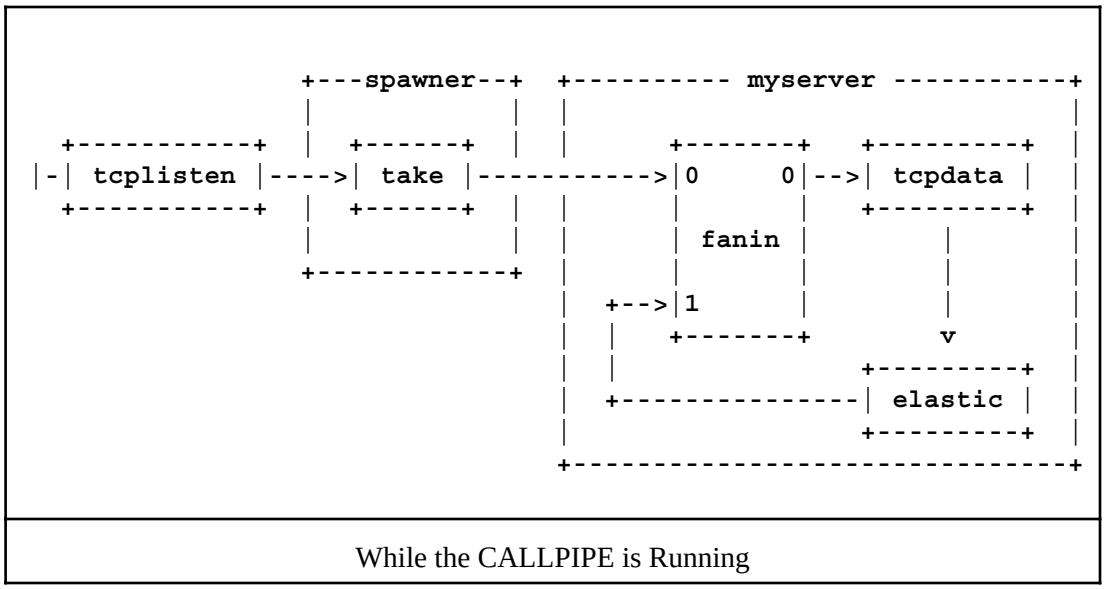
2. When the PEEKTO completes, the ADDPIPE command is issued. It adds a pipeline that *begins* with a connector (*.output:) that attaches to the *output* of the spawner stage. (Thus, as you will see from the following diagram, the myserver stage in the added pipeline has its input stream connected to the output of the spawner stage, and, thus, the fanin in the subroutine pipeline in myserver has *its* input connected to the output of spawner.)



After the ADDPIPE is Issued

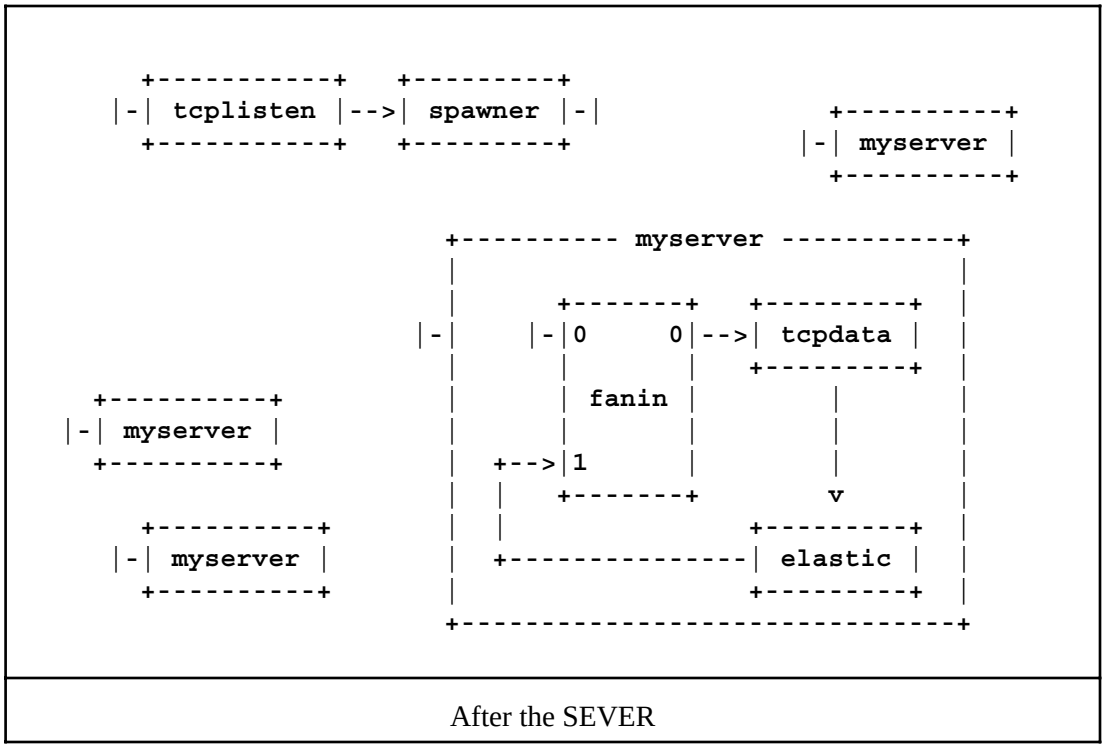
One of the differences between ADDPIPE and CALLPIPE is that an ADDPIPE command completes as soon as the added pipeline has been started, rather than when the added pipeline completes, so spawner is free to issue its next command (the CALLPIPE) immediately after the pipeline added by the ADDPIPE has started up.

3. The CALLPIPE has connectors to both the input and the output of the spawner stage. It will continue to run until one record has passed through it from its input to its output, satisfying its take 1. Thus, because the pipeline created by the ADDPIPE is connected to the output of spawner, which has been taken over temporarily by the CALLPIPE, the CALLPIPE does not complete until the record from tcplisten has been read by the myserver stage in the added pipeline (that is, until the socket descriptor record from tcplisten has been consumed by the tcpdata stage in the subroutine pipeline in the myserver stage). Then the take 1 terminates and the CALLPIPE command completes.



While the CALLPIPE is Running

4. After the CALLPIPE has completed, spawner severs its output stream to break the connection to the added pipeline, which can continue running independently for as long as it likes, continuing to communicate with its client and leaving the spawner stage free to receive more client connection requests from tcplisten and to add more pipelines to process them.



After the SEVER

Much can be accomplished using this very simple structure and these very powerful new stages. Note, however, that the spawner example, as I have shown it here, has no governor to prevent the creation of too many simultaneous tasks. I would advise you **not** to use this scheme in production without devising such a mechanism. Go instead with the scheme used in the SMTP example.

Other pipeline stages supporting IP programming

Several utility stages have been added to *CMS/TSO Pipelines* recently to facilitate IP programming:

hostbyaddr: The hostbyaddr stage resolves an IP address to a fully qualified domain name. It uses the TCPIP DATA file to determine the name server(s) to use. It requires RXSOCKET Version 2 or later. Input records contain “dotted-decimal” IP addresses. The domain name is written to the primary output. If the secondary output is defined, unresolved addresses are written to it.

hostbyname: The hostbyname stage resolves an Internet host name to an IP address. It uses the TCPIP DATA file to determine the name server(s) to use. It requires RXSOCKET. Input records contain host names in the local domain (if they have no periods) or fully qualified host names. The IP address (dotted-decimal) is written to the primary output. If the secondary output is defined, unresolved host names are written to it.

hostid: The hostid stage writes the default IP address associated with the host or the IP address associated with the TCP/IP server whose userid was specified as an argument.

hostname: The hostname stage writes the host name (not including the domain name) associated with a given TCP/IP server.

ip2socka: The ip2socka stage builds a "sockaddr_in" structure, converting a readable port number and IP address or host name to the 16-byte socket address structure.

socka2ip: The socka2ip stage converts a sockaddr_in structure to a readable port number and IP address or converts a 4-byte binary input to a readable IP address.

urldeblock: The urldeblock stage deblocks and decodes the "url-x" format used to encode and block data from Web forms, *etc.* The processing assumes an ASCII encoding of the input data. Each input record is one URL to be expanded. It is split into multiple records at ampersands and semicolons. Plus signs are changed to blanks. "%xx" is replaced by the single character with the hexadecimal value "xx".

udp: The udp stage implements the IP "User Datagram Protocol". Its syntax is:

```
UDP             localport  [ASYNChronously]  [USERid vmid]
                [LOCALIPaddress [ANY|HOSTID|<IPaddress>]]
                [GETSOCKName] [BROADCAST] [STATistics]

ASYNChronously Receive and send asynchronously.
BROADCAST      Enable socket option.
GETSOCKName    Write port number as first record.
LOCALIPaddr    Network interface to be used.
STATistics     Report statistics.
USERid         TCP/IP server virtual machine name.
```

udp was added to *Pipes* for CMS 10, and some minor enhancements were made for CMS 12. Several new options have been added since CMS 12. When udp is the first stage in a pipeline, it listens on the specified local port and writes any messages it receives into the pipeline. When udp is not a first stage, it reads records from its input stream and sends them as UDP messages to the host and port specified in the record headers. It also receives UDP messages on the specified local port and writes them to its output stream. (If "0" is specified for the local port number, the TCPIP server assigns an available port.) If the option asynchronously is used, the messages are read and written as they arrive. Otherwise, one record is written into the pipeline for each message sent to the UDP port; if no response has been received within the timeout interval, a null record is written into the pipeline to indicate a timeout. ECHO EXEC shows udp being used in that mode:

```
/* ECHO EXEC:                          Send to UDP Echo port on    */
/*                                     ponyexpress.princeton.edu   */

PIPE (endchar ? name UDPEcho)', /* Send to UDP echo port:      */
     'literal AF_INET 7 ponyexpress |',   /* Pony's echo port. */
     'ip2socka |',               /* Build sockadr_in structure. */
     'spec pad 00',              /* Include in SENDTO record:   */
       'x00000010 1',            /* Specify 16-second timeout.  */
       '1-* 9',                  /* Protocol, port, IP address. */
       '/Hello, Ponyexpress/ 25 |',           /* The datagram. */
     'udp 0 |',                  /* Write to Pony's echo port.  */
  '1: locate 1 |',               /* Divert response if null.    */
     'spec 5.4 1 |',             /* Extract just IP address.    */
     'socka2ip |',               /* Make IP address readable.   */
     'hostbyaddr |',             /* Convert to host name.       */
     'change //Reply received from / |',        /* Garnish it. */
     'console',                  /* Display on console.         */
'?',
  '1: |',                        /* Null records to here.       */
     'spec /Time-out/ 1 |',      /* Request timed out.          */
     'console'                   /* Display on console.         */
```

A message consisting of a datagram preceded by a 24-byte header is sent to UDP port 7 on ponyexpress.princeton.edu and the response, if any, is written into the pipeline and reformatted as the message “Reply received from ponyexpress.Princeton.EDU”. If no response is received within the specified interval (16 seconds), a null record is written into the pipeline and reformatted as a “Time-out” message.

Other Pipelines capabilities to learn about

Writing pipelined servers, whether TCP-based or not, is much easier if one becomes familiar with some facilities of *Pipes* that are less often used in other types of applications:

ADDPIPE: ADDPIPE can greatly reduce the complexity of your pipelines, in addition to being very useful in spawning independent tasks. If you are not yet conversant with ADDPIPE, Appendix E will give you a tutorial on its capabilities.

beat: The beat stage added in CMS 12 provides a “heartbeat monitor” for a pipeline; this is particularly useful in constructing servers.

block and deblock: block and deblock have had many new options added in recent releases. You will do well to learn all the permutations, as they can help you achieve very good performance.

delay: The delay stage is used to wait until a particular time or for a specified interval. Other portions of the pipeline can run while the stream with the delay stage is waiting, so delay can be used to make your server perform certain tasks at a predefined time.

immcmd: immcmd allows you to set up a pipeline stream that will be executed when you enter a console command, so that the command will be acted upon by your pipeline while it is running other streams. immcmd can provide you with powerful probes into your server, but you will need to take some care to get it to terminate when you want the pipeline to shut down. The following fragment shows one way of doing that:

```
'CALLPIPE (endchar ?)',
      . . .
   'o: fanout |',              /* Second copy to kill IMMCMD. */
      '*:',                    /* Output to the pipeline.     */
'?',
   'o: |',                     /* Copy of output to here.     */
      'take last 1 |',         /* Wait until have it all.     */
   'g: gate',                  /* Close GATE to kill IMMCMD.  */
'?',
      'immcmd JEREMY |',       /* Get JEREMY requests.        */
   'g: |',                     /* (For terminating IMMCMD.)   */
      'jeremy |',              /* Format state of pipeline.   */
      'console',               /* And display it.             */
'?',
       . . .
```

juxtapose: juxtapose is one of the most powerful of all the stages in *Pipelines*. It allows you to mate records on one stream with records arriving asynchronously on another stream. Although it may appear formidable at first, juxtapose soon becomes intuitive, and it can greatly reduce the complexity of your pipelines. You can find several examples of using juxtapose in my *Pipe Dreams* paper.

listrc/listerr: Learn to use the listerr and listrc global options. listerr causes a trace to be displayed when a pipeline stage returns with a non-zero return code. After being burned many times, I finally resolved always to use listerr in any production pipeline that is not expected to produce a non-zero return code. This makes diagnosing failures much easier.

One doesn't need listrc as often, but it can be very helpful. It specifies that a trace is to be displayed whenever a stage starts or ends. This can often provide the clue you need when you don't understand why your pipeline is misbehaving.

Rita and jeremy: Rita is a pipeline execution profiler that is shipped on the CMS "Samples and Examples" disk. Profiling your pipeline with Rita will tell you where to place your efforts to improve its performance.

jeremy is a companion to Rita. It displays the state of all the stages in a pipeline set. jeremy is for use when your pipeline is going Nowhere. It can be invoked as an immediate command to help you diagnose a situation in which a sub-pipe is stalled, but the overall pipeline is waiting for

an external event and, thus, the stall is not detected by the pipeline dispatcher. The display produced by jeremy is enhanced over the stall trace produced by the *Pipelines* dispatcher in two ways: It tells you which streams are selected, so you don't have to guess whether a stage is trying to write output to its primary or its secondary output stream, for example, and it displays a portion of the output record for each stage that is blocked waiting for its output to be consumed.

jeremy is built into the version of *Pipelines* shipped in CMS 14. Versions of jeremy and Rita that will run with older levels of *Pipelines* can be found on the Runtime Library Web page (http://pucc.princeton.edu/~pipeline/).

Pipeline termination: Until one begins writing pipelined servers, there is seldom a need to understand how end-of-file flows (or does not flow) backward and forward through a pipeline. Once one begins building seriously pipelined servers, understanding the pipeline termination process becomes essential. Before beginning such a project, it is worth learning about stages such as gate, pipestop, and fanouttwo and about the stop options on various other stages. The author's help files give detailed information on the termination behavior of each stage, as, for example, whether they terminate "prematurely" when their output is severed. You can find an overall discussion of pipeline termination in my *Pipe Dreams* paper.

Record delay: It becomes equally important to understand the concept of "record delay" and to know how to control the flow of records through your pipeline. You will want to learn to use synchronise and elastic and copy and to know when elastic is overkill. There are discussions of synchronise and of record delay in *Pipe Dreams*. I particularly recommend the discussion of record delay in John Hartmann's paper *CMS Pipelines Explained*. For an expanded discussion, see my paper *Record Flow in CMS Pipelines*.

Where to find examples

Before embarking on a project that uses the *Pipelines* support for TCP, you may wish to look at some full-scale examples that use it. John Hartmann has provided an extensive example in the Plumbers' Workbench, which is available from the Runtime Library Web page.

In Appendix A, you will find my SMTPLite program, which has been quite successful running as an "Interbit" (a BITNET/Internet gateway) on a production P/370 system in Germany that had been being badly destabilized by the excessive resource requirements of the IBM SMTP for VM.

In Appendix B, you will find a description of the steps in converting Rick Troth's Webshare Web server for VM from RXSOCKET to the built-in TCP stages (and of making it multi-tasking, so that it can support multiple clients simultaneously).

In Appendix D, you will find Rob van der Heij's LURKY EXEC, which simulates a Web browser for the purpose of measuring service times. Rob's paper *Writing a Piped TN3270 Client* can be found on the Runtime Library Web page.

The Web page also has Ross Patterson's UDPSNIFF program, which you will want to consult if you are implementing a UDP application. Also on the Web page, you will find a finger client, a finger daemon, and an ident daemon.

Conclusion

At Princeton, we have several critical Internet-based applications built on the *CMS Pipelines* TCP support. These were all written in an amazingly short time by somebody who still can't spell "socket".

VM needs many more Internet tools. In the *CMS Pipelines* TCP support, the VM community has a programming language that makes writing such tools almost trivial. I hope I have persuaded you to try your hand at it, too, as I think you will be astonished at how easy it is.

Appendix A

EXAMPLE: A LIGHT-WEIGHT CLIENT SMTP

This appendix provides an extended example of using the *CMS Pipelines* TCP support for a production application, a light-weight client-only SMTP called SMTPLite.

Description of SMTPLite

SMTPLite receives well-formed BSMTP ("Batch SMTP") files in NETDATA format in its virtual reader and uses the SMTP protocol to transmit them to a server SMTP without regard to the ultimate destination of the files. (The server SMTP is assumed to be a high-performance, full-function SMTP, typically on a high-performance workstation.)

Efficiency, particularly I/O efficiency, was the primary design goal of the SMTPLite project, as the purpose was to build an SMTP that would run well on a production Personal/370 system. (The SMTP distributed with the IBM TCP/IP for VM product (*aka* "FAL") runs particularly badly on a P/370, due to its extremely high minidisk I/O rate.)

The power of *CMS Pipelines* made it easy to achieve the performance goals without resorting to Assembler code. An important consideration for good performance was that the files would be moved straight from the virtual reader to the TCP connection without being written to the minidisk and without being written into a REXX stemmed variable.

Considerable attention was also given to achieving the fastest possible pipelined NETDATA RECEIVE function. (The current version has been clocked at 2.25 million files per day on an IBM 3090-600E, for simply receiving small files from the virtual reader, converting them to plaintext, and throwing them into a hole.)

SMTPLite supports multiple connections to the server SMTP. The optimal number depends on the receiving SMTP. These timings are from SMTPLite running on a 3090; all were taken during periods of light load on the receiving systems:

- To Babybear (P/370)

  — 1 stream 0.9 files/second
  — 3 streams 1.1

- To Ponyexpress (SPARCserver 10/71)

  — 1 stream 2.0 files/second
  — 3 streams 2.8
  — 5 streams 2.5

- To PUCC (3090-600E):

  — 1 stream 5.4 files/second
  — 3 streams 8.7
  — 4 streams 10.8

— 5 streams 10.9
— 8 streams 11.0

In order to achieve high performance, the implementation was stripped down as much as possible. SMTPLite was designed to recognize cases it cannot handle and to transfer the spool files to a full-function SMTP for it to process. (For example, if a “5yz” response is received from the SMTP server while a mail file is being transmitted, that spool file is transferred to the full-function SMTP). Thus, SMTPLite never needs to send delivery failure mail.

Except for the “HELO” command, which it creates itself, SMTPLite simply transmits the SMTP commands contained in a BSMTP file. (It strips out the “HELO” and “QUIT” commands, because it transmits multiple files during the life of the connection.) It assumes that the BSMTP commands are in upper-case and that the producer of the BSMTP file did a proper job of making leading periods in the mail text transparent. (There is some checking of the latter and that the BSMTP file is well-formed.)

SMTPLite appears to comply with RFC821, RFC1123, and the draft RFC “Requirements for Internet Mail Transport Agents” (March 24, 1995), with the exception that it does not insert a “Received” line. (This violates section 4.1.1 of RFC821 and section 5.2.8 of RFC1123.)

SMTPLite implements none of the standard SMTP service extensions. Experiments with implementing the extension described in the draft RFC “SMTP Service Extension for Command Pipelining” (April 2, 1995), make it clear that that would be a definite win. (I have no server SMTP that implements command pipelining with which to test, but I have used this protocol while sending to the FAL SMTP on Babybear and have measured a substantial increase in delivery rate right up to the time that Babybear’s SMTP program checks.)

SMTPLite does implement a private extension that allows the server SMTP not to acknowledge the individual RCPT commands. This results in a major increase in throughput when handling mail for large mailing lists. (Controlled timing tests on a P/370 with 700-recipient mail files showed an increase from 11 to 350 recipients per second when this extension was enabled.)

Implementing SMTP as a pipeline

Implementing SMTP as a pipeline was at first awkward, because the SMTP protocol has traditionally been defined in terms of state diagrams, not data flow. For example, RFC821 speaks of “resetting state tables” and “clearing buffers”. Here (slightly abbreviated) is the description from the RFC of the SMTP dialogue and the corresponding state diagrams:

> **4.3. Sequencing of commands and replies**
>
> The communication between the sender and receiver is intended to be an alternating dialogue, controlled by the sender. As such, the sender issues a command and the receiver responds with a reply. The sender must wait for this response before sending further commands.

One important reply is the connection greeting. Normally, a receiver will send a 220 "Service ready" reply when the connection is completed. The sender should wait for this greeting message before sending any commands.

The table below lists alternative success and failure replies for each command. These must be strictly adhered to; a receiver may substitute text in the replies, but the meaning and action implied by the code numbers and by the specific command reply sequence cannot be altered.

Command-Reply Sequences. Each command is listed with its possible replies. The prefixes used before the possible replies are "P" for preliminary (not used in SMTP), "I" for intermediate, "S" for success, "F" for failure, and "E" for error. The 421 reply (service not available, closing transmission channel) may be given to any command if the SMTP-receiver knows it must shut down. This listing forms the basis for the State Diagrams in Section 4.4.

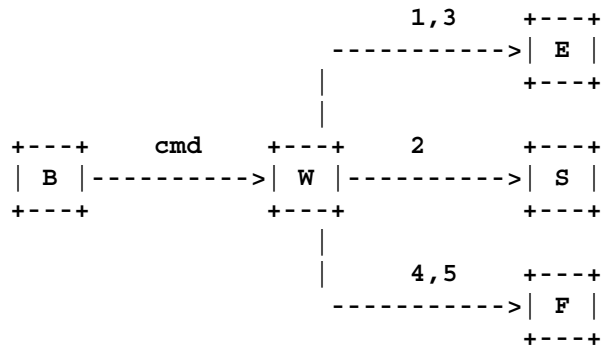
```
CONNECTION ESTABLISHMENT
   S: 220
   F: 421
HELO
   S: 250
   E: 500, 501, 504, 421
MAIL
   S: 250
   F: 552, 451, 452
   E: 500, 501, 421
RCPT
   S: 250, 251
   F: 550, 551, 552, 553, 450, 451, 452
   E: 500, 501, 503, 421
DATA
   I: 354 -> data -> S: 250
                     F: 552, 554, 451, 452
   F: 451, 554
   E: 500, 501, 503, 421
QUIT
   S: 221
   E: 500
```

4.4. State Diagrams

Following are state diagrams for a simple-minded SMTP implementation. Only the first digit of the reply codes is used. There is one state diagram for each group of SMTP commands. The command groupings were determined by constructing a model for each command and then collecting together the commands with structurally identical models.

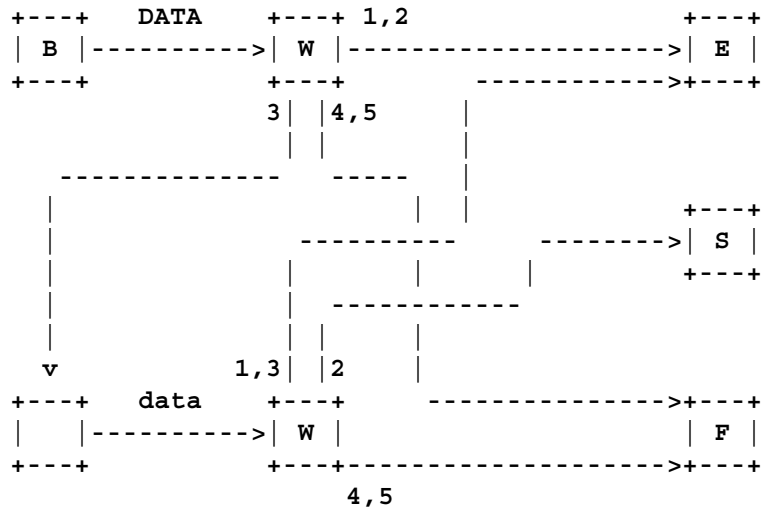
For each command there are three possible outcomes: “success” (S), “failure” (F), and “error” (E). In the state diagrams below we use the symbol B for “begin”, and the symbol W for “wait for reply”.

First, the diagram that represents most of the SMTP commands:



This diagram models the commands: HELO, MAIL, RCPT, QUIT.

A more complex diagram models the DATA command:



Note that the “data” here is a series of lines sent from the sender to the receiver with no response expected until the last line is sent.

In the end, it was not difficult to “pipethink” this process, for careful control of the flow of records through a pipeline has much the same effect as the state transitions described in the RFC. An input is not consumed until the server has replied with a response indicating successful or unsuccessful processing of that input. When that response is consumed, end-of-file travels backwards until the original input is also consumed, allowing the next input to be processed, in effect changing the state of the pipeline.

The pipelined implementation (shown on the following pages) has multiple identical processes, one for each connection between the client and the server. Arriving BSMTP files are dispatched to the next ready SMTP process, where they are tested for being well-formed and then fed through a tcpclient stage to the SMTP server.

To achieve the transition between the two main states of sending commands or sending mail, SMTPLite first forms its input into two records, one containing all of the commands in the BSMTP envelope and the other containing all of the lines of the mail text. These two records are fed into a synchronise stage to control their flow through the pipeline. synchronise will write neither of them until both exist. Once it has both, it sends the first (the commands) through the appropriate pipeline path and then holds the second until the first has been consumed. The first record is consumed when the pipeline should make a state transition; this causes the second record (the mail data) to be sent along a different path through the pipeline.

The dialogue between the server and the client in the normal case is as follows:

```
S> 220 babybear.Princeton.EDU running IBM VM SMTP V2R2 on . . .
C> HELO pucc.Princeton.EDU
S> 250 babybear.Princeton.EDU is my domain name.
C> MAIL FROM:<MAINT@PUCC.PRINCETON.EDU>
S> 250 OK
C> RCPT TO:<MAINT@BABYBEAR.PRINCETON.EDU>
S> 250 OK
C> DATA
S> 354 Enter mail body.  End by new line with just a '.'
C> Date:  Wed, 06 Jan 93 17:27:54 EST  From:  Melinda Varian . . .
S> 250 Mail Delivered
   . . .
   (Iterations of sequence between MAIL command and "250 Mail Delivered" response)
   . . .
C> QUIT
S> 221 babybear.Princeton.EDU closing connection
```

When all of the commands from the envelope have been executed successfully, the server sends a 354 response (the prompt to begin sending the mail text). This 354 record is consumed, causing end-of-file to travel backwards in the pipeline until the single record containing all of the envelope commands is also consumed. Once that record has been consumed, the synchronise stage is free to produce the single record containing the entire mail text.

When all the lines of the mail text have been sent, the server sends a 250 response. This 250 record is consumed, causing end-of-file to travel backwards in the pipeline until the single record containing the entire mail text has been consumed. When that record has been consumed, the pipeline reverts to the state in which it can begin processing another BSMTP envelope.

When a transaction is not successful, the response record indicating the failure or error state is converted into an SMTP QUIT command that flows back to the SMTP server to close the connection. The connection closing message (221) then flows into a gate stage that terminates the pipeline, allowing it to start up again in the initial state.

It was a relief to me to realize that the somewhat formidable diagram that models the DATA command could be implemented by simply passing the mail text to tcpclient as a single CRLF-delimited record and letting *Pipes* handle the process shown in the diagram.

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/*                          SMTPLITE EXEC                             */
/*                                                                    */
/* Copyright Princeton University, 1995, 1996.  Permission is         */
/* granted to SHARE, Inc., and its members to reproduce this work     */
/* for use as a coding example.  This work may not be executed in     */
/* production without permission of the Office of Computing and       */
/* Information Technology of Princeton University.                    */
/*                                                                    */
/*--------------------------------------------------------------------*/

Signal On Novalue
Signal On Syntax
Signal On Error

Address Command


/*--------------------------------------------------------------------*/
/* Install the field-test version of CMS Pipelines.                   */
/*--------------------------------------------------------------------*/

Signal Off Error
'SET CMSTYPE HT'
'NUCXDROP PIPMOD'                    /* So will use NXPIPE segment. */
'SEGMENT PURGE PIPES'                /* Don't want built-in pipes.  */
'SEGMENT PURGE NXPIPE'               /* (So return code will be 0.) */
'SEGMENT LOAD NXPIPE'                /* Load experimental Pipes.    */
If RC ¬= 0                           /* If segment didn't load,...  */
   Then 'EXEC PIPINIT TEST'          /* ...install a private copy.  */
'SET CMSTYPE RT'
Signal On Error


/*--------------------------------------------------------------------*/
/* Load parameters from PARMS file and environment.                   */
/*--------------------------------------------------------------------*/

'EXEC PARMSMTP'                      /* Initialize PARM. stem.      */


/*--------------------------------------------------------------------*/
/* Set some variables we need.                                        */
/*--------------------------------------------------------------------*/

myid = Left(parm.myself,8,'_')       /* Userid constant for FIND.   */

If parm.isitsp  Then Do              /* VM/SP or VM/ESA 370 Feature?*/
   avail = ''                        /* Yes, AVAIL not available.   */
   arrival_msg = '00000007'myid'PUN_FILE_    _FROM'
End
Else Do                              /* VM/ESA ESA Feature.         */
```

```
   avail = 'AVAIL'
   arrival_msg = '00000007'myid'RDR_FILE_    _SENT_FROM'
End


/*--------------------------------------------------------------------*/
/* SPOOL the virtual reader suitably.                                 */
/*--------------------------------------------------------------------*/

'CP SPOOL RDR CLASS * HOLD'           /* So READER won't purge.      */


/*--------------------------------------------------------------------*/
/* Construct a pipeline segment containing the specified number of    */
/* client-SMTP processes for inclusion in the central pipeline.       */
/*                                                                    */
/* The first process looks like this:                                 */
/*                                                                    */
/*         . . .                                                      */
/*     'd: |',                   /* Next spool file id to here. */    */
/*     't1: smtplite 1 |',       /* Parse file and send to SMTP.*/    */
/*     'y:',                     /* "1" recs from here to DEAL. */    */
/*  '?',                                                              */
/*     'o: |',                   /* STOP record to here.        */    */
/*     't1:',                    /* Into SMTPLITE1's secondary. */    */
/*  '?',                         /* --------------------------- */    */
/*         . . .                                                      */
/*                                                                    */
/* Processes 2-n are identical to process 1, except for the argument  */
/* to the SMTPLITE stage and the label on the SMTPLITE stage.         */
/*--------------------------------------------------------------------*/

processes = ''                        /* Initialize pipeline segment.*/

Do i = 1 to parm.processcount         /* Build process segments.     */

   processes = processes,             /* Append another process:     */
      'd: |',                         /* Next spool file id to here. */
      't'i': smtplite' i '|',         /* Parse file and send to SMTP.*/
      'y:',                           /* Streamnumber recs from here.*/
   '?',
      'o: |',                         /* STOP record to here.        */
      't'i':',                        /* Into SMTPLITE's secondary.  */
   '?'

End


/*--------------------------------------------------------------------*/
/*          Pass BSMTP spool files to client-SMTP processes.          */
/*                                                                    */
/* This is the central pipeline of SMTPLite.  Its function is to      */
/* dispatch work to the user-specified number of client-SMTP          */
```

```
/* processes.                                                        */
/*                                                                   */
/* When this pipeline starts up, four immediate commands are         */
/* established:                                                      */
/* 1.  CP, which is used to issue CP commands from the console;      */
/* 2.  DUMP, which writes a PIPDUMP to the A-disk;                   */
/* 3.  JEREMY, which formats the status of the pipeline for display  */
/*     on the console; and                                           */
/* 4.  STOP, which stops SMTPLite.                                   */
/*                                                                   */
/* The STOP immediate command notifies each client-SMTP process to   */
/* send a QUIT command to the SMTP server.  It does this by sending  */
/* a STOP record into the secondary input of each SMTPLITE stage.    */
/* The SMTPLITE stage converts that to a QUIT command, which it      */
/* sends to the SMTP server; the client-SMTP process then terminates */
/* as soon as it gets a 221 (connection closing) response from the   */
/* SMTP server.                                                      */
/*                                                                   */
/* The STOP immediate command also sends a record into the primary   */
/* input of the GATE stage in this pipeline, which terminates and,   */
/* thus, severs the primary input of DEAL, causing it to terminate   */
/* without dispatching any more processes (and causing end-of-file to*/
/* propagate back to the STARMSG, causing it to terminate).  Because */
/* the output of each of the IMMCMD stages passes through the GATE,  */
/* GATE's termination causes each IMMCMD stage to see end-of-file on */
/* its output, which causes it to terminate.  Thus, the single GATE  */
/* stage is used to terminate all of the never-ending stages in the  */
/* pipeline complex, except the TCPCLIENT stages, which stop when    */
/* the server breaks the connection after having received the QUIT   */
/* command.                                                          */
/*                                                                   */
/* When this pipeline starts up, it queries the virtual reader and   */
/* writes a record containing the spoolid of each file it finds in   */
/* the reader.  It then begins listening for the arrival of other    */
/* reader files, writing a record with the spoolid of each of those  */
/* as they arrive.                                                   */
/*                                                                   */
/* These spoolid records are fed into the FANIN, which feeds them    */
/* to the primary input of DEAL, to be dealt out to the client-SMTP  */
/* processes as they become ready for more work.                     */
/*                                                                   */
/* The primary input of each SMTP process is connected to an output  */
/* stream from DEAL, and the process number is the same as the       */
/* stream number.  The primary output of each client-SMTP process    */
/* is connected to the FANINANY, which feeds its output into the     */
/* secondary input of DEAL (through an ELASTIC stage, which may      */
/* buffer the stream number records until DEAL is ready for them).   */
/*                                                                   */
/* When a process is ready for work, it writes a record containing   */
/* its process/stream number to its primary output, which sends it   */
/* to FANINANY, which sends it to the secondary input of DEAL.       */
/*                                                                   */
/* DEAL with the SECONDARY option deals the records from its primary */
```

```
/* input to the output streams described by the records it receives  */
/* on its secondary input.  Thus, DEAL dispatches the processes in   */
/* the order in which they ask for work.                             */
/*-------------------------------------------------------------------*/

'PIPE (endchar ? listerr name LightSMTP)',
      'immcmd STOP |',                /* STOP was typed on console.  */
      'take 1 |',                     /* Terminate the IMMCMD.       */
      'spec /STOP/ 1 |',              /* Remember why we stopped.    */
   'o: fanout |',                     /* Tell each process to stop.  */
      'copy |',                       /* (Unblock FANOUT.)           */
   'g: gate',                         /* And don't dispatch more.    */
'?',
      'immcmd CP |',                  /* Immediate CP command.       */
   'g: |',                            /* GATE closed by STOP command.*/
      'cp',                           /* Pass command to CP.         */
'?',
      'immcmd DUMP |',                /* DUMP was typed on console.  */
   'g: |',                            /* GATE closed by STOP command.*/
      'pipdump |',                    /* Format a PIPDUMP.           */
      '>> pipdump listing d',         /* Append to existing PIPDUMP. */
'?',
      'immcmd JEREMY |',              /* JEREMY was typed on console.*/
   'g: |',                            /* GATE closed by STOP command.*/
      'jeremy |',                     /* Format pipeline status.     */
      'console',                      /* Display status.             */
'?',
      'starmsg CP SET IMSG IUCV |',   /* Listen for RDR arrivals.    */
      'find' arrival_msg'|',          /* Select file arrival message.*/
      'spec 26.4 1 |',                /* Isolate arriving spoolids.  */
   'f: fanin 1 0 |',                  /* Pick up spoolids in reader. */
   'g: |',                            /* (GATE closed by STOP cmd.)  */
   'd: deal secondary',               /* Hand out files on request.  */
'?',
   'y: faninany |',                   /* Stream of streamnumber recs.*/
      'elastic |',                    /* Hold until ready for them.  */
       processes,                     /* Feed to SMTPLITE processes. */
',
      'cp QUERY RDR' avail '|',       /* Maybe files in rdr already. */
      'drop 1 |',                     /* Discard response title.     */
      'spec word 2 1 |',              /* Isolate existing spoolids.  */
   'f:'                               /* Process these files too.    */


Error:

pipeRC = RC                           /* Remember how pipe ended.    */

'CP SET IMSG ON'                      /* Restore normalcy.           */
'CP SPOOL RDR NOHOLD'

Exit pipeRC                           /* Exit.                       */
```

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/*        PARMSMTP EXEC:  Set up the PARM. stem for SMTPLITE          */
/*                                                                    */
/* Copyright Princeton University, 1995, 1996.  Permission is         */
/* granted to SHARE, Inc., and its members to reproduce this work     */
/* for use as a coding example.  This work may not be executed in     */
/* production without permission of the Office of Computing and       */
/* Information Technology of Princeton University.                    */
/*                                                                    */
/*--------------------------------------------------------------------*/

Address Command
Signal On Novalue
Signal On Syntax


/*--------------------------------------------------------------------*
/*--------------------------------------------------------------------*/
/*                                                                    */
/*              Sample SMTPLite Local Parameter File                  */
/*                                                                    */
/*--------------------------------------------------------------------*/
processcount = 3                      /* Number of processes to run. */
timeout = 300                         /* Secs to wait for timeout.   */
delay = 5                             /* Secs to pause after timeout.*/
host = pucc.princeton.edu             /* SMTP server's host name.    */
port = 25                             /* Well-known port number.     */
tagto = PUCC SMTPB                    /* SMTP to handle tricky files.*/
tranto = RSCS                         /* RSCS to get to that SMTP.   */
ackrcpt = 1                           /* RCPTs to be acknowledged?   */
*--------------------------------------------------------------------*/


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

'PIPE (name ParmSet1)',               /* Set local parameters:       */
   '< smtplite parms |',              /* Read parameter file.        */
   'chop string =/*= |',              /* Chop off comments.          */
   'nfind *|',                        /* Discard comment records.    */
   '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 parameters from file.  */
```

```
/*--------------------------------------------------------------------*/
/*  Issue a CMS IDENTIFY command and a CP QUERY CPLEVEL command to    */
/*  get more items for the "PARM." stem.                              */
/*--------------------------------------------------------------------*/

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

'PIPE (endchar ? name ParmSet2)',
      'cp QUERY CPLEVEL |',           /* Query the CP level.         */
      'spec w1 1 write word -2 1 |',  /* Select interesting words.   */
   'f: find VM/SP|',                  /* Is it still VM/SP?          */
   'y: faninany |',                   /* Gather possible answers.    */
      'take 1 |',                     /* Need 1 or 0.                */
      'count lines |',                /* Make value Boolean.         */
      'var parm.isitsp',              /* And remember it.            */
'?',
   'f: |',                            /* Here if not VM/SP.          */
      'find 370|',                    /* Possibly VM/ESA 370 Feature?*/
   'y:'                               /* Yes, just as bad.           */


/*--------------------------------------------------------------------*/
/*  Require TCPIP DATA and RXSOCKET V2 for use by HOSTBYADDR.         */
/*--------------------------------------------------------------------*/

'ESTATE TCPIP DATA'                   /* Do we have the TCP/IP disk? */
If RC ¬= 0  Then Do                   /* No, this won't work.        */
   Say 'TCPIP DATA not found.'        /* Explain the problem.        */
   Exit RC                            /* Exit with error.            */
End

Parse Value Socket("Version")  With rc . ver .
If ver < 2  Then Do
   Say 'RXSOCKET Version 2 is required.'
   Exit -1
End


/*--------------------------------------------------------------------*/
/*  Determine fully-qualified domain name of our host system          */
/*--------------------------------------------------------------------*/

'PIPE (name GetHost)',                /* Determine "server_name":    */
   'hostid |',                        /* Ask TCPIP for host IP addr. */
   'hostbyaddr |',                    /* Resolve to FQDN.            */
   'append literal |',                /* In case can't resolve.      */
   'var parm.hostnode'                /* Set "hostnode" parameter.   */

If parm.hostnode == ''  Then Do       /* Unable to resolve address?  */
   Say "Unable to determine local host's name."
   Exit RC                            /* Yes, go no further.         */
End
```

```
/*--------------------------------------------------------------------*/
/*  Load the "PARM." stem 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 as VARLOADable, too.    */


Exit                                    /* Return with parms set.      */


/*--------------------------------------------------------------------*/
/*                            Error Subroutines                       */
/*--------------------------------------------------------------------*/

Novalue:

Say 'Novalue condition for symbol' Condition('D') 'on line' ,
   Sigl':' Sourceline(Sigl)

Address Command 'EXEC TELL MAINT AT PUCC Novalue Error'/* Phone home.*/

Exit 20                                 /* Exit with error setting.    */


Syntax:

Say 'Syntax error on line' Sigl':' Sourceline(Sigl)
Say 'Error was:' Errortext(RC)

Address Command 'EXEC TELL MAINT AT PUCC Syntax error'/* Phone home. */

Exit 24                                 /* Exit with error setting.    */
```

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/*               SMTPLITE REXX:  A Client-SMTP Process                */
/*                                                                    */
/* Copyright Princeton University, 1995, 1996.  Permission is         */
/* granted to SHARE, Inc., and its members to reproduce this work     */
/* for use as a coding example.  This work may not be executed in     */
/* production without permission of the Office of Computing and       */
/* Information Technology of Princeton University.                    */
/*                                                                    */
/*--------------------------------------------------------------------*/

Signal On Error
Signal On Novalue
Signal On Syntax

| 'EOFREPORT ALL'                          /* Propagate end-of-file.      */

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

Parse Arg seqno .                        /* My process number.          */

crlf = '0D25'x                           /* Handy constant.             */


/*--------------------------------------------------------------------*/
/* This REXX filter is a single SMTP process.  It asks for a file to  */
/* work on by writing a record containing its process number to its   */
/* primary output.  When originally invoked, it first does a short    */
/* wait based on its process number (to stagger the startups of the   */
/* processes).  It then writes its process number to its primary      */
/* output in order to get the first file to work on.  Later, as it    */
/* completes the processing of each file or completes the error       */
/* handling procedure in the case of bad files, it again writes its   */
/* process number to its output.  (The process number corresponds     */
/* to the number of the output stream from the DEAL stage in the      */
/* calling pipeline that is connected to the primary input stream     */
/* of the main subroutine pipeline in this stage.)                    */
/*--------------------------------------------------------------------*/

'OUTPUT' seqno                           /* Queue for a file.           */

'CALLPIPE literal +' || (seqno-1)*2 '| delay' /* Be gentle w/server. */


Do Forever                               /* Do until stopped.           */

   /*-----------------------------------------------------------------*/
   /* This subroutine pipeline processes mail files in a continuous   */
   /* stream until it is terminated by a badly formed file, an error  */
   /* from the SMTP server, a TCP/IP error, or a console command.     */
```

```
/*                                                                  */
/* Its input record contains the spool file id of the next file     */
/* to be processed.  The spoolid is stored in the REXX variable     */
/* "spoolid" and the spoolid record is passed to GETBSMTP, which    */
/* reads the file, verifies that it is well-formed BSMTP, and       */
/* sends it back reformatted as two records, one containing all     */
/* the lines of the BSMTP envelope, the other containing the        */
/* mail text.                                                       */
/*                                                                  */
/* The envelope is split into individual SMTP commands and fed      */
/* into CALLSMTP to be sent to the server.  CALLSMTP writes the     */
/* responses from the server to its output stream for analysis.     */
/*                                                                  */
/* The normal sequence of responses from SMTP will lead finally     */
/* to a 354 prompt (authorizing the mail text to be transmitted).   */
/* When the 354 is received, it is consumed, causing the record     */
/* on the primary output of GETBSMTP to be consumed, which allows   */
/* GETBSMTP to write the mail text record to its secondary output,  */
/* which is also routed into CALLSMTP (as a single record).  The    */
/* normal response to this is a 250 (indicating that the mail has   */
/* been delivered).  When the 250 is received, it is consumed,      */
/* which allows the subroutine pipeline in GETBSMTP to terminate.   */
/* GETBSMTP purges the spool file and writes a null record to its   */
/* tertiary output.  That record is fed to VAR, to clear the        */
/* REXX variable "spoolid".  It is then converted into a record     */
/* containing the process number and fed into the calling           */
/* pipeline to request another file to process.                     */
/*                                                                  */
/* There are a number of error cases, but they are treated much     */
/* the same, because the RFC requires that the client issue an      */
/* explicit QUIT command before terminating.  The errors that       */
/* are detected include:                                            */
/* 1.  GETBSMTP found badly-formed BSMTP.                           */
/* 2.  The SMTP server issued a transient error ("4yz").            */
/* 3.  The SMTP server issued a permanent error ("5yz").            */
/* 4.  A STOP immediate command was typed on the console.           */
/*                                                                  */
/* In each case, the error results in a QUIT record being fed       */
/* into CALLSMTP, which sends it to the server, resulting in a      */
/* 221 (connection closing) response.                               */
/*                                                                  */
/* The other condition in which the connection may be closed        */
/* (which does not result in a 221 message) is when TCPIP issues    */
/* an ERRNO.                                                        */
/*                                                                  */
/* Any ERRNO records or 221 records are fed into the primary        */
/* input of a pair of GATE stages that have their tentacles         */
/* spread throughout this subroutine pipeline.  Thus, when the      */
/* GATEs see a record that indicates that the connection to the     */
/* server no longer exists, they pull plugs here and there (at      */
/* input and output points) to collapse this subroutine pipeline    */
/* gracefully.                                                      */
/*------------------------------------------------------------------*/
```

```
   errno = ''                          /* Initialize for this pass.   */
   spoolid = ''
   smtperr = ''

   pipeRC = 0                          /* Initialize.                 */
   'PEEKTO'                            /* Exit if input severed.      */

   Signal Off Error

   'CALLPIPE (endchar ? listerr name SMTPLite'seqno')',
      'F: faninany |',                 /* Here if STOP/221, 4nn/221,..*/
|     'FO: fanout |',
         'copy |',                     /* ...5nn/221, TCPIP errno,... */
      'G: gate strict',                /* ...or unparseable file/221. */
|  '?',
|     'FO: |',                         /* Here if STOP/221, 4nn/221,..*/
|        'copy |',                     /* ...5nn/221, TCPIP errno,... */
|     'GG: gate strict',               /* ...or unparseable file/221. */
   '?',
         '*.input.0: |',               /* Get the next spool file id. */
      'G: |',                          /* (For shutting off input.)   */
         'copy |',                     /* Unblock caller's DEAL.      */
         'var spoolid tracking |',     /* Remember file we're doing.  */
      'g: getbsmtp' seqno '|',         /* Get file's BSMTP commands.  */
         'split after string x0D25 |',/* One command per record.     */
      'y: faninany |',                 /* Feed inputs to CALLSMTP.    */
      'c: callsmtp' seqno '|',         /* Communicate w/SMTP server.  */
      't: tolabel 221|',               /* Divert connection-closing.  */
      'n: nfind 4|',                   /* Divert SMTP transient error.*/
      'N: nfind 5|',                   /* Divert SMTP permanent error.*/
         'between /354/ /250/ |',      /* Before and after mail sent. */
      'b: find 354|',                  /* SMTP is ready to read mail. */
         'copy |',                     /* Unblock GETBSMTP's primary. */
         'hole',                       /* Keep COPY running.          */
   '?',
      'g: |',                          /* Mail data from GETBSMTP.    */
      'y:',                            /* Read it into CALLSMTP.      */
   '?',
      'b: |',                          /* Here while mail being sent. */
         'find 250|',                  /* SMTP has delivered the mail.*/
         'copy |',                     /* Unblock GETBSMTP's 2ndry.   */
         'hole',                       /* Keep COPY running.          */
   '?',
      'g: |',                          /* Null here when file purged. */
         'eofback var spoolid tracking |',/* Remember no longer have.*/
         'spec /'seqno'/ 1 |',         /* Record with my streamnumber.*/
|     'GG: |',                         /* (For shutting off output.)  */
         '*.output.0:',                /* Ask for another spool file. */
   '?',
      'c: |',                          /* Here if TCP errno/timeout.  */
         'var errno tracking |',       /* Save the TCPIP errno.       */
      'F:',                            /* Can now stop this pipeline. */
   '?',
```

```
   't: |',                          /* Here if connection closed.  */
   'F:',                            /* Can now stop this pipeline. */
'?',
      '*.input.1: |',               /* STOP was typed on console.  */
   'G: |',                          /* (For shutting off input.)   */
   'Y: faninany |',                 /* Any requests to close link. */
      'take 1 |',                   /* Don't send multiple QUITs.  */
      'copy |',                     /* Unblock error producer.     */
      'spec word 1 1 |',            /* Isolate termination reason. */
      'var smtperr tracking |',     /* Store reason for analysis.  */
      'spec /QUIT'crlf'/ 1 |',      /* Build SMTP QUIT command.    */
   'y:',                            /* Close the SMTP connection.  */
'?',
   'g: |',                          /* GETBSMTP couldn't parse file*/
   'Y:',                            /* Remember error; close link. */
'?',
   'n: |',                          /* Here if transient SMTP err. */
   'Y:',                            /* Remember error; close link. */
'?',
   'N: |',                          /* Here if permanent SMTP err. */
   'Y:'                             /* Remember error; close link. */

pipeRC = RC                         /* Remember how pipe ended.    */

Signal On Error

/*-------------------------------------------------------------------*/
/* The subroutine pipeline has terminated, so something is wrong.    */
/* First log the status.  Then check for the TCPIP server being      */
/* down and exit if it is.                                           */
/*-------------------------------------------------------------------*/

Say 'seqno='seqno'; errno='errno'; spoolid='spoolid';',
   'smtperr='smtperr'; pipeRC='pipeRC'.'       /* Log to console. */

If pipeRC = 314  Then Exit pipeRC  /* Can't talk to TCP/IP.        */

/*-------------------------------------------------------------------*/
/* That wasn't the problem, so decide what to do with the file       */
/* we were processing, if we still have it in the reader.  If        */
/* the error was such that we are unlikely ever to be able to        */
/* process the file, transfer it to the full-function SMTP.  If      */
/* the error is transient (and this is VM/ESA), transfer the file    */
/* to this virtual machine to be retried after other queued files.   */
/*-------------------------------------------------------------------*/

If spoolid ¬== '' & smtperr ¬== 'STOP' & Left(errno,4) ¬== '5001'
Then Do                             /* Spool file not purged?      */

   Signal Off Error

   If Left(smtperr,1) == '5' |,     /* Permanent SMTP error?       */
      pipeRC ¬= 0 |,                /* Pipe stage reported error?  */
```

```
         parm.isitsp |,               /* VM/SP or ESA 370 Feature?   */
         Left(smtperr,7) == 'BADFILE' /* Badly-formed file?          */
      Then Do
         Address Command 'CP TAG FILE' spoolid parm.tagto
         Address Command 'CP TRANSFER' spoolid 'TO' parm.tranto
      End

      Else                            /* Signal caller about file... */
         Address Command 'CP TRANSFER' spoolid 'TO *' /* ...to retry.*/

      Signal On Error

   End

   /*-----------------------------------------------------------------*/
   /* Once the spool file has been taken care of, decide whether to  */
   /* exit or start the subroutine pipeline up again.                */
   /*-----------------------------------------------------------------*/

   Select

      When smtperr == 'STOP'          /* Requested to stop?          */
         Then Exit 8                  /* Yes, exit.                  */

      When errno == '5001 EpipeStopped'  /* PIPMOD STOP done?        */
         Then Exit 8                  /* Yes, exit.                  */

      When Wordpos(pipeRC, '0 12 72 73 510') = 0 /* Problem in pipe? */
         Then Leave                   /* Yes, don't loop.            */

      When errno ¬== ''  Then Do      /* TCP/IP encountered error?   */
         Parse Var errno . errtext    /* Yes, parse error text.      */
         Say 'TCP/IP returned error:' errtext'.'      /* Display it. */
      End

      Otherwise Nop

   End

   /*-----------------------------------------------------------------*/
   /* If the decision is to keep running, first allow the server a   */
   /* moment to compose itself; then ask our caller for another file.*/
   /*-----------------------------------------------------------------*/

   'CALLPIPE literal +'parm.delay '| delay'  /* Give server a break. */

   'OUTPUT' seqno                  /* Queue for a BSMTP file.     */

End

Error: Exit pipeRC*(pipeRC<>12)       /* RC = 0 if normal end.       */
```

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/*    GETBSMTP REXX:  Put next BSMTP file into pipe in two pieces     */
/*                                                                    */
/* Copyright Princeton University, 1995, 1996.  Permission is         */
/* granted to SHARE, Inc., and its members to reproduce this work     */
/* for use as a coding example.  This work may not be executed in     */
/* production without permission of the Office of Computing and       */
/* Information Technology of Princeton University.                    */
/*                                                                    */
/*--------------------------------------------------------------------*/

Signal On Error; Signal On Novalue; Signal On Syntax

Parse Upper Arg seqno .                /* My process number.          */

crlf = '0D25'x                         /* Handy constant.             */
delivered = 0                          /* Initialize.                 */

'EOFREPORT ALL'                        /* Propagate end-of-file.      */

'SELECT OUTPUT 2'                     /* Default to caller's tertiary. */

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

   /*-----------------------------------------------------------------*/
   /* Wait for the next input record containing a spool file id.      */
   /* Exit if the input stream is severed.                            */
   /*-----------------------------------------------------------------*/

   'PEEKTO spoolid'                    /* Wait for next spoolid.      */

   /*-----------------------------------------------------------------*/
   /* Issue a sipping subroutine pipeline to process one spool file. */
   /*                                                                 */
   /* The file is read from the virtual reader and deblocked from     */
   /* NETDATA format.  (If it is not in NETDATA format, the DEBLOCK   */
   /* stage will fail with return code 72 or 73.)                     */
   /*                                                                 */
   /* Test the file for being well-formed BSMTP:                      */
   /* 1.  Nothing before the HELO line.                               */
   /* 2.  Only one HELO line.                                         */
   /* 3.  Only one DATA line.                                         */
   /* 4.  Only one end-of-data indicator.                             */
   /*                                                                 */
   /* If the file is not well-formed, a record is written to the      */
   /* primary input of the GATE stage, which terminates and severs    */
   /* its streams, causing the entire subroutine pipeline to          */
   /* terminate.                                                      */
   /*                                                                 */
   /* If the file is well-formed, it is blocked into two records:     */
   /* 1.  The BSMTP envelope with all records joined by CRLF and a    */
   /*     trailing CRLF.                                              */
   /* 2.  The mail text with all records joined by CRLF and the       */
   /*     trailing QUIT command removed.                              */
   /*                                                                 */
   /* When both output records are available to SYNCHRONISE, the      */
```

```
/* envelope record is written to this stage's primary output.     */
/* Once that has been consumed (which is not until the server     */
/* has sent the 354 prompt asking for the mail text), the         */
/* mail text record is written to this stage's secondary output.  */
/* That is not consumed until the server has sent the 250         */
/* response indicating that the mail has been delivered.          */
/*                                                                */
/* If the second output record is consumed, COUNT LINES will set  */
/* a count of 1, causing the value of the variable "delivered"    */
/* to be set to "true".  If the second output record is not       */
/* consumed, because the mail was not delivered, the count will   */
/* be 0 and the variable "delivered" will be set to "false".      */
/*----------------------------------------------------------------*/

Signal Off Error

'CALLPIPE (endchar ? listerr name GetBSMTP'seqno')',
   'f: faninany |',                 /* Gather any bad records.     */
   'g: gate strict',                /* Slam the gate if any bad.   */
'?',
      'reader file' spoolid '|',    /* Get next file from reader.  */
   'c: change 1.1 x41 // |',        /* Select data; remove opcode. */
      'pad 80 |',                   /* Pad for DEBLOCK.            */
      'deblock netdata |',          /* Unravel DMSDDL damage.      */
   'd: change 1.1 /{// |',          /* Get data; remove ctl char.  */
      'strip trailing |',           /* Discard trailing blanks.    */
   't: tolabel HELO|',              /* Anything before HELO?       */
   'f:',                            /* Yes, file bad; close GATE.  */
'?',
   'c:',                            /* Discard tags, etc.          */
'?',
   'd:',                            /* Discard NETDATA headers.    */
'?',
   't: |',                          /* Entire BSMTP file to here.  */
   'b: between /HELO/ /DATA/ |',    /* Select BSMTP commands.      */
      'drop 1 |',                   /* Discard the HELO command.   */
   'n: nfind HELO|',                /* Error if there are two sets.*/
      'join * x0D25 |',             /* Join into single record.    */
   'e: change -4;-1 /DATA/DATA'crlf'/ |', /* Bad if DATA not end. */
   's: synchronise |',              /* Need both commands and mail.*/
   'g: |',                          /* Pass through GATE.          */
      '*.output.0:',                /* BSMTP commands to caller.   */
'?',
   'b: |',                          /* Mail data come to here.     */
      'join * x0D25 |',             /* Join into a single record.  */
   'q: change -9;-1 /'crlf'.'crlf'QUIT/'crlf'.'crlf'/ |',/* -QUIT.*/
   'p: nlocate 1;-6 /'crlf'.'crlf'/ |',    /* Two eod indicators? */
   's: |',                          /* Need both commands and mail.*/
   'g: |',                          /* Pass through GATE.          */
   'l: count lines |',              /* Remember whether sent mail. */
      '*.output.1:',                /* Mail data to caller.        */
'?',
   'e: |',                          /* Here if bad mail headers.   */
   'f:',                            /* Stop the pipeline.          */
'?',
   'n: |',                          /* Here if double mail headers.*/
   'f:',                            /* Stop the pipeline.          */
```

```
   '?',
      'q: |',                          /* Here if bad mail trailers.  */
      'f:',                            /* Stop the pipeline.          */
   '?',
      'p: |',                          /* Here if two eod indicators. */
      'f:',                            /* Stop the pipeline.          */
   '?',
      'l: |',                          /* Line count to here.         */
         'var delivered tracking'      /* Remember whether delivered. */

   pipeRC = RC                         /* Remember how pipe ended.    */

   Signal On Error

   /*-----------------------------------------------------------------*/
   /* If the mail file was not delivered, write a record to our       */
   /* quaternary output stream to cause an SMTP QUIT command to be    */
   /* sent to the server and then exit.  (Once the connection to      */
   /* the server has been taken down, the calling pipeline will       */
   /* terminate, so that the problem can be analyzed.)                */
   /*-----------------------------------------------------------------*/

   If ¬ delivered | pipeRC ¬= 0        /* Did we send the file?        */
   Then Do                             /* No, need to close session.  */
      'SELECT OUTPUT 3'                /* Cause QUIT to go to SMTP.   */
      'OUTPUT' 'BADFILE'spoolid        /* Reason for quitting.        */

      If pipeRC = 72 | pipeRC = 73     /* Failed because not NETDATA? */
         Then Say 'Input file' spoolid 'not in NETDATA format.'

      Exit pipeRC*(pipeRC<>12)         /* Return to caller.           */
   End

   /*-----------------------------------------------------------------*/
   /* If the mail file was delivered, PURGE the spool file, write     */
   /* a null record to our tertiary output (to show that the file     */
   /* has been purged), and consume the input record.                 */
   /*-----------------------------------------------------------------*/

   Else Do                             /* Mail has been delivered.    */
      Signal Off Error
      'CALLPIPE cp PURGE RDR' spoolid /* Purge the spool file.        */
      Signal On Error
      'OUTPUT'                         /* Queue for another file.     */
      'READTO'                         /* Unblock the caller.         */
   End

End

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

```
/*--------------------------------------------------------------------*/
/*                                                                    */
/*     CALLSMTP REXX:  Low-level routine to communicate with SMTP     */
/*                                                                    */
/* Copyright Princeton University, 1995, 1996.  Permission is         */
/* granted to SHARE, Inc., and its members to reproduce this work     */
/* for use as a coding example.  This work may not be executed in     */
/* production without permission of the Office of Computing and       */
/* Information Technology of Princeton University.                    */
/*                                                                    */
/*--------------------------------------------------------------------*/

Signal On Novalue
Signal On Syntax

Parse Arg seqno .                       /* My process number.         */

crlf = '0D25'x                          /* Handy constant.            */

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


/*--------------------------------------------------------------------*/
/* Determine the SMTP command(s) to be issued when connecting to      */
/* the server SMTP by testing the ACKRCPT parameter.  If the user     */
/* specified that RCPTs are not to be acknowledged, connect to the    */
/* server, test whether it is accepting the private ACKRCPT OFF       */
/* extension to SMTP, and close the connection.  If the extension     */
/* is supported, build the pipeline stages needed to specify the      */
/* extension and to block all RCPT commands and the subsequent DATA   */
/* command into a single input record for TCPCLIENT, so that it will  */
/* expect a single response to that set of commands.                  */
/*--------------------------------------------------------------------*/

startup =,
   'literal HELO' parm.hostnode || crlf'|'  /* First cmd of session. */

If ¬ parm.ackrcpt  Then Do          /* Don't want RCPTs acknowledged? */

   ackrcpt_avail = 0                    /* Default in case timeout.   */

   'CALLPIPE (listerr name CheckExt'seqno')',
      'literal EHLO' parm.hostnode || crlf'|',/* Support extensions? */
      'append literal QUIT'crlf'|',
      'xlate 1-* from 1047 to 819 |', /* Translate input to ASCII.  */
      'tcpclient' parm.host parm.port,/* Send cmds to SMTP server.  */
         'timeout' parm.timeout,      /* Exit if no reply.          */
         'oneresponse',               /* Synchronize cmds/replies.  */
|        'deblock string x0D0A',      /* A reply ends with CRLF.    */
         'greeting',                  /* Don't count connected msg. */
|        'group /joincont column 4.1 x2D/ |',/* Group continuations. */
```

```
      'xlate 1-* from 819 to 1047 |', /* Translate replies to EBCDIC.*/
      'split before string /250/ |',  /* Deblock continued lines.    */
      'find 250 XLSMTP-ACKRCPT|',     /* Look for the one we need.   */
      'take 1 |',                     /* Make it Boolean.            */
      'count lines |',                /* Value will be 0 or 1.       */
      'var ackrcpt_avail'             /* Remember which.             */

   If ackrcpt_avail                   /* ACKRCPT OFF supported.      */
      Then startup =,                 /* So change startup commands. */
         'literal XLSMTP-ACKRCPT OFF'crlf'|',
         'literal EHLO' parm.hostnode || crlf'|',
|        'joincont column 1.4 string /RCPT/ |'   /* For ONERESPONSE. */

End


/*--------------------------------------------------------------------*/
/* Pass SMTP commands and mail to the SMTP server, waiting for a      */
/* single response from the server for each input record sent to      */
/* the server.  Response lines are delimited by CRLF.  A single       */
/* response may extend across multiple lines, in which case all but   */
/* the last line contain a hyphen in the fourth column.  When the     */
/* connection is made, a "greeting" is received from the server.      */
/* A timeout should occur if the greeting message is not received,    */
/* but the greeting message should not be counted as a response to    */
/* the first command.                                                 */
/*--------------------------------------------------------------------*/

'CALLPIPE (endchar ? listerr name CallSMTP'seqno')',
      '*.input.0: |',                 /* Input for SMTP from caller. */
       startup,                       /* First commands of session.  */
      'xlate 1-* from 1047 to 819 |', /* Translate input to ASCII.   */
   't: tcpclient' parm.host parm.port,/* Send cmds to SMTP server.   */
         'timeout' parm.timeout,      /* Exit if no reply.           */
         'oneresponse',               /* Synchronize cmds/replies.   */
|        'deblock string x0D0A',      /* A reply ends with CRLF.     */
         'greeting',                  /* Don't count connected msg.  */
|        'group /nlocate 4.1 x2D/ |', /* Don't count if continued.   */
      'xlate 1-* from 819 to 1047 |', /* Translate replies to EBCDIC.*/
      '*.output.0:',                  /* Output from SMTP to caller. */
'?',
   't: |',                            /* TCPCLIENT's secondary output*/
      '*.output.1:'                   /* Return ERRNO to caller.     */

pipeRC = RC                           /* Return code from CALLPIPE.  */

If pipeRC ¬= 0  Then Do               /* If not 0, no ERRNO written. */
   'SELECT OUTPUT 1'                  /* So fake an ERRNO record.    */
   'OUTPUT' 9000+pipeRC 'EpipeFailed'
   Exit pipeRC                        /* And exit with error.        */
End

Else Exit 0                           /* RC = 0 if normal end.       */
```

Appendix B

CONVERTING WEBSHARE TO THE PIPELINES BUILT-IN TCP SUPPORT

Without Arty Ecock's RXSOCKET and the wonderful Internet tools that Rick Troth has built over the years using RXSOCKET, my VM systems would no longer be viable. There is no way we can ever thank either of these great guys for their splendid contributions to the VM community.

Rick's Webshare[3] Web server for VM has been particularly critical to us at Princeton over the past few years. We have a number of very important, very visible applications running on Webshare. We chose VM for these applications primarily because Webshare CGI routines are so easy to develop, as they are written in REXX as pipeline stages.

Rick designed Webshare in a very layered way. The highest layer is HTTPD EXEC, which analyzes parameters and sets up the general environment for the Web server. HTTPD EXEC invokes Rick's generalized TCP interface, TCPSHELL EXEC. TCPSHELL uses RXSOCKET to make all the necessary socket calls to set up for TCP communications with clients. Then it loops listening on its assigned port for client connection requests. Each time a new client connects, TCPSHELL executes a pipeline of the form:

```
pipe pipesock read | server | pipesock write
```

PIPESOCK REXX is Rick's low-level pipeline stage to move data between a socket and a pipeline. It, too, uses RXSOCKET. In the Webshare case, the server stage is HTTPD REXX:

```
pipe pipesock read | httpd | pipesock write
```

When a Web client sends packets across the network, they are captured by pipesock read and fed into the httpd stage. httpd acts on the client requests and sends its replies to the client through pipesock write. HTTPD REXX contains all of the logic for the Web server; TCPSHELL EXEC and PIPESOCK REXX are general interfaces that can be used in other servers and have no Webshare-specific function.

Webshare handles only one client at a time. However, one can get the effect of a multi-tasking Web server by letting the TCPIP service machine allocate Web clients among multiple Webshare virtual machines listening on the same port. In general, this works quite well, and it is trivial to set up. (To make a pipelined server do this, one specifies the reuseaddr and backlog 0 options on tcplisten and lists all of the servers with the same port in the PORT statement in PROFILE TCPIP.)

---

[3] Beyond Software Incorporated has exclusive rights to distribute, publish or sell derivative works based upon Webshare. The author of this paper presents information about Webshare for academic and educational purposes only. Actually making the modifications to Webshare discussed in this paper and distribution, publication or sale of such modifications is prohibited without the approval of Beyond Software. To request approval contact: Beyond Software Incorporated, 1040 East Brokaw Road, San Jose, CA 95131; phone 408-436-5900; fax 408-436-5915.

The TCPIP service machine uses a very simple strategy for dispatching clients among multiple servers listening on the same port—it tries always to use the virtual machine at the top of its list. If that one has a client attached, it tries the next one, and so on. The only problem with this scheme is that TCPIP may assign a new client to a server virtual machine before the server is ready for another client. TCPIP checks only whether there is a client connected; it has no way of detecting that the server may still be busy doing something for its previous client.

We had a serious problem because of this scheme, as we have a very long-running CGI that mounts tapes and loads files from them (and optionally ZIPs those files). This CGI checks after each step in the process to make sure the client is still connected. If the client has gone away, the CGI terminates immediately. However, if a client disconnects early in one of the long-running steps and another client connects shortly after that, TCPIP will assign the new client to the busy server, with the result that our Web service was sporadically providing quite long service times.

My attempts to redesign the CGI to get around this problem got messier and messier, and I feared that we were likely to develop other CGIs with similar problems, so I decided instead to modify Webshare to support multiple simultaneous clients. This turned out to be very easy to do using the TCP support in *CMS Pipelines*.

One of the reasons it was so easy to convert Webshare to use the *Pipes* TCP support was that Rick planned it that way. When he wrote PIPESOCK REXX, he intended that it would someday be replaced by a built-in stage. Indeed, at the beginning of PIPESOCK, there is the comment, "until the day when there's a true asynch socket stage". Inevitably, the interface that Rick imagined was not exactly the one that the Piper ultimately provided, but they were so similar that the conversion was essentially trivial.[4] And, of course, once one has a server using the *Pipes* built-in TCP support, making it support multiple simultaneous clients is also trivial. Our modified Webshare now frequently has four processes running simultaneously when it is simply serving out files. When our long-running CGI is loading a file from a tape (using a *Pipes* tape stage), the other processes continue to serve out files or run CGIs, and there is only a very slight degradation in their responsiveness.

---

[4] I found only two differences in the interface when converting from the binary mode of pipesock to the *Pipes* built-in stages:

1. pipesock read expected that the last packet it wrote for a transaction would not be consumed; that is, it terminated only as the result of seeing return code 12 on an OUTPUT command when the server stage terminated without having consumed its last input record. (This was probably not a desirable situation. It could have been avoided if the loop in pipesock read had contained a SUSPEND command followed by a STREAMSTATE OUTPUT 0 command to test for the output stream having been severed. However, SUSPEND was not added to *Pipes* until version 1.1.7, so it was not available to Rick to use in Webshare, given his wish to make Webshare work on systems still running the *CMS Pipelines* PRPQ (level 1.1.5).) The tcpdata stage that replaced pipesock read and pipesock write required that all input from the client be consumed, lest tcpdata become blocked and unable to accept data to send to the client. A very small change was made to HTTPD REXX to make it consume all input.

2. For timing reasons (to allow the server to send a prompt to the client as soon as the connection was established), pipesock read in binary mode wrote a null record to its output as soon as it started. This was not necessary when using tcpdata, so the PEEKTO in HTTPD REXX that had waited for that null input record was removed.

In addition to making Webshare multi-tasking, I wanted also to make it faster, if possible, because that, too, would improve service for the end users. I was optimistic that I could make Webshare faster because I had a considerable advantage over Rick in the performance area. In writing Webshare, he restricted himself to using the 1990 version of *Pipes* (contained in level 1.1.5 of the *CMS Pipelines* PRPQ). By doing that, he assured that as many sites as possible would be able to run Webshare. However, today anyone can get a very current version of *Pipes* from the *CMS Pipelines* Runtime Library Distribution, so that constraint no longer exists. The current version of *Pipes* has twice as much function as the 1990 version, so much more of the work can be done "in the pipeline", rather than in REXX, with the expected performance benefits. Using modern *Pipes*, I have been able to cut the CPU utilization of my Webshare test case in half and its elapsed time by a third.

The conversion process

I started the conversion process by setting up a simple test case against which to measure my efforts. This consisted of loading the main Web page from our VM system (one HTML file and seven images) four times. A private port was allocated for testing, and only one server machine was used. All measurements were taken during periods of very low activity on the processor (a 3090-600E) and the network. All files were on minidisks (rather than in the Shared File System). The VERBOSE and IDENT options of Webshare were both on. The system was running VM/ESA 1.0 with CMS 7 and the field-test version of *CMS Pipelines* (level 1.1.10).

Server resource utilization was measured using Pat Ryall's TIMES program. Measurements were done accessing the server with two different clients, OS/2 WebExplorer 1.1b and Rob van der Heij's Lurky (see Appendix D). (The server had quite different performance characteristics in handling these two clients.) Lurky was also used to measure the elapsed time to complete the four loads of the page and its images. (I was not able to devise a scheme for getting accurate elapsed time measurements with WebExplorer.)

The conversion was accomplished in six phases:

1. Replacing RXSOCKET with the built-in TCP stages;
2. Replacing MAKETEXT REXX with a "pure-*Pipes*" solution;
3. Replacing GLOBALV with stemmed arrays;
4. Improving the multi-tasking;
5. Replacing other chunks of REXX with pipelines; and
6. Putting back the blocking.

Each of these phases will be discussed below. The current modified version of TCPSHELL EXEC and a new filter called TCPSHELL REXX can be found at the end of this appendix.

Phase 0. Mostly unmodified Webshare 1.2.3: With my test case, our mostly unmodified Webshare used 22 seconds of CPU time and issued about 2,600 pipeline specifications when the client was WebExplorer, but was much less expensive when the client was Lurky:

Client	TOTCPU	Elapsed	SVCs	I/Os	Pipes
WebEx	21.8	---	33799	1247	~2575
Lurky	13.1	20	26087	1247	~1350

I was unable at first to understand the difference in the cost of serving the same files to two different clients. Ultimately, it became clear that the bulk of the difference was due to the fact that WebExplorer (as I have it configured) sends 19 lines of optional headers in each request (*e.g.*, "Accept:"), while Lurky sends none. A typical pair of timing runs showed 651 input lines from WebExplorer and only 51 from Lurky. The CPU time attributed to the stage that reblocked and translated the input lines (maketext local) was 7.1 seconds in the WebExplorer case *vs.* 0.5 seconds in the Lurky case. When one factors in the *Pipes* overhead for running that many more pipelines, essentially all of the difference between the two cases is explained.

Phase 1. Replacing RXSOCKET with the built-in TCP stages: The first phase of the modifications to Webshare consisted of five changes:

1. TCPSHELL EXEC was modified to replace pipesock and the other RXSOCKET calls with *Pipes* TCP built-in stages, but otherwise the flow of data between the client and HTTPD REXX was unchanged.

2. TCPSHELL EXEC was modified to establish a pipeline with permanent processes, rather than issuing the main pipeline once per client connection request.

3. TCPSHELL EXEC was split into TCPSHELL EXEC and TCPSHELL REXX. The REXX filter contains the process-specific portion of the function that had been in TCPSHELL EXEC. The main pipeline issued by TCPSHELL EXEC includes one copy of the tcpshell stage for each process. The number of processes is a runtime parameter; I used four.

4. The use of GLOBALV to store processing variables was modified to account for the multiple processes. Since there could now be more than one active client, there had to be a way of storing more than one remote host name, more than one client userid, *etc.* The server-wide global variables continued to be stored in a variable pool named for the server, and a separate pool was established for each process to contain the client-specific variables.

5. The seven pipeline specifications in WEBSRVDC REXX, which decodes URL-encoded data, were replaced by a single pipeline.

With these changes in place, we had a Web server that could handle multiple simultaneous clients, so the original problem of blocking by long-running CGIs was solved. Otherwise, however, performance was little changed, which was not surprising, as the data flow was little changed. CPU utilization, elapsed time, and the SVC count were all down slightly:

Client	TOTCPU	Elapsed	SVCs	I/Os	Pipes
WebEx	20.8	---	30856	1247	2386
Lurky	12.0	19	23059	1209	1170

Phase 2. Replacing MAKETEXT REXX with a "pure-Pipes" solution: The next step was to analyze the Web server with Rita to determine areas that might benefit from use of new *Pipelines* function. One area was conspicuous, the maketext filter.

maketext network appended CRLF to output records and translated them from EBCDIC to ASCII on their way from the server to the client. maketext network was used relatively lightly in my test case. With a current version of *Pipes*, it could be replaced by two built-in stages. With either client, this saved half a second of CPU and reduced the number of pipeline specifications issued by 36.

maketext local, which did most of its processing in REXX, reblocked packets received from the client into the original records by flowing the packets together and breaking them at linefeeds. While doing that, it also translated the data from ASCII to EBCDIC. maketext local was much more heavily used than maketext network; it was invoked once per input record. Each invocation resulted in the issuing of one CALLPIPE and one ADDPIPE. So, the strategy for replacing maketext local was not simply to replace it with a "pure *Pipes*" solution, but also to implement the function in such a way that all input data flowed through one pipeline, rather than having the two pipeline specifications issued for each input record. Reducing the number of pipeline specifications issued reduces the *Pipelines* overhead, of course, but, more importantly, keeping the data in the pipeline, rather than moving them between *Pipes* and REXX, avoids the high cost of building and breaking down REXX variables.

Using modern plumbing, it was possible to do all the reblocking and translating of incoming data in a continuous fashion by prefixing this ADDPIPE to the input stream of HTTPD REXX:

```
'ADDPIPE (endchar ? listerr name HTTPD20)',
      '*.input.0: |',             /* Pre-process filter's input. */
      'xlate from 819 to 1047 |', /* From ASCII to EBCDIC.       */
      'joincont trailing x0D keep |', /* So can spot null lines. */
      'split after x25 |',        /* Break after line feeds.     */
   's: strtolabel x0D25 |',       /* Select up to the null line. */
   'f: fanin |',                  /* And the null line itself.   */
      'deblock CRLF |',           /* Deblock those normally.     */
   'i: fanin |',                  /* Pull in POST packets, too.  */
      '*.input.0:',               /* Now feed into this filter.  */
'?',
   's: |',                        /* What we really wanted...    */
   't: take 1 |',                 /* ...at label s: above was... */
   'f:',                          /* ..."strafterlabel x0D25".   */
'?',
   't: |',                        /* Packets after null line...  */
   'i:'                           /* ...are not to be deblocked. */
```

The input to the httpd stage is records from the browser spread randomly across packets. First comes a set of records that are delimited by carriage-return and line-feed. Then comes a null record (only carriage-return and line-feed). That may be followed by data from a POSTed Web form, which may or may not be terminated by a CRLF sequence.

What made the reblocking of the input to httpd difficult to achieve with an early version of *Pipes* was the fact that the data from POSTed forms (the portion that comes after the null input record) must be handled differently from what comes before. If there were no forms data, deblocking this input stream would be very easy. One would use deblock CRLF and stop reading when one got to the null line. Because there may be forms data, however, one cannot apply deblock CRLF to the entire input stream, because any CRLF sequences embedded in the forms data must be left intact. Thus, one must find the null line separating the two kinds of input *before* applying deblock CRLF to the first portion of the input stream. Furthermore, when there are forms data, it is more difficult to know when to stop trying to read from the input stream. There is no null line at the end of the forms data, nor any other indication of end-of-file. Instead, one parses the

"Content-Length:" header record found in the first portion of the input stream and reads the specified number of bytes beyond the null record.

When an added pipeline has connectors configured like the ones in HTTPD20 (above), the pipeline is inserted into the input stream of the stage that issues the ADDPIPE command; thus, it becomes a pre-processor for data flowing into that stage. In this example, incoming packets are first translated from ASCII to EBCDIC and are then reblocked appropriately. The first requirement is to be able to detect the null line (CRLF only) denoting the start of any forms data. joincont trailing x0D keep says to join two consecutive packets if the first one ends with CR, but not to remove that CR character in the process of joining the packets. split after 25 then splits the packets after each LF character to make sure that the null line will be at the start of a record, where the strtolabel can find it. The null line and the packets before it are then reblocked by deblock CRLF, which has the effect of removing the CRLF sequences and joining lines that had been split across packets. No further reblocking is performed on the packets that come after the null line (the forms data); in particular, any CRLF sequences they contain are left intact.

With the prefixed ADDPIPE doing the reblocking, HTTPD REXX needs no reblocking code itself and no longer invokes maketext. The records before the null line are read from the prefixed pipeline with a READTO command, and the data from the POSTed form are read using the take bytes capability that was added to *Pipes* recently to support the Byte File System. The REXX variable "content_length" contains the value from the "Content-Length:" header record, so this pipeline segment takes that many bytes without regard to record boundaries:

```
 . . .
'*: |',                              /* Get specified count of ...  */
'take' content_length 'bytes |',     /* ... bytes from the form.    */
'join * |',                          /* Make sure all in one piece. */
 . . .
```

Using the prefixed pipeline to do the translation and reblocking of incoming data cut all of the measurements dramatically and made the numbers for WebExplorer and Lurky essentially the same, confirming the conclusion that the difference in the cost of serving the two clients was due to the handling of the input lines:

Client	TOTCPU	Elapsed	SVCs	I/Os	Pipes
WebEx	10.5	---	19846	1052	971
Lurky	10.2	17	20069	1016	1004

Phase 3. Replacing GLOBALV with stemmed arrays: Because the Web server now supported multiple simultaneous clients, the CGI interface had to be changed at least to account for the fact that client-specific processing variables had a different name for each process. The modified GLOBALV interface was working, but the consensus on the WWW-VM list seemed to be that it was time to replace GLOBALV with stemmed arrays, so I did an implementation that uses CALLPIPEs to retrieve variables from stemmed arrays stored in HTTPD EXEC. The server-wide variables are stored in the array HTTPD., and the client-specific variables are stored in arrays with roots of the form HTTPDn., where "n" is the process number. Code using the

client variables refers to them as having the root CV. (to avoid the need to use Interpret). For example, this pipeline loads the server-wide and client-specific variables required by WEBSRVHT REXX:

```
cv = 'HTTPD'process                /* Stem for client variables.  */

'CALLPIPE (endchar ? listerr name LoadVars)', /* Load variables: */
     'literal CGIUSERS VERBOSE SERVER_NAME SERVER_PORT |',
     'split |',                    /* One at a time.              */
     'change //HTTPD./ |',         /* Stem name is server name.   */
  'f: fanin |',                    /* Server and client variables.*/
     'varfetch main 1 toload |',   /* They're in server EXEC.     */
     'change autofield 1 /'cv'./CV./ |',  /* Uninterpreted name. */
     'varload',                    /* Store variables locally.    */
'?',
     'literal PATH PATH_USER |',   /* Client-specific variables.  */
     'split |',                    /* One at a time.              */
     'change //'cv'./ |',          /* Server name || process num. */
  'f:'
```

The first pipeline segment here inserts a list of the required server variables, to which it prefixes the root HTTPD. The second pipeline segment inserts a list of the required client variables, to which it prefixes the root specified by the REXX variable “cv”. Both streams are fed to fanin, which feeds them to varfetch main 1 toload. varfetch fetches the values of the REXX variables named in its input records. main 1 says to go back to the REXX environment one level earlier than the one that issued the PIPE command. TCPSHELL EXEC issued the PIPE command, and it was called by HTTPD EXEC, so HTTPD EXEC is the REXX environment from which the variables are fetched. The toload keyword says to load the variables in the format required by varload, *i.e.*, a delimited string containing the name of the variable, followed by the value of the variable. change autofield 1 specifies that the change is to be made to an input range defined as a delimited string, that the delimiter character is whatever character is in column 1 of the input record, and that the range is the first such delimited string in the input record. Thus, this change changes the root of the names of the client variables to the constant “CV”. varload then stores the values of the required variables in the local environment.

Since we all know that EXECCOMM is expensive, I should not have been surprised that I had difficulty getting the non-GLOBALV server to perform well. In fact, I never got it to be quite as inexpensive as Rick’s code that used GLOBALVs (some issued with pipelines and some issued with REXX Address Command). By carefully reducing the number of pipelines used and the number of variables fetched, I finally got a version that was only slightly degraded:

Client	TOTCPU	Elapsed	SVCs	I/Os	Pipes
WebEx	11.2	---	16693	1053	1446
Lurky	10.9	18	16936	1018	1482

I have so far stuck with this interface, because it seems slightly nicer for use by CGIs, but I may go back to using GLOBALV.

Phase 4. Improving the multi-tasking: The method for dispatching work to the multiple processes in TCPSHELL EXEC is the same as was used earlier in SMTPLite. (See Appendix A.) A process queues for a client by writing a record containing its stream number into the secondary input of a deal secondary stage, which hands out client connection requests to the processes in the order in which it receives their stream number records.

What deal was doling out in the SMTPLite case was spoolids, but in TCPSHELL it is handing out socket descriptor records. This difference poses a problem for TCPSHELL that didn't arise with SMTPLite, because a socket descriptor record is a "living thing", as it were, and it must be handled more carefully than one handles a static record containing a spoolid number.

When tcplisten is ready to write a socket descriptor record, it performs a Givesocket() function. When tcplisten's write of the socket descriptor record completes, it closes the socket. If the socket has not been taken before then, TCPIP will terminate the connection. Indeed, the mechanism for rejecting a connection request is to consume the descriptor record without doing a Takesocket(). Thus, in the normal case, the descriptor record must not be consumed before tcpdata has done a Takesocket(). This means that the stages between tcplisten and tcpdata must not delay the socket descriptor record.

This was a problem for me in tcpshell, because there was so much work that needed to be done after the connection request was received but before interaction with the client could begin. The tcpshell filter consisted of a Do Forever loop that began with a PEEKTO that waited for deal secondary to write a socket descriptor record indicating a client connection request. Once that descriptor record had been parsed, tcpshell had then to:

1. Resolve the host name of the client's system;
2. Send the remote host an IDENT query to find out who the user was; and
3. Re-ACCESS any minidisks that had been changed since the last transaction.

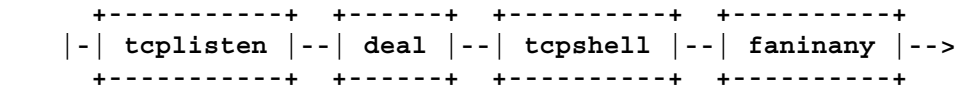
All three of these operations do I/O, so their elapsed time is significant. Only after all three operations had been performed could a CALLPIPE containing the tcpdata stage for the process be issued to consume the descriptor record, launch the server stage, and begin interacting with the client. You will note that this subroutine pipeline looks much like myserver (page 6):

```
'CALLPIPE (endchar ? listerr name TCPShellProcess)',
     '*: |',                    /* Connect to calling pipeline.   */
     'take 1 |',                /* Get the socket descriptor.     */
  'f: fanin |',                 /* Make feedback loop.            */
  't: tcpdata |',               /* Communicate with client.       */
     'elastic |',               /* Buffer/consume client output.  */
      server '|',               /* Run the specified server.      */
  'f:',                         /* Send server output to client.  */
'?',
  't: |',                       /* Any TCP/IP ERRNO to here.      */
     'nfind 0_OK|',             /* Normal socket-close ERRNO.     */
     'spec /'self '(tcpdata)/ 1 1-* nw |',    /* Identify ERRNO. */
     'console'                  /* Display any non-0 ERRNO.       */
```

This meant, of course, that tcplisten and deal secondary were blocked throughout those three steps, waiting for the descriptor record to be consumed, so they could not accept and dispatch other clients. That obviously limited the amount of multi-tasking that could occur.

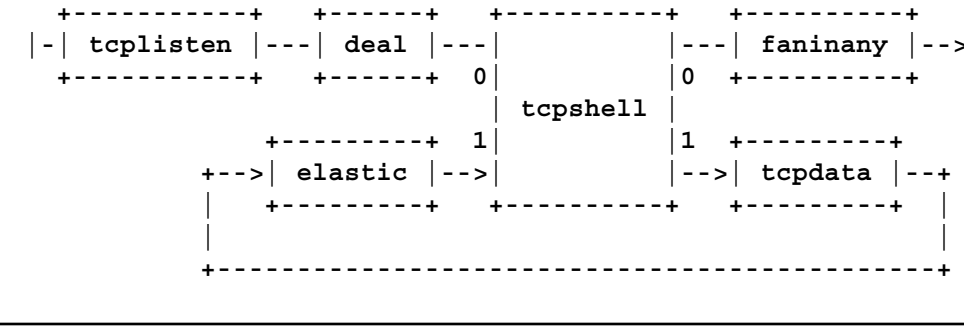
When I described this problem to John Hartmann, he suggested a very elegant solution that allows the socket descriptor record to be consumed as soon as deal secondary writes it. His idea was to add another pipeline to contain the tcpdata stage for the process as soon as the socket descriptor record arrives. That tcpdata could do the TakeSocket() immediately, and then the added pipeline could sit idle until tcpshell had completed the preliminary processing for the new client. Then tcpshell could connect to the added pipeline to use the tcpdata stage in that pipeline.

As before, tcpshell sits in a PEEKTO waiting to be given a socket descriptor record. While it is waiting, tcpshell has only a single input stream and a single output stream:



After the PEEKTO sees a record and completes, tcpshell uses an ADDSTREAM to add a set of secondary streams for itself and then issues this ADDPIPE command to create an “affixed” pipeline, *i.e.*, 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:

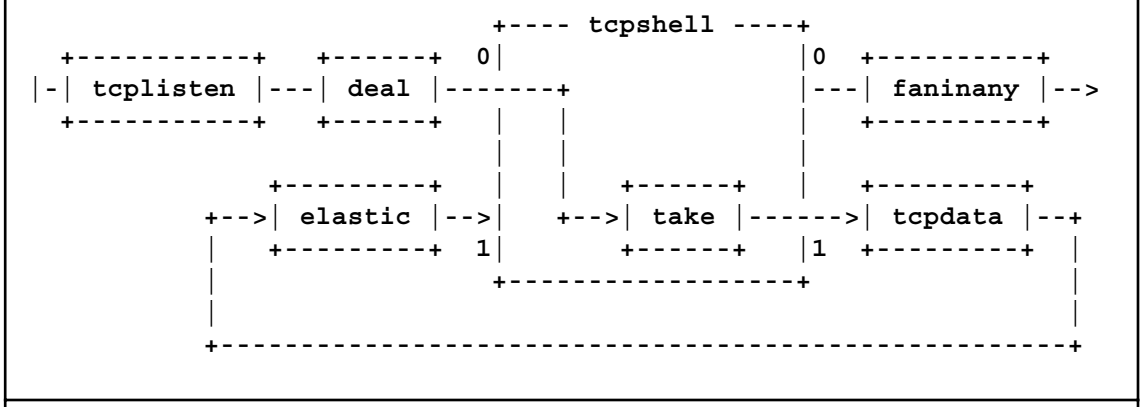
```
'ADDPIPE (endchar ? listerr name TCPShellComm)',/* Affix TCPDATA:*/
     '*.output.1: |',        /* Input from stage's 2ndry output. */
  't: tcpdata |',            /* Communicate with client.         */
     'elastic |',            /* Buffer/consume input for stage.  */
     '*.input.1:',           /* Output to stage's 2ndry input.   */
'?',
  't: |',                    /* Any TCP/IP ERRNO to here.        */
     'nfind 0_OK|',          /* Normal socket-close ERRNO.       */
     'spec /'self '(tcpdata)/ 1 1-* nw |',/* Identify the ERRNO. */
     'console'               /* Display any non-0 ERRNO.         */
```



After the ADDSTREAM and the ADDPIPE

As soon as the added pipeline has been affixed to the secondary input and output of tcpshell, tcpshell regains control and can issue this simple CALLPIPE:

```
'CALLPIPE (listerr name TakeSocket)',    /* Unblock TCPLISTEN:   */
  '*.input.0: |',             /* Get descriptor on primary.      */
  'take 1 |',                 /* Want just socket descriptor.    */
  '*.output.1:'               /* Feed TCPDATA on secondary.      */
```



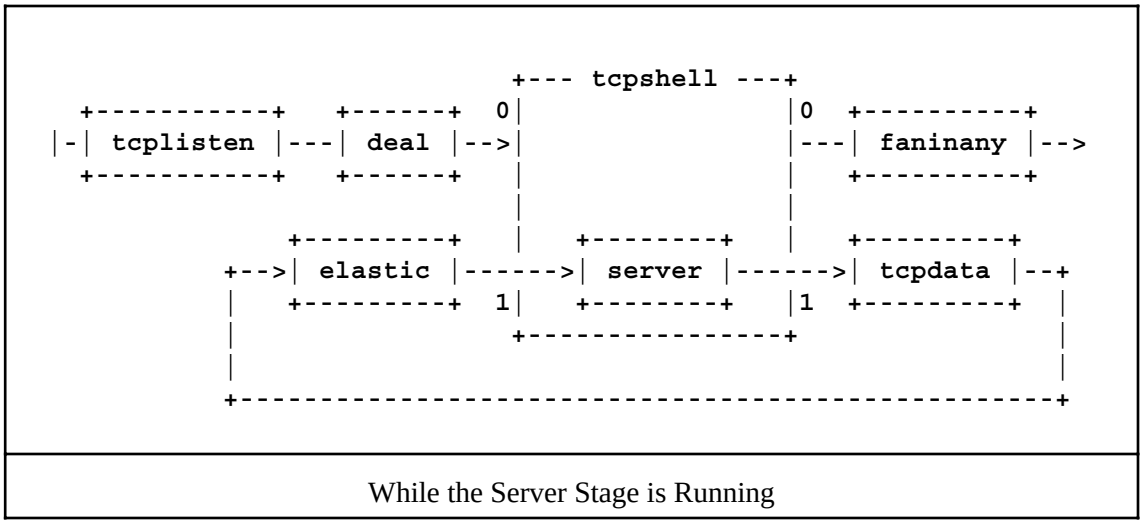
While the Take Stage is Running

The subroutine pipeline created by the CALLPIPE reads one record (the descriptor record) from tcpshell's primary input stream and writes it to tcpshell's *secondary* output stream, which is connected to the input of the affixed pipeline. When the tcpdata in the affixed pipeline receives the descriptor record from the take 1, it does a TakeSocket() and then consumes the descriptor record. The take 1 in the subroutine pipeline completes, which allows the CALLPIPE command to complete. deal secondary and tcplisten then see their writes of the descriptor record complete, too, which frees them to accept more client requests to be given to the other processes.

The affixed pipeline remains attached to tcpshell, but it has no further work to do for the time being. tcpshell continues running and can do the work necessary to resolve the host address, identify the client, and re-ACCESS the minidisks. Then, when tcpshell is ready to launch the server stage to interact with the client, it issues another simple CALLPIPE to connect the server stage (HTTPD REXX) to the affixed pipeline:

```
'CALLPIPE (listerr name TCPShellProcess)',  /* Run server stage: */
  '*.input.1: |',          /* Input from client via affixed pipe. */
   server '|',                        /* Run the specified server. */
  '*.output.1:'             /* Output to client via affixed pipe. */
```

The subroutine pipeline created by that CALLPIPE reads from the secondary input of tcpshell, which is connected to the output of the affixed pipeline, and writes to the secondary output of tcpshell, which is connected to the input of the affixed pipeline:

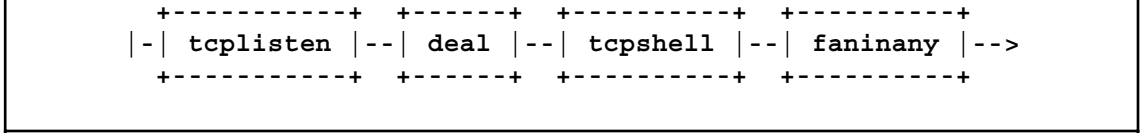


While the Server Stage is Running

Thus, the server stage receives data from the client through the tcpdata and elastic in the affixed pipeline and sends data to the client through that tcpdata.

When the transaction is complete, the server stage terminates, causing the CALLPIPE command to complete. tcpshell then severs its secondary streams:

```
'SELECT BOTH 1'                    /* Streams to affixed pipeline.*/
'SEVER INPUT'                      /* Sever affixed input.        */
'SEVER OUTPUT'                     /* Sever affixed output.       */
```



After the SEVERs

Because those streams had been connected to the affixed pipeline, severing them also causes the affixed pipeline to terminate. tcpshell reverts to its original single-stream state and issues another PEEKTO to wait to be assigned another client.

With that trick, tcplisten was able to listen for connection requests fairly continuously, and deal secondary was able to keep passing out clients to the server processes as soon as their requests arrived. The log clearly showed that the processes were overlapping; all four were frequently active simultaneously.

Due to the improved multi-tasking, elapsed time for my test case was reduced by 13%:

Client	TOTCPU	Elapsed	SVCs	I/Os	Pipes
WebEx	11.3	---	16813	1042	1509
Lurky	11.0	16	16877	994	1533

I suspected, however, that there might be a problem of a process executing for too long without giving up control to the *Pipelines* dispatcher, so I inserted half a dozen SUSPEND commands at strategic points in tcpshell and the server stage. (The SUSPEND command causes the stage that issues it to give up control to the *Pipelines* dispatcher and to be put at the end of the run list. If there are any other stages ready to run, at least one of them will be run before control is returned to the stage that issued the SUSPEND.) That did not improve the elapsed time:

Client	TOTCPU	Elapsed	SVCs	I/Os	Pipes
WebEx	11.2	---	16760	1044	1505
Lurky	11.0	16	16757	1011	1539

So, I removed the SUSPENDs and let the *Pipelines* dispatcher take care of the multi-tasking without my help.

Phase 5. Replacing other chunks of REXX with pipelines: The next phase involved lots of running Rita and trying my hand at recoding chunks of REXX as pipelines. Substantial portions of the websrvsp and websrvls filters proved amenable to being piped. A change to use GETFMADR MODULE, rather than GETFMADR EXEC, also proved beneficial. After considerable hacking, the results with my test case were:

Client	TOTCPU	Elapsed	SVCs	I/Os	Pipes
WebEx	9.0	---	14852	1072	1385
Lurky	9.0	14	15009	1038	1416

This improvement was achieved primarily by using a strategy of getting more of the function into fewer pipes. The number of pipeline specifications issued was reduced by eight percent; the average number of stages per pipeline grew from 4.3 to 4.8.

Phase 6. Putting back the blocking: I then turned my attention to the performance of the server when it was delivering a large file, and I soon found a bad problem of my own making. To deliver a 68-megabyte file cost 123 seconds of CPU time, the bulk of which was used by the tcpdata stage, which was being given many small packets to deliver to the client. That reminded me that several months earlier, while desperately trying to get a CGI working, I had removed an fblock 61440 stage of Rick's. That stage had been blocking all output to the client to gain I/O efficiency, but my CGI needed to be able to write small amounts of HTML to the client incrementally, to give the user updates on the progress of a long-running request. Since I still need to be able to write HTML to the client immediately, I could not put Rick's fblock back in the same place. Instead, I moved it to his websrvof filter, where it now blocks only binary output files. That reduced the CPU utilization for delivering the 68-megabyte file to 7.8 seconds from 123 and cut the elapsed time by more than half. Thus, I learned (yet again) the primal rule of performance: **Block it!**

I repeated my earlier tests with a truly unmodified version of Webshare:

Client	TOTCPU	Elapsed	SVCs	I/Os	Pipes
WebEx	20.3	---	31371	1247	~2575
Lurky	11.5	19	23659	1247	~1350

and with Webshare enhanced by modern *Pipes*, including the blocking of binary output:

Client	TOTCPU	Elapsed	SVCs	I/Os	Pipes
WebEx	8.7	---	14804	1082	1395
Lurky	8.5	13	14870	1030	1427

CGI considerations

It becomes important when writing CGIs for this environment that the CGIs perform as much of their function as possible using *CMS Pipelines*, to assure that the *Pipelines* dispatcher can regain control to handle external interrupts in a timely manner. Long-running CMS commands, which block the entire virtual machine, should be avoided, as should long periods executing REXX code. The *Pipelines* SUSPEND command can be used in blocks of REXX code to offer the dispatcher the opportunity to service another process.

With this version of Webshare, CGI routines must never issue a PIPMOD STOP command nor feed a record to a pipestop stage, as their doing so would terminate any active tcplisten, tcpdata, and tcpclient stages in the server. This difficulty is the subject of a SHARE requirement for a local option on pipestop.

```
/*   Copyright 1994, Richard M. Troth, all rights reserved.  <plaintext>
 *   Reproduced with permission
 *
 * THIS IS A NEW VERSION OF TCPSHELL WITH IMPROVED INVOCATION SYNTAX
 * (TCPSHELL doesn't have a version number because it is
 * included with other software which do have version numbers)
 *
 *        Name: TCPSHELL EXEC
 *              General purpose TCP/IP socket-to-pipeline interface
 *      Author: Rick Troth, Rice University, Information Systems
 *        Date: 1993-Feb-29, 1994-Feb-17
 *      Author: Rick Troth, Houston, Texas, USA
 *        Date: 1994-Feb-27, 1994-Oct-15, 1994-Jan-15
 *
 *              Modified by Melinda Varian, Princeton University, 1996
 *              o  Replaced RXSOCKET with Pipes built-in TCP support
 *              o  Replaced GLOBALV with stemmed arrays
 *              o  Made server multi-tasking
 *
 *     Co-reqs: CMS Pipelines version 1.0110
 *              REXX/Sockets (RXSOCKET V2)
 *              VM TCP/IP V2
 *
 *   "Globals": This routine stores the following in the stemmed array
 *              with the caller's name as the stem (in the caller):
 *              SERVER_NAME - FQDN of this server's host system
 *              SERVER_PORT - TCP port at which this server listens
 *              LOCALHOST - FQDN of this server's host system
 *              LOCALPORT - TCP port at which this server listens
 */

Signal On Novalue
Signal On Syntax
Signal On Error

Address Command

Parse Source . . self .
self = self':'                          /* For use in messages.        */


/*--------------------------------------------------------------------*/
/* Parse arguments from caller                                        */
/*--------------------------------------------------------------------*/

Parse Arg '(' opts ')' arg0.server_port arg0 processno backlog args

If arg0 == '' | arg0.server_port == ''/* Required arguments missing? */
Then Do                                 /* Yes, exit with error.       */
   'XMITMSG 386 (ERRMSG'
   Exit 24
End  /*  If  ..  Do  */
```

```
If ¬ Datatype(arg0.server_port,'N')   /* Is port number numeric?     */
Then Do                               /* No, exit with error.       */
   'XMITMSG 70 SERVER_PORT (ERRMSG'
   'XMITMSG 8205 (ERRMSG'
   Exit 24
End  /*  If  ..  Do  */

Say self "PORT" arg0.server_port      /* Display local port number.  */
arg0.localport = arg0.server_port     /* Remember as server variable.*/

If processno == ''                    /* Default other arguments.    */
   Then processno = 1
If backlog == ''                      /* Default to overflowing to...*/
   Then backlog = 0                   /* ...other virtual machines.  */

ident = 1                             /* Default options.            */
verbose = 1

Do While opts ¬= ""                   /* Parse user's options.       */
   Parse Var opts op opts
   Select  /*  op  */
      When  Abbrev("IDENT",op,2)      Then ident = 1
      When  Abbrev("NOIDENT",op,3)    Then ident = 0
      When  Abbrev("VERBOSE",op,1)    Then verbose = 1
      When  Abbrev("NOVERBOSE",op,3)  Then verbose = 0
      When  Abbrev("TERSE",op,5)      Then verbose = 0
      Otherwise Address "COMMAND" 'XMITMSG 3 OP (ERRMSG'
   End  /*  Select  op  */
End  /*  Do  While  */

If verbose  Then Do                   /* Say restarting the server.  */
   time = Date('S') Time()
   'XMITMSG 2323 ARG0 TIME (ERRMSG'
End  /*  If .. Do  */


/*-------------------------------------------------------------------*/
/* Verify that we have TCPIP DATA and RXSOCKET V2 for HOSTBYADDR     */
/*-------------------------------------------------------------------*/

'ESTATE TCPIP DATA'                   /* Do we have the TCP/IP disk? */
If RC ¬= 0  Then Do                   /* No, this won't work.        */
   Say 'TCPIP DATA not found.'        /* Explain the problem.        */
   Exit RC                            /* Exit with error.            */
End

Parse Value Socket("Version")  With rc . ver .
If ver < 2  Then Do
   Say self arg0 "server requires REXX/Sockets (RXSOCKET version 2)"
   Exit -1
End  /*  If  ..  Do  */
```

```
/*--------------------------------------------------------------------*/
/* Determine fully-qualified domain name of our host system           */
/*--------------------------------------------------------------------*/

Signal Off Error                    /* HOSTBYADDR may give non-0 RC. */

'PIPE (name GetHost)',                /* Determine "server_name":    */
   'hostid |',                        /* Ask TCPIP for host IP addr. */
   'hostbyaddr |',                    /* Resolve to FQDN.            */
   'append literal |',                /* In case can't resolve.      */
   'var arg0.localhost |',            /* Set "localhost".            */
   'xlate lower |',                   /* Lower-case it.              */
   'var arg0.server_name'             /* Set "server_name".          */

Signal On Error

If arg0.localhost == ''  Then Do      /* Unable to resolve address?  */
   Say self "Unable to determine local host's name."
   Exit RC                            /* Yes, go no further.         */
End

If verbose  Then Do                   /* Display local host name.    */
   Say self "SERVER_NAME" arg0.server_name
   'XMITMSG 8201 ARG0 (ERRMSG'
   'XMITMSG 740 (ERRMSG'
End  /*  If .. Do  */


/*--------------------------------------------------------------------*/
/* Store "global variables" in caller's REXX environment              */
/*--------------------------------------------------------------------*/

'PIPE (listerr name StoreStem)',      /* Save "global variables":    */
   'rexxvars toload |',               /* Load all local variables.   */
   'find  ARG0.|',                    /* Select with right stem.     */
   'change autofield 1 /ARG0./'arg0'./ |', /* Change to server name. */
   'varload 1'                        /* Store in server EXEC.       */

Say self "Ready;"                     /* Say that the server is up.  */


/*--------------------------------------------------------------------*/
/* Build the pipeline segments defining the specified number of       */
/* server processes                                                   */
/*--------------------------------------------------------------------*/

processes = ''                        /* Initialize pipeline segment.*/

Do i = 1 to processno                 /* Build server processes.     */

   processes = processes,             /* Append another process.     */
      'd: |',                         /* Next spool file id to here. */
         'tcpshell',                  /* Invoke server process:      */
```

```
            i,                          /*   Pass process number       */
            ident,                      /*   Pass IDENT option.        */
            verbose,                    /*   Pass VERBOSE option.      */
            arg0.server_port,           /*   Pass local port number.   */
            arg0,                       /*   Pass server name.         */
            args '|',                   /*   Pass arguments for server.*/
      'y:',                             /* Streamnumber recs from here.*/
   '?'

End


/*--------------------------------------------------------------------*/
/* Build the pipeline segments defining several immediate commands    */
/*                                                                    */
/* Note that while the STOP immediate command stops the server        */
/* quickly by posting the ECBs for all of the stages that wait for    */
/* external events, the QUIT immediate command is designed to         */
/* stop the server only after all active transactions are complete.   */
/*                                                                    */
/* STOP feeds a record into a PIPESTOP stage, which causes all of     */
/* the other IMMCMD stages to be posted, as well as all of the TCP    */
/* stages.  QUIT feeds a record into a GATE stage, which causes it    */
/* to terminate and sever all of its streams.  All of the other       */
/* IMMCMD stages have GATE on their output side, so they all stop.    */
/* The TCPLISTEN stage also has GATE on its output side, so it also   */
/* terminates; therefore, no more connection requests are received.   */
/* Because DEAL SECONDARY has GATE on its input side, it terminates   */
/* without dispatching another process.                               */
/*--------------------------------------------------------------------*/

immcmds =,                              /* Pipe segment used below.    */
      'immcmd QUIT |',                  /* QUIT was typed on console.  */
   'g: gate',                           /* Don't dispatch any more.    */
'?',
      'immcmd STOP |',                  /* STOP was typed on console.  */
   'g: |',                              /* (GATE closed by QUIT cmd.)  */
      'pipestop',                       /* Pull all the plugs now.     */
'?',
      'immcmd CP |',                    /* Immediate CP command.       */
   'g: |',                              /* (GATE closed by QUIT cmd.)  */
      'cp',                             /* Pass command to CP.         */
'?',
      'immcmd CMS |',                   /* Immediate CMS command.      */
   'g: |',                              /* (GATE closed by QUIT cmd.)  */
      'xlate upper |',                  /* Upper-case the command.     */
      'command',                        /* Pass command to CMS.        */
'?',
      'immcmd DUMP |',                  /* DUMP was typed on console.  */
   'g: |',                              /* (GATE closed by QUIT cmd.)  */
      'pipdump |',                      /* Format a dump.              */
      '>> pipdump listing d',           /* Write it to disk.           */
'?',
```

```
      'immcmd JEREMY |',              /* JEREMY was typed on console.*/
   'g: |',                            /* (GATE closed by QUIT cmd.)  */
      'jeremy |',                     /* Format pipeline status.     */
      'console'                       /* Display status.             */


/*--------------------------------------------------------------------*/
/* Execute the pipeline to dispatch clients to waiting server         */
/* processes                                                          */
/*                                                                    */
/* Note that the primary and secondary output of TCPLISTEN are both  */
/* run through GATE.  This is necessary because TCPLISTEN does not   */
/* terminate "prematurely" when only one of its output streams is    */
/* severed.                                                           */
/*--------------------------------------------------------------------*/

'PIPE (endchar ? listerr name TCPShell)',   /* Run server processes. */
       immcmds,                       /* Set up immediate commands.  */
'?',
   'l: tcplisten' arg0.server_port,   /* Wait for the next client.   */
         'BACKLOG' backlog,
         'REUSEADDR |',               /* Share port with other VMs.  */
   'g: |',                            /* (GATE closed by QUIT cmd.)  */
   'd: deal secondary',               /* Clients to ready processes. */
'?',
   'y: faninany |',                   /* Stream of streamnumber recs.*/
      'elastic |',                    /* Hold until ready for them.  */
       processes,                     /* Feed to server processes.   */
,
   'l: |',                            /* Any TCP/IP ERRNO to here.   */
   'g: |',                            /* (GATE closed by QUIT cmd.)  */
      'nfind 5001|',                  /* 5001 is not really an error.*/
      'spec /'self '(tcplisten)/ 1 1-* nw |',     /* Identify ERRNO. */
      'console'                       /* Display the ERRNO, if any.  */


/*--------------------------------------------------------------------*/
/* Exit                                                               */
/*--------------------------------------------------------------------*/

Error:

If verbose  Then Do                   /* Say shutting server down.   */
   time = Date('S') Time()
   'XMITMSG 2324 ARG0 TIME (ERRMSG'
End  /*  If .. Do  */

Exit
```

```
/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/
/*          TCPSHELL REXX:  This is a single server process           */
/*--------------------------------------------------------------------*/
/*--------------------------------------------------------------------*/

Signal On Error
Signal On Novalue
Signal On Syntax

Parse Source . . self .
Parse Arg process  . . . name .       /* Bases for stem name.        */
cv = name || process                  /* Client variable stem name.  */
self = self 'process' process':'      /* Identifier for messages.    */


/*--------------------------------------------------------------------*/
/* Parse arguments from caller                                        */
/*--------------------------------------------------------------------*/

Parse Arg ,                           /* Args from TCPSHELL EXEC:    */
   cv.server_process,                 /* Our process (stream) number.*/
   ident,                             /* Setting of IDENT option.    */
   verbose,                           /* Setting of VERBOSE option.  */
   server_port,                       /* Local port number.          */
   arg0,                              /* Name of server to be run.   */
   torun                              /* Arguments for server.       */


/*--------------------------------------------------------------------*/
/* Initialize variables and queue for a client                        */
/*--------------------------------------------------------------------*/

accessoutput = ''                     /* Pipe segment used below.    */
If verbose
   Then accessoutput = '| console'

ident_port = 113                      /* Well-known port for IDENT.  */

'OUTPUT' cv.server_process            /* Ready; queue for a client.  */


/*--------------------------------------------------------------------*/
/* Main loop:  Process client requests                                */
/*--------------------------------------------------------------------*/

| 'EOFREPORT ALL'                     /* Propagate eof quickly.      */

Do Forever                            /* Do until STOP or QUIT cmd.  */

   'PEEKTO record'                    /* Wait for client connection. */

   cv.remote_ident = ''               /* Re-initialize variables.    */
```

```
cv.remote_system = ''
cv.remote_user = ''


/*-----------------------------------------------------------------*/
/* Take the socket now to unblock TCPLISTEN.  Hold the client     */
/* connection in an "affixed" pipeline until ready to communicate */
/* with the client.                                                */
/*                                                                 */
/* The affixed pipeline gets its input from the secondary output  */
/* of TCPSHELL REXX and writes its output to the secondary input  */
/* of TCPSHELL REXX.  It runs in parallel with TCPSHELL REXX, so  */
/* TCPSHELL REXX can attach and unattach CALLPIPEs to those       */
/* secondary streams at will, when it needs to use the TCPDATA    */
/* stage in the affixed pipeline.                                  */
/*                                                                 */
/* Note the placement of ELASTIC between TCPDATA and the server   */
/* stage (which will later be connected to "*.input.1:").  This   */
/* means that all packets from the client will flow from TCPDATA  */
/* through ELASTIC to the server stage.  As there is likely to    */
/* be less traffic going from the client to the server than vice  */
/* versa, this configuration is preferred for that reason.        */
/*                                                                 */
/* However, there is a more important reason for positioning the  */
/* ELASTIC between TCPDATA and the server.  In this configuration,*/
/* input from the client is consumed immediately, thus freeing    */
/* TCPDATA to accept input from the server to be sent to the      */
/* client.  If ELASTIC were placed between the server stage and   */
/* TCPDATA, output records would tend to be buffered in ELASTIC   */
/* rather than being sent to the client immediately, as TCPDATA   */
/* might not be dispatched for a long while after the server      */
/* stage had consumed the input written by TCPDATA, and TCPDATA   */
/* might therefore be unable to read output records from ELASTIC. */
/*                                                                 */
/* The TakeSocket subroutine pipeline moves one record from the   */
/* primary input of TCPSHELL REXX (i.e., the socket descriptor    */
/* record already PEEKed) to the input of the affixed pipeline,   */
/* where it is read and consumed by TCPDATA, thus unblocking the  */
/* DEAL SECONDARY and TCPLISTEN in the calling pipeline in        */
/* TCPSHELL EXEC.                                                  */
/*-----------------------------------------------------------------*/

'ADDSTREAM BOTH'                    /* Add secondary streams.      */

'ADDPIPE (endchar ? listerr name TCPShellComm)',/* Affix TCPDATA: */
      '*.output.1: |',              /* Input from stage's output.  */
   't: tcpdata |',                  /* Communicate with client.    */
      'elastic |',                  /* Buffer input for stage.     */
      '*.input.1:',                 /* Output to stage's input.    */
'?',
   't: |',                          /* Any TCP/IP ERRNO to here.   */
      'nfind 0_OK|',                /* Normal socket-close ERRNO.  */
      'spec /'self '(tcpdata)/ 1 1-* nw |',/* Identify the ERRNO. */
```

```
      'console'                    /* Display any non-0 ERRNO.    */

   'CALLPIPE (listerr name TakeSocket)',       /* Unblock TCPLISTEN: */
      '*.input.0: |',               /* Get descriptor on primary.  */
      'take 1 |',                   /* Want just socket descriptor.*/
      '*.output.1:'                 /* Feed TCPDATA on secondary.  */


   /*-----------------------------------------------------------------*/
   /* Store client-specific "global" variables describing client      */
   /*-----------------------------------------------------------------*/

   'CALLPIPE (endchar ? listerr name DescribeClient)',
         'var record |',            /* Load socket descriptor.     */
      's: spec',                    /* Extract useful parts:       */
            '57.4 c2d 1',           /* Socket number to decimal.   */
            'write',                /* Write to primary stream.    */
            'outstream 1',          /* Switch to secondary stream. */
            '65.16 1 |',            /* Write client sockaddr_in.   */
         'strip |',                 /* Get bare socket number.     */
         'var cv.socket',           /* Store in client variable.   */
   '?',
      's: |',                       /* Client sockaddr_in here.    */
|        'socka2ip |',             /* Convert to readable form.   */
         'split |',                 /* Make 3 one-word records.    */
         'drop 1 |',                /* Discard "AF_INET" record.   */
         'var remote_port |',       /* Store client's port number. */
         'drop 1 |',                /* Discard that record, too.   */
         'var cv.remote_addr |',    /* Store client's host IP add. */
         'var cv.hostaddr |',       /* Store client's host IP add. */
|     'h: hostbyaddr |',           /* Resolve client's IP addr.   */
         'append literal |',        /* In case not resolved.       */
         'var cv.hostname |',       /* Store in client variable.   */
         'xlate lower |',           /* Lower-case the FQDN.        */
         'var cv.remote_host',      /* Store in client variable.   */
   '?',
      'h:'                          /* Force RC 0 from HOSTBYADDR. */


   /*-----------------------------------------------------------------*/
   /* Optionally IDENTify the client per RFC 1413                     */
   /*                                                                 */
   /*  Sample IDENT query: "49201 , 80"                               */
   /*  Sample IDENT reply: "38 49201 , 80 : USERID : VM/CMS : maint" */
   /*-----------------------------------------------------------------*/

   If ident  Then Do                /* Identify per RFC 1413.      */

      query = remote_port ',' server_port'0D25'x   /* Build command. */

      'CALLPIPE (endchar ? listerr name RFC1413)',
            'var query |',          /* Load IDENT query command.   */
            'xlate from 1047 to 819 |',/* Query from EBCDIC to ASCII.*/
```

```
         't: tcpclient',              /* Send IDENT query command.   */
                cv.remote_addr,       /* Browser host IP address.    */
                ident_port,           /* Standard IDENT port.        */
               'oneresponse',         /* Enforce 1 output per input. */
               'deblock string x0D0A',/* Delimited by CRLF.          */
               'timeout 10 |',        /* Time out after 10 seconds.  */
            'xlate from 819 to 1047 |',/* Reply from ASCII to EBCDIC.*/
            'xlate fs : f2 upper |',  /* Upper-case "userid".        */
            'not chop after string /USERID/ |', /* Discard header.   */
            'split at : |',           /* Isolate the two items.      */
            'strip |',                /* Tidy them up.               */
            'locate 1 |',             /* Discard if no "USERID".     */
            'append literal |',       /* In case no system type.     */
            'var cv.remote_system |', /* Store system type.          */
            'drop 1 |',               /* Drop first record.          */
            'append literal |',       /* In case no userid returned. */
            'var cv.remote_ident',    /* Store remote_ident.         */
      '?',
         't: |',                      /* Any TCP/IP ERRNO to here.   */
            'nfind 5002|',            /* Normal socket-close ERRNO.  */
            'spec',                   /* Identify ERRNO source.      */
               '/'self '(tcpclient)/ 1 1-* nw |',
            'console'                 /* Display any non-0 ERRNO.    */

   End

   If cv.remote_host == ""
      Then cv.client = cv.remote_ident'@'cv.remote_addr
   Else cv.client = cv.remote_ident'@'cv.remote_host


   /*-----------------------------------------------------------------*/
   /* Log acceptance of client                                        */
   /*-----------------------------------------------------------------*/

   If verbose
      Then Say self "Accepted" cv.socket "at" Time() "client" cv.client


   /*-----------------------------------------------------------------*/
   /* Store the client-specific variables in the server EXEC          */
   /*-----------------------------------------------------------------*/

   'CALLPIPE (listerr name StoreClient)',/* Save "client variables": */
      'rexxvars toload |',           /* Load all variables.         */
      'find  CV.|',                  /* Select with right stem.     */
      'change autofield 1 /CV./'cv'./ |', /* Store with server name. */
      'varload main 1'               /* Store in ultimate caller.   */


   /*-----------------------------------------------------------------*/
   /* Refresh disk access (same procedure as used by GONE EXEC)       */
   /*                                                                 */
```

```
/* Gary Buhrmaster's DISKWRIT program is used to test whether     */
/* any of the read-only minidisks (other than the S-disk and the  */
/* Y-disk) have been modified.  A CMS QUERY DISK command is       */
/* issued to get a list of all accessed disks.  The records for   */
/* R/W disks, accessed SFS directories, and disks accessed as     */
/* mode S are discarded.  Each remaining record is transformed    */
/* into an ACCESS command and a DISKWRIT command.  The DISKWRIT   */
/* command is issued.  If it gives a return code of 1 (meaning    */
/* that the disk has been changed), PREDSELECT is triggered to    */
/* pass the ACCESS command to a COMMAND stage to be issued.  If   */
/* DISKWRIT gives any other return code, PREDSELECT is triggered  */
/* to discard the ACCESS command without its being issued.        */
/*----------------------------------------------------------------*/

'CALLPIPE (endchar ? listerr name Refresh)',
      'command QUERY DISK |',      /* Query the disks.            */
      'locate 17.3 &R/O& |',       /* Ignore R/W minidisks.       */
      'nlocate 8.3 /DIR/ |',       /* Ignore accessed directories.*/
      'nlocate 13.3 /S/ |',        /* Ignore CMS system disks.    */
   'o: fanout |',                  /* R/O minidisk: divert copy.  */
      'copy |',                    /* Unblock FANOUT.             */
      'spec',                      /* Build ACCESS commands.      */
        '/ACCESS/ 1',
        '8-15 nw |',
   'p: predselect |',              /* Pass if DISKWRIT got RC=1.  */
      'command',                   /* Issue ACCESS command.       */
       accessoutput,               /* Possibly display result.    */
'?',
   'o: |',                         /* Copy of minidisk to here.   */
      'spec',                      /* Build DISKWRIT command.     */
        '/DISKWRIT/ 1',
        '13.1 nw |',
   'm: command |',                 /* Issue DISKWRIT command.     */
      'console',                   /* Display result (error).     */
'?',
   'm: |',                         /* DISKWRIT RC to here.        */
   'f: find 1|',                   /* Divert if not "1".          */
   'p:',                           /* Trigger selection.          */
'?',
   'f: |',                         /* DISKWRIT 0 RC to here.      */
   'p:'                            /* Trigger rejection.          */


/*----------------------------------------------------------------*/
/* Invoke the server stage (arg0) to process the client request   */
/*                                                                */
/* First check to be sure the TCPDATA pipeline is still affixed.  */
/* It would have terminated if a PIPMOD STOP was done.            */
/*                                                                */
/* The input of this subroutine pipeline is connected to the      */
/* output of the affixed pipeline containing TCPDATA, and its     */
/* output is connected to the input of that pipeline.  Thus,      */
/* packets from the client flow from TCPDATA into this subroutine */
```

```
   /* pipeline, and output from the server stage flows into TCPDATA  */
   /* to be sent to the client.                                      */
   /*                                                                */
   /* The server process is responsible for reblocking its input     */
   /* stream into proper records.  It should terminate when it has   */
   /* processed a transaction end record (as defined by the protocol)*/
   /* or has received some other indication that the client has sent */
   /* all the data it will send.  And the server process must create */
   /* responses to be sent to the client as defined by the protocol. */
   /*----------------------------------------------------------------*/

   Signal Off Error
   'STREAMSTATE OUTPUT 1'             /* Still have secondary output?*/
   If RC == 12  Then Exit 0           /* No, PIPMOD STOP was done.   */
   Signal On Error

   'CALLPIPE (listerr name TCPShellProcess)',   /* Run server stage: */
      '*.input.1: |',         /* Input from client via affixed pipe. */
       arg0 cv.server_process torun '|',    /* Run specified server. */
      '*.output.1:'            /* Output to client via affixed pipe. */


   /*----------------------------------------------------------------*/
   /* Clean up the affixed pipeline                                  */
   /*----------------------------------------------------------------*/

   'SELECT BOTH 1'                    /* Streams to affixed pipeline.*/
   'SEVER INPUT'                      /* Sever affixed input.        */
   'SEVER OUTPUT'                     /* Sever affixed output.       */

   If verbose  Then Say self "Closed" cv.socket "at" Time()


   /*----------------------------------------------------------------*/
   /* Queue for another client request                               */
   /*----------------------------------------------------------------*/

   'SELECT BOTH 0'                    /* Revert to primary streams.  */
   'OUTPUT' cv.server_process         /* Queue for a client.         */

End

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

Appendix C

JOHN HARTMANN'S THROTTLE SUBTASKER

One day a couple of years ago, John Hartmann sent Rob van der Heij and me a filter called THROTTLE REXX and included this test case:

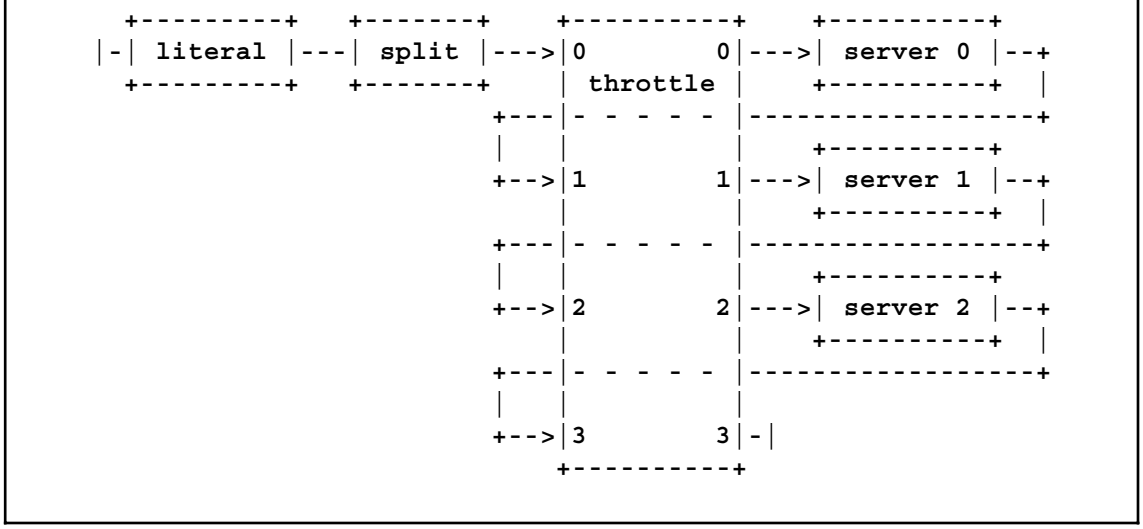
```
/* Test the throttler                                              */
/*                            John Hartmann  10 Apr 1996 15:24:06 */
Signal on novalue
Address COMMAND
'ERASE PIPDUMP LISTING'
'PIPE',
   '   literal a b c d e f g h i j k',
   '| split',
   '|t:throttle',
   '| change ,,0: ,',
   '| cons',
   '| spec /+0.2/ 1',
   '| copy',
   '| delay',
   '|t:',
   '| change ,,1: ,',
   '| cons',
   '| spec /+0.3/ 1',
   '| copy',
   '| delay',
   '|t:',
   '| change ,,2: ,',
   '| cons',
   '| spec /+0.5/ 1',
   '| copy',
   '| delay',
   '|t:'

Exit RC
```

There was no explanation of the test case other than a laconic, "Notice the multistream topology without end-characters". Sure enough, the throttle stage in this example has four input streams and three output streams connected, but no end character is defined for the pipeline.

THROTTLE REXX said of itself, "This subroutine distributes work amongst server pipelines, which are connected from output 0 to input 1, from output 1 to input 2, and so on (this makes for easy coding of the main pipeline)". The throttle stage implemented a scheme for distributing work among subtasks similar to that used in SMTPLite, but rearranged to support this rather unusual topology. Rob was then writing Lurky (see Appendix D), so he rose to the challenge and put throttle to work at once.

The current version of throttle is shown on the next page, and the current version of the test case is annotated below. The literal | split sequence creates work to be dispatched among the simulated server processes. Server process *n* reads a record describing work to do from its primary input, which is connected to throttle's output stream *n*. Later, the server process queues for more work by writing a record to its primary output, which is connected to throttle's input stream *n+1*:

```
/* Test the throttler                                          */
/*                           John Hartmann  20 Jul 1996 11:37:08 */

Signal on novalue
Address COMMAND
'PIPE',                             /* Test the throttler:         */
   '    literal a b c d e f g h i j k',       /* Some test data. */
   '|   split',                     /* Make multiple records.      */
   '| t:throttle',                 /* Dispatch multiple processes.*/
       server(0, 0.15),            /* Generate server process 0.  */
       server(1, 0.3),             /* Generate server process 1.  */
       server(2, 0.7)              /* Generate server process 2.  */
Exit RC

server:                             /* Generate pipeline segment:  */

parse arg task, time                /* Task num., delay interval.  */
return,
   '|   copy',                      /* Consume quickly to unblock. */
   '|   change //+'time '/',       /* Convert to delay interval.  */
   '|   delay',                    /* Wait a while.               */
   '|   literal ready',            /* Prime the dispatcher.       */
   '|   change ,,'task': ,',       /* Prepend the task number.    */
   '|   console',                  /* Display on console.         */
   '| t:'                          /* Request some work.          */
```



```
/* THROTTLE REXX:  Pass an input record to whichever output is ready */
/*                               John Hartmann  21 Jul 1996 11:35:40 */
Signal on novalue

/*********************************************************************/
/* This subroutine distributes work  amongst server pipelines, which */
/* are connected from output 0 to input 1, from output 1 to input 2, */
/* and so on (this makes for easy coding of the main pipeline).      */
/*                                                                   */
/* A server writes  a record when it is ready  to process a request. */
/* An input record  will then be made available.   The server should */
/* consume this  input record  "quickly" (to  avoid blockage  of the */
/* DEAL stage that doles out work to the performing pipelines).      */
/*                                                                   */
/* Each  server pipeline  should  contain a  sipping pipeline  along */
/* these lines:                                                      */
/*                                                                   */
/*     output ready!                                                 */
/*     callpipe *:|take 1|<server>                                   */
/*                                                                   */
/* Each "ready"  token is then  turned into the stream's  number and */
/* fed into the buffer of available streams.                         */
/*********************************************************************/

'maxstream output'                     /* How many server processes?  */
If RC=0                                /* None; nonsense!             */
   Then exit 999
maxstream = RC - 1                     /* Number of servers we have.  */

pipe=''                                /* Start generating a pipeline.*/
first='o:|'                            /* Reference FANOUT 1st pass.  */

do j=0 to maxstream                    /* Generate the connections:   */
   pipe=pipe,
      '?   *.input.'j+1':',            /* "Done" recs from process j. */
      '|   spec /'j'/ 1',              /* Convert to stream number.   */
      '| i:',                         /* Queue for work.             */
      '?'first,                        /* Feed DEAL's secondary input.*/
        'd:',                          /* Work from DEAL stream j+1.  */
      '|   *.output.'j+1':'            /* Work to server process j+1. */
   first=''                            /* No longer reference FANOUT. */
end

'CALLPIPE (listerr endchar ? name Throttle)',
   '? i:faninany',                     /* Stream number records here. */
   '|   elastic',                      /* Form a queue to get work.   */
   '| o:fanout',                       /* Shunt to DEAL's 2ndry input.*/
   '?   *.input.0:',                   /* Incoming stream of work.    */
   '| d:deal secondary',               /* Distribute work to servers. */
   '|   *.output.0:',                  /* Work to server process 0.   */
   pipe                                /* Processes generated above.  */

exit RC
```

This is the pipeline built and executed by throttle when it has four output streams defined:

```
CALLPIPE (endchar ? name Throttle)
     i:faninany                    /* Stream number records here. */
   |   elastic                     /* Form a queue to get work.   */
   | o:fanout                      /* Shunt to DEAL's 2ndry input.*/
   ?
       *.input.0:                  /* Incoming stream of work.    */
   | d:deal secondary              /* Dispatch work to processes. */
   |   *.output.0:                 /* Work to server process 0.   */
   ?
       *.input.1:                  /* "Done" recs from process 0. */
   |   spec /0/ 1                  /* Convert to stream number.   */
   | i:                            /* Queue for work.             */
   ?
     o:                            /* Stream of stream numbers..  */
   | d:                            /* ..to DEAL's stream 1 and..  */
   ,                               /* ..work from DEAL's stream 1.*/
   |   *.output.1:                 /* Work to server process 1.   */
   ?
       *.input.2:                  /* "Done" recs from process 1. */
   |   spec /1/ 1                  /* Convert to stream number.   */
   | i:                            /* Queue for work.             */
   ?
     d:                            /* Work from DEAL's stream 2.  */
   |   *.output.2:                 /* Work to server process 2.   */
   ?
       *.input.3:                  /* "Done" recs from process 2. */
   |   spec /2/ 1                  /* Convert to stream number.   */
   | i:                            /* Queue for work.             */
   ?
     d:                            /* DEAL's stream 3.            */
   |   *.output.3:                 /* Unconnected.                */
```

This pipeline will receive incoming work requests on its primary input stream and will distribute those requests to three server processes connected to its primary, secondary, and tertiary output streams. If any of those processes is missing or subsequently terminates, it is skipped over in the dispatching of work. When the input stream goes to end-of-file or all of the server processes have terminated, this pipeline terminates.

When a server process is ready for work, it writes a record on its primary output stream, which is connected to an input of throttle. throttle uses a specs stage to convert the record from the server process to a stream number and sends the stream number record to the secondary input (“stream 1”) of its deal secondary stage. As deal secondary receives work requests on its primary input, it assigns each one to the server represented by the next stream number record on its secondary input. Thus, the server processes are dispatched in the order in which they queued for work.

Appendix D

ROB VAN DER HEIJ'S LURKY WEB BROWSER SIMULATOR

While I was working on making Webshare multi-tasking, Rob van der Heij kindly offered to help by writing a tool I could use to measure the elapsed time to fetch a Web page and its embedded images. The result was LURKY EXEC, which simulates a Web browser:

```
/* LURKY EXEC:  Time fetching of a Web page and embedded images  */
/*              Author: Rob van der Heij, 29 Apr 1996            */

Parse Arg url .
If url = ''  Then url = '/'       /* Start at root by default.   */

server = 'pucc.princeton.edu 80'  /* Server and port we want.    */

'PIPE (endchar ? name Lurky)',    /* Measure loading a Web page: */
      'var url |',                /* Insert the URL for the page.*/
   'l: lurker' server 'linger 10 |',    /* Measure loading page. */
      'spec ,r, 1.3 ri 1-* nw |', /* Mark statistics for root.   */
   'f: faninany |',               /* Bring in IMG statistics.    */
      'change ,, , |',            /* Garnish.                    */
      'preface var url |',        /* Put URL at start of output. */
      '>> lurky output a',        /* Append the statistics file. */
'?',
   'l: |',                        /* HTML for page to here.      */
      'preface var url |',        /* Make base URL first record. */
      'lurkharv |',               /* Get the URLs for the IMGes. */
      'not lookup autoadd |',     /* Pass URLs haven't yet seen. */
      'console |',                /* Display on console.         */
      'lurkimg 4' server 'linger 10 |',/* Measure getting IMGes. */
   'f:'                           /* Shunt statistics to output. */

Return RC
```

LURKY loads in the URL for the page to be measured and passes it to the lurker stage. lurker uses tcpclient to send an HTTP GET command to the Web server to get the HTML file for the page. lurker measures the elapsed time to load the page and returns a statistics record on its primary output. If its secondary output is connected, as it is here, lurker also returns the requested file on that stream.

In LURKY EXEC, the statistics record from lurker is written to a log file, and the requested file is fed to the lurkharv stage, which "harvests" the image URLs from the file (assuming that it is an HTML file) and feeds them into not lookup autoadd.

not lookup autoadd deserves an explanation, for it is the simulated browser's image cache. lookup autoadd reads a detail record from its primary input and searches for a match in its in-memory reference file (the "master file"). If there is no match, that detail record (which is an image URL in this case) is written to lookup's secondary output *and* it is also added to the reference, so that it will be matched in the future, should there be another occurrence of the same URL. Thus, lookup's reference, which starts out empty here, simulates a browser's cache.

What not does is swap the primary and secondary output streams of lookup. Its purpose here is simply to avoid an unnecessary use of multiple streams, since only the secondary output of lookup (the unmatched details) is of interest in this case. The same thing could have been done without not, but it would have required multiple streams.

Those unmatched detail records (image URLs that have not been fetched before) are fed to the lurkimg stage, which uses John Hartmann's throttle stage (see Appendix C) to allocate work amongst four concurrent copies of lurker, to overlap the fetching of four images at a time, thus simulating a browser that loads multiple images at once. The output of lurkimg is the statistics records from lurker's loading of the image files; those statistics records are fed into the faninany, which routes them to the log file.

Rob's filters are included in the following pages and are well worth reading as examples of heavy-duty plumbing.

The elapsed time calculation in LURKER REXX is of particular interest. Just before the HTTP GET request is transmitted to the Web server *via* tcpclient, a null record is generated and then timestamped (using the tod keyword of specs). As each packet arrives from the Web server, it is turned into a record that says how many bytes were in the packet (addrdw is used for this). These records are also timestamped. Thus, raw packets from the Web server are turned into the information needed to calculate both elapsed time and transfer rate. The specs "407 emulation" is used to do those calculations.

```
/* LURKER REXX:                            Measure time to load file   */
/*                    Author: Rob van der Heij, 14 Apr 1996            */

Parse Arg httpd
If httpd = ''  Then httpd = 'pucc.princeton.edu 8000 linger 10'

Trace Off; 'STREAMSTATE OUTPUT 1'; Trace Normal
If Find('0 4 8', RC) = 0              /* Secondary output defined?   */
   Then display = ''                  /* No, don't try to write it.  */
Else display = '? o: | *.output.1:'   /* Yes, raw file to there.     */

'CALLPIPE (endchar ? name Lurker)',   /* Measure fetching a file:    */
      '*: |',                         /* URL to fetch from caller.   */
      'buffer |',                     /* Let STRLITERAL get going.   */
      'spec ,GET, 1 1-* nw ,HTTP/1.1, nw |',/* Format HTTP request.  */
      'xlate from 1047 to 819 |',     /* Must be in ASCII..          */
      'append literal |',             /* .. terminated with a null.. */
      'spec 1-* 1 x0D0A next |',      /* .. delimited with CRLF.     */
   't: tcpclient' httpd '|',          /* Connect to Web server.      */
   'o: fanout |',                     /* Maybe divert copy of file.  */
      'strliteral ,, |',              /* Prepend null record (start).*/
      'addrdw cms4 |',                /* Prefix 4-byte length.       */
      'spec 1.4 c2d 1.10 ri |',       /* Convert that to decimal.    */
      'spec tod c2b 1 1-* nw |',      /* Mark with TOD value.        */
      'spec ,0, 1 22.31 n 65-* nw |', /* Bits 22-52 of TOD, length.  */
      'spec 1.32 b2d 1 34-* nw |',    /* TOD to decimal, length.     */
      'spec',                         /* Calculate time and rate:    */
         'a: 2.10 -',                 /* Field a is TOD value.       */
         'b: 14.10 -',                /* Field b is packet length.   */
         '1-* 1',                     /* Output entire record.       */
         'if #0>0 then',              /* If not first packet, ...    */
            'set #5:=(a-#0)/100',     /* ...set Ctr 5 to elapsed...  */
            'endif',                  /* ...since last packet /100.  */
         'print #5 nw',               /* Display elapsed interval.   */
         'if #5>0 then',              /* If interval non-zero, ...   */
            'print b/#5 nw endif',    /* ...display interval rate.   */
         'set #0:=a;#1+=b;#6+=#5',    /* Counter 0 = previous TOD.   */
          ,                           /* Counter 1 = total bytes.    */
          ,                           /* Counter 6 = total elapsed.  */
         'eof',                       /* When entire file received:  */
            'if #6>0 then',           /* If have elapsed time:       */
               'print #1 13',         /* Display total bytes.        */
               'print #6 nw',         /* Display total elapsed /100. */
               'print #1/#6 nw endif |',    /* Display overall rate. */
      'lurksumm |',                   /* Tabulate measurements.      */
      '*:',                           /* Statistics to caller.       */
'?',
   't: |',
      'var returns',                  /* Save any TCP/IP ERRNO.      */
,
       display                        /* Possibly output raw file.   */

Return RC * ( RC <> 12 )
```

```
/* LURKHARV REXX:                          Harvest IMGes in HTML file  */
/*                   Author: Rob van der Heij, 14 Apr 1996             */

'READTO url'                            /* URL to be harvested is 1st. */
If RC ¬= 0  Then Return RC

Trace Off; 'STREAMSTATE OUTPUT 1'; Trace Normal
If Find('0 4 8', RC) = 0                /* Secondary output defined?   */
   Then copy = ''                       /* No, don't try to write it.  */
Else copy = 'fo: fanout | *.output.1: ? fo: |'   /* Yes, copy input. */

'ADDPIPE (endchar ? name Harvest1)',    /* Pre-process stage's input:  */
   '*.input.0: |',                      /* Input for this stage.       */
   'xlate from 819 to 1047 |',          /* ASCII to EBCDIC.            */
   'deblock crlf |',                    /* Deblock into lines.         */
    copy,                               /* Possibly copy to output.    */
   '*.input.0:'                         /* Input to this stage.        */

'CALLPIPE (name Harvest2)',             /* Determine content type:     */
   '*: |',                              /* Get header records.         */
   'totarget nlocate 1 |',              /* Until first null line.      */
   'strfind anycase /content-type: / |', /* Select Content-Type rec. */
   'spec w2 1 |',                       /* Isolate content type.       */
   'var typ'                            /* Store in REXX variable.     */

If typ == 'text/html'  Then             /* Is our input stream HTML?   */
   'CALLPIPE (endchar ? name Harvest3)',   /* Yes, harvest IMG URLs: */
         '*: |',                        /* Get rest of input stream.   */
         'drop 1 |',                    /* Drop the null line.         */
         'xlate *-* 25 40 |',           /* Convert linefeeds to blanks.*/
         'strip trailing |',            /* Discard trailing blanks.    */
         'deblock linend < |',          /* One HTML tag per record.    */
         'chop > |',                    /* Discard after HTML tags.    */
         'strfind anycase /IMG_/ |',    /* Select only IMG tags.       */
         'split |',                     /* Break into "words".         */
         'strfind anycase /SRC=/ |',    /* Select only SRC attribute.  */
         'spec 5-* 1 |',                /* Isolate the URL.            */
         'change /"// |',               /* Remove quotes.              */
      'f: find / |',                    /* Divert if need full path.   */
      'y: faninany |',                  /* Back in with path prefixed. */
         '*:',                          /* Output full path to caller. */
   '?  f: |',                           /* Here if no leading slash.   */
      'c: change anycase 1.7 ,http://,, |',  /* Select if "HTTP://". */
         'not chop before / |',         /* Strip host and port.        */
      'y:',                             /* And output path to caller.  */
   '?     var url |',                   /* Load path argument.         */
      'j: juxtapose |',                 /* Prefix to relative paths.   */
      'y:',                             /* And output path to caller.  */
   '?  c: | j:'                         /* Here if partial path.       */

Else 'CALLPIPE *: | hole'               /* Not HTML; discard.          */

Return RC
```

```
/* LURKIMG REXX:      Generate the streams to retrieve images        */
/*                    Author: Rob van der Heij, 8 May 1996           */

Parse Arg n parms                      /* Engine count and parms.     */

/*-------------------------------------------------------------------*/
/*                                                                   */
/* Generate and execute a pipeline to invoke multiple LURKER stages  */
/* to retrieve multiple Web image files concurrently.  If the engine */
/* count is 3, the following pipeline will be executed:              */
/*                                                                   */
/*   CALLPIPE (endchar ? name LurkIMG)                               */
/*   ? fi: faninany               /* Stats records to here.      */  */
/*   |     *.output.0:            /* Send to output stream.      */  */
/*   ?     *.input.0:             /* Stream of image URLs in.    */  */
/*   ,                            /* Input 0: work to dispatch.  */  */
/*   | th: throttle               /* Distribute to engines.      */  */
/*   ,                            /* Output 0: URL for process 0 */  */
/*   |     lurkproc 0 parms       /* Measure fetching IMG.       */  */
/*   | f0: fanout                 /* Divert statistics copy.     */  */
/*   ,                            /* Input 1: record to queue.   */  */
/*   | th:                        /* Output 1: URL for process 1 */  */
/*   |     lurkproc 1 parms       /* Measure fetching IMG.       */  */
/*   | f1: fanout                 /* Divert statistics copy.     */  */
/*   ,                            /* Input 2: record to queue.   */  */
/*   | th:                        /* Output 2: URL for process 2 */  */
/*   |     lurkproc 2 parms       /* Measure fetching IMG.       */  */
/*   | f2: fanout                 /* Divert statistics copy.     */  */
/*   | th:                        /* Input 3: record to queue.   */  */
/*   ? f0: | drop 1 | fi:        /* Process 0 stats records.    */  */
/*   ? f1: | drop 1 | fi:        /* Process 1 stats records.    */  */
/*   ? f2: | drop 1 | fi:        /* Process 2 stats records.    */  */
/*                                                                   */
/*-------------------------------------------------------------------*/

pip1 = ''; pip2 = ''                   /* To generate pipe segments.  */

Do i = 0 to n-1                        /* Two segments per engine:    */
   pip1 = pip1 '| lurkproc' i parms '| f'i': fanout | th:'
   pip2 = pip2 '? f'i': | drop 1 | fi:'
End

'CALLPIPE (endchar ? name LurkIMG)',   /* Get IMGes concurrently:     */
  'fi: faninany |',                    /* All stats records here.     */
      '*.output.0:',                   /* And out to caller.          */
'?',
      '*.input.0: |',                  /* IMG URLs come in here.      */
  'th: throttle',                      /* Dispatch URLs to engines.   */
       pip1,                           /* Pipe segment from above.    */
,
       pip2                            /* Pipe segment from above.    */

Return RC * ( RC <> 12 )
```

```
/* LURKPROC REXX:                          Image-fetching engine       */
/*                    Author: Rob van der Heij, 8 May 1996             */

Signal On Error

Parse Arg nr parms                    /* Engine number and parms.    */

'OUTPUT'                              /* Prime THROTTLE.             */

Do i = 1                              /* Do until end-of-file.       */

   'READTO url'                       /* Unblock THROTTLE.           */

   'CALLPIPE (name LurkProcess)',     /* Invoke LURKER to get file:  */
      'var url |',                    /* Load URL into pipeline.     */
      'lurker' parms '|',             /* Measure getting the file.   */
      'spec ,'nr', 1.3 ri 1-* nw |',  /* Engine number before stats. */
      '*:'                            /* Output to report file.      */

End

Error: Return RC * ( RC <> 12 )
```

```
/* LURKSUMM REXX:                          Summarize time to load file */
/*                    Author: Rob van der Heij, 5 May 1996             */

'CALLPIPE (endchar ? name LurkSumm) *: | spec 3-* 1 | stem x.'

n = x.0                               /* Count of SPEC output lines. */
If n = 2                              /* Is there a totals record?   */
   Then Return                        /* No, nothing for us to do.   */

Parse Var x.n t.bytes t.time t.rate   /* Values from totals record.  */

Parse Var x.2 . b t r .               /* Values for first packet.    */

'OUTPUT'  'Tot:' Right(t.bytes,7) Right(t.time,8) ,
          'First:' k(b) Right(t,8) ,
          'Rest:'  k(t.bytes-b) Right(t.time-t,8)

Return RC * ( RC <> 12 )


k: Procedure                          /* Convert bytes to K-bytes.   */
   Return Right((512+Arg(1))%1024,6)'K'
```

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.[5]

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:

---

[5] 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

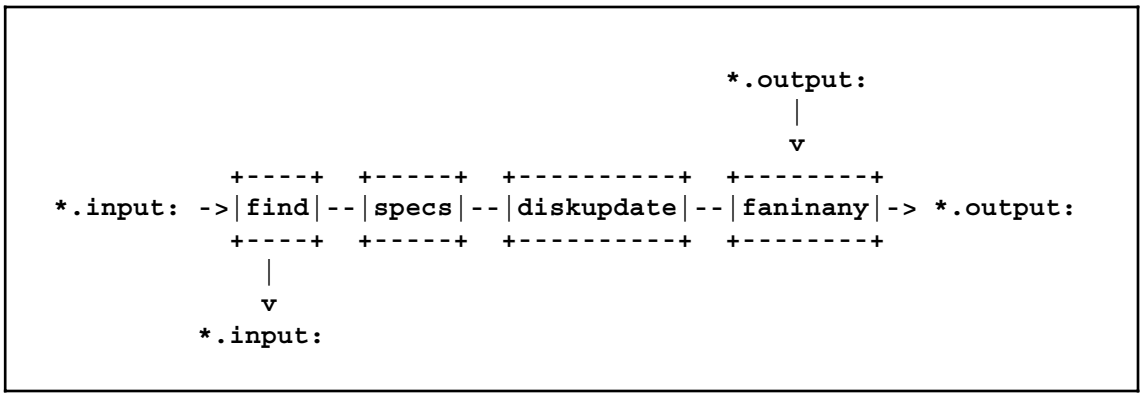
package. (As there is nothing here to prevent a file from being named twice, there would typically be a sort unique stage further on in the pipeline to discard duplicate records.)

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:[6]

```
/* 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.      */
```

[6] 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.