Writing a Piped TN3270 Client

Rob van der Heij (VIS)
Origin IT Systems Management
P.O. Box 218, VA-5,
5600 MD Eindhoven, The Netherlands

c772503@nlevdpsb.snads.philips.nl
rob@e-mail.com
NLC2L52L at IBMMAIL

SHARE 87
Session 9117
July 1996

Introduction

Access to IBM mainframes via TCP/IP is frequently done using the TN3270 protocol (a variation on the Telnet protocol). The client (a workstation) connects to the server (a host) and establishes a TCP/IP connection that allows for transparent transmission of 3270 orders.

This document describes the development of a CMS based TN3270 client, a program running under CMS (using the CMS console) that uses *CMS/TSO Pipelines* TCP/IP support stages to connects to a remote host using the TN3270 protocol. This function is indeed similar to what the TELNET command (part of TCP/IP VM) does, but I had a few good reasons to write my own.

- The proxy we use to control access to the Internet requires a dialogue (to specify the remote host) before the actual connection is made. The standard TELNET confuses this dialogue with a real host session and connects me in line mode to the remote host.
- Several of my favorite TN3270 clients (e.g. IBM Personal Communications V4, OS/2 Warp's PMANT, my version of X3270) do not support the so called “transparent mode” required to complete the dialogue with the proxy.
- The new built-in TCP/IP stages in CMS/TSO Piplines are so elegant that they simply beg to be exploited for a real application.

What at first appeared to be a simple task turned out to be a serious plumbing effort that taught me various new tricks that I would like to share with you. The code presented in this document does not yet solve the issue of my PC based clients, but I hope to extend the code later to deal with that.

Telnet Protocol

The Telnet protocol is used to form a bi-directional connection between a terminal and a host using TCP/IP. The connection is used to transmit both data and control information. In its basic form both sides use the concept of a Network Virtual Terminal (NVT) which is in fact a very simplistic ASCII keyboard and display.

Telnet controls are used by both parties (server and client) to negotiate options that extend the capabilites of the connection beyond that basic NVT.

Telnet controls

The Telnet controls are distinguished from the data by prefixing them with a x'ff' byte, referred to as the Interpret As Command (IAC) code. The popular control codes are in the range from x'f0' to x'fe'. Depending on the code there may be a number of bytes following the control code that are considered to be part of the Telnet command.

TN3270 Protocol

TN3270 consists of at least two options on top of the Telnet protocol (binary transmission and End-of-Record) that allow for transparent transmission of 3270 orders. As soon as these two options are agreed among the parties the Telnet session is considered to be in TN3270 mode, and the data carried by the connection is considered to be a 3270 data stream (and the emulator will complain if it is not).

RFC's

The different protocols are defined in so called RFC (Request For Comment) documents. Each of these documents describes a certain protocol or part of it. The documents are available on the Internet at various locations (and in different formats).

Note: When retrieving RFC's from a site it may be useful to also get a recent rfc-index. The index not only lists all documents but also indicates when a specific one is being replaced by a later version (but you will probably find both RFC's on the Net, and some people still refer to the outdated one).

A Simplistic Client

The following is the basic idea that could be used to form the TN3270 client. You will see this kind of pipeline more often when using the TCP/IP stages so it's worth to spend some time on this to understand the concepts involved.

```
'PIPE (end ?)',
    'f: faninany |',
    'tcpclient |',
    'fullscr |',
    'elastic |',
    'f:'
```

We assume the Telnet connection already being in 3270 mode, so the output of the tcpclient stage consists of 3270 orders that need to be written to the console. They are fed into the fullscr stage and will be displayed on the console. The output of fullscr are the commands typed on the console. These are fed back to the input of the tcpclient stage which will pass that as terminal input to the remote host.

The elastic is used to make sure fullscr can write to its output without being blocked by tcpclient by buffering the records when necessary. The faninany basically is a typographical trick to be able to specify a loop in the pipeline.

The simplified client has a number of problems that have to be solved to make it work. These problems (and the chosen solution) will be presented in the following sections.

The Programs

The following sections show the programs that form a working TN3270 client for CMS. The programs need CMS level 12 (i.e. VM/ESA V2) or *CMS/TSO Pipelines* 1.09. VM/ESA customers running an older level can get a copy of the Runtime Library from Princeton University at the following World Wide Web pages.

```
http://pucc.princeton.edu/.pipeline
```

The Runtime Library is also available via anonymous FTP from

```
ftp pucc.princeton.edu
cd anonymous.376
```

Inspired by Donald E Knuth's "Literate Programming" I will present the program text interleaved with an explanation about what the code does (the actual program text is marked with a bar in the left margin). There also is a quick and dirty pipeline that extracts the program texts from my DCF input, this restricts me a bit in the order in which I present the program sections. I'm aware of that and I should be working on it, but it normally only gets a burden when I'm close to a deadline.

PX3270 EXEC The TN3270 Client Program

This is the whole program. I decided to put the IP address of our Telnet Proxy server hardcoded in the program to try things. The virtual address of the graphics device (if other than the virtual console) can be customized when necessary.

Note: When testing it is very handy to use a separate screen for the TN3270 session so you can output some debugging information on the console.

```
addr = ''
'PIPE (end ? name N8)|',
   'f: faninany |',
   'tcpclient 130.144.199.37 23 |',
   'px3270m' addr '|',
   'elastic | f:'
return rc
```

PX3270M REXX Main processing stage

This is the main process of the TN3270 client. The primary input of this stage should be connected to the output of tcpclient and the primary output should be fed back to the input of that one.

Comparing with the scheme of the simplistic client the "px3270m" will in fact be a wrapper around the fullscr stage in there.

Even when taking the TN3720 negotiation for granted for a moment, we still need to decode the Telnet Controls to be able to find the End-Of-Record control and deblock the byte stream from tcpclient into records that start with a CCW and a WCC byte.

The EOR control is a x'ffef' pair of bytes but it would be incorrect to simply use a

```
deblock linend xffef
```

This would not cover the case where the first x'ff' is escaped by an IAC (the x'ffffef' would be the literal x'ffef' rather than the control, etc).

The following pipeline deals with this by combining the IAC and the next byte (the between stage) and then checks for the sequence that forms the literal x'ff' byte.

```
pipe = ,
       'fblock 1 |',
       'ff: between xff 2 |',          /* Pick the IAC                */
       'join |',                       /* .. with the next byte       */
       'fz: locate 2.1 xff |',         /* Followed by xff again       */
       'chop 1 |',                     /* .. makes it a real ff       */
       'fi: faninany |'

tail = ,
     '? ff: | fi:',                    /* The normal data             */
     '? fz: | fi:'                     /* The real controls           */
```

While this does mark the real IAC and the following byte, any parameters on the Telnet command are still not distinguished from the real data.

The following pipeline deals with the four telnet commands that have a single parameter byte (these are controls x'fb' to x'fe'). Similar to the thing above joining the IAC and the control byte, this one uses the between and join to build the three-byte control sequences. Sine I need to do this for each of those 4 codes I have a set of cascaded between stages.

```
 ,
       'fb: between xfffb 2 |',                          /* IAC Will  */
       'f2: faninany | join |',       /* Join the option byte in     */
       'fy: faninany |'
tail = tail ,
     '? fb: | fc: between xfffc 2 | f2:',                /* IAC Won't */
     '? fc: | fd: between xfffd 2 | f2:',                /* IAC Do    */
     '? fd: | fe: between xfffe 2 | f2:',                /* IAC Don't */
     '? fe: | fy:'
```

There is one other important telnet control that is using a variable number of bytes as its parameters. This one in fact uses a terminator that indicates the end of the parameters. This is documented in RFC854 as negotiation of sub-options. Building on top of the plumbing shown above this means anything between a x'fffa' and x'fff0' inclusive is considered as one single control.

The classical trick to join all records between the two brackets into a single record is to add a special character between the groups and use the deblock linend to join them. Since the x'15' could also appear within the data I have used c2x to protect the data.

```
pipe = pipe ,
      'fa: between xfffa xfff0 |',    /* From IAC SB to IAC SE       */
      'nl: locate 1.2 xfff0 |',       /* IAC SE (end sub)            */
      'spec 1-* c2x 1 x15 n |',
      'f0: faninany |',
      'deblock linend | locate 1 |',
      'spec 1-* x2c 1 |',
      'fk: faninany |'
tail = tail ,
    '? nl: | spec 1-* c2x 1 | f0:',
    '? fa: | fk:'                    /* All outside the brackets    */
```

Note: It is important in this case to put the x'15' **after** the records to be joined, and not **before** them. While the basic idea is just to separate the groups, having the x'15' before the records would delay the record to when the next IAC SB sequence arrives.

```
'addpipe (end ? name px3270m.a) *: |' pipe '*.input:' tail
```

The data stream we have now consists of telnet control sequences (the records longer than one byte) and ordinary data (be it either ASCII NVT or 3270 data stream). The protocol and the telnet controls imbedded determine whether data is NVT or 3270, you cannot tell otherwise. This means there is little more to be done now on deblocking the data before we know in which mode we are (you don't want to block too much since that would delay records when it should not).

```
arg addr .
call initialize
signal on error
```

One thing we do know about the protocol is that the connection starts in NVT mode. It will switch over to 3270 mode as soon as the required options have been negotiated by both sides. What we need are two pipelines that process data according to the protocol and terminate when they determine the mode has changed.

```
do forever
  'peekto'; call nvt
  'peekto'; call tn3270
end
error: return rc * ( rc ¬= 12 )
```

tn3270: Run the TN3270 session

This is the pipeline that runs the entire TN3270 session. It only terminates when the session is switched back to NVT mode or when we end the connection.

Note: Actually, to make sure the pipeline terminates properly can be quite difficult. Both the listrc option and the help of Melinda Varian are extremely useful for that.

```
tn3270:
```

The first section of the pipeline is to make sure we can use a gate to cut the various feedback sections in the pipeline. The faninany is used to receive the various records that may cause the pipeline to terminate. The typical structure would be like this

```
    'f0: faninany | g: gate',
'? *: | g: |'
```

to make sure the gate will close the primary input as early as possible. The following real code is a bit more complex since I want the gate to be closed on eof of the primary input too.

```
'callpipe (end ? name px3270m.tn3270)',
    '*: | fo: fanout stop anyeof | hole | append literal |',
    'f0: faninany | g: gate',
  '? fo: | g: |',
```

We now need to restore the blocking into 3270 datastream records. These records are separated by special Telnet “EOR” controls. The nlocate separates all controls from the normal data and the “EOR” is merged back into the stream (as a x15) after which the deblock linend joins all the characters upto the linend that was merged in. Since we cannot guarantee there are no x15 characters in the data we have to protect that by translating to hex and back again. This is where the c2x and the x2c fit in.

```
    'x1: nlocate 2 |',
    'spec 1.1 c2x 1 |',
    'f1: faninany |',
    'deblock linend |',
    'spec 1-* x2c 1 |',
```

What we have now are the pure 3270 data stream records starting with a CCW and WCC as produced by the host we connect to. The CCW codes have to be translated to the CCW accepted by fullscr for erase/write and erase write/alternate.

Note: I found one situation where our proxy suddenly starts to talk ASCII when the connection is dropped (proper procedure would be for the proxy to send Telnet controls to end Binary mode first). To prevent this from causing errors when writing to the console I'll just ignore them. I have no way to let NVT use them again (cannot push it back in the pipe from here, deblocking the data has already consumed to many records). According to RFC1041 there is no way defined to get back from 3270 mode to NVT, I'm not sure I read it like that.

```
    'pick 1.1 <<= x0d |',
    'xlate 1.1 05 80 0d c0 |',
    'fullscr' addr 'console path tn3270 condread asynch |',
    'strnfind x02 |',
```

The output of the fullscr stage are the commands entered at the console (we ignore the x02 representing the suppressed read when no data was available). We need to escape any imbedded x'ff' bytes and add the “EOR” after each record.

```
    'change xff xffff |',             /* Escape the FF               */
    'spec 1-* 1 xffef n |',           /* Add the EOR                 */
    'fx: faninany |',
    '*:',
```

The following section deals with the different Telnet controls. Most are processed in a REXX stage “px3270c” but two of them are too simple not to do them here (and doing so reduces overhead of the REXX stage).

The first one is the escaped x'ff' which should be inserted as 'FF' since it bypassed the c2x used for the other data.

```
  '? x1: | x2: locate 1.2 xffff | spec ,FF, 1 | f1:',
```

and the other one is the EOR that is fed into the pipeline as a x'15' for deblocking.

```
  '? x2: | x3: locate 1.2 xffef | spec x15 1 | f1:',
```

The other controls are more complex and are processed in a separate REXX filter, its output bypasses the fullscr and is fed to the remote host immediately.

```
  '? x3: | n: px3270c' options '| fx:',
```

The snapshot of the critical state variables is output through the secondary output, it is used to close the gate as soon as one of them drops back to 0.

```
  '? n: | nfind 1 1 1 1| f0:'         /* Shut off when not tn3270    */
return rc
```

nvt: Run the NVT session

The structure for the pipeline that runs the session in NVT mode is similar to the 3270 version.

```
nvt:
'callpipe (end ? name px3270m.nvt)',
    'f0: fanin | g: gate strict',
  '? *:  | g: |',
```

Again we need to separate the controls from the data and merge with the result of that processing.

```
    'x1: nlocate 2 |',
    'f1: faninany |',
```

We also have to reblock the records. The definition for NVT is that CRLF is used to separate the records.

```
    'deblock string x0d0a |',
    'split string x0d00 |',
```

The CR without LF is also allowed according to the RFC, this is probably from the days where they wanted to overstrike passwords when using printing terminals. Overstrike on a 3270 is less useful so we just interpret it as if it were a CR LF combination (i.e. split the record on it). It is good the RFC854 actually specifies the CR should be followed by either a LF or a NUL, just leaving out the LF would cause us to delay the record since you could not tell whether you had to wait for the LF until the next byte arrives.

Note: Recently I found my client not displaying the proxy prompt where I type in the remote host to connect to. It turns out the folks who maintain the proxy introduced the feature to position the cursor behind the prompt by suppressing the CR LF (it gets sent after echoing the name of the host to connect to). You will understand what happened with the check that would detect the first record containing the prompt and automatically replied the name of the host to connect to. This is still within the rules in the RFC but I doubt this was indeed the desired effect when they changed this (and most clients will not work that way either).

The final step is to translate the characters to EBCDIC using your favorite translate table.

```
    'xlate *-* from 437 to 1047 |',
```

We now have single records (in EBCDIC) with no control. The REXX stage “px3270l” builds proper 3270 data stream out of these records which can be written to the fullscr stage. be displayed on a

```
    'l: px3270l' qry '|',             /* Play line mode on 3270      */
    'fullscr' addr 'console path tn3270 condread asynch |',
    'g: |',                           /* Prepare to cut the loop     */
    'strnfind x02 |',                 /* Data read from console      */
```

The output from these stages are the commands typed at the screen. They are fed back to the “px3270l” to allow it to reset the keyboard and erase the input area.

```
    'elastic |',
    'l: |',
```

The secondary output of the “px3270l” stage has the commands entered at the screen in clean format (stripped off the AID byte and cursor position).

Note: The section above shows that careful selection of the function of primary and secondary output can simplify coding the topology of the multi stream pipeline (there are no pipeline end characters in the section). The section above is not just a loop in the pipeline but rather a salto...

Next thing to do is convert the code back to ASCII, add a CRLF, and output it to be sent to the host.

Note: I'm not really sure why I would need to escape any x'ff' bytes since it is not likely these are produced by the xlate and they are not allowed on the NVT anyway.

```
    'xlate *-* from 37 to 437 |',     /* Make ASCII                */
    'change xff xffff |',             /* Escape the FF             */
    'spec 1-* 1 x0d0a n |',           /* .. and add CRLF           */
    'fx: faninany |',
    '*:',
```

The following sections again process the simple Telnet controls like we did with the “tn3270” part.

```
  '? x1: | x2: locate 1.2 xffff | chop 1 | f1:',
  '? x2: | x3: locate 1.2 xffef | spec x0d0a 1 | f1:',
```

Note: The RFC does not specify the meaning of EOR in NVT mode (it's only used in 3270 mode) but it probably doesn't hurt to do something with it. The “px3270c” again processes the other controls until we have set all options that indicate the switch to 3270 mode.

```
  '? x3: | n: px3270c' options '| fx:',
  '? n: | frlabel 1 1 1 1| take 1 | copy | f0:'
return rc
```

initialize: Initialize some variables

The variable “options” lists the telnet options that are processed by the “px3270c” stage.

```
initialize:
options = '00 19 / 01 03 06 18'
```

The RFC's refer to them by their decimal number:

0 Binary Transmission (RFC 856)

When in binary transmission the sender and receiver have agreed not to use the limited NVT type of data (i.e. printable ASCII plus a few controls) but use the full 8 bit transparently.

25 End of Record (RFC 885)

This option indicates that each logical record is terminated by a special Telnet control (rather than the CRLF in ASCII data that is normally considered to be part of the logical record).

1 Echo (RFC 857)

Normally both parties of a connection do their own local redisplay of typed characters on the screen. The Echo option is used to have the other side produce that echo. When the redisplay of type characters is under control of the host it use that to suppress or mask a password.

3 Suppress Go Ahead (RFC 858)

The basic NVT connection is considered to be a half-duplex connection that requires a special Telnet control to switch between sending and receiving. It is common practice to ignore this part of the protocol and just assume the connection is full-duplex.

6 Timing Mark (RFC 860)

This must be replied to to allow the other party to check whether its peer is still alive. It may also be used to measure network delays. It does not matter whether the control is rejected or accepted, in either case it shows the client has received the control.

24 Terminal-Type (RFC 1091)

The option signals that the party is prepared to exchange information about the terminal type. The actual exchange is in sub-negotiation.

Note: Though I accept the “echo” and “suppress go-ahead” I have not coded any function for them.

The next variables are set to reflect the initial state of the various options.

```
opts. = 0                              /* Default options are off     */
togl. = 1
togl.18 = 0                            /* Terminal Type               */
togl.06 = 0                            /* Timing Mark                 */
```

Next we query the screen to prepare the terminal type we need to answer when the host asks for it.

```
'callpipe fullscrs' addr '| spec w1.4 1 | var qry'
parse var qry rows cols . ewa .
model = word('2 3 4 5', find('32x80 43x80 27x132', rows'x'cols)+1)
'callpipe literal IBM-3278-'model'| xlate e2a | var opts.this.18'
return
```

PX3270C REXX Negotiate Telnet Options

There are a number of different options the parties can agree on to use for the remainder of the Telnet session. There are four controls used for the negotiation process (do, don't, will, won't).

```
c_will = 'fb'x
c_wont = 'fc'x
c_do   = 'fd'x
c_dont = 'fe'x
```

The negotiation now is such that the other party can offer to use a certain option by sending a “will option” that is replied to with either “do option” to accept it, or with “don't option” to reject it. Alternatively one can request the other party to use a specific option by “do option” that is replied to with either “will option” or “won't option” depending on whether the other side supports the option.

The arguments passed to the stage are the options that can be negotiated, any other option requested by its peer is rejected.

The primary output of the stage consists of the telnet controls being produced by the negotiation process, the secondary output is a snapshot of selected states that allows the calling pipeline to determine when the required combination of states has been reached.

```
arg options                            /* Negotiatable options        */
opts. = 0                              /* Default options are off     */
togl. = 1                              /* Default negotiatable        */
call loadvars
do forever
  'peekto inp'
  if rc ¬= 0 then leave
  call process
  'readto'
end
call savevars
return rc * ( rc ¬= 12 )
```

loadvars: Load state variables from caller

The state variables are kept in the REXX environment of the caller so we copy them from the caller on entry using varfetch and a varload stages.

The list of options (passed as argument to this stage) is used to list the variables that need to be retained.

```
loadvars:
  'callpipe (name px3270c.loadvars) var options | split | nfind /|',
         'spec ,opts.this., 1 1-* n write',
              ',opts.that., 1 1-* n write',
              ',togl., 1 1-* n | xlate |',
         'varfetch 1 toload | buffer | varload'
return
```

savevars: Save state variables in caller

This is the reverse from the procedure to load the variables. On exit of the stage we copy the current values to the caller.

```
savevars:
  'callpipe (name px3270c.savevars) var options | split | nfind /|',
         'spec ,opts.this., 1 1-* n write',
              ',opts.that., 1 1-* n | xlate |',
         'varfetch toload | buffer | varload 1'
return
```

process: Process one telnet control

This is the code that does the actuall processing of Telnet controls, based on the flags set in various REXX variables.

```
process:
  key = c2x(substr(inp,3,1))
  c = substr(inp,2,1)
  rc = 0
  select
    when c = c_will then call sw_on  that, key, c_do  , c_dont
    when c = c_wont then call sw_off that, key, c_dont
    when c = c_do   then call sw_on  this, key, c_will, c_wont
    when c = c_dont then call sw_off this, key, c_wont
```

The “Terminal-Type” option is one that uses sub-negotiation (i.e. it has a variable number of parameters to pass from client to host). The section above has already agreed to do the sub-negotiation.

```
    when left(inp,3) = 'fffa18'x then
      do
        select
          when substr(inp,4,1) = '01'x then
             'output' 'fffa1800'x || opts.this.key || 'fff0'x
          when substr(inp,4,1) = '00'x then
          'callpipe var inp | spec 5;-3 | xlate a2e |',
                    'var opts.that.'key
          otherwise nop
        end
      end
    otherwise nop
  end
```

Now that we have possibly changed some state variables we need to output the current state so that the calling pipeline can determine whether we have either entered or left the 3270 mode.

```
   'callpipe (name px3270c.stateout) var options | chop / | split |',
          'spec ,opts.this., 1 1-* n write',          /* Specify all   */
               ',opts.that., 1 1-* n | xlate |',      /* variables we  */
          'buffer |',
          'varfetch | join * , , | *..1:'
return rc
```

sw_on: Switch an option on

This section deals with the attempt to enable an option. When the option is not in the list of supported options a “nak” is sent back.

```
sw_on: procedure expose opts. togl. options
  parse arg who, key, ack, nak
  if find(options, key) = 0 then 'output' 'ff'x || nak || x2c(key)
```

If the option is not yet on we acknowledge by sending the “ack” to the other side.

Note: An already enabled option is not acknowledged to prevent endless loops in the negotiation.

```
   else
     do
       if togl.key = 0 | opts.who.key = 0
         then 'output' 'ff'x || ack || x2c(key)
```

For those options that actually have a state to be maintained, the state variable is updated.

```
       if togl.key > 0 then opts.who.key = togl.key
     end
return rc
```

sw_off: Switch an option off

The process for dealing with the request to disable an option is somewhat similar to the one for enabling it. The major difference is that the request to disable the option can never be rejected.

```
sw_off: procedure expose opts. togl.
  parse arg who, key, ack, nak
  if opts.who.key > 0 then
    do
      opts.who.key = 0
      'output' 'ff'x || ack || x2c(key)
    end
return rc
```

PX3270L REXX Do line mode output on 3270

This stage allows lines of text to be written to a fullscreen 3270. The major part of it was copied from an example of John P. Hartmann.

The streams are used as follows:

input.0: Lines of data in EBDIC with no CRLF or EOR bytes

input.1: Feedback from the console stage (i.e. AID + data)

output.0: The 3270 DS to drive the console

output.1: Entered data as a single record without AID or cursor

Because of this the typical configuration for the stage is

```
... | l: line3270 qry | fullscr console asynch | l: | ...
```

The advantage now is that I can replace the fullscr with something else (e.g. using tcpdata) to get the display on something else than my virtual console.

The program starts by interpreting the parameters that describe the size of the screen.

```
arg rows cols . ewa .
rows = word(rows 24, 1)
cols = word(cols 80, 1)
bottom_line = rows-1
```

The screen uses an input are on the bottom line. The following code will erase the are after input was accepted (and to initialize the field).

```
clear_inp=  '11'x || d2c(bottom_line*cols, 2),          /* SBA     */
          || '1dc813'x ,                                /* SF & IC */
          || '3c'x || d2c(rows*cols-2, 2) || '00'x,     /* RBA 00  */
          || '1d60'x                                    /* SF      */
```

The loop will wait for a record on either input and read it in. When it is read on the primary input we got a new line to be written to the screen, the secondary input has the responses from the screen.

```
line.0 = 0; clearinput = 1            /* Force clear screen          */
stopping = 0
do until stopping
  'select anyinput'
  if rc ¬= 0 then leave
  'streamnum input *'
  s = rc

  'readto line'
  if rc ¬= 0 then leave
  if s = 0 then call nuline line
  else call command

  'streamstate input 0'
  if rc = 12 then exit rc

  call display
  'streamstate output 0'
  if rc = 12 then leave
end
error: return rc * ( rc ¬= 12 )
```

nuline: Add one line to the buffer

```
nuline:
parse arg add
If line.0 < rows-1
   Then 'callpipe var add|stem line. append'
   Else 'callpipe var add|preface stem line.|drop 1|buffer|stem line.'
return
```

command: Process command from the console

```
command:
parse var line aid+1 cursor+2 data
clearinput=1                          /* Clear input and reset       */
if aid ¬= '7d'x then stopping = 1     /* Any PF-key will stop        */
if left(data,1) = '11'x then data = substr(data,4)
'callpipe var data | *..1:'           /* Output the data             */
return
```

display: Write the new buffer to the console

```
display:
If clearinput
   Then clr='|append var clear_inp'
   Else clr=''
```

The following pipeline will take all lines from the array and build a proper 3270 data stream, positioning each record on the correct line.

```
'callpipe',
   '|stem line.',
   '|chop' cols,
   '|pad' cols '00',
   '|spec x11 1',
       'number from' (rows-line.0-1)*cols 'by' cols 'd2c 2.2 r 1-* n',
     clr,
   '|join *',
   '|change // x'substr('01c1', 1+clearinput*2, 2)'c2',
   '|*:'
clearinput=0
return
```

Conclusion

There are still a lot of things that should be done (though I am actually using this code to connect to VMSHARE and PUCC from a plain 3270 terminal).

- Reduce CPU usage, especially in the “tn3270” pipeline. I probably should review the part that needs the fblock 1 now.
- Handle the case where CP drops the console out of full screen mode (probably using a VMFCLEAR and simulate a CLEAR key to the host to signal we lost the screen).
- Add a PA1 escape to enter specific Telnet controls (e.g. to abort output).
- Allow for PC based TCP/IP connected clients. This would mean that I have to run the code in a separate userid, I might add a few GRAF devices to dial to as well.
- Make automatic dialogue with the proxy to connect to the host without manual intervention (though that's a waste of the nice code I made).

- Include the a call to John Hartmann's “socks” routines.

Writing this program gave me a better understanding of the way a pipeline terminates, on how end-of-file is propagated through the pipeline and how to deal with stalled pipelines.

While writing the code I have discarded it a few times when I felt it was getting far too complex to deal with (mainly the part that does the negotiation). When I cheated and looked at the C source for a Unix TN3270 program I noticed that really was much longer than mine.

Looking back at the basic plumbing involved in writing this, I think this certainly illustrates the power of the TCP/IP stages

Playing with the TCP/IP protocols showed me that you can indeed get any information you need from the Net, but some may be of less quality than you would expect.