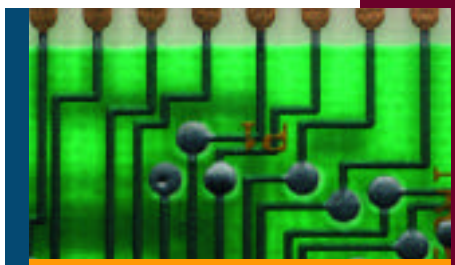
Everything You Need to Know About Socket Programming in a VSE Environment



BY LEO J. LANGEVIN

With TCP/IP socket programming, if you're used to writing code that can open, read, write, and close files, you can write a TCP/IP for VSE client or server using five basic commands.

SOCKET DENTISTRY

I live in a small town in Illinois, and if you don't have a local map, it's unlikely that you'll find Island Lake. Just on the outskirts of town is a small office with a sign that reads "Gentle Dental." I can count on Dr. Sherwin to dip a Q-Tip in some fluid that will numb my mouth and then wait a few minutes until he uses a syringe. I don't feel any pain. That's right, when it comes to pain, I'm a wimp and I'll avoid it at all costs. I would never go to a dentist who would say, "Well, it's a small cavity, so let's skip the Novocain." No way, no how.

So what does this have to do with socket programming?

There are basically two flavors of socket programming. The first flavor is a masochistic method that uses a "standard" called BSD (Berkley Software Distribution) sockets. This has been given to us by Berkley, the same folks who gave us the painful "vi" editor. As an example, some years ago, I was taking a C++ class at a local community college and we were forced to use a UNIX environment to test our projects. Although MS-Windows was far easier to use, we had to use the "vi" editor. The only woman in the class finally spoke out saying, "What is this, some sort of macho thing?"

The second flavor of socket programming uses TCP/IP and should be easy and simple to use. Indeed it can be, but for some unfathomable reason the UNIX techies have it in their head that if it's easy to use, there must be something wrong.

Unlike the straightforward TCP/IP programming, using the BSD "standard" forces the programmer to issue far more commands and operations than necessary. For example, we would normally say, "Let's drive to the dentist," but using BSD terminology we would have to say, "Let's go to the car, enable the car, load the car, and send the car with us to the dentist." All of the other operations are assumed in the first instance, but BSD forces you to state those commands again and again.

In short, BSD sockets can be moderately useful for taking a convoluted C-program and porting it to VSE, but as a developer or programmer, I would avoid its use.

THE PROBLEM WITH PLATFORM INDEPENDENCE

There is a legend that success comes from having the same code run on all platforms. The belief is that this will eliminate the need for knowledge about the operating system and reduce the number of people necessary to maintain the same product. This is fine in theory. The problem is that many of us have had actual experiences with, "Oh, this product was ported from MVS" or "This was originally a UNIX application that we ported to VSE." And our experience is that unless the program was modified to take advantage of the VSE operating system, it was difficult to use.

TCP/IP SOCKET PROGRAMMING IN A VSE ENVIRONMENT

Using the vendor product TCP/IP for VSE from Connectivity Systems, you can use the BSD-type of process. However, for those of you with a low pain tolerance, you really only need to know five commands to be an expert in TCP/IP application programming:

- **OPEN** - Establish connection with TCP/IP
- **CLOSE** - Terminate your TCP/IP connection

Figure 1: Starting the Server Program

```
SOCKET OPEN, TCP,LOPORT=666,
                  WAIT=YES,DESC=DESC,
                   PASSIVE=YES
```

Figure 2: Starting the Client Program

```
SOCKET OPEN, TCP,LOPORT=555,
                  WAIT=YES,DESC=DESC,
                   PASSIVE=NO,FOIP=IPADDR,FOPORT=666
```

Figure 3: Server Waiting for a Client Request

```
SOCKET RECEIVE,TCP,DESC=DESC,
              WAIT=YES,DATA=(POINTER,LENG)
```

- **SEND** - Transmit data to TCP/IP
- **RECEIVE** - Accept data from TCP/IP
- **ABORT** - Flush the data-transmission-in-progress

And there you have it; all of the commands that you need to know. This is all that you will ever use. Now wouldn't it be great if the UNIX world followed the VSE implementation? In this way, we can write TCP/IP applications in the same manner that we write application programs that use files. We open, read, write, and close files. If someone understands that type of processing, then writing a TCP/IP application is a snap.

USING DIFFERENT LANGUAGES

Using TCP/IP for VSE you can write socket code in most of the more popular languages. If you know CICS command-level programming, you know that you first pass the "EXEC CICS" commands through a pre-processor, which will convert these calls to the more intense instructions that will perform the actual activities. This is true for Assembler, COBOL, PL/I and so forth. With TCP/IP for VSE, you can use this pre-processor type of operation. It will convert the code for you, and you pass the output through the compiler just as you would with CICS.

We also use macro-level programming for assembler or a function level call for REXX. In this way, you can issue the five different commands in a manner that you are most comfortable with. Personally, I like the macro level because I code only in assembler and I find it relatively simple.

WHAT IS A SOCKET?

Back in the old days, I used to program an IBM 407 EAM (electronic accounting machine). The process was very complicated; we had a circuit board riddled with holes and each hole had a specific purpose. We would program this device by plugging a wire in one hole and then plugging the other end into another hole. This would complete the circuit. When we wanted to start a new application, we would pop out one board and insert a new one. This was an early form of socket programming.

When you start up TCP/IP for VSE in a partition, you give it an IP address, which is a special format of four numbers separated by dots. Think of this as the entire foundation from which you will be programming, as though it were a machine in itself. Every application needs to subscribe itself to TCP/IP so that other programs in the network can communicate with it. Think of a port number as a special number, just as a special board was used for each application program on that old accounting machine. There are standards out there so that port number 23 is usually used for telnet, and port 666 is reserved for playing DOOM over the Internet. If you won't be using these applications on VSE (well, maybe not DOOM), then you could always use those special numbers.

Now the socket part is like the holes in that programming board. It completes the circuit connection so that every other socket connection can be in communication, provided that there is a connection with that port number. Yes, you can have 20 different programs all using the same port number, but they all take up different sockets. If they are all the same program type, such as a dozen HTTP daemons,then there is not a problem. However, if you have several different programs that have different capabilities, then there can be a problem. Here's an example. Let's say you have a few HTTP daemons active on port 80. You use your web browser to access the web page from VSE. TCP/IP for VSE will look at the incoming request for port 80 and pass control to the first socket that is listening in on that port. If three HTTP daemons are active, then control will be passed to the fourth one.

If you write a program that performs email functions and decide to use port 80, it's possible that your email socket may receive a connection with some user's web browser, causing confusion and bedlam! So it is important to note that you should know the standard port numbers and use the non-standard ones when writing your own specialty programs.

LISTENING AT THE SOCKET LEVEL

When you are listening to a socket, you are waiting to receive data. You might be a server awaiting requests to process, or you might be a client that sent a request and is waiting for a response. When data comes in it might be a few bytes or several kilobytes in length. You might receive binary or text data. TCP/IP doesn't care. It just sends the data, and it's up to your program to make sense of it.

THE PROGRAM PROCESS

Let's say that you want to write an application for a server to wait for a connection, then process the request, send back a response, close the connection, and restart a connection. Let's run though this by creating a client and a server program as shown in Figure 1. The client code will be listed in italics. Again, you can do this in any language, but this figure shows what you would do at the macro level (assume the continuation settings on column 72!).

In Figure 1 we're saying, "Open a socket on port 666 and wait for a connection." The PASSIVE setting says to stay in a wait state until some connection occurs. We don't indicate a foreign port or an IP address because we don't care who connects to us. Next, the client starts up with the code shown in Figure 2. The client is going to be the active participant in this connection. By issuing this code, the socket request causes the ECB at the server to be posted. As a side note, the DESC is a full-word pointer that points to the socket control block that is allocated when an OPEN occurs and is deallocated when a CLOSE occurs. The IPADDR is a full-word that contains the IP address, so "100.100.1.1" would be stored as X'64640101'.

Now that we have a socket-to-socket connection (and notice that they can even be across different port numbers), the server issues a RECEIVE as shown in Figure 3. This will queue up a request. The full-word POINTER contains an address pointing to where the contents will reside. The full-word LENG indicates the maximum size of the buffer pointed to by POINTER. It's important to have a big enough buffer allocated. Generally, 8KB is often big enough.

Figure 4: Client Sending a Request

```
SOCKET SEND,TCP,DESC=DESC,
          WAIT=YES,DATA=(POINTER,LENG)
```

Figure 5: Client Waiting for a Server Response

```
SOCKET RECEIVE,TCP,DESC=DESC,
            WAIT=YES,DATA=(POINTER,LENG)
```

Figure 6: Server Sending Response and Resetting Connection

```
SOCKET SEND,TCP,DESC=DESC,
            WAIT=YES,DATA=(POINTER,LENG)
SOCKET CLOSE,TCP,DESC=DESC,
            WAIT=YES
SOCKET OPEN, TCP,LOPORT=666,
            WAIT=YES,DESC=DESC,
               PASSIVE=YES
```

Figure 7: Client/Server REXX Pseudocode

```
/*  SERVER  */
rc=SOCKET('TCP','OPEN',666)
rc=SOCKET(handle,'RECEIVE')
/*   perform some activity and make a response */
rc=SOCKET(handle,'SEND',response)
rc=SOCKET(handle,'CLOSE')

/*  CLIENT  */
rc=SOCKET('TCP','OPEN',555,'100.100.1.1',666)
rc=SOCKET(handle,'SEND',request)
rc=SOCKET(handle,'RECEIVE')
rc=SOCKET(handle,'CLOSE')
```

And if you haven't noticed yet, the type of "TCP" means to use the TCP protocol. This is non-translated data. You can also use such types as UDP (which is what NFS primarily uses), TELNET (terminal type of I/O with translation), CLIENT (to pass control to one of many of the TCP/IP for VSE clients, such as LPR) or FTP (file transfer I/O with translation). There are other types as well, but they are for more complicated issues.

The client will now send a request to the server by executing the code shown in Figure 4. This will cause the server to be posted. It will examine the length returned in the ECB and the data in the buffer. The data does not arrive to the server application until the RECEIVE has been successfully queued and the program is listening at the socket level. Until everything is ready, both sides are waiting. Once the data arrives, the ECB is posted at both the server and the client. The client will then issue a request to receive the response just as the server did. See Figure 5. It will wait until the server sends data back to the client, causing the ECB at the client to be posted. Figure 6 presents the server code.

The server will send back the data. The SEND waits until the data arrives at the listening socket at the client level. Once done, the ECB is posted on both sides.

The CLOSE request occurs, terminating the socket connection, once the client socket connection is severed either from TCP/IP or from the client issuing a CLOSE. The CLOSE also deallocates the socket control block and the DESC field is cleared.

The server re-opens the socket so that it can process another request. The socket can only process one request at a time, so it needs to terminate the session with one application, sever its connection with TCP/IP, and then re-establish its connection, awaiting another request. And of course, the client would finish the session with the following code:

```
SOCKET CLOSE,TCP,DESC=DESC,
             WAIT=YES
```

Note: If I were going to write this in REXX, it would look something like Figure 7.

SUMMARY

Writing socket code isn't that hard. Granted, with the use of the BSD socket "standard" it does boggle the mind, but with TCP/IP socket programming, if you're used to writing code that can open, read, write, and close files, you can write a TCP/IP for VSE client or server. ts

NaSPA member Leo J. Langevin has been involved in the VSE community since its inception. He is employed by Connectivity Systems and is the lead developer of NFS for VSE. He can be reached at leo@tcpip4vse.com.