How to Access Your Online Data From a Web Page

BY LEO J. LANGEVIN

In the November 1996 issue of ***Technical Support,*** I examined how to write a Common Gateway Interface (CGI) so that a web page can invoke a native VSE program, which can dynamically build HTML (Hyper Text Markup Language) commands and pass them back to the user. A typical example of this is a web page counter that you often see (e.g., "Welcome, you are visitor 000123").

But what if you want to access your corporate database? Wouldn't it be nice to be able to have a web page pass parameters to your CGI,and then have the CGI pass the query to your CICS, IDMS, or other form of transaction processor? Personally, I think it's a really cool idea.

Why? Well, by using this process, we can provide simple, yet graphically appealing menus, without the back-breaking work that most screen scrapers require. What's more, you don't have to worry about client programs, changes behind the screens or any other nonsense because none of that is necessary any more.

The TCP/IP system that I will be talking about this month is TCP/IP for VSE from Connectivity Systems. Because of its native implementation of the TCP/IP stack, providing a CGI interface into any area of VSE becomes a simple task.

There are several methods that you can invoke to have a program build web pages. The first is to have a program update the VSE HTML source library whenever data changes. For example, you could have a batch job run nightly to build a page to show what shipments went out the night before. Such an application does not require real-time data queries, since the data will never change.

For queries where you want to see some specific information, such as the price of pork bellies in 1987 during the third week of May, you really want to use your transaction processor to perform the query for you, and then return the result in a method similar to a 3270-session, but with a completely different look.

Wouldn't it be nice to be able to have a web page pass parameters to your CGI, and then have the CGI pass the query to your CICS, IDMS, or other form of transaction processor?

This brings us to the idea of client/server. In a client/server relationship, the client makes a request, and the server responds. In this example, the user running Windows has a client called Netscape. The Netscape client talks to the HTTP server. The HTTP server sees that a CGI is being invoked and passes control to the CGI. The CGI establishes communication with the online processor program, which performs the request and returns the data to the CGI, which, in turn, returns the data to the HTTP server, which returns the data to the Netscape client. As you can see, the role of client and server can flip-flop.

In the following example, I will be using the standard XPCC interface. Typically, you can use a number of different communication methodologies. Personally, I prefer to use UDP/IP in such an application, but that would require a great deal more explanation than I can provide here. If there is a great deal of interest in this area, perhaps a future column on this subject will be provided.

WHAT IS XPCC?

XPCC is a programming interface that allows programs to exchange data among themselves. Typically, we use this when we want an application program to talk to POWER. Information in this area can be found in the *IBM VSE/POWER Application Programming Guide* (SC33-6574-01) as well as the *IBM VSE/Advanced Functions Diagnosis Reference Supervisor* (LY33-9172) manual. The first resource provides a rudimentary understanding, since it only relates to communicating with POWER. The second manual goes into even greater depth. Finally, IBM has also provided an example of communicating with POWER in IJSYSRS.SYSLIB in a member named PWRSASEX.A.

So how do you code an XPCC client and server? First, you will need to code the XPCC control block using the XPCCB macro. This control block contains several fields that are needed during the process of your XPCC operations. The MAPXPCCB macro will generate a DSECT that will map the layout of this control block. Figure 1 shows a typical way of coding the XPCCB in a program. In this example, we point to the XPCCB and then map it to the DSECT layout so that we can refer to the various fields by field name instead of specific offset addresses. Later on, we define the XPCCB. Not all of the fields are necessary when defining this control block, however, the first two (APPL and TOAPPL) are required.

The APPL field defines the name of the application. It must be a unique name that will be registered with the XPCC interface when the program identifies itself to the system. If there is already a program active with the same identifier, then your request will fail. Therefore, you must determine how you will be forcing this uniqueness. In our case, however, there will only be one CGI asking the request (although a number of different HTML pages might use the same CGI), and one program accepting the response, so we won't worry about providing a unique name (you can always insert the partition ID, task number and any other unique information as part of the name). Just remember that you cannot have blanks inside of your name.

The TOAPPL will define the name of the other side of the connection. If you are the client, then you must know the server's name. If you are the server, then you don't need to know who is going to connect to you, and you should use the value of ANY. In this way, a server could receive multiple requests from an assortment of outside programs (just as POWER does) by having TOAPPL=ANY and waiting for a connection to occur. In our example, the CGI will have an APPL=CGICICS1,TOAPPL=CICSCGI1. The CICS program, however, will have APPL=CICSCGI1,TOAPPL=ANY. Remember, if you don't use CICS at your shop, you can use whatever naming convention works best for you.

This is all that it takes to have a web page driven by your transaction processor — two simple programs that use a VSE interface that have been around for quite some time.

Since we want to exchange data between the two programs, we will also define a buffer and a buffer length. The BUFFER parameter defines the sending buffer, while the REPAREA defines the area that will receive the reply. In our example, CGICICS1 would have BUFFER=(BUFF,8) to send 8 bytes of data to the recipient and REPAREA=(REPLY,256) to be able to receive up to a 256-byte response. If your buffer is too small, and you issued a SENDR request, you will receive the truncated reply with an error indicator.

There are three methods of sending data to the other side:SEND, SENDR and SENDI. Each of these provide a method for XPCC to send data to the other application program. SEND simply sends the data and does not wait for a response. SENDR sends the data, and requires a reply from the recipient. SENDI works like SENDR, except that if the receiving buffer is not large enough, you can issue multiple RECEIVE requests to get the rest of the data. After all of the data has been retrieved, the replying program posts that the transfer is complete.

Speaking of posting, there are three ECBs in the XPCCB: the SEND ECB (IJBXSECB), the RECEIVE ECB (IJBXRECB) and the CONNECT ECB (IJBXCECB). As with any ECB, these fields are a fullword in length, and when the high order bit is set at +2 from the beginning of any of these fields (e.g., "TM IJBXRECB+2,X'80'" would test if any data has yet been received) your program can take action. Your program would then either issue a WAIT on any or all of these ECBs or perform some other processing and check for a POST periodically. In CICSCGI1,since its only purpose is to wake up when there is a request, process the request, and then go back to sleep, it will do a WAIT on the CONNECT ECB.

When IJBXCECB is posted, then there is a request from another application to communicate with CICSCGI1. In a simple application, such as CGICICS1, we could issue a CONNECT to CICSCGI1 when the CGI is initially opened, and keep the connection open for the lifetime of the HTTPD. This is because CGICICS1 is the only application that will communicate with CICSCGI1. For example, let's say we decided to add a batch interface as a second program that had the same APPL name as CGICICS1. For this second program to be able to communicate with the CICS program, it would be best to continuously CONNECT and DISCONN between requests to provide an opening for the other program. This also has the additional benefit of not needing to delete and define the CGI in case CICS is brought down and the CICSCGI1 program is no longer available.

There are 15 different functions available to an XPCC program:

- IDENT: Register with XPCC as an application;
- CONNECT: Link to another XPCC user;
- SEND: Send data — no response;
- SENDR: Send data and expect a reply;
- SENDI: Send data, expect a reply, and allow multiple RECEIVE;
- REPLY: Reply to data that was received;
- RECEIVE: Receive data;
- CLEAR: Purge data that is being sent;
- PURGE: Purge data that is being received;
- DISCONN: End connection after the exchange is over;
- DISCPRG: End the connection now!;
- DISCALL: End all connections;
- TERMIN: Remove link to XPCC when it's ready;
- TERMPRG: Remove link from XPCC right now!; and
- TERMQSCE: Schedule an XPCC drop.

Now that we have the XPCC information, let's go through a sample session between the Netscape user and the CICS program:

1. The Netscape user enters: "HTTP://hostname/INQUIRY.HTML".

2. The HTTPD goes into the VSE library, reads INQUIRY.HTML and sends it back to the user.

Figure 1: Coding the XPCCB in a Program

```
              ...
              LA   R3,XPCCB
              USING IJBXPCCB,R3
              ...
       XPCCB  XPCCB  APPL=applname,                  X
              TOAPPL=[applname|ANY],                 X
              [BUFFER=(addr|addr,len),               X
              REPAREA=(addr,len),                    X
              MECB=[addr|(reg)],                     X
              VERSION=1|2
              ...
              MAPXPCCB DSECT
              ...
```

3. The user sees a screen of information. There is also a request field. "10/12/95" is entered into the date field and the SUBMIT button is pressed. This button is defined to invoke CGICICS1. The input field is defined as &DATE. Obviously, this form is not year 2000 friendly!

4. The HTTPD receives the request "CGICICS1?&DATE=10%2f12%2f95 &SUBMIT". The "/" is automatically converted into ASCII HEX "%2f". The name of the SUBMIT button is appended to the rest of the request.

5. CGICICS1 is invoked, passing control into its OPEN section, where it issues an IDENT to XPCC with TOAP-PL=CICSCGI1.

6. Now back at the CICS side CICSCGI1 is active and waiting. It had already performed an XPCC IDENT, with its own unique APPL name and used the parameter "TOAPPL=ANY". It executed a CONNECT request, allowing incoming connections to occur, and was waiting for an outside program to connect to CICSCGI1. When CGICICS1 issued a CONNECT, the system set the CICSCGI1 IJBXCECB posting bit, and CICSCGI1 "woke up" and acknowledged the connection by issuing a RECEIVE. Now CICSCGI1 waits for data to come from the outside program and into it's receiving buffer.

7. CGICICS1, upon receiving acknowledgement from XPCC that the connection succeeded, issues a SENDR of 8 bytes of data, which had been converted to "10/12/95". It then waits for a response against the RECEIVE ECB.

8. CICSCGI1 wakes up, receives the data, and performs a specific process. In our example, it verifies the date. This obviously should be expanded to perform a database query, but I'll leave that type of logic up to you!

This is all that it takes to have a web page driven by your transaction processor — two simple programs that use a VSE interface that have been around for quite some time.

9. CICSCGI1 builds a response and issues a SEND. Since CICSCGI1 did what it was designed to do, it issues a DISCONN, which is queued.

10. CGICICS1 wakes up, accepts the response from CICSCGI1, and builds a series of HTML commands and text that are kept in storage. It then returns control to HTTPD.

11. HTTPD is ready to read some HTML data, so it passes control back to CGICICS1's READ section. The CGI reads the formatted lines from storage that it built during OPEN processing and keeps sending them until there are no more to read. Between each READ, control is passed back to HTTPD, which accepts the line, sends it back to the Netscape user, and then passes control back to the CGI which repeats the process. When there are no more lines to send, the CGI signals the HTTPD via a return code, and passes control back to the HTTPD.

12. The HTTPD acknowledges the termination of the CGI and passes control to CGICICS1's CLOSE section. Here it issues an XPCC DISCONN.

13. CICSCGI1 wakes up and the disconnection is complete. It then resets a few fields and issues a CONNECT once more, waiting for another session to occur.

14. CGICICS1 issues a TERMPRG request, dropping its identity from XPCC.

This is all that it takes to have a web page driven by your transaction processor — two simple programs that use a VSE interface that have been around for quite some time. To the user, it's a seamless process; all they know is that they enter a command on the web page and they see data a few moments later. There is no need for using a VTAM programming interface, an AIX processor, or any other stage that can slow down this process or add significant costs to implementing an online interface. ts

NaSPA member Leo J. Langevin is a systems programmer with Connectivity Systems, creators of TCP/IP for VSE. His life currently revolves around RPC, UDP, NFS and a lot of other three-letter words. He can be reached via email at leo@tcpip4vse.com.