Practical Ports And Serviceable Sockets

By Nick Decker

We've written about using TCP/UDP sockets in PRO/5®, Visual PRO/5® and BBj® before, but as time goes on, we invariably think of new ways to use them. On the BASIS demo CD, for example, we include server and client programs that utilize sockets to communicate with one another. These programs do a good job demonstrating sockets in general, how to write client and server programs, and even give an example of a user-defined protocol between the two. This article takes sockets in a different direction - writing a socket client that communicates with pre-established servers using pre-defined protocols. The two programs discussed in this article function as clients to a POP mail server and to an FTP server.

Although you probably don't want to reinvent the wheel and write your own client applications, these examples can still be useful, as they demonstrate how BBj interacts with networked servers in ways not previously covered. They also explain how to incorporate some of the basic functionality of more complex client applications into your application. For example, you may not want or need all of the bells and whistles provided by a full-blown web browser or FTP client. However, you could use BBj programs similar to these examples to provide a means for your application to query your web or FTP site and to determine if updates are available. If they are available, you can arrange for an automatic download. Features such as these enhance your application in many ways, and the fact that all of the functionality is integrated into your application is even more appealing. There's no need to rely on specific client software to exist on a workstation, and no distracting third-party applications will pop up on the user's desktop.

Additionally, these program examples provide insight on how to use sockets in BBj in a variety of different ways, and perhaps in ways you never considered. With sockets you can communicate with almost any TCP or UPD-based server, regardless of whether it's a web server, FTP server, mail server, etc. By accessing servers like this, you can send and retrieve email, download web pages and files, obtain news and weather information, get stock quotes, track packages, and more!

Before digging into the programs themselves, it's worth mentioning that they're fairly basic client implementations. They are not fully-featured nor do they do much in the way of error handling, as that would add to their complexity and detract from their value as learning tools. They also take advantage of some BBj-specific features, such as symbolic labels (*NEXT) and the MASK function's regular expression operators for this same reason. However, they can be adapted to run in PRO/5 with little effort. There are also some links at the end of this article that provide all of the necessary information regarding the POP and FTP protocols.

pop.lst Program

The **pop.lst** program (**Figure 1**) is fairly straightforward due to the simplistic nature of the POP3 protocol. As a brief overview, it establishes a TCP connection to the mail server, sends the desired login and password, and then retrieves the account's number and size of unread mail messages. All of the communications between the BBj socket client program and the POP mail server are done via an alternating exchange of commands and responses.

The **pop.lst** program initiates the connection to the server by opening a TCP socket connection. This is accomplished with the OPEN statement to the "N0" alias line. The "N0" alias line exists in the config.bbx file as:

```
ALIAS N0 tcp "" NODELAY
```

Because the alias line is generic, the OPEN statement includes the hostname of the mail server and the port on which you'll be talking. Port 110 is the default for POP-based mail servers, so that is what is used in this example.

Once the connection is opened to the server, the program enters the REPEAT-UNTIL looping construct that is responsible for guiding the subsequent conversation between the two. Because the server sends a greeting upon connection, the loop starts off with an INPUT from the socket channel. This reads in data that was sent by the mail server and assigns it to the tcpData$ string variable.

The next step is to analyze what was returned by the server. The POP protocol dictates that the server will always preface messages with a "-ERR" in the case of an error and a "+OK" for everything else. Therefore, the program does a quick check to see if the server returned a POP error or not. If there was a problem, such as an incorrect password for the specified login, the program prints out the error message, closes the socket connection and exits. In all other cases, the "+OK" is assumed, and the program continues.

Since the communication between the client and server follows a structured series of events, the program goes from event to event via a series of numbered states. This lends itself nicely to the SWITCH verb and makes the code easy to follow. As the program executes, it follows the pattern of sending a command to the server, verifying that it got a successful response, and moving on to the next command.

The command of particular interest is the STAT command. It instructs the server to respond with the total number of mail messages on the server and the size of the messages. The format of the response will be "+OK xx yy", where xx is the number of messages and yy is the total size of the messages. These numbers are parsed out of the return message and printed out. Once this has been done, the program sends a QUIT command to the server.

There's more to the POP protocol than demonstrated in this article, and the program could easily be enhanced to download e-mail messages and even delete them from the server.

Figure 1. pop.lst listing:

```
REM Initial Setup
dim mail$:"Description:C(30),Server:C(30),Login:C(30),Password:C(30)"
mail.Description$ = "Work Account"
mail.Server$ = "myserver.mycompany.com"
mail.Login$ = "mylogin"
mail.Password$ = "mypassword"
POP3_CONNECT = 0
POP3_USER = 1
POP3_PASS = 2
POP3_STAT = 3
POP3_QUIT = 4
POPState = POP3_CONNECT

REM Open a socket connection to the Mail Server
tcpChan = unt
open(tcpChan,mode="host=" + cvs(mail.Server$,3) + ",port=110",err=noServerAvailable)"N0"
print "Connected to mail server '"+ cvs(mail.Server$,3) + "'..."

REM Communicate with the host
Repeat

        input (tcpChan) tcpData$
        if tcpData$(1,1) = "-" then

          REM We got an error from the POP server
          print "POP error: ",tcpData$
          goto closeConnection

        else
```

```
    REM Since we didn't get a -ERR, we must have a +OK response
    switch POPState

      case POP3_CONNECT
        REM Send the server the login information
        POPState = POP3_USER
        print(tcpChan) "USER " + cvs(mail.Login$,3)
        print "Sent login information..."
        break

      case POP3_USER
        REM Send the server the password information
        POPState = POP3_PASS
        print(tcpChan) "PASS " + cvs(mail.Password$,3)
        print "Sent password information..."
        break

      case POP3_PASS
        REM Request message information
        POPState = POP3_STAT
        print(tcpChan) "STAT"
        print "Requested statistics..."
        break

      case POP3_STAT
        REM Ensure we got a valid string back from the server
        if (mask(tcpData$,"^\+OK\s\d+\s\d+")) then

          pos1 = mask(tcpData$,"\s\d+\s")
          print "There are" + tcpData$(pos1,tcb(16)),
          print "unread email messages totaling ",
          print cvs(tcpData$(pos1+tcb(16)),3) + " bytes"

        endif
        POPState = POP3_QUIT
        print(tcpChan) "QUIT"
        print "Sent quit command..."
        break

      swend

    endif

until POPState = POP3_QUIT

REM Closing connection to server
closeConnection:
close(tcpChan)
print "Connection to server closed."
end

REM Connection failure
noServerAvailable:
print "Failed to connect to the mail server '" + cvs(mail.Server$,3) + "'."
end
```

```
Output of the pop.lst program
=====================================================
Connected to mail server 'myserver.mycompany.com'...
Sent login information...
Sent password information...
Requested statistics...
```

```
There are 2 unread email messages totaling 2567 bytes
Sent quit command...
Connection to server closed.
```

ftp.lst Program

The **ftp.lst** program (**Figure 2**) is very similar to the **pop.lst** program in **Figure 1**. It also establishes a connection to a server and exchanges a series of commands and responses. However, the FTP protocol is a bit more involved as it requires two sockets - one for commands and one for data.

The program starts out with initialization, which involves creating the empty target file on the local drive. In this example, the target file is downloaded from the FTP server, and it is the Windows port of BBj from the nightly builds section on the BASIS FTP site.

Once the login and password are sent and accepted, the server responds with a welcome message which can be several lines long. The program looks for a "230" in return, which signifies the acceptance of the login and the last line of the welcome message. After the program logs in, it changes to the desired download directory, specifies a binary download type, and gives the server the PASV command. This command tells the server to go into passive mode, which means that it will determine a free, non-default port to listen for the data connection. This allows the program to make another TCP socket connection to the FTP server and download the desired file from it on that port. The key here is that the FTP server is still acting as the TCP server, and the program is still acting as a client to the server. If the server were not put into passive mode, it would require the program to act as a TCP server on the default port, and the FTP server would then act as a client, connect to the port, and send the data.

The next step is to request the file via the RETR command and determine the port on which it should connect to the FTP server for the transfer of data. The port is computed using the return message from the PASV command which is in the format of:

227 Entering Passive Mode (h1,h2,h3,h4,p1,p2)

The h's specify the internet host address, and the p's specify the data port. p1 is the high order byte, and p2 is the low order byte of the 16-bit port, so the decimal form is calculated by multiplying p1 by 256 and adding that value to p2.

Once the port is constructed, the program opens a second TCP socket connection to the same server on the new port. Because this port is dedicated to the transfer of data instead of commands, the program immediately goes into a read/write loop. It reads the data from the byte-oriented channel and writes it directly out to the target file on the local drive. When it hits an end-of-file on the read, it closes the two channels, cleans up and exits. The result is an exact copy of the 2166000.exe file that exists on the BASIS FTP server.

Figure 2. ftp.lst listing:

```
REM Initial Setup
dim ftp$:"Description:C(30),Server:C(30),Login:C(30),Password:C(30)"
ftp.Description$ = "BASIS FTP Server"
ftp.Server$ = "ftp.basis.com"
ftp.Login$ = "anonymous"
ftp.Password$ = "me@mycompany.com"
FTP_CONNECT = 0
FTP_USER = 1
FTP_PASS = 2
FTP_TYPE = 3
FTP_PASV = 4
FTP_LIST = 5
FTP_RETR = 6
FTP_QUIT = 7
FTPState = FTP_CONNECT

REM Setup downloaded file
erase "\2166000.exe",ERR=*NEXT
string "\2166000.exe"
```

```
fileChan=unt
open(fileChan,isz=-1)"\2166000.exe"

REM Open a socket connection to the FTP Server
tcpChan = unt
open(tcpChan,mode="host=" + cvs(ftp.Server$,3) + ",port=21",err=noServerAvailable)"N0"
print "Connected to FTP server..."

REM Communicate with the host
Repeat

  input (tcpChan) tcpData$
  if len(tcpData$)=0 then FTPState = FTP_QUIT

  switch FTPState

    case FTP_CONNECT
      REM Send the server the login information
      FTPState = FTP_USER
      print(tcpChan) "USER " + cvs(ftp.Login$,3)
      print "Sent login information..."
      break

    case FTP_USER
      REM Send the server the password information
      FTPState = FTP_PASS
      print(tcpChan) "PASS " + cvs(ftp.Password$,3)
      print "Sent password information..."
      break

    case FTP_PASS
      REM Bypass message information, CD to the BBj-Developer area
      if (pos("230 " = tcpData$)) then
        FTPState = FTP_TYPE
        print(tcpChan)"CWD /private/bbj-developer/current/windows"
        print "Changed directory..."
      endif
      break

    case FTP_TYPE
      REM Set binary transfer
      FTPState = FTP_PASV
      print(tcpChan)"TYPE i"
      print "Forced binary transfer mode..."
      break

    case FTP_PASV
      REM Tell the server to give me a port to retrieve the file on
      FTPState = FTP_RETR
      print(tcpChan)"PASV"
      print "Forced passive mode..."
      break

    case FTP_RETR
      REM Initiate file transfer
      FTPState = FTP_QUIT
      print(tcpChan)"retr 2166000.exe"

      REM Determine which port the Server specified as the data port to retrieve the file
      dataPort$ = tcpData$(mask(tcpData$,"\d+,\d+\)"),tcb(16)-1)
      dataPort$ = str((num(dataPort$(1,mask(dataPort$,",")-1)) * 256) +
num(dataPort$(mask(dataPort$,",")+1)))
```

```
        print "Server specified port ",dataPort$
        print "Retrieving 2166000.exe",

        REM Open up a new connection to the server on the data port and download the file
        dataChan = unt
        open(dataChan,mode="host=" + cvs(ftp.Server$,3) +
",port="+dataPort$,err=noServerAvailable)"N0"
        retrieveFile:
          read record(dataChan,siz=-10240,end=retrieveComplete) temp$
          write record(fileChan) temp$
          print ".",
          goto retrieveFile
        retrieveComplete:
        print " "
        print "File transfer complete!"
        close (dataChan)
        close (fileChan)
        break

      case FTP_QUIT
        REM We're done - send the server the quit command.
        print(tcpChan) "QUIT"
        print "Sent quit command..."
        break

    swend

until FTPState = FTP_QUIT

REM Closing connection to server
closeConnection:
close(tcpChan)
print "Connection to server closed."
end

REM Connection failure
noServerAvailable:
print "Failed to connect to the ftp server '" + cvs(ftp.Server$,3) + "'."
end
```

```
Output of the ftp.lst program
=====================================================
Connected to FTP server...
Sent login information...
Sent password information...
Changed directory...
Forced binary transfer mode...
Forced passive mode...
Server specified port 52966
Retrieving 2166000.exe.....................................................
.....................................................................
.....................................................................
.................................................
File transfer complete!
Connection to server closed.
```

Sockets offer a great deal of potential when it comes to communicating with other machines, servers, and programs all over the world. Because they can do so much, it is sometimes difficult to imagine the many ways in which sockets can be used to enrich your applications. These sample programs will hopefully spark your imagination and give you ideas on new ways to use sockets to their fullest potential.

The code in this article is available on the Advantage CD and also on our web site at:
www.basis.com/devtools/coolstuff/index.html

For further information on the POP and FTP protocols, see: