TCP/IP Socket Programming with REXX

Christian Michel
REXX Development
IBM Germany
German Software Development Laboratory
Boeblingen, Germany
e-mail: cmichel@vnet.ibm.com

Date of document: February 28, 1997

TCP/IP Socket Programming with REXX

by Christian Michel, REXX Development, IBM Germany
document loaded from REXX tutorials homepage at http://www2.hursley.ibm.com/rexxtut/

Introduction

The TCP/IP protocol stack is now available for a long period of time (in computer business terms at least) but in the last few years it has become more and more important for the communication between individual computers, especially since the evolution of the rapidly growing World Wide Web applications. Many applications have since been developed on various platforms but how often have you searched for a tool matching your requirements but never found it? This gap in the list of tools could be closed with programs written by yourselves - and this is where REXX comes into the game.

REXX is a very simple programming language that can be used for a large variety of purposes. It's easy to learn and has a number of useful function packages. One of these function packages is the REXX socket library available on different platforms which will be covered in this tutorial. We will learn the basic concept in TCP/IP communications and how you can communicate with other computers using REXX. One chapter will cover a higher level protocol used on the TCP/IP protocol stack: **HTTP** (HyperText Transfer Protocol). This chapter will give you the basic knowledge of writing TCP/IP socket programs in REXX. With this knowledge it will be easy for you to expand the sample programs and adapt them to your own needs. This tutorial expects basic knowledge of the REXX programming language and will only explain TCP/IP related topics.

TCP/IP Basics

One of the APIs to the TCP/IP protocol stack available is the **socket API**. A socket is a special type of file handle which is used to transfer data between two parties. The data that is written on one side of the socket can immediately be read on the other side. The transfer between the two parties is handled transparently by the underlying TCP/IP. Various applications have defined high level protocols on this socket interface for their purposes.

A client needs to follow the following steps to access a server via sockets:

- Create a socket with a call to **socket()**.
- Connect the socket with the server with a call to **connect()**.
- Do iterations of sending and receiving data with calls to **send()** and **recv()**.
- Shutdown the connection with a call to **shutdown()** and **close()**.

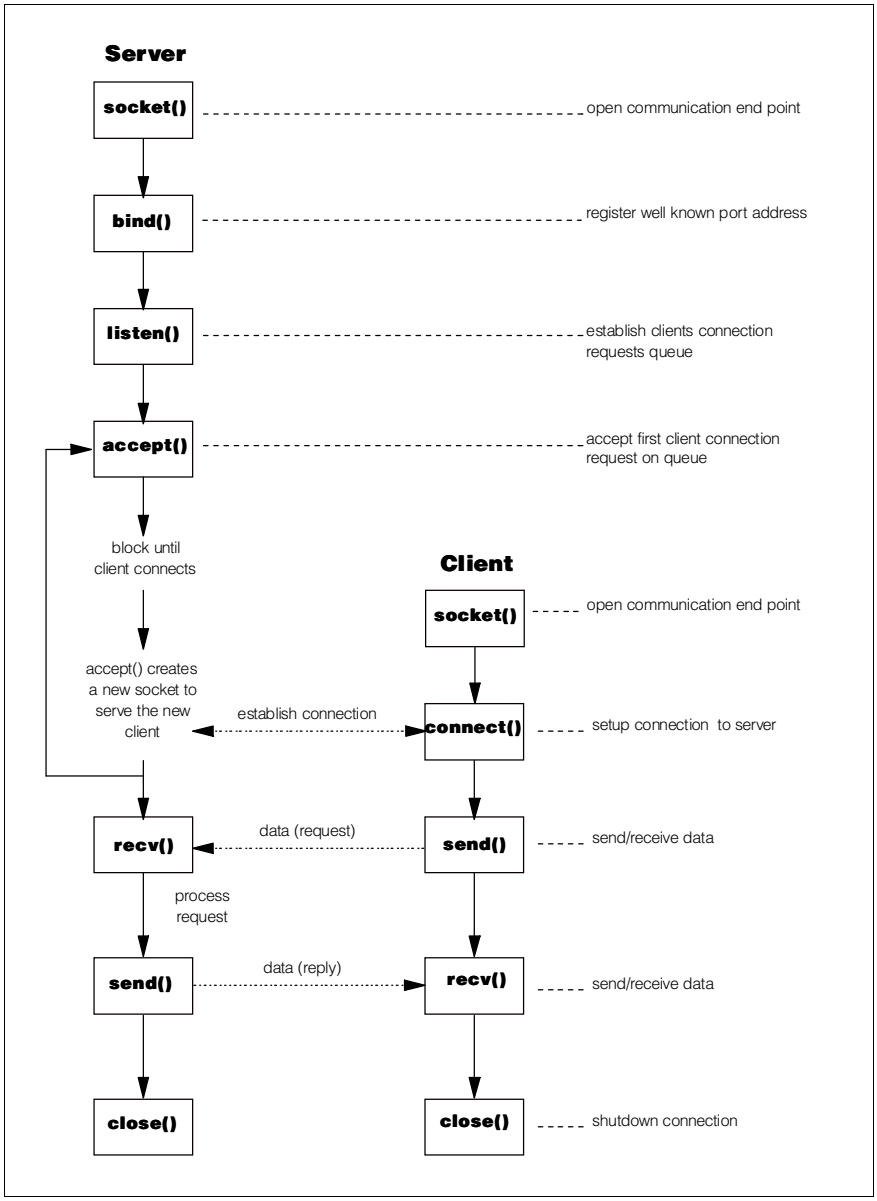
A server needs some preparation steps before it can accept clients:

- Create a socket with a call to **socket()**.
- Register the socket with a well-known port address with a call to **bind()**.
- Create a queue where clients can place connection requests with a call to **listen()**.
- Accept client request with a call to **accept()**.

The following graphic illustrates this flow:

TCP/IP Socket Programming with REXX

by Christian Michel, REXX Development, IBM Germany
document loaded from REXX tutorials homepage at http://www2.hursley.ibm.com/rexxtut/



The REXX Socket APIs

All the function calls mentioned in the previous section have their equivalent in the REXX socket APIs. The API for the OS/2 platform is either part of TCP/IP Version 3.0 as RXSOCK.DLL (which is included in OS/2 Warp Connect or OS/2 Warp V4) or can be found in the collection of IBM's employee written software (EWS). This EWS package is available from different locations in the Internet, e.g. from:

- ftp.pc.ibm.com, file /pub/pccbbs/os2_ews/rxsock.zip
- ftp.cdrom.com, file /os2/ibm/ews/rxsock.zip

TCP/IP Socket Programming with REXX

by Christian Michel, REXX Development, IBM Germany
document loaded from REXX tutorials homepage at http://www2.hursley.ibm.com/rexxtut/

The package contains a description of all available functions, so this tutorial will only list the functions that are used in the sample programs.

The same package (with exactly the same usage) is also included in Object REXX for Windows 95 and Windows NT.

The REXX/SOCKETS function package for VM was developed at the City University of New York and is available through different channels:

- BITNET: TELL LISTSERV at CUNYVM GET RXSOCKET $PACKAGE
- Internet: <http://ua1vm.ua.edu/~troth/rickvmsw/rickvmsw.html> for an overview of packages available from this server (the '1' in the server is digit 'One'!)

The VM function package has many differences in the syntax used to work with sockets within REXX. The basic functionality however is the same as for the OS/2 and Windows packages. This tutorial will use the OS/2 implementation as a reference and list differences for the VM application at the end in a separate topic.

The general TCP/IP function calls used in the previous section translate into the following REXX socket library function calls:

TCP/IP function	REXX function
socket()	SockSocket()
bind()	SockBind()
listen()	SockListen()
accept()	SockAccept()

TCP/IP function	REXX function
connect()	SockConnect()
send()	SockSend()
recv()	SockRecv()
close()	SockClose()

This tutorial will also use the following functions from the REXX socket library:

- SockGetHostByName() - resolve a TCP/IP alias to a dotted IP address.
- SockGetHostId() - find out local dotted IP address.
- SockGetPeerName() - find out address of communication partner.
- SockShutDown() - shutdown and cleanup a socket before closing it.

A simple mirror server and client

With the above information we can already write the first small socket application in REXX: a mirror server with a matching client program. The mirror server receives a string from the client, reverses the order of the characters in the string and returns it back to the client. After returning the mirrored string the communication will be closed down.

Mirror client

Let's have a look at the client program first:

```
/* MCLIENT.CMD - IBM REXX Sample Program                    */
/* Parameters:                                              */
/*   Server: alias name of mirror server                    */
```

```
Parse Arg Server

/* Load REXX Socket library if not already loaded       */
If RxFuncQuery("SockLoadFuncs") Then
 Do
   Call RxFuncAdd "SockLoadFuncs","RXSOCK","SockLoadFuncs"
   Call SockLoadFuncs
 End

/* Ask user for string to send to the mirror server     */
Say "Please enter a string that should be mirrored"
Parse Pull InpString

/* create a TCP socket                                  */
Socket = SockSocket("AF_INET", "SOCK_STREAM", "0")

/* resolve server name alias to dotted IP address       */
Call SockGetHostByName Server, "Host.!"

/* connect the new socket to the specified server       */
Host.!family = "AF_INET"
Host.!port = 1996
Call SockConnect Socket, "Host.!"

/* send the input string to the mirror server           */
Call SockSend Socket, InpString

/* receive answer from mirror server and close socket   */
Call SockRecv Socket, "OutString", 256
Call SockShutDown Socket, 2
Call SockClose Socket

Say "The string '" || InpString || "' was mirrored to"
Say "'" || OutString || "'."
```

The client program first loads the REXX socket library if it is not already loaded (which is checked with a call to 'RxFuncQuery'). The socket is created after the user has been asked for the string that will be sent to the mirror server. The parameters used on the 'SockSocket' call will create a default '**stream type**' socket. Stream type sockets support full duplex communication between two partners. There are other socket types, e.g. 'datagram' or 'raw' sockets which operate connectionless by sending only messages to the recipients. These socket types will not be discussed in this tutorial.

The new socket is not connected initially. To connect it to another workstation we have to call 'SockConnect'. 'SockConnect' expects a stem variable with the following information:

- Addressing family, this is usually "AF_INET".
- Dotted IP address to connect to.
- Well known port number on the server to connect to.

Normally the server alias will be specified at the command line which has to be resolved to its dotted IP address with a call to 'SockGetHostByName'. The resolved address can then be used to

connect the socket to the server with a call to 'SockConnect'. After the connection has been established the input string is sent to the server and the program waits for the response from the server which is then stored in the variable '**OutString**'. The call to 'SockRecv' is a blocking call, i.e. it will not return before the server has sent an answer or an error occurred. The socket is finally cleaned up, closed, and then the result from the mirror server is shown on the screen.

Mirror server

The server program needs only a few more lines to setup the server connection queue:

```
/* MSERVER.CMD - IBM REXX Sample Program                */

/* Load REXX Socket library if not already loaded       */
If RxFuncQuery("SockLoadFuncs") Then
 Do
   Call RxFuncAdd "SockLoadFuncs","RXSOCK","SockLoadFuncs"
   Call SockLoadFuncs
 End

/* create a TCP socket for client connection requests   */
Socket = SockSocket("AF_INET", "SOCK_STREAM", "0")

/* find out local IP address                            */
Host.!addr = SockGetHostId()

/* Bind socket to well known port 1996                  */
Host.!family = "AF_INET"
Host.!port = 1996
Call SockBind Socket, "Host.!"

/* create a connection queue for 1 client               */
Call SockListen Socket, 1

/* wait for a client to connect                         */
Say "Waiting for a client to connect..."
ClientSocket = SockAccept(Socket)
Say "Client has established connection."

/* we don't want any more clients, close request socket */
Call SockShutDown Socket, 2
Call SockClose Socket

/* read string from client, reverse it and send it back */
Call SockRecv ClientSocket, "InpString", 256
Say "String read from client: '" || InpString || "'"
OutString = Reverse(InpString)
Call SockSend ClientSocket, OutString

/* close client socket                                  */
Call SockShutDown ClientSocket, 2
Call SockClose ClientSocket
```

Initialization of the server program is done in exactly the same way as for the client. The REXX socket library is loaded and then a stream type socket is created. The new socket will then be

registered with the well known port number 1996 on the server by calling 'SockBind'. The dotted IP address of the server needed by this call is retrieved with 'SockGetHostId'. The IP address of the server and the well known port number will be the only information needed by a client to connect to the mirror server. Port numbers below 1024 should not be used for private applications because they are reserved for TCP/IP specific applications, such as FTP, TELNET, etc..

Before a client can connect to the server a client connection request queue has to be created using the 'SockListen' function. This queue, whose size can be specified on the 'SockListen' call, is used to serialize multiple connection requests on the server. Every client that wants to connect to the server will open a connection to the well known port on the server and will get added to the connection request queue automatically. The server has to call 'SockAccept' to accept such a connection request from the client. 'SockAccept' will return a new socket id that can be used to communicate with the client. The original socket still exists and can be used to allow more clients to connect to the server by issuing another 'SockAccept' call.

In our example we won't accept any more clients so we can simply close the original socket, read the input string from the client socket, reverse the string, send it back to the client, and close the socket. It's not important that the server keeps its end of the socket open until the client has read all the data. Data still in the socket will only be destroyed if both ends of the socket are closed.

These two programs do not check any error conditions since they are only for demonstration purposes. Of course all calls to the REXX socket library report error conditions if they occur. For a complete reference of these error conditions have a look at the documentation of the REXX socket library and also at the TCP/IP programming reference.

A sample remote control application

The next sample shown in this tutorial is a remote control application. It consists of a server process waiting for clients to connect, and offering them several commands to be executed on the server. Multiple clients are able to connect to the server at the same time.

The following commands are available to the client when connected to the server:

- **DIR [filemask]**
  Show the directory on the server. If specified, use the filemask to list only the files matching the filemask.
- **TYPE filename**
  List the contents of the file with name **filename** on the server. The server program assumes that the user only tries to type readable files. The results when typing binary files are unpredictable.
- **QUIT**
  Close connection to the server.

From what we have seen in the previous example it will not be too difficult to extend the client and server logic to handle the commands from the client. In order to handle multiple connection

requests at the same time we will have to split up the processing on the server into two programs. The first program will create a socket and bind it to a well known port address, waiting for a client to connect. As soon as a client connects to the server the second program will be started in another session taking over the communication and processing the commands sent by the client. Programs such as the first one are called a '**daemon process**'. These daemons normally run in the background waiting for clients and from their nature do not allocate too much system resources. The second program contains the actual program logic and is called the '**handler process**'.

Remote control client

Let's have a look at the client program first. The program listing will be shown in this tutorial function by function with comments for each function. It has basically the same logic for initialization of the communication with the server except that the communication will not stop before the user has entered the 'QUIT' command. This is the source code for the main program of the client:

```
/* RCLIENT.CMD - IBM REXX Sample Program                */
Parse Arg Server

/* check command line arguments, server is required     */
If Server = "" Then
 Do
   Say "Usage: RCLIENT Servername"
   Say "  Servername may contain a port number separated",
       "with a colon."
   Exit 1
 End

/* Load REXX Socket library if not already loaded       */
If RxFuncQuery("SockLoadFuncs") Then
 Do
   Call RxFuncAdd "SockLoadFuncs","RXSOCK","SockLoadFuncs"
   Call SockLoadFuncs
 End

/* Connect to remote control server                     */
Socket = Connect(Server)
If Socket = -1 Then
  Exit 1

/* loop until QUIT command was entered                  */
Do Until Command = "QUIT"
  Say "Please enter one of: 'DIR [path]', 'TYPE name'",
      " or 'QUIT'"
  Parse Pull CommandLine
  Parse Upper Var CommandLine Command Option
  If Length(Command) > 0 Then
    Call SendCommand Socket, CommandLine
End

/* Close connection to server                           */
Call Close Socket
Exit
```

The client program requires the alias name of the remote control server as a commandline argument. In addition to the alias you can choose to use a different well known port address on the server instead of the predefined value of 1234. To specify another port address you have to append a colon and the new port number to the server alias (without any inserted blanks). To connect to server 'myserver' at port 4321 you would start the client with:

```
RCLIENT myserver:4321
```

If a valid server alias was supplied as an argument the program connects to the server by calling the function '**Connect**'. If for any reason no socket could be created to connect to the server, this function will return -1 and the main program will be terminated.

The main part of the client program is the loop where the user is asked to enter a command to be executed on the server until 'QUIT' is entered. The command read from the keyboard will be sent to the server with the '**SendCommand**' function that will also display the response from the server. Finally, if the user has entered 'QUIT' to leave the client, the socket communication will be closed and the program terminated.

The '**Connect**' function itself separates the server alias from the port number (if specified), determines the IP address of the server, creates a socket and connects it to the specified server. On completion it will return the socket handle to the caller or -1 if an error occurred:

```
/*********************************************************/
/*                                                       */
/* Function:  Connect                                    */
/* Purpose:   Create a socket and connect it to server.  */
/* Arguments: Server - server name, may contain port no.*/
/* Returns:   Socket number if successful, -1 otherwise */
/*                                                       */
/*********************************************************/
Connect: Procedure
  Parse Arg Server

  /* if the servername has a port address specified     */
  /* then use this one, otherwise use the default port  */
  /* for the remote control server (1234)               */
  Parse Var Server Server ":" Port
  If Port = "" Then
    Port = 1234

  /* resolve server name alias to dotted IP address     */
  rc = SockGetHostByName(Server, "Host.!")
  If rc = 0 Then
   Do
     Say "Unable to resolve server:" Server
     Return -1
   End

  /* create a TCP socket                                */
  Socket = SockSocket("AF_INET", "SOCK_STREAM", "0")
  If Socket < 0 Then
```

```
 Do
   Say "Unable to create socket"
   Return -1
 End

/* connect the new socket to the specified server     */
Host.!family = "AF_INET"
Host.!port = Port
rc = SockConnect(Socket, "Host.!")
If rc < 0 Then
 Do
   Say "Unable to connect to server:" Server
   Call Close Socket
   Return -1
 End

Return Socket
```

The '**SendCommand**' function sends the specified command to the server where it will be processed. The response from the server will be read and acknowledged line by line. The reason for this implementation is the following: by sending only one line from the server and then waiting for an acknowledgement from the client we save some program logic to handle more than one line received on the client at once. This could happen, when multiple 'SockSend' calls on the server are processed before the client reads the results from the socket. In this case all the data would be read from the socket at once. We could add CR/LF characters to the data sent to the client but then we also would have to split the lines on the client to be able to recognize the end of transmission string. By sending line by line we can always assume that the end of transmission indicator will be received as a standalone string. The value used for the acknowledgement from the client to the server is not relevant, as long as it is not an empty string.

```
/*******************************************************/
/*                                                     */
/* Procedure: SendCommand                              */
/* Purpose:   Send a command via the specified socket  */
/*            and display the full response from server.*/
/* Arguments: Socket - active socket number            */
/*            Command - command string                 */
/* Returns:   nothing                                  */
/*                                                     */
/*******************************************************/
SendCommand: Procedure
  Parse Arg Socket, Command

  /* send the command to the remote control server     */
  Call SockSend Socket, Command
  Do Forever
    BytesRcvd = SockRecv(Socket, "RcvData", 1024)

    /* error or end of response encountered            */
    If BytesRcvd <= 0 |,
       RcvData = ">>>End_of_transmission<<<" Then
      Leave
```

```
    /* display response and send acknowledge to server  */
    Say RcvData
    Call SockSend Socket, "OK!"
  End

  Say "----- end of output from command:" Command "-----"
  Return
```

The '**Close**' function finally shuts down the used socket and closes it. According to the socket documentation a call to 'SockClose' should be sufficient to close down the socket connection. However experience shows that a preceding call to 'SockShutDown' cleans up the environment better and the sample programs can be run again immediately without experiencing problems with ports being still in use.

```
/*******************************************************/
/*                                                     */
/* Procedure: Close                                    */
/* Purpose:   Close the specified socket.              */
/* Arguments: Socket - active socket number            */
/* Returns:   nothing                                  */
/*                                                     */
/*******************************************************/
Close: Procedure
  Parse Arg Socket
  Call SockShutDown Socket, 2
  Call SockClose Socket
  Return
```

Remote control daemon

This is the source code for the daemon program running on the server:

```
/* RSERVERD.CMD - IBM REXX Sample Program              */
Parse Arg Port
If Port = "" Then
  Port = 1234

/* Load REXX Socket library if not already loaded      */
If RxFuncQuery("SockLoadFuncs") Then
 Do
   Call RxFuncAdd "SockLoadFuncs","RXSOCK","SockLoadFuncs"
   Call SockLoadFuncs
 End
/* Open socket at well known port and wait for clients  */
Socket = ListenPort(Port)
If Socket = -1 Then
  Exit 1

/* close the socket when program is interrupted        */
Signal On Halt
Do Forever
  /* wait for client to connect and start handler      */
  Say "Waiting for client to connect."
  Say "Press Ctrl-C to exit program."
  ClientSocket = SockAccept(Socket)
```

```
  Say "Client connected, starting handler process."
  "start rserverh.cmd" ClientSocket
End

Halt:
Call Close Socket
Exit
```

The logic of the remote control daemon is quite simple. It creates a socket, connects it to a well known port address (which can be either the default value for our application or a user defined port address) and then enters an endless loop waiting for clients to connect. As soon as a client has connected it starts a new session with the handler program which then takes over the communication with the client. The loop waiting for clients is an infinite loop which can only be interrupted by stopping the program with Ctrl-C. A signal handler for the 'HALT' signal has been added to cover this event to correctly close down the socket.

The '**ListenPort**' function of the daemon program creates a new socket at a well known port address waiting for clients to connect:

```
/*******************************************************/
/*                                                     */
/* Function:  ListenPort                               */
/* Purpose:   Create a socket, bind it to a port and   */
/*            listen at the port for connecting clients.*/
/* Arguments: Port - port number                       */
/* Returns:   Socket number if successful, -1 otherwise */
/*                                                     */
/*******************************************************/
ListenPort: Procedure
  Parse Arg Port

  /* create a TCP socket                               */
  Socket = SockSocket("AF_INET", "SOCK_STREAM", "0")
  If Socket < 0 Then
   Do
     Say "Unable to create socket"
     Return -1
   End

  /* find out local IP address and bind socket to port  */
  Host.!addr = SockGetHostId()
  Host.!family = "AF_INET"
  Host.!port = Port

  rc = SockBind(Socket, "Host.!")
  If rc < 0 Then
   Do
     Say "Unable to bind to port:" Port
     Call Close Socket
     Return -1
   End

  /* listen at the port, allow 5 clients in queue       */
  rc = SockListen(Socket, 5)
```

```
If rc < 0 Then
 Do
   Say "Unable to listen at port:" Port
   Call Close Socket
   Return -1
 End

Return Socket
```

The '**ListenPort**' creates a connection request queue for up to 5 concurrent connection requests from clients. If any of the socket calls fails the function will return a value of -1 to indicate a failure. If no problem occurred the socket handle will be returned.

The last function in the daemon program is the '**Close**'. Since it is identical to the 'Close' function of the client program it is not listed again.

Remote control handler

The last part of the remote control application is the handler process. When the handler is started from the daemon process it will receive the socket handle which is used to communicate with the client. It will then print out some information on the connected client. The dotted IP address of the connected client can be retrieved with a call to '**SockGetPeerName**' and the alias name is then retrieved with a call to '**SockGetHostByAddr**'. This information is printed on the screen and then a loop is entered where the handler waits for commands from the client. The command is received and processed in the function '**ReceiveRequest**'. This function returns the command string which is used to leave the loop when the command 'QUIT' has been received. To clean up the resources the socket will be closed and finally the OS/2 session in which the handler process is running will be terminated.

As already mentioned in the client program the protocol used for the communication between client and handler expects that every line sent from the handler is acknowledged by the client. The end of the result of a processed command will be marked with a special 'End of transmission' indicator.

The handler process uses two functions to send lines back to the client. The function '**Anser**' sends a single line to the client and waits for an acknowledgement while the function '**AnswerQueue**' is used to send all lines available in the REXX session queue to the client. The latter function makes it extremely easy to implement the two commands provided by the server. Both commands can be executed on the server workstation without modifications by the OS/2 command processor. The output of the commands will be captured and redirected into the REXX session queue by piping the output into RXQUEUE.EXE. Unnamed session queues are used in this example because such a queue is needed for every handler process running on the server. Named queues are shared among all processes on a PC and are therefore not suited for our purposes.

The server commands itself are processed in the functions '**ProcessDirCommand**' and '**ProcessTypeCommand**' respectively. This is the complete listing of the handler program:

```
/* RSERVERH.CMD - IBM REXX Sample Program                    */
```

```
Parse Arg Socket

/* Load REXX Socket library if not already loaded       */
If RxFuncQuery("SockLoadFuncs") Then
 Do
   Call RxFuncAdd "SockLoadFuncs","RXSOCK","SockLoadFuncs"
   Call SockLoadFuncs
 End

/* Show some information about connected client         */
Call SockGetPeerName Socket, "ClientAddr.!"
Call SockGetHostByAddr ClientAddr.!addr, "ClientAddr.!"
Say "Established connection with client '" ||,
    ClientAddr.!name || "'."

/* Process commands until QUIT is reached               */
Command = ""
Do Until Command = "QUIT"
  Command = ReceiveRequest(Socket)
End

/* Close socket and OS/2 session                        */
Call Close Socket
"Exit"

/******************************************************/
/*                                                    */
/* Function:  ReceiveRequest                          */
/* Purpose:   Wait for a command from the client and  */
/*            execute it. Return the identifier of the*/
/*            command to the caller.                  */
/* Arguments: Socket - active socket number           */
/* Returns:   command identifier                      */
/*                                                    */
/******************************************************/
ReceiveRequest: Procedure
  Parse Arg Socket

  /* Wait for the command from the client              */
  BytesRcvd = SockRecv(Socket, "CommandLine", 1024)
  Say "Command line from client:" CommandLine

  Parse Var CommandLine Command Option
  Command = Translate(Command)

  Select
    When Command = "DIR" Then
      Call ProcessDirCommand Socket, Option
    When Command = "TYPE" Then
      Call ProcessTypeCommand Socket, Option
    When Command = "QUIT" Then
      Nop
    Otherwise
      Call Answer Socket, "Invalid command."
  End
```

```
  /* send end of answer marker back to client          */
  Call SockSend Socket, ">>>End_of_transmission<<<"
  Return Command

/*********************************************************/
/*                                                       */
/* Procedure: Close                                      */
/* Purpose:   Close the specified socket.                */
/* Arguments: Socket - active socket number              */
/* Returns:   nothing                                    */
/*                                                       */
/*********************************************************/
Close: Procedure
  Parse Arg Socket
  Call SockShutDown Socket, 2
  Call SockClose Socket
  Return

/*********************************************************/
/*                                                       */
/* Procedure: Answer                                     */
/* Purpose:   Send one answer line back to the client    */
/*            and wait for acknowledgement from client.  */
/* Arguments: Socket - active socket number              */
/*            AnswerString - line to send to client      */
/* Returns:   nothing                                    */
/*                                                       */
/*********************************************************/
Answer: Procedure
  Parse Arg Socket, AnswerString
  Call SockSend Socket, AnswerString
  Call SockRecv Socket, "Ack", 256
  Return

/*********************************************************/
/*                                                       */
/* Procedure: AnswerQueue                                */
/* Purpose:   Send all lines from the session queue      */
/*            back to the client as the answer of the    */
/*            previous executed command.                 */
/* Arguments: Socket - active socket number              */
/* Returns:   nothing                                    */
/*                                                       */
/*********************************************************/
AnswerQueue: Procedure
  Parse Arg Socket

  /* send answer lines until session queue is empty     */
  Do While Queued() > 0
    Parse Pull Line

    /* empty lines will be sent as a space              */
    If Line = "" Then
      Line = " "

    Call Answer Socket, Line
```

```
  End
  Return

/*********************************************************/
/*                                                       */
/* Procedure: ProcessDirCommand                          */
/* Purpose:   Process the DIR command that was received */
/*            from the client and send back the result. */
/* Arguments: Socket - active socket number              */
/*            FileMask - optional file mask for DIR      */
/* Returns:   nothing                                    */
/*                                                       */
/*********************************************************/
ProcessDirCommand: Procedure
  Parse Arg Socket, FileMask

  /* redirect output from DIR command to session queue  */
  "DIR" FileMask " | RXQUEUE"
  Call AnswerQueue Socket
  Return

/*********************************************************/
/*                                                       */
/* Procedure: ProcessTypeCommand                         */
/* Purpose:   Process the TYPE command that was received*/
/*            from the client and send back the result. */
/* Arguments: Socket - active socket number              */
/*            FileName - required filename to type       */
/* Returns:   nothing                                    */
/*                                                       */
/*********************************************************/
ProcessTypeCommand: Procedure
  Parse Arg Socket, FileName

  /* TYPE needs a filename as argument                  */
  If FileName = "" Then
   Do
     Call Answer Socket, "You have to specify a filename",
          " with the TYPE command."
     Return
   End

  /* Check if the file really exists                    */
  If Stream(FileName, "C", "QUERY EXISTS") = "" Then
   Do
     Call Answer Socket, "The specified file '" ||,
          FileName || "' does not exist."
     Return
   End

  /* redirect output from TYPE command to session queue */
  "TYPE" FileName " | RXQUEUE"
  Call AnswerQueue Socket
  Return
```

TCP/IP Socket Programming with REXX

by Christian Michel, REXX Development, IBM Germany
document loaded from REXX tutorials homepage at http://www2.hursley.ibm.com/rexxtut/

A REXX client for the WWW

After having created a standalone application with its own high level protocol definition we now jump into public protocols already used in the TCP/IP world. One of those protocols is the HyperText Transfer Protocol, abbreviated as '**HTTP**'. The specifications for this and other protocols are freely available in the Internet in so called '**RFC**s' (abbreviation for Request For Comments). See the bibliography at the end of this tutorial where these documents can be found in the Internet.

The following sample program connects to a WWW server and retrieves information about the document specified as an '**URL**' (Uniform Resource Locator) on the command line. As an example of the information available for the document it will print the date when the document was last modified.

In RFC 1945 which describes the HTTP protocol we can find the following information needed for this task:

- The default well known port address for an HTTP server is 80.
- To obtain information about the last modification date the **HEAD** command can be used. This saves transmission time because the body of the document does not have to be sent through the network.

The HEAD command can be sent in two formats: the simple request or the full request. The full request format of the HEAD command is defined as follows:

```
HEAD documentname HTTP/1.0<CRLF>
request header<CRLF>
```

For our purpose we don't need to pass additional options in the request header field so we can leave this field blank. However we may not omit the closing CRLF character pair terminating the request header field otherwise the server would not accept it as a valid command. The full request sent to a server will return a full response in the format:

```
HTTP/1.0 statuscode reasonphrase<CRLF>
response body<CRLF>
```

The HTTP specification lists several information fields for the response body that can appear in any order. Currently we are only interested in the Last-Modified field and ignore all other fields.

The following line shows a sample HEAD command sent to a server with the appropriate response:

```
HEAD / HTTP/1.0<CRLF><CRLF>
```

Response from server:

```
HTTP/1.0 200 OK<CRLF>
Server: GoServe/2.45<CRLF>
Date: Thu, 18 Jul 1996 15:40:47 GMT<CRLF>
Content-Type: text/html<CRLF>
```

```
Content-Length: 1081<CRLF>
Content-Transfer-Encoding: binary<CRLF>
Last-Modified: Thu, 19 Oct 1995 16:27:52 GMT<CRLF>
```

Since we are only interested in the date when the document has been last modified we have to search the response for this keyword. During development of this sample I discovered that most web servers use the exact string as shown above to identify this field, some other servers however don't. To be able to find the date in responses from all servers we can simply uppercase the whole string before searching the last-modified field.

This is already everything we need to know for our program. This is the implementation of the main program:

```
/* SHOWDATE.CMD - IBM REXX Sample Program               */
Parse Arg

/* Load REXX Socket library if not already loaded       */
If RxFuncQuery("SockLoadFuncs") Then
 Do
   Call RxFuncAdd "SockLoadFuncs","RXSOCK","SockLoadFuncs"
   Call SockLoadFuncs
 End

/* retrieve the header of the document specified by URL */
Header = GetHeader(URL)

If Length(Header) \= 0 Then
  Do
    /* header could be read, find date                  */
    DocDate = GetModificationDate(Header)
    Say "Document date is:" DocDate
  End
Else
  Say "Document information could not be retrieved."

Exit
```

The '**Connect**' function to connect to the server is exactly the same as already seen in the remote control application except that it now uses port number 80 if no port was specified by the caller:

```
/*******************************************************/
/*                                                     */
/* Function:  Connect                                  */
/* Purpose:   Create a socket and connect it to server. */
/* Arguments: Server - server name, may contain port no.*/
/* Returns:   Socket number if successful, -1 otherwise */
/*                                                     */
/*******************************************************/
Connect: Procedure
  Parse Arg Server

  /* if the servername has a port address specified     */
  /* then use this one, otherwise use the default http  */
  /* port 80                                            */
```

```
Parse Var Server Server ":" Port
If Port = "" Then
  Port = 80

/* resolve server name alias to dotted IP address     */
rc = SockGetHostByName(Server, "Host.!")
If rc = 0 Then
 Do
   Say "Unable to resolve server:" Server
   Return -1
 End

/* create a TCP socket                                */
Socket = SockSocket("AF_INET", "SOCK_STREAM", "0")
If Socket < 0 Then
 Do
   Say "Unable to create socket"
   Return -1
 End

/* connect the new socket to the specified server     */
Host.!family = "AF_INET"
Host.!port = Port
rc = SockConnect(Socket, "Host.!")
If rc < 0 Then
 Do
   Say "Unable to connect to server:" Server
   Call Close Socket
   Return -1
 End

Return Socket
```

The '**SendCommand**' function expects a single line command from the caller. As needed by the HTTP protocol two pairs of CRLF are appended to the command string to classify the command as a full request. After the command has been sent the function receives the response from the server until no more characters can be read and returns the response:

```
/*****************************************************/
/*                                                   */
/* Function:  SendCommand                            */
/* Purpose:   Send a command via the specified socket*/
/*            and return the full response to caller.*/
/* Arguments: Socket - active socket number          */
/*            Command - command string               */
/* Returns:   Response from server or empty string if*/
/*            failed.                                */
/*                                                   */
/*****************************************************/
SendCommand: Procedure
  Parse Arg Socket, Command

  /* append two pairs of CRLF to end the command string */
  Command = Command || "0D0A0D0A"x
```

```
  BytesSent = SockSend(Socket, Command)
  Response = ""
  Do Forever
    BytesRcvd = SockRecv(Socket, "RcvData", 1024)
    If BytesRcvd <= 0 Then
      Leave
    Response = Response || RcvData
  End

  Return Response
```

The '**Close**' function is already well known from the previous samples:

```
/*******************************************************/
/*                                                     */
/* Procedure: Close                                    */
/* Purpose:   Close the specified socket.              */
/* Arguments: Socket - active socket number            */
/* Returns:   nothing                                  */
/*                                                     */
/*******************************************************/
Close: Procedure
  Parse Arg Socket
  Call SockShutDown Socket, 2
  Call SockClose Socket
  Return
```

The '**GetHeader**' function isolates the server name and document name from the passed URL, connects to the server, retrieves the full header information and closes the connection again, returning the full header to the caller:

```
/*******************************************************/
/*                                                     */
/* Function:  GetHeader                                */
/* Purpose:   Request the header for the specified URL */
/*            from the network.                        */
/* Arguments: URL - fully specified document locator   */
/* Returns:   Full header of specified document or     */
/*            empty string if failed (also if no header*/
/*            exists).                                 */
/*                                                     */
/*******************************************************/
GetHeader: Procedure
  Parse Arg URL

  /* Isolate server name and document name, document   */
  /* name is always preceded with a slash              */
  Parse Var URL "http://" Server "/" Document
  Document = "/" || Document

  Socket = Connect(Server)
  If Socket = -1 Then
    Return ""

  Command = "HEAD" Document "HTTP/1.0"
```

```
Header = SendCommand(Socket, Command)
Call Close Socket
Return Header
```

Finally the function '**GetModificationDate**' searches the full header (which is passed in a single string) for the last modification date. As already mentioned we search only the uppercased header to avoid problems with some web servers. To find the last modification date it looks for the keyword "LAST-MODIFIED:" and a trailing linefeed character ("0A"x). The extracted modification date now could still contain leading or trailing blanks or carriage return characters that will be removed before the result is returned to the caller. Searching only for the linefeed character as a delimiter ensures that the program will also work with web servers that use only the UNIX style line separation character:

```
/*******************************************************/
/*                                                     */
/* Function:  GetModificationDate                      */
/* Purpose:   Find the last-modified date in the passed */
/*            header and return just the date.         */
/* Arguments: Header - full header of document         */
/* Returns:   Date string when document was last       */
/*            modified or empty string if date was not  */
/*            found.                                   */
/*                                                     */
/*******************************************************/
GetModificationDate: Procedure
  Parse Arg Header

  /* isolate date string and strip all unwanted chars   */
  Parse Upper Var Header "LAST-MODIFIED:" ModDate "0A"x
  ModDate = Strip(ModDate)
  ModDate = Strip(ModDate,,"0D"x)

  Return ModDate
```

An automated URL checker

Another sample program is useful when you have to observe several documents in the World Wide Web and need to know when these documents have been modified. Often it is very time consuming to load these documents with your web browser just to see that nothing has changed (e.g. if the documents are available via slow links only). In this case it would help you to have an automated checking process running during the night on your server preparing a list of changed documents. This task will be accomplished by the following '**URL**' (Uniform Resource Locator) checker program. It reads a list of URLs for WWW documents, retrieves the modification dates for these documents and based on the previously retrieved information classifies the documents into changed, unchanged and unchecked documents. The output of the program will be formatted in HTML with links to the documents so you can use this output document as your initial home page at browser startup.

Based on the previous sample program the main program of the URL checker is pretty short:

by Christian Michel, REXX Development, IBM Germany
document loaded from REXX tutorials homepage at http://www2.hursley.ibm.com/rexxtut/

```
/* URLCHECK.CMD - IBM REXX Sample Program                */
Parse Arg URLList HTMLFile

/* Load REXX Socket library if not already loaded       */
If RxFuncQuery("SockLoadFuncs") Then
 Do
   Call RxFuncAdd "SockLoadFuncs","RXSOCK","SockLoadFuncs"
   Call SockLoadFuncs
 End

/* check all URLs in the specified file for expiration  */
URLS.0 = 0
Changed = ""
Unchanged = ""
Commented = ""
Call CheckURLs URLList
Call WriteHTML HTMLFile, Changed, Unchanged, Commented
Exit
```

The following variables are used in the main program:

- '**URLS**' - this stem is used to hold all URLs from the input file including the last modification date. As usual URLS.0 indicates the number of elements in this stem variable.
- '**Changed**' - this string variable will contain the indices of all changed documents in the URLS stem variable separated by blanks, such as "1 4 5 8".
- '**Unchanged**' - this string variable will contain the indices of all documents that have not been changed since the last check.
- '**Commented**' - this string variable will contain the indices of all documents that have not been checked during this execution of the check program but have been commented in the input list of URLs to check.

The URLs to be checked are listed in the input file one URL per line. If you want to exclude an URL temporarily from the check you can comment it out by preceding it with a number sign "#". The function '**CheckURLs**' reads this list, determines if the document has changed and adds the index to one of the variables 'Changed', 'Unchanged' or 'Commented'. After all documents have been checked, an updated URL list will be written containing not only the URL but also the string with the last modification date to be used for comparison at the next run of the check program.

Based on the information in the the three index variables and the 'URLS' stem the function '**WriteHTML**' then writes out a HTML file with links to all URLs from the input file grouped by their status.

The following functions are reused from the previous sample and are not listed again:

- Connect
- SendCommand
- Close
- GetHeader
- GetModificationDate

The check of all URLs will be done in the function '**CheckURLs**'. At the beginning it exposes the global index variables for the different URL states and the stem variable for the list of URLs since they will be used to pass on this information to the next function. It then reads the list of URLs from the specified file, retrieves the header and modification date for all active documents and compares the result with the input from the URL list file (where available). According to the result of this comparison the index of the current URL will be appended to the appropriate index list for later use, as will the modified date be updated in the URLS stem variable.

```
/*******************************************************/
/*                                                     */
/* Procedure: CheckURLs                                */
/* Purpose:   Check the modification dates of all URLs  */
/*            listed in the specified file. If the date */
/*            has changed, update the list file with    */
/*            the new date.                             */
/* Arguments: URLFile - file containing URL list       */
/* Returns:   nothing                                  */
/*                                                     */
/*******************************************************/
CheckURLs: Procedure Expose URLS. Changed Unchanged,
                            Commented
  Parse Arg URLFile

  Index = 0
  Do While Lines(URLFile)
    /* read line with URL and last modification date    */
    URLLine = LineIn(URLFile)

    /* remember line for later update of file           */
    Index = Index + 1
    URLS.0 = Index
    URLS.Index = URLLine

    /* if first character is not a "#" then process URL */
    If SubStr(URLLine, 1, 1) \= "#" Then
     Do
       /* retrieve header for specified URL             */
       Parse Var URLLine URL ModDate
       Header = GetHeader(URL)

       If Length(Header) \= 0 Then
        Do
          /* header could be read, find date           */
          DocDate = GetModificationDate(Header)

          If Length(ModDate) = 0 | ModDate \= DocDate Then
           Do
             /* this URL has been changed, add to list  */
             /* of changed URLs and update the date     */
             Changed = Changed Index
             URLS.Index = URL DocDate
           End
          Else
            /* add index to list of unchanged URLs     */
```

```
            Unchanged = Unchanged Index
          End
         Else
           /* add index to list of unchanged URLs        */
           Unchanged = Unchanged Index
       End
      Else
        /* add index to list of all commented out URLs   */
        Commented = Commented Index
    End

    /* close input stream, erase it and then rewrite it   */
    Call Stream URLFile, "C", "CLOSE"
    "@DEL" URLFile

    Do Index = 1 To URLS.0
      Call LineOut URLFile, URLS.Index
    End

    Call Stream URLFile, "C", "CLOSE"
    Return
```

After the all documents have been checked the result will be formatted into a '**HTML**' file with links to the original documents. For details of HTML (HyperText Markup Language) see RFC 1866. The output file is created in the function '**WriteHTML**'. It deletes an already existing version of the output file, creates a simple header, formats the lists of changed, unchanged and commented documents, and finally closes the file with a simple trailer containing the current time:

```
/*******************************************************/
/*                                                     */
/* Procedure: WriteHTML                                */
/* Purpose:   Create a new HTML document with links to  */
/*            the input URLs grouped by modification.  */
/* Arguments: HTML - output filename                   */
/*            Changed - list of changed URL indices    */
/*            Unchanged - list of unchanged URL indices */
/*            Commented - list of commented URL indices */
/* Returns:   nothing                                  */
/*                                                     */
/*******************************************************/
WriteHTML: Procedure Expose URLS.
  Parse Arg HTML, Changed, Unchanged, Commented

  /* write new HTML document with links to URLs        */
  "@DEL" HTML "1>NUL 2>NUL"

  Call LineOut HTML, "<html><head>"
  Call LineOut HTML, "<title>My link list</title>"
  Call LineOut HTML, "</head><body>"


  Call LineOut HTML, "<h1>Changed documents</h1>"
  Call FormatURLList HTML, Changed
  Call LineOut HTML, "<h1>Unchanged documents</h1>"
```

by Christian Michel, REXX Development, IBM Germany
document loaded from REXX tutorials homepage at http://www2.hursley.ibm.com/rexxtut/

```
Call FormatURLList HTML, Unchanged
Call LineOut HTML, "<h1>Commented documents</h1>"
Call FormatURLList HTML, Commented

Call LineOut HTML, "<p><i>Documents checked at",
             Date() "on" Time() "</i>"
Call LineOut HTML, "</body></html>"
Return
```

The function '**FormatURLList**' is used to format a single index list into the HTML output format with one URL per line. This version of the formatter simply creates a hyper link to the document and lists the URL of the document followed by a line break. Another solution would be to format the URLS in an unordered list, etc., see the HTML reference for more formatting options.

```
/*******************************************************/
/*                                                     */
/* Procedure: FormatURLList                            */
/* Purpose:   Format a list of URL indices into a HTML  */
/*            formatted list with links to the URLs.   */
/* Arguments: HTML - output filename                   */
/*            List - list of indices                   */
/* Returns:   nothing                                  */
/*                                                     */
/*******************************************************/
FormatURLList: Procedure Expose URLS.
  Parse Arg HTML, List

  /* are there any indices in the list?                */
  If Words(List) > 0 Then
   Do
    Do Index = 1 To Words(List)
      Idx = Word(List, Index)
      Parse Var URLS.Idx URL ModDate
      URL = Strip(URL, "L", "#")

      Call LineOut HTML, "<br><a href=""" || URL || """>"
      Call LineOut HTML, URL || "</a>"
      If Length(ModDate) > 0 Then
        Call LineOut HTML, ", last modified at" ModDate
    End
   End
  Else
    Call LineOut HTML, "<p><i>no documents in list</i><p>"
  Return
```

This is is a sample input file for the URL checker:

```
http://www.ibm.com
http://www.myhost.mydomain/users/chris.html
#http://www2.hursley.ibm.com/rexx/
```

After running the checker the resulting HTML file could look like that:

```
<html><head>
<title>My link list</title>
</head><body>
<h1>Changed documents</h1>
<p><i>no documents in list</i><p>
<h1>Unchanged documents</h1>
<br><a href="http://www.ibm.com">
http://www.ibm.com</a>
, last modified at THU, 18 JUL 1996 17:41:10 GMT
<br><a href="http://www.myhost.mydomain/users/chris.html">
http://www.myhost.mydomain/users/chris.html</a>
, last modified at MONDAY, 22-JUL-96 19:51:25 GMT
<h1>Commented documents</h1>
<br><a href="http://www2.hursley.ibm.com/rexx/">
http://www2.hursley.ibm.com/rexx/</a>
<p><i>Documents checked at 24 Jul 1996 on 18:43:56 </i>
</body></html>
```

Running the program every night or early morning on a server gives you a daily updated list of the documents you want to follow. By using the generated HTML file as the startup page in your web browser you can access the changed documents directly via their links.

The shown URL checker can be improved in many ways, e.g.:

- sort documents descending by date
- extract title of each document to show in the link list
- mirror documents on a local web server, ideally with all embedded graphics

You have seen connecting to a WWW server is not difficult. If you use the command ‘GET’ instead of ‘HEAD’ the server will send you the whole document preceded by the same header information which we have already used. Based on the information from this tutorial you could for example write a program that maintains a local shadow of a distant web server. You would have to retrieve a document, extract the links in it and follow them recursively. Sure you have other ideas what can be done with REXX in the Internet.

REXX/Sockets on VM

The last topic of this tutorial will show you the differences of the REXX/SOCKET implementation on VM. The main difference of this implementation is in the used syntax. Nearly all functions have equivalents on both platforms. Only the handling of multiple clients on the server side is done differently on VM since you cannot start a new session to communicate with each new client.

TCP/IP Socket Programming with REXX

by Christian Michel, REXX Development, IBM Germany
document loaded from REXX tutorials homepage at http://www2.hursley.ibm.com/rexxtut/

Function equivalents

Initialize the socket interface at program startup:

OS/2	`If RxFuncQuery("SockLoadFuncs") Then Do`
	`  Call RxFuncAdd "SockLoadFuncs","RXSOCK","SockLoadFuncs"`
	`  Call SockLoadFuncs`
	`End`

VM	`/* initialize socket interface for MYPROG EXEC */` `Call Socket "Initialize", "MYPROG"`

Deregister socket interface:

OS/2	`Call SockDropFuncs`
VM	`/* deregister socket interface for MYPROG EXEC */` `Call Socket "Terminate"`

Create an unconnected socket:

OS/2	`Socket = SockSocket("AF_INET", "SOCK_STREAM", "0")`

VM	`/* create a new socket and activate ASCII translation */`
	`Parse Value Socket("Socket") With SockRc Sock`
	`Call Socket "SetSockOpt", Sock, "SOL_SOCKET", "SO_ASCII", "On"`

Connect a socket to a host/port address:

OS/2	`Host.!family = "AF_INET"`
	`Host.!address = "9.164.0.197"  /* set remote host address */`
	`Host.!port = 1996`
	`Call SockConnect Socket, "Host.!"`

VM	`Call Socket "Connect", Sock, "AF_INET 1996 9.164.0.197"`

Send a string over a socket:

OS/2	`Call SockSend Socket, "This is my test string"`
VM	`Call Socket "Send", Sock, "This is my test string"`

Receive a string from a socket:

OS/2	`/* receive max. 1024 characters */`
	`BytesRcvd = SockRecv(Socket, "InBuffer", 1024)`
	`Say InBuffer`

VM	`Parse Value Socket("Recv", Sock) With SockRc BytesRcvd,`
	`  InBuffer`
	`Say InBuffer`

Shutdown a socket:

OS/2	`Call SockShutDown Socket, 2`
VM	`Call Socket "ShutDown", Sock`

Close a socket:

OS/2	`Call SockClose Socket`
VM	`Call Socket "Close", Sock`

Bind a socket to a well known port on the server:

OS/2	`Host.!family = "AF_INET"`
	`Host.!port = 1996`
	`Host.!address = "9.164.0.197"  /* set local host address */`
	`Call SockBind Socket, "Host.!"`

VM	`LocalAddr = "9.164.0.197"` `Call Socket "Bind", Sock, "AF_INET 1996" LocalAddr`

Listen on a socket for a client connection request:

OS/2	`/* accept max. 5 clients simultaneously */` `Call SockListen Socket, 5`
VM	`/* accept max. 5 clients simultaneously */` `Call Socket "Listen", Sock, 5`

Accept a connecting client:

OS/2	`ClientSock = SockAccept(Socket)`

VM	`/* socket handling on VM is done via REXX/WAIT package! */`
	`Parse Value Socket("Accept", Sock) With SockRc ClientSock,`
	`  ClientInfo`

Find out local IP address:

OS/2	`LocalAddr = SockGetHostId()`
VM	`LocalAddr = Socket("GetHostId")`

Resolve a host name alias to an IP address:

OS/2	`Call SockGetHostByName "www2.hursley.ibm.com", "Host.!"` `Say Host.!addr`

VM	`Parse Value Socket("GetHostByName", "www2.hursley.ibm.com"),`
	`  With SockRc HostAddr`
	`Say HostAddr`

VM implementation of mirror sample

The following two listings show the mirror client and server sample in the VM implementation. Besides the different function call syntax the main difference can be found in the server program. To be able to process requests from multiple sockets the REXX/WAIT package is used. The 'Wait' function of the package receives a condition string on which events the program wants to react. The mirror server wants to react on every read required for any socket. The matching condition string for this would be "`Socket Read *`". Instead of the asterisk you can also specify one or more socket handles on which a program wants to react. This would be useful if more than one server programs is running on the system.

This is the client program:

```
/* MCLIENT EXEC - IBM REXX Sample Program                */
/* Parameters:                                           */
/*   Server: alias name of mirror server                 */
Parse Arg Server

/* Initialize REXX Socket function package               */
Call Socket "Initialize", "MCLIENT"
```

```
/* Ask user for string to send to the mirror server     */
Say "Please enter a string that should be mirrored"
Parse Pull InpString

/* create a TCP socket and activate ASCII translation   */
Sock = Socket("Socket")
Call Socket "SetSockOpt",Sock,"SOL_SOCKET","SO_ASCII","On"

/* connect the new socket to the specified server       */
Call Socket "Connect", Sock, "AF_INET 1996" Server

/* send the input string to the mirror server           */
Call Socket "Send", Sock, InpString

/* receive answer from mirror server and close socket   */
Parse Value Socket("Recv", Sock) With BytesRcvd OutString
Call Socket "ShutDown", Sock
Call Socket "Close", Sock

Say "The string '" || InpString || "' was mirrored to"
Say "'" || OutString || "'."

/* deregister socket function package for this process  */
Call Socket "Terminate"
Exit

/*******************************************************/
/*                                                     */
/* Procedure: Socket                                   */
/* Purpose:   Forward the socket call and return the   */
/*            return string only, not the return code. */
/* Arguments: arguments to 'SOCKET' function           */
/* Returns:   return string of 'SOCKET' function       */
/*            without return code                      */
/*                                                     */
/*******************************************************/
Socket: Procedure Expose SockRc
  a0 = Arg(1)
  a1 = Arg(2)
  a2 = Arg(3)
  a3 = Arg(4)
  a4 = Arg(5)
  a5 = Arg(6)
  a6 = Arg(7)
  a7 = Arg(8)
  a8 = Arg(9)
  a9 = Arg(10)
  Parse Value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) With SockRc
Res
  Return Res
```

And finally the server program:

```
/* MSERVER EXEC - IBM REXX Sample Program               */
/* Initialize REXX Socket function package              */
```

by Christian Michel, REXX Development, IBM Germany
document loaded from REXX tutorials homepage at http://www2.hursley.ibm.com/rexxtut/

```
Call Socket "Initialize", "MSERVER"
/* create a TCP socket for client connection requests   */
Sock = Socket("Socket")
Call Socket "SetSockOpt",Sock,"SOL_SOCKET","SO_ASCII","On"

/* find out local IP address                            */
HostAddr = Socket("GetHostId")

/* Bind socket to well known port 1996                  */
Call Socket "Bind", Sock, "AF_INET 1996" HostAddr

/* create a connection queue for 1 client               */
Call Socket "Listen", Sock, 1

/* set mode of socket to non blocking                   */
Call SetValue "Socket" Sock "Non-Blocking"
Say "Waiting for a client to connect..."

/* now wait for a client to connect                     */
Do Forever
  /* wait for a socket read event or console input      */
  Event = Wait("Socket Read *", "Cons")
  Parse Upper Var Event WRc Event Rest
  Select
    When Event = "CONS" Then Do
      If Rest = "EXIT" Then
        Leave
      Else
        Say "Enter 'Exit' to leave program"
    End

    When Event = "SOCKET" Then Do
      Parse Var Rest KeyWord EvtSock

      If KeyWord = "READ" & EvtSock = Sock Then Do
        /* a client has connected to the request socket */
        /* show info available on the client           */
        Parse Value Socket("Accept", Sock) with CltSock CltInfo
        Say "Client has established connection:" CltInfo

        /* we don't want more clients, close request socket */
        Call Socket "ShutDown", Sock
        Call Socket "Close", Sock
      End

      If KeyWord = "READ" & EvtSock \= Sock Then Do
        /* a client has sent a string to mirror         */
        Parse Value Socket("Recv", EvtSock) with Bytes InpString
        Say "String read from client: '" || InpString || "'"
        OutString = Reverse(InpString)
        Call Socket "Send", EvtSock, OutString

        /* close the client socket                      */
        Call Socket "ShutDown", EvtSocket
        Call Socket "Close", EvtSocket
```

```
          /* we have answered the request, quit program   */
          Leave
        End
      End

      Otherwise Nop
    End
  End

  /* deregister socket function package for this process  */
  Call Socket "Terminate"
  Exit

  /*****************************************************/
  /*                                                   */
  /* Procedure: Socket                                 */
  /* Purpose:   Forward the socket call and return the */
  /*            return string only, not the return code.*/
  /* Arguments: arguments to 'SOCKET' function         */
  /* Returns:   return string of 'SOCKET' function     */
  /*            without return code                    */
  /*                                                   */
  /*****************************************************/
  Socket: Procedure Expose SockRc
    a0 = Arg(1)
    a1 = Arg(2)
    a2 = Arg(3)
    a3 = Arg(4)
    a4 = Arg(5)
    a5 = Arg(6)
    a6 = Arg(7)
    a7 = Arg(8)
    a8 = Arg(9)
    a9 = Arg(10)
    Parse Value 'SOCKET'(a0,a1,a2,a3,a4,a5,a6,a7,a8,a9) With SockRc
  Res
    Return Res
```

Summary

I hope I was able to show you how you can access different kinds of servers used in the TCP/IP world from REXX. Most of the other server services available use access methods similar to HTTP. Since the documentation for the protocols is freely available in the Internet you can easily create clients matching your own needs. The following popular protocols come immediately to mind for further REXX programs:

- Network News Server Protocol
- Time Server Protocol
- Mail Server Protocols (SMTP and POP servers)
- WhoIs Protocol
- Finger Protocol

If you want to test a protocol interactively you can use the TELNET command included with TCP/IP right out of the box. If you only specify a host to connect to then TELNET will use the default port number of 21 for the connection. However in most implementations of TELNET you can specify a port number, so if you want to connect to a POP3 mail server at port 110 for example, you would enter:

```
C:\TELNET -p 110 mypopsrvr
```

You can then try the commands available for pop servers interactively before starting to write any code to test your ideas.

Bibliography

- RFC 821 - “SIMPLE MAIL TRANSFER PROTOCOL”, available as <http://ds.internic.net/rfc/rfc821.txt>
- RFC 977 - “Network News Transfer Protocol”, available as <http://ds.internic.net/rfc/rfc977.txt>
- RFC 1305 - “Network Time Protocol (Version 3), Specification, Implementation and Analysis”, available as <http://ds.internic.net/rfc/rfc1305.txt>
- RFC 1725 - “Post Office Protocol - Version 3”, available as <http://ds.internic.net/rfc/rfc1725.txt>
- RFC 1800 - “INTERNET OFFICIAL PROTOCOL STANDARDS”, available as <http://ds.internic.net/rfc/rfc1800.txt>
- RFC 1866 - “Hypertext Markup Language - 2.0”, available as <http://ds.internic.net/rfc/rfc1866.txt>
- RFC 1945 -”Hypertext Transfer Protocol -- HTTP/1.0”, available as <http://ds.internic.net/rfc/rfc1945.txt>
- IBM Redbook “TCP/IP Tutorial and Technical Overiew”, Form number GG24-3376

More information

You can find more information on this topic in the World Wide Web at the following addresses:

- IBM REXX homepage at <http://www2.hursley.ibm.com/rexx/>
- IBM REXX tutorials homepage at <http://www2.hursley.ibm.com/rexxtut/>
- IBM Object REXX homepage at <http://www2.hursley.ibm.com/orexx/>
- Search engine for RFCs at <http://ds.internic.net/ds/dspg1intdoc.html>

The sample programs and a Postscript file of this document are available on the IBM REXX tutorials homepage.