DYALOG

The tool of thought for expert programming

Dyalog™ for Windows

TCP/IP using the

TCPSocket Object

Version: 14.0

Dyalog Limited

email: support@dyalog.com

http://www.dyalog.com

Version: 14.0

Revision: 20150120

Contents

Chapter 1:

TCP/IP Support

Introduction

The TCPSocket object provides an event-driven interface to the WinSock network API, which itself is an implementation of TCP/IP for Microsoft Windows.

The TCPSocket object allows you to communicate with other TCP/IP applications running on any computer in your network, including the World Wide Web.

It also provides the mechanism for client/server operation between two Dyalog APL workspaces.

From Version 12.0,Conga is the recommended mechanism for connecting to the internet. The code samples in this chapter have not been ported to the Unicode Edition, but continue to work in Classic Editions. For information on accessing the internet and other TCP services in the Unicode Edition, see the *Conga User Guide*.

Two types of TCP/IP connections are possible; Stream and UDP. Stream connections are by far the most commonly used, but both types are supported by Dyalog APL.

Stream Sockets

A Stream socket is a connection-based transport that is analogous to a telephone service. A Stream socket handles error correction, guarantees delivery, and preserves data sequence. This means that if you send two messages to a recipient, the messages are sure to arrive and in the sequence that you sent them. However, individual messages may be broken up into several packets (or accumulated into one), and there is no predetermined protocol to identify message boundaries. This means that Stream-based applications must implement some kind of message protocol that both ends of a connection understand and adhere to.

User Datagram Protocol (UDP)

User Datagram Protocol (UDP) is a connection-less transport mechanism that is somewhat similar to a postal service. It permits a sending application to transmit a message or messages to a recipient. It neither guarantees delivery nor acknowledgement, nor does it preserve the sequence of messages. Messages are also limited to fit into a single packet which is typically no more than 1500 bytes in size. However, a UDP message will be delivered in its entirety.

You may wonder why anybody would use a service that does not guarantee delivery. The answer is that although UDP is technically an unreliable service, it is perfectly possible to implement reliable applications on top of it by building in acknowledgements, time-outs and re-transmissions.

Clients and Servers

A Stream based TCP/IP connection has two endpoints one of which is called the *server* and the other the *client*. However, this distinction is only relevant in describing how the connection is made.

The server initiates a connection by creating a socket which is identified by its (local) IP address and port number. The server is effectively making its service available to any client that wishes to connect. Notice that the server does not, at this stage, specify in any way which client or clients it will accept.

A client connects to a server by creating its own socket, specifying the IP address and port number of the service to which it wishes to connect.

Once the connection is established, both ends are capable of sending and receiving data and the original client/server relationship need no longer apply. Nevertheless, certain protocols, such as HTTP, do maintain the client/server model for the duration of the connection.

APL as a TCP/IP Server

A Stream based APL server initiates a connection by creating a TCPSocket object whose SocketType is `'Stream'` (the default).

The service is uniquely identified on the network by the server's IP Address and port number which are specified by the LocalAddr and LocalPort properties respectively. Note that unless you have more than one network adapter in your computer, LocalAddr is normally allowed to default.

This TCPSocket object effectively defines the availability of a particular service and at this stage is known as a *listening socket* which is simply waiting for a client to connect. This is reflected by the value of its CurrentState property which is `'Listening'`.

For example:

```
      'SO'⎕WC'TCPSocket' ('LocalPort' 2001)
      'SO'⎕WG'CurrentState'
 Listening
```

When a client connects to the APL server, the state of the TCPSocket object (which is reported by the CurrentState property) changes from `'Listening'` to `'Connected'` and it generates a TCPAccept event. Note that the connection cannot be nullified by the return value of a callback function attached to this event.

At this point, you can identify the client by the value of the RemoteAddr property of the TCPSocket object. If you wish to reject a particular client, you must immediately expunge the (connected) TCPSocket and then create a new one ready for another client.

Serving Multiple Clients

Dyalog APL provides a special mechanism to enable a single server to connect to multiple clients. This mechanism is designed to accommodate the underlying operation of the Windows socket interface in the most convenient manner for the APL programmer.

What actually happens when a client connects to your server, is that Windows automatically creates a new socket, leaving the original server socket intact, and still listening. At this stage, APL has a single name (the name of your TCPSocket object) but two sockets to deal with.

As it would be inappropriate for APL itself to assign a new name to the new socket, it disassociates the TCPSocket object from its original socket handle, and re-associates it with the new socket handle. This is reflected by a corresponding change in its SocketNumber property. The original listening socket is left, temporarily, in a state where it is not associated with the name of any APL object.

Having performed these operations, APL checks to see if you have attached a callback function to the TCPAccept event. If not, APL simply closes the original listening socket. This then satisfies the simple case where the server is intended to connect with only a single client and your socket has simply changed its state from `'Listening'` to `'Connected'`.

If there *is* a callback function attached to the TCPAccept event, APL invokes it and passes it the window handle of the listening socket. What the callback must do is to create a new TCPSocket object associated with this handle. If the callback exits without doing this, APL closes the original listening socket thereby preventing further clients from connecting.

If you wish to serve multiple clients, you must continually allocate new TCPSocket objects to the listening socket in this way so that there is always one available for connection.

The following example illustrates how this is done. Note that when the callback creates the new TCPSocket object, you **must not** specify any other property except SocketNumber, Event and Data in the `⎕WC` statement that you use to create it. This is important as the objective is to associate your new TCPSocket object with the original listening socket whose IP address and port number must remain unaltered.

Example

The original listening socket is created with the name `S0` and with a callback function `ACCEPT` attached to the TCPAccept event. The `COUNT` variable is initialised to `0`. This variable will be incremented and used to generate new names for new TCPSocket objects as each client connects.

```
      COUNT←0
      'S0'⎕WC'TCPSocket' ('LocalPort' 2001)
                         ('Event' 'TCPAccept' 'ACCEPT')
```

Then, each time a client connects, the `ACCEPT` function clones the original listening socket with a sequence of new TCPSocket objects using the name `S1`, `S2`, and so forth.

```
      ∇ ACCEPT MSG
[1]     COUNT+←1
[2]     ('S',⍕COUNT)⎕WC 'TCPSocket'('SocketNumber'(3⊃MSG))
      ∇
```

APL as a TCP/IP Client

A Stream based APL client makes contact with a server by creating a TCPSocket object whose SocketType is `'Stream'` (the default), specifying the RemoteAddr and RemotePort properties which identify the server's IP Address and port number respectively. Note that the client must know the identity of the server in advance.

For example:

```
IP←'193.32.236.43'
'C0'⎕WC'TCPSocket'('RemoteAddr 'IP)
                  ('RemotePort' 2001)
```

If the values of the RemoteAddr and RemotePort properties match the IP address and port number of any listening socket on the network, the association is made and the client and server sockets are connected.

When the connection succeeds, the state of the client TCPSocket object (which is reported by the CurrentState property) changes from `'Open'` to `'Connected'` and it generates a TCPConnect event. Note that the connection cannot be nullified by the return value of a callback function.

Host and Service Names

Although basic TCP/IP sockets must be identified by IP addresses and port numbers, these things are more commonly referred to by host and service names.

For example, the AltaVista web search engine is more easily identified and remembered by its name *www.altavista.com* than by any one of its IP addresses.

Port numbers are also often referred to by service names which are more convenient to remember. Furthermore, port numbers, even the so-called *well-known port numbers*, sometimes change, and your application will be more robust and flexible if you use names rather than hard-coded port numbers.

The WinSock API provides functions to resolve host names to IP addresses and service names to port numbers and these facilities are included in the Dyalog APL TCP/IP support.

Name resolution, in particular the resolution of host names, is performed *asynchronously*. This means that an application requests that a name be resolved, and then receives a message some time later reporting the answer. The asynchronous nature of name resolution is reflected in the way it is handled by Dyalog APL. Note that in certain cases, the resolution of a host name may take several seconds.

Each of the properties LocalPort, RemotePort, LocalAddr and RemoteAddr has a corresponding *name* property, i.e. LocalPortName, RemotePortName, LocalAddrName and RemoteAddrName. When you create a TCPSocket object, you may specify one or the other, but not both. For example, wherever you would use RemoteAddr, you may use RemoteAddrName instead.

If you use a *name* property, when you create a TCPSocket object, the TCPSocket will raise a TCPGotAddr or TCPGotPort event when the name is resolved to an IP address or a port number respectively. There is no need to take any action when these events are raised, so there is no specific need to attach callback functions. However, it may be useful to do so in order to track the progress of the requested connection.

The use of RemoteAddrName and TCPGotAddr is illustrated by the `BROWSER.QUERY` function that is described in the next Chapter.

Sending and Receiving Data

Once your TCPSocket object is connected, you can send and receive data. It makes no difference whether it was originally a server or a client; the mechanisms for data transfer are the same.

The *type* of data that you can send and receive is defined by the Style property which was established when you created the TCPSocket object. The default Style is `'Char'` which allows you to send and receive character vectors. Conversion to and from your `⎕AV` is performed automatically.

If you choose to set Style to `'Raw'`, you can send and receive data as integer vectors whose values are in the range -127 to 255. This allows you to avoid any character translation.

If you set Style to `'APL'`, you may transmit and receive arbitrary arrays, including arrays that contain `⎕OR`'s of namespaces. Furthermore, however the data is actually fragmented by TCP/IP, an array transmitted in this way will appear to be sent and received in single atomic operation. Data buffering is handled automatically by APL itself. Style `'APL'` is normally only appropriate for communicating between two Dyalog APL sessions. Note however, that there is no mechanism to ensure that both ends of the connection use the same Style.

To send data, you execute the TCPSend method. For example, the following expression will transmit the string "Hello World" to the remote task connected to the TCPSocket object `SO`:

```
2 ⎕NQ'SO' 'TCPSend' 'Hello World'
```

To receive data, you must attach a callback function to the TCPRecv event. Note that for a Stream connection you are not guaranteed to receive a complete message as transmitted by the sender. Instead, the original message may be received as separate packets or several messages may be received as a single packet. This means that you must perform your own buffering and you must implement a specific protocol to recognise message boundaries.

Output Buffering

When you use TCPSend to transmit a block of data, APL copies the data into a buffer that it allocates *outside* your workspace from Windows memory. APL then asks TCP/IP to send it.

However, the amount of data that can be transmitted in one go is limited by the size of various TCP/IP buffers and the speed of the network. Unless the block is very small, the data must be split up and transmitted bit by bit in pieces. This process, which is handled by APL in the background, continues until the entire data block has been transmitted. It could be several seconds or even minutes after you execute TCPSend before the entire block of data has been sent from your PC.

If in the meantime you call TCPSend again, APL will allocate a second buffer in Windows memory and will only try to send the second block of data when all of the first block has been transmitted.

If you call TCPSend repeatedly, APL will allocate as many buffers as are required. However, if you attempt to send too much data too quickly, this mechanism will fail if there is insufficient Windows memory or disk space to hold them.

If you need to transmit a very large amount of data, you should break it up into chunks and send them one by one. Having sent the first chunk, you can tell when the system is ready for the next one using the TCPReady event. This event is reported when the TCP/IP buffers are free *and* when there is no data waiting to be sent in the internal APL buffers. You should therefore attach a callback, whose job is to send the next chunk of data, to this event.

Note that a further level of buffering occurs in the *client* if the Style property of the TCPSocket is set to `'APL'`. This is done to prevent the partial reception of an APL array which would represent an invalid data object.

User Datagram Protocol (UDP) and APL

You may communicate with another application with User Datagram Protocol (UDP) by creating a TCPSocket object whose SocketType is `'UDP'`. For two APL applications to exchange data in this way, each one must have a UDP TCPSocket.

You make a UDP socket by creating a TCPSocket object, specifying the LocalAddr and LocalPort properties, and setting the SocketType to `'UDP'`. Unless your computer has more than one network card (and therefore more than one IP address), it is sufficient to allow LocalAddr to assume its default value, so in practice, only the port number is required. For example:

```
      'SO' ⎕WC 'TCPSocket' ('LocalAddr' '') 2001
               ('SocketType' 'UDP')
      'SO'⎕WG'CurrentState'
Bound
```

Once you have created a UDP TCPSocket, it is ready to send and receive data.

To send data to a recipient, you use the TCPSend method, specifying its RemoteAddr and RemotePort. The data will only be received if the recipient has a UDP socket open with the corresponding IP address and port number. However, note that there is no *absolute guarantee* that the recipient will ever get the message, nor, if you send several messages, that they will arrive in the order you sent them.

For example, the following statement might be used to send the character string `'Hello'` to a UDP recipient whose IP address is 123.456.789.1 and whose port number is 2002:

```
      2 ⎕NQ 'SO' 'TCPSend' 'Hello' '123.456.789.1' 2002
```

Note that the maximum length of a UDP data packet depends upon the type of your computer, but is typically about 1500 bytes.

To receive data from a UDP sender, you must attach a callback to the TCPRecv event. Then, when the data is received, your callback function will be invoked. The event message passed as the argument to your callback will contain not only the data, but also the IP address and port number of the sender.

For example, if you created a TCPSocket called **S1** as follows:

```
      'S1' ⎕WC 'TCPSocket' ('LocalAddr' '') 2002
               ('SocketType' 'UDP')
               ('Event' 'TCPRecv' 'RECEIVE')
```

Where the callback function **RECEIVE** is as follows:

```
     ∇ RECEIVE MSG
[1]    DISPLAY MSG
     ∇
```

the following message would be displayed in your Session when the message **'Hello'** was received from a sender whose IP address is 193.32.236.43 and whose port number is 2001.

```
.→-------------------------------------------.
| .→-. .→------. .→----. .→------------.      |
| |S1| |TCPRecv| |Hello| |193.32.236.43| 2001 |
| '--' '-------' '-----' '-------------'      |
'∈-------------------------------------------'
```

Client/Server Operation

We have seen how Dyalog APL may act as a TCP/IP server and as a TCP/IP client. It follows that full client/server operation is possible whereby an APL client workspace can execute code in an APL server workspace on the same or on a different computer.

A deliberately simple example of client/server operation is provided by the workspace `samples\tcpip\rexec.dws` whose operation is described below.

A more complex example, which implements a client/server APL component file system, is illustrated by the `samples\tcpip\qfiles.dws` workspace. See `DESCRIBE` in this workspace for details.

REXEC contains a small namespace called `SERVER`.

To start a server session, start Dyalog APL, and type:

```
)LOAD REXEC
SERVER.RUN
```

To use the server from an APL client, start Dyalog APL (on the same computer or on a different computer), and type:

```
)LOAD REXEC
IP SERVER.EXECUTE expr
```

where `IP` is the IP Address of the server computer and `expr` is a character vector containing an expression to be executed.

If you are testing this workspace using two APL sessions on the same computer, you can use either `'127.0.0.1'` or the result of the expression `(2 ⎕NQ '.' 'TCPGetHostID')` for `IP`. This expression simply obtains the IP Address of your computer. Note however, that you **do** have to have an IP Address for this to work.

The RUN function

```
     ∇ RUN;CALLBACKS
[1]    ⎕EX↑'TCPSocket'⎕WN''
[2]    CALLBACKS←⊂('Event' 'TCPAccept' 'ACCEPT')
[3]    CALLBACKS,←⊂('Event' 'TCPRecv' 'RECEIVE')
[4]    COUNT←0
[5]    'SO'⎕WC'TCPSocket' ''PORT('Style' 'APL'),CALLBACKS
     ∇
```

`RUN[1]` expunges all TCPSocket objects that may be already defined. This is intended only to clear up after a potential error.

`RUN[2-3]` set up a variable `CALLBACKS` which associates various functions with various events.

`RUN[4]` initialises a variable `COUNT` which will be incremented and used to name new TCPSocket objects as each client connects. `COUNT` is global within the `SERVER` namespace.

`RUN[5]` creates the first TCPSocket server using your default IP address and the port number specified by the `PORT` variable (5001). Note that the `Style` property is set to `'APL'` so that data is transmitted and received in internal APL format. Furthermore, however each message gets fragmented by TCP/IP, it will always appear to be sent and received in an atomic operation. There is no need for the client to do any buffering.

Once the server has been initiated, the next stage of the process is that a client makes a connection. This is handled by the `ACCEPT` callback function.

The ACCEPT function

```
     ∇ ACCEPT MSG;SOCK;EV
[1]    COUNT←COUNT+1
[2]    SOCK←'SocketNumber'(3⊃MSG)
[3]    EV←'Event'((⊃MSG)⎕WG'Event')
[4]    ('S',⍕COUNT)⎕WC'TCPSocket'SOCK EV
     ∇
```

The ACCEPT function is invoked when the TCPAccept event occurs. This happens when a client connects to the server.

Its argument MSG, supplied by APL, is a 3-element vector containing:

MSG[1]	The name of the TCPSocket object
MSG[2]	The name of the event ('TCPAccept')
MSG[3]	The socket handle for the original listening socket

ACCEPT[1] increments the COUNT variable. This variable is global to the SERVER namespace and was initialised by the RUN function.

ACCEPT[4] makes a new TCPSocket object called Sxx, where xx is the new value of COUNT. By specifying the socket handle of the original listening socket as the value of the SocketNumber property for the new object, this effectively clones the listening socket. Note that the cloned socket inherits 'Style' 'APL'. For further discussion of this topic, see Serving Multiple Clients.

The RECEIVE function

```
     ∇ RECEIVE MSG;RSLT
[1]    :Trap 0
[2]       RSLT←0('#'⍎(3⊃MSG))
[3]    :Else
[4]       RSLT←⎕EN
[5]    :EndTrap
[6]    2 ⎕NQ(⊃MSG)'TCPSend'RSLT
     ∇
```

The `RECEIVE` function is invoked when the TCPRecv event occurs. This happens whenever an APL array is received from a client. Note that it is guaranteed to receive an entire APL array in one shot because the Style property of the TCPSocket object is set to `'APL'`.

Its argument `MSG` , supplied by APL, is a 5-element vector containing:

`MSG[1]`	The name of the TCPSocket object
`MSG[2]`	The name of the event (`'TCPRecv'`)
`MSG[3]`	The data
`MSG[4`	IP address of the client
`MSG[5]`	Port number of the client

`RECEIVE[1-5]` executes the expression `(3⊃MSG)` received from the client. Assuming it succeeds, `RSLT` is a 2-element vector containing a zero followed by the result of the expression. If the execute operation fails for any reason, `RSLT` is set to the value of `⎕EN` (the error number).

`RECEIVE[6]` transmits the result back to the client.

The EXECUTE function

```
     ∇ RSLT←SERVER_IP EXECUTE EXPR;P;SOCK
[1]   ⍝ Execute expression in server
[2]
[3]   P←⊂'TCPSocket'
[4]   P,←⊂'RemoteAddr'SERVER_IP  ⍝ IP Address
[5]   P,←⊂'RemotePort'PORT       ⍝ Port Number
[6]   P,←⊂'Style' 'APL'
[7]   P,←⊂'Event'('TCPRecv' 1)('TCPClose' 1)('TCPError' 1)
[8]   'SOCK'⎕WC P
[9]
[10]  2 ⎕NQ'SOCK' 'TCPSend'EXPR ⋄ RSLT←⎕DQ'SOCK'
[11]
[12]  :Select 2⊃RSLT
[13]  :Case 'TCPRecv'
[14]       RSLT←3⊃RSLT
[15]       :If 0=⊃RSLT
[16]          RSLT←2⊃RSLT
[17]       :Else
[18]          ('Server: ',⎕EM RSLT)⎕SIGNAL RSLT
[19]       :EndIf
[20]  :Case 'TCPError'
[21]      ('Server Error: ',,⎕FMT 2↓RSLT)⎕SIGNAL 201
[22]  :Else
[23]      'Unknown Server Error'⎕SIGNAL 201
[24]  :EndSelect
     ∇
```

This function is executed by a client APL session. Its right argument is a character vector containing an expression to be executed. Its left argument is the IP Address of a server APL session in which the expression is to be run. The server session may be running on the same computer or on a different computer on the network.

`EXECUTE[3-8]` makes a client TCPSocket object called `SOCK` for connection to the specified server IP address and port number `PORT`. Note that the `Style` property is set to `'APL'` so that data is transmitted and received in internal APL format. Furthermore, however each message gets fragmented by TCP/IP, it will always appear to be sent and received in an atomic operation. There is no need for the client to do any buffering.

The Event property is set so that events TCPRecv, TCPClose and TCPError will terminate the `⎕DQ`. In this case, this is easier than using callback functions.

`EXECUTE[10]` transmits the expression to the server for execution and then `⎕DQ`s the socket. As the only events enabled on the socket are TCPRecv, TCPClose and TCPError it effectively *waits* for one of these to occur. When one of these events does happen, the `⎕DQ` terminates, returning the corresponding event message as its result.

The reason for using a diamond expression is to ensure that the TCPRecv, TCPClose or TCPError event will not be fired before the □DQ was called.

A second point worth noting is that the TCPSend request is automatically queued until the socket gets connected. In this case, there is no need to trigger the TCPSend from a callback on the TCPConnect event.

`EXECUTE[12-]` process the TCPRecv, TCPClose or TCPError event that was generated by the socket. If the operation was successful, `RSLT[2]` contains `'TCPRecv'` and `RSLT[3]` contains a zero followed by the result of the expression.

Chapter 2:

APL and the Internet

Introduction

This chapter describes the use of TCPSocket objects to access the internet. From Version 12.0, Conga is the recommended mechanism for connecting to the internet. The code samples in this chapter have not been ported to the Unicode Edition, but continue to work in Classic Editions. For information on accessing the internet and other TCP services in the Unicode Edition, see the *Conga User Guide*.

A complete description of how Web browsers and servers work is beyond the scope of this document. Nevertheless, the following basic introduction should prove a useful introduction before trying to write a server or client application in Dyalog APL.

A Web server is simply a TCP/IP server that adheres to a particular protocol known as Hypertext Transfer Protocol (HTTP). Every request for a document from a Web browser to a Web server is a new connection. When a Web browser requests an HTML document from a Web server, the connection is opened, the document transferred, and the connection closed.

The Web server advertises its availability by opening a Stream socket. Conventionally, it uses Port Number 80 although other port numbers may be used if and when required.

A client (normally referred to as a *browser*) connects to a server and immediately sends it a *command*. A command is a text string containing sub-strings separated by CR,LF pairs and terminated by CR,LF (an empty line). This terminator is an **essential** part of the protocol as it notifies the server that the entire command has been received. An example of a command sent by Netscape Navigator 3.0 Gold is:

```
GET / HTTP/1.0<CR,LF>
Proxy-Connection: Keep-Alive<CR,LF>
User-Agent: Mozilla/3.0Gold (Win95; I) <CR,LF>
Host: pete.dyalog.com<CR,LF>
Accept: image/gif, image/x-xbitmap, image/jpeg,
                   image/pjpeg, */*<CR,LF>
<CR,LF>
```

The first line of the command is a statement that tells the server what the client wants. The simplest statement is of the form GET <url>, which instructs the server to retrieve a particular document. In the example, <url> is "/" which is a relative Universal Resource Locator (URL) that identifies the home page of the current server. The client may also specify the level of http protocol that it understands as a second parameter. In the above example, the client is requesting HTTP version 1.0. Subsequent statements provide other information that may be useful to the server.

The server receives the command, actions it, and then sends back the result; in this case, the content of the Web page associated with the given URL. Using the original HTTP version 1.0 protocol, the server then **closes** the TCP/IP socket. This act informs the client that all of the data has been received and that the entire transaction is complete.

Today's web servers commonly use HTTP 1.1 which supports persistent connections. This means that the socket may not be closed, but is instead left open (for a time) for potential re-use. This behaviour is specified by *Connection: Keep-Alive HTTP headers* which are beyond the scope of this discussion. However, to support persistent connections, even the simplest client must be able to detect that the transaction is complete in some other way. A simple solution, as implemented in the `BROWSER.QUERY` function, is to look for the HTML end-tag.

The protocol can therefore be summarised as:

a. Client connects to server
b. Client sends command (terminated by CR,LF)
c. Server sends requested data to client
d. Server disconnects from client (HTTP 1.0 only)

A Web page normally contains text and embedded hyperlinks which connect it to other WWW pages. When the user activates a hyperlink, the browser connects to the corresponding server and requests the relative URL.

However, if you are using a secure proxy server, as most corporate users do, the browser connects repeatedly to your proxy (rather than to specific servers) and requests the *absolute* URL (which contains the name of the server) instead.

Writing a Web Client

A sample Web client is provided in the BROWSER namespace in the workspace samples\tcpip\www.dws.

Before you can use BROWSER.QUERY you must be connected to the Internet. See *APL and the Internet* for details.

The main function is BROWSER.QUERY. This function is intended to be used in one of two ways:

Using a Proxy Server

If you are connected to the Internet through a secure proxy server or *firewall* (as is common in many commercial organisations), you may **only** connect to your firewall; you cannot connect directly to any other server. Effectively, the **only external** IP address to which you may connect a TCPSocket as a client is the IP address of your firewall.

In this case, you should set the values of the variables BROWSER.IP_ADDRESS and BROWSER.PORT_NUMBER to the IP address and port number of your firewall.

The right argument to BROWSER.QUERY is a character string that includes the name of the web site or server as part of the query. For example, the following statement will retrieve the Microsoft home page:

```
      BROWSER.QUERY'GET http://www.microsoft.com/'
```

Using a Direct Connection

If you are directly connected to the Internet or you use dial-up networking to connect to an Internet provider, you may create TCPSocket objects that are directly connected to any server on the Internet.

In this case, the *left* argument to the function is the address and port number of the server to which you wish to connect (the port number is optional and defaults to 80). The *right* argument is the command that you wish the server to execute. Furthermore, the address may be expressed as the *IP address* of the server or as the *name* of the server.

For example, to obtain the Microsoft home page :

```
      '207.46.192.254'BROWSER.QUERY'GET /'
or
      'www.microsoft.com'BROWSER.QUERY'GET /'
```

The result of the query is not returned by the BROWSER.QUERY function, but is instead obtained from the server *asynchronously* by callback functions and then deposited in the variable BROWSER.HTML. In this example, the call-backs report the progress of the transaction as shown below. This approach is perhaps unusual in APL, but it perfectly illustrates the event-driven nature of the process.

Using a Firewall

```
      BROWSER.QUERY 'GET http://www.microsoft.com'
Connected to 193.32.236.22
... Done
Received 39726 Bytes
Response is in:
#.BROWSER.HTML
```

Using a Direct Connection

```
      'www.microsoft.com' BROWSER.QUERY 'GET /'
www.microsoft.com resolved to IP Address 207.46.192.254
Connected to 207.46.192.254
... Done
Received 39726 Bytes
Response is in:
#.BROWSER.HTML
```

There are two points to note. In the first case (using a firewall) the IP address reported is the IP address of your firewall. In the second case, there is an additional first step involved as the name of the server is resolved to its IP address (note too that this web site provides a number of IP addresses).

To keep the examples simple, BROWSER.QUERY has been written to handle only a single query at a time. Strictly speaking, it could initiate a second or third query before the result of the first had been received. This would merely entail creating multiple sockets instead of a single one.

The various functions in the BROWSER namespace are as follows:

QUERY	User function to initiate a Web query
GOTADDR	callback: reports name resolution (server name to IP address)
CONNECT	callback: handles the connection to the server
RECEIVE	callback: collects the data packets as they arrive from the server
CLOSE	callback: stores the result of the query and expunges TCPSocket
ERROR	callback: handles errors

The QUERY function

```
     ∇ {LARG}QUERY QRY;IP;PN;CALLBACKS;NS;P;SERVER
[1]   ⍝ Perform world wide web query
[2]    :If 0=⎕NC'LARG'
[3]        IP←IP_ADDRESS
[4]        PN←PORT_NUMBER
[5]        QRY,←' HTTP/1.0',⎕AV[4 3 4 3]
[6]    :Else
[7]        :If (¯2≡≡LARG)^(,2)≡⍴LARG
[8]            IP PN←LARG
[9]        :Else
[10]           IP PN←LARG 80
[11]       :EndIf
[12]       QRY,←' HTTP/1.1',⎕AV[4 3],'Host:',IP,4⍴⎕AV[4 3]
[13]   :EndIf
[14]
[15]  ⍝ Server specified by name or IP address ?
[16]   :If ^/IP∈'. ',⎕D
[17]       SERVER←('RemoteAddr'IP)
[18]   :Else
[19]       SERVER←('RemoteAddrName'IP)
[20]   :EndIf
[21]
[22]   NS←(''⎕NS''),'.'
[23]   CALLBACKS←⊂('TCPGotAddr'(NS,'GOTADDR'))
[24]   CALLBACKS,←⊂('TCPConnect'(NS,'CONNECT'))
[25]   CALLBACKS,←⊂('TCPRecv'(NS,'RECEIVE'))
[26]   CALLBACKS,←⊂('TCPClose'(NS,'CLOSE'))
[27]   CALLBACKS,←⊂('TCPError'(NS,'ERROR'))
[28]
[29]  ⍝ Expunge TCPSocket in case of previous error
[30]   ⎕EX'SO'
[31]
[32]  ⍝ Make new SO namespace containing QRY
[33]   'SO'⎕NS'QRY'
[34]  ⍝ Then make SO a TCPSocket object
[35]   P←SERVER('RemotePort'PN)('Event'CALLBACKS)
[36]   SO.⎕WC('TCPSocket'),P ⋄ ⎕DQ'SO'
     ∇
```

The first 13 lines of the function process the optional left argument and are largely unremarkable.

However, note that if you are using a firewall or proxy (no left argument), `QUERY[5]` adds a header to request HTTP/1.0 protocol. If you are using a direct connection, `QUERY[12]` instead adds a request for HTTP/1.1 protocol and a *Host* header (which in this case it knows). A Host header is mandatory for HTTP/1.1 and your firewall may add one for you.

QUERY[16-20] sets the variable SERVER to specify either RemoteAddr or RemoteAddrName according to whether the user specified the IP address or the name of the server.

QUERY[22-27] set up a variable CALLBACKS which associates various functions with various events. Full path-names are used for the callback functions because they will be associated by a ⎕WC statement that is executed *within* the SO namespace.

QUERY[30] expunges the object SO. This is done only in case an error occurred previously and the object has been left around.

QUERY[33] makes a new namespace called SO and copies the variable QRY into it. This is done because the query cannot be submitted to the server until after a connection has been made. Thus the query is encapsulated within the TCPSocket object to make it available to the callback function CONNECT that will handle this event. A less elegant solution would be to use a global variable.

QUERY[36] creates a new client TCPSocket object associated with the namespace SO.

A more obvious solution would be..

```
[33]    'SO' ⎕WC(⊂'TCPSocket'),P
[34]    SO.QRY←QRY
```

However, this is inadvisable because TCP events can occur as soon as the object has been created. If the TCPConnect event fired before QUERY[34] could be executed, the CONNECT callback function would fail with a VALUE ERROR because SO.QRY would not yet be defined. This is also a reason for attaching the callback functions in the ⎕WC statement and not in a subsequent ⎕WS. You do not want the TCPConnect event to fire when there is no callback attached.

Note that these timing issues are only relevant because BROWSER.QUERY is a user-called function and not a callback. If it were a callback, APL would automatically queue the incoming TCP events until it (the callback) had terminated.

Depending upon how the function was called, the next part of the process is handled by the GOTADDR or the CONNECT callback.

The GOTADDR callback function

```
     ∇ GOTADDR MSG;NAME;IP
[1]    NAME IP←(⊃MSG)⎕WG'RemoteAddrName' 'RemoteAddr'
[2]    NAME,' resolved to IP Address ',IP
     ∇
```

The GOTADDR callback function is invoked when the TCPGotAddr event occurs. This happens if the RemoteAddrName was specified when the TCPSocket was created.

The function merely obtains the name and newly resolved IP address of the server from the RemoteAddrName and RemoteAddr properties of the TCPSocket object and reports them in the session.

The CONNECT callback function

```
     ∇ CONNECT MSG
[1]    ⎕CS⊃MSG
[2]    'Connected to ',⎕WG'RemoteAddr'
[3]    BUFFER←''
[4]    2 ⎕NQ'' 'TCPSend' QRY
     ∇
```

The CONNECT function is invoked when the TCPConnect event occurs. This happens when the server accepts the client.

Its argument MSG, supplied by APL, is a 2-element vector containing:

MSG[1]	The name of the TCPSocket object
MSG[2]	The name of the event ('TCPConnect')

CONNECT[1] changes to the namespace of the TCPSocket object.

CONNECT[2] displays the IP address of the server to which the client has successfully connected. This is obtained from the RemoteAddr property of the TCPSocket object.

CONNECT[3] initialises a variable BUFFER which will be used to collect incoming data from the server. Notice that as the function has changed to the TCPSocket namespace, this variable is encapsulated within it rather than being global.

CONNECT[3] uses the TCPSend method to send the query (the QRY variable was encapsulated within the TCPSocket object when it was created by the QUERY function) to the server.

The next part of the process is handled by the RECEIVE callback.

The RECEIVE callback function

```
     ∇ RECEIVE M
[1]    ⎕CS⊃M
[2]    BUFFER,←3⊃M
[3]    :If ∨/'</html>'⍷##.lcase 3⊃M
[4]        (⊃M)##.⎕WS'TargetState' 'Closed'
[5]    :EndIf
     ∇
```

The `RECEIVE` function is invoked when the TCPRecv event occurs. This happens whenever a packet of data is received. As the response from the server can be of arbitrary length, the job of the `RECEIVE` function is simply to collect each packet of data into the `BUFFER` variable.

Its argument `MSG` , supplied by APL, is a 3-element vector containing:

`MSG[1]`	The name of the TCPSocket object
`MSG[2]`	The name of the event (`'TCPRecv'`)
`MSG[3]`	The data
`MSG[4`	IP address of the client
`MSG[5]`	Port number of the client

`RECEIVE[1]` changes to the namespace of the TCPSocket object.

`RECEIVE[2]` catenates the data onto the variable `BUFFER`, which is encapsulated within the TCPSocket object.

`RECEIVE[3]` checks for the presence of an *end-of document* HTML tag, which indicates that the entire page has arrived, and if so

`RECEIVE[4]`s ets the TargetState property of the TCPSocket to `'Closed'`. This initiates the closure of the socket. Although a client can close a socket by expunging the associated TCPSocket namespace, our data (`BUFFER`) is in this namespace and we do not want to lose it.

Note that it is necessary for `RECEIVE` to detect the end-of-document in this way, so as to support HTTP/1.1 protocol.

The CLOSE callback function

```
     ∇ CLOSE MSG
[1]    HTML←(⊃MSG)⍎'BUFFER'
[2]    ⎕EX⊃MSG
[3]    '... Done'
[4]    'Received ',(⍕⍴HTML),' Bytes'
[5]    'Response is in:'
[6]    (''⎕NS''),'.HTML'
     ∇
```

The `CLOSE` function is invoked when the TCPClose event occurs. This happens when the server closes the socket. If the protocol is HTTP/1.0, this will be done immediately after the server has sent the data in response to the query. If the protocol is HTTP/1.1, the closure may be performed at the request of the client.

Note that the data has been buffered by the `RECEIVE` function as it arrived.

Its argument `MSG` , supplied by APL, is a 2-element vector containing:

`MSG[1]`	The name of the TCPSocket object
`MSG[2]`	The name of the event (`'TCPClose'`)

`CLOSE[1]`copies the contents of the `BUFFER` variable (that is local to the TCPSocket object) into the `HTML` variable that is global within the current (`BROWSER`) namespace. (Clearly this design would be inadequate if `BROWSER` was extended to support multiple concurrent queries.)

`CLOSE[2]`expunges the TCPSocket object

`CLOSE[3-6]`reports a successful end to the query and displays the size of the result.

The ERROR callback function

```
     ∇ R←ERROR MSG
[1]    DISPLAY MSG
[2]    ⎕EX⊃MSG
[3]    R←0
     ∇
```

The `ERROR` function is invoked if and when a TCP/IP error occurs.

Its argument `MSG` , supplied by APL, is a 4-element vector containing:

`MSG[1]`	The name of the TCPSocket object
`MSG[2]`	The name of the event (`'TCPError'`)
`MSG[3]`	An error number
`MSG[4`	An error message

`ERROR[1]` displays the contents of `MSG` using the `DISPLAY` function.

`ERROR[2]` expunges the TCPSocket object (it is no longer usable)

`ERROR[3]` returns a 0. This tells APL *not* to perform the normal default processing for this event, which is to display a message box.

Writing a Web Server

A sample Web server is provided in the SERVER namespace in the workspace samples\tcpip\www.dws. This is a deliberately over-simplified example to illustrate the principles involved. It is capable of handling concurrent connections from several clients, but (for simplicity) does not use multi-threading.

The main function is SERVER.RUN which is niladic and initialises the APL Web server using your default IP Address and port number 81.

To use the server, you must start a Web browser such as Firefox or Microsoft Internet Explorer. You may do this on another PC on your network or on your own PC. If so, you will probably find it most convenient to position your Dyalog APL Session Window and your browser window alongside one another.

To connect to the server, simply enter your user name (or your IP address, or "127.0.0.1" or "localhost") followed by 81 in the appropriate field in your browser, and then press Enter. For example:

http://localhost:81

When you press Enter, your browser will try to connect with a server whose IP address is your IP address and whose port number is 81; in short, the APL server. Upon connection, the following messages (but with different IP addresses) will appear in your Session window:

```
      SERVER.RUN
Connected to 193.32.236.22
URL Requested:
Connected to 193.32.236.22
URL Requested: images/dyalog.gif
```

and the Dyalog APL home page will appear in your browser. This has in fact been supplied by your APL server.

The functions in the SERVER namespace are as follows:

RUN	user function to initiate an APL Web server
ACCEPT	callback which handles client connections
RECEIVE	callback which handles client commands
ERROR	callback which handles errors

The RUN function

```
     ∇ RUN;CALLBACKS
[1]    ⎕EX↑'TCPSocket'⎕WN''
[2]    CALLBACKS←⊂('Event' 'TCPAccept' 'ACCEPT')
[3]    CALLBACKS,←⊂('Event' 'TCPRecv' 'RECEIVE')
[4]    CALLBACKS,←⊂('Event' 'TCPError' 'ERROR')
[5]    COUNT←0
[6]    'SO'⎕WC'TCPSocket' '' 81,CALLBACKS
     ∇
```

`RUN[1]` expunges all TCPSocket objects that may be already defined. This is intended only to clear up after a potential error.

`RUN[2-4]` set up a variable `CALLBACKS` which associates various functions with various events.

`RUN[5]` initialises a variable `COUNT` which will be incremented and used to name new TCPSocket objects as each client connects. `COUNT` is global within the `SERVER` namespace.

`RUN[6]` creates the first TCPSocket server using your default IP address and port number 81.

Once the server has been initiated, the next stage of the process is that a client makes a connection. This is handled by the `ACCEPT` callback function.

The ACCEPT callback function

```
     ∇ ACCEPT MSG;EV
[1]    COUNT←COUNT+1
[2]    EV←'Event'((⊃MSG)⎕WG'Event')
[3]    ('S',⍕COUNT)⎕WC'TCPSocket'('SocketNumber'(3⊃MSG))EV
[4]    ⎕CS⊃MSG
[5]    'Connected to ',(⎕WG'RemoteAddr')
[6]    BUFFER←⎕AV[4 3]
     ∇
```

The `ACCEPT` function is invoked when the TCPAccept event occurs. This happens when a client connects to the server.

Its argument `MSG` , supplied by APL, is a 3-element vector containing:

`MSG[1]`	The name of the TCPSocket object
`MSG[2]`	The name of the event (`'TCPAccept'`)
`MSG[3]`	The socket handle for the original listening socket

`ACCEPT[1]` increments the `COUNT` variable. This variable is global to the `SERVER` namespace and was initialised by the `RUN` function.

`ACCEPT[3]` makes a new TCPSocket object called Sxx, where xx is the new value of `COUNT`. By specifying the socket handle of the original listening socket as the value of the SocketNumber property for the new object, this effectively clones the listening socket. For further discussion of this topic, see Serving Multiple Clients.

`ACCEPT[4]` changes to the namespace of the TCPSocket object, that is now connected to a client.

`ACCEPT[5]` displays the message `Connected to xxx.xxx.xxx.xxx`, the IP address of the client, which is obtained from the value of the RemoteAddr property.

`ACCEPT[6]` initialises a variable `BUFFER` to `⎕AV[4 3]` (CR,LF). This variable is global to the `SERVER` namespace and is used by the `RECEIVE` callback function to accumulate the command that is transmitted by the client. This happens next.

The RECEIVE callback function

```
     ∇ RECEIVE MSG;CMD;OLD;URL;FILE;DATA
[1]    OLD←⎕CS⊃MSG
[2]    BUFFER,←3⊃MSG
[3]    :If ⎕AV[4 3 4 3]≠¯4↑BUFFER ⍝ Have we all?
[4]        :Return
[5]    :EndIf
[6]
[7]    CMD←2↓¨(⎕AV[4 3]⍷BUFFER)⊂BUFFER
[8]    CMD←⊃CMD ⍝ Ignore everything except client request
[9]    ⎕CS OLD
[10]   :If 'GET /'≡5↑CMD
[11]       URL←5↓CMD
[12]       URL←(¯1+URL⍳' ')↑URL
[13]       ⎕←'URL Requested: ',URL
[14]       :If 0=⍴URL ⋄ URL←'index.htm' ⋄ :EndIf
[15]       URL←(-'.html'≡¯5↑URL)↓URL
[16]       FILE←(2 ⎕NQ'.' 'GetEnvironment') 'Dyalog'
[17]       FILE,←HOME,URL
[18]       DATA←GETFILE FILE
[19]       DATA,←(0<⍴DATA)/'File not found'
[20]       2 ⎕NQ(⊃MSG)'TCPSend'DATA
[21]   :EndIf
[22]
[23]   :If 9=⎕NC⊃MSG ⋄ (⊃MSG)⎕WS'TargetState' 'Closed' ⋄ :EndIf
[24]   :If 9=⎕NC⊃MSG ⋄ ⎕DQ⊃MSG ⋄ :EndIf
     ∇
```

The `RECEIVE` function is invoked whenever a TCPRecv event occurs. This happens when a data packet is received from a client

Its argument `MSG` , supplied by APL, is a 3-element vector containing:

`MSG[1]`	The name of the TCPSocket object
`MSG[2]`	The name of the event (`'TCPRecv'`)
`MSG[3]`	The data

`RECEIVE[1]` changes to the namespace of the TCPSocket object. The name of the current namespace is stored in the local variable `OLD`.

`RECEIVE[2]` catenates the newly received data packet to the `BUFFER` variable that is encapsulated in the TCPSocket object and was initialised by the `ACCEPT` function.

`RECEIVE[3-5]` tests whether or not all of the command sent by the client has been received. This is true only if the last 4 characters of `BUFFER` are CR,LF,CR,LF. If there is more data to come, `RECEIVE` exits; otherwise it goes on to process the command.

`RECEIVE[7-8]` splits the command into sub-strings and then discards all but the first one.

`RECEIVE[9]` changes back into the `SERVER` namespace

`RECEIVE[10-11]` parses the client request for a URL. For the sake of simplicity, the request is assumed to begin with the string `'GET /'`. If not, the request is ignored.

`RECEIVE[12]` removes all trailing information that might be supplied by the browser after the URL.

`RECEIVE[14]` checks for a request for an empty URL (which equates to the home page). If so, it substitutes `index.htm` which is the name of the file containing the home page.

`RECEIVE[15]` drops the file extension of the URL if `.html` to `.htm` if required.

`RECEIVE[16]` sets the value of local variable `FILE` to the name of the directory in which Dyalog APL is installed.

`RECEIVE[17]` appends the path-name of the sub-directory `\help` and the name of the URL. `FILE` now contains the full path-name of the requested web page file.

`RECEIVE[18]` uses the utility function `GETFILE` to read the contents of the file into the local variable `DATA`.

`RECEIVE[19]` checks that the result of `GETFILE` was not empty and if so, appends an appropriate message. This would be the case if the file did not exist.

`RECEIVE[20]` uses TCPSend to transmit the contents of the file to the browser.

`RECEIVE[23-24]` closes the TCPSocket object. This is a fundamental part of the HTTP protocol because when the client socket subsequently gets closed, it knows that all of the data transmitted by the server has been received. Notice that the function does not simply expunge the socket which could result in loss of yet untransmitted data. Instead, it closes the socket by setting its TargetState property to `'Closed'`, and then (if necessary) waiting. Once all the buffered data has been transmitted, the socket will be closed and the TCPSocket object will disappear. This causes the `⎕DQ` to terminate.

For further information on the HTTP protocol, see the *Introduction* to this chapter.

Index

U

W