Module14: TCP/IP Sockets

§ 14.1 The *TCPSocket* Object

§§ 14.1.1 IP Addresses and Ports

An *Internet Protocol* (IP) address uniquely identifies a specific network card on a specific computer. Associated with an IP address are one or more ports. Communication between computers requires specification of the IP address and port number of both ends of the connection.

14.1.1.1 To discover the IP address that has been assigned to your computer network card, enter

```
      #.TCPGetHostID
```

(Remember [Options][Object Syntax][Expose Root Properties] should be checked.)

A (32-bit) IP address is written as 4 numbers between 0 and ¯1+2*8 ↳ 255, separated by dots. For example, 216.239.39.99 is one of the IP addresses of Google.com.

IP addresses beginning 10. or 172. or 192. are internally assigned within an intranet and are not reachable from outside your local area network. 127.0.0.1 is the standard IP address used for a loopback network connection. If you try to connect to 127.0.0.1, you are immediately looped back to your own machine.

A (16-bit) port number can be anything between 1 and ¯1+2*16 ↳ 65535. Internet traffic generally uses port number 80. FTP (*File Transfer Protocol*) conventionally uses port 21. SMTP (*Simple Mail Transfer Protocol*) conventionally uses port 25.

§§ 14.1.2 *SocketType* and *Style* Properties

TCP/IP stands for *Transmission Control Protocol/Internet Protocol*. It is a fundamental part of the standard protocol for communications on the internet.

Dyalog APL provides an object called a *TCPSocket* that enables access to TCP/IP communications by way of the API functions in winsock.dll.

There are two types of TCP/IP connections, both of which are supported by the *TCPSocket* object. A *UDP* (User Datagram Protocol) socket is like a postal service. A single small package is sent to an address. Packages may arrive in any order, and sometimes might not arrive at all.

The second more common type of connection is a *stream* socket, which is like a telephone service. Someone initiates a call. Once a connection is established, both parties have equal status and either party can terminate the call at any stage. Packets are received in the order in which they are transmitted. They are guaranteed to arrive - with automatic error correction.

The equivalent of waiting for a telephone call is creating a *listening* socket. The minimum information required to create a listening socket is the local port number that is to be used for communications, and the name of the new APL object.

```
      'S0'⎕WC'TCPSocket' ('LocalPort' 123)
```

The default socket type is stream, the alternative being UDP,

```
      S0.SocketType ↳ Stream
```

(Remember to make sure that ⊢*S0.⎕WX*.)
The current state of the socket is 'Listening'

```
      S0.CurrentState ↳ Listening
```

The default `Style` specifies that character data will be transmitted. This is the standard `Style` for internet traffic.

```
      S0.Style ↳ Char
```

Alternative `Style`s are *Raw* and *APL*. 'Raw' communication is via integer vectors between ¯128 and 255 (negative numbers, such as ¯50, are added to 256 and sent as, for example, 256-50=206, making the range effectively 0 to 255). `Style` 'APL' communicates via arbitrary APL arrays. The latter is only suitable for communication between two APL workspaces.

§§ 14.1.3 Workspace to Workspace Communications

In the first instance, we are going to create a listening socket in a workspace on our computer by running the following function, where function `∇show∇` is just `⎕FX'show Msg' 'Msg'` (not `show←{⍵}`).

```
      ∇ listen;w
[1]     w←⊂'Type' 'TCPSocket'
[2]     w,←⊂'LocalPort' 123
[3]     w,←⊂'Style' 'APL'
[4]     w,←⊂'Event' 'All' 'show'
[5]     'S0'⎕WC w
      ∇
```

The `Create Event` displays immediately (via `∇show∇`) the message

```
 S0  Create  1
```

The `TCPSocket` object has been assigned a socket number

```
      S0.SocketNumber ↳ 696
```

which is the Windows handle of the socket. The current state is 'Listening'. The intended final state is 'Server'

```
      S0. TargetState ↳ Server
```

as opposed to possible states 'Client' or 'Closed' for a stream socket. Setting the target state to 'Closed' is the approved method of closing a socket because then APL waits until all data has been sent before issuing a `TCPClose Event`.

```
      S0 . TargetState←'Closed'
 S0  TCPClose
```

Next we start a new instance of Dyalog APL and use the function below to create a `TCPSocket` that will connect to a listening socket on port 123. Again, function `∇show∇` is just `⎕FX'show Msg' 'Msg'`.

```
      ∇ connect;w
[1]     w←⊂'Type' 'TCPSocket'
[2]     w,←⊂'RemoteAddr' '127.0.0.1'
[3]     w,←⊂'RemotePort' 123
[4]     w,←⊂'Style' 'APL'
[5]     w,←⊂'Event' 'All' 'show'
[6]     'S1'⎕WC w
      ∇
```

The minimum information required to create a connecting socket, apart from its arbitrary APL name, is the remote address and the remote port number, which must be set to the number of the listening socket's local port. When `RemoteAddr` and `RemotePort` properties match the IP address and port number of any listening socket, the client and server sockets connect. In this simple case we use our own IP address (found from `#.TCPGetHostID`) or, equivalently, the standard loopback address 127.0.0.1 .

14.1.3.1 Run `∇listen∇` in one workspace and `∇connect∇` in another. After the initial `Create` event in the listening workspace, you should get a `Create` event in the connecting workspace, and

simultaneously, a `TCPAccept` event in the listening workspace. Check that the `TCPAccept` event is immediately followed by a `TCPConnect` event in the connecting workspaces followed by a `TCPReady` event in both workspaces.

The sockets, both of `Style` APL, are now connected through port number 123 (on the listening side). As with a telephone, once the connection has been established, the communication is peer-to-peer and neither party monopolises the connection communication resources.

Either party can send arbitrary APL arrays, including arrays that contain `⎕OR`'s of namespaces, to the other in a single atomic operation using the `TCPSend` method. For example, the 'connecting' workspace can send matrix `(3 3⍴⍳9)` to the 'listening' workspace by

```
2 ⎕NQ'S1' 'TCPSend' (3 3⍴⍳9)
```

which is reported in the 'listening' workspace through the `TCPRecv Event` as

```
S0 TCPRecv 1 2 3 127.0.0.1 1568 19313036
           4 5 6
           7 8 9
```

Or the 'listening' workspace can send vector `⎕A` to the 'connecting' workspace by

```
2 ⎕NQ'S0' 'TCPSend' ⎕A
```

which is reported in the 'connecting' workspace through the `TCPRecv` event as

```
S1 TCPRecv ABCDEFGHIJKLMNOPQRSTUVWXYZ 127.0.0.1 123 1927140
```

Erasing the socket object on either side (*eg* `S1`) closes the connection and removes the partner object (`S0`).

14.1.3.2 Change the last line in `∇listen∇` and `∇connect∇` to

```
'S..'⎕WC w ⋄ ⎕DQ'S...'
```

By running `{listen}&1` in one workspace and `{connect}&1` in the other, show that TCP messages are received independent of the thread which owns the socket. This is intentional as otherwise, if messages are not processed immediately, there is a chance that they might get lost.

If you try to run `∇listen∇` and `∇connect∇` in the same workspace but in different threads then you will get error number 10061 reported by the `TCPError` event. The interpretation of these error codes is to be found in many places on the internet, *eg* http://www.sockets.com/err_lst1.htm. In this case the connection has been refused because there is a conflict over multiple port allocations for a single process.

14.1.3.3 Write a callback on the `TCPRecv` event in both workspaces which will execute the linear APL expression being sent by the other party.

Hint: Include a line like `⍎(3⊃Msg)~⎕AV[6 9 10]`

The distributed workspace REXEC.DWS, described in the *Interface Guide*, furnishes a much more sophisticated example of remote execution by a web server.

§ 14.2 A simple character Socket

§§ 14.2.1 Connecting to a server Socket with `TCPConnect`

14.2.1.1 Create a simple character listening socket in the first workspace, which we shall call the *server*.

```
'S0'⎕WC'TCPSocket'('LocalPort' 123)
```

In a separate workspace, create another character socket, the *client*, which will connect to the server socket

```
      w←'TCPSocket'('RemoteAddr' '127.0.0.1')('RemotePort' 123)
      'S1'⎕WC w ⋄ S1.onTCPConnect←'show'
 #.S1  TCPConnect
```

Note the immediate response of the *TCPConnect* event as soon as the two sockets connect. When a connection succeeds, the *CurrentState* of the client *TCPSocket* object changes from 'Open' to 'Connected' and it generates a *TCPConnect* event.

As you can verify, it is important that the *TCPConnect* event is active as soon as the socket is created. If the *Event* is set on the line after the *⎕WC* then the response from the server can easily be lost through sloth of the client and zeal of the server.

§§ 14.2.2 Sending to the server Socket using *TCPSend*

We may use the connect event on the client to immediately send a request to the server socket. In order to see the request we set a callback on the server *TCPRecv* event.

```
      S0.onTCPRecv←'show'
```

On the *TCPConnect* event in the client we put the callback function

```
      ∇ conn Msg
[1]     2 ⎕NQ(⊃Msg)'TCPSend' 'file1'
      ∇
```

This will send the character string *'file1'* to the server as soon as the connection is achieved. So the server begins with

```
      'S0'⎕WC'TCPSocket'('LocalPort' 123)
      S0.onTCPRecv←'show'
```

And some time later the client runs

```
      w←'TCPSocket'('RemoteAddr' '127.0.0.1')('RemotePort' 123)
      'S1'⎕WC w ⋄ S1.onTCPConnect←'conn'
```

As soon as *S1* is created, the server gets the message below, where 1323 is the remote port number.

```
 #.S0  TCPRecv  file1  127.0.0.1  1323
```

§§ 14.2.3 Receiving from the server Socket with *TCPRecv*

Having received this message from a client, the server can respond to it by, for example, sending back to the client the contents of file1.

First we create some native files containing ‘valuable’ information for the client:

```
      'file1'⎕NCREATE ¯1 ⋄ 'this is file 1...'⎕NAPPEND ¯1 ⋄ ⎕NUNTIE ¯1
      'file2'⎕NCREATE ¯1 ⋄ 'THIS IS FILE 2...'⎕NAPPEND ¯1 ⋄ ⎕NUNTIE ¯1
```

Then we replace the *TCPRecv* callback on the server with function

```
      ∇ recv Msg;t;d
[1]     :If 'file1'≡3⊃Msg       ⍝ did they ask for file1?
[2]         t←'file1'⎕NTIE 0    ⍝ tie file1
[3]     :ElseIf 'file2'≡3⊃Msg   ⍝ did they ask for file2? etc...
[4]         t←'file2'⎕NTIE 0    ⍝ tie file2
[5]     :End
[6]     d←⎕NREAD t 82(⎕NSIZE ¯1)0 ⋄ ⎕NUNTIE t ⍝ read the file
[7]     2 ⎕NQ(⊃Msg)'TCPSend'd   ⍝ send the contents
      ∇
```

One further consideration before this will work is: What will the client do with the requested information sent from the server? We need at least a basic callback on the client `TCPRecv` event.

So now the server runs:

```
'S0'⎕WC'TCPSocket'('LocalPort' 123)
S0.onTCPRecv←'recv'
```

And the client runs:

```
w←'TCPSocket'('RemoteAddr' '127.0.0.1')('RemotePort' 123)
'S1'⎕WC w ⋄ S1.onTCPConnect←'conn' ⋄ S1.onTCPRecv←'show'
```

On running this last line, the immediate response from the receive event is:

```
#.S1  TCPRecv  this is file 1...  127.0.0.1  123
```

Note that the receive event must be set in the `⎕WC`, or at least on the same line as the `⎕WC`, in order to ensure that the incoming message is not lost.

14.2.3.1 Using the appropriate remote address, the remote `TCPGetHostID`, repeat the above example using two separate computers rather than two workspaces on the same computer.

§ 14.3 Some Complications

§§ 14.3.1 HTTP and HTML

The above trivial example of a character `TCPSocket` connection affords a very simple model of how the Internet is generally used at the socket level. A remote server waits in a listening state for a call. A browser connects to the server and immediately sends a request for a file – the home page say. The server receives the request and sends the file contents, which the browser then presents on the screen. The server then closes the connection.

Details of the question of exactly how the incoming information is formatted on the client screen is solved by way of HTML plain ASCII text. Server files do not contain arbitrary text strings, but rather text specified in *HyperText Markup Language*. This is a method of incorporating format information into plain text. Browsers such as Microsoft Internet Explorer, Netscape Navigator and Mozilla Firefox all recognise this format and present text thus expressed in a uniform, precise and detailed fashion.

HTML is constantly developing and the language is described in many places, for example in http://www.w3.org/TR/xhtml1/. Each page presented on a browser may be viewed, in its approximate original HTML form, by selecting menu item [View][Source] in the browser.

Not only does an Internet the browser expect HTML text, the information sent between browser (client) and server must conform to a special protocol, as the title TCP/IP - Transmission Control Protocol / Internet Protocol – would suggest. Requests and responses between browsers and servers must be wrapped in *Hypertext Transfer Protocol* (HTTP). The authoritative guide to HTTP is in the Request For Comments RFC2616 found, for example, at http://www.w3.org/Protocols/rfc2616/rfc2616.txt.

§§ 14.3.2 Buffering received Data

Data sent using `TCPSend` may be split into packets and sent sequentially. Similarly, data received from a server may come in packets. It is therefore necessary to implement a mechanism in `TCPRecv` that will reconstitute an entire message. The simplest way to do this is simply to catenate incoming data in the `TCPRecv` callback until a `TCPClose` event is encountered (because the server closed the connection) at which point the incoming data is assumed to be complete and is saved appropriately.

More specifically, HTTP defines the sequence carriage return, linefeed (CRLF) as the end of line marker. So command lines end with CRLF (*ie* `⎕AV[4 3]`). Further, at the end of the message HTTP requires an empty line followed by CRLF. Therefore the receive callback looks for CRLF CRLF at which point it knows that the end of the entire message has been reached.

§§ 14.3.3 Servicing multiple Connections

14.3.3.1 In the server workspace, write a function `∇listen∇` which sends a reply as soon as a message is received from a client.

```
      ∇ listen;w
[1]     w←,⊂'Type' 'TCPSocket'
[2]     w,←⊂'LocalPort' 123
[3]     w,←⊂'Event' 'TCPRecv' 'recvByServer'
[4]     'S0'⎕WC w
      ∇
```

where the callback on the receive event is

```
      ∇ recvByServer Msg
[1]     ⎕←3⊃Msg
[2]     2 ⎕NQ(⊃Msg)'TCPSend' 'Server says, "Goodbye Client!"'
[3]     (⍎⊃Msg).TargetState←'Closed'
      ∇
```

In the client workspace, write a function `∇connect∇` which creates a connection socket and immediately sends a message.

```
      ∇ connect;w
[1]     w←,⊂'Type' 'TCPSocket'
[2]     w,←⊂'RemotePort' 123
[3]     w,←⊂'RemoteAddr' '127.0.0.1'
[4]     w,←⊂'Event' 'TCPConnect' 'conn'
[5]     w,←⊂'Event' 'TCPRecv' 'recvByClient'
[6]     'S1'⎕WC w
      ∇
```

where the connect callback is

```
      ∇ conn Msg
[1]     2 ⎕NQ(⊃Msg)'TCPSend' 'Client says, "Hello Server!"'
      ∇
```

and the receive callback is

```
      ∇ recvByClient Msg
[1]     ⎕←3⊃Msg  ⍝ display message from server
      ∇
```

Run `∇listen∇` and `∇connect∇`. Note that the messages got through, but that at the end of the brief conversation all sockets have disappeared. No further communication is possible until the server creates a new listening socket.

On sending its reply to a request from a client, a server generally sets its `TargetState` to `'Closed'`. This closes the server socket and terminates the connection. At this point the connection is closed and further requests from the client must open a new connection. This means that the server should have a new listening socket available immediately in order to be ready for the next request.

In order to deal with this situation, `TCPSocket`s have a special mechanism to create a new listening socket as soon as a connection has been made to the current listening socket.

The `TCPConnect` event triggers in the client when the server attempts to connect to the client. The `TCPAccept` event triggers in the server when the client actually connects to the server. This event includes the socket handle in the callback message, and at this point the server has the opportunity to clone the listening socket. In the `TCPAccept` callback it is possible to create a new socket which is a clone of the current socket by setting the `SocketNumber` property of the new socket to the socket number reported in the callback. The only other properties that may be set are the `Event` and `Data` properties. This new socket then becomes a new listening socket, thus maintaining the server's ability to serve.

14.3.3.2 Add a new line in the `∇listen∇` function:

```
      w,←⊂'Event' 'TCPAccept' 'acc'
```

where the `∇acc∇` callback is

```
      ∇ acc Msg;w
[1]     w←,⊂'Type' 'TCPSocket'
[2]     w,←⊂'SocketNumber'(3⊃Msg)
[3]     w,←⊂'Event'(⍎2⊃Msg).Event
[4]    ⍝w,←⊂'Data' (⍎2⊃Msg).Data
[5]     ('S',⍕1+⍎1↓2⊃Msg)⎕WC w
      ∇
```

Run `∇listen∇` and `∇connect∇` again. Each time a client connects, the callback on `TCPAccept` clones the original listening socket with a sequence of new `TCPSocket` objects using the name `S1`, `S2`,… The server workspace always has a listening socket available and therefore `∇connect∇` may be run again and again…

14.3.3.3 Why doesn't `connect⋄connect` work? Fix it by changing the line that creates the client socket:

```
[6]     ('S',⍕COUNT←COUNT+1)⎕WC w
```

where `COUNT` is a suitably initialised global variable.

Details regarding the `TCPSocket` object and its properties and methods are to be found in the *Object Reference* and in the Dyalog 7.3 or 8.1 release notes. Detailed examples of its use can be found in the *Interface Guide*, www.dyalog.com [Products][Dyalog for Windows][TCP/IP Support], APL97.RTF available from www.dyalog.com and example workspaces in ..\samples\tcpip\ .

14.3.3.4 Please ask for the next module on **Web Servers**.