Module15: APL Web Servers

§ 15.1 Making a simple Server

It is possible to use *TCPSocket*s on your computer to host a web server that will be accessible by everyone on your network. A good example, trivialised below, is to be found in the supplied workspace ..\Samples\tcpip\www.dws, namespace *#.SERVER*, and is described in detail in the *Interface Guide*. A more robust example server is to be found in the distributed workspace ..\aplserve\server.dws and associated files.

§§ 15.1.1 Creating a listening Socket

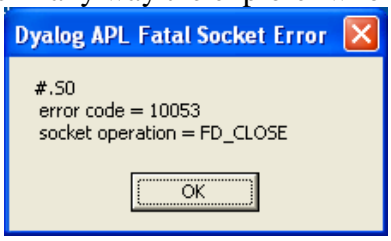
15.1.1.1 In a clear WS, create a socket on *LocalPort* 80, the default for simple web servers:

```
'S0'⎕WC'TCPSocket' ('LocalPort' 80)
```

Check that it's *CurrentState* is *'Listening'*. Now open your Internet Explorer and type **http://127.0.0.1** in the address bar and hit **Enter**. Check that the *CurrentState* is now *'Connected'*.

Had we created a socket on a different port, say 2020, typing **http://127.0.0.1:2020** into the address bar we would have established a connection to this port, but **:80** is the assumed default. (Different ports relate to different services. The IP address plus port combination uniquely identifies a web site.)

The socket connection is not much use as it stands. IE and our server remain connected indefinitely but the connection does nothing useful. If you go to the IE address bar and hit **Enter** again then an error box pops up in APL. It tells us that we did not get the communication protocol right. This is not surprising as our server socket did not acknowledge in any way the explorer who made the link.



The *CurrentState* of *S0* is now *'Closed'* and the socket soon disappears.

Placing a callback function with result 0 on the *TCPError* event, or setting the *Event* action code to ¯1, will stop these popup error boxes from appearing.

15.1.1.2 In a clear WS, create a socket on *LocalPort* 80, and set all *Event*s to ▿*show*▿, which is just

```
⎕FX'show Msg' 'Msg'
'S0'⎕WC'TCPSocket' ('LocalPort' 80)('Event' 'All' 'show')
```

The *Create* event callback (*show*) should respond immediately with the message

```
S0 Create 1
```

Now repeat the above experiment by entering **http://127.0.0.1** into your Internet Explorer.

Inside IE the explorer creates a socket using the IP address specified in its address bar as the remote address. It then gets a connect event and immediately sends an HTTP request to our server.

The subsequent events from the APL host server's point of view occur as follows. First the *TCPAccept* event triggers and reports the socket number (handle) of the original socket we created (272).

```
S0 TCPAccept 272
```

The original `S0` has actually been replaced by a new socket `S0` that has taken on the responsibility of connecting to the client browser that requested the connection. The original listening socket, now nameless, continues to exist, at least until any callback on the `TCPAccept` event has finished processing.

The `TCPAccept` event is followed by a `TCPReady` event and then by a `TCPRecv` event. This last event, the `TCPRecv` event, triggers on receipt of some data from IE. The event message contains the received data concatenated with the IP address and port number of the network source of the data (the address and port of the client – our IE browser socket). The actual data received from IE, which prints out in the session as a consequence of the `∇show∇` callback, looks something like:

```
GET / HTTP/1.1
Accept:  */*
Accept-Language: en-gb
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: 127.0.0.1
Connection: Keep-Alive
```

This message is composed in HTTP protocol and says something like "Hello Host, please send me your home page. I understand all sorts of stuff if you talk *HTTP/1.1* language. I'll keep the line open until I hear from you. Over." (Over is CRLF CRLF.)

Details of the precise meaning of the message may be extracted from the official World-Wide Web specification of the HTTP/1.1 protocol - in http://www.w3.org/Protocols/rfc2616/rfc2616.txt.

The first and most important line is made up of a *request type*, a *request document URI* (Uniform Resource Identifier) and a *protocol*, all separated by a single space:
`GET / HTTP/1.1`
The network location of the URI is also transmitted in a Host header field.

The above is a GET request, for the domain root index file (/ followed by a space or just a space) in protocol HTTP/1.1 . RFC2616 says,

> "The GET method means retrieve whatever information (in the form of an entity) is identified by the Request-URI. If the Request-URI refers to a data-producing process, it is the produced data which shall be returned as the entity in the response and not the source text of the process, unless that text happens to be the output of the process."

All we need to understand so far is that GET may be followed (after a space) by the name of a file on the server whose hypertext content is to be returned (inside a valid HTTP message) to the client, and if no file is specified, but only a /, then this implies that the file to be returned is the site home page.

Setting the `TargetState` of `S0` to `'Closed'` will close the connection gracefully and erase the socket.

§§ 15.1.2 Cloning a listening Socket on `TCPAccept`

The main thing an APL server has to do in the `TCPAccept` event is to clone the listening socket so that someone else can establish a connection (even while the other is still connected). This is done by creating a socket in the `TCPAccept` callback with the same `SocketNumber` as the original listening socket.

15.1.2.1 In a clear WS, create a socket on `LocalPort 80` (or `LocalPortName` http), with a callback on the `TCPAccept` event and a global variable `COUNT`, set to zero:

```
      'S0'⎕WC'TCPSocket'('LocalPort' 80)('Event' 'TCPAccept' 'acc')
```

in which ∇acc∇ is

```
      ∇ acc Msg;w      ⍝ WHEN ACCEPT CONNECTION TO CLIENT
[1]     COUNT+←1                        ⍝ increment COUNT on each accept.
[2]     w←,⊂'Type' 'TCPSocket'          ⍝ create a socket,
[3]     w,←⊂'SocketNumber'(3⊃Msg)       ⍝ with the handle of the original,
[4]     w,←⊂'Event'((⊃Msg)⎕WG'Event')⍝ with all the events as before.
[5]     ('S',⍕COUNT)⎕WC w               ⍝ create with next name.
      ∇
```

After connecting from IE, check `(S0 S1).CurrentState ↳ 'Connected' 'Listening'`.
Set the `TargetState` of `S0` to `'Closed'` and repeat the request from IE. After connecting from IE, check that `(S1 S2).CurrentState ↳ 'Connected' 'Listening'`.
Note the hopeful message in the IE status bar saying "Opening page http://127.0.0.1/…".

§§ 15.1.3 Sending an HTML File on *TCPRecv*

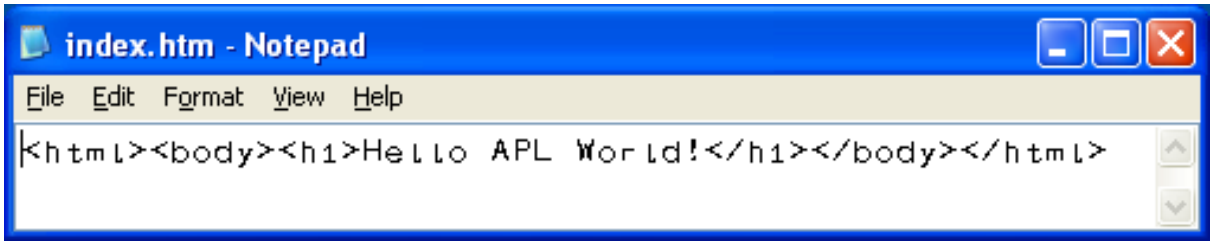
The simple general pattern of events in a working server is this:

1. Wait for a request to connect from a web browser
2. Connect and clone listener in preparation for another request
3. Send the information requested by the browser, *eg* the home page
4. Close the connected socket
5. REPEAT AS REQUIRED

A `TCPRecv` callback implements items 3 and 4. In the simplest scenario, all the receive function has to do is read the web site home page, send it back to the server and close the connection.

```
      ∇ rec Msg;t;d      ⍝ WHEN RECEIVE MESSAGE FROM CLIENT
[1]     t←'C:\homepage\index.htm'⎕NTIE 0
[2]     d←⎕NREAD t 82,2↑⎕NSIZE t          ⍝ read
[3]     ⎕NUNTIE t
[4]     2 ⎕NQ(⊃Msg)'TCPSend'd             ⍝ send
[5]     (⊃Msg)⎕WS'TargetState' 'Closed'   ⍝ close
      ∇
```

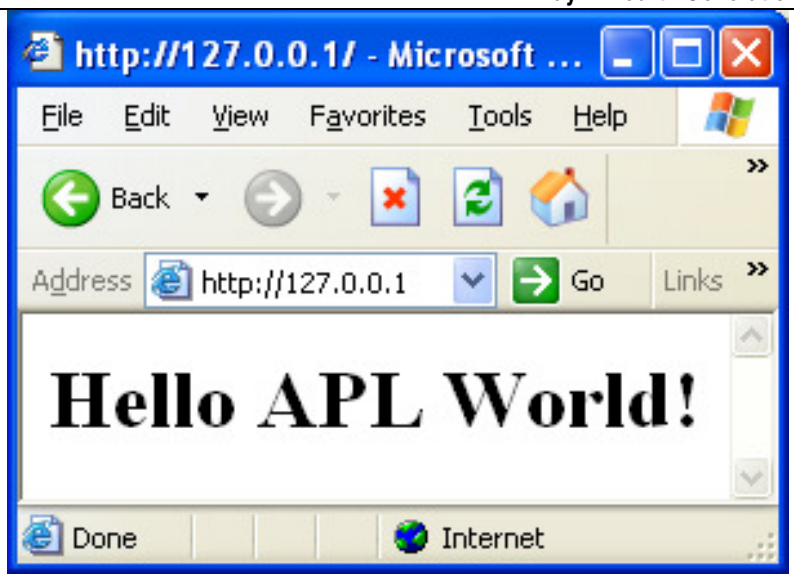
A trivial home page in the file C:\homepage\index.htm, written in HTML so that IE can present it nicely, could be just one simple line:



15.1.3.1 Erase all sockets and restart IE. Create a new socket with `TCPAccept` and `TCPRecv` callbacks:

```
      COUNT←0
      'S0'⎕WC'TCPSocket'('LocalPort' 80)⍝('LocalAddrName' 'LocalHost')
      'S0'⎕WS('Event'('TCPAccept' 'acc')('TCPRecv' 'rec'))
```

With line `[1]` of ∇rec∇ pointing to a suitable root file, such as index.htm above, use IE to navigate to your web site. IE should then display the intended page.



Note that to run/trace this repeatedly it might be necessary to close IE between connections as IE remembers (caches) and doesn't deem it necessary to request a static page again before displaying it.

§ 15.2 Making a realistic Server

Of course a real web page would be expected to have more on it than just 'Hello…'. Not only that, a real web designer would aspire to having more than one page on his web site. This and much more is possible using hypertext HTML. See a beginner's guide at http://www.put.com/HTMLPrimer.html.

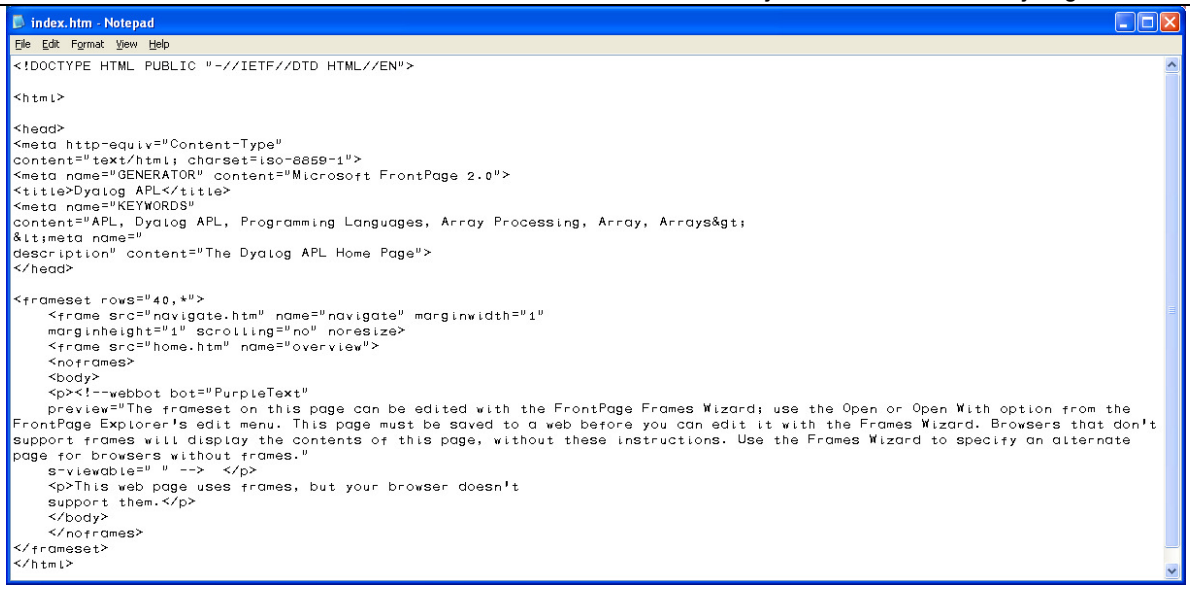
The HTML in the main web site file (traditionally called index.htm - the one that is loaded if nothing special is requested), can tell a client browser that other HTML files and pictures to be found on the web site host are required in order to construct the complete home page. The browser opens another connection to the web site and requests that file next. If that file points to others which are required to complete the page being displayed then the browser will open another connection and ask for that file, and so on .. until the entire page has been built on the screen.

Also embedded in HTML might be links to other files on the web site that define (through a number of sub-files) other complete pages. The links can be defined such that one click from a user can send the browser off to request all the required files and then construct the page even while more pieces are being delivered from the web site host server. The links can equally refer to pages (or, by default, the main page) of any other web site on the network by including an IP address of the site in question in the HTML.

Thus, for example, the server in the distributed workspace ..\server\tcpip\www.dws refers to a homepage to be built from ..\samples\tcpip\homepage\index.htm. This homepage file, shown below, includes links to other files such as that in:

```
<frame src="home.htm" name="overview">
```

The HTML in home.htm includes calls to other files that call yet more files - all required in order to build the home page.



Calls to pages other than the site home page are to be found in navigate.htm. When the area in the specified rectangle is clicked, the browser will take the user to the URL (*Uniform Resource Locator*) specified by the HREF (*Hypertext Reference*) value.

```
<AREA SHAPE="RECT" COORDS="338, 6, 392, 27" HREF="support.htm">
```

The requested resource in HREF may be anywhere on the network:

```
<AREA SHAPE="RECT" COORDS="460, 6, 537, 27" HREF="http://195.212.12.1:8081/frserve.htm">
```

Developing web sites in raw HTML can be an onerous task. However, numerous WYSIWYG applications have been developed to write HTML for you. This makes it much easier to create complex web sites with many pages, each containing text, graphics and other controls. Microsoft FrontPage, Macromedia Dreamweaver or even Microsoft Word 9.0, are some of the applications that you may use to design and alter WYSIWIG pages of your web site.

§§ 15.2.1 Threading multiple Connections

A web server is usually intended to be able to host a service to a number of clients simultaneously. So far we have enabled this by ensuring that a new listening socket is always present. However, a new client cannot connect to a new socket or expect a reply until processing for previous clients has finished. Creating each new listening socket in a separate APL thread can dissipate this potential queue and thus make the service more responsive to multiple simultaneous connections.

15.2.1.1 Change ∇*acc*∇ as below so that new listeners are cloned in a separate thread. Use the unique thread ID to name the socket and process socket events through ⎕*DQ*.

```
      ∇ acc Msg;w      ⍝ WHEN ACCEPT CONNECTION TO CLIENT
[1]     :If 9=⎕NC⊃Msg  ⍝ if socket exists,
[2]         clone&Msg  ⍝ clone in a separate thread.
[3]     :End
      ∇
```

where

```
      ∇ clone Msg;w      ⍝ CLONE THE OLD LISTENING SOCKET
[1]     w←,⊂'Type' 'TCPSocket'        ⍝ create socket,
[2]     w,←⊂'SocketNumber'(3⊃Msg)     ⍝ with the handle of the original,
[3]     w,←⊂'Event'((⊃Msg)⎕WG'Event')⍝ and with all previous events.
[4]     ⎕DQ('S',⍕⎕TID)⎕WC w           ⍝ create with unique name.
[5]     ⍝ a line after a line with a ⎕DQ is helpful for tracing
      ∇
```

Check that your web server still works with this enhancement in place.

A number of other essential features and useful suggestions are to be found in the supplied SERVER workspace. For example, you are advised to include a callback, even if empty, on the `TCPClose` event to ensure that a socket does not close prematurely. Also the `TCPError` event callback returns `0` to stop popup error boxes appearing during the transaction process. The callback simply erases any socket causing an error. The `TCPError` event message contains the error details and therefore more error handling could be done in this callback. To avoid potential problems in development, all sockets should be expunged before initiating the service.

In preparation for the information expected in the `TCPRecv` callback, a variable called `BUFFER` is initialised to `⎕AV[4 3]` (CRLF) in the connected `TCPSocket` namespace. CRLF is the recognised HTTP command string separator and `BUFFER` is going to be filled with the HTTP commands from the client browser connected to this socket.

§§ 15.2.2 Communicating through HTTP

If large amounts of data are being sent between stream sockets in a single transaction then the data is automatically broken into manageable packets and sent one at a time. The HTTP identifier for the end of a complete set of packets is CRLF CRLF – two empty lines. The possibility of receiving incomplete message packets is incorporated in a robust server by adding lines to the `TCPRecv` callback which add packets to a buffer and look for the end marker CRLF CRLF.

```
(⍎⊃Msg).BUFFER,←3⊃Msg
 :If ⎕AV[4 3 4 3]≠¯4↑(⍎⊃Msg).BUFFER ⍝ if we have not got everything,
     :Return                        ⍝ stop and wait for more.
 :EndIf
```

In the simple examples above, a small amount of data was sent so no `:Return` was necessary, but this is not guaranteed in a stream socket, although the order of receipt is always the same as the order of transmission.

When a browser first establishes a connection with our server, the HTTP data shown in section §§15.1.1 is received, terminated with `⎕AV[4 3 4 3]`. Our APL server could simply send the requested file or it could proceed with a *challenge* to the browser saying that our site has restricted access and requires a user ID and password to be supplied before entry will be granted. This our server may do by sending the client an HTTP request for authorization looking something like:

```
HTTP/1.1 401 Authorization Required
Date: Fri, 24 Mar 2006 13:12:55 GMT
Server: Dyalog APL
WWW-Authenticate: Basic realm="User Area 100"
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Transfer-Encoding: chunked
Content-Type: text/html; charset=iso-8859-1
1c0
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML>
  <HEAD>
    <TITLE>401 Authorization Required</TITLE>
  </HEAD>
  <BODY>
    <H1>Authorization Required</H1>
    <P>This server could not verify that you are authorized to access the document requested.  Either you supplied the wrong credentials (eg, bad password),
       or your browser doesn't understand how to supply the credentials required.</P>
    <HR>
    <ADDRESS>Dyalog APL 10.0 Server</ADDRESS>
  </BODY>
</HTML>
0
```

This tells the browser to prompt the user for an ID and password. The arbitrary words "User Area 100" chosen by the server site programmer are displayed on the password dialog box to indicate the *realm* to which access is being offered. An HTML error message is embedded (with a 'parity check' 1c0). This is to be displayed by the client in the event that the supplied credentials are insufficient.

Note that in order to specify which version of the HTML standard they conform to, all HTML documents should start with a *document type declaration* (informally, a "DOCTYPE"), which makes reference to a *document type definition* (DTD). Using the line
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML//EN">
is adequate for our purpose, and indeed it may be omitted entirely.

When an ID and password have been entered, the two strings, separated by a colon, are encoded and included in the form of an Authorization field in all further messages sent by the browser to the server. On this subject, RFC2616 says,

> "A user agent that wishes to authenticate itself with a server--usually, but not necessarily, after receiving a 401 response--does so by including an Authorization request-header field with the request. The Authorization field value consists of credentials containing the authentication information of the user agent for the realm of the resource being requested."

The server has an opportunity to validate credentials at the head of each GET request before processing the specific contents of the request URI and sending a response to the client browser. The credentials are sent in a base-64 encoded string in an Authorization field such as
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
The distributed PATCH workspace contains functions for encoding and decoding credentials. For example (noting the changed translation vector in ⎕*NXLATE*):

```
      enco'Aladdin:open sesame' ↳ 'QWxhZGRpbjpvcGVuIHNlc2FtZQ=='
```

and

```
      deco 'QWxhZGRpbjpvcGVuIHNlc2FtZQ==' ↳ 'Aladdin:open sesame'
```

where

```
    ∇ strg←deco code;raw;alph
[1]   alph←⎕A,(26↑17↓⎕AV),⎕D,'+/'
[2]   raw←⊃{(⎕DR ⍵)11 ⎕DR ⍵},⍉(6⍴2)⊤(alph⍳code~'=')-⎕IO
[3]   strg←82 ⎕DR(-8|⍴raw)↓raw
    ∇
```

This gives a *Basic* level of security. There is also a higher *Digest* level of security as described in HTTP Authentication: Basic and Digest Access Authentication in http://www.ietf.org/rfc/rfc2617.txt.

§§ 15.2.3 Running APL Functions on a Server

So a conversation between IE and a server could proceed as

IE Client:	GET / HTTP/1.1…
APL Server:	HTTP/1.1 200 OK…
IE Client:	GET /images/tennis.gif HTTP/1.1…
IE Client:	GET /banners/scoresheet.gif HTTP/1.1…
APL Server:	HTTP/1.1 200 OK…
APL Server:	HTTP/1.1 200 OK…

But negotiations between clients and servers are not restricted to requesting and supplying static pages. Often significant background processing is desired. The GET request type can fulfil both roles.

If the GET request URI does **not** contain a question mark then the URI is assumed to be an HTML file name whose content is to be returned. If, however, the URI **does** contain a question mark then a browser recognises this as a *query URI* that can perform operations with significant side effects.

Such a request URI may be embedded in an HTML page in a hyperlink such as
```
<a href="driller.RUN? title="drill down">Sage Driller<\a>
```
This creates an element that becomes a hyperlink (with an optional 'hover box' title). In the event that a user clicks on "Sage Driller", the browser will send
```
GET driller.RUN? ...
```
On receipt, the server in ..\aplserve\server.dws interprets this as a request to run a function `∇RUN∇` in a namespace called `driller`. The function must be defined dyadically and is automatically given a left argument of the name of the socket involved. The right argument consists of parameters following the ?, and separated by & if there is more than one parameter in the argument. A number of parameters may be needed for a particular function, or none at all as in the case of `driller.RUN`. The other requirement of the APL function is that any result is in the form of an HTML string. The browser will display this HTML automatically on receipt.

Note that this syntax is not APL-specific. For example
```
<a id=2a class=q href="http://groups.google.co.uk/grphp?hl=en&tab=wg&ie=UTF-8">Groups</a>
```
is to be found on the Google front page.

By adding a new namespace with a top level function adhering to the above syntax, new applications may be added to the server workspace in a very straightforward and elegant fashion. See, for example, functions `rain.Climate`, `rain.Fourier` or `CODEVIEW.FUNCTION`.

The POST request type may be used as an alternative way of initiating a function call in the server.

> "The POST method is used to request that the origin server accept the entity enclosed in the request as a new subordinate of the resource identified by the Request-URI in the Request-Line."

The POST request type is suitable for calling a function from an HTML form. For example, the loan.htm file contains the line
```
<form action="loan.RUN" method="POST" ...
```
In this case `loan.RUN` is recognised as a function call when a POST request is received from the client as a result of the user clicking on the form.

The ..\aplserve\server.dws workspace supports both GET and POST requests. More examples of client-server negotiations may be found in ..\ws\FTP.dws and ..\ws\PATCH.dws.

A very useful tool for intercepting TCP interactions between a computer and the outside world may be downloaded free from http://www.westbrooksoftware.com/tsdownload.shtml.

§ 15.3 Internet Practicalities

§§ 15.3.1 Domain Name Servers

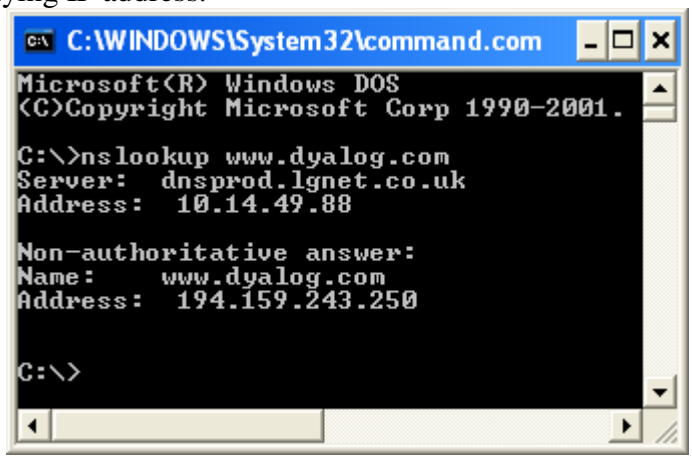
Web sites are better known by their names than by their IP addresses. For example, www.dyalog.com is more memorable than 194.159.243.250. But an IP address in `LocalAddr` or `RemoteAddr` is essentially equivalent to a domain name in `LocalAddrName` or `RemoteAddrName`.

Either the address or the address name may be used in the specification of a `TCPSocket` property. A name is converted into the equivalent IP address by a *Domain Name Server* (DNS) which is always

accessible from an ISP via the Winsock API. A *TCPSocket* object has a *TCPGotAddr* event. This event is triggered when an address name is resolved into an IP address. For example

```
      'S0'⎕WC'TCPSocket'('RemoteAddrName' 'www.dyalog.com')↵
            ('RemotePort' 80)('Event' 'TCPGotAddr' 'show')¶
 S0  TCPGotAddr
      S0.RemoteAddr
194.159.243.250
```

In Windows XP the DNS may be invoked by the nslookup utility which can be applied to any domain name to extract the underlying IP address.



Port numbers are also often referred to by the service names.

15.3.1.1 Create a socket with *LocalPortName* http and show the event message from *TCPGotPort*. Check the *LocalPort* property when the port name has been resolved.

§§ 15.3.2 Firewalls and proxy Servers

If you are inside the walls of a large company, the chances are that you have to go through a firewall every time you communicate with the outside world via your personal computer. A firewall is a proxy server which filters and controls the traffic between the company Intranet, a trusted zone, and the outside Internet, which is not trustworthy and is teeming with parasites. A proxy server is both a client and a server. It acts as a server for all requests to the outside world from the Intranet, and as a client to all Internet servers. It therefore offers a protective barrier between order and anarchy.

In this case, a proxy server responds to a request, not with HTTP/1.1 401 Authorization Required.., but with HTTP/1.1 407 Proxy Authentication Required... With a simple request for an Internet page

```
GET http://www.google.co.uk/ HTTP/1.0
Accept: */*
Accept-Language: en-gb
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: www.google.co.uk
Proxy-Connection: Keep-Alive
```

the response of a proxy server may be something like:

```
HTTP/1.1 407 Proxy Authentication Required ( The ISA Server requires authorization to fulfill the request. Access to the Web Proxy service is denied. )
Via:1.1 OURPRXY
Proxy-Authenticate: Basic realm="Privileged User Access Area"
Pragma: no-cache
Cache-Control: no-cache
Content-Type: text/html
Content-Length: 2370
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML dir=ltr><HEAD><TITLE>The page cannot be displayed</TITLE>
...
```

to which a suitable reply, containing an encoded Proxy-Authorization field, could be

```
GET http://www.google.co.uk/ HTTP/1.0
Accept: */*
Accept-Language: en-gb
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; .NET CLR 1.1.4322)
Host: www.google.co.uk
Proxy-Connection: Keep-Alive
Proxy-Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

and the proxy server's response, having checked the credentials might be

```
HTTP/1.1 200 OK
Via: 1.0 OURPRXY
Date: Thu, 30 Mar 2006 07:59:35 GMT
Content-Type: text/html
Cache-Control: private
Server: XYZ/2.2
…
```

The command `Proxy-Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==` must be included in all further requests to the proxy server from the client.

§§ 15.3.3 An ISP running Dyalog.DLL

In order to have a web server that runs Dyalog APL code and is accessible on the Internet it is clearly necessary to have Dyalog APL running on a computer that is connected to the Internet.

Your Internet Service Provider may be willing, at a price, to run Dyalog APL but this gives you less control than you would wish, at least during the development phase. Rather than ask your ISP to host your site, why not host it yourself from an old computer in the shed outside?

15.3.3.1 Please ask for the next module on **Web Clients**.