Advanced Network Programming with API19/W for Dyalog APL

—*by Dr. Andrei Kondrashev*
Chicago, Illinois

This article is a continuation of the paper published in the previous issue of APL Quote Quad, entitled "Introduction to Network Programming with APL."

•

ICMP protocol: The PING program

MOST TCP/IP IMPLEMENTATIONS provide a "ping" program as a diagnostic tool. This program sends an echo request message to a specified host. The purpose of such a message is to test whether the host is reachable or not. The echo request message uses the *Internet Control Message Protocol* (ICMP). Despite the fact that the ICMP protocol is on the transport level of the TCP/IP stack (see Figure 1 in the previous article), it is not considered to be a transport protocol. ICMP allows you to do many interesting things on a network. Usually this protocol is not available to application programmers. However, if your TCP/IP implementation supports so-called *raw* sockets, you can emulate ICMP and, therefore, write your own ping program. (Note that the Microsoft TCP/IP implementation that is commonly used with Microsoft Windows does not support raw sockets. Example shown below was run using *Super-TCP* by Frontier Technologies Corp.)

Raw sockets provide access to the transport layer level (see Figure 1 in the previous article). When we were using TCP, we didn't care about how the TCP/IP stack packs and unpacks our data to/from TCP messages. It was done automatically when we used the SEND and RECV commands. With raw sockets, before you can use the SEND command, you have to create a header that corresponds to the transport protocol you want to use. The data you send go directly to the network protocol level, without processing at the transport level. When you receive data using the RECV command, they come from the network protocol level. As a result, they contain a transport protocol header in front. You have to process this header and, if necessary, reply with an appropriate message. The use of raw sockets is not recommended.

The format of the ICMP message for an echo request is shown in Figure 1.

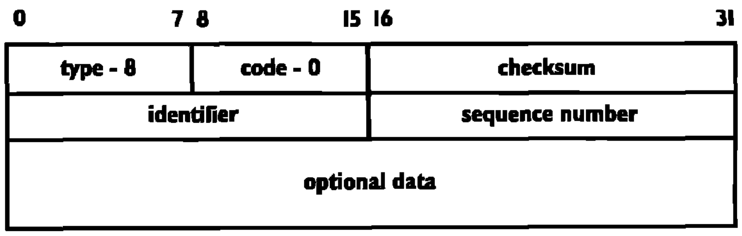


Figure 1: ICMP echo request message

In the simplest case, we need to send only 8 bytes to the IP level to generate an echo request message. Here is a simple program that does the job. Please refer to the previous article for definitions of *SHARE*, *SAY*, and *END* functions.

```
    ∇ PING ADDR;S;A
[1]   ⍝ PINGS AN INTERNET ADDRES FROM "ADDR"
[2]   ⍝ RAW SOCKETS SUPPORT IS REQUIRED!
[3]    SHARE
[4]    S←SAY 'SOCKET' 'RAW' 1
[5]    SAY 'SENDTO' S 0 'N' 2 0 ADDR (8 0 247 255 0 0 0 0)
[6]    →(1⊃SAY 'SELECT' S '' '' 30)/OK
[7]    ADDR,' is not responding.' ⋄ →EX
[8]   OK:A←SAY 'RECVFROM' S 0 'N'
[9]    'Received response from ',⍕3⊃2⊃A
[10]  EX:SAY 'CLOSE' S
[11]   END
    ∇
```

The SOCKET command uses an additional argument that we didn't use in our previous programs. The third parameter (that normally can be omitted) specifies the transport protocol that will be used in connection with the new socket. For stream and datagram sockets the protocols are chosen by default—TCP and UDP. Because we are going to use a raw socket, the system needs to know what protocol number to put into the Internet datagram header that will be formed on the network protocol level of the TCP/IP stack. ICMP has protocol number 1. TCP and UDP have protocol numbers 6 and 17 respectively.

The *PING* function also uses two new commands for sending and receiving data: SENDTO and RECVFROM. SENDTO accepts a recipient's address as a parameter. RECVFROM provides the address of the sender in its result.

PING sends 8 bytes (the minimal ICMP echo request message) to an address in question (line 5) and waits for its response (line 6). If the address does not respond in 30 seconds, it concludes that it is unreachable. This is an example:

```
      PING '146.241.92.78'
Received response from 146.241.92.78
      PING '146.241.92.99'
146.241.92.99 is not responding
```

You could send a sequence of pings storing the sequence number in the appropriate field (see Figure 1). That will allow you to check whether all your messages returned back and whether they are still in the sequence they were sent.

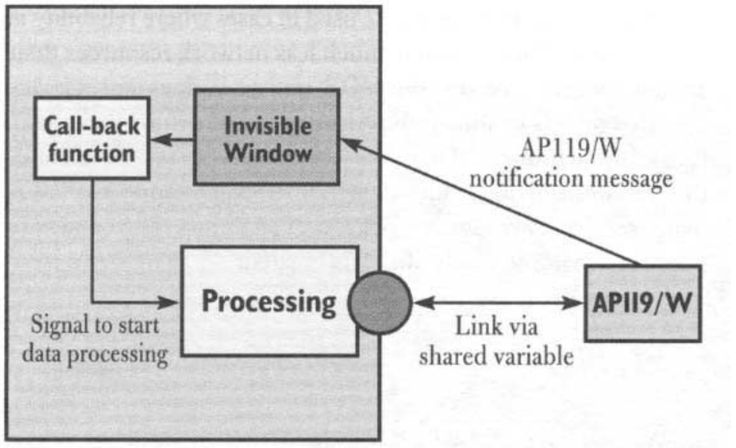


Figure 2: AP119/W notification mechanism

Event-driven programming with AP119/W

The following commands will block your system, if an appropriate network event does not have place:

AP119/W Command	Network Event	Event Name
ACCEPT	Incoming connection request	ACCEPT
CONNECT	Connection established	CONNECT
RECV	Data available for reading	READ
RECV with OOB flag	Out-of-band data available for reading	OOB
RECVFROM	Data available for reading	READ
SEND	TCP/IP kernel ready to transfer data	WRITE
SENDTO	TCP/IP kernel ready to transfer data	WRITE
GETHOSTNAME	DNS request completed	DNS
GETHOSTID	DNS request completed	DNS

You can use the SELECT command to examine the status of a socket and see whether an AP119 call will block or not. Note, that the CONNECT and DNS events cannot be examined before the corresponding command is executed. For many applications the use of SELECT (or blocking operations) is good enough. However, if you need to write a TCP/IP server that has to effectively process requests from many clients simultaneously, or you want to write something as a Telnet program, the use of SELECT becomes unsatisfactory. To have good response time, such a program should interrupt itself very often to check the status of active sockets. In most cases it will be just a waste of time. A better approach is to make the TCP/IP kernel interrupt the application when an event of interest occurs on the network.

The NOTIFY command is used to specify a Dyalog APL window (form) that should receive notification messages when selected events occur on a particular socket. Before we can use this command we have to create an invisible top-level window. The call-back function that should be called when a network event occurs, must be attached to Dyalog APL event 5 (MouseDblClick):

```
'AP119F'⎕WC'FORM'('COORD' 'PIXEL') ('CAPTION' '119')('EVENT'5'TCP_CB')('VISIBLE' 0)
```

We do not care about the size and position of this window; it is invisible. However, you must set up the coordinate system in *PIXEL* mode. The generic form of a call-back function is shown below (it has the name *TCP_CB*, but you may use any name you like):

```
    ∇ TCP_CB MSG;EVENT;ERROR;S
[1]   ⍝ GENERIC CALLBACK FUNCTION FOR A AP119/W NOTIFICATION MESSAGE
[2]    S EVENT←0 256⊤3⊃MSG ⋄ ERROR←4⊃MSG
[3]    →(EVENT=1 2 4 8 16 32 128)/READ,WRITE,OOB,ACCEPT,CONNECT,CLOSE,DNS
[4]    →0
[5]   READ:'READ EVENT ON SOCKET ',(⍕S),(×ERROR)/'. ERROR CODE: ',⍕ERROR
[6]    →0
[7]   WRITE:'WRITE EVENT ON SOCKET ',(⍕S),(×ERROR)/'. ERROR CODE: ',⍕ERROR
[8]    →0
[9]   OOB:'OOB EVENT ON SOCKET ',(⍕S),(×ERROR)/'. ERROR CODE: ',⍕ERROR
[10]   →0
[11]  ACCEPT:'ACCEPT EVENT ON SOCKET ',(⍕S),(×ERROR)/'. ERROR CODE: ',⍕ERROR
[12]   →0
[13]  CONNECT:'CONNECT EVENT ON SOCKET ',(⍕S),(×ERROR)/'. ERROR CODE: ',⍕ERROR
[14]   →0
[15]  CLOSE:'CLOSE EVENT ON SOCKET ',(⍕S),(×ERROR)/'. ERROR CODE: ',⍕ERROR
[16]   →0
[17]  DNS:'DNS REQUEST COMPLETED. ',(×ERROR)/'. ERROR CODE: ',⍕ERROR
    ∇
```

The first argument of the NOTIFY command specifies the handle or the caption of the window that should receive notification messages. If you use window's caption (version 6.3 of Dyalog APL/W didn't have the HANDLE property), you have to use a unique caption for this window (something very unusual). The second argument is the socket number, and the third argument is an event mask. In our case, we want to receive notifications about all network events that occur on socket *S*. Figure 2 explains how the AP119/W notification mechanism works.

When the NOTIFY command is executed, AP119/W finds the window that has the specified handle. The events of interest are defined by the third argument of the command. Number 1 corresponds to the READ event, number 2 to the WRITE event, and so on (see Chapter 6 for more details). When an event of interest occurs, AP119/W posts the MouseDblClick (event 5) message to the specified window. Parameter 3 of this message shows the network event number and the socket number pertaining to the network event. Line 2 of the *TCP_CB* function above decodes this information. Parameter 4 of the message is the error number. Error number 0 indicate successful operation. Each network event generates a separate message. Line 3 of the *TCP_CB* function re-directs execution according to the network event (in our case we only display an appropriate message).

The following is a summary of how to use event-driven programming with AP119/W:

1. Create a socket with required characteristics.
2. Create a hidden window and attach a call-back function to event 5.
3. Issue the NOTIFY command to specify a window that will receive messages for events in interest for the socket.
4. Establish a TCP connection (for stream sockets only).
5. Do background processing (processing messages from other application windows).
6. Respond with AP119/W commands on notification of network messages.

There is an important difference between the CONNECT and DNS events and other networks events. CONNECT and DNS events are posted to the application *after* the execution of the CONNECT, GETHOSTNAME, or GETHOSTID commands. CONNECT event informs that the connection is established or failed. DNS event informs that asynchronous DNS requested is completed. If the CONNECT event is enabled (the corresponding socket is in non-blocking mode) the CONNECT command will return immediately with the result 0. The application can wait until the CONNECT event is posted, or it can cancel the connection attempt using the CANCEL command (this is how Web browsers work). If the application uses asynchronous version of GETHOSTID or GETHOSTNAME commands the AP119/W will return immediately with the result 0. The application can wait until the DNS event is posted, or it can cancel the connection attempt using the CANCEL command. When DNS request is posted, the application can use FETCH command to get the results of the DNS request. All other network events are posted to the application *before* the corresponding AP119/W command is used. They inform the application that TCP/IP stack is ready for the corresponding operation and it can be completed without blocking.

You might choose to use a mixed strategy that uses NOTIFY and SELECT commands together. However, it is not recommended for the same socket, because in this case you may receive notification messages that incorrectly reflects the real situation on the socket.

Example of event-driven application: Datagram sockets

Datagram sockets are much easier to understand than stream sockets. They are very easy to use, but in most cases they require more sophisticated application protocols in comparison to stream sockets. Datagram sockets use UDP transport and handle the data flow in the form of user datagrams. In fact, everything that we said earlier about IP datagrams can be applied to user datagrams.

Datagram sockets should be used in cases where reliability is not essential. They consume much less network resources than stream sockets, because the UDP transport does not use acknowledgments or time-outs. An application protocol can take care of these features, if necessary. Stream sockets provide one-to-one connectivity, while datagram sockets support one-to-many service. A datagram socket acts as a post office and it is ready to use immediately after it is created:

```
SHARE
D←SAY 'SOCKET' 'DGRAM'
SAY 'SENDTO' D 0 'A' 2 1000 '146.249.92.76' 'MY MESSAGE'
```

We just have created a datagram socket *D*, and have sent a message to the addressee with port number 1000 on the computer with IP address 146.240.92.76. Because the socket *D* is not connected to any particular recipient on the network, we had to specify the delivery address explicitly using the SENDTO command. Because UDP is a connectionless protocol, the CONNECT command is not needed. If you use the CONNECT command with a datagram socket, it simply sets a default address for the SEND command.

There is no way to know whether our message was received, unless our partner confirms this somehow. If there was an opened datagram socket with the specified parameters on the network, and if it was expecting to receive something from us, and if it sent something to us as a confirmation (too many "if"s!), we can retrieve its message:

```
      1⊃SAY 'SELECT' D '' '' 5
1
      ⎕←SAY 'RECVFROM' D 0 'A'
I RECEIVED YOUR MESSAGE  2 1000 146.240.92.76
```

First we check whether the socket is ready to read, in order to avoid blocking. Again, because we don't know where the message comes from, we use the RECVFROM command to retrieve the message. This command returns not only the message itself, as RECV does, but the sender's parameters also (in the second item of a two-item result). You can use the RECV command instead if you don't care about the source of the message.

The following is an example of a simple event-driven network application that uses the Dyalog APL interface to Windows. The *ECHO* program creates an application window with two buttons (see Figure 3). We will "ping" a standard echo server using a datagram socket. (If your TCP/IP stack supports raw sockets, you can easily modify this program into another version of the *PING* program.) There is a timer object in the system. It generates an event every second. When the call-back function *ECHΔH* is fired, as the result of the timer event, it sends a datagram to the echo server. We will trap only the READ event from the created socket. When this event occurs, the program receives a message. The "Stop" button stops the timer and disables the datagram sending. The program displays the current number of sends and receives. If the standard echo service is not available to you, you can write your own echo server in APL.

The main program *ECHO* does only initial setup, leaving the processing on the call-back function.

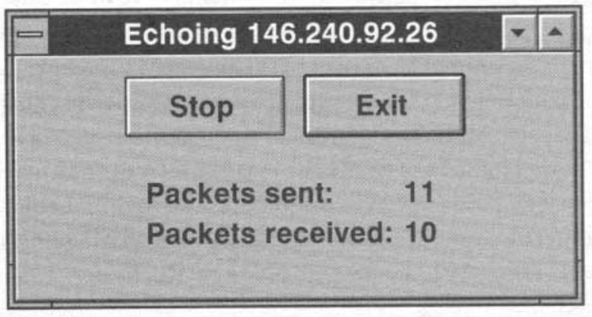
```
    ∇ ECHO ADDR;S;PS;PR
      ⍝ STANDARD ECHO SERVICE (7). CLIENT PROGRAM
       →SHARE↓0
       S←SAY 'SOCKET' 'DGRAM'
       PS←PR←0
      'TF'⎕WC'FORM'('CAPTION'('Echoing ',ADDR))('COORD' 'PIXEL')
      'TF'⎕WS'SIZE' 120 280
      'TF.ST'⎕WC'BUTTON'('CAPTION' 'Stop')('SIZE' 30 80)('POSN' 10 55)
      'TF.ST'⎕WS'EVENT' 30 'ECH∆H' 0
      'TF.EX'⎕WC'BUTTON'('CAPTION' 'Exit')('SIZE' 30 80)('POSN' 10 145)
      'TF.EX'⎕WS('EVENT' 30 'ECH∆H' 1)('DEFAULT' 1)
      'TF.T1'⎕WC'TEXT'('POINTS' 60 65)('TEXT' 'Packets sent:')
      'TF.T2'⎕WC'TEXT'('POINTS' 60 195)('TEXT' '0')
      'TF.T3'⎕WC'TEXT'('POINTS' 80 65)('TEXT' 'Packets received:')
      'TF.T4'⎕WC'TEXT'('POINTS' 80 195)('TEXT' '0')
      'ECHF'⎕WC'FORM'('CAPTION' 'ECHO_CB')('COORD' 'PIXEL')('VISIBLE' 0)
      'ECHF'⎕WS'EVENT' 5 'ECH∆H' 3
      SAY 'NOTIFY'('ECHF'⎕WG'HANDLE')S 1
      'TF.TIMER'⎕WC'TIMER'('INTERVAL' 1000)('EVENT' 140 'ECH∆H' 2)
      ⎕DQ '.'
      SAY 'CLOSE'S
      END
    ∇
```

The callback function that processes all (not only network) events for this application is:

```
    ∇ B ECH∆H MSG
       →(B=1 2 3)/L1,L2,L3
      ⍝ START/STOP BUTTON
       'TF.TIMER'⎕WS'ACTIVE'(B←~'TF.TIMER'⎕WG'ACTIVE')
       'TF.ST'⎕WS'CAPTION'((1+B)⊃'Start' 'Stop')
       →0
      ⍝ EXIT BUTTON
      L1:⎕EX '.'
       →0
      ⍝ TIMER EVENT: SENDING PACKET
      L2:SAY 'SENDTO' S 0 'N' 2 7 ADDR 'A MESSAGE'
       'TF.T2'⎕WS'TEXT'(B←⍕PS←PS+1)
       →0
      ⍝ TCP/IP EVENT: RECEIVING ECHO
      L3:SAY 'RECV' S 0 'N'
       'TF.T4'⎕WS'TEXT'(⍕PR←PR+1)
    ∇
```

Run it:

```
      ECHO '146.240.92.26'
```



```
Echoing 146.240.92.26

Packets sent:      11
Packets received:  10
```

Figure 3

■

Dr. Andrei Kondrashev can be reached at Lingo Allegro USA, Inc., 113 McHenry Rd, Suite 161, Buffalo Grove, IL 60089 USA; or via e-mail at "71303.3224@CompuServe.com".

An interview with Jeffrey Zweiben of Health Economics Group, Inc.

AS INTERVIEWED BY RAY POLIVKA

WE HAVE OFTEN SPOKEN TO AND INTERVIEWED PEOPLE from large corporations. We are all pleased that APL finds a place in such the large corporations; however we must not overlook the importance of APL to the small entrepreneurial company. Jeffrey Zweiben represents just such a small company. Let's see what he has to say.

•

QQ: Jeffrey, thank you for letting me interview you. I would like to ask you who you are and what you do corporately.

Jeffrey: The company name is Health Economics Group, Inc., in Rochester, New York. It is a third-party administrator known in the industry as a "TPA." What that means is that we design and manage employee benefit plans.

QQ: You manage health benefit plans?

Jeffrey: In general, we manage employee benefit plans. That includes medical plans, dental plans, prescription drug plans, cafeteria plans, flexible benefit plans.

QQ: For whom?

Jeffrey: We do it for employers or employee groups. For example, industrial companies, city and county municipalities, and any group that is large enough to self-insure. We do not take financial risk. Our customers take the financial risk, and if their actuarial numbers work out correctly, they can save money—and they usually do, compared to buying an insured plan. If the numbers do *not* work out, we suggest that they remain insured. The difference between insurance and self-insurance is, in the insurance plan you pay a premium to the insurance company and they take all the financial risk and the customer or individual takes no financial risk. With self-insurance, *you* take *all* the financial risk. There are combinations of insurance and self-insurance. For example, your own car insurance with a deductible of say $500 is one. For the first $500, you are self-insuring and above $500 you are insured. So typically, with medical plans, companies buy what is called *stop-loss insurance* to insure against catastrophic risk and they self-insure all the bottom risk, or the *non*-catastrophic risk.