Revista de la

Universidad Politécnica

de Puerto Rico

Publicado semestralmente por la Universidad Politécnica de Puerto Rico para difundir los hallazgos de la investigación científica que en ella se hace.

VOL. 4 **Junio 1994** **NUM. 2**

A utility for Netware: REMOTE

Steven García
Jorge L. Quintero
Candidatos a graduación

Abstract

We used the C language code to develop a utility that allows two or more workstations connected to the same physical network running the network operating system Netware to communicate and perform remote operations. The utility is based on the client-server theory.

Sinopsis

Usamos el código del lenguaje C para desarrollar un sistema que le permite a dos estaciones en la misma red física que corre el sistema operativo Netware comunicarse y hacer operaciones remotas. El sistema se basa en el concepto de cliente-servidor.

Introduction

In 1982, Superset Software (now known as Novell) developed an operating system that would give a new meaning to networking,as it was known at the time, and set a standard that would be imitated by other companies. Today, that network operating system carries the name Netware, and has aproximately between 70 and 85% of the market share. Its popularity is highly based on the fact that it was the first network operating system to offer true file sharing from a storage unit, protection against equipment failure (fault tolerance) and against data loss (transaction tracking). Besides the support that the program offers to network administrators and users, Netware can to be improved and expanded by hardware and software developers who want to create their own applications using tools provided by the Netware software package.

Included in the Netware package is a program that provides the link between the local operating system and the network operating system. This program, IPX. COM, provides the IPX (Internetwork Packet Exchange) protocol. IPX sets the rules and regulations involved in exchanging data and breaks a message into packets as indicated by the protocol and uses low level drivers to route the packet to its destination.

IPX is a datagram type protocol, which means that IPX does not require stations to establish a connection and allows messages to be broadcasted and picked up by any station that is listening. This characteristic opens a new field in network communications where two stations are able to swap files, share peripherals and send screen messages

to each other. This type of communication, where two workstations are able to establish a direct communication, is known as peer-to-peer networking.

The REMOTE utility

Traditional networking is known as server-client communication, where one terminal (the server) services the needs of all the workstations attached to it (clients). This means that a dedicated workstation is needed to overlook all network operations. For example, appendix 1 includes the listing for a sample program called REMOTE. This program will run "as is" with no need for modifications, but it has no frills. We also included some suggestions to give the user an idea on how to make the program more powerful and useful.

REMOTE permits two or more workstations connected to the same physical network to communicate and perform remote operations. In order to achieve these communication and remote operations we need first to designate one station to work as a server and a second station to act as the client. The server station will receive all commands and execute them. To setup the server station. type the following command:

REMOTE *server*

at the DOS command prompt. This command will cause the station to go into an infinite loop that will check for incoming messages. To setup the client station the user must enter the following instruction:

REMOTE *command*

where *command* is any executable file with an. EXE,. COM or .BAT extension or any other DOS command. Once the user hits the <RETURN> key, the command is sent through the network transmission lines and the station acting as a server listens. picks up the message and then executes the command.

How REMOTE works

To make it easier for the reader to understand the code of the program **REMOTE** we have identified some sections of this article with reference numbers within a box that are associated with some portion of the program code, identified in the same way. We suggest that the reader examine the program listing in appendix 1 to understand each section.

1 First the program parses the command line checking for an argument that will either send the station into a constant loop listening for messages on the transmission lines or cause the program to send a command onto the network. Once the station is setup, the actual sending or receiving can take place.

There are three steps involved in the transmission of information from one station to the next:

- Open a socket.
- Fill in the necessary fields in the ECB, IPX and message structures.
- Initiate the desired listen or send command.

2 Sockets are lines that the programmer chooses to open to send and to receive messages. To open a socket we have to assign certain values to the internal registers of the computer and then make a call to the IPX interrupt entry point (Int 7A) so as to cause the protocol to start the open operation. Into the BX register we assign the value of the desired function (in this case OPEN, which corresponds to 00H). The AL register will hold either 00, signaling IPX to maintain the socket open while the program is running and to automatically close it once the program ends or is canceled, or FFH, which will hold the socket open until the CLOSE function of the IPX program is explicitly called (a value of 01H in the BX register).

After the socket is opened, an error code is returned to the AL register. If this code has a non-zero value it signals that some type of failure has occurred during the OPEN function. In some cases we would trap this value to check for non-critical errors, but in most cases an error would indicate that the socket is already in use. It is desirable that the socket we choose for transmission not be in use in order to avoid data loss due to collisions.

The socket number is assigned to the DL register. This number is totally arbitrary but must not lie between 0000H and 0BB9H or be greater than 0x8000. These socket values have been reserved for use by the network operating system. One point to remember is that IPX uses the socket number in reverse order to the way the CPU does. So, socket numbers should be provided with the least significant byte first. An easy convention is to use socket numbers such as 5050H or 3535H. In this way the low byte and the high byte can be interchanged without altering the original number.

The next step is to prepare the structures to be passed on to IPX as part of the transmission packet. The first of these structures is the event control block or ECB. The ECB is not actually transferred along with the packet, but it does contain important information that IPX will use for its handling. At the same time, the programmer obtains information of the state of the current IPX operation. The programmer should compliment the *ESR, socket, imdt_add, frag_count, *frag_address and frag_size fields. These fields are explained as follows:

*ESR — This field holds the address of the event service routine, a sub-routine that is to be executed once IPX completes the indicated service (i.e., send or listen). This field should contain a null value if it is not to be used.

socket — This field stores the number of the line "opened" for listening or sending. The sending and receiving stations must open the

same sockets for effective transmission of the message.

imdt_add The immediate address corresponds to the address of the closest bridge the message must cross in order to arrive at its destination. If there is no bridge, the field will carry the address of the destination station. This information is obtained through the IPX GET_TARGET function. IPX uses this field to redirect the message to its destination.

frag_count This field contains the integer number of fragments that compose the entire message packet.

*frag_address and frag_size

These fields hold the address in memory where the fragment is located and its respective size. The size is calculated as the total number of bytes occupied by the IPX header and the message structure. The following line makes this task simple:

sizeof(struct ipx) + sizeof(struct message).

These fields can be held in an array to include a number of packets.

The following fields are solely manipulated by IPX in the following way:

*link This field is used internally by IPX for control.

*in_use This field contains a flag to indicate the current state of the last command issued to IPX. IPX maintains the value of this field different to zero until the event has been completed, after which it will change it to zero.

cmpt_code This field will have a value representing the result of the current operation. Use it in conjunction with the in_use flag and the ESR subroutine in order to create error trapping routines.

IPX_wrk and driver_wrk

These two fields are work spaces reserved by IPX for use as temporary registers.

The next two structures form the actual message fragment. The first structure is the IPX header. It contains information about the fragment (e.g., the source, the destination and the length of contents).

The IPX HEADER fields have the following meaning:

chksum Originally this field held a value that represented the sum of the bytes contained in the packet. It is a holdover from the original Xerox Network Standard protocol and it is not used because IPX makes its own internal checking. IPX sets this field to the value FFFFH.

length This field contains the number of bytes contained in the message structure.

transport-control

This field is used by IPX to control the number of bridges that the packet is to cross. It is incremented by one each time it reaches a bridge. The message will not be delivered when the count reaches 16.

packet_type

This field must contain a value of four (4) to identify the

packet of the Packet Exchange Packet type. The use of this field permits different protocols to coexist on the same network, each differentiated by the value of this field. This field is a holdover from the original Xerox Network Standard, predecessor of IPX.

dest_num, dest_node, dest_socket

These fields serve as the postal address of the receiving node. A valid value in both the dest_num and dest_node will direct the packet to a specific station. A value of 0H as the dest_num and a value FFFFFFH as the dest_node will transmit the packet to every station on the same physical network that is listening in on the specified socket.

sour_num, sour_node, sour_socket

These fields serve as the postal address of the sending node. It is not necessary to fill in these fields because IPX will do so.

The next structure is the message structure itself. It can have any form that is needed for any particular purpose. One important limitation is that the length of this structure is limited to 546 bytes per fragment, up to 1500 bytes per packet. Once the necessary ECB or IPX fields have been filled, the "send" or "listen" command can be issued. If you look at the function listen(), you will notice that none of the IPX fields are used. There is no need to fill these fields; this information will be filled out with the arriving message. However, it is necessary to complement some of the ECB fields. Be sure that the socket field has the same value as the one used for "sending", or the message will never arrive. IPX must also know the number of fragments that it expects to receive (frag_-count) and where it will start to store the arriving information (frag_address). Once the listen is initiated, control returns to the program where it can loop checking the ecb.in_use flag until the arrival of a message or continue with other functions.

There is more work to be done in order to send a message. It is necessary to fill more fields of the control structure and header. Besides the fields filled out to initiate a listen() command, the following IPX HEADER fields must be complemented:

chksum — This field is optionally set to FFFFH because it is not used. It is good practice to assign this value to avoid random values to be assigned to this field upon initialization.

packet_type — This field is always set to four.

dest_socket & sour_socket — These fields are set to the value of the socket used for transmission. Make sure that sour_socket and the socket the program is listening on are the same.

dest_node — This field stores the address of the destination node. The value FFFFFFH means that the packet will be transmitted to every station, not one in particular.

dest_num — This field stores the address of the physical network the message will be travelling to. The message will not cross bridges if the value zero is used. It will rather stay within the same physical network.

Using these last two fields we can obtain the immediate address. Since we will not cross a bridge, IPX will place a value in this field that will permit it to reach every node within the network.

To obtain the immediate address we use the IPX GET_TARGET_FN. This function retrieves the value based on the

information provided to it by the dest_node and dest_num. It will return six bytes (the immediate address) into the area we designate for holding this value.

With the destination information set in the variables we may issue the send command (IPX function 03H). One important fact is that sending the information does not guarantee its arrival. The in_use flag changes state to indicate that it has successfully transmitted the information onto the line, not that it has been received.

Finally, once the information arrives, the receiving node will process it using the TURBO C system() function. This function loads another copy of COMMAND.COM, the PC's main command parser and executer, and provides a DOS environment were the command will execute.

One last comment: to provide a way so that control of the server is returned to the user without need of rebooting, we have included an interrupt routine which, upon a keyboard interrupt (i.e., every time the user strikes a key), checks for the CTRL-C sequence. If the sequence has been keyed, the program gives a final message and returns safely to the DOS environment.

Recommendations

It was stated that the program contained the basic essentials so that it will run. The following suggestions will make the program more versatile:

- Use the DOS interrupt 21H function 31H to create terminate and stay resident programs. For example, keep an event service routine resident

in memory constantly checking for the arrival of messages, after which it will interrupt the current PC task to perform the requested operation.

- The system() function is not the most resourceful way to execute .COM, .BAT and .EXE files because it loads another copy of COMMAND. COM (necessary to execute DOS commands). Try using exec() and spawn() functions instead. However, each one has its own advantages and disadvantages.

- REMOTE does not verify that the message is received by the next station. A second socket could be open where the receiving workstation could send a reply message back to the transmitting workstation and another second socket needs to be used to send replies to prevent the transmitting station from listening in on its own message.

- Try new ideas, experiment a little, modify the source code and try them out.

Appendix 1
Source code of program REMOTE

```
#include <stdlib.h>
#include <string.h>
#include <dos.h>
#include <stdio.h>
#include <conio.h>
#include <process.h>
/*
                    PROTOTYPES
*/
void main (int argc, char *argv[]);
void send (void);
void get_local_conn(void);
void open_sockets(void);
void listen(void);
void check_arg(int argc, char *argv[]);
void quit (char *);

/*
                    MESSAGES
*/
char installed[]   = "Listen is already installed",
dos[]              = "You need DOS version 3.x or higher to run"
  " this program",
     network[]     = "Network not installed at this terminal",
     no_socket[]   = "ERROR while trying to opening socket",
     help[]        = "Usage:\n"
                     "       REMOTE SERVER\n"
                     "       REMOTE command\n";
/*
```

STRUCTURES

```
*/

union  REGS  regs;
struct SREGS sregs;

[3] struct ECB
{
void far      *link;
void far      (*ESR)();
char          in_use,
              cmpt_code;
int           socket;
long          IPX_wrk;
char          driver_wrk[12],
              imdt_add[6];
int           frag_count;
void far      *frag_address;
int           frag_size;
};

[4] struct IPX_HEADER
{
int           chksum,
              length;
char          transport_control,
              packet_type;
long          dest_num;
char          dest_node[6];
int           dest_socket;
```

```
long            sour_num;
char            sour_node[6];
int             sour_socket;
};

struct MESSAGE_AREA
{
char            command[128];
};

struct
{
struct          ECB             ecb;
struct          IPX_HEADER      ipx;
struct          MESSAGE_AREA    msg;
}LISTEN,SEND;

/*
```

DEFINITIONS

```
*/

#define   CONN_NUM       0xdc
#define   IPX_VECT       0x7a
#define   MULTI_INT      0x2f
#define   OPEN_FN        0x00
#define   CLOSE_FN       0x01
#define   SEND_FN        0x03
#define   LISTEN_FN      0x04
#define   GET_TARGET_FN  0X07

int i;
char *p;
/*
```

CHECK FOR INCOMMING MESSAGES

```
*/
```

```
void execute()
{
system(LISTEN.msg.command);
}
/*
```

LISTEN FOR PACKET

```
*/
```

```
void listen(void)
{
/* Prepare event control block for receiving information */

LISTEN.ecb.in_use = 1;
LISTEN.ecb.socket = 0x4545;
LISTEN.ecb.frag_count = 1;
LISTEN.ecb.frag_address = (char far *) &LISTEN.ipx;
LISTEN.ecb.frag_size=(sizeof(LISTEN.ipx)+sizeof(LISTEN.msg));

/* Listen for incomming commands */

regs.x.bx = LISTEN_FN;
regs.x.si = (unsigned int) &LISTEN;
sregs.es = FP_SEG(&LISTEN);
int86x(IPX_VECT,&regs,&regs,&sregs);
}
/*
```

SEND COMMANDS TO OTHER TERMINALS

```
*/
```

6

```
void send_command(void)
{
/* Fill in event control block */
SEND.ecb.socket = 0x4545;
SEND.ecb.frag_count = 1;
SEND.ecb.frag_address = (char far *) &SEND.ipx;
SEND.ecb.frag_size = sizeof(SEND.ipx)+sizeof(SEND.msg);

/*Fill in IPX packet header */
SEND.ipx.chksum = 0xffff;
SEND.ipx.packet_type=4;

/* Send commands to every listening terminal */

memset(SEND.ipx.dest_node,0xff,sizeof(SEND.ipx.dest_node));
SEND.ipx.dest_socket = SEND.ipx.sour_socket =0x4545;

/* Get address of receiving terminal */
```

7

```
regs.x.bx = GET_TARGET_FN;
regs.x.si = (unsigned)&SEND.ipx.dest_num;
regs.x.di = (unsigned)SEND.ecb.imdt_add;
sregs.es = FP_SEG(&SEND);
int86x(IPX_VECT,&regs,&regs,&sregs);

/* Send commands */

printf("sending\n");
```

8

```
regs.x.bx = SEND_FN;
```

```
sregs.es = FP_SEG(&SEND.ecb);
regs.x.si = FP_OFF(&SEND.ecb);
int86(IPX_VECT,&regs,&regs);

}
/*
```

GET THIS STATION'S LOGIN INFORMATION

```
*/
void get_local_conn(void)
{
/* Verify that the station is connected to the network */

regs.h.ah = CONN_NUM;
intdos(&regs,&regs);
if (!regs.h.al)
    quit(network);
}
/*
```

OPEN SOCKET FOR LISTENING

```
*/

2
void open_socket(void)
{
/* Open socket for listening */

regs.x.bx = OPEN_FN;

/* Open short-lived socket */

regs.h.al = 0;
regs.x.dx = 0x4545;
int86(IPX_VECT,&regs,&regs);
```

```
    if (regs.h.al)
        quit(no_socket);
    }
    /*
```

GIVE ERROR AND EXIT ROUTINE

```
    */
    void quit (char *msg)
    {
    printf(msg);
    exit (1);
    }
    /*
```

MAIN

```
    */
    void main(int argc, char *argv[])
    {
    get_local_conn();
    open_socket();
    listen();
    check_arg(argc,argv);
    }
    /*
```

GET COMMAND AND PREPARE FOR SENDING

```
*/

void check_arg(int argc, char *argv[])
{
if (argc<2)
    quit(help);
if (!strcmp(argv[1],"server"))
```

```
    {
    printf("This terminal is acting as a SERVER\n");
    while(1)
        {
        if (! LISTEN.ecb.in_use)
            {
            execute();
            listen();
            printf("This terminal is acting as a SERVER\n");
            }
        }

    }
else
    {
    /* Organize all arguments into one line */

    for (i = 1; i < argc; i++)
        {

        /* Eliminate blank spaces between arguments
and make new string */

        if (strchr(argv[i], ' '))
            {
            p = malloc(strlen(argv[i]) + 3);
            sprintf(p, "\"%s\"", argv[i]);
            argv[i] = p;
            }
        /* place 1 space between arguments and make
one string */
        if (i != 1)
            strcat(SEND.msg.command, " ");
        strcat(SEND.msg.command, argv[i]);
```

```
            }
        send_command();
        }
    }
    /*
```


```
    */
    I 0
    void interrupt far cb (void)
    {
    printf("quitting REMOTE");
    return(0);
    }
```