

Figure 1: The host maintains an internal table of clients.

```
process_listen_queue()
{
    ECB * ptr, * available_ecb;
    DATA_STRUCT * data_received;
    IPXRelinquishControl();
    if (Globals.ListensPosted == (ECB *) NULL)
    {
        return 0;
    }
    disable(); /* enter critical section */
    ptr = Globals.ListensPosted;
    Globals.ListensPosted =
    Globals.ListensPostedTail = NULL;
    enable(); /* exit critical section */
    while (ptr)
    {
        data_received = (DATA_STRUCT *)FP_OFF(ptr->
            fragmentDescriptor[1].address);
        switch(ptr->completionCode)
        {
            case IPX_SUCCESSFUL:
#ifdef DEBUG
                printf("  IPX packet fragment[1]: %s:%d\n",
                    data_received->databuffer, data_received->sequenceNumber);
#endif
                if (strcmp("AVAIL", data_received->databuffer) == 0)
                {
                    process_AVAIL(ptr);
                }
                else if (strcmp("READY", data_received->databuffer) == 0)
                {
                    process_READY(ptr);
                }
                else if (strcmp("ACK", data_received->databuffer) == 0)
                {
                    process_ACK(ptr);
                }
                break;
            case IPX_REQUEST_CANCELED:
                fatal_err("IPXListenForPacket() cancelled unexpectedly!");
                break;
            case IPX_BAD_PACKET:
                fatal_err("Internal data corruption resulted in bad packet.");
                break;
```

```
        case IPX_SOCKET_NOT_OPEN:
            fatal_err("Internal data corruption resulted in closed socket.");
            break;
        default:
            fatal_err("Unknown error completion code for listen packet.");
            break;
    }
    available_ecb = ptr;
    ptr = (ECB *) ptr->linkAddress;
    IPXListenForPacket(available_ecb);
}
}
```

Figure 2: Both the host and the client have functions that send messages to each other.

```
send_messg_to_client(CLIENT_INFO * client_info, char * messg_to_client)
{
    ECB * sendECB,
        * listenECB;
    DATA_STRUCT * data_to_send; /* for ease of use in debugging */
    int retry = 0;
    int ccode = 0;

    /******
    * The protocol for sending packets will be:
    * 1. post a listen for an acknowledgement
    * 2. if more retries left then send the packet
    * 3. look for the acknowledgement but if timeout occurs goto 2
    * NOTE: the listens for this protocol gets posted in either
    *       process_listen_queue(), and setup_test_server().
    *****/

    client_info->status = IN_USE_STATE; /* NOTE the state transition! */

    /* prepare the data for transmission */
    sendECB = get_send_ecb(client_info);
    data_to_send=(DATA_STRUCT *)FP_OFF(sendECB->fragmentDescriptor[1].address);
    strcpy(data_to_send->databuffer, messg_to_client);

    /******
    * NOTE that the host's sequenceNumber and the client's
    * ackSequenceNumber must never start equal! Insuring this here
    * is a simple solution.
    *****/
    data_to_send->sequenceNumber = Globals.sequenceNumber;
    client_info->ackSequenceNumber = Globals.sequenceNumber - 1;

    /******
    * You need to feel comfortable about what is going on here
    * with IPXSendPacket(), IPXListenForPacket(), and
    * IPXRelinquishControl(). Check the NetWare C-Interface
    * documentation if you need help.
    *****/
    IPXSendPacket(sendECB);
    while(sendECB->inUseFlag)
    {
        IPXRelinquishControl();
    }
    /******
    * Waiting for the send to complete looks very time consuming
    *****/
}
```

```

* but it isn't really. Anyway you can't do much until you
* know that the packet was successfully sent.
*****/
}
ccode = sendECB->completionCode;
if (!ccode)
{
    printf("sent '%s':%d\n", messg_to_client, Globals.sequenceNumber);

    start_timer();
    while (client_info->status != ACK_STATE)
    {
        process_listen_queue(); /* filter in the ACK */

        /* here's your timeout check */
        if (Globals.time_units > MAX_WAIT_FOR_ACK) /* timed out yet? */
        {
            /******
            * Frequent timeouts probably imply slow processing on
            * the part of either the client or the host. In the
            * latter case, the culprit is likely process_listen_queue().
            * Try experimenting with the value for MAX_WAIT_FOR_ACK.
            *****/
            printf("timeout occurred in send_messg_to_client()\n");
            if (retry++ < MAX_RETRIES) /* here's your retry counter */
            {
                stop_timer();
                IPXSendPacket(sendECB);
                while (sendECB->inUseFlag)
                    IPXRelinquishControl(); /* wait for send to
complete */

                ccode = sendECB->completionCode;
                if (ccode)
                    break;
                printf("re-sent '%s':%d\n", messg_to_client,
Globals.sequenceNumber);
                start_timer();
            }
            else
            {
                ccode = -1;
            }
        }
    }
    stop_timer();
}
}

```

```

if (!ccode)
{
    printf("Received ACK:%d for '%s'.\n", Globals.sequenceNumber,
        messg_to_client);
}
/*****
* I guess you could probably reuse the sequenceNumber for all of
* the clients getting the same command, but it's more conventional
* to increment it after each individual send is complete.
*****/
Globals.sequenceNumber++;
return ccode;
}

acknowledge_receipt(CLIENT_INFO * client_info)
{
    ECB * sendECB;
    /*****
    * The protocol for receiving packets will be:
    * 1. post a listen for a packet
    * 2. upon receipt of a packet, send an "ACK"nowledgement
    *
    * NOTE: the posting of the listen and the subsequent receipt
    *       for this protocol took place in either
    *       process_listen_queue(), or setup_test_server().
    * NOTE: like the canned messages in process_job_script() I'm
    *       using strings for the ACK. In a real application
    *       you'd surely use an enumerated type in the packet's
    *       data area to indicate the purpose of the packet.
    *****/

    /* prepare the data */
    sendECB = get_send_ecb(client_info);
    strcpy(((DATA_STRUCT *)
        FP_OFF(sendECB->fragmentDescriptor[1].address))->databuffer,
        "ACK");
    ((DATA_STRUCT *)
        FP_OFF(sendECB->fragmentDescriptor[1].address))->sequenceNumber =
        client_info->messgSequenceNumber;

    /* transmit the acknowledgement */
    IPXSendPacket(sendECB);
    while(sendECB->inUseFlag)
        IPXRelinquishControl();
    if(sendECB->completionCode)

```

```

{
    fatal_err("IPXSendPacket() failed");
}
/*****
* If this ACK gets dropped, the client will eventually timeout
* and resend its packet. Then this routine will get called
* again by the host.
*****/
printf("sent 'ACK':%d\n", client_info->messgSequenceNumber);
}

```

Figure 3: The ESR places the posted listen ECBs in a queue.

```
void IPXReceiveESR(ECB *ecb)
{
    ECB * new_listenECB;

    /******
    * This is just a queue. A stack could be used just as well. The
    * order in which packets are processed isn't very important.
    *****/
    if (Globals.ListensPosted)
    {
        (ECB *)Globals.ListensPostedTail->linkAddress = ecb;
        (ECB *)Globals.ListensPostedTail = ecb;
    }
    else
    {
        (ECB *)Globals.ListensPostedTail =
        (ECB *)Globals.ListensPosted = ecb;
    }
    ecb->linkAddress = (void far *) NULL; /* it's now at the end of the queue */
}
```

Figure 4: The host processes the AVAIL packet with process_AVAIL.

```
process_AVAIL(ECB * ecb)
{
    static CLIENT_INFO * tmp;
    IPXHeader * client_header;
    DATA_STRUCT * data_received;

    /* these next two lines make debugging easier */
    client_header = (IPXHeader *)FP_OFF(ecb->fragmentDescriptor[0].address);
    data_received = (DATA_STRUCT*)FP_OFF(ecb->fragmentDescriptor[1].address);

    tmp = find_client_info(client_header);
    if (!tmp)
    {
        /******
        * This is where the host's table of clients is added to.
        *****/
        tmp = (CLIENT_INFO *) calloc(1, sizeof(CLIENT_INFO));
        if (!tmp)
        {
            fatal_err("Out of memory!"); /* no return from here */
        }
        else
        {
            /******
            * Store the information needed for IPX packets sent by the host.
            *****/
            memcpy((char *)&tmp -> ipx_address, (char *)&client_header->source,
                sizeof(IPXAddress));
            memcpy(tmp -> immediateAddress, ecb->immediateAddress, 6);

            /******
            * At this point, the host believes the connection is established.
            *****/
            tmp->status = AVAIL_STATE;
            tmp->messgSequenceNumber = data_received -> sequenceNumber;

            /******
            * Update the Globals information to include this client.
            *****/
            tmp -> next = Globals.client_list;
            Globals.client_list = tmp;

            /* this is only interesting for debugging purposes */
            Globals.clients_available++;
        }
    }
}
```



```

    }
}
else
{
    /******
    * It's still possible that some bugs exist here. NOTE that
    * it's really okay for additional instances of AVAIL packets
    * to exist for this client. They could occur from the host
    * being slow enough that client timeouts took place on the
    * sends for the AVAIL. That would cause resends to occur.
    * However, no client should try to reenter the AVAIL after
    * having been in another. This is true even if your application
    * has occasion to abolish a connection with a client and then
    * allow the client to reestablish the connection. Note that in
    * that case the client would not have an entry in the host's
    * client table at the time of the new connection and there
    * would be no conflict with formerly existing states.
    *****/
    if (tmp->status != AVAIL_STATE)
        fatal_err("Unexpected internal state!"); /* no return from here */
}

/******
* If the client successfully receives this ACK it will know
* that the connection has been established. If not it will
* try again.
*****/
acknowledge_receipt(tmp);
}

process_READY(ECB * ecb)
{
    CLIENT_INFO * client_ptr;

    client_ptr = find_client_info((IPXHeader *)
        FP_OFF(ecb->fragmentDescriptor[0].address));
    if (client_ptr)
    {
        client_ptr->messgSequenceNumber = ((DATA_STRUCT *)
            FP_OFF(ecb->fragmentDescriptor[1].address))->sequenceNumber;
        /******
        * The correct transition from one state to READY is controlled
        * here.
        *****/
        switch(client_ptr->status)
        {
            /* BE AWARE THAT I'M INTENTIONALLY FALLING THROUGH THE CASES */

```

```

        case AVAIL_STATE:
        case ACK_STATE:
            Globals.clients_ready++;
            client_ptr->status = READY_STATE;
            case READY_STATE:
                acknowledge_receipt(client_ptr);
                break;
            case IN_USE_STATE:
                /******
                * Ignore it since I'm probably looking for an ACK
                * and this is an extra READY from when the client
                *   timed out.
                *****/
                break;
            default:
                /* It's still possible that some bugs exist here. */
                fatal_err("Unexpected internal state!"); /* no return from here */
                break;
        }
    }
    else
    {
        nonfatal_err("Received a READY from an unknown client!");
    }
}

process_ACK(ECB * ecb)
{
    CLIENT_INFO * client_ptr;

    client_ptr = find_client_info((IPXHeader *)
        FP_OFF(ecb->fragmentDescriptor[0].address));
    if (client_ptr)
    {
        if(client_ptr->status == IN_USE_STATE)
        {
            /******
            * This is where you discover that the transmission has been
            * successful. Verify it with the sequenceNumber and perform
            * the state transition on the client.
            *****/
            client_ptr->ackSequenceNumber = ((DATA_STRUCT *)
                FP_OFF(ecb->fragmentDescriptor[1].address))->sequenceNumber;
            if (client_ptr->ackSequenceNumber == Globals.sequenceNumber)
                client_ptr->status = ACK_STATE;
        }
    }
}

```

```
        else
        {
            /* It's still possible that some bugs exist here. */
            fatal_err("Unexpected internal state!"); /* no return from here */
        }
    }
    else
    {
        nonfatal_err("Received an ACK from an unknown client!");
    }
}
```

Figure 5: TSERVER.C source code.

```

/*****
* File:      TSERVER.C
* Program:   TSERVER.EXE
* Purpose:   Act as a test server to companion TCLIENT.EXE
* Programmer: Steve Shanafelt
*****/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mem.h>
#include <dos.h>
#include <nit.h>
#include <nxt.h>
#include <ipxdefs.h>
#include <mapcap.h>

#define MAX_WAIT_FOR_CLIENT 4
#define MAX_WAIT_FOR_ACK 1
#define MAX_RETRIES 30
#define SEND_ECB 0
#define RECEIVE_ECB 1
#define LISTEN_POOL_SIZE 10
#define SHUTDOWN_SEQUENCE -1

typedef struct client_struct {
    CLIENT_STATE status;
    int messgSequenceNumber;
    int ackSequenceNumber;
    IPXAddress ipx_address;
    BYTE immediateAddress[6];
    struct client_struct * next;
} CLIENT_INFO;

struct tserver_globals {
    int clients_available,
        clients_ready,
        server_status;
    CLIENT_INFO * client_list;
    int seconds;
    int timerEnabled;
    int vas_index;
    ECB timerECB;
} Globals;
```

```
ECB * AvailECBPool = (ECB *) NULL;
ECB * ListensPosted = (ECB *) NULL;
ECB * ListensPostedTail = (ECB *) NULL;
```

```
extern void far      IPXReceiveESRHandler(void);
extern void far  IPXSendESRHandler(void);
extern void far      timerESRHandler();
```

```
main(void);
fatal_err(char * messg);
nonfatal_err(char * messg);
setup_test_server(void);
process_job_script(void);
send_messg_to_client(CLIENT_INFO * client_info, char * messg_to_client,
    int sequenceNumber);
cleanup(void);
shutdown_clients(void);
free_all_ecbs(void);
void timerESR(ECB *ecb);
void IPXReceiveESR(ECB *ecb);
void IPXSendESR(ECB *ecb);
ECB * get_ecb(void);
ECB * get_listen_ecb(void);
ECB * get_send_ecb(CLIENT_INFO * client_info);
process_listen_queue(void);
CLIENT_INFO * find_client_info(IPXHeader * client_header);
process_AVAIL(ECB * ecb);
process_READY(ECB * ecb);
process_ACK(ECB * ecb);
start_timer(void);
stop_timer(void);
StartAdvertise(char * vas_name, WORD vas_type, WORD vas_socket);
StopAdvertise(char * vas_name, WORD vas_type, WORD vas_socket, int vas_index);
```

```
main()
{
    Globals.timerEnabled = 1;
    setup_test_server();
    process_job_script();
    cleanup();
    return 0;
}
```

```
fatal_err(char * messg)
```

```

{
    printf("%s\n", messg);
    cleanup();
    exit(1);
}

nonfatal_err(char * messg)
{
    printf("%s\n", messg);
}

setup_test_server()
{
    int i;
    WORD testServerSocket;
    ECB * listenECB;
    int ccode;

    Globals.client_list = (CLIENT_INFO *) NULL;
    Globals.clients_available = 0;
    Globals.seconds = 0;
    /* initialize C routines for IPX calls */
    if (IPXInitialize() != 0)
    {
        fatal_err("IPX not installed.");
    }

    /* open socket, short lived */
    testServerSocket = IntSwap(MASTER_SOCKET);
    ccode = IPXOpenSocket((BYTE *)&testServerSocket, 0);
    if(ccode != 0 && ccode != IPX_SOCKET_ALREADY_OPEN)
    {
        fatal_err("IPXOpenSocket() failed.");
    }

    /* advertise this as a VAS */
    Globals.vas_index = StartAdvertise("TFSERVER", MASTER_TYPE, MASTER_SOCKET);

    /* post some listens for packets */
    for (i=0; i < LISTEN_POOL_SIZE; i++)
    {
        listenECB = get_listen_ecb();
        IPXListenForPacket(listenECB);
    }
    /* set up Globals.timerECB */
    Globals.timerECB.ESRAddress = timerESRHandler;

```

```

}

process_job_script()
{
    char * job_script[] = {
        "FIRST",
        "SECOND",
        "THIRD",
        NULL
    };
    char ** script_line;
    int clients_needed;
    CLIENT_INFO * client_ptr;
    int sequenceNumber = 1;
    int retry = 0;

    script_line = job_script;

    /*****
    * This loop in its final state will continue until the job script *
    * is complete and all clients are shutdown. *
    *****/

    while(*script_line)
    {
        clients_needed = 1;
        IPXRelinquishControl();          /* do nothing while waiting */
        start_timer();
        while (Globals.clients_ready < clients_needed)
        {
            IPXRelinquishControl();      /* do nothing while waiting */
            process_listen_queue();
            if (Globals.seconds > MAX_WAIT_FOR_CLIENT)
            {
                stop_timer();
                if (retry < MAX_RETRIES)
                {
                    printf("still waiting for %d ready clients\n", clients_needed);
                }
                else
                {
                    fatal_err("maximum number of retries exceeded for ready
clients\n");
                }
                start_timer();
            }
        }
    }
}

```

```

    }
    stop_timer();
    for (client_ptr = Globals.client_list;
         client_ptr;
         client_ptr = client_ptr -> next
        )
    {
        if (client_ptr->status == READY_STATE)
        {
            send_messg_to_client(client_ptr, *script_line,
                                sequenceNumber);
            Globals.clients_ready--;
            clients_needed--;
        }
    }
    script_line++;
    sequenceNumber++;
    if (sequenceNumber < 1) /* sequence numbers < 1 are reserved */
        sequenceNumber = 1;
}
shutdown_clients();
}

```

```

send_messg_to_client(CLIENT_INFO * client_info, char * messg_to_client,
                    int sequenceNumber)

```

```

{
    ECB * sendECB,
        * listenECB;
    DATA_STRUCT * data_to_send; /* for ease of use in debugging */
    /******
    * The protocol for sending packets will be:
    * 1. post a listen for an acknowledgement
    * 2. send the packet
    * 3. wait for an acknowledgement until timeout
    * 4. if timeout occurs goto 2
    *****/
}

```

```

    client_info->status = IN_USE_STATE;
    sendECB = get_send_ecb(client_info);
    data_to_send=(DATA_STRUCT *)FP_OFF(sendECB->fragmentDescriptor[1].address);
    strcpy(data_to_send->databuffer, messg_to_client);
    data_to_send->sequenceNumber = sequenceNumber;
    IPXSendPacket(sendECB);
    while(sendECB->inUseFlag)
        IPXRelinquishControl();
    if(sendECB->completionCode)

```



```

{
    fatal_err("IPXSendPacket() failed");
}

start_timer();
while(client_info->status != ACK_STATE &&
    client_info->ackSequenceNumber != sequenceNumber
    )
{
    process_listen_queue(); /* filter in the ACK */
    IPXRelinquishControl();
    if (Globals.seconds > MAX_WAIT_FOR_ACK)
    {
        stop_timer();
        IPXSendPacket(sendECB);
        while (sendECB->inUseFlag)
        {
            IPXRelinquishControl();          /* wait for send to complete */
        }
        if (sendECB->completionCode)
        {
            fatal_err("IPXSendPacket() failed.");
        }
        start_timer();
    }
}
stop_timer();
printf("Received ACK:%d for '%s'.\n", sequenceNumber, messg_to_client);
}

```

```

acknowledge_receipt(CLIENT_INFO * client_info)
{
    ECB * sendECB;
    /******
    * The protocol for receiving packets will be:
    * 1. post a listen for a packet
    * 2. upon receipt of a packet, send an "ACK"nowledgement
    *
    * NOTE: the listen for this protocol gets posted in either
    * process_listen_queue(), and setup_test_server().
    *****/

    sendECB = get_send_ecb(client_info);
    strcpy(((DATA_STRUCT *)
        FP_OFF(sendECB->fragmentDescriptor[1].address))->databuffer,
        "ACK");
}

```

```

((DATA_STRUCT *)
    FP_OFF(sendECB->fragmentDescriptor[1].address))->sequenceNumber =
    client_info->messgSequenceNumber;
IPXSendPacket(sendECB);
while(sendECB->inUseFlag)
    IPXRelinquishControl();
if(sendECB->completionCode)
{
    fatal_err("IPXSendPacket() failed");
}
printf("sent 'ACK':%d\n", client_info->messgSequenceNumber);
}

cleanup()
{
    ECB * ptr;

    stop_timer();
    StopAdvertise("TFSERVER", MASTER_TYPE, MASTER_SOCKET,
        Globals.vas_index);
    IPXCloseSocket(MASTER_SOCKET);
    if (ListensPosted == (ECB *) NULL)
    {
        return 0;
    }
    for (ptr = ListensPosted; ptr; ptr = (ECB *) ptr->linkAddress)
    {
        if (ptr->inUseFlag)
            IPXCancelEvent(ptr);
    }
}

shutdown_clients()
{
    CLIENT_INFO * client_ptr;

    for (client_ptr = Globals.client_list; client_ptr;
        client_ptr = client_ptr -> next
        )
    {
        send_messg_to_client(client_ptr, "QUIT", SHUTDOWN_SEQUENCE);
    }
}

free_all_ecbs()
{

```

```

    /* stub */
}

/*****C-side of the ESRs*****/

void timerESR(ECB *ecb)
{
    Globals.seconds++;
    IPXScheduleSpecialEvent(18, ecb);
}

void IPXReceiveESR(ECB *ecb)
{
    ECB * new_listenECB;

    if (ListensPosted)
    {
        (ECB *)ListensPostedTail->linkAddress = ecb;
        (ECB *)ListensPostedTail = ecb;
    }
    else
    {
        (ECB *)ListensPostedTail =
        (ECB *)ListensPosted = ecb;
    }
    ecb->linkAddress = (void far *) NULL; /* it's now at the end of the queue */
}

void IPXSendESR(ECB *ecb)
{
    ECB * tmp;
    /*-----main body */
    /*
    tmp = AvailECBPool;
    AvailECBPool = ecb;
    (ECB *)AvailECBPool->linkAddress = tmp;
    */
}

/*****end of C-side of the ESRs*****/

ECB * get_ecb()
    /* Descrip:
        get ECB pool member if available, otherwise allocate one
    */
{

```

```

    ECB * ecb;
    IPXHeader * ipxheader;
    DATA_STRUCT * data;
    /*-----main body */
    /* the linked list is empty */
    ecb = (ECB *) calloc(1, sizeof(ECB)); /* avail is set to 0 */
    if (ecb == NULL)
    {
        printf("Out of memory allocating for ConnectionECB\n ");
        exit(-1);
    }
    ipxheader = (IPXHeader *) calloc(1, sizeof(IPXHeader));
    if (ipxheader == (IPXHeader *) NULL)
    {
        printf("Out of memory allocating for IPX header.\n");
        exit(-1);
    }
    data = (DATA_STRUCT *) calloc(1, sizeof(DATA_STRUCT));
    if (data == (DATA_STRUCT *) NULL)
    {
        printf("Out of memory allocating for IPX header's data buffer.\n");
        exit(-1);
    }

    ecb->fragmentCount = 2;
    ecb->fragmentDescriptor[0].address = (char far *) ipxheader;
    ecb->fragmentDescriptor[0].size = sizeof(IPXHeader);
    ecb->fragmentDescriptor[1].address = (char far *) data;
    ecb->fragmentDescriptor[1].size = sizeof(DATA_STRUCT);

    return ecb;
} /* end of get_ecb */

ECB * get_listen_ecb()
{
    ECB * listenECB;

    listenECB = get_ecb();
    listenECB->ESRAddress = IPXReceiveESRHandler;
    listenECB->socketNumber = IntSwap(MASTER_SOCKET);

    return listenECB;
}

ECB * get_send_ecb(CLIENT_INFO * client_info)
{

```

```

static ECB sendECB;
static IPXHeader dest_header;
static DATA_STRUCT data;

/*****/
* NOTE: this approach of using only one sendECB will only *
* work if an ACK is waited for with each send.          *
*****/

int first_time = 1;

if (first_time)
{
    sendECB.socketNumber = IntSwap(CLIENT_SOCKET);
    sendECB.ESRAddress = IPXSendESRHandler;
    sendECB.fragmentCount = 2;
    sendECB.fragmentDescriptor[0].address =
        (char far *) &dest_header;
    sendECB.fragmentDescriptor[0].size =
        sizeof(IPXHeader);
    sendECB.fragmentDescriptor[1].address =
        (char far *) &data;
    sendECB.fragmentDescriptor[1].size =
        sizeof(DATA_STRUCT);
    first_time = 0;
}

memcpy ((char *) &dest_header.destination,
        (char *) &client_info->ipx_address, sizeof(IPXAddress));
/*****/
* NOTE: client_info only contains the socket used by the client *
* to send packets on...not the one it is required to receive on! *
*****/
*(WORD *)dest_header.destination.socket = IntSwap(CLIENT_SOCKET);

memcpy(sendECB.immediateAddress, client_info->immediateAddress,
        sizeof(BYTE) * 6);

return &sendECB;
}

/*****/

process_listen_queue()
/* Descrip:
    process the queue of ECBs received by SPX

```

```

    */
{
    /* // */
    ECB * ptr, * prevnode, * available_ecb;
    DATA_STRUCT * data_received;
    /*-----main body */
    if (ListensPosted == (ECB *) NULL)
    {
        return 0;
    }
    for (ptr = ListensPosted, prevnode = NULL; ptr; )
    {
        if (ptr->inUseFlag)
        {
            prevnode = ptr;
            ptr = (ECB *) ptr -> linkAddress;
            continue;
        }
        else
        {
            data_received = (DATA_STRUCT *)FP_OFF(ptr->
                fragmentDescriptor[1].address);
            switch(ptr->completionCode)
            {
                case IPX_SUCCESSFUL:

#ifdef DEBUG
                    printf("  IPX packet fragment[1]: %s:%d\n",
                        data_received->databuffer, data_received-
>sequenceNumber);
#endif
                    if (strcmp("AVAIL", data_received->databuffer) == 0)
                    {
                        process_AVAIL(ptr);
                    }
                    else if (strcmp("READY", data_received->databuffer) == 0)
                    {
                        process_READY(ptr);
                    }
                    else if (strcmp("ACK", data_received->databuffer) == 0)
                    {
                        process_ACK(ptr);
                    }
                    break;
                case IPX_REQUEST_CANCELED:
                    fatal_err("IPXListenForPacket() cancelled unexpectedly!");
                    break;
                case IPX_BAD_PACKET:

```

```

        fatal_err("Internal data corruption resulted in bad packet.");
        break;
    case IPX_SOCKET_NOT_OPEN:
        fatal_err("Internal data corruption resulted in closed socket.");
        break;
    default:
        fatal_err("Unknown error completion code for listen packet.");
        break;
    }
    if (prevnode)
        (ECB *) prevnode -> linkAddress = (ECB *) ptr->linkAddress;
    else
        ListensPosted = ptr;
    available_ecb = ptr;
    ptr = (ECB *) ptr->linkAddress;
    IPXListenForPacket(available_ecb);
}
}
/* end of process_listen_queue */

```

```

CLIENT_INFO * find_client_info(IPXHeader * client_header)
{
    static CLIENT_INFO * client_ptr;

    for (client_ptr = Globals.client_list; client_ptr;
         client_ptr = client_ptr -> next)
    {
        if (memcmp (&client_ptr->ipx_address, &client_header->source,
                    sizeof(IPXAddress)) == 0)
        {
            break;
        }
    }
    return client_ptr;
}

```

```

process_AVAIL(ECB * ecb)
{
    static CLIENT_INFO * tmp;
    IPXHeader * client_header;
    DATA_STRUCT * data_received;

    /* these next two lines make debugging easier */
    client_header = (IPXHeader *)FP_OFF(ecb->fragmentDescriptor[0].address);
    data_received = (DATA_STRUCT *)FP_OFF(ecb->fragmentDescriptor[1].address);
}

```

```

tmp = find_client_info(client_header);
if (!tmp)
{
    /* set up client information */
    tmp = (CLIENT_INFO *) calloc(1, sizeof(CLIENT_INFO));
    if (!tmp)
    {
        fatal_err("Out of memory!"); /* no return from here */
    }
    else
    {
        memcpy((char *)&tmp -> ipx_address, (char *)&client_header->source,
            sizeof(IPXAddress));
        memcpy(tmp -> immediateAddress, ecb->immediateAddress, 6);
        tmp->status = AVAIL_STATE;
        tmp->messgSequenceNumber = data_received -> sequenceNumber;
        /* update Globals with new client */
        tmp -> next = Globals.client_list;
        Globals.client_list = tmp;
        Globals.clients_available++;
    }
}
else
{
    if (tmp->status != AVAIL_STATE)
        fatal_err("Unexpected internal state!"); /* no return from here */
}
acknowledge_receipt(tmp);
}

```

```

process_READY(ECB * ecb)
{
    CLIENT_INFO * client_ptr;

    client_ptr = find_client_info((IPXHeader *)
        FP_OFF(ecb->fragmentDescriptor[0].address));
    if (!client_ptr)
    {
        nonfatal_err("Received a READY from an unknown client!");
    }
    else
    {
        client_ptr->messgSequenceNumber = ((DATA_STRUCT *)
            FP_OFF(ecb->fragmentDescriptor[1].address))->sequenceNumber;
        switch(client_ptr->status)

```



```

{
    /* BE AWARE THAT I'M INTENTIONALLY FALLING THROUGH THE CASES */
    case AVAIL_STATE:
    case ACK_STATE:
        Globals.clients_ready++;
        client_ptr->status = READY_STATE;
    case READY_STATE:
        acknowledge_receipt(client_ptr);
        break;
    case IN_USE_STATE:
        /******
        * ignore it since I'm probably looking for an ACK  *
        * and this is an extra READY from when the client  *
        * got impatient                                     *
        *****/
        break;
    default:
        fatal_err("Unexpected internal state!"); /* no return from here */
        break;
}
}
}

```

```

process_ACK(ECB * ecb)
{
    CLIENT_INFO * client_ptr;

    client_ptr = find_client_info((IPXHeader *)
        FP_OFF(ecb->fragmentDescriptor[0].address));
    if (!client_ptr)
    {
        nonfatal_err("Received an ACK from an unknown client!");
    }
    else
    {
        if(client_ptr->status == IN_USE_STATE)
        {
            client_ptr->status = ACK_STATE;
            client_ptr->ackSequenceNumber = ((DATA_STRUCT *)
                FP_OFF(ecb->fragmentDescriptor[1].address))->sequenceNumber;
        }
        else
        {
            fatal_err("Unexpected internal state!"); /* no return from here */
        }
    }
}

```

```
}
```

```
start_timer()
```

```
{
```

```
    Globals.seconds = 0;
```

```
    IPXScheduleSpecialEvent(18, &Globals.timerECB);
```

```
}
```

```
stop_timer()
```

```
{
```

```
    IPXCancelEvent(&Globals.timerECB);
```

```
}
```

Figure 6: TCLIENT.C source code.

```

/*****
* File:          tclient.c          *
* Program:       tclient.exe        *
* Purpose:       Act as a client to companion tserver.exe *
* Programmer:    Steve Shanafelt    *
*****/

#include <stdio.h>
#include <stdlib.h>
#include <dos.h>
#include <mem.h>
#include <string.h>
#include <nit.h>
#include <nxt.h>
#include <ipxdefs.h>
#include <mapcap.h>

#define MAX_WAIT_FOR_MASTER 10
extern void far timerESRHandler();

main(void);
fatal_err(char * messg);
setup_test_client(void);
find_test_server(void);
establish_connection(void);
process_cmds(void);
cleanup(void);
prep_sendECB(void);
prep_listenECB(void);
start_timer(void);
stop_timer(void);
int GetOtherSidesAddress(char * searchObjectName,
    WORD searchObjectType,
    IPXAddress * OtherGuysAddress);
void timerESR(ECB * ecb);
void IPXReceiveESR(ECB * ecb);
void IPXSendESR(ECB * ecb);
void AESESR(ECB * ecb);
void SendESR(ECB * ecb);
send_messg_to_master(char * messg, int sequenceNumber);
acknowledge_receipt(int sequenceNumber);

ECB timerECB,
    listenECB,
```

```

        sendECB;
IPXHeader IPXSend, IPXReceive;
DATA_STRUCT dataToReceive;
DATA_STRUCT dataToSend;
int seconds;
int timer_enabled;
int sequenceNumber = 1;

#define TESTSERVERNAME "TFSERVER"

main()
{
    timer_enabled = 1;
    setup_test_client();
    send_msg_to_master("AVAIL", sequenceNumber);
    process_cmds();
    cleanup();
}

fatal_err(char * messg)
{
    printf("%s\n", messg);
    cleanup();
    exit(1);
}

setup_test_client()
{
    WORD testServerSocket;
    int ccode;

    seconds = 0;
    IPXInitialize();
    memset(&IPXSend, '\0', sizeof(IPXHeader));
    timerECB.ESRAddress = timerESRHandler;
    testServerSocket = IntSwap(MASTER_SOCKET);
    testServerSocket = IntSwap(CLIENT_SOCKET);
    ccode = IPXOpenSocket((BYTE *)&testServerSocket, 0);
    if(ccode != 0 && ccode != IPX_SOCKET_ALREADY_OPEN)
    {
        fatal_err("IPXOpenSocket() failed.");
    }
    find_test_server();
    prep_sendECB();
    prep_listenECB();
}

```

```

find_test_server()
{
    start_timer();
    while (GetOtherSidesAddress(TESTSERVERNAME,
        MASTER_TYPE, (IPXAddress *)&IPXSend.destination) != 0)
    {
        IPXRelinquishControl();          /* do nothing while waiting */
        if (seconds > MAX_WAIT_FOR_MASTER)
        {
            stop_timer();
            printf("Client internal timeout waiting for Master\n");
            start_timer();
        }
    }
    stop_timer();
}

send_messg_to_master(char * messg, int sequenceNumber)
{
    int ACK_not_recvd = 1;
    DATA_STRUCT * data;

    /******
    * The protocol for sending packets will be:
    * 1. post a listen for an acknowledgement
    * 2. send the packet
    * 3. wait for an acknowledgement until timeout
    * 4. if timeout occurs goto 2
    *****/

    while(ACK_not_recvd)
    {
        IPXListenForPacket(&listenECB);
        strcpy(dataToSend.databuffer, messg);
        dataToSend.sequenceNumber = sequenceNumber;
        IPXSendPacket(&sendECB);
        while (sendECB.inUseFlag)
            IPXRelinquishControl();      /* wait for send to complete */
        if (sendECB.completionCode)
        {
            fatal_err("IPXSendPacket() failed.");
        }
        printf("sent '%s':%d\n", messg, sequenceNumber);
        start_timer();
        while (listenECB.inUseFlag)

```

```

{
    IPXRelinquishControl();          /* do nothing while waiting */
    if (seconds > MAX_WAIT_FOR_MASTER)
    {
        printf("timeout occurred in send_messg_to_master()\n");
        stop_timer();
        IPXSendPacket(&sendECB);
        while (sendECB.inUseFlag)
            IPXRelinquishControl();
        if (sendECB.completionCode)
        {
            fatal_err("IPXSendPacket() failed.");
        }
        printf("re-sent '%s':%d\n", messg, sequenceNumber);
        start_timer();
    }
}
stop_timer();
if (listenECB.completionCode)
{
    fatal_err("IPXListenForPacket() failed.");
}
data = (DATA_STRUCT *)FP_OFF(listenECB.fragmentDescriptor[1].address);
ACK_not_recvd = strcmp("ACK", data->databuffer) &&
    data->sequenceNumber == sequenceNumber;
}
printf("Received ACK:%d for '%s':%d.\n", data->sequenceNumber, messg,
    dataToSend.sequenceNumber);
}

```

```

process_cmds()
{
    int quit_not_recvd = 1;

    sequenceNumber++;
    if (sequenceNumber < 1)
        sequenceNumber = 1;
    while(quit_not_recvd)
    {
        send_messg_to_master("READY", sequenceNumber);
        /* post listen for command */
        IPXListenForPacket(&listenECB);
        /* wait for command */
        while(listenECB.inUseFlag)
        {
            IPXRelinquishControl();

```

```

    }
    if (listenECB.completionCode)
    {
        fatal_err("IPXListenForPacket() failed.");
    }
    else
    {
        /* command parsing and execution will take place here */
        printf("received message: %s:%d\n", dataToReceive.databuffer,
            dataToReceive.sequenceNumber);
        if (strcmp(dataToReceive.databuffer, "ACK"))
        {
            /* don't acknowledge ACKs from earlier repeated READYs */
            acknowledge_receipt(dataToReceive.sequenceNumber);
        }
        quit_not_recvd = strcmp("QUIT", dataToReceive.databuffer);
    }
}
}

```

```

acknowledge_receipt(int sequenceNumber)
{

```

```

    /******
    * The protocol for receiving packets will be:
    * 1. post a listen for a packet
    * 2. upon receipt of a packet, send an "ACK"nowledgement
    *
    * NOTE: the listen for this protocol gets posted in process_cmds().
    *****/

```

```

    strcpy(dataToSend.databuffer, "ACK");
    dataToSend.sequenceNumber = sequenceNumber;

```

```

    IPXSendPacket(&sendECB);
    while(sendECB.inUseFlag)
        IPXRelinquishControl();
    if(sendECB.completionCode)
    {
        fatal_err("IPXSendPacket() failed");
    }
    printf("sent 'ACK':%d\n", sequenceNumber);

```

```

}

```

```

cleanup()
{

```

```

    /* be nice and clean up */

```

```

    IPXCancelEvent(&listenECB);
    IPXCancelEvent(&sendECB);
    IPXCloseSocket(MASTER_SOCKET);
    IPXCloseSocket(CLIENT_SOCKET);
    if (timerECB.inUseFlag)
        stop_timer();
}

prep_sendECB()
{
    int                TransportTime = 0;

    /* get immediate address */
    if(IPXGetLocalTarget(IPXSend.destination.network, sendECB.immediateAddress,
        &TransportTime))
    {
        fatal_err("IPXGetLocalTarget() failed.");
    }

    /* now fill in ECB fields for send */
    IPXSend.packetType = 4;
    sendECB.ESRAddress = NULL;
    sendECB.socketNumber = IntSwap(MASTER_SOCKET);
    sendECB.fragmentCount = 2;
    sendECB.fragmentDescriptor[0].address = (char far *)&IPXSend;
    sendECB.fragmentDescriptor[0].size = sizeof(IPXHeader);
    sendECB.fragmentDescriptor[1].address = (char far *)&dataToSend;
    sendECB.fragmentDescriptor[1].size = sizeof(DATA_STRUCT);
}

prep_listenECB()
{
    /* get ready to receive a packet also */
    memset(&IPXReceive, '\0', sizeof(IPXHeader));
    /* fill out listen ECB */
    listenECB.ESRAddress = NULL;
    listenECB.socketNumber = IntSwap(CLIENT_SOCKET);
    listenECB.fragmentCount = 2;
    listenECB.fragmentDescriptor[0].address = (char far *)&IPXReceive;
    listenECB.fragmentDescriptor[0].size = sizeof(IPXHeader);
    listenECB.fragmentDescriptor[1].address = (char far *)&dataToReceive;
    listenECB.fragmentDescriptor[1].size = sizeof(DATA_STRUCT);
}

start_timer()
{

```



```

        seconds = 0;
        IPXScheduleSpecialEvent(18, &timerECB);
    }

stop_timer()
{
    int ccode;

    if ((ccode = IPXCancelEvent(&timerECB)) != 0)
    {
        printf("Unable to cancel timer.\n");
    }
}

int GetOtherSidesAddress(searchObjectName, searchObjectType, OtherGuysAddress)
char * searchObjectName;
WORD searchObjectType;
IPXAddress *OtherGuysAddress;
{
    int                ccode;
    WORD                objectType;
    char                objectName[SERVER_NAME_LEN], objectHasProperties;
    char                objectFlag, objectSecurity, propertyName[16];
    long                ObjectID=-1;
    int                segmentNumber=1;
    unsigned char        moreSegments, propertyFlags, Data[128];

/*-----
* Quoting from Novell API Reference, vol 2. 608/Rev 1.00
* Service Advertising Protocol: Chapter 4, pg 1.
*
* Each file server creates a bindery object for every known server on the
* internetwork. The object's name and type are equivalent to the value-added
* server's Server name and type. A property identified by the name
* 'NET_ADDRESS' is attached to the object. The first twelve bytes of this
* property's value are the network, node, socket address at which the VAS
* can be contacted.
*-----*/
    ccode = ScanBinderyObject(searchObjectName, searchObjectType, &ObjectID,
                             objectName, &objectType, &objectHasProperties, &objectFlag,
                             &objectSecurity);

    if (ccode)
    {
/*
        printf("Error 0x%x occurred calling ScanBinderyObject.\n", ccode);*/
        return(-1);
    }
}

```

```

/*****
** Get other guy's network number, physical node address, and socket.
*****/
strcpy(propertyName, "NET_ADDRESS");
ccode = ReadPropertyValue(objectName, searchObjectType, propertyName,
                          segmentNumber, Data, &moreSegments, &propertyFlags);

/*****
** Set other guy's network number, physical node address, and socket.
*****/
memcpy(OtherGuysAddress, Data, NET_ADDR_LEN);

return 0;
}

void timerESR(ECB *ecb)
{
    seconds++;
    IPXScheduleSpecialEvent(18, ecb);
}

/*****/
void IPXReceiveESR(ECB *ecb)
{
}

/*****/

void IPXSendESR(ECB *ecb)
{
}

/*****/

void AESESR(ECB *ecb)
{
}

/*****/

void SendESR(ECB *ecb)
{
}

/*****/

```

