```
//
//      OTVirtualClient         by Eric Okholm          Version 1.0
//
//      This is an OpenTransport sample client application which can be used
//      to exercise and test the OpenTransport Virtual Server sample.
//      It also demonstrates coding techniques for OT client applications.
//
//      You are welcome to use this code in any way to create you own
//      OpenTransport applications.   For more information on this program,
//      please review the document "About OTVirtual Server".
//
//      What's new in version 1.0.1:
//
//          - No changes, it just kept the version number parallel
//            to OTVirtualServer version 1.0.1.
//
//      Go Bears, beat Stanford !!!
//

#define DoAlert(x)          { sprintf(gProgramErr, x); gProgramState = kProgramError; }
#define DoAlert1(x, y)      { sprintf(gProgramErr, x, y); gProgramState = kProgramError; }
#define DoAlert2(x, y, z)   { sprintf(gProgramErr, x, y, z); gProgramState = kProgramError;}

//
//  Program mode
//
//  Before compiling,
//  set kDebugLevel to 0 for production
//                  or 1 for debug code.
//
//  In production mode, the code attempts to recover cleanly from any problems in encounters.
//  In debug mode, the unexplained phenomenon cause an alert box highlighting the situation
//  to be delivered and then the program exits.
//

#define kDebugLevel 1

#if kDebugLevel > 0

#define DBAlert(x)          DoAlert(x)
#define DBAlert1(x, y)      DoAlert1(x, y)
#define DBAlert2(x, y, z)   DoAlert2(x, y, z)

#else

#define DBAlert(x)          { }
#define DBAlert1(x, y)      { }
#define DBAlert2(x, y, z)   { }

#endif

//
//  Include files
//
#include <Dialogs.h>
#include <Events.h>
#include <Fonts.h>
#include <GestaltEqu.h>
#include <Memory.h>
#include <Menus.h>
#include <QuickDraw.h>
#include <SegLoad.h>
#include <stdio.h>
#include <StdLib.h>
#include <String.h>
#include <strings.h>
#include <ToolUtils.h>
```

```
#include <Windows.h>

#include <OpenTptInternet.h>        // includes OpenTransport.h
#include <OpenTptClient.h>          // needed for OTReleaseBuffer()

//
//  Defines, enums, resource IDs
//
#define kInFront                    (WindowPtr) -1
#define kWindowResID                128

    // Apple Menu
#define kAppleMenuResID             128
#define kAppleMenuAbout             1

    // File Menu
#define kFileMenuResID              129
#define kFileMenuOpen               1
#define kFileMenuClose              2
#define kFileMenuQuit               4

    // Edit Menu
#define kEditMenuResID              130             // Edit menu is disabled

    // Client Menu
#define kClientMenuResID            131
#define kClientMenuTCPPrefs         1

    // Alerts, etc.
#define kAlertExitResID             128
#define kAboutBoxResID              130

    // TCP Prefs Dialog
#define kTCPPrefsDlogResID          129
#define kServerAddrDItem            2
#define kServerPortDItem            4
#define kMaxConnectionsDItem        6
#define kStartStopDItem             7

    // Overall program states
enum
{
    kProgramRunning     = 0,
    kProgramDone        = 1,
    kProgramError       = 2
};

    // Server states
enum
{
    kClientStopped      = 0,
    kClientRunning      = 1,
    kClientShuttingDown = 2
};

    // Bit numbers in EPInfo stateFlags fields
enum
{
    kOpenInProgressBit              = 0
};

    // Misc stuff
enum
{
    kTimerIntervalInSeconds     = 3,
    kTimerInterval              = (kTimerIntervalInSeconds * 1000),
    kServerRequestSize          = 128,
    kOTVersion111              = 0x01110000
};

    // Endpoint Info Structure
struct EPInfo
{
    EndpointRef     erf;                //  actual endpoint
    EPInfo*         next;               //  used to link all acceptor's EPInfos (not atomic)
    OTLink          link;               //  link into an OT LIFO (atomic)
    UInt8           stateFlags;         //  various status fields
};
typedef struct EPInfo EPInfo;

//
//  Globals
//
EPInfo*             gDNS                         = NULL;
EPInfo*             gConnectors                  = NULL;
InetHostInfo        gServerHostInfo;
Boolean             gWaitForServerAddr;
int                 gClientState                 = kClientStopped;
int                 gProgramState                = kProgramRunning;
char                gProgramErr[128];
DialogPtr           gDialogPtr                   = NULL;
WindowPtr           gWindowPtr                   = NULL;
long                gSleepTicks                  = 60;
Str255              gServerAddrStr               = "\p17.202.32.195";
long                gServerAddr                  = 0;
Str255              gServerPortStr               = "\p2001";
long                gServerPort                  = 0;
Str255              gMaxConnectionsStr           = "\p100";
long                gMaxConnections              = 0;
Boolean             gClientRunning               = false;
Str255              gStartStr                    = "\pStart";
Str255              gStopStr                     = "\pStop";
SInt32              gCntrEndpts                  = 0;
SInt32              gCntrIdleEPs                 = 0;
SInt32              gCntrBrokenEPs               = 0;
SInt32              gCntrPending                 = 0;
SInt32              gCntrConnections             = 0;
SInt32              gCntrTotalConnections        = 0;
SInt32              gCntrBytesRcvd               = 0;
SInt32              gCntrTotalBytesRcvd          = 0;
SInt32              gCntrDiscon                  = 0;
OTLIFO              gIdleEPLIFO;
OTLIFO*             gIdleEPs                     = &gIdleEPLIFO;
OTLIFO              gBrokenEPLIFO;
OTLIFO*             gBrokenEPs                   = &gBrokenEPLIFO;
OTConfiguration*    gCfgMaster                   = NULL;
Boolean             gWaitForEventLoop            = false;
long                gTimerTask                   = 0;
SInt32              gCntrIntervalConnects        = 0;
SInt32              gCntrIntervalBytes           = 0;
SInt32              gConnectsPerSecond           = 0;
SInt32              gConnectsPerSecondMax        = 0;
SInt32              gKBytesPerSecond             = 0;
SInt32              gKBytesPerSecondMax          = 0;
SInt32              gCntrIntervalEventLoop       = 0;
SInt32              gEventsPerSecond             = 1;
SInt32              gEventsPerSecondMax          = 1;
Boolean             gDoWindowUpdate              = true;
unsigned char       gServerRequest[kServerRequestSize];
OSType              gOTVersionSelector           = 'otvr';
UInt32              gOTVersion;

//
//  OpenTransport Networking Code Prototypes
```

```
//
static void         DoConnect(EPInfo*);
static Boolean      EPClose(EPInfo*);
static Boolean      EPOpen(EPInfo*);
static void         NetEventLoop(void);
static void         NetInit(void);
static void         NetShutdown(void);
static pascal void  Notifier(void*, OTEventCode, OTResult, void*);
static void         ReadData(EPInfo*);
static void         Recycle(void);
static void         SendRequest(EPInfo*);
static void         StartClient(void);
static void         StopClient(void);
static void         TimerInit();
static void         TimerDestroy();
static pascal void  TimerRun(void*);

//
//  Macintosh Program Wrapper Prototypes
//
static void         AboutBox(void);
static void         AlertExit(Str255);
static void         DialogClose(void);
static Boolean      EventDialog(EventRecord*);
static void         EventDrag(WindowPtr, Point);
static void         EventGoAway(WindowPtr, Point);
static void         EventKeyDown(EventRecord*);
static void         EventLoop(void);
static void         EventMouseDown(EventRecord*);
static void         MacInit(void);
static void         MacInitROM(void);
static void         C2PStr(char*, Str255);
static void         P2CStr(Str255, char*);
static void         MenuDispatch(long);
static void         SetupMenus(void);
static void         TCPPrefsDialog(void);
static void         TCPPrefsReset(void);
static void         WindowClose(void);
static void         WindowOpen(void);
static void         WindowUpdate(void);


//////////////////////////////////////////////////////////////////////////////////////
//
//  OpenTransport Networking Code
//
//  The code in this section provides the networking portions of the
//  OpenTransport Virtual Client.
//
//////////////////////////////////////////////////////////////////////////////////////

//
//  DoBind
//
//  This routine requests a wildcard port binding from the transport protocol.
//  Since the program doesn't care what port is returned, it passes in NULL
//  for the bind return parameter.  The bind request structure is ephemeral
//  and can be a local stack variable since OT is done with it when the call returns.
//  The bind is done when the notifier receives a T_BINDCOMPLETE event.
//
static void DoBind(EPInfo* epi)
{
    OSStatus err;
    TBind bindReq;
    InetAddress inAddr;

    //
    //  Bind the endpoint to a wildcard address
    //  (assign us a port, we don't care which one).
    //
    OTInitInetAddress(&inAddr, 0, 0);
    bindReq.addr.len = sizeof(InetAddress);
    bindReq.addr.buf = (unsigned char*) &inAddr;
    bindReq.qlen = 0;

    err = OTBind(epi->erf, &bindReq, NULL);
    if (err != kOTNoError)
    {
        DBAlert1("DoBind: OTBind error %d", err);
        return;
    }
}

//
//  DoConnect
//
//  This routine attempts establish a new connection to the globally known
//  server address and port.   If the program is still trying to use the
//  DNR to resolve the server's host name into an IP address, the endpoint
//  is queued for later connection.
//
static void DoConnect(EPInfo* epi)
{
    OSStatus err;
    TCall sndCall;
    InetAddress inAddr;
    OTLink* link;

    //  Don't want new connections if already shutting down.
    if (gProgramState != kProgramRunning || gClientState != kClientRunning)
        return;

    if (gWaitForServerAddr || gWaitForEventLoop)
    {
        if (epi != NULL)
        {
            OTLIFOEnqueue(gIdleEPs, &epi->link);
            OTAtomicAdd32(1, &gCntrIdleEPs);
        }
        return;
    }

    //  If we weren't passed a specific EPInfo, try to get an idle one.
    if (epi == NULL)
    {
        link = OTLIFODequeue(gIdleEPs);
        if (link == NULL)
            return;
        OTAtomicAdd32(-1, &gCntrIdleEPs);
        epi = OTGetLinkObject(link, EPInfo, link);
    }

    OTInitInetAddress(&inAddr, gServerPort, gServerAddr);
    OTMemzero(&sndCall, sizeof(TCall));
    sndCall.addr.len    = sizeof(InetAddress);
    sndCall.addr.buf    = (unsigned char*) &inAddr;

    OTAtomicAdd32(1, &gCntrPending);
    err = OTConnect(epi->erf, &sndCall, NULL);
    if (err != kOTNoDataErr)
    {
        OTAtomicAdd32(-1, &gCntrPending);
        DBAlert2("DoConnect: OTConnect error %d state %d", err, OTGetEndpointState(epi->erf));
        return;
```

```
    }

    //
    //  If OTConnect didn't return an error, this thread will
    //  resume when the notifier gets a T_CONNECT event...
    //
}

//
//  EPClose
//
//  This routine is a front end to OTCloseProvider.
//  Centralizing closing of endpoints makes debugging and instrumentation easier.
//
static Boolean EPClose(EPInfo* epi)
{
    OSStatus err;

    //
    //  If an endpoint is still being opened, we can't close it yet.
    //  There is no way to cancel an OTAsyncOpenEndpoint, so we just
    //  have to wait for the T_OPENCOMPLETE event at the notifier.
    //
    if ( OTAtomicTestBit(&epi->stateFlags, kOpenInProgressBit) )
        return false;

    err = OTCloseProvider(epi->erf);
    epi->erf = NULL;
    if (err)
        DBAlert1("EPClose: OTCloseProvider error %d", err);

    if (epi != gDNS)
        OTAtomicAdd32(-1, &gCntrEndpts);
    return true;

}

//
//  EPOpen:
//
//  A front end to OTAsyncOpenEndpoint.
//  A status bit is set so we know there is an open in progress.
//  It is cleared when the notifier gets a T_OPENCOMPLETE where the context
//  pointer is this EPInfo.  Until that happens, this EPInfo can't be cleaned
//  up and released.
//
static Boolean EPOpen(EPInfo* epi, OTConfiguration* cfg)
{
    OSStatus err;

    OTAtomicSetBit(&epi->stateFlags, kOpenInProgressBit);
    err = OTAsyncOpenEndpoint(cfg, 0, NULL, &Notifier, epi);
    if (err != kOTNoError)
    {
        OTAtomicClearBit(&epi->stateFlags, kOpenInProgressBit);
        DBAlert1("EPOpen: OTAsyncOpenEndpoint error %d", err);
        return false;
    }
    return true;
}

//
//  NetEventLoop
//
//  This routine is called once during each pass through the program's event loop.
//  If the program is running on OT 1.1.2 or an earlier release, this is where
//  outbound orderly releases are started (see comments in DoSndOrderlyRelease
//  for more information on that).   This is also where endpoints are "fixed" by
//  closing them and opening a new one to replace them.   This is rarely necessary,
//  but works around some timing issues in OTUnbind().  Having passed through the
//  event loop once, we assume it is safe to turn off throttle-back.  And, finally,
//  if we have deferred handing of a T_LISTEN, here we start it up again.
//
static void NetEventLoop()
{
    Recycle();
    gWaitForEventLoop = false;
    DoConnect(NULL);
}

//
//  NetInit:
//
//  This is nothing but a front end to InitOpenTransport.
//  The only reason for having this routine is to get the call to InitOpenTransport
//  up into the "networking" section of the program and out of the
//  "macintosh program wrapper" section of the program.
//
static void NetInit()
{
    OSStatus err;

    err = InitOpenTransport();
    if (err)
    {
        DBAlert1("NetInit: InitOpenTransport error %d", err);
        return;
    }
    err = Gestalt(gOTVersionSelector, (long*) &gOTVersion);
    if (err || (gOTVersion < kOTVersion111))
    {
        DoAlert("Please install Open Transport 1.1.1 or later");
        return;
    }
    TimerInit();
}

//
//  NetShutdown:
//
//  Ditto...
//
static void NetShutdown()
{
    TimerDestroy();
    CloseOpenTransport();
}


//
//  Notifier:
//
//  Most of the interesting networking code in this program resides inside
//  this notifier.   In order to run asynchronously and as fast as possible,
//  things are done inside the notifier whenever possible.  Since almost
//  everything is done inside the notifier, there was little need for specical
//  synchronization code.
//
//  Note: The DNR events are combined with normal endpoint events in this notifier.
//  The only events which are expected from the DNR are T_DNRSTRINGTOADDRCOMPLETE
//  and T_OPENCOMPLETE.
//
//  IMPORTANT NOTE:  Normal events defined by XTI (T_LISTEN, T_CONNECT, etc)
//  and OT completion events (T_OPENCOMPLETE, T_BINDCOMPLETE, etc.) are not
```

```
//  reentrant.  That is, whenever our notifier is invoked with such an event,
//  the notifier will not be called again by OT for another normal or completion
//  event until we have returned out of the notifier - even if we make OT calls
//  from inside the notifier.   This is a useful synchronization tool.
//  However, there are two kinds of events which will cause the notifier to
//  be reentered.   One is T_MEMORYRELEASED, which always happens instantly.
//  The other are state change events like kOTProviderWillClose.
//

static pascal void Notifier(void* context, OTEventCode event, OTResult result, void* cookie)
{
    OSStatus err;
    OTResult epState;
    EPInfo* epi = (EPInfo*) context;

    //
    //  Once the program is shutting down, most events would be uninteresting.
    //  However, we still need T_OPENCOMPLETE and T_MEMORYRELEASED events since
    //  we can't call CloseOpenTransport until all OTAsyncOpenEndpoints and
    //  OTSends with AckSends have completed.   So those specific events
    //  are still accepted.
    //
    if (gProgramState != kProgramRunning)
    {
        if (event != T_OPENCOMPLETE)
            return;
    }

    //
    //  This really isn't necessary, it's just a sanity check which should be removed
    //  once a program is debugged.   It's just making sure we don't get event notifications
    //  after all of our endpoints have been closed.
    //
    if (gClientState == kClientStopped)
    {
        DBAlert1("Notifier: got event %d when client not running!", event);
        return;
    }

    //
    //  Within the notifier, all action is based on the event code.
    //  In this notifier, fatal errors all break out of the switch to the bottom.
    //  As long as everything goes as expected, the case returns rather than breaks.
    //
    switch (event)
    {
        //
        //  T_BINDCOMPLETE:
        //
        //  This event is returned when an endpoint has been bound to a wildcard addr.
        //  No errors are expected.   The program immediately attempts to establish
        //  a connection from this endpoint to the server.
        //
        case T_BINDCOMPLETE:
        {
            if (result != kOTNoError)
            {
                DBAlert1("Notifier: T_BINDCOMPLETE result %d", result);
                return;
            }
            DoConnect(epi);         // resumes at T_CONNECT
            return;
        }

        //
        //  T_CONNECT:
        //
//  This event is returned when a connection is established to the server.
//  The program must call OTRcvConnect() to get the conenction information
//  and clear the T_CONNECT event from the stream.  Since OTRcvConnect()
//  returns immediately (rather than via a completion event to the notifier)
//  we can use local stack structures for parameters.
//
case T_CONNECT:
{
    TCall call;
    InetAddress caddr;

    if (result != kOTNoError)
    {
        DBAlert1("Notifier: T_CONNECT result %d", result);
        return;
    }

    call.addr.maxlen = sizeof(InetAddress);
    call.addr.buf = (unsigned char*) &caddr;
    call.opt.maxlen = 0;
    call.opt.buf = NULL;
    call.udata.maxlen = 0;
    call.udata.buf = NULL;

    err = OTRcvConnect(epi->erf, &call);
    if (err != kOTNoError)
    {
        DBAlert1("Notifier: T_CONNECT - OTRcvConnect err %d", err);
        return;
    }
    OTAtomicAdd32(-1, &gCntrPending);
    OTAtomicAdd32(1, &gCntrConnections);
    OTAtomicAdd32(1, &gCntrTotalConnections);
    OTAtomicAdd32(1, &gCntrIntervalConnects);

    SendRequest(epi);

    //
    //  Since we won't be sending any data,
    //  might as well send along the orderly release now.
    //
    err = OTSndOrderlyDisconnect(epi->erf);
    if (err != kOTNoError)
    {
        if (err != kOTLookErr)
        {
        DBAlert1("Notifier: T_CONNECT: OTSndOrderlyDisconnect error %d", err);
        }
    }
    return;             // Wait for a T_DATA...
}

//
//  T_DATA:
//
//  The main rule for processing T_DATA's is to remember that once you have
        //  a T_DATA, you won't get another one until you have read to a kOTNoDataErr.
        //  The advanced rule is to remember that you could get another T_DATA
        //  during an OTRcv() which will eventually return kOTNoDataErr, presenting
        //  the application with a synchronization issue to be most careful about.
        //
        //  In this application, since an OTRcv() calls are made from inside the notifier,
        //  this particular synchronization issue doesn't become a problem.
        //
        case T_DATA:
        {
        ReadData(epi);
```

```
            return;
        }

        //
        //  T_DNRSTRINGTOADDRCOMPLETE:
        //
        //  This event occurs when the DNR has finished an attempt to translate
        //  the server's name into an IP address we can use to connect to.
        //
        case T_DNRSTRINGTOADDRCOMPLETE:
        {
            if (result != kOTNoError)
            {
                DBAlert1("Notifier: T_DNRSTRINGTOADDRCOMPLETE result %d", result);
                return;
            }
            gServerAddr = gServerHostInfo.addrs[0];
            gWaitForServerAddr = false;
            return;
        }

        //
        //  T_DISCONNECT:
        //
        //  An inbound T_DISCONNECT event usually indicates that the other side of the
        //  connection did an abortive disconnect (as opposed to an orderly release).
        //  It also can be generated by the transport provider on the system (e.g. tcp)
        //  when it decides that a connection is no longer in existance.
        //
        //  We receive the disconnect, but this program ignores the associated reason (NULL para
        //  It is possible to get back a kOTNoDisconnectErr from the OTRcvDisconnect call.
        //  This can happen when either (1) the disconnect on the stream is hidden by a
        //  higher priority message, or (2) something has flushed or reset the disconnect
        //  event in the meantime.   This is not fatal, and the appropriate thing to do is
        //  to pretend the T_DISCONNECT event never happened.   Any other error is unexpected
        //  and needs to be reported so we can fix it.  Next, unbind the endpoint so we can
        //  reuse it for a new inbound connection.
        //
        //  It is possible to get an error on the unbind due to a bug in OT 1.1.1 and earlier.
        //  The best thing to do for that is close the endpoint and open a new one to replace it
        //  We do this back in the main thread so we don't have to deal with synchronization pro
        //
        case T_DISCONNECT:
        {
            epState = OTGetEndpointState(epi->erf);
            if (epState == T_OUTCON)
            {
                OTAtomicAdd32(-1, &gCntrPending);
            }
            OTAtomicAdd32(1, &gCntrDiscon);
            err = OTRcvDisconnect(epi->erf, NULL);
            if (err != kOTNoError)
            {
                if (err == kOTNoDisconnectErr)
                    return;
                DBAlert1("Notifier: T_DISCONNECT - OTRcvDisconnect error %d", err);
                return;
            }

            err = OTUnbind(epi->erf);
            if (err != kOTNoError)
            {
                OTLIFOEnqueue(gBrokenEPs, &epi->link);
                OTAtomicAdd32(1, &gCntrBrokenEPs);
            }
            return;
        }
```

```
//
//  T_GODATA:
//
//  Because of the complexity involved in the implementation of OT flow control,
//  it is sometimes possible to receive a T_GODATA even when we aren't subject
//  to flow control - normally only at the start of a program.   If this happens,
//  ignoring it is the correct thing to do.
//
case T_GODATA:
{
    return;
}

//
//  T_OPENCOMPLETE:
//
//  This event occurs when an OTAsyncOpenEndpoint() completes.   Note that this event,
//  just like any other async call made from outside the notifier, can occur during
//  the call to OTAsyncOpenEndpoint().  That is, in the main thread the program did
//  the OTAsyncOpenEndpoint(), and the notifier is invoked before control is returned
//  to the line of code following the call to OTAsyncOpenEndpoint().   This is one
//  event we need to keep track of even if we are shutting down the program since there
//  is no way to cancel outstanding OTAsyncOpenEndpoint() calls.
//
case T_OPENCOMPLETE:
{
    char serverCString[256];

    OTAtomicClearBit(&epi->stateFlags, kOpenInProgressBit);
    if (result == kOTNoError)
        epi->erf = (EndpointRef) cookie;
    else
    {
        DBAlert1("Notifier: T_OPENCOMPLETE result %d", result);
        return;
    }

    if (gProgramState != kProgramRunning)
        return;

    if (epi == gDNS)
    {
        P2CStr(gServerAddrStr, serverCString);
        err = OTInetStringToAddress((InetSvcRef)epi->erf, serverCString, &gServerHostInf
        if (err != kOTNoError)
        {
            //
            //  Can't translate the server address string
            //
            DBAlert1("Notifier: T_OPENCOMPLETE - OTInetStringToAddress error %d", err);
        }
        return;     // DNS resumes at T_DNRSTRINGTOADDRCOMPLETE
    }
    else
    {
        OTAtomicAdd32(1, &gCntrEndpts);

        //
        //  Set to blocking mode so we don't have to deal with kEAGAIN errors.
        //  Async/blocking is the best mode to write an OpenTransport application in.
        //
        err = OTSetBlocking(epi->erf);
        if (err != kOTNoError)
        {
            DBAlert1("Notifier: T_OPENCOMPLETE - OTSetBlocking error %d", err);
            return;
```

```
        }

        DoBind(epi);
        return;         // resumes at T_BINDCOMPLETE
    }
}

//
//  T_ORDREL:
//
//  This event occurs when an orderly release has been received on the stream.
//
case T_ORDREL:
{
    err = OTRcvOrderlyDisconnect(epi->erf);
    if (err != kOTNoError)
    {
        DBAlert1("Notifier: T_ORDREL - OTRcvOrderlyDisconnect error %d", err);
        return;
    }
    epState = OTGetEndpointState(epi->erf);
    if (epState != T_IDLE)
        return;
    OTAtomicAdd32(-1, &gCntrConnections);
    err = OTUnbind(epi->erf);
    if (err != kOTNoError)
    {
        OTLIFOEnqueue(gBrokenEPs, &epi->link);
        OTAtomicAdd32(1, &gCntrBrokenEPs);
    }
    return;
}

//
//  T_UNBINDCOMPLETE:
//
//  This event occurs on completion of an OTUnbind().
//  The endpoint is ready for reuse on a new inbound connection.
//  Put it back into the queue of idle endpoints.
//  Note that the OTLIFO structure has atomic queue and dequeue,
//  which can be helpful for synchronization protection.
//
case T_UNBINDCOMPLETE:
{
    if (result != kOTNoError)
    {
        //
        //  Unbind errors can occur as a result of a bug in OT 1.1.1 and earlier
        //  versions.   The best recovery is to put the endpoint in the broken
        //  list for recycling with a clean, new endpoint.
        //
        OTLIFOEnqueue(gBrokenEPs, &epi->link);
        OTAtomicAdd32(1, &gCntrBrokenEPs);
        return;
    }
    DoBind(epi);
    return;
}


//
//  default:
//
//  There are events which we don't handle, but we don't expect to see
//  any of them.   When running in debugging mode while developing a program,
//  we exit with an informational alert.   Later, in the production version
//  of the program, we ignore the event and try to keep running.
```

```
        //
        default:
        {
            DBAlert1("Notifier: unknown event <%x>", event);
            return;
        }
    }
}

//
//  ReadData:
//
//  This routine attempts to read all available data from an endpoint.
//  Since this routine is only called from inside the notifier in the current
//  version of OTVirtualClient, it is not necessary to program to handle
//  getting back a T_DATA notification DURING an OTRcv() call, as would be
//  the case if we read from outside the notifier.   We must read until we
//  get a kOTNoDataErr in order to clear the T_DATA event so we will get
//  another notification of T_DATA in the future.
//
//  Currently this application uses no-copy receives to get data.  This obligates
//  the program to return the buffers to OT asap.  Since this program does nothing
//  with data other than count it, that's easy.  Future, more complex versions
//  of this program will do more interesting things with regards to that.
//
static void ReadData(EPInfo* epi)
{
    OTBuffer* bp;
    OTResult  res;
    OTFlags   flags;

    while (true)
    {
        res = OTRcv(epi->erf, &bp, kOTNetbufDataIsOTBufferStar, &flags);

        if (res > 0)
        {
            OTAtomicAdd32(res, &gCntrBytesRcvd);
            OTAtomicAdd32(res, &gCntrTotalBytesRcvd);
            OTAtomicAdd32(res, &gCntrIntervalBytes);
            OTReleaseBuffer(bp);
            continue;
        }
        if (res == kOTNoDataErr)
        {
            //
            //  Since ReadData is only called from inside the notifier
            //  we don't have to worry about having missed a T_DATA
            //  during the OTRcv.
            //
            return;
        }
        if (res <= 0)
        {
            if (res == kOTLookErr)
            {
                res = OTLook(epi->erf);
                if (res == T_ORDREL)
                    return;
                if (res == T_GODATA)
                {
                    //
                    //  This isn't expected, but it has happened occasionally.
                    //  The correct way to proceed is to ignore it.
                    //
                    continue;
                }
```

```
                else
                {
                    DBAlert1("ReadData: OTRcv got OTLookErr 0x%08x", res);
                }
            }
            else
            {
                DBAlert1("ReadData: OTRcv error %d", res);
            }
        }
    }
}

//
//  Recycle:
//
//  This routine shouldn't be necessary, but it is helpful to work around both
//  problems in OpenTransport and bugs in this program.   Basicly, whenever an
//  unexpected error occurs which shouldn't be fatal to the program, the EPInfo
//  is queued on the BrokenEP queue.  When recycle is called, once per pass around
//  the event loop, it will attempt to close the associated endpoint and open
//  a new one to replace it using the same EPInfo structure.   This process of
//  closing an errant endpoint and opening a replacement is probably the most
//  reliable way to make sure that this program and OpenTransport can recover
//  from unexpected happenings in a clean manner.
//
static void Recycle()
{
    OTLink*     list = OTLIFOStealList(gBrokenEPs);
    OTLink*     link;
    EPInfo*     epi;

    while ( (link = list) != NULL )
    {
        list = link->fNext;
        epi = OTGetLinkObject(link, EPInfo, link);
        if (!EPClose(epi))
        {
            OTLIFOEnqueue(gBrokenEPs, &epi->link);
            continue;
        }
        OTAtomicAdd32(-1, &gCntrBrokenEPs);
        EPOpen(epi, OTCloneConfiguration(gCfgMaster));
    }
}

//
//  SendRequest:
//
//  Tell the OT Virtual Server we want it to send us some data.
//  For demonstration purposes, the server will wait for a 128 byte
//  "request" to come in before sending us data.   It doesn't care
//  what the request looks like, it just allows us to better simulate
//  true client/server interactions.
//
static void SendRequest(EPInfo* epi)
{
    OTResult res;

    res = OTSnd(epi->erf, gServerRequest, kServerRequestSize, 0);

    //
    //  This is bogus and needs to add flow control.
    //  The only reason we get away with it here is because flow control
    //  will never happen in the first 128 bytes sent, and that is all
    //  we are sending.
    //
    if (res != kServerRequestSize)
    {
        DBAlert1("SendRequest: got result %d", res);
    }
    OTAtomicAdd32(res, &gCntrIntervalBytes);
}


//
//  StartClient:
//
//  Open one InetServices (DNS) object,
//  and as many connection endpoints as the program will use.
//  Start making connections as soon as the server's name is translated
//  to an IP address.
//
static void StartClient()
{
    int i;
    EPInfo* epi;
    OSStatus err;

    gCntrEndpts             = 0;
    gCntrPending            = 0;
    gCntrConnections        = 0;
    gCntrBrokenEPs              = 0;
    gCntrTotalConnections   = 0;
    gIdleEPs->fHead         = NULL;
    gBrokenEPs->fHead       = NULL;
    gClientState            = kClientRunning;
    TCPPrefsReset();
    gWaitForServerAddr      = true;

    //
    //  Open an InternetServices so we have access to the DNR
    //  to translate the server's name into an IP address (if necessary).
    //
    gDNS = (EPInfo*) NewPtr(sizeof(EPInfo));
    if (gDNS == NULL)
    {
        DBAlert("StartClient: NewPtr cannot get memory for EPInfo");
        return;
    }
    OTMemzero(gDNS, sizeof(EPInfo));
    OTAtomicSetBit(&gDNS->stateFlags, kOpenInProgressBit);
    err = OTAsyncOpenInternetServices(kDefaultInternetServicesPath, 0, Notifier, gDNS);
    if (err != kOTNoError)
    {
        OTAtomicClearBit(&gDNS->stateFlags, kOpenInProgressBit);
        DBAlert1("OTAsyncOpenInternetServices error %d", err);
        return;
    }

    //
    //  Get memory for EPInfo structures
    //
    for (i = 0; i < gMaxConnections; i++)
    {
        epi = (EPInfo*) NewPtr(sizeof(EPInfo));
        if (epi == NULL)
        {
            DBAlert("StartClient: NewPtr cannot get memory for EPInfo");
            return;
        }
        OTMemzero(epi, sizeof(EPInfo));
        epi->next = gConnectors;
        gConnectors = epi;
```

```
    }

    //
    //  Open endpoints which can be used for outbound
    //  connections to the server.
    //
    gCfgMaster = OTCreateConfiguration("tcp");
    if (gCfgMaster == NULL)
    {
        DBAlert("StartClient: OTCreateConfiguration returned NULL");
        return;
    }
    for (epi = gConnectors; epi != NULL; epi = epi->next)
    {
        if (!EPOpen(epi, OTCloneConfiguration(gCfgMaster)))
            break;
    }
}

//
//  StopClient:
//
//  This is where the client is shut down, either because the user clicked
//  the stop button, or because the program is exiting (error or quit).
//  The tricky part is that we can't quit while there are outstanding
//  OTAsyncOpenEndpoint calls (which can't be cancelled, by the way).
//
static void StopClient()
{
    EPInfo *epi, *last;

    gClientState = kClientShuttingDown;

    //
    //  First, make sure the DNS is closed.
    //
    if (gDNS != NULL)
    {
        if (!EPClose(gDNS))
            return;
        DisposPtr((char*)gDNS);
        gDNS = NULL;
    }

    //
    //  Start closing connector endpoints.
    //  While we could be rude and just close the endpoints,
    //  we try to be polite and wait for all outstanding connections
    //  to finish before closing the endpoints.   The is a bit easier
    //  on the server which won't end up keeping around control blocks
    //  for dead connections which it doesn't know are dead.  Alternately,
    //  we could just send a disconnect, but this seems cleaner.
    //
    epi = gConnectors;
    last = NULL;
    while (epi != NULL)
    {
        if (!EPClose(epi))
        {
            //  Can't close this endpoint yet, so skip it.
            last = epi;
            epi = epi->next;
            continue;
        }
        else
        {
            if (last != NULL)
            {
                last->next = epi->next;
                DisposPtr((char*)epi);
                epi = last->next;
            }
            else
            {
                gConnectors = epi->next;
                DisposPtr((char*)epi);
                epi = gConnectors;
            }
        }
    }

    //
    //  If the list is empty now, then all endpoints have been successfully closed,
    //  so the client is stopped now.   At this point we can either restart it or
    //  exit the program safely.
    //
    if (gConnectors == NULL)
    {
        gClientState            = kClientStopped;
        gCntrEndpts             = 0;
        gCntrIdleEPs                = 0;
        gCntrPending            = 0;
        gCntrConnections        = 0;
        gCntrBrokenEPs              = 0;
        gCntrTotalConnections   = 0;
        gIdleEPs->fHead         = NULL;
        gBrokenEPs->fHead       = NULL;
        OTDestroyConfiguration(gCfgMaster);
    }
}

//
//  TimerInit
//
//  Start up a regular timer to do housekeeping.   Strictly speaking,
//  this isn't necessary, but having a regular heartbeat allows us to
//  detect if we are so busy with network notifier processing that the
//  program's event loop isn't ever firing.   We want to know this so
//  we can at least allow the user to quit the program if they want to.
//
static void TimerInit()
{
    gTimerTask = OTCreateTimerTask(&TimerRun, 0);
    if (gTimerTask == 0)
    {
        sprintf(gProgramErr, "TimerInit: OTCreateTimerTask returned 0");
        gProgramState = kProgramError;
        return;
    }
    OTScheduleTimerTask(gTimerTask, kTimerInterval);
}

//
//  TimerDestroy
//
static void TimerDestroy()
{
    if (gTimerTask != 0)
    {
        OTCancelTimerTask(gTimerTask);
        OTDestroyTimerTask(gTimerTask);
        gTimerTask = 0;
    }
}
```

```
//
//  TimerRun
//
//  Fires every N seconds, no matter how busy the system is.
//  We use this to detect if the program's main event loop is getting no time,
//  in which case we can slow the client down by doing a throttle-back until
//  the event loop can run at least once.  It also is a convenient statistics
//  gathering point.
//
static pascal void TimerRun(void*)
{
    gConnectsPerSecond = (gCntrIntervalConnects / kTimerIntervalInSeconds);
    gKBytesPerSecond = (gCntrIntervalBytes / (kTimerIntervalInSeconds * 1024));
    gEventsPerSecond = (gCntrIntervalEventLoop / kTimerIntervalInSeconds);
    if (gCntrIntervalEventLoop == 0)
        gWaitForEventLoop = true;

    if (gConnectsPerSecond > gConnectsPerSecondMax)
        gConnectsPerSecondMax = gConnectsPerSecond;
    if (gKBytesPerSecond > gKBytesPerSecondMax)
        gKBytesPerSecondMax = gKBytesPerSecond;
    if (gEventsPerSecond > gEventsPerSecondMax)
        gEventsPerSecondMax = gEventsPerSecond;

    gCntrIntervalConnects   = 0;
    gCntrIntervalBytes      = 0;
    gCntrIntervalEventLoop  = 0;
    gDoWindowUpdate         = true;
    gCntrConnections        = gCntrEndpts - gCntrPending - gCntrBrokenEPs;

    OTScheduleTimerTask(gTimerTask, kTimerInterval);
}



////////////////////////////////////////////////////////////////////////////////////////
//
//  Macintosh Program Wrapper
//
//  The code from here down deals with the Macintosh environment, events,
//  menus, command keys, etc.   Networking code is in the section above.
//  Since this code is fairly basic, and since this isn't really intended
//  to be a "sample Macintosh application" (just a sample OpenTransport application)
//  this section isn't heavily commented.   There are much better Macintosh
//  application samples for handling mouse, keyboard, event loops, etc.
//
////////////////////////////////////////////////////////////////////////////////////////

static void AboutBox()
{
    Alert(kAboutBoxResID, NULL);
}

static Boolean EventDialog(EventRecord* event)
{
    DialogPtr   dp;
    short       item;
    short       itemType;
    Handle      itemHandle;
    Rect        itemRect;

    if (event->modifiers & cmdKey)
    {
        EventKeyDown(event);        // this allows menu commands while dialog is active window
        return false;               // note if I add cut/paste I will have to rework this.
    }
    if ((DialogSelect(event, &dp, &item)) && (dp == gDialogPtr))
    {
        GetDItem(gDialogPtr, item, &itemType, &itemHandle, &itemRect);
        switch (item)
        {
            case kServerAddrDItem:
                GetIText(itemHandle, gServerAddrStr);
                return true;

            case kServerPortDItem:
                GetIText(itemHandle, gServerPortStr);
                return true;

            case kMaxConnectionsDItem:
                GetIText(itemHandle, gMaxConnectionsStr);
                return true;

            case kStartStopDItem:
                GetDItem(gDialogPtr, kStartStopDItem, &itemType, &itemHandle, &itemRect);
                if (gClientRunning)
                {
                    StopClient();
                    SetCTitle((ControlHandle)itemHandle, gStartStr);
                    gClientRunning = false;
                }
                else
                {
                    StartClient();
                    SetCTitle((ControlHandle)itemHandle, gStopStr);
                    gClientRunning = true;
                }
                DrawDialog(gDialogPtr);
                return true;
        }
    }
    return false;
}

static void TCPPrefsReset()
{
    StringToNum(gServerPortStr, &gServerPort);
    StringToNum(gMaxConnectionsStr, &gMaxConnections);
}

static void TCPPrefsDialog()
{
    short   itemType;
    Handle  itemHandle;
    Rect    itemRect;

    gDialogPtr = GetNewDialog(kTCPPrefsDlogResID, NULL, kInFront);
    SetWTitle(gDialogPtr, "\pTCP Preferences");

    GetDItem(gDialogPtr, kServerAddrDItem, &itemType, &itemHandle, &itemRect);
    SetIText(itemHandle, gServerAddrStr);

    GetDItem(gDialogPtr, kServerPortDItem, &itemType, &itemHandle, &itemRect);
    SetIText(itemHandle, gServerPortStr);

    GetDItem(gDialogPtr, kMaxConnectionsDItem, &itemType, &itemHandle, &itemRect);
    SetIText(itemHandle, gMaxConnectionsStr);

    GetDItem(gDialogPtr, kStartStopDItem, &itemType, &itemHandle, &itemRect);
    if (gClientRunning)
        SetCTitle((ControlHandle)itemHandle, gStopStr);
    else
        SetCTitle((ControlHandle)itemHandle, gStartStr);
```

```
    DrawDialog(gDialogPtr);
}

static void DialogClose()
{
    DisposDialog(gDialogPtr);
    gDialogPtr = NULL;
    TCPPrefsReset();
}

static void MenuDispatch(long menu)
{
    short menuID;
    short cmdID;

    menuID = HiWord(menu);
    cmdID  = LoWord(menu);
    switch(menuID)
    {
        case kAppleMenuResID:
        {
            switch (cmdID)
            {
                case kAppleMenuAbout:
                    AboutBox();
                    break;

                default:
                    break;
            }
            break;
        }

        case kFileMenuResID:
        {
            switch (cmdID)
            {
                case kFileMenuQuit:
                    gProgramState = kProgramDone;
                    break;

                case kFileMenuOpen:
                    WindowOpen();
                    break;

                case kFileMenuClose:
                    WindowClose();
                    break;

                default:
                    break;
            }
            break;
        }

        case kEditMenuResID:
            break;

        case kClientMenuResID:
        {
            switch (cmdID)
            {
                case kClientMenuTCPPrefs:
                    TCPPrefsDialog();
                    break;

                default:
                    break;
            }
            break;
        }
    }

}

static void EventDrag(WindowPtr wp, Point loc)
{
    Rect dragBounds;

    dragBounds = qd.screenBits.bounds;
    DragWindow(wp, loc, &dragBounds);
}

static void EventGoAway(WindowPtr wp, Point loc)
{
    if (TrackGoAway(wp, loc))
    {
        if (wp == gWindowPtr)
            WindowClose();
        else if (wp == gDialogPtr)
            DialogClose();
    }
}

static void EventMouseDown(EventRecord* event)
{
    short       part;
    WindowPtr   wp;
    long        menu;

    part = FindWindow(event->where, &wp);
    switch (part)
    {
        case inMenuBar:
            menu = MenuSelect(event->where);
            HiliteMenu(0);
            MenuDispatch(menu);
            break;

        case inDrag:
            EventDrag(wp, event->where);
            break;

        case inGoAway:
            EventGoAway(wp, event->where);
            break;

        case inContent:
            SelectWindow(wp);
            break;

        case inGrow:        // no grow box
        case inZoomIn:      // no zoom box
        case inZoomOut:     // no zoom box
        case inSysWindow:
        case inDesk:
        default:
            break;
    }
}

static void EventKeyDown(EventRecord* event)
{
    char        c;
```

```
    long        menu;

    c = event->message & charCodeMask;
    if (event->modifiers & cmdKey)
    {
        // cmd key
        menu = MenuKey(c);
        HiliteMenu(0);
        if (menu != 0)
            MenuDispatch(menu);
    }
    else
    {
        // normal keystroke
    }
}

static void EventLoop()
{
    EventRecord event;

    while ((gProgramState == kProgramRunning) || (gClientState != kClientStopped))
    {
        OTAtomicAdd32(1, &gCntrIntervalEventLoop);
        if (WaitNextEvent(everyEvent, &event, gSleepTicks, 0))
        {
            if ((gDialogPtr != NULL) && (IsDialogEvent(&event)))
            {
                if (EventDialog(&event))
                    continue;
            }
            switch (event.what)
            {
                case keyDown:
                    EventKeyDown(&event);
                    break;

                case mouseDown:
                    EventMouseDown(&event);
                    break;

                case updateEvt:
                    // redraw window now
                    break;

                case activateEvt:
                    // activate or deactivate window controls
                    break;

                case mouseUp:
                case keyUp:
                case autoKey:
                case diskEvt:
                case app4Evt:
                default:
                    break;
            }
        }

        if (((gProgramState == kProgramRunning) && (gClientState == kClientShuttingDown)) ||
            ((gProgramState != kProgramRunning) && (gClientState != kClientStopped)))
            StopClient();
        else if ((gProgramState == kProgramRunning) && (gClientState == kClientRunning))
            NetEventLoop();
        WindowUpdate();
    }
}
```

```
static void WindowClose()
{
    if (gWindowPtr == NULL)
        return;
    DisposeWindow(gWindowPtr);
    gWindowPtr = NULL;
}

static void WindowOpen()
{
    if (gWindowPtr != NULL)
        return;
    gWindowPtr = GetNewWindow(kWindowResID, NULL, kInFront);
    SetWTitle(gWindowPtr, "\pOTVirtualClient");
}

static void WindowUpdate()
{
    char gStrBuf[128];
    int len;

    if (gWindowPtr == NULL)
        return;

    if (gDoWindowUpdate == false)
        return;
    gDoWindowUpdate = false;

    SetPort(gWindowPtr);
    EraseRgn(gWindowPtr->visRgn);

    gCntrConnections = gCntrEndpts - gCntrIdleEPs - gCntrPending - gCntrBrokenEPs;

    MoveTo(20, 20);
    sprintf(gStrBuf, "EPs: total %d idle %d", gCntrEndpts, gCntrIdleEPs);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 40);
    sprintf(gStrBuf, "Connects: current %d total %d", gCntrConnections, gCntrTotalConnections);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 60);
    sprintf(gStrBuf, "Pending connections %d", gCntrPending);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 80);
    sprintf(gStrBuf, "KBytes received %d", (gCntrTotalBytesRcvd / 1024));
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 100);
    sprintf(gStrBuf, "Conn/sec: current %d max %d", gConnectsPerSecond, gConnectsPerSecondMax);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 120);
    sprintf(gStrBuf, "KBy/sec: current %d max %d", gKBytesPerSecond, gKBytesPerSecondMax);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 140);
    sprintf(gStrBuf, "Events/sec: %d/%d", gEventsPerSecond, gEventsPerSecondMax);
    len = strlen(gStrBuf) ;
```

```
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 160);
    sprintf(gStrBuf, "Running at %d%% of capacity.",
            (100 - ((100 * gEventsPerSecond)/gEventsPerSecondMax)));
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

    MoveTo(20, 180);
    sprintf(gStrBuf, "Disconnects %d", gCntrDiscon);
    len = strlen(gStrBuf) ;
    DrawText(gStrBuf, 0, len);

}


static void SetupMenus()
{
    MenuHandle mh;
    mh = GetMenu(kAppleMenuResID);
    AddResMenu( mh, 'DRVR' );           /* Add DA list */
    InsertMenu(mh, 0);
    mh = GetMenu(kFileMenuResID);
    InsertMenu(mh, 0);
    mh = GetMenu(kEditMenuResID);
    InsertMenu(mh, 0);
    mh = GetMenu(kClientMenuResID);
    InsertMenu(mh, 0);
    DrawMenuBar();
}

static void C2PStr(char* cstr, Str255 pstr)
{
    //
    //  Converts a C string to a Pascal string.
    //  Truncates the string if longer than 254 bytes.
    //
    int i, j;

    i = strlen(cstr);
    if (i > 254)
        i = 254;
    pstr[0] = i;
    for (j = 1; j <= i; j++)
        pstr[j] = cstr[j-1];
}

static void P2CStr(Str255 pstr, char* cstr)
{
    int i;

    for (i = 0; i < pstr[0]; i++)
        cstr[i] = pstr[i+1];
    cstr[i] = 0;
}

static void AlertExit(char* err)
{
    Str255 pErr;

    C2PStr(err, pErr);
    ParamText(pErr, NULL, NULL, NULL);
    Alert(kAlertExitResID, NULL);
    ExitToShell();
}

static void MacInitROM()
```

```
{
    MaxApplZone();
    MoreMasters();
    InitGraf(&qd.thePort);
    InitCursor();
    InitFonts();
    InitWindows();
    InitMenus();
    TEInit();
    InitDialogs(NULL);
    FlushEvents(everyEvent, 0);
}

static void MacInit()
{
    MacInitROM();
    WindowOpen();
    SetupMenus();
}

static void MiscInit()
{
    int i;

    //  This is just so the data is a little better than random for tracing
    for (i = 0; i < kServerRequestSize; i++)
        gServerRequest[i] = i;
}

void main()
{
    MacInit();
    NetInit();
    MiscInit();
    EventLoop();
    NetShutdown();
    if (gProgramState == kProgramError)
        AlertExit(gProgramErr);
}
```