# 跟踪CAsyncSocketEx类族

## 类图：



## 二、各类关系及主要工作：

CEncryptStreamSocket主要负责数据的加密、解密，加密过程的协商等事务，CEMSocket主要负责EDonkey协议数据的发送与接收，而CServerSocket主要负责接收到的数据的处理，以及拟发送数据的封包

## 三、服务器连接过程

### (一)、程序启动后先进行一些，程序初始化工作：

#### 1、初始化本机IP

CeMuleAPP::InitInstance()🡪CServerConnect::CServerConnect()🡪InLocalIP(),

m\_nLocalIP=192.168.1.115

#### 2、初始化服务器列表

CServerListCtrl::OnNmCustomDraw()根据服务器列表反复调用🡪CServerConnect::GetCurrentServer()，

#### 3、启动监听窗口，等待其它客户端连接请求，步骤1.3和1.4与连接服务器无关

eMuleDlg::StartupTimer()🡪CListenSocket::StartListen()

3.1、新建Listen SOCKET

🡪CAsyncSocketEx::Create (UINT nSocketPort /\*=0\*/, int nSocketType /\*=SOCK\_STREAM\*/,long lEvent /\*=FD\_READ | FD\_WRITE | FD\_OOB | FD\_ACCEPT | FD\_CONNECT | FD\_CLOSE\*/, LPCSTR lpszSocketAddress /\*=NULL\*/, BOOL bReuseAddr /\*=FALSE\*/ )

3.1.1、🡪CAsyncSocketEx::InitAsyncSocketExInstance() { m\_spAsyncSocketExThreadDataList==NULL,执行else分支},

3.1.2、将Socket添加到Helper窗口

🡪CAsyncSocketEx::AttachHandle(SOCKET hSocket)🡪 CAsyncSocketExHelperWindow::AddSocket(CAsyncSocketEx \*pSocket, int &nSocketIndex){ m\_nWindowDataSize=512, nSocketIndex=-1, m\_nSocketCount=0, m\_nWindowDataPos=0, m\_pAsyncSocketExWindowData[i % m\_nWindowDataSize].m\_pSocket == NULL,执行if语句块一次，为各变量赋值，nSocketIndex=0, m\_nSocketCount=1, m\_nWindowDataPos=1, m\_pAsyncSocketExWindowData[i % m\_nWindowDataSize].m\_pSocket = 0x0494b3a8 {CListenSocket}, return true;}

3.1.3、选择监听事件

🡪CAsyncSocketEx::AsyncSelect(long lEvent /\*= FD\_READ | FD\_WRITE | FD\_OOB | FD\_ACCEPT | FD\_CONNECT | FD\_CLOSE\*/=8(FD\_ACCEPT)){ m\_lEvent=8(FD\_ACCEPT)，WSAAsyncSelect(m\_SocketData.hSocket, GetHelperWindowHandle(), m\_SocketData.nSocketIndex + WM\_SOCKETEX\_NOTIFY, lEvent)，return true}

3.1.4、 绑定IP

🡪bReuseAddr=0🡪 CAsyncSocketEx::Bind(UINT nSocketPort=55358, LPCSTR lpszSocketAddress=NULL){ sockAddr.sin\_addr.saddr=0.0.0.0}

(2)开始监听

🡪CAsyncSocketEx::Listen(int nConnectionBacklog=5)

#### 4、定时器保证连接处于活跃状态

CUploadQueue::UploadTimer()🡪CServerConnect::KeepConnectionAlive(){if(false)},定时调用，保持连接

### (二)、服务器连接开始

通过点击“连接”按钮，或根据选项设置，程序自动启动连接，首先读取服务器列表中的服务器 ，停止当前正在进行的连接，开始连接尝试

CemuleDlg::OnBnClickedConnect()🡪StartConnect()🡪 CServerConnect:: ConnectToAnyServer（UINT startAt=0, bool prioSort=true, bool isAuto=true, bool bNoCrypt=false）🡪StopConnectionTry(){m\_idRetryTimer=0,for不执行}

🡪Disconnect()(if(false),return false；)

🡪TryAnotherConnectionRequest(){next\_server=first Sever in list}

🡪ConnectToServer(CServer\* server=next\_server, bool multiconnect=true, bool bNoCrypt=true)

#### 1、新建ServerSocket

🡪CAsyncSocketEx::Create(UINT nSocketPort=0, int nSocketType=1/\*=SOCK\_STREAM\*/,

long lEvent=51 /\*=FD\_READ | FD\_WRITE | FD\_CONNECT | FD\_CLOSE\*/,

LPCSTR lpszSocketAddress=NULL /\*=NULL\*/, BOOL bReuseAddr=0 /\*=FALSE\*/ )

1.1🡪 CAsyncSocketEx::InitAsyncSocketExInstance(){ if(m\_spAsyncSocketExThreadDataList)==true,执行if分支,if(!pList)==false}

1.2同（一）3.1.2 m\_nWindowDataPos=2

1.3同（一）3.1.3 lEvent=51

1.4 同（一）3.1.4 nSocketPort=0

#### 2、向服务器发出连接请求

🡪 CServerConnect::ConnectTo(CServer\* server, bool bNoCrypt)

根据bNoCrypt是否为true，分别发起加密连接或未加密连接。当Enable protocol obfuscation未选中时，IsServerCryptLayerTCPRequested()返回false，m\_bTryObfuscated为false。

##### 2.1、未加密连接

###### 2.1.1发起连接

🡪 ConnectTo(CServer\* server=0x04652650, bool bNoCrypt=true)

###### 2.1.2 设置连接加密状态为m\_StreamCryptState = ECS\_NONE

🡪 CEncryptedStreamSocket::SetConnectionEncryption(bool bEnabled=false, const uchar\* pTargetClientHash=NULL, bool bServerConnection=true){m\_streamCryptState=ECS\_UNKNOWN,执行最后的else块，置**m\_StreamCryptState = ECS\_NONE;**}

###### 2.1.3 设置连接状态为CS\_CONNECTING

🡪SetConnectionState(**CS\_CONNECTING=1**){if,else if均为false,直接返回};

###### 2.1.4 调用socket类发起connect

🡪CEMSocket::Connect(LPCSTR lpszHostAddress=0x05115568, UINT nHostPort=5041)🡪 InitProxySupport(){直接返回}🡪 CAsyncSocketEx::Connect(LPCSTR lpszHostAddress=0x05115568, UINT nHostPort=5041)🡪 connect(m\_SocketData.hSocket=1136, lpSockAddr, nSockAddrLen=16)

###### 2.1.5、处理服务器返回连接信息

如果服务器回应了连接请求，HelperWinodw的FD\_CONNECT事件触发，CServerSocket::OnConnect()被调用，根据OnConnect中返回的nErrorCode，如果成功，设置连接状态为等待登录（CS\_WAITFORLOGIN）,调用ConnectionEstablished，如果失败，设置连接状态为CS\_SERVERDEAD或CS——SERVERFATAL，调用ConnectionFailed

(1)、连接成功

①、设置连接状态为等待登录（**CS\_WAITFORLOGIN**）, ConnectionEstablished函数被调用，准备登录信息，调用SendPacket向服务器发送登录信息包，信息包存入待发送队列等待发送, 由UploadBandwidthThrottler根据带宽情况进行择机发送。：

🡪CAsyncSocketExHelperWindow::WindowsProc(message=1284>=WM\_SOCKETEX\_NOTIFY){ hSocket=wParam=1136,nEvent=16(FD\_CONNECT)}🡪CServerSocket::OnConnect(nErrorCode=0)🡪 SetConnectionState(CS\_WAITFORLOGIN)

//SetConnectionState()调用CServerConnect::ConnectionEstablished()，处理CS\_WAITFORLOGIN分支：

🡪 CServerConnect::ConnectionEstablished(CServerSocket\* sender=0x050c5d10)🡪InitLocalIP(){m\_nLocalIP=192.168.1.115}

🡪if(sender->GetConnectionState() == CS\_WAITFORLOGIN),SendPacket(Packet\* packet,bool delpacket= true, CServerSocket\* sender)🡪 CServerSocket::SendPacket(Packet\* packet, bool delpacket=true, bool controlpacket=true, uint32 actualPayloadSize=0, bool bForceImmediateSend=false)🡪

CEMSocket:: SendPacket(Packet\* packet, bool delpacket=true, bool controlpacket=true, uint32 actualPayloadSize=0, bool bForceImmediateSend=false){if(controlpacket){ controlpacket\_queue.AddTail(packet); // queue up for controlpacket

theApp.uploadBandwidthThrottler->QueueForSendingControlPacket(this, HasSent());},if(bForceImmediateSend)==false}



②、可以发送，UploadBandwidthThrottler发送信息包，触发OnSend()

因为是未加密连接，CEncryptedStreamSocket::OnSend()未做出相应动作，没有启动加密连接协商。这边需要理清一下，为什么先发一个9字节的控制包，再发送一个33800字节的控制包，需要跟踪一下发送列表的包的详细情况

？？？///snow:下面这两部分需要厘清

UploadBandwidthThrottler::RunProc(LPVOID pParam)启动🡪RunInternal()🡪CEMSocket:: SendControlData(uint32 maxNumberOfBytesToSend=9, uint32 minFragSize=536)🡪Send(uint32 maxNumberOfBytesToSend=9, uint32 minFragSize=536, bool onlyAllowedToSendControlPacket=true)

🡪CAsyncSocketExHelperWindow::WindowProc(){case FD\_WRITE}🡪CEMSocket::OnSend(0)

🡪 CEncryptedStreamSocket::OnSend(int nErrorCode=0){nothing to do}

byConnected = ES\_CONNECTED;

2.2.3、

UploadBandwidthThrottler::RunProc(LPVOID pParam)启动🡪RunInternal()🡪CEMSocket:: SendControlData(uint32 maxNumberOfBytesToSend=33800, uint32 minFragSize=1300)🡪Send(uint32 maxNumberOfBytesToSend=33800, uint32 minFragSize=1300, bool onlyAllowedToSendControlPacket=true)🡪CEncryptStreamSocket::Send(const \*lpBuf=0x051fbc40,int nBuflen=86,int nFlags=0)🡪CAsyncSocketEx:Send(const \*lpBuf=0x051fbc40,int nBuflen=86,int nFlags=0)

？？？

③、接收服务器返回信息，触发OnReceive()

🡪CAsyncSocketExHelperWindow::WindowsProc(message=1284>=WM\_SOCKETEX\_NOTIFY){ hSocket=wParam=1136,nEvent=2(FD\_READ)}🡪CServerSocket::OnReceive(nErrorCode=0)🡪 CEMSocket::OnReceive(nErrorCode)

判断是否存在下载速度限制，在未超速的情况下，OnReceive()调用Receive()接收当前情况下可以接收的最大字节数，存入[GlobalReadBuffer + pendingHeaderSize]处。

逐级调用Receive()，最终调用socket函数recv函数，接收数据

🡪 CEMSocket::Receive(GlobalReadBuffer + pendingHeaderSize, readMax);

因为是未加密连接，所以不需要进行解密处理

🡪 CEncryptedStreamSocket::Receive(lpBuf,nBufLen,nFlags){case ECS\_NONE:return m\_nObfuscationBytesReceived;}🡪 CAsyncSocketEx::Receive(lpBuf, nBufLen, nFlags);🡪 recv(m\_SocketData.hSocket, (LPSTR)lpBuf, nBufLen, nFlags);

④、接收到数据后，OnReceive()调用PacketReceive()，因为CEMSocket::PacketReceived()是个虚函数，所以调用CServerSocket::PacketReceived()进行处理

⑤、PacketReceived()先判断包是否进行打包了，若是先进行解包；然后调用CServerSocket::ProcessPacket()对包进行处理

⑥、ProcessPacket()根据opcode对包分别进行处理，在连接时主要是OP\_IDCHANGE、OP\_SERVERMESSAGE等

⑦、OP\_IDCHANGE处理分支设置连接状态为CS\_CONNECTED ,调用SetConnectionState(CS\_CONNECTED)

⑧、SetConnectionState()调用CServerSocket::ConnectionEstablished()，处理CS\_CONNECTED分支，向服务器发送共享文件列表，根据选项中的“从服务器更新服务器列表”，发送请求服务器列表信息包。这里一共两次向服务器发送数据。

⑨、重复③∽⑥，只是⑥中处理opcode的分支不同。

(2)、连接失败

连接失败可能发生在三个阶段：OnHostNameResolved、OnConnect、OnClose中，OnHostNameResolved中返回的是CS\_ERROR，OnConnect中返回的是CS\_FATALERROR、CS\_SERVERDEAD，OnClose中返回的是CS\_DISCONNECTED、CS\_SERVERFULL、CS\_NOTCONNECTED。它们全部通过SetConnectionState设置，然后调用CServerConnect::ConnectionFailed()进行处理，根据情况重新发起连接尝试：

①、case CS\_FATALERROR:暂停30秒，从下一服务器开始连接尝试

case CS\_DISCONNECTED:从服务器列表开始重新连接尝试

②、case CS\_ERROR:

case CS\_NOTCONNECTED:直接退出

③、case CS\_SERVERDEAD:

case CS\_SERVERFULL:如果是单个连接且是加密连接，试着进行非加密连接，否则从下一服务器开始连接尝试。

##### 2.2、加密连接

* 客户端进行加密连接的准备已就绪，Enable protocol obfuscation选项被选中，第三项Disable support for obfuscated connections未选，第二项只跟客户端与客户端之间的连接有关。ConnectToAnyServer （）中bNoCrypt参数默认为false，优先进行乱序加密连接。
* 

###### 2.1.1发起连接

🡪 ConnectTo(CServer\* server=0x04652650, bool bNoCrypt=false)

if ( !bNoCrypt && thePrefs.IsServerCryptLayerTCPRequested() && server->GetObfuscationPortTCP() != 0 && server->SupportsObfuscationTCP()){

nPort = cur\_server->GetObfuscationPortTCP();

SetConnectionEncryption(true, NULL, true);

}

###### 2.1.2 设置连接加密状态为m\_StreamCryptState = ECS\_NONE

🡪 CEncryptedStreamSocket::SetConnectionEncryption(bool bEnabled=true, const uchar\* pTargetClientHash=NULL, bool bServerConnection=true){m\_streamCryptState=ECS\_UNKNOWN,执行else if块，m\_bServerCrypt =true;**m\_StreamCryptState = ECS\_PENDING\_SERVER;**}

###### 2.1.3 设置连接状态为CS\_CONNECTING

🡪SetConnectionState(**CS\_CONNECTING=1**){if,else if均为false,直接返回};

###### 2.1.4 调用socket类发起connect

🡪CEMSocket::Connect(LPCSTR lpszHostAddress=0x05115568, UINT nHostPort=5041)🡪 InitProxySupport(){直接返回}🡪 CAsyncSocketEx::Connect(LPCSTR lpszHostAddress=0x05115568, UINT nHostPort=5041)🡪 connect(m\_SocketData.hSocket=1136, lpSockAddr, nSockAddrLen=16)

###### 2.1.5、处理服务器返回连接信息



如果服务器回应了连接请求，HelperWinodw的FD\_CONNECT事件触发，CServerSocket:: OnConnect() 被调用，根据OnConnect中返回的nErrorCode，如果成功，设置连接状态为等待登录（CS\_WAITFORLOGIN）,调用ConnectionEstablished，如果失败，设置连接状态为CS\_SERVERDEAD或CS——SERVERFATAL，调用ConnectionFailed

(1)、连接成功

①、设置连接状态为等待登录（**CS\_WAITFORLOGIN**）, ConnectionEstablished函数被调用，准备登录信息，调用SendPacket向服务器发送登录信息包，信息包存入待发送队列等待发送, 由UploadBandwidthThrottler根据带宽情况进行择机发送。：

🡪CAsyncSocketExHelperWindow::WindowsProc(message=1284>=WM\_SOCKETEX\_NOTIFY){ hSocket=wParam=1136,nEvent=16(FD\_CONNECT)}🡪CServerSocket::OnConnect(nErrorCode=0)🡪 SetConnectionState(CS\_WAITFORLOGIN)

//在SetConnectionState()中设置连接状态（CS\_WAITFORLOGIN）后，根据连接状态调用CServerConnect::ConnectionEstablished()，处理CS\_WAITFORLOGIN分支：

🡪 CServerConnect::ConnectionEstablished(CServerSocket\* sender=0x050c5d10) 🡪InitLocalIP(){m\_nLocalIP=192.168.1.115}

🡪if(sender->GetConnectionState() == CS\_WAITFORLOGIN),

准备登录信息包，然后调用SendPacket将包添加到发送队列末尾

🡪SendPacket(Packet\* packet,bool delpacket= true, CServerSocket\* sender)🡪 CServerSocket::SendPacket(Packet\* packet, bool delpacket=true, bool controlpacket=true, uint32 actualPayloadSize=0, bool bForceImmediateSend=false)🡪

CEMSocket:: SendPacket(Packet\* packet, bool delpacket=true, bool controlpacket=true, uint32 actualPayloadSize=0, bool bForceImmediateSend=false){if(controlpacket){ controlpacket\_queue.AddTail(packet); // queue up for controlpacket

theApp.uploadBandwidthThrottler->QueueForSendingControlPacket(this, HasSent());},if(bForceImmediateSend)==false}

到这里，OnConnect()处理完成。



②、可以发送，UploadBandwidthThrottler发送信息包，触发OnSend()

在数据被正式发送出去之前，OnSend()首先触发！！！先于send()。问题是OnSend()谁来触发？？？connect()会不会触发OnSend()?看起来不会，有待下一步研究。

因为是加密连接，CEncryptedStreamSocket::OnSend()执行m\_StreamCryptState == ECS\_PENDING\_SERVER语句块，🡪StartNegotiation(true);启动加密连接协商。同样，在StartNegotiation()函数中,执行m\_StreamCryptState == ECS\_PENDING\_SERVER语句块，准备Client端的握手协商报文，设置

m\_NegotiatingState = ONS\_BASIC\_SERVER\_DHANSWER;

m\_StreamCryptState = ECS\_NEGOTIATING;

然后调用🡪 SendNegotiatingData()，bDelaySend参数未赋值，默认为false。nBufLen – nStartCryptFromByte=0，不加密数据，直接调用CAsyncSocketEx::Send()发送。

这边需要理清一下，为什么先发一个9字节的控制包，再发送一个33800字节的控制包，需要跟踪一下发送列表的包的详细情况

？？？///snow:下面这两部分需要厘清

UploadBandwidthThrottler::RunProc(LPVOID pParam)启动🡪RunInternal()🡪CEMSocket:: SendControlData(uint32 maxNumberOfBytesToSend=9, uint32 minFragSize=536)🡪Send(uint32 maxNumberOfBytesToSend=9, uint32 minFragSize=536, bool onlyAllowedToSendControlPacket=true)

🡪CAsyncSocketExHelperWindow::WindowProc(){case FD\_WRITE}🡪CEMSocket::OnSend(0)

🡪 CEncryptedStreamSocket::OnSend(int nErrorCode=0){startNegotiation()}

byConnected = ES\_CONNECTED;

2.2.3、

UploadBandwidthThrottler::RunProc(LPVOID pParam)启动🡪RunInternal()🡪CEMSocket:: SendControlData(uint32 maxNumberOfBytesToSend=33800, uint32 minFragSize=1300)🡪Send(uint32 maxNumberOfBytesToSend=33800, uint32 minFragSize=1300, bool onlyAllowedToSendControlPacket=true)🡪CEncryptStreamSocket::Send(const \*lpBuf=0x051fbc40,int nBuflen=86,int nFlags=0)🡪CAsyncSocketEx:Send(const \*lpBuf=0x051fbc40,int nBuflen=86,int nFlags=0)

？？？

③、接收服务器返回信息，触发OnReceive()

🡪CAsyncSocketExHelperWindow::WindowsProc(message=1284>=WM\_SOCKETEX\_NOTIFY){ hSocket=wParam=1136,nEvent=2(FD\_READ)}🡪CServerSocket::OnReceive(nErrorCode=0)🡪 CEMSocket::OnReceive(nErrorCode)

判断是否存在下载速度限制，在未超速的情况下，OnReceive()调用Receive()接收当前情况下可以接收的最大字节数，存入[GlobalReadBuffer + pendingHeaderSize]处。

逐级调用Receive()，最终调用socket函数recv函数，接收数据

🡪 CEMSocket::Receive(GlobalReadBuffer + pendingHeaderSize, readMax);

🡪 CEncryptedStreamSocket::Receive(lpBuf,nBufLen,nFlags){ 🡪 CAsyncSocketEx::Receive(lpBuf, nBufLen, nFlags);🡪 recv(m\_SocketData.hSocket, (LPSTR)lpBuf, nBufLen, nFlags);

因为在StartNegotiation()时，m\_StreamCryptState = ECS\_NEGOTIATING;所以Receive函数执行case ECS\_NEGOTIATING：语句块，首先调用Negotiate()，在switch中执行case ONS\_BASIC\_SERVER\_DHANSWER:语句块，准备DH密钥，设置m\_NegotiatingState = ONS\_BASIC\_SERVER\_MAGICVALUE;

④、接收到数据后，OnReceive()调用PacketReceive()，因为CEMSocket::PacketReceived()是个虚函数，所以调用CServerSocket::PacketReceived()进行处理

⑤、PacketReceived()先判断包是否进行打包了，若是先进行解包；然后调用CServerSocket::ProcessPacket()对包进行处理

⑥、ProcessPacket()根据opcode对包分别进行处理，在连接时主要是OP\_IDCHANGE、OP\_SERVERMESSAGE等

⑦、OP\_IDCHANGE处理分支设置连接状态为CS\_CONNECTED ,调用SetConnectionState(CS\_CONNECTED)

⑧、SetConnectionState()调用CServerSocket::ConnectionEstablished()，处理CS\_CONNECTED分支，向服务器发送共享文件列表，根据选项中的“从服务器更新服务器列表”，发送请求服务器列表信息包。这里一共两次向服务器发送数据。

⑨、重复③∽⑥，只是⑥中处理opcode的分支不同。

(2)、连接失败

连接失败可能发生在三个阶段：OnHostNameResolved、OnConnect、OnClose中，OnHostNameResolved中返回的是CS\_ERROR，OnConnect中返回的是CS\_FATALERROR、CS\_SERVERDEAD，OnClose中返回的是CS\_DISCONNECTED、CS\_SERVERFULL、CS\_NOTCONNECTED。它们全部通过SetConnectionState设置，然后调用CServerConnect::ConnectionFailed()进行处理，根据情况重新发起连接尝试：

①、case CS\_FATALERROR:暂停30秒，从下一服务器开始连接尝试

case CS\_DISCONNECTED:从服务器列表开始重新连接尝试

②、case CS\_ERROR:

case CS\_NOTCONNECTED:直接退出

③、case CS\_SERVERDEAD:

case CS\_SERVERFULL:如果是单个连接且是加密连接，试着进行非加密连接，否则从下一服务器开始连接尝试。

# [emule中节点加入Kad网络过程（源代码详解）【对原文部分改进】](http://blog.csdn.net/jo_say/article/details/8082491)

emule中节点加入Kad网络过程（源代码详解）

程序启动：

**EmuleDlg.cpp中函数**BOOL CemuleDlg::OnInitDialog()**，此函数用于对话框的初始化，在这个函数里添加了定时器**：VERIFY( (m\_hTimer = ::SetTimer(NULL, NULL, 300, StartupTimer)) != NULL );

**在这里添加了函数**void CALLBACK CemuleDlg::**StartupTimer**(HWND /\*hwnd\*/, UINT /\*uiMsg\*/, UINT /\*idEvent\*/, DWORD /\*dwTime\*/)，

case 2:

theApp.Kad\_Dlg->status++;

if(!theApp.listensocket->StartListening())

ASSERT(0);

if(!theApp.clientudp->Create())

ASSERT(0);

theApp.Kad\_Dlg->status++;

break;

**[PS: 现在已经不是这样了，没有了Kad\_Dlg, 在cemuleDlg.cpp的2087行调用了Kad的Start()函数]**

**在StartupTimer这个函数里，添加了一个ListenSocket的侦听端，并且在本地节点创建了一个CClientUDPSocket\* clientudp;**

然后程序启动。

顺便说一句，在CEmule类中定义了许多的类的实例，这都在今后使用到：

UploadBandwidthThrottler\* uploadBandwidthThrottler;

CClientList\* clientlist;

CClientUDPSocket\* clientudp;

CListenSocket\* listensocket;

CSharedFileList\* sharedfiles;

CDownloadQueue\* downloadqueue;

CUploadQueue\* uploadqueue;

CServerList\* serverlist;

LastCommonRouteFinder\* lastCommonRouteFinder;

CServerConnect\* serverconnect;

CIPFilter\* ipfilter;

CClientCreditsList\* clientcredits;

CSearchList\* searchlist;

CKnownFileList\* knownfiles;

CMMServer\* mmserver;

AppState m\_app\_state; // defines application state for shutdown

CMutex hashing\_mut;

CString m\_strCurVersionLong;

CPeerCacheFinder\* m\_pPeerCache;

CFriendList\* friendlist;

CFirewallOpener\* m\_pFirewallOpener;//hyper added

节点加入网络：

**[emuledlg.cpp的：2087行 ]**

**Emule连接Kad网络时**，调用函数：Kademlia::CKademlia::Start(); Start()这个函数没有做什么实际意义上的事情，主要是new了几个类：

m\_pInstance = new CKademlia();

m\_pInstance->m\_pPrefs = pPrefs;

m\_pInstance->m\_pUDPListener = NULL;

m\_pInstance->m\_pRoutingZone = NULL;

m\_pInstance->m\_pIndexed = new CIndexed();

m\_pInstance->m\_pRoutingZone = new CRoutingZone();

m\_pInstance->m\_pUDPListener = new CKademliaUDPListener();

并且更改了几个定时器的时间。

**接着程序转入到routingzone.cpp中执行**。

在上面那部分的Start ()函数体内部初始化了CRoutingZone这个类，这个类的构造函数CRoutingZone::CRoutingZone()体中调用函数 Init(NULL, 0, CUInt128((ULONG)0));来初始化根节点（应该就是本地节点）。

             // Can only create routing zone after prefs

             // Set our KadID for creating the contact tree

             CKademlia ::GetPrefs ()-> GetKadID(& uMe );

             m\_sFilename = szFilename ;

             // Init our root node.

             Init (NULL , 0, CUInt128(( ULONG )0));

在void CRoutingZone::Init(CRoutingZone \*pSuper\_zone, int iLevel, const CUInt128 &uZone\_index)函数体内部创建了一个新的m\_pBin = new CRoutingBin();

             // Init all Zone vars

             // Set this zones parent

             m\_pSuperZone = pSuper\_zone ;

             // Set this zones level

             m\_uLevel = iLevel ;

             // Set this zones CUInt128 Index

             m\_uZoneIndex = uZone\_index ;

             // Mark this zone has having now leafs.

             m\_pSubZones [0] = NULL ;

             m\_pSubZones [1] = NULL ;

             // Create a new contact bin as this is a leaf.

             m\_pBin = new CRoutingBin();

             // Set timer so that zones closer to the root are processed earlier.

             m\_tNextSmallTimer = time ( NULL) + m\_uZoneIndex .Get32BitChunk (3);

             // Start this zone.

             StartTimer ();

             // If we are initializing the root node, read in our saved contact list.

             if ((m\_pSuperZone == NULL) && ( m\_sFilename .GetLength () > 0))

                         ReadFile ();

**接着调用函数StartTime（）**，用来开始这个区域。在StartTime（）函数内部添加事件CKademlia::AddEvent(this);

             time\_t tNow = time( NULL );

             // Start filling the tree, closest bins first.

             m\_tNextBigTimer = tNow + SEC(10);

             CKademlia ::AddEvent ( this);

在调用完函数StartTime（）函数后，从文件中读取以前保存的联系人。

在调用完函数Kademlia::CKademlia::Start();之后，Kademlia开始处理，转入函数**Kademlia:: CKademlia::Process()**开始执行，在函数void CKademlia::Process()中调用函数**pZone->OnSmallTimer();即CRoutingZone中 OnSmallTimer().。**

line 274:

if (pZone -> m\_tNextSmallTimer <= tNow )

                        {

                                     pZone ->OnSmallTimer ();

                                     pZone ->m\_tNextSmallTimer = MIN2S(1) + tNow ;

                        }

**CRoutingZone中OnSmallTimer()**，在此函数体内，当判断联系人为非空时，调用函数 CKademlia::GetUDPListener()->SendMyDetails\_KADEMLIA2(KADEMLIA2\_HELLO\_REQ, pContact->GetIPAddress(), pContact->GetUDPPort());来发送本地节点的一些信息，其中函数的第一个参数是消息的类型， KADEMLIA2\_HELLO\_REQ表明是Kademlia 2.0网络的加入请求，相当于TCP/IP中的ACK，即表明这个消息是用来加入网络的。第二个参数是本地节点的IP，第三个节点是本地节点的端口。

             if (pContact != NULL)

            {

                         pContact ->CheckingType ();

                         if (pContact -> GetVersion() >= 6){ /\*48b\*/

                                     if (thePrefs . GetDebugClientKadUDPLevel() > 0)

                                                 DebugSend ("KADEMLIA2\_HELLO\_REQ" , pContact ->GetIPAddress (), pContact-> GetUDPPort ());

                                     CUInt128 uClientID = pContact-> GetClientID ();

                                     CKademlia ::GetUDPListener ()-> SendMyDetails( KADEMLIA2\_HELLO\_REQ , pContact ->GetIPAddress (), pContact-> GetUDPPort (), pContact -> GetVersion(), pContact ->GetUDPKey (), & uClientID, false );

                                     if (pContact -> GetVersion() >= KADEMLIA\_VERSION8\_49b ){

                                                 // FIXME:

                                                 // This is a bit of a work arround for statistic values. Normally we only count values from incoming HELLO\_REQs for

                                                 // the firewalled statistics in order to get numbers from nodes which have us on their routing table,

                                                 // however if we send a HELLO due to the timer, the remote node won't send a HELLO\_REQ itself anymore (but

                                                 // a HELLO\_RES which we don't count), so count those statistics here. This isn't really accurate, but it should

                                                 // do fair enough. Maybe improve it later for example by putting a flag into the contact and make the answer count

                                                 CKademlia ::GetPrefs ()-> StatsIncUDPFirewalledNodes( false );

                                                 CKademlia ::GetPrefs ()-> StatsIncTCPFirewalledNodes( false );

                                    }

**接着转入**KademliaUDPListener.cpp中函数void CKademliaUDPListener::SendMyDetails\_KADEMLIA2(byte byOpcode, uint32 uIP, uint16 uUDPPort)运行，主要是调用函数SendPacket(byPacket, uLen, uIP, uUDPPort);，SendPacket(byPacket, uLen, uIP, uUDPPort);函数在KademliaUDPListener.cpp内部，此函数体内部调用函数theApp.clientudp-> SendPacket(pPacket, ntohl(uDestinationHost), uDestinationPort);来发送包。

uint32 uLen = sizeof( byPacket ) - byteIOResponse . GetAvailable();

                         if (byKadVersion >= KADEMLIA\_VERSION6\_49aBETA){

                                     if (isnulmd4 ( uCryptTargetID-> GetDataPtr ())){

                                                 DebugLogWarning (\_T ( "Sending hello response to crypt enabled Kad Node which provided an empty NodeID: %s (%u)"), ipstr (ntohl ( uIP)), byKadVersion );

                                                 SendPacket (byPacket , uLen,  uIP , uUDPPort , targetUDPKey, NULL );

                                    }

                                     else

                                                 SendPacket (byPacket , uLen,  uIP , uUDPPort , targetUDPKey, uCryptTargetID );

                        }

                         else {

                                     SendPacket (byPacket , uLen,  uIP , uUDPPort , 0, NULL);

                                     ASSERT ( targetUDPKey . IsEmpty() );

                        }

KademliaUDPListener.cpp内部CKademliaUDPListener ::SendPacket之一：

{

             if (uLenData < 2) {

                         ASSERT (0);

                         return ;

            }

             AddTrackedOutPacket (uDestinationHost , pbyData[1]);

             Packet \* pPacket = new Packet (OP\_KADEMLIAHEADER );

             pPacket ->opcode = pbyData[1];

             pPacket ->pBuffer = new char [uLenData +8];

             memcpy (pPacket -> pBuffer, pbyData +2, uLenData -2);

             pPacket ->size = uLenData-2;

             if ( uLenData > 200 )

                         pPacket ->PackPacket ();

             theStats .AddUpDataOverheadKad ( pPacket-> size );

**theApp .clientudp -> SendPacket( pPacket , ntohl ( uDestinationHost), uDestinationPort , true**

**, ( uCryptTargetID != NULL ) ? uCryptTargetID-> GetData () : NULL**

**, true , targetUDPKey . GetKeyValue( theApp .GetPublicIP ( false)));**

}

**ClientUDPSocket.cpp中（565line）函数**theApp.clientudp->SendPacket(pPacket, ntohl(uDestinationHost), uDestinationPort);体内部将刚才的消息包（或者叫数据包）加入到controlpacket\_queue的队尾，

**controlpacket\_queue.AddTail(newpending); // line586**

 controlpacket\_queue是一个链表，类型是CTypedPtrList<CPtrList, UDPPack\*> controlpacket\_queue;，

CTypedPtrList <CPtrList , UDPPack\*> controlpacket\_queue ;

// ZZ:UploadBandWithThrottler (UDP) -->

    sendLocker. Lock ();

             controlpacket\_queue .AddTail ( newpending);

    sendLocker. Unlock ();

    theApp. uploadBandwidthThrottler ->QueueForSendingControlPacket ( this);

             return true ;

// <-- ZZ:UploadBandWithThrottler (UDP)

是通过**模板**来实现的。接着继续调用函数theApp.uploadBandwidthThrottler- >QueueForSendingControlPacket(this);此时数据包在链表UploadBandwidthThrottler\* uploadBandwidthThrottler;中排队。

**类UploadBandwidthThrottler继承自CWinThread类，主要是作为线程来运行的。**

类在初始化，在构造函数中调用函数 UINT AFX\_CDECL UploadBandwidthThrottler::RunProc(LPVOID pParam)，

UploadBandwidthThrottler ::UploadBandwidthThrottler ( void) {

             m\_SentBytesSinceLastCall = 0;

             m\_SentBytesSinceLastCallOverhead = 0;

    m\_highestNumberOfFullyActivatedSlots = 0;

             threadEndedEvent = new CEvent(0, 1);

             pauseEvent = new CEvent( TRUE , TRUE );

             doRun = true ;

             AfxBeginThread (RunProc , ( LPVOID) this );

}

UINT AFX\_CDECL UploadBandwidthThrottler:: RunProc (LPVOID pParam) {

             DbgSetThreadName ("UploadBandwidthThrottler" );

             InitThreadLocale ();

             UploadBandwidthThrottler \* uploadBandwidthThrottler = ( UploadBandwidthThrottler\*) pParam ;

             return uploadBandwidthThrottler -> RunInternal();

}

这个函数调用uploadBandwidthThrottler->RunInternal();，RunInternal()函 数主要用来发送来自socket的数据包，函数体内调用两个函数：

SocketSentBytes socketSentBytes = socket->SendControlData(allowedDataRate > 0?(UINT)(bytesToSpend - spentBytes):1, minFragSize);

以及

   if( socket != NULL ) {

                    SocketSentBytes socketSentBytes = socket-> SendControlData (allowedDataRate > 0?(UINT )(bytesToSpend - spentBytes):1, minFragSize );

                                                    uint32 lastSpentBytes = socketSentBytes .sentBytesControlPackets + socketSentBytes. sentBytesStandardPackets ;

                                                    spentBytes += lastSpentBytes ;

                                                    spentOverhead += socketSentBytes . sentBytesControlPackets;

                                        }

  if( neededBytes > 0) {

                                                                            SocketSentBytes socketSentBytes = socket ->SendFileAndControlData ( neededBytes, minFragSize );

                                                                            uint32 lastSpentBytes = socketSentBytes .sentBytesControlPackets + socketSentBytes. sentBytesStandardPackets ;

                                                                            spentBytes += lastSpentBytes ;

                                                                            spentOverhead += socketSentBytes .sentBytesControlPackets ;

                            if (lastSpentBytes > 0 && slotCounter < m\_highestNumberOfFullyActivatedSlots ) {

                                m\_highestNumberOfFullyActivatedSlots = slotCounter ;

                            }

                                                                }

SocketSentBytes socketSentBytes = socket->SendFileAndControlData(neededBytes, minFragSize);

其中的socket类型是ThrottledFileSocket\*，在类ThrottledFileSocket中这两个函数被定义为虚函数，

class ThrottledFileSocket : public ThrottledControlSocket

{

public :

    virtual SocketSentBytes SendFileAndControlData ( uint32 maxNumberOfBytesToSend , uint32 minFragSize ) = 0;

    virtual DWORD GetLastCalledSend () = 0;

    virtual uint32   GetNeededBytes () = 0;

             virtual bool           IsBusy () const = 0;

    virtual bool     HasQueues () const = 0;

             virtual bool           UseBigSendBuffer ()                                                                                            { return false ; }

};

而 且在这个类内部没有具体实现，它们的实现在类CClientUDPSocket中，类CClientUDPSocket继承自**CAsyncSocket**以 及**ThrottledControlSocket**，如下代码：

class CClientUDPSocket : public CAsyncSocket, public ThrottledControlSocket // ZZ:UploadBandWithThrottler (UDP)。

socket->SendControlData(allowedDataRate > 0?(UINT)(bytesToSpend - spentBytes):1, minFragSize);

class CClientUDPSocket : public CAsyncSocket , public CEncryptedDatagramSocket, public ThrottledControlSocket // ZZ:UploadBandWithThrottler (UDP)

{

public :

             CClientUDPSocket ();

             virtual ~CClientUDPSocket ();

             bool       Create ();

             bool       Rebind ();

             uint16    GetConnectedPort ()                               { return m\_port ; }

             bool       SendPacket ( Packet\* packet , uint32 dwIP, uint16 nPort , bool bEncrypt , const uchar \* pachTargetClientHash );

    SocketSentBytes  SendControlData (uint32 maxNumberOfBytesToSend, uint32 minFragSize ); // ZZ:UploadBandWithThrottler (UDP)

protected :

以及

SocketSentBytes socketSentBytes = socket->**SendFileAndControlData**(neededBytes, minFragSize);的实现体在**ClientUDPSocket**.cpp中424行：[ps:newversion中可能没这个了]

SocketSentBytes CClientUDPSocket::**SendControlData**(uint32 maxNumberOfBytesToSend, uint32 /\*minFragSize\*/){ // ZZ:UploadBandWithThrottler (UDP)

**在它们内部调用了函数SendTo**，if (!SendTo(sendbuffer, cur\_packet->packet->size+2, cur\_packet->dwIP, cur\_packet->nPort))（在ClientUDPSocket.cpp中528行）。这个函数是类CClientUDPSocket 的成员函数。int CClientUDPSocket::SendTo(char\* lpBuf,int nBufLen,uint32 dwIP, uint16 nPort)，在这个函数体内调用类CAsyncSocket的成员函数uint32 result = CAsyncSocket::SendTo(lpBuf,nBufLen,nPort,ipstr(dwIP));，类CAsyncSocket是MFC 的类库中的一个类。【NND，终于找到头了】

  if (! SendTo ((char \*) sendbuffer, nLen , cur\_packet -> dwIP, cur\_packet ->nPort )){

                sentBytes += nLen ; // ZZ:UploadBandWithThrottler (UDP)

                                                 controlpacket\_queue .RemoveHead ();

                                                 delete cur\_packet -> packet;

                                                 delete cur\_packet ;

            }

int CClientUDPSocket :: SendTo( char \* lpBuf , int nBufLen ,uint32 dwIP, uint16 nPort ){

             // NOTE: \*\*\* This function is invoked from a \*different\* thread!

**uint32 result = CAsyncSocket:: SendTo (lpBuf , nBufLen, nPort ,ipstr ( dwIP));**

             if (result == ( uint32) SOCKET\_ERROR ){

                         uint32 error = GetLastError();

                         if (error == WSAEWOULDBLOCK){

                                     m\_bWouldBlock = true ;

                                     return -1;

                        }

                         if (thePrefs . GetVerbose())

                                     DebugLogError (\_T ( "Error: Client UDP socket, failed to send data to %s:%u: %s"), ipstr( dwIP ), nPort , GetErrorMessage( error , 1));

            }

             return 0;

}

至此，本地节点加入网络的请求就发送完毕。

* 下面讲述本地节点在接收到来自其他节点的回应后在本地采取的一些措施从而把自己加入到网络内。

**当网络事件发生时（即本地网卡接收到数据包），“socket窗口”接收WM\_SOCKET\_NOTIFY消息，消息处理函数OnSocketNotify被调用，。“socket窗口”的定义和消息处理是MFC实现的，其中OnSocketNotify函数定义如下：**

LRESULT CSocketWnd::OnSocketNotify(WPARAM wParam, LPARAM lParam)

{

CSocket::AuxQueueAdd(WM\_SOCKET\_NOTIFY, wParam, lParam);

CSocket::ProcessAuxQueue();

return 0L;

}

**在CSocket::ProcessAuxQueue();函数中回调CAsyncSocket的成员函数DoCallBack，DoCallBack调用事件处理函数OnReceive。**

int PASCAL CSocket::ProcessAuxQueue()

{

……………………//省略部分

if (pMsg->message == WM\_SOCKET\_NOTIFY)

{

CAsyncSocket::DoCallBack(pMsg->wParam, pMsg->lParam);

}

………………//省略部分

return nCount;

}

void PASCAL CAsyncSocket::DoCallBack(WPARAM wParam, LPARAM lParam)

{

……………………//省略部分

pSocket->OnReceive(nErrorCode);

**/\*pSocket类型是：CClientUDPSocket，因为类CClientUDPSocket继承了类 CAsyncSocket，而OnReceive在CAsyncSocket定义的虚函数，OnReceive在CClientUDPSocket中重新 做了实现，因此调用的时候会转到CClientUDPSocket中OnReceive执行。\*/**

}

void CClientUDPSocket::OnReceive(int nErrorCode)

{

……………………

case OP\_KADEMLIAHEADER:

{

// theStats.AddDownDataOverheadKad(length);

if (length >= 2)

Kademlia::CKademlia::ProcessPacket(buffer, length, ntohl(sockAddr.sin\_addr.S\_un.S\_addr), ntohs(sockAddr.sin\_port));

else

throw CString(\_T("Kad packet too short"));

break;

}

……………………

}

**接着调用在kademlia.cpp中定义的函数ProcessPacket。**

void CKademlia::ProcessPacket(const byte \*pbyData, uint32 uLenData, uint32 uIP, uint16 uPort)

{

if( m\_pInstance && m\_pInstance->m\_pUDPListener )

m\_pInstance->m\_pUDPListener->ProcessPacket( pbyData, uLenData, uIP, uPort);

}

**转入KademliaUDPListener类中ProcessPacket函数运行。**

void CKademliaUDPListener::ProcessPacket(const byte\* pbyData, uint32 uLenData, uint32 uIP, uint16 uUDPPort)

{

//………………………………省略部分

switch (byOpcode)

{

………………………………//省略部分

case KADEMLIA\_RES:

if (thePrefs.GetDebugClientKadUDPLevel() > 0)

DebugRecv("KADEMLIA\_RES", uIP, uUDPPort);

Process\_KADEMLIA\_RES(pbyPacketData, uLenPacket, uIP, uUDPPort);

break;

………………………………//省略部分

}

}

**转入函数Process\_KADEMLIA\_RES(pbyPacketData, uLenPacket, uIP, uUDPPort);执行：**

**void CKademliaUDPListener::Process\_KADEMLIA\_RES (const byte \*pbyPacketData, uint32 uLenPacket, uint32 uIP, uint16 uUDPPort) 【我拦截它就ok了】**

{

//……………………

if(CKademlia::GetPrefs()->GetRecheckIP())

{

FirewalledCheck(uIP, uUDPPort);

if (thePrefs.GetDebugClientKadUDPLevel() > 0)

DebugSend("KADEMLIA\_HELLO\_REQ", uIP, uUDPPort);

SendMyDetails(KADEMLIA\_HELLO\_REQ, uIP, uUDPPort);

}

if(::IsGoodIPPort(ntohl(uIPResult),uUDPPortResult))

{

pRoutingZone->Add(uIDResult, uIPResult, uUDPPortResult, uTCPPortResult, 0);

pResults->push\_back(new CContact(uIDResult, uIPResult, uUDPPortResult, uTCPPortResult, uTarget, 0));

}

}

}

CSearchManager::ProcessResponse(uTarget, uIP, uUDPPort, pResults);

}

在这个函数体内部主要包括对4个函数的调用，分别是：

SendMyDetails(KADEMLIA\_HELLO\_REQ, uIP, uUDPPort);

pRoutingZone->Add(uIDResult, uIPResult, uUDPPortResult, uTCPPortResult, 0);

pResults->push\_back(new CContact(uIDResult, uIPResult, uUDPPortResult, uTCPPortResult, uTarget, 0));

CSearchManager::ProcessResponse(uTarget, uIP, uUDPPort, pResults);

其中第一个函数是在判断自己在防火墙或者NAT之后重新发送本地节点信息的函数，包括重新得到的IP地址以及端口。

第二和第三个函数用来添加此节点作为联系人之一。

**第三个函数是将此消息转入到CSearchManager中相应处理响应的函数进行处理。**

void CSearchManager::ProcessResponse(const CUInt128 &uTarget, uint32 uFromIP, uint16 uFromPort, ContactList \*plistResults)

{

pSearch->ProcessResponse(uFromIP, uFromPort, plistResults);// pSearch是 CSearch类的指针

}

**进一步转入到pSearch->ProcessResponse(uFromIP, uFromPort, plistResults)中执行。**

void CSearch::ProcessResponse(uint32 uFromIP, uint16 uFromPort, ContactList \*plistResults)

{

// Not interested in responses for FIND\_NODE.

// Once we get a results we stop the search.

// These contacts are added to contacts by UDPListener.

if (m\_uType == NODE)

{

// Note we got an answer

**m\_uAnswers++;**

// We clear the possible list to force the search to stop.

// We do this so the user has time to visually see the results.

m\_mapPossible.clear();

delete plistResults;

// Update search on the GUI.

//IMPREVIEW theApp.emuledlg->kademliawnd->searchList->SearchRef(this);

return;

}

}

在这个函数内部我们将响应的节点数目增加一。

后面陆续接收到的消息处理流程与上述情形相似，只是对于不同的消息采取的响应以及动作并不相同。

# DH交换密钥建立加密连接流程

## 客户端到服务器的呼出连接

1、CServerSocket::ConnectTo(CServer\* server, bool bNoCrypt)

根据bNoCrypt，客户端设置里是否要求加密连接，以及服务器是否支持加密连接

是：SetConnectionEncryption(true, NULL, true);

否：SetConnectionEncryption(false, NULL, true);

**m\_StreamCryptState为ECS\_UNKNOWN 或 ECS\_NONE**

2、CEncryptedStreamSocket::SetConnectionEncryption(bool bEnabled, const uchar\* pTargetClientHash, bool bServerConnection)

/// snow :不是到服务器的连接，目标客户端的ID哈希值不是NULL,且目的是要启动加密连接

if (bEnabled && pTargetClientHash != NULL && !bServerConnection){

**m\_StreamCryptState = ECS\_PENDING;** ///snow:是outgoing connection，状态为连接等待

///snow start：建立混淆密钥 - Client A (Outgoing connection):

/// Sendkey: Md5(<UserHashClientB 16><MagicValue34 1><RandomKeyPartClientA 4>) 21

/// Receivekey: Md5(<UserHashClientB 16><MagicValue203 1><RandomKeyPartClientA 4>) 21

m\_pRC4SendKey = RC4CreateKey(md5.GetRawHash(), 16, NULL);

m\_pRC4ReceiveKey = RC4CreateKey(md5.GetRawHash(), 16, NULL);

}

3、CServerSocket:: Connect(CStringA(server->GetAddress()), nPort))🡪CEMSocket::Connect(LPCSTR lpszHostAddress, UINT nHostPort)🡪CEncryptedStreamSocket::Connect(lpszHostAddress, nHostPort);🡪 CAsyncSocketEx::Connect(LPCSTR lpszHostAddress, UINT nHostPort)🡪 connect(m\_SocketData.hSocket, lpSockAddr, nSockAddrLen)

向服务器发出连接请求

4、

## 客户端到客户端的呼出连接

## 客户端到客户端的呼入连接