3. Client TCP – Implementare MFC

Unul dintre cel mai des intâlnite protocoale de comunicare pe Internet este TCP (en. „Transmission Control Protocol”). Acesta, împreună cu protocolul IP (en. „Internet Protocol”), formează nivelul de comunicare a majorității aplicațiilor destinate Internet. În cadrul acestui capitol vom prezenta elementele de bază necesare construirii unui client TCP care se conectează la un server TCP şi schimbă mesaje text cu acesta.

3.1 Cerințele aplicației client TCP-MFC

O abordare corectă în proiectarea şi implementarea unei aplicații reprezintă stabilirea într-o primă fază a cerințelor. Din acest motiv, în cadrul acestui sub-capitol vom trece în revistă cerințele principale ce trebuie considerate în proiectarea şi implementarea unei aplicații client TCP ce utilizează MFC ca interfață grafică şi componentă de comunicare.

Aplicația client are ca scop final schimbarea de mesaje text cu o aplicație server. În cadrul acestui capitol nu vom considera conectariea mai multor clienți, întrucât aceste aspecte sunt tratate mult mai detaliat în capitolele următoare. Fiind vorba de o comunicație text între o aplicație client cu o aplicație server nu este necesară alegerea unui client destinație căruia să i se transmită mesajele. Această simplificare permite stabilirea unui set redus de cerințe:

- Conectarea la un server TCP ce ascultă pe un anumit port stabilit în prealabil;
- Transmiterea şi recepționarea mesajelor text;
- Interfață grafică ce permite introducerea textelor ce sunt transmise (utilizând arhitectura MFC construirea unei asemenea interfețe este foarte uşoară);
- Interfață grafică ce permite afişarea textelor recepționate.

Pe lângă aceste cerințe, din punctul de vedere a modalității de utilizare a soclurilor, se cere utilizarea `CAsyncSocket`, o clasă MFC construită peste soclurile Win32 ce asigură toate funcționalitățile necesare utilizării soclurilor.

Pentru testarea aplicației client, întrucât aplicația server va fi construită în capitolele următoare, vom utiliza un server TCP existent – *Hercules* – dezvoltat de HW-Group, ce poate fi descărcat de aici: http://www.hw-group.com/products/hercules/index_en.html. De fapt, Hercules este un utilitar ce asigură o serie de terminale printre care se numără şi terminale Client/Server TCP. Mai multe despre acest terminal în sub-capitolele următoare.

3.2 Arhitectura aplicației

Pentru a satisface cerințele enumerate în sub-capitolul anterior, vom utiliza o arhitectură client bazată pe o fereastră dialog, denumirea clasei asociate ferestrei fiind `CMyNetClientDlg`. Utilizând Microsoft Visual Studio 2005, vom crea un nou proiect

MFC, *MFC Application*, *Dialog Based*, cu opțiunea *Windows sockets* selectat (*Advanced Features*). Întrucât nu vom folosi caractere *Unicode*, se va deselecta *Use Unicode libraries* (*Application Type*). În continuare, vom considera că denumirea proiectului este *MyNetClient*, exemplele fiind bazate pe această denumire.

Prin selectarea opțiunii *Windows sockets*, în implementarea clasei aplicației (i.e. `CMyNetClientApp`) se apelează funcția `AfxSocketInit()` ce asigură inițializarea soclurilor Windows pentru aplicația nou creată.

Pentru utilizarea soclurilor vom apela la clasa `CMyClientSocket` ce moştenește clasa `CAsyncSocket`. În cadrul acestei arhitecturi, `CMyClientSocket` este instanțiat din `CMyNetClientDlg`. Această arhitectură este ilustrată prin diagrama de clasă din figura 3.1.

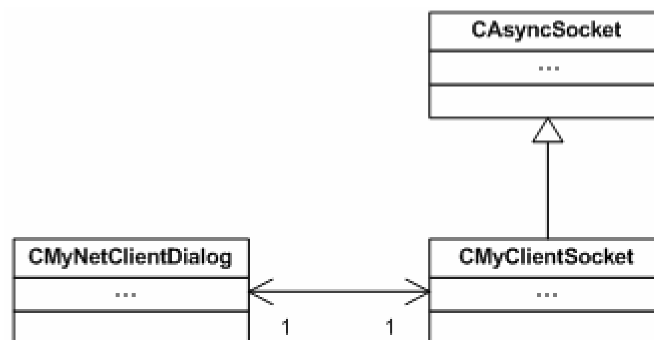

Figura 3.1 O parte din diagrama de clase a aplicației Client TCP-MFC

Rolul clasei `CMyNetClientDlg` este acela de a asigura implementarea funcționalității ferestrei dialog. Aceasta va trebui să asigure tratarea evenimentelor de apăsare a butoanelor, tratarea evenimentelor de introducere a textelor ce trebuie transmise şi de afişare a textelor recepționate. Totodată, această clasă trebuie să instanțieze clasa `CMyClientSocket`, responsabilă de administrarea conexiunii cu serverul.

Clasa `CAsyncSocket` pune la dispoziție un soclu fără blocare, ceea ce înseamnă că toate apelurile efectuate nu se blochează. O asemenea paradigmă de programare asigură returnarea controlului imediat aplicației, fără a bloca alte mecanisme vitale acesteia. De menționat este faptul că `CAsyncSocket` va beneficia de o coadă de mesaje Windows ce va fi procesată pe firul de execuție a aplicației. Din acest motiv, blocarea procesării mesajelor va duce la blocarea întregii aplicații. Pentru utilizatorii ce doresc un soclu cu blocare, există clasa CSocket ce moştenește clasa `CAsyncSocket`.

3.3 Construirea interfeței grafice

Utilizând editorul de ferestre dialog se construiește interfața grafică din figura 3.2. Pentru fiecare căsuță de editare se ataşează o variabilă membru prin clic dreapta pe controlul respectiv şi selectarea opțiunii *Add Variable...* . Interfața este împărțită în două părți: partea de mesagerie şi partea de control a conexiunii.

Partea de control a conexiunii permite introducerea adresei serverului sub forma unei adrese IP sau a unui nume de domeniu precum şi introducerea portului pe care ascultă serverul. Pentru controlul de editare adresă vom considera variabila membru `m_sAdress` de tipul `CString`. Pentru controlul de editare port vom considera variabila

membru `m_nPort` de tipul `UINT`. Desigur că pentru valoarea portului ar fi fost suficient să folosim o variabilă de tipul `unsigned short`, însă acest tip nu este disponibil pentru variabile ataşate controalelor, iar o variabilă de tipul `short` nu ar fi fost deloc corespunzătoare.

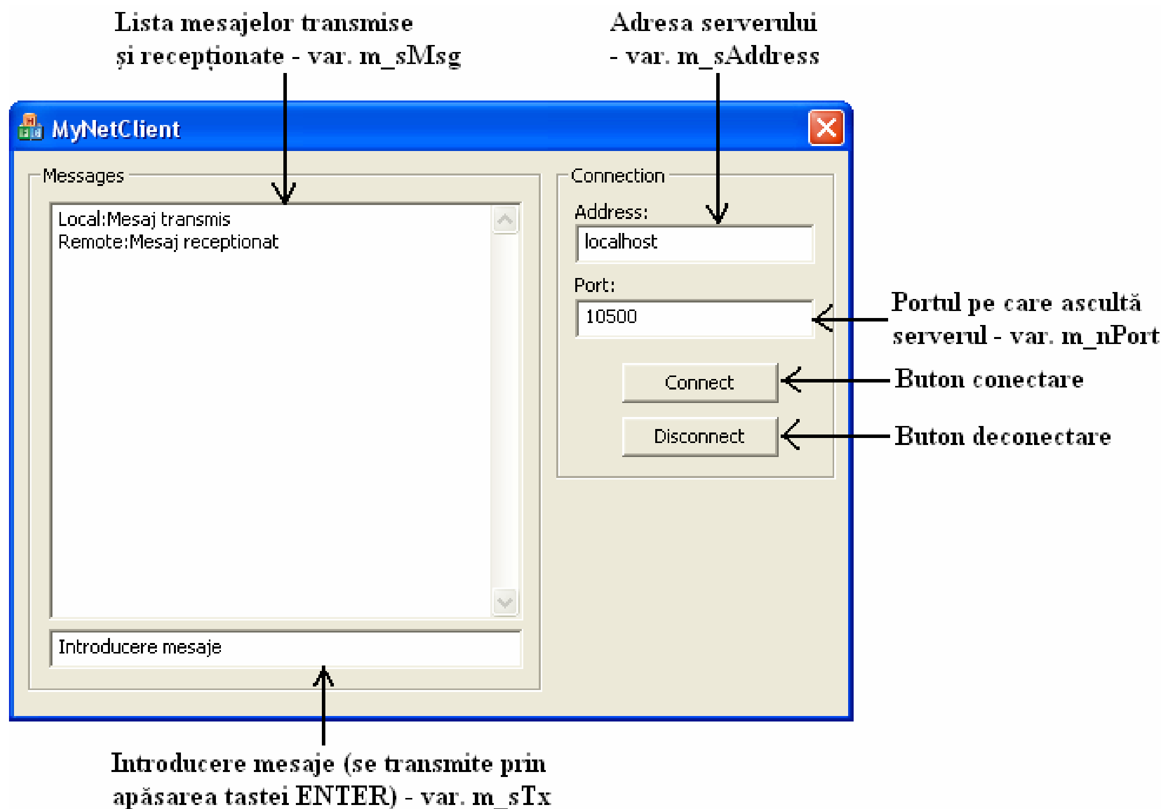

Figura 3.2 Interfaţa grafică a aplicaţiei Client TCP-MFC şi variabilele membru ataşate

Totodată, de partea controlului conexiunii interfaţa asigură şi două butoane, unul pentru stabilirea conexiunii şi altul pentru întreruperea conexiunii. Prin utilizarea acestor butoane se asigură un control rapid al conexiunii.

Partea de mesagerie conţine două componente: o componentă de listare a mesajelor transmise şi recepţionate şi o componentă de introducere a mesajelor. Prima componentă reprezintă un istoric al mesajelor transmise dar şi al celor recepţionate. Acesta este tot un control de editare, la fel ca şi controalele de adresă şi port, însă permite afişarea mai multor linii prin activarea proprietăţii *Multiline*. Variabila membru ataşată este `m_sMsg`, de tipul `CString`. A doua componentă asigură introducerea mesajelor text ce sunt transmise. Variabila membru ataşată este `m_sTx`, de tipul `CString`.

Valorile iniţiale ale variabilelor membru pot fi stabilite în cadrul constructorului clasei `CMyNetClientDlg`, de exemplu:

```
CMyNetClientDlg::CMyNetClientDlg(CWnd* pParent /*=NULL*/)
	: CDialog(CMyNetClientDlg::IDD, pParent),
	, m_sMsg(_T(""))
	, m_sTx(_T(""))
	, m_sAdress(_T("localhost"))
	, m_nPort(10500)
```

3.4 Construirea clasei de utilizare a soclului

Adăugarea unei noi clase de utilizare a soclului client se realizează prin clic dreapta asupra soluției și selectarea opțiunii *Add* urmată de selectarea opțiunii *Class*. Categoria clasei se alege *MFC*, cu șablonul (en. „Template”) *MFC Class*. Denumirea clasei se alege `CMyClientSocket` cu clasa de bază `CAsyncSocket`.

Clasa nou creată trebuie să fie capabilă să creeze conexiunea cu serverul, să transmită și să recepționeze mesaje. Din fericire, clasa de bază `CAsyncSocket` ascunde toate operațiile ce trebuie efectuate asupra soclurilor Windows pentru stabilirea unei conexiuni. Tot ceea ce trebuie să facă programatorul este să apeleze funcțiile puse la dispoziție și să suprascrie metodele virtuale pentru tratare a evenimentelor.

Legarea clasei `CMyClientSocket` de clasa `CMyNetClientDlg` se realizează printr-un pointer transmis ca parametru constructorului:

```
CMyClientSocket( CMyNetClientDlg* pDlg );
```

iar în antetul clasei, se declară o variabilă membru privată în care se stochează adresa instanței:

```
private:
      CMyNetClientDlg* m_pDlg;
```

În cadrul destructorului clasei `CMyClientSocket` vom distruge soclul creat:

```
CMyClientSocket::~CMyClientSocket ( void )
{
      CAsyncSocket::ShutDown( 2 );
      CAsyncSocket::Close();
}
```

Pentru a proteja soclul deja creat, vom folosi o variabilă booleană privată `m_bStarted` ce va indica dacă soclul a fost creat sau nu, cu valoarea inițială fals, dată în constructorul clasei. Constructorul rezultat care cuprinde și inițializarea celor două variabile este următorul:

```
CMyClientSocket::CMyClientSocket( CMyNetClientDlg* pDlg ):
      m_bStarted( false ),
      m_pDlg( pDlg )
{
}
```

În continuare, vom adăuga trei metode publice la această clasă, prin clic dreapta asupra denumirii clasei (în *Class View*) și selectarea opțiunii *Add*, urmată de selectarea opțiunii *Add Function...* . Metodele vor asigura conectarea la server și transmisia unui mesaj. Prototipul acestora este următorul:

```
bool connectToServer( const _TCHAR* pszHost,
                      const unsigned short nPort );
bool sendMessage( const _TCHAR* pszMessage );
```

```
bool isConnected( void );
```

În corpul metodei `connectToServer()` vom crea soclul şi vom apela metoda de conectare din clasa de bază. Crearea soclului TCP se realizează prin apelul metodei `Create()`, cu prototipul:

```
BOOL Create(
   UINT nSocketPort = 0,
   int nSocketType = SOCK_STREAM,
   long lEvent = FD_READ|FD_WRITE|FD_OOB|FD_ACCEPT|FD_CONNECT|FD_CLOSE,
   LPCTSTR lpszSocketAddress = NULL );
```

Primul parametru, `nSocketPort` reprezintă numărul portului la care se leagă soclul. Dacă valoarea acestui parametru este 0, atunci numărul portului este asignat automat – aceasta fiind abordarea în cazul soclurilor client. Al doilea parametru, `nSocketType` denotă tipul soclului transport, valorile suportate fiind `SOCK_STREAM` (pentru TCP) şi `SOCK_DGRAM` (pentru UDP). Următorul parametru, `lEvent` denotă evenimentele pentru care aplicaţia client doreşte să fie notificată, iar ultimul parametru, `lpszSocketAddress` reprezintă adresa de reţea la care să fie legat soclul creat.

Pentru soclul client vom utiliza valorile implicite ale funcţiei `Create()`, astfel:

```
if ( !CAsyncSocket::Create() ) {
      return false;
}
```

Metoda de conectare are două forme, cea pe care o vom folosi în continuare are următorul prototip:

```
BOOL Connect( LPCTSTR lpszHostAddress, UINT nHostPort );
```

unde primul parametru reprezintă adresa staţiei dat sub forma unui şir de caractere, iar al doilea parametru reprezintă numărul portului pe care ascultă serverul. În cazul în care această funcţie returnează o valoare ne-nulă, conexiunea s-a realizat cu succes. Altfel, trebuie tratat codul de eroare returnat:

```
if ( CAsyncSocket::Connect( pszHost, nPort ) ) {
      AfxMessageBox( _T("Connection successfull"),
                     MB_OK | MB_ICONINFORMATION );
            return ( m_bStarted = true );
}
else {
      int nError = CAsyncSocket::GetLastError();
      if ( WSAEWOULDBLOCK == nError ) {
            return ( m_bStarted = true );
      }
      else {
            return false;
      }
}
```

Codul de eroare poate să indice că nu a fost o eroare, dar apelul s-ar fi blocat (i.e. WSAEWOULDBLOCK), caz în care aplicația va fi notificată prin apelul metodei virtuale OnConnect de succesul sau eşuarea stabilirii conexiunii. Pentru ca aplicația să fie notificată de acest eveniment, ea trebuie să suprascrie metoda OnConnect() din clasa de bază. Suprascrierea acestei metode se realizează prin selectarea opțiunii *Overrides* din cadrul meniul de proprietăți a clasei urmată de selectarea metodei OnConnect. *Overrides* conține lista metodelor ce pot fi suprascrise.

În cadrul acestei metode se tratează codul de eroare transmis, pentru valoarea 0, conexiunea s-a realizat cu succes şi se poate afişa un mesaj corespunzător:

```
void CMyClientSocket::OnConnect( int nErrorCode )
{
      if ( nErrorCode ) {
            AfxMessageBox( _T("Connection ERROR"),
                           MB_OK | MB_ICONERROR );
      }
      else {
            AfxMessageBox( _T("Connection successfull"),
                           MB_OK | MB_ICONINFORMATION );
      }

      CAsyncSocket::OnConnect(nErrorCode);
}
```

A doua funcție membru adăugată este sendMessage() ce asigură transmisia unui mesaj. Această funcție verifică dacă soclul şi conexiunea au fost create în prealabil şi apelează funcția Send() a clasei de bază, cu prototipul:

```
virtual int Send( const void* lpBuf, int nBufLen, int nFlags = 0 );
```

Primul parametru al funcției Send(), lpBuf reprezintă un pointer către bufferul ce conține mesajul transmis, iar al doilea parametru, nBufLen reprezintă numărul de octeți din buffer ce trebuie transmişi. De menționat că această funcție nu este dedicată transmisiei şirurilor de caractere ci este construită pentru transmisia unei secvențe de octeți aleatori. Ultimul parametru reprezintă opțiuni de transmisie referitoare la rutarea mesajului, ceea ce nu reprezintă interes pentru aplicația client TCP, motiv pentru care poate fi ignorat.

Funcția Send() returnează numărul de octeți transmişi sau SOCKET_ERROR în caz de eroare, motiv pentru care aplicația client trebuie să ia în considerare şi acest aspect şi să re-apeleze această funcție cu numărul de octeți rămaşi a fi transmişi. În cazul returnării valorii SOCKET_ERROR, pentru codul erorii WSAEINPROGRESS sau WSAEWOULDBLOCK trebuie încercată retransmisia pachetului întrucât apelul curent al funcției Send() ar fi dus la blocarea acesteia.

Codul rezultat pentru funcția sendMessage() este următorul:

```
if ( !m_bStarted ) {
      return false;
}
const int nLen = _tcslen( pszMessage );
const int nRet = CAsyncSocket::Send( pszMessage, nLen );
```

```
if ( SOCKET_ERROR == nRet )
{
      int nError = GetLastError();
      if ( ( WSAEINPROGRESS == nError ) ||
           ( WSAEWOULDBLOCK == nError ) ) {
            // Retransmisie pachet
            …
      }
      else {
            AfxMessageBox( _T("Error on send"), MB_OK | MB_ICONERROR );
            return false;
      }
}
else {
      if ( nLen != nRet ) {
            // Transmiterea octeților rămaşi
            …
      }
      else {
            // Transmisie cu succes a tuturor octeților
            return true;
      }
}
```

În exemplul anterior pot exista şi cazuri în care nu toți octeții sunt transmişi la un singur apel. În această situație, octeții rămaşi trebuie transmişi printr-un nou apel al funcției `Send()` până când se transmit toate datele.

Pentru recepționarea mesajelor, în clasa `CMyClientSocket` vom suprascrie metoda virtuală `OnReceive()`. Apelul acestei metode semnalează că există date disponibile ce pot fi citite, sau a avut loc o eroare. Citirea efectivă a datelor disponibile se va realiza prin apelul metodei `Receive()` din clasa de bază, având următorul prototip:

```
virtual int Receive( void* lpBuf, int nBufLen, int nFlags = 0 );
```

Primul parametru, `lpBuf` reprezintă un pointer către adresa unui buffer în care sunt transferați octeții recepționați. Al doilea parametru, `nBufLen` reprezintă dimensiunea bufferului în octeți. Al treilea parametru reprezintă un set de opțiuni printre care se numără `MSG_PEEK` ce asigură copiera octeților fără ştergerea lor din coada de mesaje. Implicit (i.e. pentru `nFlags = 0`), octeții transferați sunt şterşi din coada de mesaje.

După citirea octeților, aceştia trebuie transferați interfeței grafice, pentru care implementăm metoda `OnRecvMessage()` în clasa `CMyNetClientDlg`. Metoda `OnReceive()` rezultată este următoarea:

```
void CMyClientSocket::OnReceive( int nErrorCode )
{
      if ( !nErrorCode )
      {
            // Citim maxim 2048 de octeți, ultimul octet
            // este rezervat pentru 0
            _TCHAR pBuf[ 2049 ];
            const int nRet = CAsyncSocket::Receive( pBuf, 2048 );
            if ( SOCKET_ERROR == nRet ) {
                  AfxMessageBox( _T("Receive ERROR"),
```

```
                              MB_OK | MB_ICONERROR );
        }
        else {
              if ( nRet > 0 ) {
                    // Terminarea şirului de caractere
                    pBuf[ nRet ] = _T('\0');

                    // Apelul metodei de afişare din
                    // interfaţa grafică
                    m_pDlg->OnRecvMessage( pBuf );
              }
        }
  }

  CAsyncSocket::OnReceive( nErrorCode );
}
```

A treia metodă ce trebuie adăugată este `isConnected()`, ce va returna valoarea `true` pentru un soclu conectat cu succes. Adăugarea codului aferent sunt lăsate ca exerciţiu pentru cititor.

3.5 Legarea claselor

Clasa `CMyClientSocket` este instanţiată din clasa `CMyNetClientDlg`, după cum se oate observa şi din figura 3.1. Pentru aceasta, se declară o variabilă membru privată în `MyNetClientDlg.h`, iniţializată cu `NULL` în constructorul clasei `CMyNetClientDlg`.

```
private:
      CMyClientSocket* m_pSocket;
```

În destructorul clasei `CMyNetClientDlg` trebuie să ne asigurăm că obiectul `m_pSocket` este distrus:

```
CMyNetClientDlg::~CMyNetClientDlg( void )
{
      if ( NULL != m_pSocket ) {
            // Soclul este închis în
            // destructorul obiectului
            delete m_pSocket;
      }
}
```

Obiectul `m_pSocket` este creat la apăsarea butonului *Connect*, moment în care se apelează şi funcţia `connectToServer()` cu adresa şi portul completate:

```
if ( NULL == m_pSocket ) {
      m_pSocket = new MeClientSocket( this );
}

// Transferul valorilor din controale în variabile membru ataşate
UpdateData( TRUE );

if ( !m_pSocket->connectToServer( m_sAdress, m_nPort ) {
      AfxMessageBox( _T("Connection ERROR"),
```

```
                    MB_OK | MB_ICONERROR );
}
```

La apăsarea butonului *Disconnect* se distruge obiectul `m_pSocket`:

```
if ( NULL != m_pSocket ) {
       delete m_pSocket;
       m_pSocket = NULL;
 }
```

Transmisia mesajelor se realizează la apăsarea tastei ENTER. La apăsarea acestei taste, în cazul ferestrelor dialog se apelează automat metoda `OnOK()`, care la rândul său apelează funcțiile de distrugere a ferestrei. Pentru a preveni acest lucru şi pentru a asigura că la apăsarea tastei ENTER se transmite mesajul introdus vom suprascrie metoda `OnOK()`, având prototipul:

```
public:
     void OnOK();
```

Transmisia mesajului trebuie să se realizeze numai dacă următoarele sunt îndeplinite:

- Controlul activ pe care este poziționat cursorul este controlul de introducere a mesajelor ce trebuie transmise;
- Obiectul `m_pSocket` a fost creat în prealabil;
- Există o conexiune validă.

Verificarea primei cerințe se realizează prin utilizarea funcției `GetFocus()` care returnează adresa controlului pe care este poziționat cursorul. Această adresă este comparată cu adresa controlului de editare, determinată prin apelul funcției `GetDlgItem()` căreia i se transmite ID-ul controlului. Codul rezultat este următorul:

```
const CWnd* pfwnd = GetFocus();
if ( NULL == pfwnd ) {
      return;
}

// IDC_EDIT2 reprezintă ID-ul controlului de
// introducere a mesajelor ce trebuie transmise
const CWnd* pewnd = GetDlgItem( IDC_EDIT2 );
if ( pfwnd != pewnd ) {
      return;
}
```

Verificarea următoarelor cerințe se realizează foarte simplu:

```
if ( ( NULL == m_pSocket ) || ( !m_pSocket->isConnected() ) ) {
       return;
}
```

În continuare, mesajul introdus este transmis şi este adăugat la lista mnesajelor transmise, prin adăugarea în prealabil a textului *Local:* pentru a diferenția mesajele transmise de cele recepționate. La mesajele recepționate se adaugă textul *Remote:*. Totodată, textul din controlul de introducere a mesajului este şters, codul rezultat fiind următorul:

```
// Transmisia mesajului
UpdateData( TRUE );
if ( !m_pSocket->sendMessage( m_sTx ) ) {
      return;
}
// Adăugarea la lista de mesaje transmise
m_sMsg += _T("Local:");
m_sMsg += m_sTx;

// Ştergerea mesajului transmis
m_sTx = _T("");
UpdateData( FALSE );

// Scroll la ultima linie din lista mesajelor transmise
// IDC_EDIT1 este ID-ul controlului ce conține lista mesajelor
// transmise şi recepționate
CEdit* pReadWnd = (CEdit*)GetDlgItem( IDC_EDIT1 );
pReadWnd->LineScroll( pReadWnd->GetLineCount() );
```

Ceea ce a rămas de detaliat este metoda publică `OnRecvMessage()` ce trebuie adăugată la clasa `CMyNetClientDlg`. Apelul acestei metode se realizează din metoda `OnReceive()` a clasei `CMyClientSocket` la recepționarea unui mesaj text. Prototipul metodei `OnRecvMessage()` este:

```
void OnRecvMessage( const _TCHAR* pszMsg );
```

Singura operație ce trebuie realizată în definiția acestei metode reprezintă adăugarea mesajului text la lista de mesaje recepționate împreună cu un delimitator linie nouă, precum şi execuția unui scroll a ferestrei asupra ultimei linii adăugate:

```
m_sMsg += _T("Remote:");
m_sMsg += pszMsg;
m_sMsg += _T("\r\n");
UpdateData( FALSE );

CEdit* pwnd = (CEdit*)GetDlgItem( IDC_EDIT1 );
pwnd->LineScroll( pwnd->GetLineCount() );
```

3.6 Testarea apicației client TCP-MFC

Pentru testarea aplicației vom utiliza utilitarul *Hercules*. Acesta asigură o serie de terminale printre care se numără şi terminalele server TCP. După cum se poate observa din figura 3.3, la pornire utilitarul Hercules permite selectarea unui terminal, pentru testarea aplicației client TCP vom selecta tab-ul *TCP Server*. În controlul de editare *Port*

se introduce portul pe care va asculta serverul, în cazul de faţă se va introduce valoarea 10500 întrucât aceasta a fost valoarea dată ca exemplu în cazul clientului TCP.

Textul recepţionat este afişat în controlul de editare *Received data*. În controlul *Sent data* se introduce textul transmis, cu menţiunea că fiecare caracter este transmis separat. Pentru a transmite mai multe caractere deodată, se va utiliza secţiunea *Send* cu butonul *Send*.

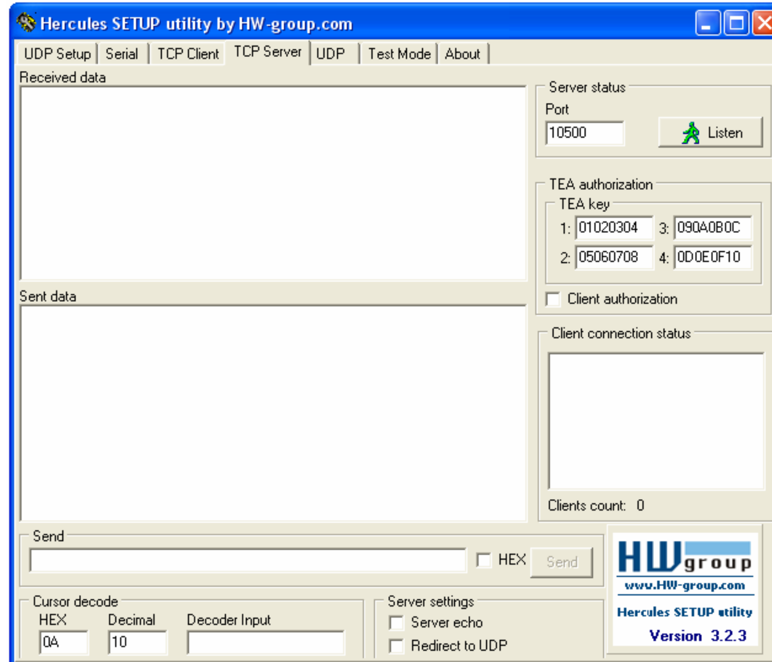

Figura 3.3 Utilitarul *Hercules* – modulul Server TCP

Exerciţiu.

Să se implementeze o aplicaţie client TCP utilizând MFC, folosind CAsyncSocket. Aplicaţia se va conecta la un server TCP căruia îi va transmite mesaje text şi de la care va recepţiona mesaje text. Testarea aplicaţiei se va realiza utilizând utilitarul *Hercules* descris în cadrul acestui capitol.