# 5. Client DNS folosind UDP – Implementare MFC

În capitolele anterioare s-a ilustrat utilizarea TCP pentru implementarea unei comunicări între mai mulți clienți prin intermediul unei aplicații server şi mai multe aplicații client. Există însă situații când comunicarea este de scurtă durată (i.e. câteva secunde), necesită un schimb redus de mesaje. În aceste situații utilizarea TCP nu este necesară întrucât acesta vine cu un set de mesaje în plus pentru stabilirea conexiunii, algoritmii de fereastră glisantă, etc. În schimb, se poate utiliza UDP, ce nu necesită stabilirea unei conexiuni, nu necesită mesaje în plus pentru menținerea conexiunii și pentru închiderea conexiunii. Dezavantajul major al UDP, pierderea pachetelor, poate fi rezolvat prin implementarea unor mecanisme simple de retransmisie a pachetelor (R. Stevens, 1998). În continuarea acestui capitol şi a acestei lucrări vom utiliza termenul *datagramă* pentru a face referire la un pachet transmis pe UDP.

În cadrul acestui capitol vom utiliza UDP şi arhitectura MFC pentru implementarea unui client didactic DNS (en. „Domain Name System"). Serverele DNS pot fi implementate atât prin UDP cât şi prin TCP, simplitatea protocolului utilizat fiind ideală pentru implementarea mesajelor cerere-răspuns prin intermediul UDP. Rolul principal al unui server DNS este acela de a rezolva adresele IP pentru un nume de domeniu dat. În acest sens, clientul implementat va transmite serverului DNS un nume de domeniu, pentru care i se va returna o adresa IP sau o eroare. Testarea aplicației implementate se va realiza cu utilitarul *Hercules* ce asigură şi o comunicare prin UDP.

## 5.1 Cerințele aplicației client UDP-MFC

În continuare, vom trece în revistă cerințele unei aplicații client UDP ce trebuie să transmită cereri unui server DNS. Această aplicație va beneficia de o interfață grafică cu controale de editare pentru introducerea denumirii domeniului interogat şi pentru afişarea rezultatului primit de la server. Interfața grafică va asigura totdată informațiile necesare configurării soclului UDP.

Pe lângă această interfață grafică, vom utiliza arhitectura MFC pentru comunicarea UDP. Clasa utilizată este tot `CAsyncSocket`, o clasă ce permite realizarea unei comunicări UDP prin simpla modificare a parametrilor utilizați. În concluzie, cerințele principale ce trebuie satisfăcute de aplicația client UDP sunt următoarele:

- Interfață grafică pentru introducerea cererilor DNS şi afişarea răspunsurilor primite;
- Controale grafice pentru configurarea soclului UDP;
- Transmiterea şi recepționarea mesajelor UDP.

## 5.2 Arhitectura aplicației

Pentru a satisface cerințele enumerate, vom utiliza o arhitectură client bazată pe o fereastră dialog, denumirea clasei asociate ferestrei fiind `CMyDNSClientDlg`. 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 *MyDNSClient*.

Arhitectura generală a aplicației ilustrată printr-o diagramă de clase este ilustrată în figura 5.1. Funcționalitatea soclului UDP este încapsulată în clasa `CAsyncSocket`, moștenită de clasa `CMyDNSClientSocket` ce va asigura apelul metodelor de transmisie și recepționare. Clasa `CMyDNSClientSocket` este instanțiată din clasa `CMyDNSClientDlg` atașată ferestrei dialog.

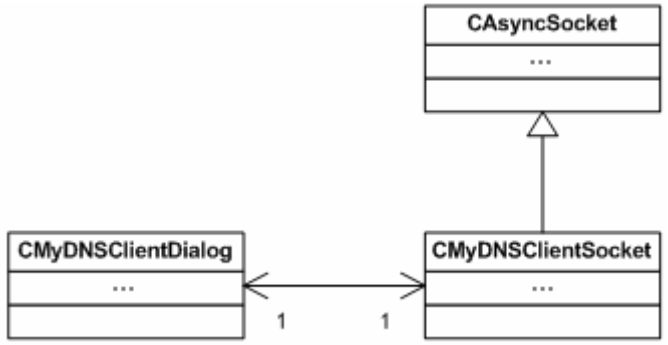


**Figura 5.1** O parte din diagrama de clase a aplicației Client UDP-MFC

## 5.3 Protocolul de comunicare Client-Server DNS

Protocolul utilizat în comunicarea dintre clientul construit și aplicația server este unul didactic, simplificat pentru a ilustra funcționalitatea unui server DNS. Mesajele transmise de client serverului au o singură componentă: denumirea domeniului interogat. Răspunsul primit este unul cu două sau trei componente, despărțite prin spațiu. Răspunsul cu două componente corespunde unei procesări cu succes a cererii, iar răspunsul cu trei componente corespunde unei erori.

În caz de succes, serverul DNS va returna:

- Denumirea domeniului interogat;
- Adresa IP corespunzătoare.

În caz de eroare, serverul DNS va returna:

- Denumirea domeniului interogat;
- Șirul de caractere „ERROR”;
- Eroarea sub forma unui șir de caractere fără spațiu. Eroarea returnată pentru cazul în care domeniul nu este cunoscut este „NotFound”.

## 5.4 Construirea interfeței grafice

Interfața grafică utilizată este ilustrată în figura 5.2. Aceasta include controale de introducere a cererilor transmise și de listare a răspunsurilor primite precum și controale pentru configurare soclului.

Utilizatorul va introduce datele legate de server, precum adresa IP (sau domeniul corespunzător), precum şi portul pe care rulează serverul. De regulă, serverele DNS rulează pe portul dedicat 53, însă acest port este unul configurabil. Pe lângă datele serverului, utilizatorul poate introduce şi numărul portului local sau poate selecta *Auto-assigned local port* prin care portul local este auto-asignat de către sistem. Pentru ca soclul creat să accepte datagrame doar de la sursa IP şi portul server configurat, se poate selecta *Connected socket*.

După ce datele de configurare a soclului au fost introduse, se apasă butonul de creare a soclului, *CreateSocket*. După apăsarea acestui buton, textul acestuia este schimbat cu *Close* pentru a semnala că o nouă apăsare va duce la închiderea soclului.

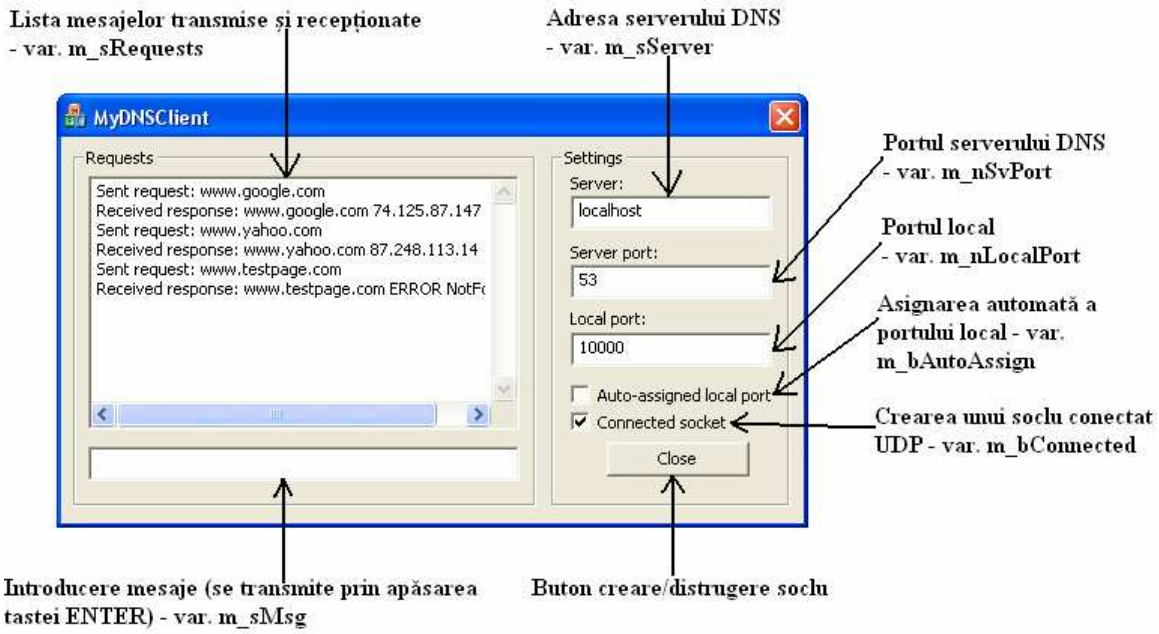


**Figura 5.2** Interfaţa grafică şi variabilele membru ataşate aplicaţiei Client UDP-MFC

## 5.5 Metodele CAsyncSocket utilizate pentru Client-UDP

Clasa `CAsyncSocket` pusă la dispoziţie de arhitectura MFC poate fi utilizată atât pentru o comunicare TCP cât şi pentru o comunicare UDP. Protocolul utilizat se specifică la crearea soclului prin apelul metodei `Create()`, prezentată în capitolele anterioare:

```
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 );
```

Pentru a crea un soclu UDP, valoarea celui de-al doilea parametru al funcţiei `Create()`, `nSocketType` va avea valoarea `SOCK_DGRAM`. Primul parametru al acestei funcţii, `nSocketPort` va asigura legarea soclului de un port. Dacă valoarea acestui parametru este diferită de 0, soclul este legat de portul dat de către utilizator (i.e. introdus

din interfața grafică), însă dacă valoarea lui este egală cu 0, valoarea portului este asignată automat de sistemul de operare.

De exemplu, pentru a crea un soclu UDP pe portul 10000 se va utiliza următorul apel:

```
if ( !Create( 10000, SOCK_DGRAM ) ) {
      return false;
}
```

Altfel, dacă se doreşte ca portul să fie asignat automat se va utiliza următorul apel:

```
if ( !Create( 0, SOCK_DGRAM ) ) {
      return false;
}
```

Cu toate că UDP nu este un protocol orientat conexiune, API-ul soclurilor permite realizarea unui soclu conectat în sensul că sunt acceptate datagrame doar de la sursa specificată, iar la transmisie datagramele sunt transmise aceleiaşi destinații. Altfel, dacă soclul nu este conectat, pentru fiecare datagramă se poate specifica o destinație diferită şi soclul va accepta datagrame de la orice sursă.

Utilizarea metodei de conectare este una opțională, utilizarea ei fiind similară cu cazul TCP-ului, singura diferență fiind că în acest caz metoda apelată nu se poate bloca. Cu alte cuvinte, o valoare returnată egală cu 0 echivalează cu eşuarea funcției, fără a fi necesară inspectarea codului de eroare:

```
if ( !Connect( “localhost”, 53 ) ) {
      return false;
}
```

Dacă soclul este unul conectat, metodele utilizate pentru transmisie şi recepționare sunt cele utilizate şi la TCP (i.e. `Send()` şi `Receive()`). Prin apelul metodei `Send()` se transmite o singură datagramă a cărei dimensiune nu este nelimitată. Față de TCP, în acest caz trebuie să avem grijă la dimensiunea datagramei pentru ca aceasta să nu depăşească dimensiunea maximă permisă. În plus, dacă o datagramă depăşeşte dimensiunea de 1400 octeți (această valoare poate varia în funcție de platformă), atunci aceasta va fi supusă fragmentărilor, motiv pentru care se recomandă ca fiecare datagramă să conțină maximum 1400 octeți – aceasta fiind şi o limitare adusă de nivelul legătură date în rețele Ethernet. Mai mult, dacă o datagramă este una destinată difuzării (en. „broadcast”) dimensiunea ei nu trebuie să depăşească 512 octeți. Prin apelul metodei `Receive()` se va recepționa o singură datagramă, iar dacă dimensiunea bufferului de recepționare este mai mică decât dimensiunea datagramei octeții care nu încap sunt pierduți, iar metoda returnează `SOCKET_ERROR` cu eroarea `WSAEMSGSIZE`.

Dacă soclul nu este unul conectat atunci se vor utiliza următoarele două metode pentru transmisie şi recepționare:

```
int SendTo( const void* lpBuf,
            int nBufLen,
            UINT nHostPort,
            LPCTSTR lpszHostAddress = NULL,
            int nFlags = 0 );
```

```
int ReceiveFrom( void* lpBuf,
                 int nBufLen,
                 CString& rSocketAddress,
                 UINT& rSocketPort,
                 int nFlags = 0 );
```

Metoda `SendTo()` asigură transmisia unei datagrame. Primul parametru `lpBuf` este un pointer la bufferul ce conține datele transmise, iar următorul parametru `nBufLen` reprezintă numărul de octeți ce trebuie transmiși din buffer. Parametrul `nHostPort` reprezintă portul destinație, iar `lpszHostAddress` este un pointer către un șir de caractere reprezentând adresa serverului. Ultimul parametru, `nFlags` permite specificarea opțiunilor de rutare pentru datagramă. În caz de succes, valoarea returnată reprezintă numărul octeților transmiși, pentru a indica o eroarea se returnează `SOCKET_ERROR`.

Metoda `ReceiveFrom()` asigură recepționarea unei datagrame. Primul parametru `lpBuf` este un pointer la bufferul în care se stochează octeții recepționați, iar următorul parametru `nBufLen` reprezintă dimensiunea bufferului. Parametrul `rSocketAddress` o referință către un obiect `CString` în care se va stoca adresa sursei, iar parametrul `rSocketPort` este o referință către o variabilă în care se stochează valoarea portului sursă. Ultimul parametru, `nFlags` permite specificarea unor opțiuni de transfer și prelucrare a datagramei, o posibilă opțiune fiind `MSG_PEEK` ce asigură transferul datagramei fără ștergerea acesteia din lista datagramelor recepționate. În caz de succes, valoarea returnată reprezintă numărul octeților transferați, pentru a indica o eroare se returnează `SOCKET_ERROR`. În caz de eroare trebuie interogat codul erorii prin apelul funcției `GetLastError()` întrucât este posibil ca bufferul dat ca parametru să nu fi fost destul de încăpător pentru a transfera întreaga datagramă (i.e. codul `WSAEMSGSIZE`).

Un exemplu de utilizare pentru cele două metode este dat în cele ce urmează:

```
// Transmisie pe un soclu neconectat
int nRet = SendTo( rMsg, rMsg.GetLength(), nDestPort, rDestHost );
if ( SOCKET_ERROR == nRet ) {
      return false;
}

// Recepționare pe un soclu neconectat
TCHAR pBuf[ 2049 ];
int nRet = 0;
CString sSrcAddr;
UINT nSrcPort = 0;
nRet = ReceiveFrom( pBuf, 2048, sSrcAddr, nSrcPort );
if ( SOCKET_ERROR == nRet ) {
      int nError = GetLastError();
      if ( WSAEMSGSIZE == nError ) {
            pBuf[ 2048 ] = _T('\0');
            … // Procesare datagramă
      }
      else {
            return false;
      }
}
```

```
else {
      // Pe UDP se pot transmite şi datagrame cu număr de octeţi de
      // date = 0, ce pot fi ignoraţi sau procesaţi, în funcţie de
      // specificul aplicaţiei
      if ( nRet > 0 ) {
            pBuf[ nRet ] = _T('\0');
            … // Procesare datagramă
      }
}
```

## 5.6 Construirea clasei de utilizare a soclului UDP

Pentru utilizarea soclului UDP vom moşteni clasa de încapsulare `CAsyncSocket` prin intermediul clasei `CMyDNSClientSocket`. Această clasă este instanțiată din `CMyDNSClientDlg` a cărei instanță este transmisă clasei `CMyDNSClientSocket` prin intermediul constructorului pentru a afișa mesajele recepționate pe interfața grafică.

Pentru crearea soclului vom adăuga o metodă publică utilizată pentru toate cazurile (i.e. conectat, neconectat, port stabilit de utilizator sau ales automat) cu prototipul:

```
bool createSocket( const CString& rServer,
                   const unsigned short nSvPort,
                   const unsigned short nLocalPort,
                   const bool bConnected );
```

Primul parametru, `rServer` reprezintă o referință către un obiect `CString` ce conține adresa serverului. Al doilea parametru, `nSvPort` reprezintă portul serverului. Al treilea parametru, `nLocalPort` reprezintă valoarea portului local. Ultimul parametru, `bConnected` reprezintă tipul soclului creat: conectat sau neconectat. Metoda va returna valoarea `true` pentru o execuție cu succes.

Dacă se dorește crearea unui soclu conectat, atunci valoarea variabilei `bConnected` va fi `true`. Pentru `nLocalPort=0` portul va fi ales automat de sistemul de operare. În implementarea acestei metode vom utiliza o variabilă membru `m_bIsCreated` pentru a identifica starea soclului, având valoarea inițială `false`. Astfel vom preveni apelurile multiple ale metodei de creare a soclului. Starea de conectare sau neconectare a soclului este reținută într-o variabilă membru `m_bConnected` cu valoare inițială `false`.

Codul rezultat pentru această metodă este următoarea:

```
if ( m_bIsCreated ) {
      return false;
}

m_bConnected = bConnected;

if ( !CAsyncSocket::Create( nLocalPort, SOCK_DGRAM ) ) {
      return false;
}

if ( bConnected )
{
      if ( !CAsyncSocket::Connect( rServer, nSvPort ) ) {
```

```
            return false;
      }
}

return ( m_bIsCreated = true );
```

Închiderea soclului se va realiza în destructorul clasei:

```
CMyDNSClientSocket::~ CMyDNSClientSocket ()
{
      ShutDown( 2 );
      Close();
}
```

După crearea soclului, aplicația ce utilizează această clasă va utiliza aceeaşi metodă pentru transmiterea datagramelor atât în cazul conectat cât şi în cel neconectat. Metoda publică adăugată are prototipul:

```
bool sendDatagram( const CString& rMsg,
                   const CString& rDestHost,
                   const unsigned short nDestPort );
```

Primul parametru, `rMsg` reprezintă o referință către un obiect de tipul `CString` ce conține mesajul transmis sub forma unui şir de caractere. Al doilea parametru, `rDestHost` reprezintă o referință către un obiect de tipul `CString` ce conține adresa destinație. Ultimul parametru reprezintă portul destinație. Metoda va returna valoarea `true` pentru o execuție cu succes.

În cazul în care soclul este conectat, primii doi parametrii nu sunt utilizați. Codul rezultat este următorul:

```
if ( !m_bIsCreated ) {
      return false;
}

int nRet = 0;
if ( m_bConnected ) {
      nRet = CAsyncSocket::Send( rMsg, rMsg.GetLength() );
}
else {
      nRet = CAsyncSocket::SendTo( rMsg,
                                   rMsg.GetLength(),
                                   nDestPort,
                                   rDestHost );
}
if ( SOCKET_ERROR == nRet ) {
      return false;
}

return true;
```

Recepționarea unei datagrame se realizează prin apelul metodei virtuale `OnReceive()`, pe care o suprascriem în această clasă utilizând metoda descrisă în capitolele anterioare. În cadrul metodei suprascrise vom trata atât cazul conectat cât şi cel

neconectat. Totodată, vom considera metoda `OnReadSocket()` implementată în cadrul clasei `CMyDNSClientDlg` atașată interfeței grafice, unde se asigură afișarea mesajului recepționat, apelată pentru fiecare datagramă recepționată. Metoda `OnReceive()` suprascrisă rezultată este:

```
void CMyDNSClientSocket::OnReceive( int nErrorCode )
{
	if ( !nErrorCode )
	{
		_TCHAR pBuf[ 2049 ];
		int nRet = 0;
		CString sSrcAddr;
		UINT nSrcPort = 0;
		if ( m_bConnected ) {
			nRet = CAsyncSocket::Receive( pBuf, 2048 );
		}
		else {
			nRet = CAsyncSocket::ReceiveFrom( pBuf,
							2048,
							sSrcAddr,
							nSrcPort );
		}

		if ( SOCKET_ERROR == nRet ) {
			int nError = GetLastError();
			if ( WSAEMSGSIZE == nError ) {
				pBuf[ 2048 ] = _T('\0');
				m_pDlg->OnReadSocket( pBuf );
			}
			else {
				AfxMessageBox( _T("Receive ERROR"),
						MB_OK | MB_ICONERROR );
			}
		}
		else {
			if ( nRet > 0 ) {
				pBuf[ nRet ] = _T('\0');
				m_pDlg->OnReadSocket( pBuf );
			}
		}
	}

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

## 5.7 Legarea claselor

Clasa `CMyDNSClientSocket` este instanțiată în cadrul clasei `CMyDNSClientDlg` şi este distrusă la apăsarea butonului *Close* sau în destructorul clasei `CMyDNSClientDlg`. La apăsarea tastei *CreateSocket/Close* se va crea un nou soclu sau se va închide soclul deja existent. Dacă considerăm ID-ul butonului `IDC_BUTTON1` şi variabilele membru prezentate în figura 5.2, codul rezultat ce se execută la apăsarea butonului pentru ambele cazuri este următorul:

```
CWnd* pwnd = GetDlgItem( IDC_BUTTON1 );
if ( NULL == pwnd ) {
      return;
}

CString sText;
pwnd->GetWindowTextA( sText );
if ( sText == _T("CreateSocket") )
{
      if ( NULL == m_pSocket ) {
            m_pSocket = new CMyUDPSocket( this );
      }

      UpdateData( TRUE );
      if ( !m_pSocket->createSocket( m_sServer,
                                     m_nSvPort,
                                     m_bAutoAssign? 0:m_nLocalPort,
                                     m_bConnected ) ) {
            AfxMessageBox( "Error creating socket!",
                           MB_OK | MB_ICONERROR );
            delete m_pSocket;
            m_pSocket = NULL;
      }
      else {
            AfxMessageBox( "Socket created successfully!",
                           MB_OK | MB_ICONINFORMATION );
            pwnd->SetWindowTextA( _T("Close") );
      }
}
else {
      if ( NULL != m_pSocket ) {
            delete m_pSocket;
            m_pSocket = NULL;
      }
      pwnd->SetWindowTextA( _T("CreateSocket") );
}
```

Transmiterea unei noi cereri sub forma unei datagrame se realizează prin apelul metodei `sendDatagram()` a clasei `CMyDNSClientSocket`. Cererea este transmisă la apăsarea tastei ENTER, când controlul de introducere a mesajelor este activ, cererea fiind adăugată la lista cererilor. Interogarea controlului activ a fost prezentat în capitolele anterioare, motiv pentru care în continuare vom prezenta doar secvența de cod pentru transmiterea unei cereri serverului DNS:

```
      if ( NULL == m_pSocket ) {
            return;
      }

      UpdateData( TRUE );

      if ( !m_pSocket->sendMessage( m_sMsg, m_sServer, m_nSvPort ) ) {
            AfxMessageBox( "Error sending message!",
                           MB_OK | MB_ICONERROR );
            return;
      }
```

```
m_sRequests += _T("Sent request: ");
m_sRequests += m_sMsg;
m_sRequests += _T("\r\n");
m_sMsg = _T("");

UpdateData( FALSE );
```

La recepționarea răspunsului, din clasa `CMyDNSClientSocket` se apelează automat metoda publică `OnReadSocket()` ce trebuie adăugată la clasa `CMyDNSClientDlg`, cu prototipul:

```
void OnReadSocket( const CString& rMsg );
```

În definiția acestei metode, răspunsul recepționat este adăugat la lista mesajelor:

```
m_sRequests += _T("Received response: ");
m_sRequests += rMsg;
m_sRequests += _T("\r\n");

UpdateData( FALSE );
```

## 5.8 Testarea apicației client UDP-MFC

Pentru testarea aplicației construite vom folosi utilitarul *Hercules* ce pune la dispoziție un modul de comunicare UDP, ilustrat în figura 5.3. Hercules folosește un soclu UDP conectat, activat prin apăsarea butonului *Listen*. Portul local pe care este legat soclul este introdus în controlul de editare *Local port*, iar portul destinație este introdus în controlul de editare *Port*.

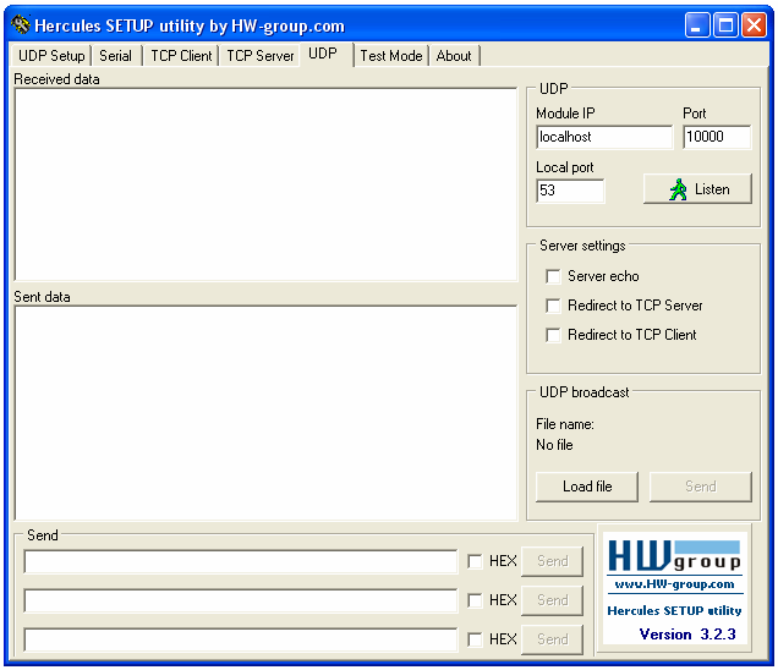


**Figura 5.3** Utilitarul *Hercules* – modulul UDP

Prin bifarea opțiunii *Server echo*, mesajele recepționate sunt transmise înapoi sursei. Transmisia mesajelor se realizează prin introducerea lor în controlul de editare *Send* şi apăsarea butonului *Send*.

**Exercițiu**.

Să se implementeze o aplicație client DNS bazat pe protocolul UDP prin utilizarea arhitecturii MFC pentru comunicarea pe socluri (i.e. clasa `CAsyncSocket`). Cerințele clientului DNS sunt cele specificate în cadrul acestui capitol. Pentru testarea aplicației se va folosi utilitarul *Hercules.*