4. Server TCP – Implementare MFC

Perechea unei aplicații client TCP este întotdeauna o aplicație server TCP. În capitolul anterior s-a prezentat proiectarea şi implementarea unui client TCP utilizând arhitectura MFC. Testarea acestei aplicații s-a realizat folosind utilitarul *Hercules*. În cadrul acestui capitol vom prezenta paşii proiectării şi implementării unei aplicații Server TCP utilizând arhitectura MFC. Această aplicație va înlocui utilitarul *Hercules*, asigurând posibilitatea conectării şi deservirii mai multor clienți.

4.1 Cerințele aplicației server TCP-MFC

În continuare, vom trece în revistă cerințele unei aplicații server TCP. De regulă serverele rulează sub forma unor servicii, fără a avea încorporate interfețe grafice, configurațiile fiind încărcate din fişiere, transmise ca parametrii aplicației, sau transmise printr-o conexiune bazată soclu sau o conexiune serială. Cu toate acestea, întrucât soclul utilizat se bazează pe arhitectura MFC, utilizarea acestei arhitecturi permite realizarea foarte uşoară a unui control grafic al serverului. Din acest motiv, vom considera în acest capitol că serverul beneficiază şi de o interfață grafică ce asigură pornirea/oprirea acestuia, configurarea portului dar şi listarea conexiunilor create/distruse.

Pe lângă această interfață grafică, serverul trebuie să asigure conectarea mai multor clienți şi deservirea acestora. Deservirea clienților este specifică tipului serverului utilizat, aceasta defineşte de fapt comenzile ce pot fi transmise serverului. Înainte de procesarea acestor comenzi, serverul trebuie să fie capabil să administreze conexiunea, să citească mesaje şi să transmită înapoi răspunsuri, precum şi să administreze timpul de viață a conexiunilor.

Serverul proiectat va asigura o comunicare bazată pe mesaje text între mai mulți clienți. Clienții sunt identificați prin nume, acestea fiind transmise de către clienți serverului. Serverul trebuie să asigure interogarea listei clienților conectați şi schimbarea de mesaje text între aceştia. În concluzie, cerințele principale ce trebuie satisfăcute de aplicația server TCP sunt următoarele:

- Interfață grafică pentru pornirea/oprirea serverului, precum şi pentru configurarea portului pe care va asculta serverul dar şi pentru listarea conexiunilor create/distruse;
- Ascultarea pe un anumit port pentru conexiuni client TCP şi acceptarea conexiunilor client;
- Recepționarea mesajelor, procesarea acestora şi transmiterea răspunsurilor;
- Comenzi implementate:
    - Configurare denumire client – cu verificarea existenței unei denumiri identice;
    - Interogare lista clienților conectați;
    - Transmitere mesaje text de la un client la celălalt;
- Administrarea timpului de viață a conexiunilor.

Pe lângă aceste cerințe, din punctul de vedere a modalității de utilizare a soclurilor, ca şi în cazul aplicației client MFC vom utiliza `CAsyncSocket` pentru asigurarea funcționalităților necesare utilizării soclurilor. Pentru testarea aplicației se poate utiliza o formulă extinsă a aplicației client prezentate în capitolul anterior sau utilitarul *Hercules*.

4.2 Arhitectura aplicației

Pentru a satisface cerințele enumerate în sub-capitolul anterior, vom utiliza o arhitectură server bazată pe o fereastră dialog, denumirea clasei asociate ferestrei fiind `CMyNetServerDlg`. 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 *MyNetServer*, exemplele date fiind bazate pe această denumire.

Funcționalitatea soclului server ce ascultă pe un anumit port pentru conexiuni client este implementat în clasa `CMyServerSocket`, ce moşteneşte clasa `CAsyncSocket`. Pentru fiecare conexiune nouă client se alocă un soclu nou, încapsulat în `CAsyncSocket`. Administrarea soclurilor client se realizează de fapt prin intermediul clasei `CMyClientHandler`, ce moşteneşte clasa `CAsyncSocket`. Această arhitectură de clase este ilustrată în figura 4.1.

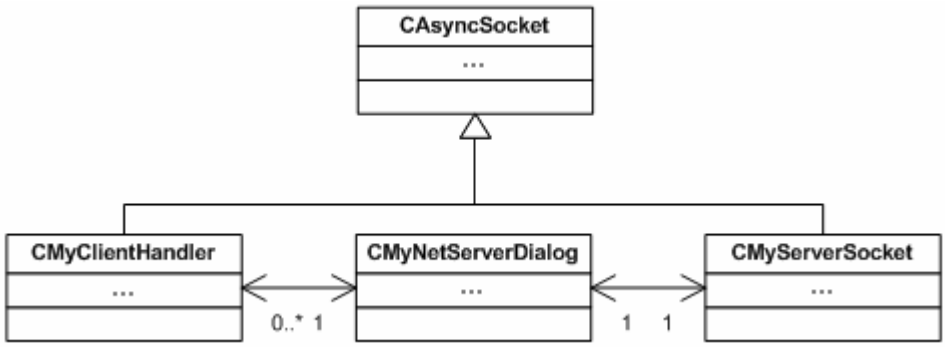

Figura 4.1 O parte din diagrama de clase a aplicației Server TCP-MFC

Clasa `CMyNetServerDlg` va asigura instanțierea şi distrugerea clasei `CMyServerSocket` precum şi administrarea listei conexiunilor client. Administrarea conexiunilor client presupune verificarea repetată a stării conexiunilor şi distrugerea obiectelor ale căror socluri au fost închise. Această verificare se va realiza prin utilizarea timerelor MFC.

4.3 Protocolul de comunicare între client şi server

În capitolul anterior s-a considerat un protocol text, fără delimitatori, care a simplificat ideea comunicării prin mesaje text, limitând sistemul la un singur client. De cele mai multe ori însă, vor exista mai mulți clienți şi un singur server. În astfel de situații, clientul va alege clientul destinație, căruia îi va transmite mesajul prin intermediul serverului. În această idee vom considera un protocol bazat pe cerere şi răspuns, cu formatul mesajelor ilustrat în figura 4.2.

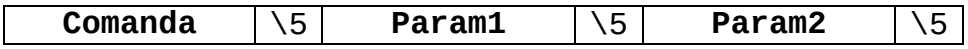

Figura 4.2 Formatul unui mesaj

Fiecare mesaj este format din trei componente: *Comanda*, *Param1* şi *Param2*. Cele trei componente au dimensiuni variabile, fiind separate printr-un singur caracter cu valoarea ASCII 5. Acest caracter poartă denumirea de *delimitator*. Acest format al mesajelor este valabil atât pentru mesaje de intrare cât şi pentru cele de ieşire. Cele trei componente ale mesajului sunt obligatorii, în cazul în care trebuie transmise mai mulți parametrii, se pot folosi mai mulți parametrii sau se pot utiliza alți delimitatori pentru separarea sub-parametrilor în cadrul unui singur parametru. Datorită acestei structuri a mesajelor, este evident că nici una din cele trei componente nu poate conține caracterul delimitator. Pentru a transmite caracterul delimitator, acesta trebuie codificat astfel încât valoarea să nu apară în cadrul mesajului. Metode de codificare utilizate în practică sunt *Base64* şi *Hex*.

Pentru cerințele enumerate în sub-capitolele anterioare, comenzile ce pot fi transmise serverului, denumite *cereri*, sunt următoarele:

- `SET_NAME`: asigură configurarea denumirii conexiunii client, cu valorile parametrilor:
  - `Param1`: Denumirea clientului sub forma unui şir de caractere, fără caracterul 0;
  - `Param2`: Neutilizat;
- `GET_USERS`: asigură interogarea listei denumirilor client conectate la server, cu valorile parametrilor:
  - `Param1`: Neutilizat;
  - `Param2`: Neutilizat;
- `SEND_MSG`: asigură transmisia unui mesaj de la un utilizator la altul, cu valorile parametrilor:
  - `Param1`: Denumirea clientului destinație;
  - `Param2`: Mesajul transmis.

Fiecare din cererile enumerate are asociat un răspuns de confirmare şi unul de eroare. Comenzile răspuns asociate cererilor sunt următoarele:

- `SET_NAME`:
  - `SET_NAME_OK`: asigură informarea clientului legat de configurarea noii denumiri, cu valorile parametrilor:
    - `Param1`: Denumirea clientului transmisă în cerere;
    - `Param2`: Neutilizat;
  - `SET_NAME_ERR`: asigură informarea clientului legat de o eroare ce a apărut în procesul de configurare, cu valorile parametrilor:
    - `Param1`: Denumirea clientului transmisă în cerere;
    - `Param2`: Eroarea sub forma unui şir de caractere neterminată cu 0. Posibile erori pot apărea dacă denumirea cerută pentru

înregistrare există deja sau în urma unei erori date de apelul funcțiilor sistem;

- `GET_USERS:`
  - `USERS`: asigură transferul listei denumirilor client conectate la server, cu valorile parametrilor:
    - `Param1`: Lista denumirilor despărțite de un delimitator diferit de '\5' (e.g. '\3');
    - `Param2`: Neutilizat;
  - `GET_USERS_ERR`: asigură informarea clientului legat de o eroare ce a apărut în urma procesului de interogare, cu valoarea parametrilor:
    - `Param1`: Neutilizat;
    - `Param2`: Eroarea sub forma unui șir de caractere neterminată cu 0. Posibile erori pot apărea în urma execuției unor funcții sistem;
- `SEND_MSG`:
  - `SEND_MSG_OK`: asigură informarea clientului legat de transmisia cu succes a mesajului la conexiunea client cerută, cu valorile parametrilor:
    - `Param1`: Denumirea clientului destinație;
    - `Param2`: Neutilizat;
  - `SEND_MSG_ERR`: asigură informarea clientului legat de o eroare ce a apărut în urma procesului de transmisie a mesajului, cu valorile parametrilor:
    - `Param1`: Denumirea clientului destinație;
    - `Param2`: Eroarea sub forma unui șir de caractere neterminată cu 0. Posibile erori pot apărea dacă clientul destinație nu există sau în urma unor erori date de execuția funcțiilor sistem.

În cazul în care comanda transmisă serverului nu este recunoscută, serverul va răspunde cu mesajul de eroare `INVALID_CMD`, cu valorile parametrilor:

- `Param1`: Comanda transmisă;
- `Param2`: Neutilizat.

4.4 Construirea interfeței grafice

Utilizând editorul de ferestre dialog se construiește interfața grafică din figura 4.3. Pentru fiecare căsuță de editare se atașează o variabilă membru prin clic dreapta pe controlul respectiv și selectarea opțiunii *Add Variable...* .

În partea stângă a interfeței grafice s-a adăugat o căsuță de editare multi-linie pentru listarea conexiunilor create și distruse. Aceasta poate fi folosită ca o logare a activității de pe server. Variabila membru atașată aceastei căsuțe este de tipul `CString`, cu denumirea `m_sConnections`.

În partea dreaptă s-a adăugat o căsuță de editare pentru introducerea portului pe care va asculta serverul. Această căsuță de editare are atașată o variabilă membru de tipul `UINT` cu denumirea `m_nPort`.

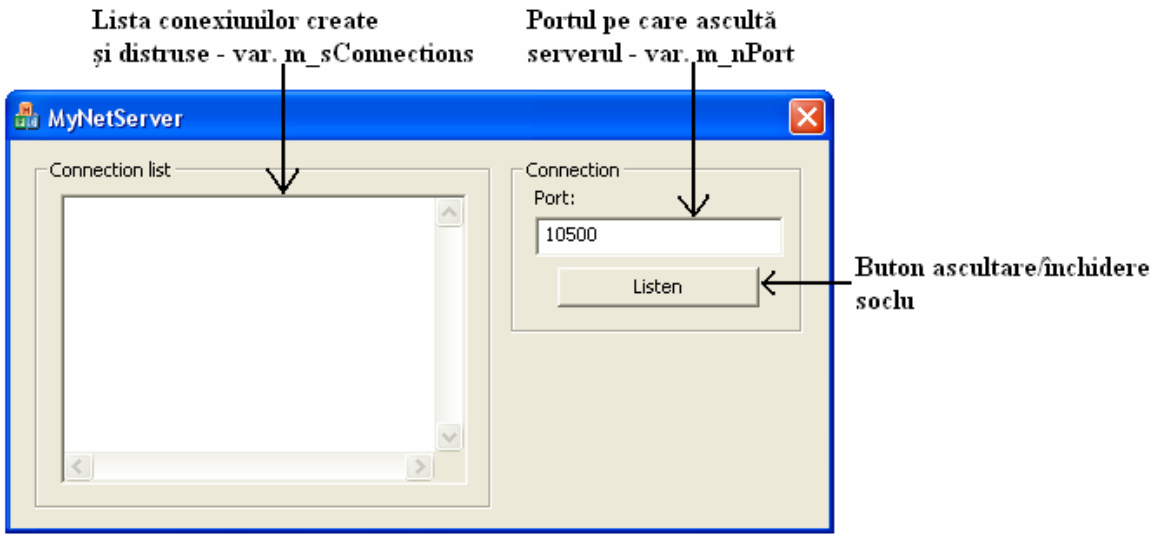

Figura 4.3 Interfața grafică a aplicației Server TCP-MFC şi variabilele membru atașate

Butonul cu eticheta afișată *Listen* asigură la o primă apăsare crearea soclului şi trecerea soclului într-o stare de ascultare. La a doua apăsare, acest buton va închide soclul şi va distruge obiectul atașat acestuia. Pentru o vizualizare a celei de-a doua funcționalități vom schimba eticheta afișată după prima apăsare în *Close*. Codul rezultat pentru această funcționalitate este următorul:

```
// IDC_BUTTON1 reprezintă ID-ul butonului
CWnd* pwnd = GetDlgItem( IDC_BUTTON1 );
if ( NULL == pwnd ) {
      return;
}

CString sText;
pwnd->GetWindowTextA( sText );

if ( sText == _T("Listen") )
{
      UpdateData( TRUE );

      if ( NULL == m_pSocket ) {
            m_pSocket = new CMyServerSocket( this );
      }

      if ( !m_pSocket->createSocket( m_nPort ) ) {
            AfxMessageBox( _T("Unable to create socket"),
                        MB_OK | MB_ICONERROR );
            delete m_pSocket;
            m_pSocket = NULL;
      }
      else {
            AfxMessageBox( _T("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("Listen") );
    }
```

Codul anterior include şi instanțierea dar şi secvența de apel a funcțiilor de creare a soclului detaliate în continuarea acestui capitol.

4.5 Construirea clasei de utilizare a soclului server

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

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

```
CMyServerSocket( CMyNetServerDlg* pDlg );
```

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

```
private:
    CMyNetServerDlg* m_pDlg;
```

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

```
CMyServerSocket::~CMyServerSocket( 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. Codul aferent este similar cu cel din cadrul aplicației client TCP şi este lăsată la latitudinea utilizatorului.

În continuare, vom adăuga o singură metodă publică la clasa `CmyServerSocket`, prin clic dreapta asupra denumirii clasei (în *Class View*) şi selectarea opțiunii *Add*, urmată de selectarea opțiunii *Add Function...* . Metoda adăugată va asigura crearea soclului şi trecerea acestuia într-o stare de ascultare. Prototipul acestei metode este următorul:

```
bool createSocket( const unsigned short nPort );
```

În corpul metodei `createSocket()` vom crea soclul şi vom apela metoda de ascultare din clasa de bază. Crearea soclului TCP se realizează prin apelul metodei `Create()`, al cărui prototip a fost deja prezentat în capitolul anterior. Față de utilizarea anterioară, în cazul soclului server TCP se schimbă doar valoarea primului argument, ce reprezintă valoarea portului pe care ascultă serverul:

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

Trecerea soclului în starea de ascultare se realizează prin apelul metodei `Listen()` din clasa de bază. Prototipul acestei metode este următorul:

```
BOOL Listen( int nConnectionBacklog = 5 );
```

unde parametrul `nConnectionBacklog` reprezintă numărul maxim de conexiuni în aşteptare premise. Valorile acceptate pentru acest parametru se află în intervalul 1-5.

Codul de utilizare a acestei funcții trebuie să ia în considerare şi cazul în care apelul funcției de ascultare ar rezulta într-o blocare al apelului, ceea ce nu reprezintă o eroare:

```
if ( CAsyncSocket::Listen() ) {
      m_bStarted = true;
      return true;
}
else {
      int nError = CAsyncSocket::GetLastError();
      if ( WSAEINPROGRESS == nError ) {
            m_bStarted = true;
            return true;
      }
      else {
            return false;
      }
}
```

Notificarea asupra acceptării unei noi conexiuni se realizează prin apelul funcției virtuale `OnAccept()`, cu prototipul:

```
virtual void OnAccept( int nErrorCode );
```

Dacă `nErrorCode = 0`, atunci se poate apela metoda `Accept()` pentru acceptarea conexiunii. Pentru ca serverul să fie notificat asupra evenimentelor de conectare, se va suprascie metoda `OnAccept()`. În cadrul acestei metode se apelează `Accept()` din clasa de bază, a cărui prototip este următorul:

```
virtual BOOL Accept( CAsyncSocket& rConnectedSocket,
                     SOCKADDR* lpSockAddr = NULL,
                     int* lpSockAddrLen = NULL );
```

Primul parametru, `rConnectedSocket` reprezintă o referință către un obiect de tipul `CAsyncSocket` ce trebuie creat în prealabil, în care se stochează descriptorul de soclu pentru noua conexiune. Al doilea parametru, `lpSockAddr` reprezintă un pointer către o structură de tipul `SOCKADDR`, în care se stochează adresa sursei de pe care s-a efectuat conexiunea. Al treilea parametru reprezintă adresa unei variabile de tip întreg în care se stochează dimensiunea structurii `SOCKADDR`.

Pornind de la structura `SOCKADDR` completată, se poate genera un şir de caractere care să conțină adresa stației de pe care s-a făcut conexiunea folosind funcția `inet_ntoa()`. Un exemplu de utilizare a acestei funcții este dat în codul metodei `OnAccept()`:

```
void CMyServerSocket::OnAccept( int nErrorCode )
{
      if ( !nErrorCode )
      {
            // Pregătirea structurii pentru stocarea adresei sursă
            struct sockaddr_in sa;
            int nLen = sizeof( sockaddr_in );

            // Crearea unui obiect pentru stocarea descriptorului
            CMyClientHandler* ps = new CMyClientHandler( m_pDlg );

            // Apelul metodei de acceptare
            if ( CAsyncSocket::Accept( *ps, ( SOCKADDR* )&sa, &nLen ) ){

                  // Crearea unui şir de caractere
                  CString s = inet_ntoa( sa.sin_addr );

                  // Adăugarea la lista de conexiuni
                  m_pDlg->addClientConnection( ps, &s );
            }
            else {
                  delete ps;
            }
      }

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

În secvența de cod anterioară s-a considerat existența funcției `addClientConnection()` în `CMyNetServerDlg` pentru adăugarea noului obiect la lista obiectelor conexiune.

4.6 Construirea clasei de administrare a conexiunii client

Administrarea unei singure conexiuni client se realizează în cadrul clasei `CMyClientHandler`. Această clasă este instanțiată din metoda `OnAccept()` a clasei

`CMyServerSocket` pentru fiecare conexiune nouă. Închiderea soclului se va realiza la distrugerea obiectelor de tipul `CMyClientHandler`, în destructorul clasei.

La închiderea coexiunii de partea clientului, obiectul trebuie distrus. Semnalarea închiderii se realizează prin apelul metodei virtuale `OnClose()`, care trebuie suprascrisă în clasa `CMyClientHandler`. Suprascrierea acestei metode nu echivalează însă şi cu distrugerea obiectului. Distrugerea trebuie efectuată din exteriorul clasei iar instanța clasei trebuie eliminată din lista obiectelor client.

Starea conexiunii este semnalată prin intermediul unei variabile boolene `m_bIsAlive`, cu valoarea inițială `true`, dată în constructorul clasei. La închiderea soclului se va apela metoda suprascrisă `OnClose()`, unde valoarea variabilei se va schimba în `false`:

```
void CMyClientHandler::OnClose( int nErrorCode )
{
      m_bIsAlive = false;
      CAsyncSocket::OnClose(nErrorCode);
}
```

Această variabilă este foarte utilă dacă vrem să închidem conexiunea în cazul unor abuzuri sau în cazul altor erori detectate pe parcurs. Tot ceea ce trebuie să facă programatorul este să atribuie variabilei `m_bIsAlive` valoarea `false`, după care la un moment dat soclul va fi închis iar obiectul va fi distrus. Pentru interogarea valorii acestei variabile se va adăuga o metodă publică, cu prototipul:

```
const bool isAlive( void ) const;
```

În continuare vom discuta despre citirea mesajelor de pe soclu şi procesarea acestora. După cum s-a prezentat în sub-capitolul 4.3, fiecare mesaj este format din trei componente despărțite de un delimitator. Față de capitolul anterior, mesajele trebuie procesate, trebuie identificați delimitatorii şi trebuie verificată validitatea celor trei componente. Din acest motiv, citirea datelor de pe soclu trebuie completată cu procesarea acestora şi identificarea celor trei componente.

Maşina de stare pentru citirea unui singur mesaj este ilustrată în figura 4.4. *Lambda*-tranziția înaintea prime stări denotă că nu există nici un caracter special care să anunțe începutul unui mesaj. Implementarea unei asemenea maşini de stare necesită:

- Definirea celor 4 stări de citire a mesajului (împreună cu starea finală):

```
#define STATE_CMD            0
#define STATE_PARAM1         1
#define STATE_PARAM2         2
#define STATE_END            3
```

- O variabilă membru pentru stocarea stării curente: `m_nState`, de tipul `unsigned char`, cu valoarea inițială `STATE_CMD`, conform figurii 4.4.

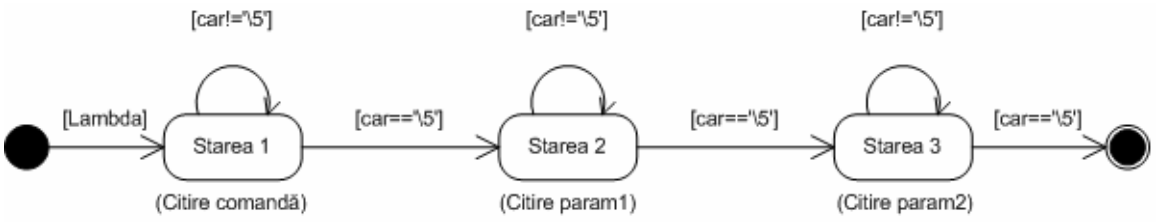

Figura 4.4 Maşina de stare pentru citirea unui mesaj

În continuarea acestui capitol vom prezenta două metode de citire de pe soclu. Prima dintre acestea asigură o citire octet cu octet, fiind deosebit de simplu de implementat, iar a doua asigură o citire a unui bloc de octeți şi procesarea acestora. Metoda din urmă este mult mai rapidă decât prima, fiind de altfel şi metoda adoptată în aplicații reale. De menționat este faptul că ambele metode respectă maşina de stare din figura 4.4.

4.6.1 Citirea octet cu octet

Una dintre cele mai simple metode de identificare a celor trei componente reprezintă citirea octet cu octet de pe soclu şi utilizarea unei maşini de stare. Maşina de stare ne va asigura identificarea pasului de citire, precum şi trecerea de la un pas la celălalt până la citirea unui mesaj complet. Această metodă de citire este una foarte simplă de implementat, cu dezavantajul că este una foarte lentă şi nu se pretează pentru mesaje de dimensiuni foarte mari (e.g. de ordinul kilo octeților). Aceasta poate fi înlocuită cu citirea mai multor octeți deodată, un asemenea mecanism fiind ilustrat în sub-capitolul următor.

Citirea octet cu octet necesită în plus, pe lângă definirea celor 4 stări şi următoarele:

- Citirea unui singur caracter de pe soclu (în metoda `OnReceive()`), cu apelul metodei `processChar()` pentru procesare:

```
if ( !nErrorCode )
{
	_TCHAR ch;
	const int nRet = CAsyncSocket::Receive( &ch, 1 );
	if ( SOCKET_ERROR == nRet ) {
		AfxMessageBox( _T("Receive ERROR"),
				MB_OK | MB_ICONERROR );
		m_bIsAlive = false;
	}
	else {
		if ( nRet > 0 ) {
			processChar( ch );
		}
	}
}
else {
	m_bIsAlive = false;
}
```

- Procesarea caracterului citit, cu trecerea în starea următoare şi stocarea valorii celor trei componente în trei variabile membru de tipul `CString`: `m_sCmd`, `m_sParam1` şi `m_sParam2`:

```
// Definirea delimitatorului
#define COMMAND_DELIMITER     _T('\5')

if ( STATE_CMD == m_nState )
{
      if ( COMMAND_DELIMITER == ch ) {
            m_nState = STATE_PARAM1;
      }
      else {
            m_sCmd += ch;
      }
}
else if ( STATE_PARAM1 == m_nState )
{
      if ( COMMAND_DELIMITER == ch ) {
            m_nState = STATE_PARAM2;
      }
      else {
            m_sParam1 += ch;
      }
}
else if ( STATE_PARAM2 == m_nState )
{
      if ( COMMAND_DELIMITER == ch ) {
            m_nState = STATE_END;
      }
      else {
            m_sParam2 += ch;
      }
}
```

- În cazul în care citirea ajunge în starea finală (i.e. `m_nState == STATE_END`) se procesează mesajul şi se reiniţializează variabilele membru pentru citirea unui nou mesaj:

```
if ( STATE_END == m_nState )
{
      if ( _T("SET_NAME") == m_sCmd ) {
            processSetName();
      }
      else if ( _T("GET_USERS") == m_sCmd ) {
            processGetUsers();
      }
      else if ( _T("SEND_MSG") == m_sCmd ) {
            processSendMessage();
      }
      else {
            sendBackUnknownMessageErr();
            m_bIsAlive = false;
      }
      m_sCmd      = _T("");
```

```
        m_sParam1   = _T("");
        m_sParam2   = _T("");
        m_nState    = STATE_CMD;
}
```

Implementarea metodelor de procesare a mesajelor este deosebit de simplă, fiind lăsată ca exercițiu pentru cititor. De exemplu, metoda `processGetUsers()` va apela metoda `getUsers()` ce trebuie implementată în cadrul `CMyNetClientDlg`, unde se va construi şi se va returna lista utilizatorilor (`user1`, `user2` se vor înlocui cu parcurgerea listei conexiunilor client):

```
#define DEL_UTILIZATOR    _T("\3")
…
CString sResp;
sResp += user1;
sResp += DEL_UTILIZATOR;
sResp += user2;
sResp += DEL_UTILIZATOR;
…
```

Construirea mesajelor răspuns se realizează în aceeaşi manieră:

```
#define CMD_GET_USERS     _T("GET_USERS")
#define CMD_USERS         _T("USERS")
… // Definirea celorlalte comenzi
#define DEL_CMD           _T("\5")
…
CString sCmd;
sCmd += CMD_USERS;
sCmd += DEL_CMD;
sCmd += sResp;
sCmd += DEL_CMD;
sCmd += DEL_CMD;
```

Codul anterior este conform descrierii protocolului din sub-capitolul 4.3. În cazul în care un parametru este *neutilizat*, se adaugă doar delimitatorul, fără a preciza o valoare a parametrului.

O dată construite mesajele răspuns, acestea sunt transmise înapoi clientului utilizând apeluri repetate ale metodei `CAsyncSocket::Send()`.

4.6.2 Citirea unui bloc de octeți

Implementarea citirii unui bloc de octeți necesită pe lângă cele 4 stări o variabilă membru în care să se stocheze valorile celor trei parametrii sub formă vectorială, declarată astfel:

```
CString m_psComp[ 3 ];
```

unde 3 reprezintă numărul de componente ale unui mesaj precum şi o variabilă membru `m_sInput` de tipul `CString` pentru stocarea blocului de octeți citiți.

Față de metoda descrisă în sub-capitolul 4.6.1, citirea de pe soclu asigură un bloc de octeți ce pot fi procesați. Metoda `OnReceive()` rezultată este următoarea:

```
if ( !nErrorCode )
{
	_TCHAR pBuff[ 1024 ];
	const int nRet = CAsyncSocket::Receive( pBuff, 1023 );
	if ( SOCKET_ERROR == nRet ) {
		AfxMessageBox( _T("Receive ERROR"),
				MB_OK | MB_ICONERROR );
		m_bIsAlive = false;
	}
	else {
		if ( nRet > 0 ) {
			pBuff[ nRet ] = _T('\0');
			m_sInput += pBuff;
			processInput();
		}
	}
}
else {
	m_bIsAlive = false;
}

CAsyncSocket::OnReceive( nErrorCode );
```

Pentru procesarea blocului de octeți vom utiliza metoda `processInput()`:

```
// Definirea delimitatorului
#define COMMAND_DELIMITER     _T('\5')

int nPos = -1;
while( ( nPos = m_sInput.Find( COMMAND_DELIMITER ) ) >= 0 )
{
	// Stocarea componentei curente şi trecerea la starea următoare
	m_psComp[ m_nState++ ] = m_sInput.Mid( 0, nPos );

	// Salvarea blocului rămas pentru procesare
	m_sInput = m_sInput.Mid( nPos + 1,
				m_sInput.GetLength() - nPos - 1 );

	// Terminarea identificării componentelor şi trecerea la
	// procesarea mesajului
	if ( STATE_END == m_nState )
	{
		if ( _T("SET_NAME") == m_psComp[ 0 ] ) {
			processSetName();
		}
		else if ( _T("GET_USERS") == m_psComp[ 0 ] ) {
			processGetUsers();
		}
		else if ( _T("SEND_MSG") == m_psComp[ 0 ] ) {
			processSendMessage();
		}
		else {
			sendBackUnknownMessageErr();
```

```
                m_bIsAlive = false;
            }

            m_psComp[ 0 ] = _T("");
            m_psComp[ 1 ] = _T("");
            m_psComp[ 2 ] = _T("");
            m_nState      = STATE_CMD;
        }
}
```

4.7 Adăugarea şi distrugerea conexiunilor client

Instanțierea şi distrugerea clasei `CMyServerSocket` a soft deja prezentată în sub-capitolul 4.4. După instanțierea clasei se apelează funcția de creare a soclului care pune în acelaşi timp soclul într-o stare de ascultare.

O problemă mult mai interesantă reprezintă administrarea conexiunilor client, care nu a fost tratată până acum. Conexiunile client sunt acceptate de soclul server, care crează câte o instanță a clasei `CMyClientHandler` pentru fiecare conexiune nouă. Aceste obiecte nou create sunt de regulă stocate în containere, implementate de cele mai multe ori prin intermediul unor liste (simplu sau dublu înlănțuite).

Nu vom prezenta teoria listelor şi nu vom utiliza listele utilizator. În schimb, vom folosi listele oferite de STL (en. „Standard Template Library”), deoarece acestea fac parte din standardul ANSI C++, rezultând un cod ce poate fi compilat şi pe platformele UNIX. STL pune la dispoziție un set de containere şablon ce pot asigura stocarea oricărui tip de dată specificat la declarare. De exemplu, o listă de întregi se declară asfel:

```
list< int > denumireVar;
```

unde `list` reprezintă tipul listă, între simbolurile ‘<>’ se specifică tipul de dată stocat, iar `denumireVar` reprezintă denumirea variabilei listă declarată. Utilizarea tipului `list` necesită includerea bibliotecii `list` şi utilizarea spațiului de nume `std`:

```
#include <list>
using namespace std;
```

În aplicația noastră server TCP, vom stoca pointeri către obiecte `CMyClientHandler` într-o variabilă membră de tipul listă:

```
list< CmyClientHandler* > m_lstConn;
```

Pentru adăugarea unei conexiuni client la această listă, se va folosi metoda `push_back` din cadrul clasei `list`. La acceptarea unei conexiuni noi, se va apela metoda `addClientConnection()` din clasa `CMyNetServerDlg`. În definirea acestei metode s-a inclus şi adăugarea conexiunii la listă precum şi afişarea sursei noii conexiuni în controlul de editare din interfața grafică:

```
void CMyNetServerDlg::addClientConnection(
            CMyClientHandler* ps, CString* psSrcAddr )
{
      // Adăugarea conexiunii la listă
      m_lstConn.push_back( ps );
```

```
            // Afişarea sursei conexiunii
            m_sConnections += _T("Connection from host '");
            m_sConnections += *psSrcAddr;
            m_sConnections += _T("'\r\n");

            UpdateData( FALSE );
        }
```

O dată adăugate, starea conexiunilor trebuie supravegheată, conexiunile închise trebuie scoase din listă, iar obiectele ataşate trebuie distruse pentru eliberarea memoriei ocupate. Lipsa unei proceduri de curățare a conexiunilor închise va duce la creşterea dimensiunii memoriei ocupate, până la ocuparea întregii memorii sistem şi eşuarea procesului în final. Aceste probleme sunt deosebit de importante dacă luăm în considerare că serverele nu rulează câteva ore sau zile ci ele trebuie să ruleze fără oprire luni sau chiar ani de zile.

Interogarea stării fiecărei conexiuni se realizează prin apelul metodei `isAlive()` discutată în sub-capitolele anterioare. De regulă, interogarea stării conexiunilor şi distrugerea acestora se realizează dintr-un alt fir de execuție. Însă, utilizarea unui nou fir ar duce la creşterea complexității aplicației prin introducerea secțiunilor critice. Pentru a evita această situație (tratată de altfel mult mai pe larg în capitolele următoare) şi pentru a profita de avantajele utilizării MFC (unde întreaga arhitectură rulează pe un singur fir, toate apelurile fiind fără blocare), vom utiliza *Timerele* MFC. La expirare aceste timere inserează în coada de mesaje a ferestrei un mesaj `WM_TIMER`, pentru tratarea căruia se poate ataşa o metodă. În această metodă vom implementa supravegherea şi distrugerea conexiunilor.

Pornirea timerului trebuie să se realizeze după crearea cozii de mesaje, o locație corespunzătoare fiind metoda `OnInitDialog()`. Pornirea timerului se realizează prin apelul funcției `SetTimer()`, cu prototipul:

```
UINT_PTR SetTimer(
        UINT_PTR nIDEvent,
        UINT nElapse,
        void (CALLBACK* lpfnTimer)(HWND,UINT,UINT_PTR,DWORD) );
```

Primul parametru, `nIDEvent` reprezintă ID-ul evenimentului (i.e. un număr natural), pentru diferențierea timerului care a expirat, întrucât pe acelaşi mesaj `WM_TIMER` pot fi tratate mai multe timere. Următorul, `nElapse` reprezintă timpul de expirare specificat în milisecunde. Ultimul parametru, `lpfnTimer` reprezintă un pointer către o funcția care să fie apelată la expirarea timerului – pentru valoarea `NULL` a acestui parametru se apelează funcția implicită de tratare a mesajului `WM_TIMER`.

Distrugerea conexiunilor nu reprezintă o operație urgentă care să fie executată la fiecare milisecundă întrucât, de regulă, numărul conexiunilor care pot fi deschise practic într-o milisecundă nu poate pune în pericol stabilitatea sistemului. Această afirmație este adevărată întrucât de regulă serverele rulează pe sisteme multi-tasking, unde timpul de execuție este împărțit între mai multe procese, iar coada de aşteptare pentru acceptarea de conexiuni noi este limitată de sistemul de operare. Din aceste considerente considerăm suficientă execuția codului de distrugere a conexiunilor la câteva zeci de milisecunde.

În continuare vom considera un timer ce expiră la fiecare 100 milisecunde pornit astfel:

```
#define SV_CLEANUP_EVENT_ID   1
…
SetTimer( SV_CLEANUP_EVENT_ID, 100, NULL );
```

După cum s-a menționat anterior, codul de supervizare a stării conexiunilor şi de distrugere a acestora se va adăuga în metoda de tratare a mesajului `WM_TIMER`. Adăugarea unei asemenea metode se realizează prin vizualizarea proprietăților ferestrei dialog, alegerea secțiunii *Messages* urmată de alegerea mesajului `WM_TIMER`. Codul rezultat este următorul:

```
void CMyNetServerDlg::OnTimer( UINT_PTR nIDEvent )
{
      if ( SV_CLEANUP_EVENT_ID == nIDEvent )
      {
            bool bRemoved = false;
            list< CMyClientHandler* >::iterator pos;
            for( pos = m_lstConn.begin() ; pos != m_lstConn.end() ; )
            {
                  if ( !(*pos)->isAlive() ) {
                        m_sConnections += _T("Client connection ")
                                          _T("from host '");
                        m_sConnections += (*pos)->getClientHost();
                        m_sConnections += _T("' has been closed \r\n");
                        delete (*pos);
                        pos = m_lstConn.erase( pos );

                        bRemoved = true;
                  }
                  else {
                        ++pos;
                  }
            }

            if ( bRemoved ) {
                  UpdateData( FALSE );
            }
      }

      CDialog::OnTimer(nIDEvent);
}
```

În codul anterior se parcurge lista conexiunilor utilizând iteratorul `pos`. Limitele listei sunt testate prin apelul metodelor `begin()` şi `end()`. Verificarea stării conexiunii se realizează prin apelul metodei `isAlive()`, în cazul în care conexiunea a fost închisă se şterge intrarea corespunzătoare poziției curente din listă şi se distruge conexiunea. Ştergerea unui element din listă se realizează prin apelul metodei `erase()` căreia i se transmite poziția elementului şters şi va returna poziția următorului element. În cazul în care conexiunea este validă se trece la elementul următor prin incrementarea iteratorului `pos`. În cadrul acestui cod se afişează în controlul de editare un mesaj de distrugere a conexiunii.

Exercițiu.

Să se implementeze o aplicație server TCP utilizând MFC, folosind CAsyncSocket. Aplicația va implementa protocolul prezentat în cadrul acestui capitol. Pentru testarea aplicației se poate folosi utilitarul *Hercules* într-o primă fază. Ulterior, se va utiliza clientul din capitolul anterior completat cu acelaşi protocolul precum şi capacitatea de transmitere a mesajelor text la utilizatorii selectați din lista returnată de server.