

Echtzeit-Kommunikation

Der eigene Chat mit C++



Neben E-Mails sind Chats und Messenger eine der am weitesten verbreiteten Anwendungen des Internet. In diesem Beitrag erfahren Sie, wie man solche Programme implementiert. Nebenbei bauen Sie dabei auf einen eigenen Client/Server basierenden Chat.

THOMAS WÖLFER

Damit die Sache rund wird, reicht ein Chat-Client und ein Chat-Server nicht aus. Um die Verbindung zwischen den Clients und dem Chat-Server herzustellen, gibt es Master-Server, die von den Clients nach anderen Servern gefragt werden können. Die Funktionalität ist direkt im Chat-Server-Programm eingebaut.

Das Ganze verwendet die CSocket-Klasse der MFC. Den Quell-Code zu den im Folgenden vorgestellten Klassen und Programmen finden Sie auf der CD zu diesem Sonderheft. Wenn Sie sich für die Details der Sockets interessieren und lieber eine eigene Socket-Klasse implementieren möchten, verwenden Sie als Ausgangspunkt am besten den Artikel über die Kommunikation mit E-Mail-Servern, den Sie an anderer Stelle in dieser Ausgabe finden. Dort wird auch mit Sockets kommuniziert, allerdings, ohne dass die MFC-Socket-Klassen zur Anwendung kämen.

Der Chat, den Sie implementieren werden, besteht aus zwei Programmen. Das eine Programm ist der Chat-Server, das andere der C-Client. Um zu chatten muss sich der Chat-Client beim Server anmelden. Dieser führt im Wesentlichen eine Liste aller angemeldeten Clients mit und reagiert im Übrigen auf Kommandos.

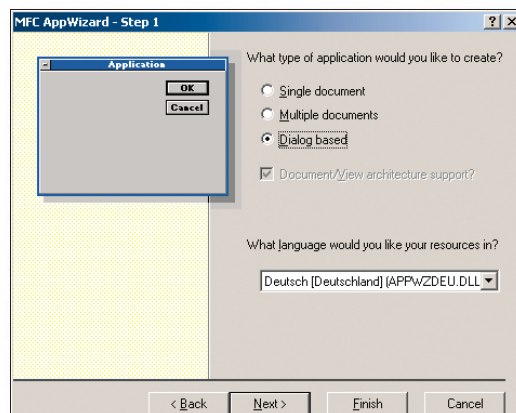
Mit diesen Kommandos können die Clients den Server dazu veranlassen, bestimmte

Dinge zu tun. Zum Beispiel können Sie dafür sorgen, dass der Chat-Server einen bestimmten Text an alle angeschlossenen Clients versendet.

Der Chat-Server meldet sich beim Start bei einem Master-Server an, und die Clients können den Master-Server nach Adressen von anderen Servern fragen. (Allerdings kann man in den Beispiel-Programmen damit noch nicht viel machen. Ein Benutzer-Interface, mit dem die IP-Adresse des zu verwendenden Servers angegeben werden kann, wäre eine sinnvolle Erweiterung der Programme.)

Insgesamt sind an der Software vier wichtige Klassen und ein paar Hilfskonstrukte beteiligt. Bei den Klassen handelt es sich um die Folgenden:

- XChatSocket
- XClientSocket
- XConnectSocket
- XChatServer



DAS SERVER-PROGRAMM wird einfach als auf eine Dialogbox basierende Anwendung entwickelt. Der Client bekommt hingegen ein richtiges Fenster.

Die XChatServer-Klasse kapselt dabei den eigentlichen Server und verwendet dazu den XConnectSocket und den XClientSocket als Hilfen. Der XConnectSocket dient ausschließlich der Verbindungsaufnahme mit dem Server, eingehende Verbindungen werden von dieser Klasse gehandhabt. Ist die Verbindung zustande gekommen, wird sie in Form eines XClientSockets abgebildet und vom Server verwaltet. Der XConnectSocket ist wieder frei und kann auf die nächste Verbindung warten.

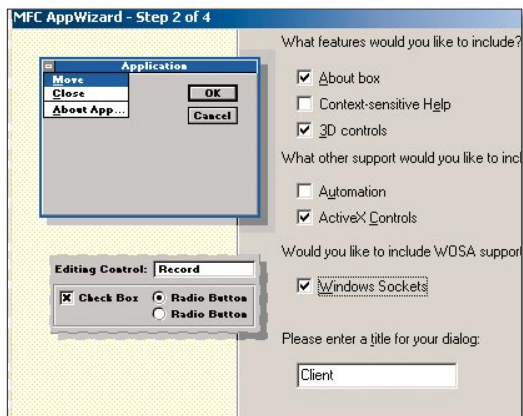
Die XChatSocket-Klasse wird nur auf der Client-Seite des Chats eingesetzt und kapselt dort die Verbindung zum Server.

Bevor die ersten Quell-Codes angezeigt werden, hier noch ein kurzer Hinweis: Im abgedruckten Quell-Code wurden aus Gründen der Übersichtlichkeit fast alle Fehlerbehandlungen entfernt. Im Quell-Code auf der CD sind diese vorhanden.

Der Chat-Server

Nach der Konstruktion des Chat-Servers wird dieser mit der Methode `Run` zum „Laufen“ gebracht. Diese Methode ist dafür zuständig, dass der Server tatsächlich in die Lage versetzt wird, eingehende Verbindungen aufzunehmen. Das macht der Server nicht selbst, sondern verwendet dazu die Hilfsklasse XConnectSocket:

```
bool
XChatServer::Run( CServerDlg* p)
{
    m_pUI = p;
    RegisterWithMaster();
    DisplayMessage("Connect
    ↳ Socket wird erzeugt");
```



DIE WINDOWS SOCKETS brauchen Sie in beiden Programmen, sonst ist die Kommunikation per Sockets etwas schwierig.

```
m_pConnectSocket = new
↳XConnectSocket( this);
m_pConnectSocket->Run();
DisplayMessage( "XChatSer
↳ver
↳wartet auf
↳Verbindungen");
return true;
}
```

Run() wird ein Zeiger auf ein User-Interface-Element übergeben. Dieser Zeiger wird abgelegt, und später für die Kommunikation mit dem User-Interface verwendet. Dazu gehört zum Beispiel die Anzeige von verschiedenen Status-Informationen. Im nächsten Schritt meldet sich der Server mit *RegisterWithMaster()* beim Master-Server an.

Die Details dieser Funktion finden Sie im Kasten „So funktioniert der Master-Server“.

Nach einer kurzen Meldung erzeugt der Server eine Instanz vom Typ *CConnectSocket* und setzt auch diese mit einem Aufruf von *Run()* in Betrieb. Momentan ist es ausreichend zu wissen, dass das *XConnectSocket* eingehende Verbindungen annimmt, und den Server darüber informiert. Wie die *XConnectSocket*-Klasse selbst funktioniert, finden Sie etwas später im Beitrag.

Zu diesem Zeitpunkt ist der Server bereits betriebsbereit:

Eine „*XConnectSocket*“-Instanz wartet auf eingehende Verbindungen, und sonst passiert nicht viel. Nimmt nun ein Client Verbindung auf, teilt der *XConnectSocket* dies dem Server mit, indem er die Funktion *OnAcceptConnect()* aufruft.

```
void
XChatServer::OnAcceptConnect()
{
    XClientSocket* p = new
    ↳XClientSocket( this);
    m_pConnectSocket->Accept
    ↳(*p);
    DisplayMessage( "Neue
    ↳Verbindung.");
    p->Run();
    m_lstClients.AddTail( p);
}
```

```
UpdateClientList();
CTime time = CTime::
↳GetCurrentTime();
CString strTime = time.
↳FormatGmt( "%H:%M:%S");
CString str = "Hallo beim
↳Chat-Server. Die lokale
↳Uhrzeit ist: " + strTime;
SendToClient( p,
↳XCMD_HANDSHAKE, str);
}
```

Der Chat-Server legt dazu als erste eine neue Instanz einer *XClientSocket* an. (Auch diese Klasse wird etwas später erläutert.) Nun wird die Verbindung mit *m_pConnectSocket->Accept()* angenommen: Dadurch wird der *XConnectSocket* wieder frei und die Client-Verbindung wird lokal durch den neu angelegten *XConnectSocket* abgebildet. Damit das funktioniert muss dieser noch mit *Run()* angestoßen werden. Danach stellt der *XClientSocket* den lokalen Endpunkt der Socket-Verbindung dar.

Der Server trägt den Client in seiner Liste der Clients ein. Bei dieser Liste handelt es sich um Folgendes

```
CTypedPtrList<CPtrList, XClient
↳Socket*> m_lstClients;
```

Wenn Sie auch diese Template-Klassen verwenden möchten, müssen Sie in Ihrem Header-File „*stdafx.h*“ die Dateien „*afxtempl.h*“ inkludieren.

Danach begrüßen Sie den neuen Client. Dazu setzen Sie zunächst mit *TimeFormat()* einen String zusammen,

CHAT-KOMMANDOS VERSENDEN

Beim Chat-Programm unterhalten sich der Client und der Server mit einem eigenen Mini-Protokoll. Dabei werden immer Meta-Daten und echte Daten versendet.

Wenn Sie den Artikel „Reden mit dem Mailserver“ bereits gelesen haben, wissen Sie, dass beim POP3-Protokoll das Ende des Datenstroms durch eine Zeile signalisiert wird, die nur einen Punkt enthält. Für den Chat-Server wurde ein etwas praktischerer Stil gewählt. Hier besteht jede Nachricht aus einem Header und einem danach folgendem Datensatz beliebiger Größe. Diese Größe wird sehr wohl festgelegt. Dazu dient der Header. Das empfangende Programm kann immer einen Block der Größe des Headers lesen, diesen auswerten und dann einen Block der im Header angegebenen Größe lesen. Der Header enthält dabei neben der Größe noch eine Angabe, die das Kommando betrifft. Folgende Kommandos sind bisher implementiert:

• XCMD_HANDSHAKE

Mit diesem Kommando meldet sich der Client beim Server an. Dabei teilt er dem Server den Namen mit, unter dem der Client den anderen Clients bekanntgemacht werden soll.

• XCMD_BROADCAST

Ein Client sendet ein solches Kommando an den Server und dieser reagiert darauf, indem er das gleiche Kommando an alle Clients schickt. Das ist die Chat-Nachricht.

• XCMD_GETUSERS

Dieses Kommando sendet ein Client an den Server, um die aktuelle Liste der Chat-Teilnehmer zu erfragen. Darauf antwortet der Server mit dem gleichen Kommando, wobei der zugehörige Datensatz die Liste aller Teilnehmer enthält. Wenn ein Chat-Teilnehmer den Chat verlassen hat, sendet der Server dieses Kommando automatisch an alle Teilnehmer.

• XCMD_REGISTERMASTER

Mit diesem Kommando registriert sich ein Server beim Master-Server.

• XCMD_UNREGISTERMASTER

Mit diesem Kommando meldet sich ein Server beim Master-Server ab.

• XCMD_QUERYSERVERS

Mit diesem Kommando erfragt ein Client die Liste der bekannten Server vom Master-Server. Der Master-Server sendet das gleiche Kommando an den Client zurück, wobei die zugehörigen Daten die Liste der Server ausmacht.

Eine wichtige Eigenschaft des hier vorliegenden Systems ist die Art und Weise der Kommunikation. Diese ist in allen wesentlichen Punkten ohne State. Die Kommunikation erfolgt immer so, dass der Client dem Server eines oder mehrere Kommandos senden kann und auch der Server Kommandos in die andere Richtung schickt.

„Antworten“ als solche werden dabei nicht erwartet. Die Antwort auf ein Kommando ist einfach ein Kommando, das beim Sender des ursprünglichen Kommandos eingeht.



NACH DEM START: Der Chat-Server meldet sich automatisch beim Master-Server an. Wenn das nicht geht, gibt es keinen Master-Server.

der die aktuelle Server-Zeit enthält. Zusammen mit einem kleinen Begrüßungstext senden Sie diesen String per *SendToClient()*-Kommando an den Client zurück.



BEI DER VERBINDUNGSAUFNABME mit dem Client müssen Sie einen Namen für sich auswählen. Unter diesem Namen erscheinen Sie dann im Chat.

Damit wäre auch das nächste zu erläuternde Kommando klar: *SendToClient()*. Dieses Kommando verwenden Sie im Server, um ein Kommando an einen Client zu senden. Bei den Clients verwenden Sie übrigens eine ähnliche Funktion, um Kommandos an den Server zu senden.

Bevor Sie *SendToClient()* implementieren, müssen Sie wissen, wie Kom-

mandos im Chat versendet werden. Alle Kommandos im Chat werden mit Hilfe einer Struktur versendet. Diese Struktur beschreibt das Kommando, oder - um genau zu sein - zwei Eigenschaften des Kommandos:

```
struct XCHAT_DATA_HEADER
{
    UINT    id;
    DWORD   dwSize;
    // char* psz;
};
```

Das „id“-Feld enthält dabei die Nummer des Kommandos, während das *dwSize*-Feld angibt, wie viele Daten zum Kommando gehören. Versendet wird immer ein Datenblock, der sich aus dem *XCHAT_DATA_HEADER* sowie den zum Kommando gehörenden Daten zusammensetzt.

Diesen Block müssen Sie mit *SendToClient()* zusammensetzen:

```
void
XChatServer::SendToClient(
    XClientSocket* pc, UINT idMsg,
    const char* pData)
{
    CString str( pData);
    DWORD dwSizeString =
        str.GetLength() + 1;
    DWORD dwSize = dwSizeString
        + sizeof( XCHAT_DATA_HEADER);
    void* p = alloca( dwSize);
    XCHAT_DATA_HEADER* psd =
        (XCHAT_DATA_HEADER*)p;
    psd->id = idMsg;
    psd->dwSize = dwSize;
    psd++;
    char* pstr = (char*)psd;
    strcpy( pstr, str);
    pc->Send( p, dwSize);
}
```

Die Funktion bekommt drei Parameter. Der erste beschreibt den Client, an den etwas gesendet werden soll, der zweite Parameter ist die *id* des Kommandos (siehe dazu auch den Kasten „Chat-Kommandos versenden“) und der dritte ist ein Zeiger auf die zu versendenden Daten.

Zunächst ermitteln Sie die Länge des Datensatzes. Das ist einfach die Anzahl an Zeichen im String. Nun brauchen Sie einen Speicherblock, der so groß, ist wie der String lang und zusätzlich noch Platz für den *XCHAT_DATA_HEADER* hat. Diese Größe berechnen Sie und allozieren einen Block entsprechender Größe:

```
DWORD dwSize = dwSizeString +
    sizeof( XCHAT_DATA_HEADER);
void* p = alloca( dwSize);
```

Dann lassen Sie einen *XCHAT_DATA_HEADER*-Pointer auf diesen Block zeigen und tragen unter Verwen-

SO FUNKTIONIERT DER MASTER-SERVER

Beim vorliegenden Beispielprogramm kann jeder Chat-Server automatisch auch Masterserver sein. Sie können übrigens zum Testen auf einem Rechner beliebig viele Clients starten, allerdings nur jeweils einen Server. Für einen Test der Master-Server-Funktionalität benötigen Sie zumindest zwei Rechner. Für die Master-Server-Funktionalität ist im Beispiel-Programm nur sehr wenig Benutzer-Interface implementiert worden. So ist zum Beispiel die IP-Adresse des Master-Server fest eincodiert. Mit ein wenig Hilfe vom Compiler ist dies

aber kein Hindernis. Der Master-Server-Modus sollte funktionstüchtig sein. Wie gesagt kann jeder Server von Haus aus auch Master-Server sein. Wenn sich ein Server bei einem Master-Server anmelden will, sendet er diesem das *XCMD_REGISTERMASTER*-Kommando, wobei die Daten die Namen des sich anmeldenden Servers bzw. seine IP-Adressen enthalten müssen. Fürs Abmelden beim Master-Server wird das Kommando *XCMD_UNREGISTERSERVER* verwendet. Beide werden allerdings nicht mit der gleichen Funktion verschickt:

```
void
XChatServer::RegisterWithMaster( bool fRegister)
{
    CSocket s;
    if( s.Create())
    {
        if( s.Connect( MASTERSERVER, CONNECT_SOCKET_PORT))
        {
            const int cbBuf = 128;
            struct REG { XCHAT_DATA_HEADER h; char sz[cbBuf]; };
            REG r;
            r.h.id = fRegister ? XCMD_REGISTERMASTER :
                XCMD_UNREGISTERMASTER;
            r.h.dwSize = cbBuf;
            CString strMe = LOCAL_SERVER_NAME;
            strcpy( r.sz, strMe);
            s.Send( &r, sizeof( REG));
            return;
        }
    }
    return;
}
```

Nachdem das Kommando nur versendet wird, der versendende Server selbst aber nicht weiter an Antworten interessiert ist, ist hierzu keine spezielle Socket-Klasse notwendig. Die Daten werden zusammen mit der passenden *XCHAT_DATA_HEADER*-Struktur in einer Variablen verpackt und mit einer ganz normalen MFC

CSocket-Instanz versendet. Auch ein Client kann mit dem Master-Server kommunizieren. Clients verwenden dabei das *XCMD_QUERYSERVER*-Kommando, mit dem der Master-Server nach einer Liste von Servern gefragt werden kann. Der Master-Server sendet in einem solchen Fall eine entsprechende Liste zurück.



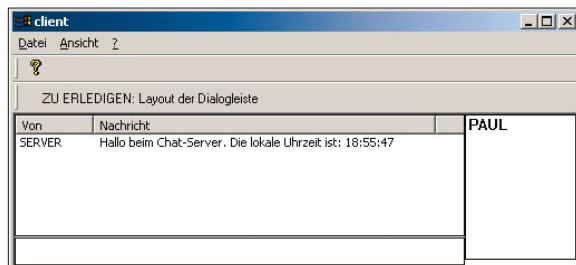
dung dieses Pointers die Größe und die Kommando-ID ein. Inkrementieren Sie den Pointer. Dieser zeigt dann an die Stelle, an die Sie die Daten kopieren können, also auf das Byte, hier dem `dwSize`-Feld.

Die Daten kopieren Sie mit `strcpy()` in den Speicherbereich. Schließlich versenden Sie das ganze Päckchen mit `pc->Send()` an den gewünschten Empfänger.

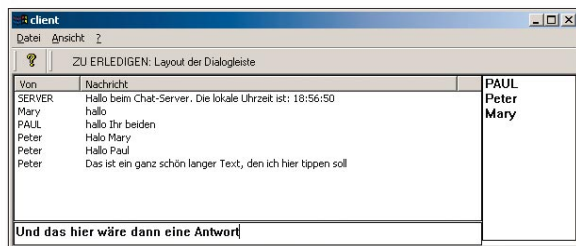
■ Requests bearbeiten

Egal, ob Sie die Daten im Client oder im Server versenden, Sie müssen diese auf der Gegenseite auch empfangen und auspacken können.

Auf der Server-Seite funktioniert dies dadurch, dass der `XClientSocket` die Daten auspackt und in bereits ausgepackter Form an den Server weiterreicht. Im Server landen die empfangene



NACH DEM ANMELDEN: Der Chat-Server begrüßt Sie unter anderem mit der aktuellen Server-Zeit. Rechts im Bild: Die Liste der momentan angemeldeten Chat-Clients.



DER CHAT IM BETRIEB: Drei Teilnehmer unterhalten sich. Ihre Nachrichten geben Sie im Edit-Control am unteren Bildschirmrand ein.

nen Daten, zusammen mit dem zugehörigen `XClientSocket` in der Funktion `ProcessRequest()`.

In dieser Funktion kümmern Sie sich um die Bearbeitung eines beim Server eingegangenen Kommandos:

```
void
XChatServer::ProcessRequest(
    XClientSocket* ps, UINT
    idRequest, const char* pData)
{
    typedef void (XChat
    Server::*pfx)(XClientSocket*,
    const char*);
```

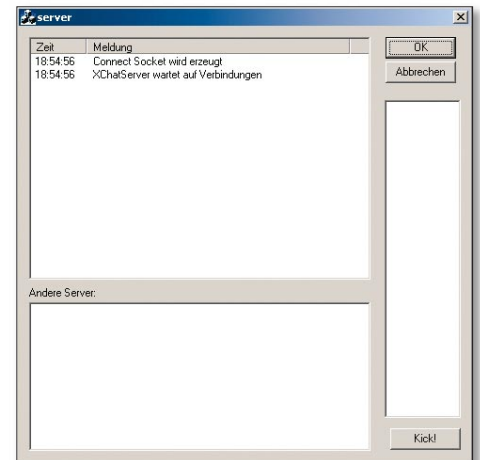
```
struct RH { UINT id; pfx pf; };
RH arr[] =
{
    XCMD_HANDSHAKE,
    ProcessHandshake,
    XCMD_BROADCAST,
    ProcessBroadcast,
    XCMD_GETUSERS,
    ProcessGetUsers,
    XCMD_REGISTERMASTER,
    ProcessRegisterMaster,
    XCMD_UNREGISTERMASTER,
    ProcessUnregisterMaster,
    XCMD_QUERYSERVERS,
    ProcessQueryServers,
};
for( int i=0;
    i<sizeof(arr)/sizeof(RH); i++)
{
    if( arr[i].id == idRequest)
    {
        (this->arr[i].pf)
        ( ps, pData);
        return;
    }
}
DisplayMessage( "Unbekannter
Request");
}
```

Die Funktion erhält drei Parameter. Das sind der `ClientSocket`, bei dem das Kommando einging, die Art des Kommandos und die mitgelieferten Daten. Sie müssen herausfinden, welcher Kommando-Handler für das Kommando zuständig ist, und diesen aufrufen.

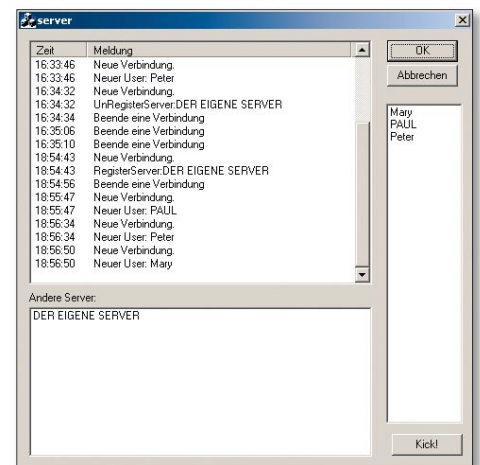
Im Beispiel ist das Folgendermaßen gelöst. Für jedes mögliche Kommando existiert im `XChatServer` eine Member-Funktion. Diese haben alle einen ähnlichen Namen, der mit dem Prefix `Process...` beginnt. Es gibt also eine `ProcessHandshake()`-Funktion, eine `ProcessBroadcast()`-Funktion und so weiter.

Um die Erweiterbarkeit des Protokolls möglichst einfach zu gestalten, verwenden Sie in `ProcessRequest()` ein Array einer Struktur. Das Array initialisieren Sie mit jeweils einem Kommando-ID und der zugehörigen Member-Funktion. Dann iterieren Sie einfach über das Array, bis Sie die eingegangene Kommando-ID gefunden haben und rufen die passenden Member-Funktion auf.

Im Beispiel ist das vielleicht nicht sofort ersichtlich, da Zeiger auf Member-Funktionen eine etwas merkwürdige



DER CHAT-SERVER nach dem Start: Warten auf Verbindungen kann recht langweilig werden.



DER CHAT-SERVER als Master-Server: Momentan, unten auf dem Dialog, ist nur ein Chat-Server beim Master-Server angemeldet. Außerdem haben sich bereits drei Teilnehmer mit dem Server verbunden.



WENN SIE VERSUCHEN einen Chat-Client rauszuwerfen, müssen Sie einen aus der Liste der Clients auswählen.

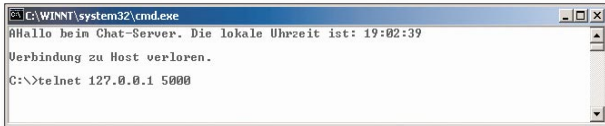
Schreibweise verlangen. Um das Ganze etwas zu vereinfachen, wird im Beispiel-Code ein neuer Typ definiert. Dieser hat den Namen „pfx“, eine Abkürzung für „Pointer to Member Function“.

```
typedef void
(XChatServer::*pfx)(XClient
```

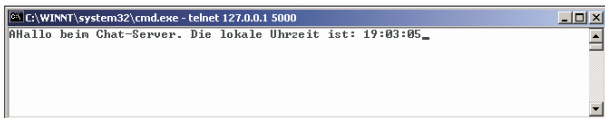


```
Socket*, const char*);
```

Nun wird eine Struktur namens RH definiert, die sich aus einer Variablen



ZUM TESTEN DES SERVERS können Sie auch Telnet verwenden: Dazu müssen Sie die IP-Adresse des Servers und die passende Port-Nummer eingeben.



NACHDEM SIE SICH mit Telnet verbunden haben, erhalten Sie die Begrüßungsmeldung vom Server.

vom Typ pfx und aus einem UNIT zusammensetzt:

```
struct RH { UINT id; pfx pf; };
```

Dann wird ein Array dieser Struktur mit den ID sun, den zugehörigen Member-Funktionen, initialisiert. Schließlich wird die passende Member-Funktion anhand des ermittelten darauf zeigenden Zeigers aufgerufen:

```
(this->arr[i].pf)( ps, pData);
```

Kommandos verarbeiten

Die Member-Funktionen zum Verarbeiten der Kommandos sind alle sehr ähnlich und werden daher an dieser Stelle nur mit ein paar Beispielen erläutert. Die Funktion *ProcessHandshake()* kümmert sich um die Behandlung des XCDM_HANDSHAKE-Kommandos, das der Client zu Beginn der Sitzung an den Server sendet. Mit diesem Kommando identifiziert sich der Client beim Server:

```
void
XChatServer::ProcessHandshake
( XClientSocket* ps, const char*
  pData)
{
    DisplayMessage("Neuer User:
    " + CString( pData));
    ps->SetDisplayName( pData);
    UpdateClientList();
    for( POSITION pos = m_lst
    Clients.GetHeadPosition();
    pos; )
        ProcessGetUsers
        ( m_lstClients.GetNext
        ( pos), 0);
}
```

Der Name des Clients ist beim Kommando in den übertragenen Daten enthalten. Diesen Namen legen Sie im

lokal vorliegenden Socket mit *SetDisplayName()* ab. Dadurch kennt der lokale XClientSocket den Namen des

Clients, für dessen Verbindung er der Endpunkt ist.

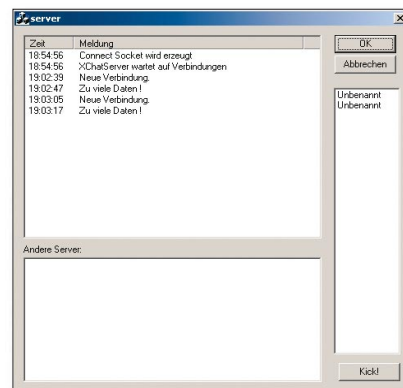
Erneuern Sie die Anzeige der „Client-Liste“. Dabei handelt es sich um eine Liste auf der Dialogbox des Servers, in der die Namen der Clients eingetragen werden.

Danach senden Sie allen angemeldeten Clients das XCMD_GETUSERS-Kommando, indem Sie *ProcessGetUsers()* für jeden einzelnen Client aufrufen. Dadurch wird allen angemeldeten Clients der Name des neuen Users mitgeteilt, und die Clients können ihre eigenen Benutzer-Interfaces entsprechende updaten.

Mit der Funktion *ProcessBroadcast()* bearbeiten Sie eine soeben von einem Client eingegangene Chat-Nachricht:

```
void
XChatServer::ProcessBroadcast
( XClientSocket* ps, const char*
  pszMsg)
{
    CString str = ps->GetDisplay
    Name() + "|" + CString
    ( pszMsg);
    for( POSITION pos = m_lst
    Clients.GetHeadPosition();
    pos; )
    {
        SendToClient
        ( m_lstClients.Get
        Next( pos),
        XCMD_BROADCAST, str);
    }
}
```

Bei einem Broadcast-Kommando reagieren Sie im Server dadurch, dass Sie



BEI DER VERBINDUNGSNAHME mit Telnet wird das „Chat-Protokoll“ nicht eingehalten: Der Server meldet einen Fehler.

den eingegangenen Text an alle angemeldeten Clients weitersenden. Zuvor erweitern Sie diesen um den Namen des Clients, der den Text gesendet hat. Diesen Namen haben Sie im Zuge des Anmeldevorgangs vom Client in der zugehörigen XClientSocket-Instanz gespeichert.

Sie brauchen nur einen String passenden Inhalts zusammenzusetzen, und diesen mit der schon bekannten *SendToClient()*-Methode an alle Clients zu versenden. (Den Namen des Absenders und den Text trennen Sie mit einem „vertical Bar“. Die Clients können dann die beiden Teile der übertragenen Information besser auseinanderhalten.)

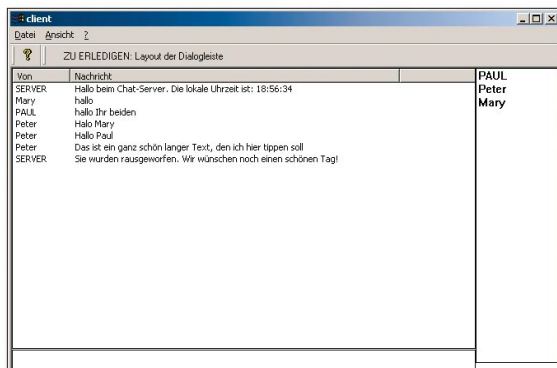
Die Server-Klasse verfügt noch über eine weitere Hilfsfunktion, die vom Benutzer-Interface des Programms aus aktiviert werden kann. Auf dem Dialog zum Server befindet sich unter anderem eine Liste aller angemeldeten Clients. Darunter gibt es einen Button mit der Beschriftung *Kick!*. Dieser ist dafür gedacht, unliebsame Benutzer aus dem Chat werfen zu können. Ein Klick auf den Button löst einen Aufruf von *KickClient()* aus:

```
void
XChatServer::KickClient( XClient
  Socket* ps)
{
    POSITION pos = m_lst
    Clients.Find( ps);
    m_lstClients.RemoveAt
    ( pos);
    UpdateClientList();
    SendToClient( ps, XCMD_
    BROADCAST, "SERVER| Wir
    wünschen einen schönen
    Tag!");
    ps->Close();
    delete ps;
}
```

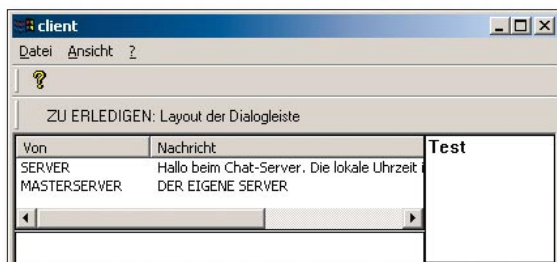
Bei *KickClient()* erhalten Sie nur einen Parameter, und das ist ein Zeiger auf die lokale XClientSocket-Instanz, die zum Client gehört. Alles was Sie nun tun müssen, ist diesen ClientSocket in der Liste Ihrer Clients zu suchen und aus der Liste zu entfernen. Schließlich senden Sie dem Client noch eine höfliche, persönliche Nachricht, schließen die Verbindung und zerstören den Socket.

Die Socket-Klassen

Wie bereits mehrfach erwähnt verwaltet die XChatServer-Klasse die Socket-Verbindungen nur, um die tatsächlichen Verbindungen kümmern sich andere Klassen. Beide sind sehr übersichtlich. Wenden wir uns zunächst der XConnectSocket-Klasse zu.



WENN SIE EINEN CLIENT „rauswerfen“, erhält dieser noch eine „nette“ Meldung.



DER CLIENT KANN vom Master-Server eine Liste der Server anfordern. Die Antwort vom Master-Server ist in der Von-Spalte entsprechend markiert.

Von dieser Klasse gibt es im Programm nur eine Instanz, und die ist im XChat-Server eingebettet. Wenn der Chat-Server per *Run()* gestartet wird, startet dieser die *XConnectSocket*-Klasse ebenfalls über einen Aufruf von *Run()*.

```
bool
XConnectSocket::Run()
{
    if(! Create( CONNECT_
        SOCKET_PORT, SOCK_
        STREAM)) return false;
    if(! Listen()) return
        false;
    return true;
}
```

Für die *XConnectSocket*-Klasse ist *Run()* praktisch der Konstruktor. Zunächst rufen Sie die *Create()*-Funktion der Basis-Klasse auf und versetzen den Socket in den Zustand, mit dem er auf eingehende Verbindungen wartet: *Listen()*. Gehen nun Verbindungen ein, wird die Funktion *OnAccept()* aufgerufen:

```
void
XConnectSocket::OnAccept( int
    nErrorCode)
{
    CSocket::OnAccept
        ( nErrorCode);
    if( nErrorCode)
    {
        m_pServer->
            DisplayMessage
                ("SOCKET_ERROR");
    }
    else m_pServer
```

```
    ->OnAccept
    ->Connect();
}
```

In dieser Funktion lassen Sie die Basisklasse ihre normale Arbeit tun, rufen dann aber über den *m_pServer* Pointer (der wurde beim Aufruf des Konstruktors übergeben) die *OnAcceptConnect()*-Methode des Servers auf. Wie bereits beschrieben, kümmert diese sich darum, eine neue *XClientSocket*-Instanz für die Verbindung zu erzeugen.

Damit ist es Zeit, die *XClientSocket*-Klasse näher zu betrachten. Diese hat nur eine relevante Funktion, und das ist *OnReceive()*. *OnReceive()* ist das Gegenstück zu *SendToClient()*. Die Funktion kümmert sich um das Auslesen der Daten und leitet die ausgelesenen Daten an die *ProcessRequest()*-Funktion des

Servers weiter:

```
void
XClientSocket::OnReceive( int
    nErrorCode)
{
    CSocket::OnReceive( nError
        Code);
    XCHAT_DATA_HEADER d;
    int cb = Receive( &d,
```

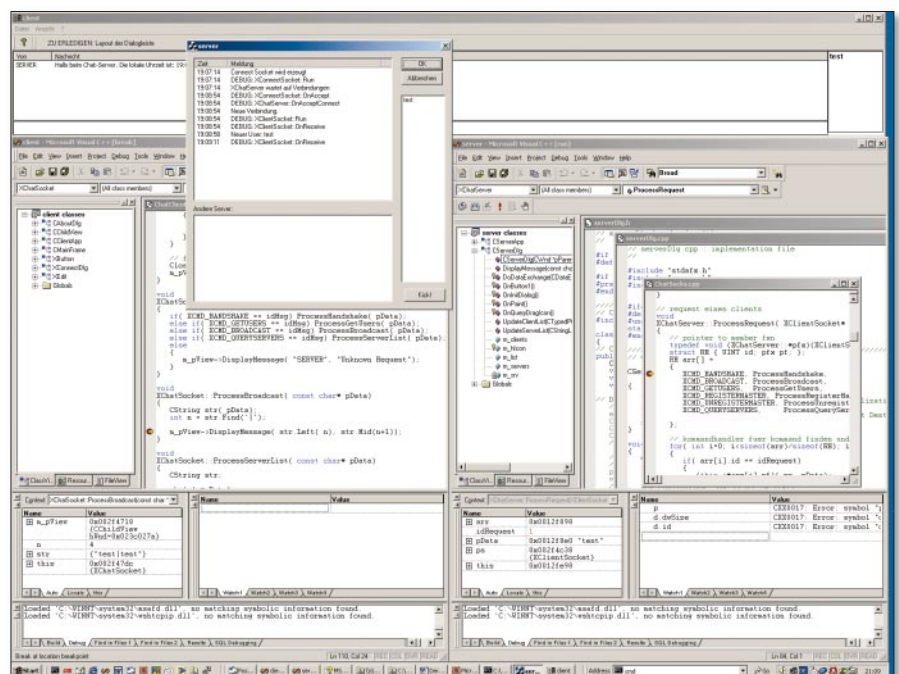
```
sizeof( XCHAT_DATA_HEADER));
    if( cb && (cb != SOCKET_ERROR))
    {
        char* p = (char*)alloca
            ( d.dwSize);
        int cb = Receive
            ( p, d.dwSize);
        if( cb && (cb !=
            SOCKET_ERROR))
        {
            m_pServer-
                >ProcessRequest( this,
                    &d.id, p);
            return;
        }
    }
}
```

Zunächst geben Sie hier der Basis-klass eine Chance, eventuell notwendige Arbeiten zu erledigen. Lesen Sie die eingehenden Daten aus, wobei Sie zunächst nur einen Block der Größe *XCHAT_DATA_HEADER* lesen. Dessen Größe ist bekannt, und nachdem der Block gelesen ist, ist auch die Größe des nun folgenden Blockes bekannt.

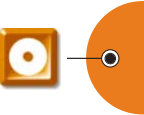
Dafür allozieren Sie einen passenden Speicherbereich und lesen die eingehenden Daten aus. Sind diese angekommen, rufen Sie die *ProcessRequest()*-Funktion des Servers auf, in der die eigentliche Behandlung des eingegangenen Kommandos vorgenommen wird.

Das Benutzer-Interface

Für den Server-Teil des Chats benötigen Sie ein wenig Quell-Code. So sind zum Beispiel eine Funktion zum Updaten der angemeldeten Mitglieder und ein Fenster, mit dem Sie die Vorgänge



BEIM ENTWICKELN UND TESTEN müssen Sie oft zwei Kopien von VC++ gleichzeitig laufen lassen: eine für den Client, die andere für den Server. Ein großer Bildschirm ist da hilfreich.



im Server überprüfen können, hilfreich.

Auf diese Funktionalität wird hier nicht weiter eingegangen. Sie finden eine Beispiel-Implementierung für diesen Dialog im Quell-Code auf der Heft-CD.

Der Chat-Client

Auf der Seite des Clients sieht die Welt schon bedeutend einfacher aus. Dort wird die XChatSocket-Klasse verwendet, und diese hat den folgenden Aufbau:

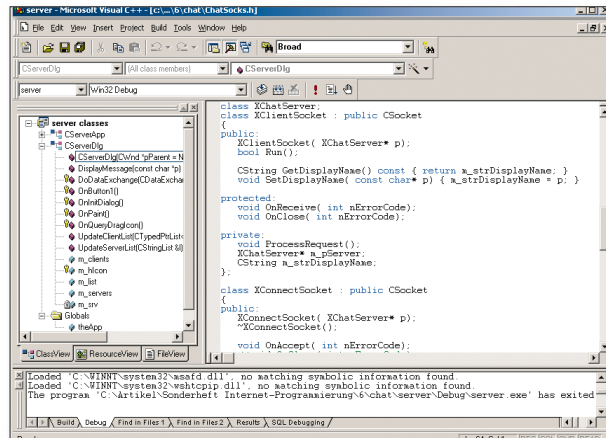
```
class XChatSocket : public CSocket
{
public:
    XChatSocket();
    bool Run(CChildView* pView,
        const char* pszServer, const
        char* pszName);
    void SendChatMessage( const
        char* p);
    void GetServerList();

protected:
    void OnReceive( int nError
        Code);
    void OnClose( int nError
        Code);

private:
    void ProcessRequest( UINT
        idMsg, const char*
        pData);
    void ProcessGetUsers( const
        char* pData);
    void ProcessHandshake
        ( const char* pData);
    void SendToServer( UINT
        idMsg, const char*
        pszData);
    void ProcessBroadcast
        ( const char* pData);
    void ProcessServerList
        ( const char* pData);
    CChildView* m_pView;
};
```

Die wesentlichen Funktionen sind dabei: *ProcessRequest()*, *OnReceive()* sowie *Run()* – die restlichen Funktionen erfüllen den Zweck, den man bereits am Funktionsnamen erkennen kann. *ProcessRequest()* ist das Äquivalent zur Funktion gleichen Namens auf der Server-Seite und auch gleichartig implementiert.

Auch *OnReceive()* wird Ihnen bekannt vorkommen: Die Implementierung entspricht weitestgehend der in *XClientSocket::OnRe-*

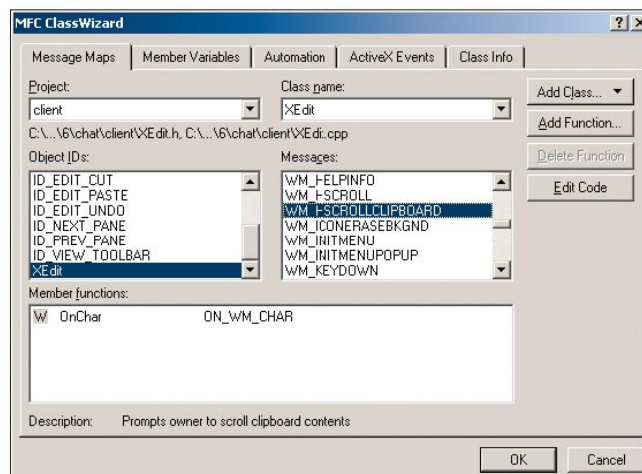


DIE DIALOGBOX DES SERVER erhält eine ganze Reihe an Hilfsmethoden, mit denen der Status des Servers dargestellt werden kann.

ceive(). In *Run()* finden sich noch neue Anwendungen:

```
bool
XChatSocket::Run( CChild
    View* pView, const char*
    pszServer, const char*
    pszName)
{
    m_pView = pView;
    if( Create()

        if( Connect
            ( pszServer,
            CONNECT_SOCKET
            _PORT))
        {
            if( AsyncSelect
                ( FD_READ
                | FD_CLOSE))
            {
                SendToServer
                    ( XCMD_HAND
                    SHAKE, psz
                    Name);
                return true;
            }
        }
    }
}
```



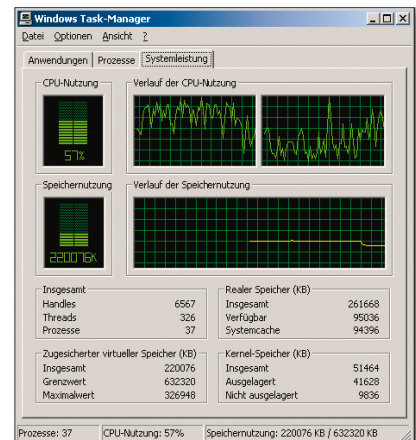
DAMIT SIE IM CLIENT die Nachrichten vernünftig eingeben können, brauchen Sie eine von CEdit abgeleitete Klasse. Die bauen Sie am besten mit dem ClassWizard.

```
return false;
}
```

Hier speichern Sie zunächst einen Zeiger auf eine View – diese View wird später verwendet, um Informationen über den Chat anzuzeigen. Dazu gehören unter anderem die eigentlichen Chat-Nachrichten. Nach dem Erzeugen des Sockets mit *Create()* verbinden Sie diesen mit dem Server.

Im Beispielprogramm wird dabei per Default der Master-Server verwendet. Sie können auch einen anderen Chat-Server verwenden. Mit *AsyncSelect()* legen Sie fest, an welchen Ereignissen Sie interessiert sind.

Sie geben damit bekannt, wann Sie gerne über ein Ereignis informiert wer-



BEIM DEBUGGEN BRAUCHEN Sie eine ganze Menge Speicher. Zwei VC++ Kopien belasten das System stark.

den möchten, damit Sie selbiges behandeln können.

Schließlich senden Sie noch ein *XCMD_HANDSHAKE*-Kommando an den Server, der sich über die Anwesenheit eines weiteren Clients freuen kann.

In diesem Beitrag haben Sie erfahren, wie Sie Client/Server-Programme mit den MFC-Sockets programmieren können. Als Nebenprodukt ist dabei noch ein ganz passabler Chat mit eigenem Client, Server und Masterserver-Verwaltung entstanden. Die vorliegende Software lädt geradezu dazu ein, zu einem großen Chat-System ausgebaut zu werden. UR