15

Internet-Anbindung

Dieses Kapitel behandelt die folgenden Themen:

- Protokolle und Adressen
- Netzwerkprogrammierung mit Sockets
- Internet-Anbindung mit HTTP
- Mini-Webserver

Netzwerkprogrammierung sowohl für interne als auch externe Netze wie das Internet, ist kein Bestandteil der Programmiersprache C++. Die notwendigen Funktionen sind in entsprechenden Bibliotheken »versteckt«, von denen es mehrere gibt. Aus Gründen der Portabilität habe ich mich für die Boost.Asio-Library, die betriebssystemabhängige Aspekte kapselt, entschieden.

Hinweis

Dieses Kapitel stellt eine kurze Einführung in die Netzwerkprogrammierung dar, um die Möglichkeiten der Realisierung mit C++ zu zeigen. Tatsächlich ist die Netzwerkprogrammierung ein komplexes Thema, sodass die hier gebotenen Informationen für professionelle Anwendungen nur ein Einstieg sein können. Am Ende des Kapitels wird auf weiterführende Dokumentationen hingewiesen.

15.1 Protokolle

Die Kommunikation zwischen Programmen verschiedener Rechner wird mittels mehrschichtiger Protokollstapel modelliert. Damit ist gemeint, dass eine Schicht auf dem Protokoll der darunterliegenden basiert. Zum Beispiel basiert HTTP (Hypertext Transfer Protocol), mit dem die HTML-Seiten übertragen werden, auf dem Protokoll TCP (Transmission Control Protocol) der sogenannten Transportschicht nach dem OSI-Modell (siehe zum Beispiel *http://www.selflinux.org/selflinux/html/osi.html*). HTTP gehört zur Anwendungsschicht. Andere Protokolle der Anwendungsschicht:

- `https` ist eine verschlüsselte HTTP-Verbindung über SSL (Secure Sockets Layer), TLS (Transport Layer Security) oder ein ähnliches Protokoll zur Verschlüsselung und Authentifizierung.
- `ftp` (file transfer protocol) zur Übertragung von Dateien über das Netz.
- `smtp` (simple mail transfer protocol) für E-Mail.
- `file` steht für die Übertragung vom lokalen Rechner.

Während das IP (Internet Protocol) die physikalische Verbindung eines Rechners zum Netzwerk identifiziert und die reine Datenübermittlung leistet, ist das TCP für die Zerlegung einer Sendung in Datenpakete und deren Versand zuständig, gegebenenfalls auf unterschiedlichen Transportwegen durch das Internet. Außerdem sorgt das TC-Protokoll dafür, dass die Pakete beim Empfänger in der richtigen Reihenfolge wieder zusammengesetzt werden. TCP/IP ist die Verbindungsart, die im Internet überwiegend verwendet wird.

Ein weiteres übliches Protokoll ist UDP (User Datagram Protocol, auch scherzhaft: Unreliable Datagram Protocol), welches Datenpakete versendet, für die weder garantiert wird, dass sie in der gesendeten Reihenfolge beim Empfänger ankommen, noch dass sie überhaupt ankommen. Das UDP-Protokoll ist einfacher und schneller als TCP/IP. Es ist besonders geeignet, wenn gelegentlicher Datenverlust tolerierbar ist, zum Beispiel bei der Übertragung von Video.

15.2 Adressen

Um eine Verbindung aufnehmen zu können, muss die Adresse bekannt sein. Dabei wird unterschieden zwischen der Adresse des Rechners im Netzwerk und, zur weiteren Unterteilung innerhalb eines Rechners, dem sogenannten Port. Eine Internet-Adresse wird URL (Uniform Resource Locator) oder URI (Uniform Resource Identifier) genannt. URL bezieht sich auf eine Untermenge von URIs. Die stark vereinfachte Syntax ist

protokoll://[host[:port]][/verzeichnisOderDatei]

Je nach Protokoll und Zweck können die Teile in eckigen Klammern optional sein. `host` kann ein Name oder eine IP-Adresse sein. Beispiele:

http://www.cppbuch.de
file:///home/user/dokument.txt
http://localhost:9090/
http://192.168.1.12/

Mehr zu URIs lesen Sie in [URI].

IP-Adresse

Die Internet-Adresse eines Rechners, wegen des Internet-Protokolls (IP) auch IP-Adresse genannt, ist eine Liste von Zahlen, getrennt durch Punkte. Bei dem zurzeit üblichen Protokoll IPv4 sind es vier Zahlen, von denen jede im Bereich von 0 bis 255 liegt, sodass sich aus den vier mal 8 Bit theoretisch insgesamt etwa 4,3 Milliarden Adressen bilden lassen. Es hat sich gezeigt, dass diese Zahl in vielen Bereichen nicht ausreicht, weswegen das IPv6 entwickelt wurde, dessen IP-Adresse aus *acht* 16-Bit-Zahlen besteht. Die damit mögliche Adressenzahl von 2^{128} oder etwa $3{,}4 \cdot 10^{38}$ ist so groß, dass sich jedem Ameisenbein und jedem Menschenhaar eine Adresse zuordnen ließe, und es blieben immer noch welche übrig. Die Internetanbieter haben noch nicht auf IPv6 umgestellt; das ist aber nur eine Frage der Zeit. Anbieter von Inhalten haben eine feste, statische IP-Adresse, private PCs wegen der knappen IPv4-Adressenanzahl dagegen in der Regel nicht – ihnen wird bei dem Verbindungsaufbau eine IP-Adresse zugeteilt. Deswegen heißt sie auch dynamische IP-Adresse, die dem Rechner mit dem Dynamic Host Configuration Protocol (DHCP) vom Server zugewiesen wird. In einem kleinen Netzwerk (Intranet) können den einzelnen Computern statische Adressen zugeordnet werden; der Router als Schnittstelle zum Internet bekommt eine dynamische Adresse und bildet sie intern auf die Adressen im Intranet ab.

Hostname

Internet-Adressen sind nicht gut zu merken, weswegen es für die Computer im Internet eingängige Namen gibt. Diesen Namen bezeichnet man als *Domain*, wobei der letzte Teil, zum Beispiel *.de* in *www.cppbuch.de*, Toplevel-Domain genannt wird. *www* kann eine Unterstruktur im Rechner sein oder ein anderer Rechner, der von dem Domain-Rechner angesteuert wird. Zu einem Namen kann es mehrere IP-Adressen geben, um die Last der Anfragen zu verteilen. Das Domain Name System (DNS) ist eine Datenbank, die auf eine großen Anzahl von Servern verteilt ist. Mit Hilfe dieser Datenbank lässt sich die IP-Adresse zu einem Hostnamen ermitteln. Die dazu notwendige Arbeit wird von einem Nameserver genannten Rechner erledigt, der eine Anfrage entweder aufgrund seiner eigenen Tabellen beantworten kann oder die Anfrage an andere Nameserver weiterreicht. Im folgenden Programm wird aus einem Rechnernamen die IP-Adresse ermittelt. Dabei wird zuerst auf dem eigenen Rechner nachgesehen, ob die Adresse notiert ist (Datei */etc/hosts* unter Linux). Wenn das keinen Erfolg hat, wird der Nameserver gefragt. Da das im Hintergrund geschieht, muss man sich darum nicht weiter kümmern. Das Beispielprogramm `ihost` erwartet die Eingabe eines Rechnernamens und gibt die IP-Adresse aus, vorausgesetzt, die Verbindung des Rechners zum Internet ist hergestellt. Der Aufruf `ihost www.example.com` zeigt IP4- und IP6-Adressen auf dem Bildschirm an. Das Pro-

gramm leistet Ähnliches wie das Unix-Kommando *host*. Einzelheiten zu den verwendeten Funktionen `getaddrinfo()` und `getnameinfo()` erhält man auf einem Unix-System mit der Abfrage `man gethostbyname`.

Listing 15.1: Host-IP-Adresse ermitteln

```
// cppbuch/k15/host/ihost.cpp
/*
HINWEISE für die Benutzung unter Windows:

1. Bei der Compilation in einem MSDOS-Eingabeaufforderungs-
   Fenster die Optionen -lwsock32 und -lws2_32 setzen, z.B.
   g++ ihost.cpp -ihost.exe -lwsock32 -lws2_32
   (wird bei Aufruf von make automatisch erledigt)
2. Bei der Compilation mit einer IDE im Fenster Werkzeuge->
   Compiler-Optionen o.ä. ein Häkchen bei
   "Diese Befehle zur Linker-Kommandozeile hinzufügen"
   machen und im Feld darunter -lwsock32 -lws2_32 eintragen.
*/

#ifdef WIN32
  #define _WIN32_WINNT 0x0501
  #include<winsock2.h>
  #include<ws2tcpip.h>
#else
  #include<netdb.h>
#endif
#include<iostream>
#include<string>
using namespace std;

int main(int argc, char* argv[]) {
   if(argc == 1) {
      cout << "Gebrauch: ihost Rechnername\n";
   }
   else {
#ifdef WIN32
      WSADATA wsaData = {0};
      if(WSAStartup(MAKEWORD( 2, 2 ), &wsaData ) !=0) {
         cerr << "WSAStartup() gescheitert!" << endl;
         return 1;
      }
#endif
      struct addrinfo* result;
      int error = getaddrinfo(argv[1], NULL, NULL, &result);
      if(error == 0) {
         struct addrinfo* item = result;
         while(item) {
            char hostname[NI_MAXHOST] = "";
            // Namen extrahieren, wenn möglich
            error = getnameinfo(item->ai_addr, item->ai_addrlen,
                                hostname, NI_MAXHOST, NULL, 0, 0);
            if(error == 0) {
```

```
            cout << hostname;
         }
         // IP-Adresse als String extrahieren, wenn möglich
         error |= getnameinfo(item->ai_addr, item->ai_addrlen,
                              hostname, NI_MAXHOST, NULL, 0,
                              NI_NUMERICHOST);
         if(error == 0) {
            cout << "  " << hostname << endl;
         }
         else {
            cout << argv[1] << " : Fehler in getnameinfo()." << endl;
         }
         item = item->ai_next;
      }
      freeaddrinfo(result);
   }
   else {
      cout << argv[1] << " kann nicht ermittelt werden." << endl;
   }
  }
#ifdef WIN32
  WSACleanup();
#endif
}
```

Manchen Namen wie etwa www.google.com sind mehrere Websites und IP-Nummern zugeordnet, die in der `while`-Schleife ausgegeben werden.

Hinweis

Die obigen Makro-Abfragen steuern die Übersetzung abhängig vom Betriebssystem. Es ist mühselig und fehleranfällig, für jedes Programm entsprechende Makro-Abfragen zu konstruieren und zu testen, um die Portabilität zu gewährleisten, ganz zu schweigen von der schlechteren Lesbarkeit. Man könnte natürlich eigene Klassen und Header-Dateien zum Verstecken betriebssystemabhängiger Programmteile schreiben – aber warum das Rad zwei Mal erfinden? Aus diesem Grund verwende ich im Folgenden die Library *Boost.Asio*, die alle betriebssystemabhängigen Teile kapselt, sodass man sich darum nicht kümmern muss.

localhost

In den obigen Adressbeispielen sehen Sie den Namen *localhost*. Dieser Name ist dem eigenen Rechner zugeordnet; die IP-Adresse ist 127.0.0.1. Beide sind nur intern gültig und keine von außen erreichbaren Adressen. Für TCP/IP-Anwendungen ist der eigene Rechner genauso wie ein entfernter Rechner ansprechbar, sodass diese Anwendungen rein lokal getestet werden können. Ein echtes Netzwerk ist dafür nicht notwendig. Im Betriebssystem ist eine Art virtuelle Netzwerkkarte eingerichtet, Loopback-Device genannt, die alle an diese Adresse gesendeten Daten verarbeitet.

Port

Ein Port ist eine Adresse innerhalb eines bestimmten Computers, die zwischen 0 und 65535 liegen kann. Jede Anwendung (Browser, ftp-Programm usw.) legt einen Port fest, damit sie feststellen kann, ob eine Sendung an sie gerichtet ist. Die Ports 0 bis 1023 sind für reservierte Netzwerkdienste vergeben, zum Beispiel wird Port 80 typischerweise für HTTP genutzt. Die reservierten Port-Nummern finden Sie in [Port]. Beispiel einer Adresse, die den Port 8080 enthält: *http://localhost:8080/*.

Datei

Eine weitere Verfeinerung der Adresse wird durch den Dateinamen gegeben. Wenn nur ein Hostname angegeben wird, ergänzt der Webserver im Fall des HTTPs meistens die Adresse durch Anhängen von *index.html* (im Webserver einstellbar). Beispiel einer Adresse, die eine Datei anspricht: *http://localhost:8080/tomcat-docs/servletapi/index.html*. In einer html-Datei können sogar bestimmte, durch eine Marke ausgezeichnete Stellen angesprungen werden: *http://www.provider.de/index.html#marke*.

15.3 Socket

Ein Socket (dt. etwa Steckdose) stellt das grundlegende Software-Element zur Verbindung zweier Programme auf verschiedenen Computern oder auch auf demselben Computer dar. Man unterscheidet zwischen dem Client-Socket des anfragenden Programms und dem Server-Socket des dienstanbietenden Programms. Die verwendeten Protokolle sind typischerweise TCP und UDP.

Es gibt einen Betriebssystemdienst, »daytime« genannt, der über Port 13 läuft und einen String mit dem aktuellen Datum und der aktuellen Zeit liefert. Der folgende TCP-Client kann diesen Dienst abfragen (daytime gibt es auch als UDP-Dienst).

Listing 15.2: Daytime-Client

```
// cppbuch/k15/tcpsocket/zeit/zeitclient.cpp
#include<iostream>
#include<cstdlib>
#include<boost/asio.hpp>
using namespace std;

int main(int argc, char* argv[]) {
  if(argc != 3) {
    cout << "Gebrauch: " << argv[0] << " <IP-Adresse> <port>" << endl;
    return 1;
  }
  boost::asio::io_service ioService;
  unsigned short port = atoi(argv[2]);
  boost::asio::ip::tcp::endpoint
    server(boost::asio::ip::address::from_string(argv[1]), port);
  boost::asio::ip::tcp::socket socket(ioService);
```

```
  socket.connect(server);
  cout << "Lokaler Endpunkt: " << socket.local_endpoint() << endl;
  cout << "Verbindung mit " << socket.remote_endpoint()
       << " hergestellt." << endl;
  const int SZ = 80;
  char buf[SZ+1];
  // Es werden maximal SZ Bytes gelesen:
  size_t anzahlBytes = socket.read_some(boost::asio::buffer(buf, SZ));
  buf[anzahlBytes] = '\0'; // Rest für Ausgabe ignorieren
  cout << buf << endl;
}
```

Das Objekt `io_service` ist die Schnittstelle zu den Ein-/Ausgabefunktionen des Betriebssystems. Die Funktion `from_string()` wandelt die als Zeichenkette übergebene IP-Adresse in ein Boost.Asio-internes Format des Typs `address` um. Wie Sie statt der Adresse einen Namen verwenden können, sehen Sie in Abschnitt 15.3.3 unten. `server` ist ein `endpoint`-Objekt, das den entfernten Endpunkt der zu schaffenden Verbindung, gekennzeichnet durch Adresse und Port, repräsentiert. Das `socket`-Objekt versucht nach Erzeugung, mit `connect(server)` die Verbindung mit dem Ziel aufzunehmen. Wenn das gelingt, werden mit `socket.read_some()` Daten vom Server empfangen. Die Anzahl der gelesenen Bytes wird zurückgegeben. `read_some()` benötigt als Parameter eine Sequenz von Puffern. `buffer(buf, SZ)` gibt eine einelementige Sequenz zurück ([asio]). Nur zur Dokumentation werden der lokale und der entfernte Endpunkt ausgegeben. Um das Programm zu testen, wird es wie folgt aufgerufen: `zeitclient.exe <IP-Adresse> 13`.

Jedoch ist der Dienst »daytime« bei den meisten Rechnern abgeschaltet. Um Hackern möglichst wenig Angriffsfläche zu bieten, werden verständlicherweise nur die unbedingt notwendigen Ports freigegeben. In diesem Fall reagiert das obige Programm mit einer Exception »Connection refused«. Man kann aber zum Testen auf dem eigenen Betriebssystem den Dienst »daytime« aktivieren. Unter Windows sehen Sie bitte in der Windows-Hilfe nach, Stichwort: einfache TCP/IP-Dienste. Dort ist das Verfahren erläutert. Unter Linux kann der Netzwerkdienst mit einem Tool aktiviert werden (yast2 bei SuSE-Linux). Der Aufruf

```
zeitclient.exe 127.0.0.1 13
```

zeigt dann die aktuelle Systemzeit an. Im Folgenden wird »daytime« mit einem eigenen Server realisiert, der dasselbe leistet wie der Dienst des Betriebssystems. Der Server kommuniziert mit dem obigen Clienten vom selben oder einem anderen Rechner aus. Weil die Portnummern bis 1023 reserviert sind, kommt eine größere Nummer zum Tragen.

Listing 15.3: Daytime-Server

```
// cppbuch/k15/tcpsocket/zeit/zeitserver.cpp
#include<ctime>
#include<iostream>
#include<boost/asio.hpp>
#include<cstring>
#include<cstdlib>
using boost::asio::ip::tcp;
using namespace std;

int main(int argc, char* argv[]) {
```

```
  if(argc != 2) {
    cout << "Gebrauch: " << argv[0] << " <port>" << endl;
    return 1;
  }
  boost::asio::io_service ioService;
  unsigned short port = atoi(argv[1]);
  tcp::acceptor acceptor(ioService, tcp::endpoint(tcp::v4(), port));
  while(true) {       // Abbruch mit Strg+C
    tcp::socket socket(ioService);
    cout << "lauschen an Port " << port
         << " ... (Abbruch mit Strg+C)" << endl;
    acceptor.accept(socket);
    cout << "Lokaler Endpunkt: " << socket.local_endpoint() << endl;
    cout << "Verbindung mit " << socket.remote_endpoint()
         << " hergestellt." << endl;
    time_t jetzt = time(NULL);
    const char* zeitstring = ctime(&jetzt);
    socket.write_some(boost::asio::buffer(zeitstring, strlen(zeitstring)+1));
  }
}
```

Im Unterschied zum Clienten gibt es ein `acceptor`-Objekt, das für die Verbindungsaufnahme zuständig ist. In der `while`-Schleife wird ein `socket`-Objekt angelegt, das der Funktion `accept()` übergeben wird. `accept()` blockiert solange, bis ein Client die Verbindung aufnimmt. An der Anzeige von lokalem und dem entfernten Endpunkt und dem Vergleich bei Client und Server können Sie gut sehen, dass der Client sich einen beliebigen freien Port als lokalen Endpunkt nimmt. Dieser Port ist der entfernte Endpunkt des Servers. Der Server wird zum Beispiel mit

```
zeitserver.exe 3000
```

gestartet. Statt 3000 kann jede andere freie Portnummer gewählt werden. Eine IP-Adresse ist nicht notwendig, da der Server auf dem Port `PortNr` des Rechners lauscht, auf dem das Programm gestartet wurde. Der Client wird zweckmäßig in einem anderen Shell-Fenster mit `zeitclient.exe <IP-Adresse des Servers> 3000` gestartet. Falls der Client sich auf demselben Rechner befindet, ist 127.0.0.1 anzugeben. Nach Verbindungsaufbau wird die Zeichenkette mit der Zeitinformation erzeugt und mit `write_some()` an den Clienten gesendet. Der Destruktor des lokalen `socket`-Objektes schließt am Ende des Schleifenkörpers die Verbindung, sodass der Server beim nächsten Schleifendurchlauf bereit ist für eine neue eingehende Anfrage. Der Aufruf `zeitclient.exe 192.168.1.2 3000` ergab in einem kleinen Netzwerk die Ausgaben

Client:

Lokaler Endpunkt: 192.168.1.4:34034
Verbindung mit 192.168.1.2:3000 hergestellt.
Sun Jan 16 13:45:32 2011

Server:

lauschen an Port 3000 ... (Abbruch mit Strg+C)
Lokaler Endpunkt: 192.168.1.2:3000
Verbindung mit 192.168.1.4:34034 hergestellt.
lauschen an Port 3000 ... (Abbruch mit Strg+C)

Im Ergebnis sind beide lokalen Endpunkte miteinander verbunden, wie hier gut zu sehen.

Was tun bei zu kleinem Puffer?

Im obigen Clienten-Programgramm ist der Puffer auf 80 Zeichen begrenzt. Letztlich werden nur Bytes übertragen, und das können sehr viele sein – wie viele, ist vorab oft nicht bekannt. Am einfachsten ist es, wenn der Puffer groß genug gewählt wird; es kann aber sein, dass das nicht praktikabel ist. Außerdem wird das Problem damit nicht gelöst, sondern nur in Richtung größerer Zahlen verschoben. Es gibt aber die Möglichkeit, nach dem Lesevorgang den Puffer auszuwerten und dann den Lesevorgang zu wiederholen usw., bis es nichts mehr zu lesen gibt. Dies sei an einem kleinen Beispiel gezeigt, in dem die Auswertung des Puffers nur aus der Ausgabe mit `cout` besteht, aber natürlich ganz anders gestaltet werden kann.

```
// bei zu kleinem Puffer
do {
   boost::system::error_code error;
   anzahl = socket.read_some(boost::asio::buffer(buf, SZ), error);
   if(error == boost::asio::error::eof) {
      break;            // kein Fehler, normales Ende
   }
   else {
      // Puffer auswerten
      for(size_t i=0; i < anzahl;++i) {
         cout << buf[i];
      }
      cout << endl;
   }
} while(anzahl > 0);
```

15.3.1 Bidirektionale Kommunikation

Im obigen Beispiel sendet der Server sofort nach Zustandekommen der Verbindung seine Antwort. Es ist aber auch möglich, dass er erst eine Nachricht des Clienten empfängt und in Abhängigkeit davon agiert. Die Kommunikation ist bidirektional. Um dieses zu zeigen, wird das obige Beispiel erweitert. Der Client fragt interaktiv das gewünschte Zeitformat ab und sendet es dem Server. Der Server wertet die Information aus und sendet die aktuelle Zeit im gewünschten Format zurück.

Listing 15.4: Client mit Formatwahl

```
// cppbuch/k15/tcpsocket/bidirectional/zeitclient2.cpp
#include<iostream>
#include<boost/asio.hpp>
using namespace std;

char formatWahl() {
   char ein;
   do {
      cout << "\nFormatwahl : 1 = Standard, 2 = Langform für Tag und Monat,\n"
         "3 = nur die Uhrzeit, 4 = Sekunden seit 1.1.1970, 0 = Programmende: ";
      cin >> ein;
```

```
  } while(ein < '0' || ein > '4');
  return ein;
}

int main(int argc, char* argv[]) {
  if(argc != 3) {
    cout << "Gebrauch: " << argv[0] << " <IP-Adresse> <port>" << endl;
    return 1;
  }
  boost::asio::io_service ioService;
  unsigned short port = atoi(argv[2]);
  boost::asio::ip::tcp::endpoint
    server(boost::asio::ip::address::from_string(argv[1]), port);
  char auftrag[2] = {0};
  while((auftrag[0] = formatWahl()) != '0') {
    boost::asio::ip::tcp::socket socket(ioService);
    socket.connect(server);
    socket.write_some(boost::asio::buffer(auftrag, 2)); // 2 Bytes
    const int SZ = 80;
    char buf[SZ+1];;
    // Antwort lesen
    size_t anzahlBytes = socket.read_some(boost::asio::buffer(buf, SZ));
    buf[anzahlBytes] = '\0'; // Rest für Ausgabe ignorieren
    cout << buf << endl;
  }
}
```

Neu ist im Vergleich, dass nach dem `connect()` der Auftrag an den Server gesendet wird. Dementsprechend wartet der Server erst diese Nachricht ab, ehe er sie auswertet und die Antwort zurückschickt.

Listing 15.5: Server mit Formatwahl

```
// cppbuch/k15/tcpsocket/bidirectional/zeitserver2.cpp
#include <ctime>
#include<iostream>
#include<boost/asio.hpp>
using boost::asio::ip::tcp;
using namespace std;

const char* getZeitstring(const char* format) {
  const size_t MAX = 80;
  static char buf[MAX] = {0};
  time_t jetzt = time(NULL);
  // Sonderbehandlung für %s, das nicht jeder Compiler kennt
  if(format[1] == 's') {
    size_t pos = MAX;
    while(jetzt > 0) { // Zahl > 0 in C-String umwandeln
      buf[--pos] = jetzt % 10 + '0';
      jetzt /= 10;
    }
    return (buf + pos);
  }
  else { // Umwandlung je nach Format-String mit strftime()
```

```
      tm *z = localtime(&jetzt);
      strftime(buf, MAX, format, z);
   return buf;
   }
}

int main(int argc, char* argv[]) {
   if(argc != 2) {
      cout << "Gebrauch: " << argv[0] << " <port>" << endl;
      return 1;
   }
   boost::asio::io_service ioService;
   unsigned short port = atoi(argv[1]);
   tcp::acceptor acceptor(ioService, tcp::endpoint(tcp::v4(), port));
   while(true) {     // Abbruch mit Strg+C
      tcp::socket socket(ioService);
      cout << "lauschen an Port " << port
           << " ... (Abbruch mit Strg+C)" << endl;
      acceptor.accept(socket);
      char daten[1] = {0};
      // 1 Byte lesen
      socket.read_some(boost::asio::buffer(daten, 1));
      const char* format = "%c"; // Vorgabe
      switch(daten[0]) {
        case '1' : format = "%c";                  break;
        case '2' : format = "%A, %d. %B %Y, %X Uhr"; break;
        case '3' : format = "%X";                  break;
        case '4' : format = "%s";                  break;
      }
      const char* zeitstring = getZeitstring(format);
      socket.write_some(boost::asio::buffer(zeitstring, strlen(zeitstring)+1));
   }
}
```

15.3.2 UDP-Sockets

Das UDP-Protokoll realisiert die Versendung von Datenpaketen, Datagramme genannt. UDP bietet weder eine Garantie dafür, dass die Pakete in der richtigen Reihenfolge, noch dass sie überhaupt beim Empfänger ankommen. Deswegen ist im Gegensatz zu TCP/IP kein Quittungsmechanismus erforderlich, und der Transport geht schneller. Es gibt weitere Unterschiede in der Anwendung und Realisierung:

- Beim TCP wird eine zuverlässige Verbindung aufgebaut. Die Kommunikation kann mit Strömen (streams) realisiert werden, auch wenn die unterliegende Schicht (IP) auf einer verbindungslosen Datenpaketübertragung beruht. Die Reihenfolge von TCP-Datenpaketen ist wesentlich.
- UDP ist dagegen verbindungslos. Jedes Datagramm enthält die Zieladresse und ist nicht abhängig von anderen Datagrammen.
- Im Gegensatz zur Client-Server-Kommunikation ist die UDP-Verbindung symmetrisch – es gibt keinen ausgezeichneten Server bzw. Client, allenfalls auf der Ebene der Anwendung, wie im Beispiel unten.

Zum Vergleich seien Zeitclient und -server in der UDP-Variante gezeigt, leicht gekürzt, um Wiederholungen zu vermeiden:

Listing 15.6: UDP-Zeitclient

```
// Auszug aus cppbuch/k15/udpsocket/clientudp.cpp
int main(int argc, char* argv[]) {
   // ...
   boost::asio::io_service ioService;
   unsigned short port = atoi(argv[2]);
   boost::asio::ip::udp::endpoint
      server(boost::asio::ip::address::from_string(argv[1]), port);
   char auftrag[2] = {0};
   while((auftrag[0] = formatWahl()) != '0') {
      boost::asio::ip::udp::socket socket(ioService);
      socket.open(boost::asio::ip::udp::v4());
      socket.send_to(boost::asio::buffer(auftrag, 2), server);
      const int SZ = 80;
      char buf[SZ+1];;
      // Antwort lesen
      boost::asio::ip::udp::endpoint hier;
      size_t anzahlBytes =
          socket.receive_from(boost::asio::buffer(buf, SZ), hier);
      buf[anzahlBytes] = '\0'; // Rest für Ausgabe ignorieren
      cout << buf << endl;
   }
}
```

Nicht nur der Namespace `udp` statt `tcp` ist anders, auch die Methodennamen: `send_to()` statt `write_some()` und `receive_from()` statt `read_some()`. Der `connect()`-Aufruf wird durch `open()` ersetzt. Ansonsten ist die Struktur recht ähnlich.

Listing 15.7: UDP-Zeitserver

```
// Auszug aus cppbuch/k15/udpsocket/serverudp.cpp
int main(int argc, char* argv[]) {
   // ...
   boost::asio::io_service ioService;
   unsigned short port = atoi(argv[1]);
   udp::socket socket(ioService, udp::endpoint(udp::v4(), port));
   while(true) {     // Abbruch mit Strg+C
      cout << "lauschen an Port " << port
           << " ... (Abbruch mit Strg+C)" << endl;
      boost::asio::ip::udp::endpoint entfernt;
      char daten[256] = {0};
      socket.receive_from(boost::asio::buffer(daten), entfernt);
      const char* format = "%c";
      // nur erstes Byte ist relevant
      switch(daten[0]) {        // wie oben
      case '1' : format = "%c";                  break;
      case '2' : format = "%A, %d. %B %Y, %X Uhr"; break;
      case '3' : format = "%X";                  break;
      case '4' : format = "%s";                  break;
      }
```

```
      const char* zeitstring = getZeitstring(format);
      socket.send_to(boost::asio::buffer(zeitstring, strlen(zeitstring)+1),
                     entfernt);
   }
}
```

15.3.3 Atomuhr mit UDP abfragen

Manche Dienste gibt es ausschließlich in der UDP-Version, zum Beispiel den Standarddienst zum Verteilen der Zeitinformation. Es gibt hochgenaue Atomuhren, deren Zeitinformation über eine geschichtete Serverstruktur dem Internet zur Verfügung gestellt wird (Schicht 1, Schicht 2, andere). Nach Anfordern einer Zeitinformation ist ein weiteres Halten der Verbindung nicht erforderlich; deshalb reicht UDP. Und wenn mal ein Paket verlorengeht, wird eben ein neues angefordert, gegebenenfalls von einem anderen Server. Der Grund für diesen Abschnitt liegt aber nicht in dieser Vorrede, sondern auf einem anderen Aspekt, der bisher nicht zum Tragen kam: Die Byte-Reihenfolge (englisch *byte order*) innerhalb eines Integer-Werts. Es gibt zwei wesentliche Varianten:

- *Big-Endian*: Das Byte mit den *höchstwertigen* Bits wird an der kleinsten Speicheradresse gespeichert.
- *Little-Endian*: Das Byte mit den *niederwertigen* Bits wird an der kleinsten Speicheradresse gespeichert.

Die Byte-Reihenfolge kann je nach Prozessortyp verschieden sein: Motorola verwendet Big-Endian, Intel hingegen Little-Endian. Bei der Übertragung von Daten ist nicht bekannt, auf welchen Rechnertyp man trifft, der die Daten interpretieren soll. Um die Kommunikation zwischen verschiedenen Computern zu ermöglichen, schreibt das Internet-Protokoll die Reihenfolge, *Network Byte Order* genannt, vor, und zwar Big-Endian. Die auf einem Rechner verwendete Reihenfolge heißt *Host Byte Order*. Sie kann mit der Network Byte Order übereinstimmen, muss es aber nicht. Zur Umwandlung gibt es die Funktionen

`htonl(u32)` : Host to Network byte order-Umwandlung (u32 = 32-Bit unsigned)

`htons(u16)` : Host to Network byte order-Umwandlung (u16 = 16-Bit unsigned)

`ntohl(u32)` : Network to Host byte order-Umwandlung (u32 = 32-Bit unsigned)

`ntohs(u16)` : Network to Host byte order-Umwandlung (u16 = 16-Bit unsigned)

Der letzte Buchstabe des Funktionsnamens steht für long bzw. short. Falls Network Byte Order und Host Byte Order übereinstimmen, tun diese Funktionen nichts. Man soll sie dennoch verwenden, weil dann der Programmcode unabhängig von der Host Byte Order geschrieben werden kann. Netzwerkadressen und Port müssen in der Network Byte Order übertragen werden. Dass man das in den obigen Beispielen nicht sieht, liegt daran, dass die Boost.Asio Library im Hintergrund dafür sorgt. Um jedoch die Zeit bei einem Zeitserver abzufragen, muss ihm ein Header im Big-Endian-Format gesendet werden, wie im »Simple Network Time Protocol« (RFC 4330, [NTP]) festgelegt. Das heißt nicht nur, dass die Anfrage entsprechend codiert sein muss, sondern es ist auch die Antwort in die Host Byte Order zu konvertieren. Beides ist im Beispiel zu sehen, das anschließend im Einzelnen erläutert wird.

Listing 15.8: Atomuhr mit UDP abfragen

```
1  // cppbuch/k15/udpsocket/ntp.cpp
2  #include<iostream>
3  #include<boost/asio.hpp>
4  #include<cassert>
5  #include<ctime>
6  using boost::asio::ip::udp;
7
8  int main() {
9    boost::asio::io_service ioService;
10   udp::socket socket(ioService);
11   assert(sizeof(size_t) == 4);
12   size_t buf[12] = {0};
13   const size_t BUFSIZE = sizeof(buf);
14   buf[0] = htonl ( ( 3 << 27) | (3 << 24)); // siehe RFC 4330 [NTP]
15   udp::resolver resolver(ioService);
16   udp::resolver::query query(udp::v4(), "europe.pool.ntp.org", "123");
17   udp::endpoint zeitserver = *resolver.resolve(query);
18   socket.open(boost::asio::ip::udp::v4());
19   socket.send_to(boost::asio::buffer(buf, BUFSIZE), zeitserver);
20
21   boost::asio::ip::udp::endpoint hier;
22   socket.receive_from(boost::asio::buffer(buf, BUFSIZE), hier);
23   time_t secs = ntohl(buf[8]) - 2208988800u;
24   tm *z = localtime(&secs);
25   double secfrac = (double)ntohl(buf[9])/4294967296.0;
26   std:: cout << z->tm_mday << '.' << z->tm_mon + 1 << '.'
27             << z->tm_year + 1900 << " "
28             << z->tm_hour << ':' << z->tm_min << ':'
29             << secfrac + z->tm_sec << std::endl;
30 }
```

Erläuterungen zu den einzelnen Zeilen:

11 Das Programm funktioniert nur auf einem System, in dem ein `int` bzw. `size_t` 32 Bits = 4 Bytes lang ist. Wenn nicht, muss das Programm angepasst werden. Bei einem neueren Compiler könnte hier das auf Seite 134 beschriebene `static_assert` eingesetzt werden, um eine entsprechende Meldung nicht erst nach Start des Programms, sondern schon während der Compilation zu erhalten.

12 `buf` nimmt erst die Anfrage und nachher die Antwort auf. Nach [NTP] sind mindestens 12 32-Bit-Felder vorgesehen.

14 Version und Modus (Details siehe [NTP]) werden entsprechend codiert und vor Zuweisung an `buf[0]` in die Network Byte Order umgewandelt. Die anderen Felder haben ihre Bedeutung, können jedoch für eine minimale Anfrage leer bleiben.

15 In den vorangegangenen Abschnitten dieses Kapitels wird mit der IP-Adresse operiert. Die Angabe des Hostnamens ist aber bequemer und aus folgenden Gründen oft besser geeignet als die Angabe einer festen IP-Adresse:

- Einem Namen können mehrere IP-Adressen zugewiesen sein, um die Last zu verteilen.

- Ein Server kann umgezogen sein, d.h. er hat seinen Namen behalten, aber seine IP-Adresse hat sich geändert.

Das Objekt `resolver` löst Namen und Port in einen Iterator auf einen Endpunkt auf und erledigt damit eine ähnliche Aufgabe wie das Programm auf Seite 476. Der `resolver` gibt immer einen gültigen Iterator zurück – oder er wirft eine Exception.

16 Das Protokoll für den Zeitdienst ist fest dem Port 123 zugeordnet. Die angegebene Adresse *europe.pool.ntp.org* ist ein Server, an den viele weitere angeschlossen sind und der die Anfrage an einen von diesen weiterleitet (Schicht »andere« im Sinne der auf Seite 485 genannten Schichten). Es ist sinnvoll, diese Adresse zu benutzen, um die Haupt-Zeitserver zu entlasten. Einer davon ist der zur Schicht 1 gehörende Server *ptbtime1.ptb.de* der Physikalisch-Technischen Bundesanstalt in Braunschweig, deren Atomuhren alle Funkwecker und Bahnhofsuhren in Deutschland steuern.

17 Der Iterator wird dereferenziert und ergibt den Endpunkt.

18 Der Socket wird für das IPv4-Protokoll geöffnet.

19 Die Anfrage wird an den Zeitserver verschickt.

22 Die Antwort wird gelesen.

23 `buf[8]`, in die Host Byte Order umgewandelt, enthält die Anzahl der seit dem 1.1.1900 verstrichenen Sekunden. Weil die Anfangszeit aller Computer-Systeme sich auf den 1.1.1970 bezieht, muss die Differenz, also 70 Jahre in Sekunden, abgezogen werden. Nach Angabe des RFC 4330-Autors sind das 2208988800 Sekunden, wie auf seiner Seite (*http://www.cis.udel.edu/~mills/y2k.html*) nachgelesen werden kann.

24 Die berechnete Anzahl der Sekunden wird in eine `tm`-Struktur umgewandelt, wie von Seite 335 bekannt.

25 `buf[9]`, in die Host Byte Order umgewandelt, enthält den Sekundenbruchteil als 32-Bit-Feld, weswegen durch 2^{32} = 4294967296 geteilt wird. Das letze Bit entspricht etwa 233 Pico-Sekunden. Die extrem hohe Auflösung ist rein hypothetisch, weil allein durch die Netzübertragung eine Genauigkeit von weniger als einer Sekunde kaum erreichbar ist. Der Sekundenbruchteil spielt nur dann eine wesentliche Rolle, wenn die Übertragungszeit herausgerechnet wird (siehe unten).

26 Datum und Uhrzeit werden ausgegeben.

29 Zum Wert der Sekunde wird der errechnete Bruchteil addiert.

Der Kürze halber wird nur ein Teil der Nachricht ausgewertet. Daher nur als Ergänzung die Information, dass sich mit Hilfe der anderen Einträge in `buf` die Dauer, wie lange die Übertragung gedauert hat, sowie der Unterschied in den Uhren von Client und Zeitserver berechnen lassen, wenn die eigene Zeit mitgeschickt wird. Wenn Sie mehr darüber wissen möchten, empfehle ich Ihnen *http://www.ntp.org* sowie [NTP]. Bitte beachten Sie bei der Nutzung der Zeitdienste die Kapitel 9 und 10 von [NTP].

15.4 HTTP

HTTP ist ein weitverbreitetes Übertragungsprotokoll auf Anwendungsebene für Hypermedia-Informationen. Es ist ein zustandsloses Protokoll, eine zweite HTTP-Anfrage weiß also nichts von der ersten, weswegen mehrere logisch zusammengehörige Informationen, die mit HTTP hin und her transportiert werden, besonderer Erkennungsmechanismen bedürfen. Ein Beispiel dafür ist ein Online-Kauf, bei dem sich erst nach und nach der virtuelle Einkaufskorb füllt und am Ende durch Angabe der Kreditkartennummer bezahlt wird. HTTP ist auch für viele andere Dinge wie zum Beispiel Nameserver nutzbar. Mit HTTP können verschiedene Rechner miteinander Datenformate »aushandeln«, und es gibt eine Menge Fehlercodes. Es benutzt verschiedene Methoden zum Datentransfer, von denen GET und POST die bekanntesten sind. Die HTTP/1.1-Spezifikation RFC 2616 ist über die Website des World Wide Web Consortiums (W3C) oder direkt von der Internet Engineering Task Force erhältlich [Fiel]. HTTP ist ein Anfrage-Antwort-Protokoll. Ein Client sendet eine Anfrage an einen Server. Die Anfrage besteht aus einer Zeile, die dem Server sagt, mit welcher Methode er die Anfrage auswerten soll (wie GET oder POST), einem URI und der Angabe des Protokolls mit Version, heutzutage HTTP/1.1. Anschließend folgt ein Kopfteil (englisch *header*) mit aus Attribut/Wert-Paaren bestehenden Zeilen zur Beschreibung des nachfolgenden Inhalts. Zum Beispiel besagen die Zeilen

```
Content-Length: 400
Content-Type: text/plain; charset=ISO-8859-1
```

dass der nachfolgende Inhalt 400 Bytes lang ist und aus schlichtem Text des Zeichensatzes ISO-8859-1 besteht. Die wichtigsten HTTP-Methoden sind GET, POST und HEAD. HEAD unterscheidet sich von GET darin, dass nur der Header, aber nicht der Inhalt vom Server zurückgesendet wird. Das erlaubt es dem Clienten, die Informationen auszuwerten und in Abhängigkeit davon die GET-Anfrage zu starten. Es könnte ja sein, dass die Content-Length so groß ist, dass bei einem mobilen Gerät (Handy, PDA) auf ein Laden verzichtet werden soll. Der Content-Type (MIME-Typ, siehe Glossar) sagt, wie der Inhalt zu interpretieren ist. Zum Beispiel steht `gzip` für einen zip-komprimierten Inhalt, und `application/x-www-form-urlencoded` heißt, dass Umlaute, Sonderzeichen und Leerzeichen durch andere Symbole ersetzt wurden. Die Regeln sind in [URI] festgelegt und relativ einfach: Alle alphanumerischen Zeichen und die Sonderzeichen `-_.!~*'()` bleiben unverändert. Alle anderen Zeichen werden hexadezimal codiert, wobei ein %-Zeichen vorangestellt wird. Nach einer Leerzeile folgt der Inhalt, falls erforderlich. Alle Zeilen enden mit CRLF bzw. \r\n, nicht nur mit LF bzw. \n.

Der Server antwortet in einem ähnlich strukturierten Format. Es enthält unter anderem einen Antwort-Code. `200 OK` ist der Code, wenn die Anfrage erfolgreich beantwortet werden kann. Der Code `404 Not Found` ist fast jedem Internet-Benutzer bekannt. Der Inhalt der Antwort ist typischerweise ein HTML-Text, wenn der Client ein Browser ist. Der Ablauf einer Interaktion zwischen HTTP-Client und HTTP-Server ist:

1. Server wartet auf eingehende HTTP-Abfrage.
2. Client erzeugt URL http://... .
3. Client baut TCP-Verbindung auf.

4. Server akzeptiert Verbindungswunsch.
5. Client sendet HTTP-Nachricht und fordert spezifizierte Ressource an.
6. Server verarbeitet die Anfrage (z.B. Datenbankabfrage und Erzeugung einer HTML-Seite mit dem Ergebnis).
7. Server sendet die geforderte Ressource (z.B. HTML-Seite oder Datei).
8. Client verarbeitet die Antwort.
9. Client und/oder Server schließen die TCP-Verbindung.

15.4.1 Verbindung mit GET

Die GET-Methode des HTTP enthält die vollständige Anforderung im URI. Die Herstellung der Verbindung löst im Server die gewünschte Reaktion aus, das Senden der Daten. Jeder hat wohl schon mal sehr lange URIs im Adressfenster eines Browsers gesehen. Häufig sind es URIs, die Parameter enthalten. Dabei folgt der eigentlichen Adresse entweder ein Pfad (Verzeichnis und Dateiname) oder eine Liste von Parametern, deren Beginn durch ein `?` markiert wird. Beispiele:

```
www.hs-bremen.de/__assets/images/logo.gif
www.google.com/webhp?hl=de&q=c%2B%2B
```

In der zweiten Zeile markiert in der Google-Festlegung `?hl=de` die Sprache, `de` steht für Deutsch. Vor dem `=`-Zeichen steht der Name des Parameters, danach der Wert. Weitere GET-Parameter können folgen, abgetrennt durch jeweils ein `&`-Zeichen, zum Beispiel `&q=Wert` für Query (Anfrage). Die Parameter müssen entsprechend dem oben schon erwähnten MIME-Typ `application/x-www-form-urlencoded` codiert sein. `c%2B%2B` ist die URL-codierte Form von `C++`. Der erste Schrägstrich nach dem Host-Namen und alles, was danach kommt, heißt *Pfad*. GET muss ein Pfad übergeben werden; falls er leer ist, muss / herhalten.

Im folgenden Beispiel wird gezeigt, wie eine GET-Anfrage und die Auswertung der Antwort realisiert werden. Die Klasse `ClientAnfrage` zerlegt eine übergebene Webadresse in Host-Name und Pfad und sorgt für das Absenden der Anfrage. Falls der Pfad einen Dateinamen enthält, wird er für die Antwort verwendet, ansonsten wird die Antwort in der Datei *antwort.html* abgelegt. Weil es sein kann, dass auch binäre Daten zurückgesendet werden, etwa wenn eine Bilddatei angefordert wird, geht das Programm zweistufig vor. Es zeigt die Statusinformation und den Header der Antwort im Klartext an, der Inhalt (englisch *content*) wird jedoch in eine Datei geschrieben. Das Programm eignet sich damit zum Download einer Datei. Die verschiedenen möglichen Codes für den Status sind in [Fiel], Abschnitt 10, definiert. Der Kürze wegen analysiert `main()` nur die Fehlerklassen:

Listing 15.9: GET-Abfrage

```
// cppbuch/k15/http/get/main.cpp
#include<iostream>
#include"ClientAnfrage.h"
#include"AntwortAuswerter.h"
using namespace std;

int main(int argc, char* argv[]) {
  if(argc != 2) {
```

```
    cout << "Gebrauch: " << argv[0]
         << " WWW –Adresse (ohne http:// eingeben)" << endl;
    return 1;
  }
  ClientAnfrage clientAnfrage(argv[1]);
  AntwortAuswerter antwortAuswerter(clientAnfrage);
  clientAnfrage.send();
  antwortAuswerter.receive();
  cout << "Header:\n" << antwortAuswerter.getHeader() << endl;
  cout << "Status (200=OK): " << antwortAuswerter.getStatus() << " ";
  string stat;
  switch(antwortAuswerter.getStatus()/100) {
  case 2 : stat = "erfolgreich"; break;
  case 3 : stat = "Umleitung u.a."; break;
  case 4 : stat = "Clienten–Fehler"; break;
  case 5 : stat = "Server–Fehler"; break;
  default: stat = "darf nicht vorkommen"; break;
  }
  cout << stat << endl;
  if(antwortAuswerter.getStatus() >= 300) {
    cout << "siehe Datei " << clientAnfrage.getDateiname() << endl;
  }
  if(antwortAuswerter.getHeader().find("chunked") != string::npos) {
    cout << "Transfer–Encoding: chunked nicht implementiert\n"
       "nur 1 Chunk gelesen, siehe "
         << clientAnfrage.getDateiname() << endl;
  }
}
```

Zur letzten Anweisung: Es gibt den Modus, die Antwort häppchenweise (englisch *chunked*) zu versenden. Der Kürze halber ist dieser Modus nicht implementiert, wie auch andere Dinge nicht, wie etwa bei Umleitung die neue Adresse anzusprechen. Alle in [Fiel] aufgeführten Möglichkeiten zu realisieren, würde den hier sinnvollen Umfang sprengen.

Der Konstruktor der folgenden Klasse `ClientAnfrage` ermittelt aus der übergebenen Web-Adresse den Host-Namen und den Pfad und stellt daraus gemäß [Fiel], Abschnitt 5.1.2, die GET-Anfrage zusammen. Die Anfrage muss mit einer Leerzeile enden. »Connection: close« bedeutet, dass der Server die Verbindung nach Übersenden der Nachricht schließen kann ([Fiel], Abschnitt 14.10). Der Konstruktor legt unter anderem den Socket an. Damit das `AntwortAuswerter`-Objekt darauf (und andere Informationen) zugreifen kann, bekommt es in `main()` das `ClientAnfrage`-Objekt übergeben.

Listing 15.10: Klasse ClientAnfrage

```
// cppbuch/k15/http/get/ClientAnfrage.h
#ifndef CLIENTANFRAGE_H
#define CLIENTANFRAGE_H
#include<string>
#include<iostream>
#include<boost/asio.hpp>
#include<cstring> // strchr()
#include<cctype>  // isalnum()
```

```
namespace {
   std::string urlencode(const std::string& s) {
      std::string ergebnis;
      for(size_t i = 0; i < s.length(); ++i) {
         char c = s[i];
         if(isalnum(c) || strchr("-_.!~*'()", c)) { // RFC 3986, 2.3
            ergebnis += c;
         }
         else {
            ergebnis += '%';
            size_t ic = c/16;
            if(ic < 10) ic += 48; // 0..9
            else        ic += 55; // A..F
            ergebnis += (char)ic;
            ic = c%16;
            if(ic < 10) ic += 48; // 0..9
            else        ic += 55; // A..F
            ergebnis += (char)ic;
         }
      }
      return ergebnis;
   }
}

class ClientAnfrage {
public:
   ClientAnfrage(const std::string& wwwseite)
      : socket(ioService), gesendet(false) {
      std::string pfad;
      std::string hostname;
      size_t schraegstrichPosition = wwwseite.find("/");
      if(schraegstrichPosition == std::string::npos) { // nicht vorhanden
         hostname = wwwseite;
         pfad = "/";
      }
      else { // extrahieren:
         hostname = wwwseite.substr(0, schraegstrichPosition);
         pfad = wwwseite.substr(schraegstrichPosition);
      }
      // neu
      std::string getQuery("");
      size_t queryPosition = pfad.find("?");
      if(queryPosition != std::string::npos) { // vorhanden
         // Query url-encodieren:
         getQuery = "?" + urlencode(pfad.substr(queryPosition+1));
         pfad = pfad.substr(0, queryPosition);
      }
      anfrage = "GET "; // s. RFC 2616 Kap. 5.1.2
      anfrage += pfad + getQuery + " HTTP/1.1\r\nHost: " + hostname
         +"\r\nAccept: */*\r\nConnection: close\r\n\r\n"; // Extra-Leerzeile

      dateiname = "antwort.html"; // Vorgegebener Dateiname für dieses Programm
      boost::asio::ip::tcp::resolver resolver(ioService);
```

```
      boost::asio::ip::tcp::resolver::query query(hostname, "http");
      server = *resolver.resolve(query);
   }

   void send() {
      // Anfrage senden
      socket.connect(server);
      socket.write_some(boost::asio::buffer(anfrage.c_str(), anfrage.length()));
      gesendet = true;
   }
   boost::asio::ip::tcp::socket& getSocket() {
      return socket;
   }
   const std::string& getDateiname() const {
      return dateiname;
   }
   bool istGesendet() const {
      return gesendet;
   }
private:
   boost::asio::io_service ioService;
   boost::asio::ip::tcp::socket socket;
   boost::asio::ip::tcp::endpoint server;
   std::string anfrage;
   std::string dateiname;
   bool gesendet;
};
#endif
```

Ein `AntwortAuswerter`-Objekt liest die Antwort mit `receive()` und wertet sie aus. Der Header ist stets durch eine extra Leerzeile vom Inhalt getrennt, was die Erkennung vereinfacht. Oft wird die Menge der zu übertragenen Daten im Header als »Content-Length« übertragen. Das ist aber nicht garantiert und in manchen Fällen sinnlos (wie etwa bei »Transfer-Encoding: chunked«). Deshalb ist es am besten, wie auf Seite 481 vorzugehen: einen Puffer fixer Kapazität anzulegen und wiederholt zu lesen, bis die `error`-Variable das Ende des Empfangs meldet.

Listing 15.11: Klasse AntwortAuswerter

```
// cppbuch/k15/http/get/AntwortAuswerter.h
#ifndef ANTWORTAUSWERTER_H
#define ANTWORTAUSWERTER_H
#include<string>
#include<cstring>
#include<cstdlib>
#include<boost/asio.hpp>
#include<fstream>
#include"ClientAnfrage.h"

class AntwortAuswerter {
public:
   AntwortAuswerter(ClientAnfrage& r)
      : clientAnfrage(r), socket(r.getSocket()), dateiname(r.getDateiname()),
```

```
      header("noch undefiniert"), status(-1) {
  }

  void receive() {
    if(!clientAnfrage.istGesendet()) {
      clientAnfrage.send();
    }
    std::ofstream ausgabe; // zur Speicherung des Contents
    ausgabe.open(dateiname.c_str(), std::ios::binary | std::ios::out);
    size_t anzahl;
    const size_t SIZE = 4096;
    char buf[SIZE];
    do {
      boost::system::error_code error;
      anzahl = socket.read_some(boost::asio::buffer(buf, SIZE), error);
      if(error == boost::asio::error::eof) {
        break;            // kein Fehler, normales Ende
      }
      else { // Puffer auswerten
        if(status == -1) {
          std::string tmp(buf, anzahl);
          size_t headerEnde = tmp.find("\r\n\r\n");
          header = tmp.substr(0, headerEnde);
          int startInhalt = headerEnde + 4; // CRLF überspringen
          // restliche Bytes wegschreiben
          int rest = (int)anzahl - startInhalt;
          if(rest > 0) {
            ausgabe.write(buf + startInhalt, rest);
          }
          status = atoi(buf + header.find(' ') + 1);
        }
        else {
            ausgabe.write(buf, anzahl);
        }
      }
    } while(anzahl > 0);
    ausgabe.close();
  }

  int getStatus() const {
    return status;
  }
  const std::string& getHeader() const {
    return header;
  }
private:
  ClientAnfrage& clientAnfrage;
  boost::asio::ip::tcp::socket& socket;
  const std::string& dateiname;
  std::string header;
  int status;
};
#endif
```

15.4.2 Verbindung mit POST

Auch POST ist eine Methode des HTTP. Im Unterschied zu GET ist die vollständige Anforderung nicht im URI enthalten, sondern sie wird im Inhaltsteil der Anfrage übertragen. Dadurch ist die Anforderung nicht in der Browser-Adresszeile sichtbar. Es können letztlich beliebige Daten gesendet werden. Eine häufige Anwendung ist die Übertragung von Formulareinträgen, wie Vorname, Nachname usw. Der Inhaltsteil besteht dann üblicherweise aus Attribut/Wert-Paaren der Form `attribut=wert`. Mehrere Paare werden bei der Übertragung durch das &-Zeichen getrennt. Die Codierung der Werte ist wie von oben bekannt `application/x-www-form-urlencoded`.

15.5 Mini-Webserver

Um das Zusammenspiel zwischen Internet-Browser und Server zu zeigen, beschreibe ich abschließend einen Mini-Webserver, der zwei einfache Aufträge erledigen kann: Anzeige des aktuellen Datums und Anzeige einer Begrüßung nach Eingabe des Vornamens. Der Funktionsumfang ist zwar minimal verglichen mit üblichen Webservern, dafür ist er aber sehr klein und kann die wesentlichen Funktionen demonstrieren. Wenn der Server mit *server.exe 9090* gestartet und der Browser auf die Webadresse gerichtet wird, ist im Browser die Abbildung 15.1 zu sehen. Bei einer lokalen Anwendung ist die Adresse *http://localhost:9090/*. 9090 ist die Portnummer.

Abbildung 15.1: Erscheinungsbild des Mini-Webservers im Browser

Klicken auf den »Datum«-Button zeigt Datum und Uhrzeit an. Eingabe eines Namens und Bestätigung mit der Return- bzw. Enter-Taste lässt eine Begrüßung erscheinen. Gleich-

zeitig protokolliert der Server die eingehenden Anfragen und die ausgehenden Header. Das `main()`-Programm erzeugt den Server und startet ihn mit `run()`:

Listing 15.12: Hauptprogramm des Servers

```
// cppbuch/k15/http/server/main.cpp
#include<iostream>
#include<cstdlib>
#include"Server.h"

int main(int argc, char* argv[]) {
   if(argc != 2) {
      std::cout << "Gebrauch: " << argv[0] << " <port>" << std::endl;
      return 1;
   }
   unsigned short port = atoi(argv[1]);
   Server server(port, "web");
   server.run();
}
```

Dem Server wird das Verzeichnis *web* übergeben; es enthält die Datei *index.html* und das angezeigte Bild. *index.html* enthält zwei Formular-Tags:

Listing 15.13: index.html

```
<html>
<head><title>C++ MiniWebServer</title></head>
<body>
 <img src="bild.png" width="80%"<br>
 <hr>
 <p>Auftrag mit Button-Klick absenden:
 <form action="" method="GET">
   <input class="button" type="submit" name="zeitabfrage" size="12"
          value="Datum"/>
 </form>
 </p>
 <p> Vornamen eingeben und mit ENTER best&auml;tigen:
  <form action="" method="GET">
   <input name="begruessung" type="text" value="" maxlength="40"/>
  </form>
  </p>
</body>
</html>
```

HTML

Einige wenige Basiskenntnisse in HTML werden für das Beispiel vorausgesetzt. Sollten diese Kenntnisse nicht vorhanden sein, bitte ich Sie, sich bei *http://de.selfhtml.org/* zu informieren, der besten deutschsprachigen Webseite zu HTML.

Die Klasse `Server` verwaltet die Verbindungen. Eine Abbruchmöglichkeit außer der Eingabe von Strg+C ist nicht vorgesehen. In der Methode `run()` wird das Objekt `conn` vom

Typ `HttpConnection` erzeugt. `conn.accept()` wartet auf eine eingehende Verbindung. Ist sie zustandegekommen, wird `conn.operator()()` ausgeführt.

Listing 15.14: Klasse Server

```
// cppbuch/k15/http/server/Server.h
#ifndef SERVER_H
#define SERVER_H
#include"HttpConnection.h"

class Server {
public:
   Server(int p, const std::string& v)
      : port(p), verzeichnis(v),
        acceptor(ioService,
                boost::asio::ip::tcp::endpoint(boost::asio::ip::tcp::v4(),
                port)) {
   }

   void run() {
      while(true) {     // Abbruch mit Strg+C
         std::cout << "lauschen an Port " << port
                   << " ... (Abbruch mit Strg+C)" << std::endl;
         HttpConnection conn(acceptor, verzeichnis);
         conn.accept();          // warten auf eingehende Verbindung
         conn();                 // bearbeiten
      }
   }
private:
   int port;
   const std::string verzeichnis;
   boost::asio::io_service ioService;
   boost::asio::ip::tcp::acceptor acceptor;
};
#endif
```

Die vom Server-Objekt aufgerufene Funktion `operator()()` liest die Daten ein und analysiert, ob es sich um eine Anfrage oder um eine gewünschte Datei handelt. Im ersten Fall wird die Aufgabe dem `AnfrageHandler` übergeben, im zweiten Fall dem `DateiHandler`:

Listing 15.15: Klasse HttpConnection

```
// cppbuch/k15/http/server/HttpConnection.h
#ifndef HTTPCONNECTION_H
#define HTTPCONNECTION_H
#include<iostream>
#include<string>
#include<boost/asio.hpp>
#include"AnfrageHandler.h"
#include"DateiHandler.h"

class HttpConnection {
public:
   HttpConnection( boost::asio::ip::tcp::acceptor& a, const std::string& v)
```

```
      : acceptor(a), socket(a.get_io_service()), verzeichnis(v) {
   }

   void accept() { acceptor.accept(socket); }

   void operator()() {
      const size_t BUFSIZE = 1024;
      char daten[BUFSIZE+1] = {0};
      // Anfrage lesen
      size_t anzahl = socket.read_some(boost::asio::buffer(daten, BUFSIZE));
      daten[anzahl] = '\0';
      std::cout << daten << std::endl;
      std::string header(daten);
      size_t leerzeichen = header.find(' ');
      // Fehlermeldung bei falschem Protokoll
      if(header.substr(0, leerzeichen) != "GET") {
         // Header zusammenbauen und senden
         HttpResponse httpResponse(400);
         httpResponse.addToHeader("Content-Type", "text/html");
         std::string msg = httpResponse.getHTMLMessage(
                                    "Protokoll nicht implementiert!");
         httpResponse.addToHeader("Content-Length", i2string(msg.length()));
         httpResponse.sendHeader(socket);
         // msg senden (html-Seite)
         socket.write_some(boost::asio::buffer(msg.c_str(), msg.length()));
         return;                                              // Abbruch!
      }
      size_t leerzeichen2 = header.find(' ', leerzeichen+1);
      std::string pfad = header.substr(leerzeichen+1,
                                  leerzeichen2-leerzeichen-1);
      // Gibt es eine Anfrage oder ist eine Datei gewünscht?
      size_t qpos = pfad.find('?');
      if(qpos != std::string::npos) {                         // Anfrage
         AnfrageHandler ah(socket, pfad.substr(qpos+1));
         ah.process();
      }
      else {                                                  // Datei
         std::string dateiname = verzeichnis + pfad;
         if(pfad == "/" || pfad == "/favicon.ico") {
            dateiname = verzeichnis + "/index.html";
         }
         DateiHandler dh(socket, dateiname);
         dh.process();
      }
      socket.close();
   }
private:
   boost::asio::ip::tcp::acceptor& acceptor;
   boost::asio::ip::tcp::socket socket;
   std::string verzeichnis;
};
#endif
```

Die Klasse `HttpResponse` bereitet den HTTP-Header vor und sendet ihn gegebenenfalls. Sie wird oben in der Reaktion auf ein nicht implementiertes Protokoll und auch in den nachfolgenden Klassen `AnfrageHandler` und `DateiHandler` verwendet.

Listing 15.16: Klasse HttpResponse

```
// cppbuch/k15/http/server/HttpResponse.h
#ifndef HTTPRESPONSE_H
#define HTTPRESPONSE_H
#include<string>
#include<iostream>

namespace {
   std::string i2string(int i) { // wandelt int in einen String um
      // Alternative: boost::lexical_cast aus Abschnitt 24.1.3
      std::string ergebnis;
      if(i == 0) {
         ergebnis = "0";
      }
      else {
         bool negativ = (i < 0);
         while(i != 0) {
            ergebnis.insert(ergebnis.begin(), abs(i % 10) + '0');
            i /= 10;
         }
         if(negativ) {
            ergebnis.insert(ergebnis.begin(), '-');
         }
      }
      return ergebnis;
   }
}

class HttpResponse {
public:
   HttpResponse(int st)
      : status(st) {
      std::string zeile("HTTP/1.1 ");
      zeile += i2string(status);
      zeile += std::string(" ") + getStatusText() + "\r\n";
      header = zeile;
   }

   void addToHeader(const std::string& key, const std::string& value) {
      header += key + ": " + value + "\r\n";
   }

   void sendHeader(boost::asio::ip::tcp::socket& socket) {
      header += "\r\n";
      socket.write_some(boost::asio::buffer(header.c_str(), header.length()));
      std::cout << "GESENDETER HEADER:\n" << header;
   }
```

```
   std::string getHTMLMessage(const std::string& text ="") {
      std::string statusMsg(i2string(status) + " " + getStatusText());
      std::string msg("<html><head><title>");
      msg += statusMsg + "</title></head>";
      msg += "<body><h1>" + statusMsg + "</h1>" + text + "</body></html>";
      return msg;
   }
private:
   std::string getStatusText() {
      std::string txt;
      switch (status) { // reduzierte Auswahl
      case 200:  txt = "OK";          break;
      case 400:  txt = "Bad Request"; break;
      case 404:  txt = "Not Found";   break;
      case 500:  txt = "Internal Server Error";   break;
      case 501:  txt = "Not Implemented";         break;
      default:   txt = "undefined";
      }
      return txt;
   }
   int status;
   std::string header;
};
#endif
```

Die Hilfsfunktion `i2string()` wandelt den Status in einen String um, weil ein String in einem bestimmten Format als erste Zeile des Headers zurückgegeben werden muss. Dem Protokoll HTTP/1.1 folgt der Status und darauf der Text, der zu diesem Status gehört. Diesen Text liefert die Funktion `getStatusText()`, wobei hier nur sehr wenige Möglichkeiten aus [Fiel] realisiert werden. Die Funktion `getHTMLMessage()` wird nur benutzt, um dem Clienten einen Fehler als HTML-Seite anzeigen zu können. So wird zum Beispiel die Meldung »Datei nicht gefunden!« gegebenenfalls zusätzlich zum Fehlerstatus im Browser angezeigt, wie die Klasse `DateiHandler` zeigt:

Listing 15.17: Klasse DateiHandler

```
// cppbuch/k15/http/server/DateiHandler.h
#ifndef DATEIHANDLER_H
#define DATEIHANDLER_H
#include<fstream>
#include<string>
#include<boost/asio.hpp>
#include<boost/filesystem/operations.hpp>
#include"HttpResponse.h"

class DateiHandler {
public:
   DateiHandler(boost::asio::ip::tcp::socket& s, const std::string& d)
      : socket(s), dateiname(d) {
   }

   void process() {
      std::ifstream quelle(dateiname.c_str(), std::ios::in | std::ios::binary);
```

```
      if(quelle) {
         sendeDatei(quelle);
      }
      else { // Datei nicht vorhanden. Header zusammenbauen und senden:
         HttpResponse httpResponse(404);     // not found
         httpResponse.addToHeader("Content-Type", "text/html");
         std::string msg = httpResponse.getHTMLMessage("Datei nicht gefunden!");
         httpResponse.addToHeader("Content-Length", i2string(msg.length()));
         httpResponse.sendHeader(socket);
         // msg senden (html-Seite)
         socket.write_some(boost::asio::buffer(msg.c_str(), msg.length()));
      }
   }
private:
   boost::asio::ip::tcp::socket& socket;
   std::string dateiname;

   std::string getContentType() {
      std::string typ("application/octet-stream");        // Vorgabe
      // Datei-Extension ermitteln und in Großschreibung umwandeln
      size_t punkt = dateiname.rfind('.');
      if(punkt < std::string::npos) {
         std::string extension = dateiname.substr(punkt+1);
         for(size_t i = 0; i < extension.length(); ++i) {
            extension[i] = toupper(extension[i]);
         }
         // unvollständige Auswahl aus http://de.selfhtml.org/diverses/mimetypen.htm:
         if(extension == "HTML") typ = "text/html";
         else if(extension == "TXT") typ = "text/plain; charset=iso-8859-1";
         else if(extension == "JPG") typ = "image/jpeg";
         else if(extension == "PNG") typ = "image/png";
         else if(extension == "PDF") typ = "application/pdf/";
      }
      return typ;
   }

   void sendeDatei(std::ifstream& quelle) {
      boost::filesystem::path p(dateiname);
      size_t bufsize = boost::filesystem::file_size(p);
      // Header zusammenbauen und senden
      HttpResponse httpResponse(200);
      httpResponse.addToHeader("Content-Type", getContentType());
      httpResponse.addToHeader("Content-Length", i2string(bufsize));
      httpResponse.sendHeader(socket);
      // Datei senden
      char* const buf = new char[bufsize];
      size_t pos = 0;
      while(quelle.get(buf[pos++])); // Datei lesen und in buf abspeichern
      socket.write_some(boost::asio::buffer(buf, bufsize));
      delete [] buf;
   }
};
#endif
```

Falls es die gewünschte Datei gibt, wird sie gesendet. Dazu muss zunächst der Header mit dem Status 200 für OK erzeugt werden. Der Header enthält auch den Content-Type (MIME-Typ), der in Abhängigkeit vom Dateityp von der Funktion `getContentType()` ermittelt wird, sowie die Größe der Datei. Letztere ist mit Standardmitteln der Sprache C++ nicht herauszufinden; deshalb kommt die Boost-Bibliothek zum Einsatz.

Mehr zu Dateioperationen lesen Sie ab Seite 727.

Die Klasse `AnfrageHandler` analysiert die Anfrage und schickt je nach Ergebnis (Begrüßung, Zeitinformation oder Fehler) eine HTML-Seite zurück. In diesem einfachen Programm wird angenommen, dass es nur ein Schlüssel/Wert-Paar gibt. Tatsächlich sind im GET-Protokoll beliebig viele erlaubt.

Listing 15.18: Klasse AnfrageHandler

```
// cppbuch/k15/http/server/AnfrageHandler.h
#ifndef ANFRAGEHANDLER_H
#define ANFRAGEHANDLER_H
#include<ctime>
#include<cstring>
#include<string>
#include<boost/asio.hpp>
#include"HttpResponse.h"

class AnfrageHandler {
public:
  AnfrageHandler(boost::asio::ip::tcp::socket& s, const std::string& a)
    : socket(s), anfrage(a) {
    std::cout << "ANFRAGE=:" << a << std::endl;
  }

  void process() {
    size_t pos = anfrage.find('=');
    std::string key = anfrage.substr(0, pos);
    std::string val = anfrage.substr(pos+1);
    std::cout << "KEY=" << key << " VAL=" << val << std::endl;
    if(key == "zeitabfrage" && val == "Datum") {
      time_t jetzt = time(NULL);
      std::string zeitstring(ctime(&jetzt));
      // Header zusammenbauen und senden
      HttpResponse httpResponse(200);
      httpResponse.addToHeader("Content-Type", "text/html");
      std::string msg =
        "<html><head><title>C++ MiniWebServer</title></head><body>"
        "<img src=\"bild.png\" width=\"80%\"<br><hr><p>";
      msg += zeitstring + "</body></html>";
      httpResponse.addToHeader("Content-Length", i2string(msg.length()));
      httpResponse.sendHeader(socket);
      // Ergebnis als html-Seite senden
      socket.write_some(boost::asio::buffer(msg.c_str(), msg.length()));
    }
    else if(key == "begruessung") {
```

```
        // Header zusammenbauen und senden
        HttpResponse httpResponse(200);
        httpResponse.addToHeader("Content-Type", "text/html");
        std::string msg =
           "<html><head><title>C++ MiniWebServer</title></head><body>"
           "<img src=\"bild.png\" width=\"80%\"<br><hr><p><h1>Guten Tag, ";
        msg += val + "!<h1></body></html>";
        httpResponse.addToHeader("Content-Length", i2string(msg.length()));
        httpResponse.sendHeader(socket);
        // Ergebnis als html-Seite senden
        socket.write_some(boost::asio::buffer(msg.c_str(), msg.length()));
     }
     else {
        // Header zusammenbauen und senden
        HttpResponse httpResponse(400);
        httpResponse.addToHeader("Content-Type", "text/html");
        std::string msg = httpResponse.getHTMLMessage("unbekannte Abfrage!");
        httpResponse.addToHeader("Content-Length", i2string(msg.length()));
        httpResponse.sendHeader(socket);
        // Fehlermeldung als html-Seite senden
        socket.write_some(boost::asio::buffer(msg.c_str(), msg.length()));
     }
   }
private:
   boost::asio::ip::tcp::socket& socket;
   std::string anfrage;
};
#endif
```

Dieses Kapitel zeigt die wichtigsten Aspekte der Netzwerkprogrammierung in kurzer und beispielhafter Form. Etliche Themen werden jedoch ausgeklammert, von denen ich einige herausgreife:

- Es gibt die Möglichkeit des asynchronen Lesens, Schreibens und Akzeptierens von Verbindungen.
- Es kann mit `iostream`-entsprechenden Strömen gearbeitet werden.
- Das Bearbeiten einer Verbindung wird einem eigenen Thread gegeben, damit der Server sofort bereit ist, eine neue Verbindung zu akzeptieren, auch wenn die Bearbeitung der vorhergehenden noch andauert.
- In der Server-Praxis werden Thread-Pools verwendet, weil Anlegen und Zerstören eines Threads teure Operationen im Sinn der notwendigen Laufzeit sind.
- Das Server-Beispiel bietet nur fest einprogrammierte Funktionen. In der Praxis ist Flexibilität möglich.

Wenn keine oder kaum HTML-Kenntnisse vorhanden sein sollten: *http://de.selfhtml.org/* bietet eine Fülle von Informationen und Beispielen. Wenn Sie mehr zu den anderen Themen wissen möchten, empfehle ich, [asio] und [SSRB] zu Rate zu ziehen.