Verteilte Systeme

...für C++ Programmierer

TCP/IP Programmierung 1

by

Dr. Günter Kolousek

asio

- plattformübeergreifende Bibliothek für Netzwerkanwendungen
- $\rightarrow$ C++ Technical Specification ”Extensions for Networking”, dann C++23
- synchrone und asynchrone Kommunikation
  - synchron: Operationen blocken
  - asynchron: nicht blockierend, Callback
- Fehlerbehandlung entweder basierend auf
  - Exceptions
  - Fehlercodes

asio Namensräume

`asio`	Kernklassen und Kernfunktionen
`asio::ip`	Netzwerkfunktionalität
`asio::error`	Errorcodes
`asio::ssl`	SSL/TLS

IP Adressen

- `ip::address` ... versionsunabhängig
- `ip::address_v4` ... IPv4 Adresse
- `ip::address_v6` ... IPv6 Adresse

IP Adressen – 2

```cpp
#include <iostream>  // ipaddress.cpp
#include <asio.hpp>
#include <typeinfo>
using namespace std;
using namespace asio;
int main() {  // ip::address
    auto a1{ip::make_address("127.0.0.1")};
    cout << a1.to_string() << ", " << a1 << endl;
    cout << a1.is_loopback() << endl;
    cout<< a1.is_v4()<< ", "<< a1.is_v6() << endl;
    cout<< a1.to_v4()<< endl; // a1.to_v6() → exc!
}
```

```
127.0.0.1, 127.0.0.1
1
1, 0
127.0.0.1
```

IP Adressen – 3

```cpp
#include <iostream>  // ipaddress2.cpp
#include <asio.hpp>
using namespace std;
using namespace asio;
int main() {  // class address_v4
    auto a1{ip::make_address_v4("10.0.0.1")};
    cout << ip::address_v4::any() << endl;
    cout << ip::address_v4::broadcast() << endl;
    auto a3{ip::address_v4::loopback()};
    cout << a3 << ", " << a3.is_loopback() << endl;
}
```

```
0.0.0.0
255.255.255.255
127.0.0.1, 1
```

IP Adressen – 4

```
#include <iostream>  // ipaddress3.cpp
#include <asio.hpp>
using namespace std;
using namespace asio;
int main() {  // class address_v6
    auto a1{ip::make_address_v6("::1")};
    cout << a1 << ", " << a1.is_link_local();
    cout << " | " << ip::address_v6::any() << endl;
    auto a2{ip::address_v6::loopback()};
    cout << a2 << ", " << a2.is_loopback() << endl;
    ip::address a3{a2}; cout << a3.is_v6() << endl;
}
```

```
::1, 0 | ::
::1, 1
1
```

Endpunkt

Endpunkt = Adresse + Port + Protokoll (TCP, UDP, ICMP)

- `ip::icmp::endpoint`, `ip::udp::endpoint`, `ip::tcp::endpoint`
  - `endpoint(address, port)`
    - bei ICMP: `port` nicht genutzt → 0 verwenden
  - `endpoint(inet_protocol, port)`: → Server
    - `inet_protocol`: z.B. IPv4: `ip::tcp::v4()`
    - `address` = `ip::address_vX::any()`
  - `endpoint()`: meist für UDP und ICMP im Client-Betrieb
    - `address` = `ip::address_v4::any()`
    - `inet_protocol` = IPv4
    - `port` = 0, d.h. beliebiger Port bzw. kein Port (bei ICMP)
- nur für UNIX:
  - `local::stream_protocol::endpoint`, `local::datagram_protocol::endpoint`

BSD Sockets API

- `<sys/socket.h>`
- Protokollfamilie
  - `AF_LOCAL` (`AF_UNIX`)
  - `AF_INET`, `AF_INET6`
- Protokolltyp
  - `SOCK_STREAM`
  - `SOCK_DGRAM`
- Protokoll: `/etc/protocols` (or equivalent)

1	ICMP
4	IPv4
6	TCP
17	UDP
41	IPv6

Endpunkt – 2

```cpp
#include <iostream>  // endpoint.cpp
#include <asio.hpp>  // server-side
// AF_*, SOCK_STREAM, SOCK_DGRAM
#include <sys/socket.h>
using namespace std;
using namespace asio;
int main() {
    ip::tcp::endpoint ep1{
      ip::address_v4::any(), 80};
    // also:
    // ip::tcp::endpoint ep1{ip::tcp::v4(), 80};
    cout << ep1 << endl;
    cout<< ep1.address()<< ":"<< ep1.port()<< endl;
```

```
0.0.0.0:80
0.0.0.0:80
```

Endpunkt – 3

```
    auto proto{ep1.protocol()};
    cout << proto.family() << " = "
      << AF_INET << endl;
    cout << proto.type() << " = "
      << SOCK_STREAM << endl;
    cout << proto.protocol() << endl;
}
```

```
2 = 2
1 = 1
6
```

Endpunkt – 4

```cpp
#include <iostream>  // endpoint2.cpp
#include <asio.hpp>  // client-side
using namespace std;
using namespace asio;
int main() {
    string a1_str{"127.0.0.1"};

    // throw asio::system_error if ip is malformed
    ip::address a1{ip::make_address_v4(a1_str)};
    ip::tcp::endpoint ep{a1, 1234};
    cout << ep << endl;  // -> 127.0.0.1:1234
}
```

Endpunkt – 5

```
#include <iostream>  // endpoint3.cpp
#include <asio.hpp>  // client-side: error_code
using namespace std;  using namespace asio;
int main() {
    string a1_str{"127.0.0.1x"};
    error_code ec;// will be set if ip is malformed
    ip::address a1{ip::make_address_v4(a1_str,ec)};
    if (ec.value() != 0) {
        cout << "Error code = " << ec.value()
             << ". Message: " << ec.message();
        return ec.value();  }
    ip::tcp::endpoint ep(a1, 1234);
    cout << ep << endl;  }
```

```
Error code = 22. Message: Invalid argument
```

→ *alle* Operationen auch mit `error_code`!

Resolver

```
#include <iostream>  // resolver.cpp
#include <asio.hpp>
using namespace std; using namespace asio::ip;
int main() {   asio::io_context ctx;
    tcp::resolver resolver{ctx};
    // type of results: resolver::results_type
    auto results = resolver.resolve("localhost",
                                    "80");
    // type of curr is a "basic_resolver_entry"
    // → resolver::results_type::value_type
    auto curr{results.begin()};
    auto end{results.end()};
    while (curr != end) { auto entry = *curr++;
        cout << entry.endpoint() << " | "
        << entry.host_name() << endl; }}
```

```
[::1]:80 | localhost
127.0.0.1:80 | localhost
```

Resolver – 2

Wie komme ich zum Hostnamen?

```
#include <iostream>  // invresolver.cpp
#include <asio.hpp>
using namespace std;
using namespace asio;
using namespace asio::ip;
int main() {  asio::io_context ctx;
    tcp::resolver resolver{ctx};
    tcp::endpoint ep{address_v4::loopback(), 80};
    auto results = resolver.resolve(ep);
    auto entry = *results.begin();
    cout << entry.host_name() << endl; }
```

```
localhost
```

Kommunikation mittels Streams

Stream-basierter Echo-Client

```
#include <iostream>  // stream_echo_client.cpp
#include <asio.hpp>
using namespace std;
using namespace asio::ip;
int main() {  // no error handling at all
    tcp::iostream strm{"localhost", "9999"};
    if (strm) {  // connected
       strm << "ping-pong" << endl;
       string data;
       getline(strm, data);
       cout << data << endl;
       strm.close();
    } else { cout << "no connection" << endl; } }
```

Kommunikation mittels Streams – 2

Stream-basierter Echo-Server

```
#include <iostream>  // stream_echo_server.cpp
#include <asio.hpp>
using namespace std; using namespace asio::ip;
int main() {  // no error handling at all
    asio::io_context ctx;
    tcp::endpoint ep{tcp::v4(), 9999};
    tcp::acceptor acceptor{ctx, ep}; // IO object
    acceptor.listen();

    tcp::socket sock{ctx};
    acceptor.accept(sock);
    tcp::iostream strm{std::move(sock)};
    //shorter: tcp::iostream strm{acceptor.accept()};

    string data;
    strm >> data;  // also: getline(strm, data)
    strm << data;
    strm.close();  }
```

Kommunikation mittels Streams – 3

```
$ server&
$ client
ping-pong
Job 1, ”server &” has ended
$
```

Kommunikation mittels Streams – 4

- Änderungen in `stream_echo_server.cpp`
  - Entferne `strm << data;`
    - damit antwortet der Server nicht mehr
  - Füge

    ```
    while (1) { this_thread::sleep_for(1s); }
    ```

    vor dem Schließen des Streams hinzu
    - und inkludiere auch `<thread>`
    - damit wird der Stream auch nicht mehr geschlossen
- Client wird ”ewig” warten

Kommunikation mittels Streams – 5

Lösung mit einfacher Fehlerbehandlung

```
#include <iostream>  // stream_echo_client2.cpp
#include <chrono>
#include <asio.hpp>
using namespace std;  using namespace asio::ip;
int main() {
    tcp::iostream strm{"localhost", "9999"};
    strm.expires_after(10s);
    if (strm) {  // connected
       strm << "ping-pong" << endl;
       string data;    getline(strm, data);
       if (strm) cout << data << endl;
       // also:  if (strm.error())
       else cout << strm.error().message() << endl;
       strm.close();  } }
```

```
Connection timed out
```

Kommunikation mittels Streams – 6

- Details
  - `strm.expires_after(chrono::seconds{10});`
    - Operationen auf Streams brechen nach Zeitablauf mit `asio::error::operation_aborted` (Typ `asio::error_code`) ab und Stream geht in Fehlerzustand
  - `strm.error()` liefert Fehlercode vom Typ `asio::error_code` zurück
    - `message()` liefert Fehlertext
- Verwendung von `tcp::iostream`
  - Für einfache Anwendungen
  - Wenn Kommunikation nicht das Hauptfeature ist
  - Abstraktion mittels Streams ausreichend

TCP/UDP Protokoll – daytime

- fragt lokale Zeit des Servers ab
- Antwort als String (ASCII); Format nicht definiert!
- TCP Port 13: Verbindung aufbauen, Antwort lesen, Verbindung schließen, fertig
  - $\rightarrow$ `/etc/services`
- UDP Port 13: leeres Paket senden, Antwort lesen, fertig
- $\rightarrow$ time.nist.gov
  - z.B. mit dem Tool `socat`

```
$ socat - TCP4:time.nist.gov:13

58083 17-11-26 20:16:01 00 0 0 923.8 UTC(NIST) *
```

- Overhead! Format! Genauigkeit! $\rightarrow$ NTP

TCP/UDP Protokoll – time

- fragt lokale Zeit des Servers ab
- Antwort Zeit in Sekunden seit 1.1.1900 UTC als 4 Bytes
- TCP/UDP Port 37
- $\rightarrow$ time.nist.gov
- Genauigkeit! $\rightarrow$ NTP

TCP/UDP Protokoll – time – 2

```
import socket, struct, time
PORT = 8037
TIME1970 = 2208988800  # sec: 1.1.1900 - 1.1.1979
serversock = socket.socket(socket.AF_INET,
                           socket.SOCK_STREAM)
serversock.bind(("", PORT))
serversock.listen(3)  # size of backlog queue

while True:
    clientsock, clientaddr = serversock.accept()
    print("Verbindung von:", clientaddr)
    t = int(time.time()) + TIME1970

    # pack into 4 byte integer network-byte-order (!)
    t = struct.pack("!I", t)

    clientsock.send(t)
    clientsock.close()
```

TCP/UDP Protokoll – time – 3

```
import socket, struct, time, datetime
PORT = 8037
# PORT = 37  # if using a real one, e.g. time.nist.gov
TIME1970 = 2208988800  # sec: 1.1.1900 - 1.1.1979

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

sock.connect(("", PORT))
# sock.connect(("time.nist.gov", PORT))

t = sock.recv(4)

t = struct.unpack("!I", t)[0] - TIME1970

sock.close()

print(datetime.datetime.fromtimestamp(t))
```

TCP Protokoll – finger

- fragt Benutzerinfos ab
- TCP Port 79: Client sendet Liste von Benutzernamen (optional) und danach CRLF, Server sendet Infos
- $\rightarrow$ Request/Response Protokoll
- eigentliche Spezifikation ist umfangreicher, aber…
  - Sicherheit!
    - daher abgeschalten/blockiert

TCP/UDP Protokoll – discard & echo

- discard
  - Funktion wie `/dev/null`
  - TCP/UDP Port 9
- echo
  - TCP/UDP Port 7
  - ursprünglich zum Testen und zum Messen der RTT (round-trip times)