Verteilte Systeme

...für C++ Programmierer TCP/IP Programmierung 2 - Clients (synchron)

by

Dr. Günter Kolousek

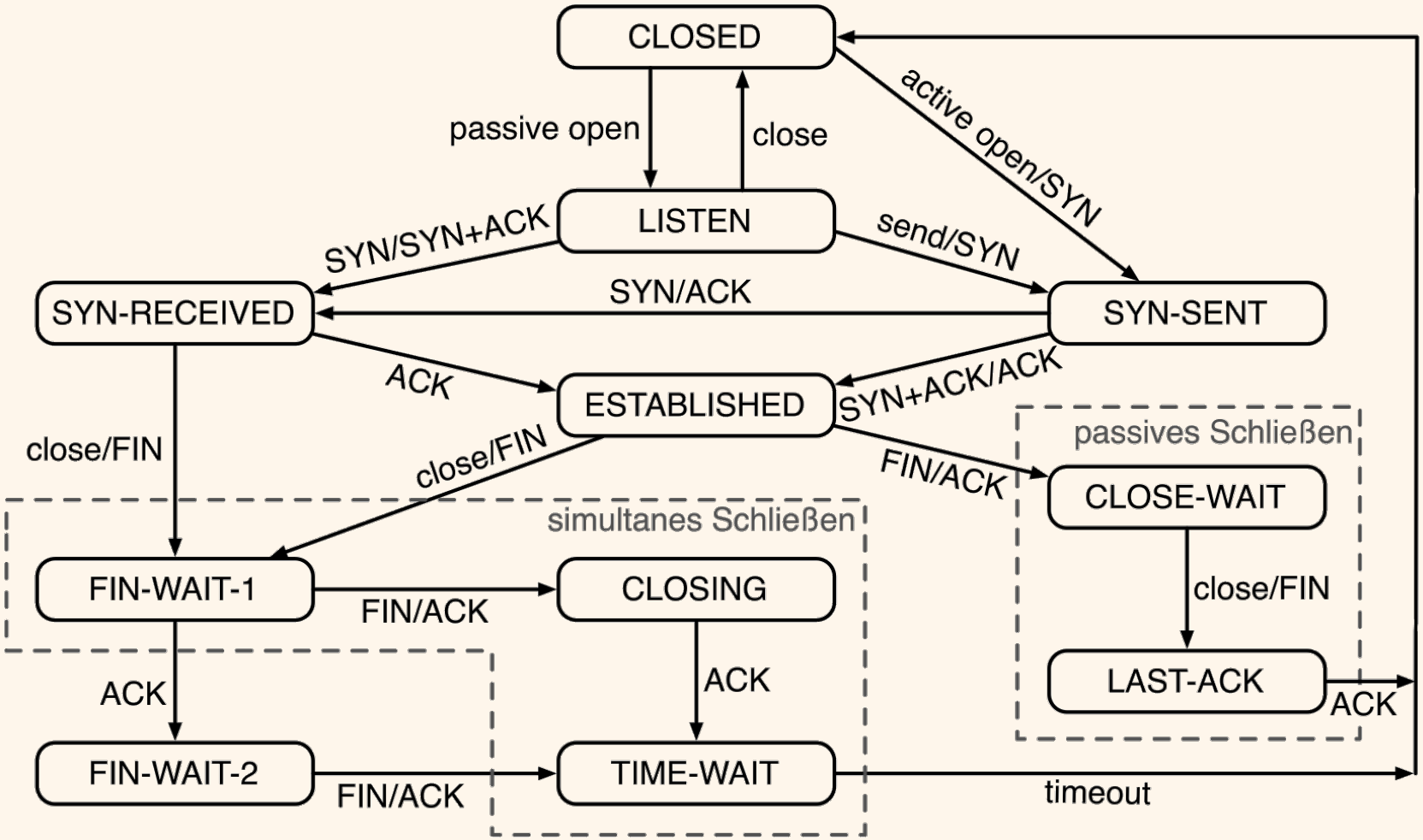
Wiederholung – Server

```
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()
```

Wiederholung – Client

```
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 Zustandsdiagramm



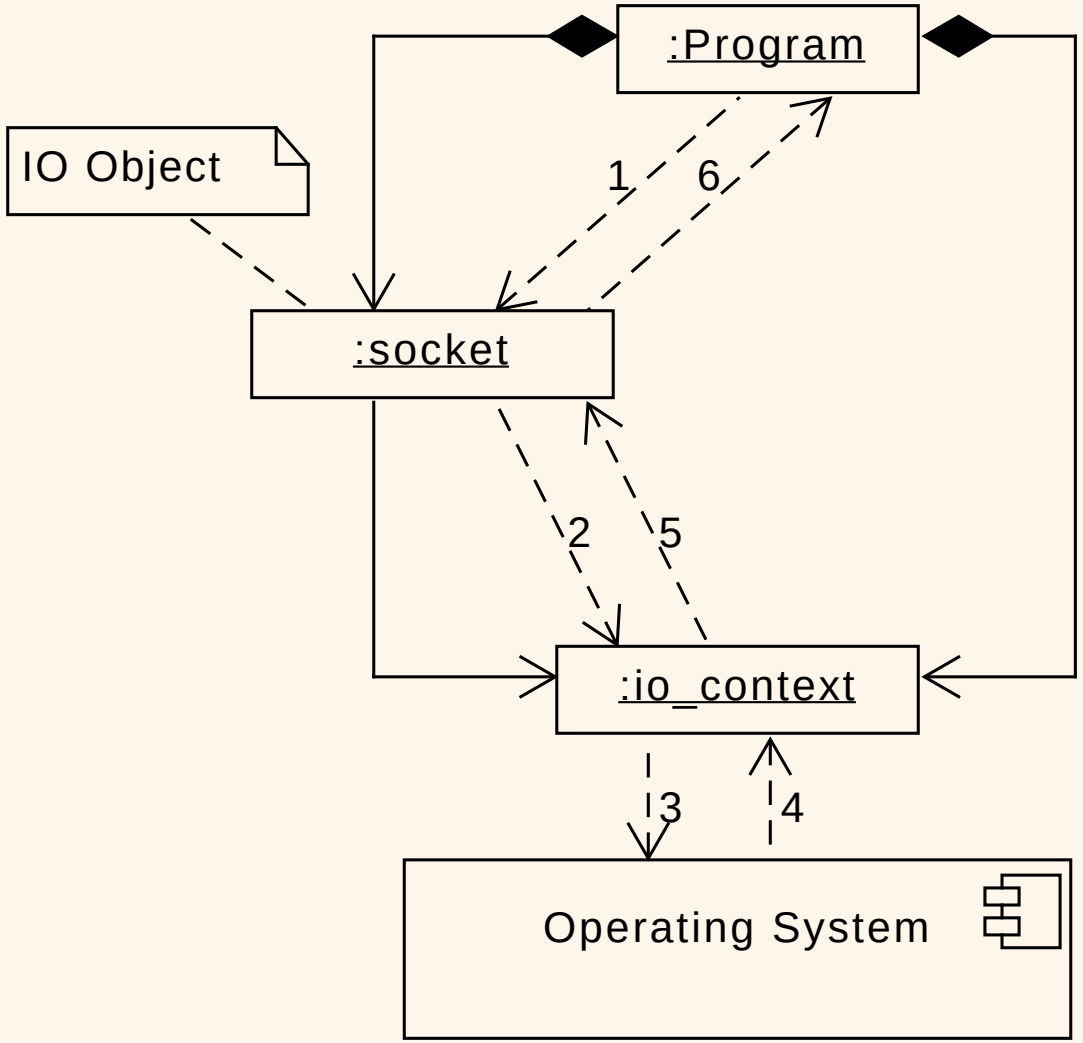
Berkeley-Sockets API – TCP-Client

1. Socket anlegen (`socket()`)
2. Port an Socket binden (`bind()`)
   - Am Client nur, wenn *über* einen bestimmten Port die Daten gesendet werden sollen
3. Verbinden (`connect()`)
4. Senden (`send()`)
5. Empfangen (`recv()`)
6. Schließen (`close()`)

Berkeley-Sockets API – TCP-Server

1. Socket anlegen (`socket()`)
2. Port an Socket binden (`bind()`)
3. Bereit zum Empfangen (`listen()`)
4. Annehmen einer Verbindung (`accept()`)
5. Senden (`send()`)
6. Empfangen (`recv()`)
7. Schließen (`close()`)

Synchrone Kommunikation mit asio



Aktive Sockets (Client)

1. Anlegen

```
tcp::socket sock{ctx};
```

2. Öffnen

```
// also with error_code (as usually)
sock.open(tcp::v4());
// 1 & 2:
// tcp::socket sock{ctx, tcp::v4()};
// but this may throw an exception!
```

3. Setzen von Optionen (optional)

```
// e.g.: TCP_NODELAY: disables Nagle algo
sock.set_option(tcp::no_delay{true});
```

Aktive Sockets (Client) – 2

4. Explizites Binden (*meist* nur bei Servern → `acceptor`)

```
sock.bind(tcp::endpoint(tcp::v4(), 1234));
// 1 & 2 & 4:
// tcp::socket sock{ctx,
//   tcp::endpoint(tcp::v4(), 1234)};
```

5. Verbinden

```
sock.connect(make_address("127.0.0.1"), 80);
```

implizites Öffnen & Binden inklusive! (d.h. Schritt 2 & 4)

Active Sockets (Client) – 3

6. Senden/Empfangen
   - `sock.send()` … sendet mind. ein Byte
     - und liefert # der gesendeten Bytes
   - `write(sock,...)` … sendet alle Daten
   - `sock.receive()` … empfängt mind. ein Byte
   - `read(sock,...)` … empfängt angegebene Datenmenge
   - `read_until(sock,...)` … bis angegebenes Zeichen
7. Schließen
   - `sock.shutdown(...)` … shutdown einer Richtung:
     `sock.shutdown(tcp::socket::shutdown_send);`
     weiters: `shutdown_receive`, `shutdown_both`
   - `sock.close()` … Socket schließen

”buffer”-Objekte

- ”buffer”-Objekt: Speicherregion als ein Tupel von Zeiger und Länge
  - auch: Array von POD, `vector` von POD, `std::string`
    - POD (plain old data): skalare Typen; Arrays von POD; Klasse, die nur nicht-statische POD Members enthält und keine vom Benutzer zur Verfügung gestellten Konstruktoren, keine Initialisierungslisten, keine Basisklasse, keine virtuellen Funktionen
- Zwei Arten
  - `const_buffer`, wenn `(const void*, size_t)`
    - d.h. zum Senden
  - `mutable_buffer`, wenn `(void*, size_t)`
    - d.h. zum Empfangen
    - kann implizit in einen `const_buffer` konvertiert werden

”buffer”-Objekte – 2

- Das asio-API verwendet allerdings **direkt** weder `const_buffer` noch `mutable_buffer`, sondern Objekte, die den Anforderungen von
  - `ConstBufferSequence` bzw.
    - repräsentiert Sequenz von `const_buffer`’s Objekten
    - `const_buffer` erfüllt `ConstBufferSequence` Anforderung
  - `MutableBufferSequence` (analog) genügen.
    - `const_buffer` erfüllt `ConstBufferSequence` Anforderung
- `buffer()`
  - Funktion, die `ConstBufferSequence` bzw. `MutableBufferSequence` anlegt
  - verschiedene überladene Funktionen!
- ”buffer”-Objekte übernehmen nicht die ’ownership’!

Synchroner Fix-Echo-Client

synchroner Echo-Client mit Puffer **fixer Größe**

```
#include <iostream>  // sync_echo_client1.cpp
#include <asio.hpp>
using namespace std;  using namespace asio::ip;
int main() { // simple error handling with exc.
    asio::io_context ctx;
    tcp::resolver resolver{ctx};
    try {
        auto results =
          resolver.resolve("localhost", "9999");
        tcp::socket sock{ctx};  // IO object
 // try each endpoint until connected (blocking):
        asio::connect(sock, results); //→ function!
        cout << "connected" << endl;
```

Synchroner Fix-Echo-Client – 2

```
        const char request[]{"ping-pong"};
        size_t request_length = strlen(request);
        asio::write(sock,
          asio::buffer(request, request_length));
        cout << "sent" << endl;

        char reply[20];
        size_t reply_length = asio::read(sock,
          asio::buffer(reply, request_length));

        cout << "reply is: ";
        cout.write(reply, reply_length);
        cout << "\n";
    } catch (asio::system_error& e) {
        cerr << e.what() << endl; } }
```

Wiederholung – Server

```cpp
#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();  }
```

Synchroner Fix-Echo-Client – 3

- Ausführung mit `stream_echo_server`
- Starten von Client → Client beendet sich nicht
- Wird Client abgebrochen, beendet sich auch Server
- Es kommt zu folgender Ausgabe:

```
$ server&
$ client
connected
sent
# pressing CTRL-C now
'server&' has ended
```

- Warum?

Synchroner Fix-Echo-Client – 3

- Ausführung mit `stream_echo_server`
- Starten von Client → Client beendet sich nicht
- Wird Client abgebrochen, beendet sich auch Server
- Es kommt zu folgender Ausgabe:

```
$ server&
$ client
connected
sent
# pressing CTRL-C now
'server&' has ended
```

- Warum?
  - Weil kein \n vom Client gesendet wird!

Synchroner Fix-Echo-Client – 4

```
    // sync_echo_client2.cpp
    const char request[]{"ping-pong\n"};  // ←
    size_t request_length = strlen(request);

    asio::write(sock, asio::buffer(request,
      request_length));
    cout << "sent" << endl;

    char reply[20];
    size_t reply_length = asio::read(sock,
      asio::buffer(reply, request_length));
    cout << "Reply is: ";
    cout.write(reply, reply_length);
    // → no output of \n necessary, isn't it?
} catch (asio::system_error& e) {
    std::cerr << e.what() << std::endl; } }
```

Synchroner Fix-Echo-Client – 5

- Ausführung mit `stream_echo_server`
- Starten von Client → Client beendet sich!
- Server beendet sich
- Es kommt zu folgender Ausgabe:

```
$ server&
$ client
connected
sent
read: End of file
'server&' has ended
```

- Warum?

Synchroner Fix-Echo-Client – 5

- Ausführung mit **`stream_echo_server`**
- Starten von Client → Client beendet sich!
- Server beendet sich
- Es kommt zu folgender Ausgabe:

```
$ server&
$ client
connected
sent
read: End of file
'server&' has ended
```

- Warum? `client` erwartet ein Zeichen zu viel → `\n`!

Synchroner Fix-Echo-Client – 6

```
        // sync_echo_client3.cpp
        const char request[]{"ping-pong\n"};
        size_t request_length = strlen(request);

        asio::write(sock, asio::buffer(request,
            request_length));
        cout << "sent" << endl;

        char reply[20];
        size_t reply_length = asio::read(sock,//↓
          asio::buffer(reply, request_length - 1));
        cout << "Reply is: ";
        cout.write(reply, reply_length);
        cout << "\n";  // ← it's necessary!
    } catch (asio::system_error& e) {
        std::cerr << e.what() << std::endl; } }
```

Synchroner Fix-Echo-Client – 7

Ausgabe:

```
connected
sent
Reply is: ping-pong
'server&' has ended
```

Synchroner Fix-Echo-Client – 7

Ausgabe:

```
connected
sent
Reply is: ping-pong
'server&' has ended
```

Buffer dzt. aus einem `char`-Array, aber auch z.B.

- `std::string`
- `std::array`

Synchroner Fix-String-Echo-Client

```
        // sync_echo_client_string.cpp
        // don't do it this way...
        const char request[]{”ping-pong\n”};
        size_t request_length = strlen(request);
        asio::write(sock, asio::buffer(request,
            request_length));
        cout << ”sent” << endl;
        string reply(20, ' ');  // ↓
        //string reply{”                    ”};
        size_t reply_length = asio::read(sock,
          asio::buffer(reply, request_length - 1));
        cout << ”Reply is: ”;
        cout.write(reply.data(), reply_length);
        cout << ”\n”;
    } catch (asio::system_error& e) {
        std::cerr << e.what() << std::endl; } }
```

Synchroner Fix-Array-Echo-Client

```
        // sync_echo_client_array.cpp
        const char request[]{"ping-pong\n"};
        size_t request_length = strlen(request);
        asio::write(sock, asio::buffer(request,
            request_length));
        cout << "sent" << endl;
        array<char, 20> reply;
        size_t reply_length = asio::read(sock,//↓
          asio::buffer(reply, request_length - 1));
        cout << "Reply is: ";
        //cout << string(reply.begin(),
        //    reply.begin() + reply_length) << endl
        cout << string(begin(reply),
              begin(reply) + reply_length) << endl;
        cout << "\n";
    } catch (asio::system_error& e) {
        std::cerr << e.what() << std::endl; } }
```

Behandlung der Eingabe von Daten

- **Erkenntnis 1**: entweder lesen bis
  - erwartete Anzahl von Zeichen erreicht

Behandlung der Eingabe von Daten

- **Erkenntnis 1**: entweder lesen bis
  - erwartete Anzahl von Zeichen erreicht oder
  - EOF der Eingabe

Behandlung der Eingabe von Daten

- **Erkenntnis 1**: entweder lesen bis
  - erwartete Anzahl von Zeichen erreicht oder
  - EOF der Eingabe oder
  - bis ein Endezeichen in Daten (z.B. \n)
- **Erkenntnis 2**: man weiß nicht immer wie lange die zu empfangenen Daten sind!

Synchroner EOF-Echo-Client

```
        // sync_eof_echo_client.cpp
        const char request[]{"ping-pong\n"};
        asio::write(sock, asio::buffer(request,
            strlen(request)));
        cout << "sent" << endl;
        char reply[20];  error_code ec;
        size_t reply_length = asio::read(sock,
          asio::buffer(reply), ec);  // ←
        if (ec.value() != asio::error::eof) {
        cout<<ec.message()<<':'<< ec.value()<<endl;
            return 1; }
        cout << "Reply is: ";
        cout.write(reply, reply_length);
        cout << "\n";
    } catch (asio::system_error& e) {
        std::cerr << e.what() << std::endl; } }
```

Synchroner EOL-Echo-Client

```
    // sync_sentinel_echo_client.cpp
    const char request[]{"ping-pong\n"};
    asio::write(sock, asio::buffer(request,
        strlen(request)));
    cout << "sent" << endl;

    asio::streambuf buf;  // ← var. length!
    string reply;
    // read until server sends '\n'
    asio::read_until(sock, buf, '\n');  // ←
    istream is{&buf};
    getline(is, reply);

    cout << "Reply is: " << reply << endl;
} catch (asio::system_error& e) {
    std::cerr << e.what() << std::endl; } }
```

Stream Buffers

- Template `std::streambuf`
  - Input und Output zu Zeichensequenz
  - wird verwendet in `istream`, `ostream`
  - spezialisierte Templates
    - `filebuf` (`fstream`, `ifstream`, `ofstream`)
    - `stringbuf` (`stringstream`, `[io]stringstream`)
    - für `cin`, `cout`
- Template `asio::basic_streambuf`
  - abgeleitet von `std::streambuf`
- Template `asio::streambuf`
  - ist Instanzierung `asio::basic_streambuf`

Stream Buffers – 2

- Template `asio::basic_socket_streambuf`
  - Input und Output von Bytes über Socket
  - abgeleitet von `asio::basic_streambuf`
- Überladungen mit `asio::streambuf` anstatt
  - `MutableBufferSequence` bei `read`
  - `DynamicBufferSequence` bei `read_until`
  - `ConstBufferSequence` bei `write`

Stream EOL-Echo-Server

```cpp
#include <iostream>  // stream_eol_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::iostream strm{acceptor.accept()};
    string data;
    strm >> data;
    strm << data << '\n';  // ←
    strm.close();  }
```