Programmentwicklung II

Bachelor of Science

Prof. Dr. Rethmann / Prof. Dr. Brandt

Fachbereich Elektrotechnik und Informatik
Hochschule Niederrhein

Sommersemester 2021

Zunächst: Nachrichtenbasierte Kommunikation mittels Sockets. Wir nutzen hier unseren Socket-Wrapper aus der letzten Vorlesung.

Wir wollen einen Dienst auf einem Server bereit stellen, wobei der Dienst verschiedene Methoden anbietet:

- `echo` schickt genau die Zeichenkette zurück, die gesendet wurde.
- `date` liefert das Systemdatum an den Aufrufer zurück.
- `time` gibt die aktuelle Systemzeit an den Aufrufer zurück.
- `random` schickt eine zufällige Zahl an den Aufrufer zurück.
- `gcd` gibt den größten gemeinsamen Teiler zurück.
- `prime` testet, ob die gesendete Zahl eine Primzahl ist.

Damit Anfragen mehrerer Clients gleichzeitig bearbeitet werden können, wird die Bearbeitung einer Anfrage in einen Thread ausgelagert. → Thread-per-Request

Wir wollen Polymorphie nutzen, um die einzelnen Dienste bereitzustellen.

service.hpp

```
#ifndef _SERVICE_HPP
#define _SERVICE_HPP

#include "socket.hpp"
#include <string>

class Service {
public:
    virtual void handleReq(Socket s, std::string req) = 0;
};

class EchoService : public Service {
    void handleReq(Socket s, std::string req);
};

class DateService : public Service {
    void handleReq(Socket s, std::string req);
};
```

```
class TimeService : public Service {
    void handleReq(Socket s, std::string req);
};

class RandomService : public Service {
    void handleReq(Socket sock, std::string);
};

class GcdService : public Service {
    void handleReq(Socket sock, std::string);
};

class PrimeService : public Service {
    void handleReq(Socket s, std::string req);
};

#endif
```

service.cpp

```
#include "service.hpp"
.....

void EchoService::handleReq(Socket s, string req) {
    s.send(req);
    s.close();
}

void RandomService::handleReq(Socket s, string req) {
    int val = stoi(req);

    string answ = to_string(rand() % val);
    s.send(answ);
    s.close();
}
```

```
void DateService::handleReq(Socket s, string req) {
    time_t now = time(NULL);
    tm *today = localtime(&now);

    ostringstream os;
    os << setfill('0') << setw(2) << today->tm_mday << "."
       << setw(2) << today->tm_mon + 1 << "."
       << today->tm_year + 1900;

    s.send(os.str());
    s.close();
}
```

```
void PrimeService::handleReq(Socket s, string req) {
    bool prime = true;
    int val = stoi(req);

    for (int i = 2; prime && i <= sqrt(val); i++) {
        if (val % i == 0)
            prime = false;
    }

    if (prime)
        sock.send("true");
    else sock.send("false");
    sock.close();
}

.....  // weitere Implementierungen der Dienste
```

server.cpp

```
#include <thread>
.....
map<string, Service *> _services;
void handleRequest(Socket sock);

int main(int argc, char **argv) {
    _services.insert({"ECHO", new EchoService()});
    _services.insert({"TIME", new TimeService()});
    _services.insert({"DATE", new DateService()});
    _services.insert({"RANDOM", new RandomService()});
    _services.insert({"GCD", new GcdService()});
    _services.insert({"PRIME", new PrimeService()});

   ServerSocket server(6200, 10);
   while (true) {
      thread t(handleRequest, server.accept());
      t.detach(); // detach from calling thread
   }
}
```

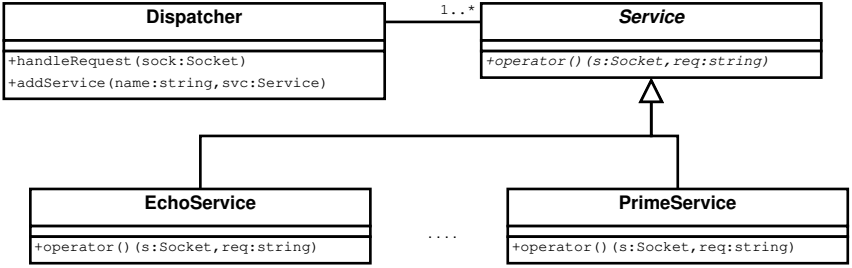
```
void handleRequest(Socket sock) {
    string req = sock.recv();
    Tokenizer tok(req, ":");

    if (tok.countTokens() <= 1) {
        sock.send(string("unknown protocol!"));
        sock.close();
        return;
    }
    string service = tok.nextToken();
    string args = tok.nextToken();
    try {
        _services.at(service)->handleReq(sock, args);
    } catch (const out_of_range &e) {
        cout << e.what() << endl;
        sock.send(string("unknown service!"));
        sock.close();
    }
}
```

Die Methode `detach` erlaubt beiden Threads eine voneinander unabhängige Ausführung. Belegte Ressourcen werden nach Beendigung des Threads automatisch freigegeben. Die Threads sind nicht mehr joinable.

Weitere Dienste können einfach hinzugefügt werden: Definiere eine weitere Unterklasse und trage diese zusammen mit einem Schlüsselwort in der Map `_services` ein.

Schauen wir uns nun an, wie man den Server mittels Funktoren implementieren kann.



service.hpp

```
#ifndef _SERVICE_HPP
#define _SERVICE_HPP

#include "socket.hpp"
#include <string>
#include <map>

class Service {
public:
    virtual void operator()(Socket sock, std::string) = 0;
};

class EchoService : public Service {
    void operator()(Socket sock, std::string);
};

.....  // weitere spezielle Service-Klassen
#endif
```

service.cpp

```
#include "service.hpp"
#include "tokenizer.hpp"
#include <iostream>
.....
using namespace std;

void EchoService::operator()(Socket sock, string req) {
    sock.send(req);
    sock.close();
}

.....  // weitere Implementierungen wie oben
```

dispatcher.hpp

```
#ifndef _DISPATCHER_HPP
#define _DISPATCHER_HPP

#include <string>
#include <map>
#include "socket.hpp"
#include "service.hpp"

class Dispatcher {
private:
    std::map<std::string, Service *> _services;

public:
    void addService(std::string name, Service *srv);
    void handleRequest(Socket sock);
};

#endif
```

dispatcher.cpp

```
#include "dispatcher.hpp"
#include "tokenizer.hpp"
#include <iostream>
#include <sstream>
#include <iomanip>
#include <stdexcept>
#include <unistd.h>
#include <cmath>
#include <thread>

void Dispatcher::addService(std::string name, Service *svc) {
    _services.insert({name, svc});
}
```

```
void Dispatcher::handleRequest(Socket sock) {
    std::string req = sock.recv();
    Tokenizer tok(req, ":");
    ..... // Fehlerbehandlung

    string service = tok.nextToken();
    string args = tok.nextToken();
    try {
        Service *svc = _services.at(service);
        std::thread t(std::ref(*svc), sock, args);
                  // ^^^^^^^^^^^^^^^ !!!!!!!!!!!!
        t.detach();
    } catch (const std::out_of_range& e) {
        cout << e.what() << endl;
        sock.send(string("unknown service!"));
        sock.close();
    }
}
```

server.cpp

```
#include "socket.hpp"
#include "dispatcher.hpp"
using namespace std;

int main(int argc, char **argv) {
    Dispatcher d;
    d.addService("ECHO", new EchoService());
    d.addService("TIME", new TimeService());
    d.addService("DATE", new DateService());
    d.addService("RANDOM", new RandomService());
    d.addService("GCD", new GcdService());
    d.addService("PRIME", new PrimeService());

    ServerSocket server(8080, 10);

    while (true) {
        d.handleRequest(server.accept());
    }
}
```

Thread-per-Request ist nicht immer sinnvoll:

- Ineffizient und nicht skalierbar aufgrund von Kontextwechseln, Synchronisation und Austausch von Daten zwischen CPUs.
- Komplizierte Kontrolle der Nebenläufigkeit um Zugriffe auf gemeinsam genutzte Daten zu organisieren.
- Aufteilung der Threads auf Ressourcen wie CPUs oder CPU-Kerne erscheint sinnvoller als eine Zuordnung auf Requests.

$\Rightarrow$ nutze Thread-Pool (wird hier nicht behandelt)

Wir wollen uns jetzt eine Architektur ansehen, bei denen unterschiedliche Dienste auf unterschiedlichen Ports bereit gestellt werden.

- Wie können gleichzeitig mehrere Ports auf Anfragen überwacht werden, ohne verschiedene Threads zu starten?

Die Funktion `accept` blockiert den aufrufenden Prozess, daher

- kann nur auf einem Socket auf Anfragen gewartet werden und
- mehrere Clients können nur mittels weiterer Prozesse oder Threads versorgt werden.

Alternative:

```
int select(int nfds, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout)
```

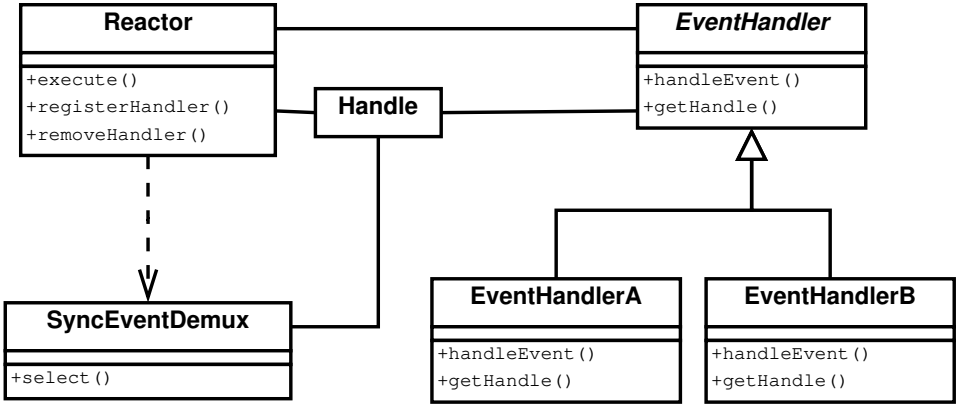
- Wartet auf Status-Änderungen bei beliebig vielen File-Deskriptoren bzw. Anfragen bei Sockets. (Bsp: `inetd`)
- Funktion blockiert, bis über einen Deskriptor `readfds`, `writefds` oder `exceptfds` Daten gelesen oder geschrieben werden können bzw. eine Datei ihren Status ändert.

Fortsetzung: `int select(int nfds, fd_set *readfds, ...)`

- Liefert die Anzahl der Deskriptoren, die das Ende von `select` ausgelöst haben.
- Handelt es sich um Client-Anfragen, können diese mit `accept` angenommen werden, ohne dass `accept` blockiert.
- Der Parameter `timeout` begrenzt die Wartezeit. Wird der Parameter auf 0 gesetzt, ist die Wartezeit unbegrenzt.
- Zur Manipulation der Deskriptor-Mengen stehen Makros zur Verfügung:
  - `FD_ZERO(fd_set *set)` leeren,
  - `FD_SET(int fd, fd_set *set)` Deskriptor eintragen,
  - `FD_CLR(int fd, fd_set *set)` Deskriptor austragen,
  - `FD_ISSET(int fd, fd_set *set)` testet, ob der angegebene Deskriptor das Ende von `select` ausgelöst hat.

Betrachten wir noch einmal unseren Dienst, der die Methoden `echo`, `date`, `time`, `random`, `gcd` und `prime` bereit stellt.

Wir wollen den Parallelen Server von Thread-per-Request auf Single-Threaded mittels `select` so ändern, dass jeder Dienst auf einem eigenen Port läuft.



Entwurfsmuster Reactor: Teilnehmer

- Der SynchronousEventDemultiplexer kapselt eine Funktion der Socket-API. Die Funktion `select` wartet auf Events auf einer Menge von Handles (hier Ports) und wird in einer Wrapper- Klasse gekapselt.
- Handles werden vom Betriebssystem bereitgestellt und identifizieren Ressourcen wie Netzwerkverbindungen oder Dateien. Auch die Handles werden durch Wrapper-Klassen gekapselt. Tritt ein Ereignis bei einer Ressource auf, so wird das Event gepuffert und das Handle als bereit markiert.
- Event-Handler spezifiziert eine Schnittstelle aus einer oder mehreren Methoden.
- Ein konkreter Event-Handler ist eine Spezialisierung des Event-Handlers und implementiert einen speziellen Dienst, den die Anwendung bereitstellt. Der konkrete Event-Handler ist mit einem Handle (hier Port) assoziiert, das den Dienst identifiziert.
- Die Applikation kann beim Reactor beliebige Event-Handler und deren zugeordneten Handles registrieren bzw. entfernen.

  Der Reactor wartet auf Events (hier TCP-Connections) und reicht das Event an den entsprechenden Handler weiter, der dieses Event bearbeitet.

dispatcher.hpp

```
#include "socket.hpp"
#include <string>
#include <map>

class Service {
public:
    virtual void handleRequest(int fd) = 0;
    virtual int getPort(void) = 0;
};

class EchoService : public Service {
    void handleRequest(int fd);
    int getPort(void);
};

.....  // weitere Service-Unterklassen
```

```
class Dispatcher {
private:
    fd_set _rfds;
    int _minFD, _maxFD;
    std::map<int, Service *> _services;

public:
    Dispatcher(void);
    void execute(void);
    void addHandler(Service *service);
};
```

dispatcher.cpp

```
#include "dispatcher.hpp"
#include <iostream>
#include <string>
.....
using namespace std;

void EchoService::handleRequest(int fd) {
    Socket sock(::accept(fd, NULL, 0));
    string req = sock.recv();

    sock.send(req);
    sock.close();
}

int EchoService::getPort(void) {
    // standard services should come from /etc/services
    return 6200;
}
```

```
void RandomService::handleRequest(int fd) {
    Socket sock(::accept(fd, NULL, 0));
    string req = sock.recv();

    int val = stoi(req);

    string msg = to_string(rand() % val);
    sock.send(msg);
    sock.close();
}

int RandomService::getPort(void) {
    // standard services should come from /etc/services
    return 6201;
}

..... // weitere Service-Unterklassen
```

```
Dispatcher::Dispatcher(void) {
    _minFD = INT_MAX;
    _maxFD = INT_MIN;
    FD_ZERO(&_rfds);
}

void Dispatcher::addHandler(Service *service) {
    ServerSocket sock(service->getPort(), 10);
    int sockfd = sock.getSocketFD();

    // _services.insert(pair<int, Service *>(sockfd, service));
    _services.insert({sockfd, service});  // seit C++11
    FD_SET(sockfd, &_rfds);
    if (sockfd < _minFD)
        _minFD = sockfd;
    if (sockfd > _maxFD)
        _maxFD = sockfd;
}
```

```
void Dispatcher::execute(void) {
    cout << "waiting for connections ...\n";
    while (1) {
        fd_set rfds = _rfds;

        int err = select(_maxFD + 1, &rfds, NULL, NULL, NULL);
        if (err == EINTR) continue;

        map<int, Service *>::iterator iter;
        for (int i = _minFD; i <= _maxFD; i++) {
            if (FD_ISSET(i, &rfds)) {
                iter = _services.find(i);
                iter->second->handleRequest(i);
            }
        }
    }
}
```

server.cpp

```
#include "dispatcher.hpp"

int main(int argc, char **argv) {
    Dispatcher reactor;

    reactor.addHandler(new EchoService());
    reactor.addHandler(new DateService());
    reactor.addHandler(new TimeService());
    reactor.addHandler(new RandomService());
    reactor.addHandler(new GcdService());
    reactor.addHandler(new PrimeService());

    reactor.execute();
}
```

Auf die Klassen `SynchEventDemux` und `Handle` wurde verzichtet. Wenn wir wirklich plattformunabhängig entwickeln wollen, wären entsprechende Wrapper-Klassen zu definieren.

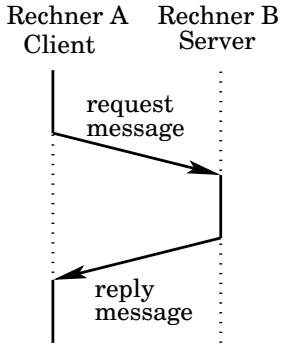
Nachteile einer auf Sockets basierenden Kommunikation:

- Bei UDP muss die Unzuverlässigkeit des Protokolls in der Anwendungsschicht ausgeglichen werden. Bei TCP sind Verbindungsaufbau und -abbau explizit zu programmieren.
- Beim Server müssen Aufgaben in Threads ausgelagert werden und Parallelität und Mechanismen zur Synchronisation müssen explizit programmiert werden.
- Die zu übertragenen Daten müssen ggf. formatiert werden.

Paradigmenbruch zur prozeduralen und zur objektorientierten Programmierung:

- Multi-Threading-Systeme kommunizieren über gemeinsamen Speicher, nicht mittels Signalen oder Nachrichten.
- Innerhalb eines Threads werden Aufgaben in Teilaufgaben zerlegt, die in Funktionen bereitgestellt oder von Klassen übernommen werden. Nachrichten zu verschicken ist unüblich.

Aufruf einer entfernten Prozedur:



- Der aufrufende Prozess wird blockiert.
- Abarbeitung der Prozedur findet auf anderer Maschine statt.
- Parameter und Ergebnisse werden zwischen den Maschinen transportiert.
- Remote Procedure Call zeigt (fast) das vertraute Verhalten von lokalen Prozeduraufrufen.

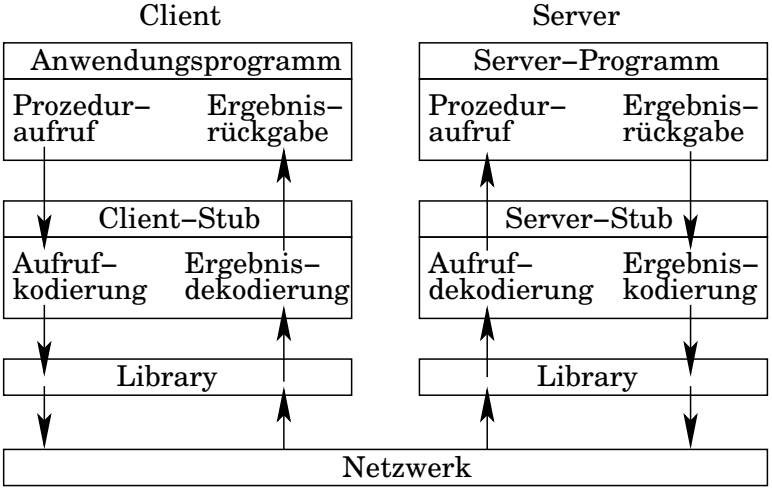
Die im Hintergrund stattfindende Nachrichtenübertragung wird vor dem Anwender versteckt. Oft: Der RPC-Compiler erzeugt automatisch alle erforderlichen Funktionen zur Formatierung der Daten.

Stubs verstecken die Nachrichtenübertragung vor dem Anwender.

- Der *Client-Stub*
  - bestimmt die Server-Adresse (durch Broadcast oder Auskunftsdienst),
  - stellt die Nachricht zusammen,
  - gleicht unterschiedliche Codierungen und Datenformate an,
  - verschickt die Nachricht,
  - überwacht die korrekte Übertragung, ...
- Der *Server-Stub*
  - führt eine Endlosschleife aus und wartet auf Nachrichten,
  - entpackt eine Nachricht und ruft die gewünschte Prozedur auf
  - und verpackt das Ergebnis und schickt es an den Client.

Client und Server laufen in unterschiedlichen Adressräumen:

- *call by value* ist kein Problem: Für die entfernte Prozedur ist ein Wert-Parameter eine initialisierte lokale Variable, die beliebig modifizierbar ist.
- *call by reference* und ein *Zugriff auf globale Variablen* ist nicht möglich!



Stub: Stummel, Stumpf

Unterschiedliche Datenrepräsentation macht Konvertierung notwendig!

- Strings: EBCDIC-, ASCII-, latin1- oder utf16-Code
- Integer: Einer- oder Zweierkomplement, little-/big-endian.
- Float: Größe von Mantisse und Exponent.

direkte Konvertierung:

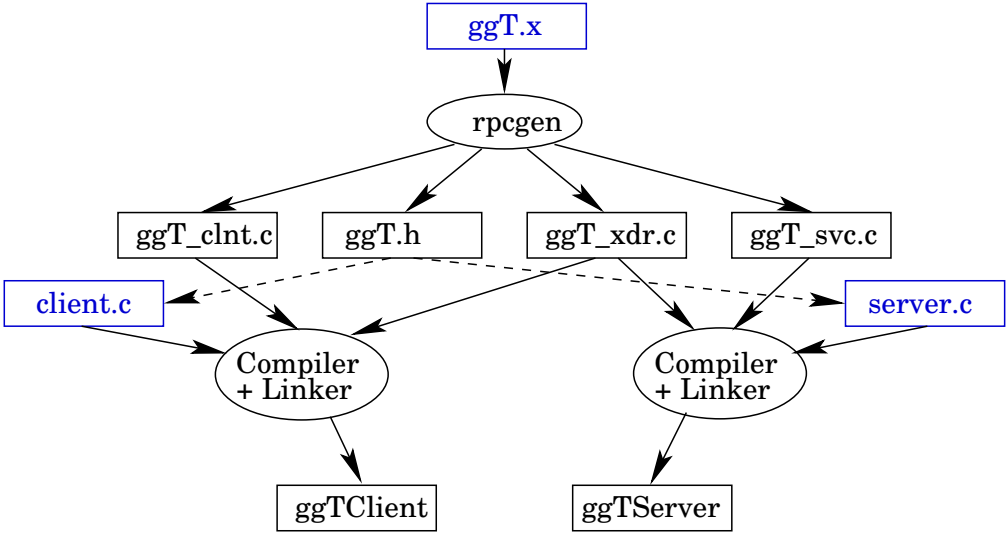
- Der Client-Stub hängt vor die Nachricht eine Indikation des verwendeten Formats.
- Der Server-Stub wandelt ggf. vom fremden Datenformat ins eigene Format um.

+ Es ist maximal eine Umwandlung nötig.
– Bei n Formaten sind $n \cdot (n-1)$ Konvertierungsfunktionen notwendig und ist schlecht um weitere Formate erweiterbar.

maschinenunabhängiges Netzwerkdatenformat: nur $2n$ Konvertierungsfunktionen nötig

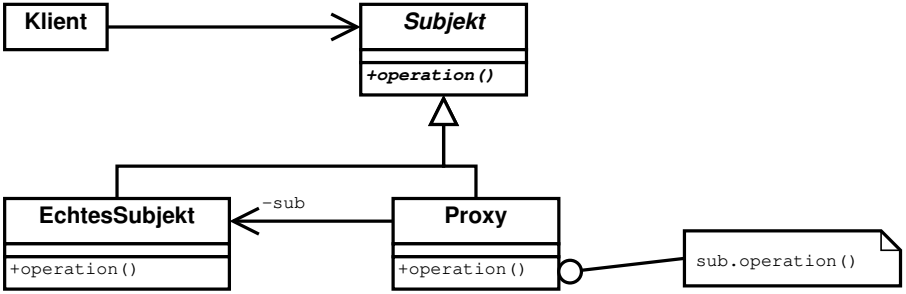
- Der Client-Stub wandelt das eigene Datenformat in die Netzwerkdatendarstellung.
- Der Server-Stub wandelt die Netzwerkdatendarstellung ins eigene Format.

+ einfach um weitere Formate erweiterbar
– unnötige Konvertierungen, falls Client und Server gleiches Format nutzen


ggT.x
rpcgen
ggT_clnt.c
ggT.h
ggT_xdr.c
ggT_svc.c
client.c
server.c
Compiler
+ Linker
Compiler
+ Linker
ggTClient
ggTServer

Der Client-Stub implementiert das Entwurfsmuster Proxy. Der Proxy kontrolliert den Zugriff auf ein Objekt mit Hilfe eines vorgelagerten Stellvertreterobjekts.

- Befindet sich das echte Objekt in einem anderen Adressraum, dann spricht man von einem *Remote-Proxy*.
- Überprüft der Proxy die Zugriffsrechte beim Zugriff auf das echte Objekt, so spricht man von einem *Schutz-Proxy*.



Sun RPC (heute als ONC RPC bezeichnet: Open Network Computing)

- Ist sehr alt und basiert auf der Sprache C.
- Es gibt Third-Party Implementierungen, die auch Sprachen wie C++, Java und .NET unterstützen.

Basierend auf RPC gibt es:

- Remote Method Invocation (RMI) stellt einen RPC-Mechanismus für Java bereit.
- Remote Function Call (RFC) ist die Implementierung von RPCs in SAP-Systemen.
- XML-RPC ist ein RPC-Ableger, der auf XML-Dokumenten und http basiert.
- Webservices können RPC als Architekturmuster nutzen.

Es gibt auch moderne RPC-Implementierungen wie Apache Thrift[1].

[1] `https://thrift.apache.org/`

Laut wikipedia[2]:

- Apache Thrift ist ein Kommunikationsprotokoll für interoperable und skalierbare Services. Es kombiniert einen Software-Protokollstapel mit einer Generatorkomponente zur Erstellung von Services, die zwischen verschiedenen Programmiersprachen wie C, C++, C#, Delphi, Erlang, Go, Haskell, Java, Perl, PHP, Python, Ruby, Smalltalk und anderen Sprachen operieren können.
- Das Projekt fand seinen Ursprung bei Facebook, um die Entwicklung von hochskalierbaren Backend-Services zu unterstützen. Jetzt ist es ein Software-Entwicklungsprojekt der Apache Software Foundation und wird unter der Apache-2.0-Lizenz veröffentlicht.

[2] `https://de.wikipedia.org/wiki/Apache_Thrift`

Thrift unterstützt Textprotokolle und binäre Protokolle. Die binären Protokolle verfügen über eine höhere Bewertung als die sekundären Textprotokolle. Textprotokolle werden oft bei der Fehlersuche angewendet. Einige von Thrift unterstützte Protokolle:

- `TBinaryProtocol` Eine Binärformat-Codierung, die numerische Werte binär darstellt, anstelle von Textkonvertierungen.
- `TCompactProtocol` Eine effiziente Compact-Codierung, die die betreffenden Daten komprimiert.
- `TDenseProtocol` Ähnlich dem `TCompactProtocol`, jedoch entfernt dieses die Metainformationen der übertragenen Daten und fügt diese wieder bei der Übertragung an den Empfänger ein.
- `TJSONProtocol` Dieses Protokoll verwendet JSON für die Codierung der Daten.
- `TSimpleJSONProtocol` Ein Schreibprotokoll mit JSON. Geeignet für das Parsen von Skriptsprachen.
- `TDebugProtocol` Dieses Protokoll formatiert in lesbare Textformate für das Debugging.

Wir implementieren ein Telefonbuch und definieren zunächst die Schnittstelle mittels der *Interface Definition Language* IDL.

PhoneBook.thrift

```
exception NotFoundException {
    1:string msg
}

service PhoneBook {
    bool add(1:string name, 2:string no),

    string get(1:string name)
        throws (1:NotFoundException ex)
}
```

Mittels Aufruf des thrift-Compilers `thrift --gen cpp PhoneBook.thrift` wird unter anderem die Datei `gen-cpp/PhoneBook.h` erzeugt:

```
class PhoneBookIf {
public:
    virtual ~PhoneBookIf() {}

    virtual bool add(const std::string& name,
                     const std::string& no) = 0;

    virtual void get(std::string& _return,
                     const std::string& name) = 0;
};
```

Dieses Interface nutzen wir nun in unserem Server, um einen `PhoneBookHandler` zu erstellen.

server.cpp

```
#include "gen-cpp/PhoneBook.h"
#include <thrift/protocol/TBinaryProtocol.h>
#include <thrift/server/TSimpleServer.h>
#include <thrift/transport/TServerSocket.h>
#include <thrift/transport/TBufferTransports.h>

#include <map>

using apache::thrift::TProcessor;
using apache::thrift::transport::TServerTransport;
using apache::thrift::transport::TServerSocket;
using apache::thrift::transport::TTransportFactory;
using apache::thrift::transport::TBufferedTransportFactory;
using apache::thrift::protocol::TBinaryProtocolFactory;
using apache::thrift::protocol::TProtocolFactory;
using apache::thrift::server::TSimpleServer;

using namespace std;
```

```
class PhoneBookHandler : virtual public PhoneBookIf {
public:
    map<string, string> _items;

    PhoneBookHandler() { ..... }

    bool add(const string& name, const string& no) {
        _items[name] = no;
        return true;
    }

    void get(string& _return, const string& name) {
        map<string, string>::iterator r = _items.find(name);
        if (r == _items.end()) {
            throw NotFoundException({name});
        }
        _return = (*r).second;
    }
};
```

```
int main(int argc, char **argv) {
    int port = 9090;

    shared_ptr<PhoneBookHandler>
        handler(new PhoneBookHandler());
    shared_ptr<TProcessor>
        processor(new PhoneBookProcessor(handler));
    shared_ptr<TServerTransport>
        serverTransport(new TServerSocket(port));
    shared_ptr<TTransportFactory>
        transportFactory(new TBufferedTransportFactory());
    shared_ptr<TProtocolFactory>
        protocolFactory(new TBinaryProtocolFactory());

    TSimpleServer server(processor, serverTransport,
                         transportFactory, protocolFactory);
    server.serve();
}
```

Zunächst kompilieren wir die vom thrift-Compiler erzeugten Hilfsklassen, dazu müssen wir zunächst in das Verzeichnis gen-cpp wechseln:

```
[entering directory gen-cpp ...]
g++ -Wall -Wextra -c PhoneBook_constants.cpp
g++ -Wall -Wextra -c PhoneBook_types.cpp
g++ -Wall -Wextra -c PhoneBook.cpp
[leaving directory gen-cpp ...]
```

Anschließend können wir den Server kompilieren:

```
g++ -Wall -Wextra -c server.cpp
g++ gen-cpp/PhoneBook_constants.o gen-cpp/PhoneBook.o \
        gen-cpp/PhoneBook_types.o server.o -o server -lthrift
```

Jetzt benötigen wir noch einen Client, der den Dienst nutzt.

client.cpp

```
#include <iostream>
#include <thrift/protocol/TBinaryProtocol.h>
#include <thrift/transport/TSocket.h>
#include <thrift/transport/TTransportUtils.h>

#include "gen-cpp/PhoneBook.h"

using apache::thrift::transport::TTransport;
using apache::thrift::transport::TSocket;
using apache::thrift::transport::TBufferedTransport;
using apache::thrift::protocol::TProtocol;
using apache::thrift::protocol::TBinaryProtocol;
using apache::thrift::TException;

using namespace std;
```

```
int main(void) {
    shared_ptr<TTransport>
        socket(new TSocket("localhost", 9090));
    shared_ptr<TTransport>
        transport(new TBufferedTransport(socket));
    shared_ptr<TProtocol>
        protocol(new TBinaryProtocol(transport));

    // die Klasse PhoneBookClient wird vom thrift-compiler
    // erzeugt
    PhoneBookClient client(protocol);
```

```
    try {
        transport->open();

        if (strcmp(argv[1], "-a") == 0) {         // APPEND
            bool result = client.add(argv[2], argv[3]);
            cout << "result: " << result << endl;
        } else {                                  // FIND
            string no;  // out-parameter
            client.get(no, argv[2]);
            cout << "no(" << argv[2] << "): " << no << endl;
        }
    } catch (NotFoundException & ex) {
        cout << "no number found for " << ex.msg << endl;
    } catch (TException & tx) {
        cout << "ERROR: " << tx.what() << endl;
    }
    transport->close();
}
```

Jetzt kompilieren wir den Client:

```
g++ -Wall -Wextra -c client.cpp
g++ gen-cpp/PhoneBook_constants.o gen-cpp/PhoneBook.o \
        gen-cpp/PhoneBook_types.o client.o -o client -lthrift
```

Anschließend starten wir den Server und können Werte hinzufügen oder abfragen:

```
./client -a walter 2378
result: 1
./client -a nora 9843
result: 1
./client -f klara
no number found for klara
./client -f nora
no(nora): 9843
```

Um einen Multi-Threaded-Server zu erstellen, ist anstelle der Klasse `TSimpleServer` die Klasse `TThreadedServer` zu nutzen. Dadurch können mehrere Clients gleichzeitig RPC-Calls ausführen. Achten Sie auf mögliche Race-Conditions. Die Message-Queue wird von allen Threads gemeinsam genutzt und muss daher Thread-Safe sein.

```
int main(int argc, char **argv) {
    int port = 9090;

    TThreadedServer server(
        make_shared<PhoneBookProcessor>(
            make_shared<PhoneBookHandler>()),
        make_shared<TServerSocket>(port),
        make_shared<TBufferedTransportFactory>(),
        make_shared<TBinaryProtocolFactory>());

    server.serve();
}
```