Middleware

Dozent: Prof. Dr. Michael Eichberg
Kontakt: michael.eichberg@dhbw.de, Raum 149B
Version: 1.0

Folien: [HTML] https://delors.github.io/ds-middleware/folien.de.rst.html
[PDF] https://delors.github.io/ds-middleware/folien.de.rst.html.pdf

Fehler melden: https://github.com/Delors/delors.github.io/issues

1. Einführung in Middleware

Was ist Middleware?

Definition

Middleware ist eine Klasse von Software-Technologien, die dazu dienen,

I. die Komplexität und
II. die Heterogenität verteilter Systeme zu verwalten.

Ein einfacher Server mit Sockets (in C)

```c
/* A simple TCP based server. The port number is passed as an argument */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

void error(char *msg){perror(msg); exit(1);}

int main(int argc, char *argv[]){
  int sockfd, newsockfd, portno, clilen;
  char buffer[256]; int n;
  struct sockaddr_in serv_addr, cli_addr;

  sockfd = socket(AF_INET, SOCK_STREAM, 0); // socket() returns a socket descriptor
  if (sockfd < 0)
  error("ERROR opening socket");

  bzero((char *) &serv_addr, sizeof(serv_addr)); // bzero() sets all values to zero.
  portno = atoi(argv[1]); // atoi() converts str into an integer

  serv_addr.sin_family = AF_INET;
  serv_addr.sin_addr.s_addr = INADDR_ANY;
  serv_addr.sin_port = htons(portno);

  if (bind(sockfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
  error("ERROR on binding");
  listen(sockfd,5); // tells the socket to listen for connections
  clilen = sizeof(cli_addr);
  newsockfd = accept(sockfd, (struct sockaddr *) &cli_addr, &clilen);
  if (newsockfd < 0) error("ERROR on accept");

  bzero(buffer,256);
  n = read(newsockfd,buffer,255);
  if (n < 0) error("ERROR reading from socket");
  printf("Here is the message: %s\n",buffer);
  n = write(newsockfd,"I got your message",18);

  if (n < 0) error("ERROR writing to socket");

  return 0;
}
```

Ein einfacher Client mit Sockets (in C)

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

void error(char *msg){ perror(msg);exit(0);}

int main(int argc, char *argv[]){
  int sockfd, portno, n;
  struct sockaddr_in serv_addr;
  struct hostent *server;
  char buffer[256];

  portno = atoi(argv[2]);

  sockfd = socket(AF_INET, SOCK_STREAM, 0);
  if (sockfd < 0)
    error("ERROR opening socket");

  server = gethostbyname(argv[1]);
  bzero((char *) &serv_addr, sizeof(serv_addr));
  serv_addr.sin_family = AF_INET;
  bcopy((char *)server->h_addr, (char *)&serv_addr.sin_addr.s_addr, server->h_length);
  serv_addr.sin_port = htons(portno);

  if (connect(sockfd,&serv_addr,sizeof(serv_addr)) < 0) error("ERROR connecting");

  printf("Please enter the message: ");
  bzero(buffer,256);
  fgets(buffer,255,stdin);
  n = write(sockfd,buffer,strlen(buffer));
  if (n < 0) error("ERROR writing to socket");
  bzero(buffer,256);
  n = read(sockfd,buffer,255);
  printf("%s\n",buffer);

  return 0;
}
```

Probleme bei der Verwendung von Sockets

Wir müssen uns kümmern um ...

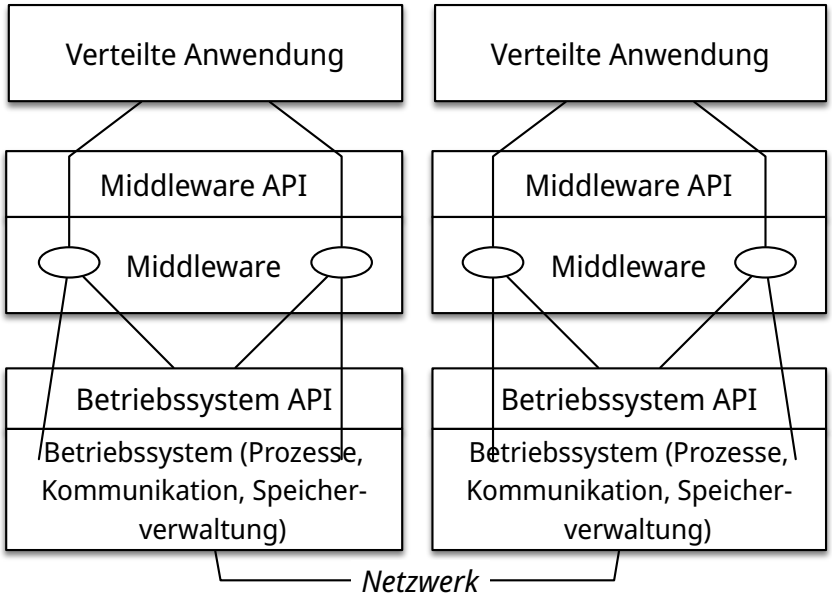
- ... die Einrichtung eines Kanals und alle Fehler, die während dieses Prozesses auftreten können.
- ... die Festlegung eines Protokolls.

  Wer sendet was, wann, in welcher Reihenfolge und welche Antwort wird erwartet?
- ... Nachrichtenformate

  Umwandlung von Daten der Anwendungsebene in Bytes, die über das Netz übertragen werden können.

Middleware als Programmierabstraktion

- Eine Softwareschicht oberhalb des Betriebssystems und unterhalb des Anwendungsprogramms, die eine gemeinsame Programmierabstraktion in einem verteilten System bietet.
- Ein Baustein auf höherer Ebene als die vom Betriebssystem bereitgestellten APIs (z. B. Sockets)



Middleware als Programmierabstraktion

Die von Middleware angebotenen Programmierabstraktionen verbergen einen Teil der Heterogenität und bewältigen einen Teil der Komplexität, mit der Programmierer einer verteilten Anwendung umgehen müssen:

- ✓Middleware maskiert immer die Heterogenität der zugrundeliegenden Netzwerke und Hardware.
- ✓Middleware maskiert meistens die Heterogenität von Betriebssystemen und/oder Programmiersprachen.
- ✓Manche Middleware maskiert sogar die Heterogenität zwischen den Implementierungen des gleichen Middleware-Standards durch verschiedene Hersteller.

Alte Middlewarestandards – wie zum Beispiel CORBA – waren sehr komplex und die Implementierungen verschiedener Hersteller meist nicht vollständig kompatibel.

Transparenzziele von Middleware aus Sicht der Programmierung

Middleware bietet (beim Programmieren) Transparenz in Bezug auf eine oder mehrere der folgenden Dimensionen:

- Standort
- Nebenläufigkeit
- Replikation
- Ausfälle (bedingt)

Middleware ist die Software, die ein verteiltes System (DS) programmierbar macht.

Middleware als Infrastruktur

- Hinter Programmierabstraktionen steht eine komplexe Infrastruktur, die diese Abstraktionen implementiert

  Middleware-Plattformen können sehr komplexe Softwaresysteme sein.
- Da die Programmierabstraktionen immer höhere Ebenen erreichen, muss die zugrunde liegende Infrastruktur, die die Abstraktionen implementiert, entsprechend wachsen.
- Zusätzliche Funktionalität wird fast immer durch zusätzliche Softwareschichten implementiert.
- Die zusätzlichen Softwareschichten erhöhen den Umfang und die Komplexität der für die Nutzung der neuen Abstraktionen erforderlichen Infrastruktur.

Seit Jahrzehnten kann beobachtet werden, dass Middleware immer komplexer wird bzw. wurde bis zu dem Punkt an dem die Komplexität kaum mehr beherrschbar war. Zu diesen Zeitpunkten wurden dann häufig neue Ansätze entwickelt, die die Komplexität reduzierten bis diese wiederum Eingang in komplexere Middleware-Produkten Eingang fand.

Ansätze, wie z. B. REST, haben sich als recht erfolgreich erwiesen stellen aber Entwickler vor neue Herausforderungen.

Middleware und nicht-funktionale Anforderungen

Die Infrastruktur kümmert sich um nicht-funktionale Eigenschaften, die normalerweise von Datenmodellen, Programmiermodellen und Programmiersprachen ignoriert werden:

- Performance
- Verfügbarkeit
- Ressourcenmanagement
- Zuverlässigkeit
- usw.

Middleware als Infrastruktur

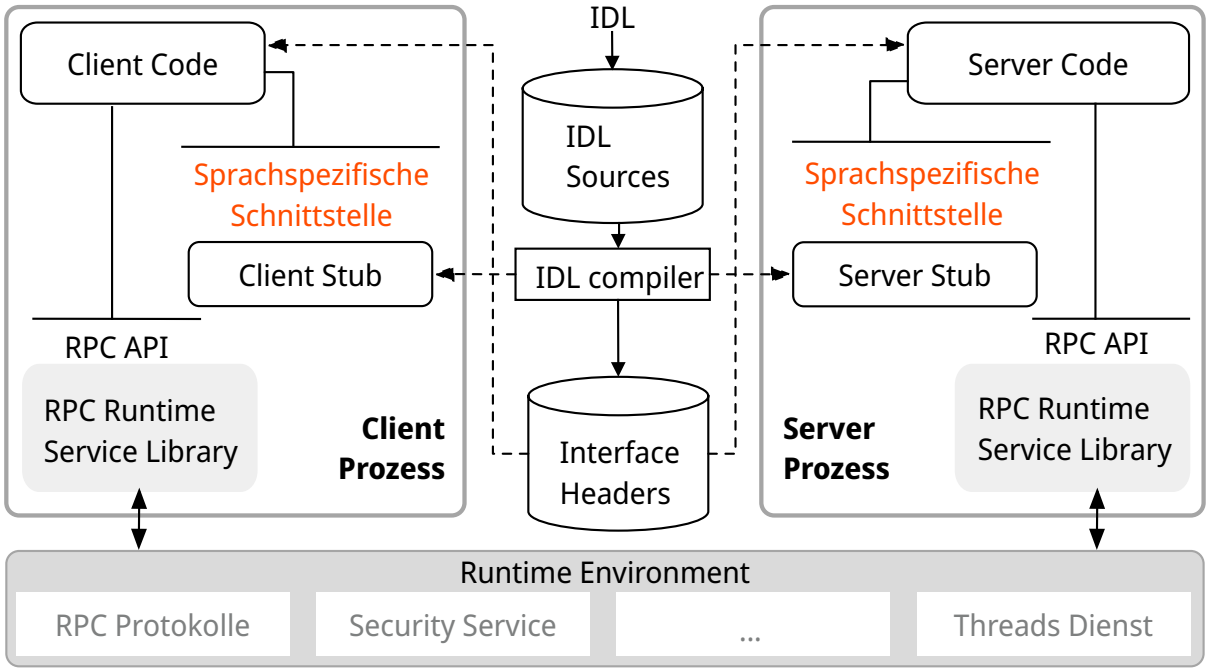
Middleware unterstützt zusätzliche Funktionen die die Entwicklung, Wartung und Überwachung einfacher und kostengünstiger machen (Auszug):

- Protokollierung ( *Logging*)
- Wiederherstellung ( *Recovery*)
- Sprachprimitive für transaktionale Abgrenzung

  (Bzw. fortgeschrittene Transaktionsmodelle (z. B. transaktionale RPC) oder transaktionale Dateisysteme)

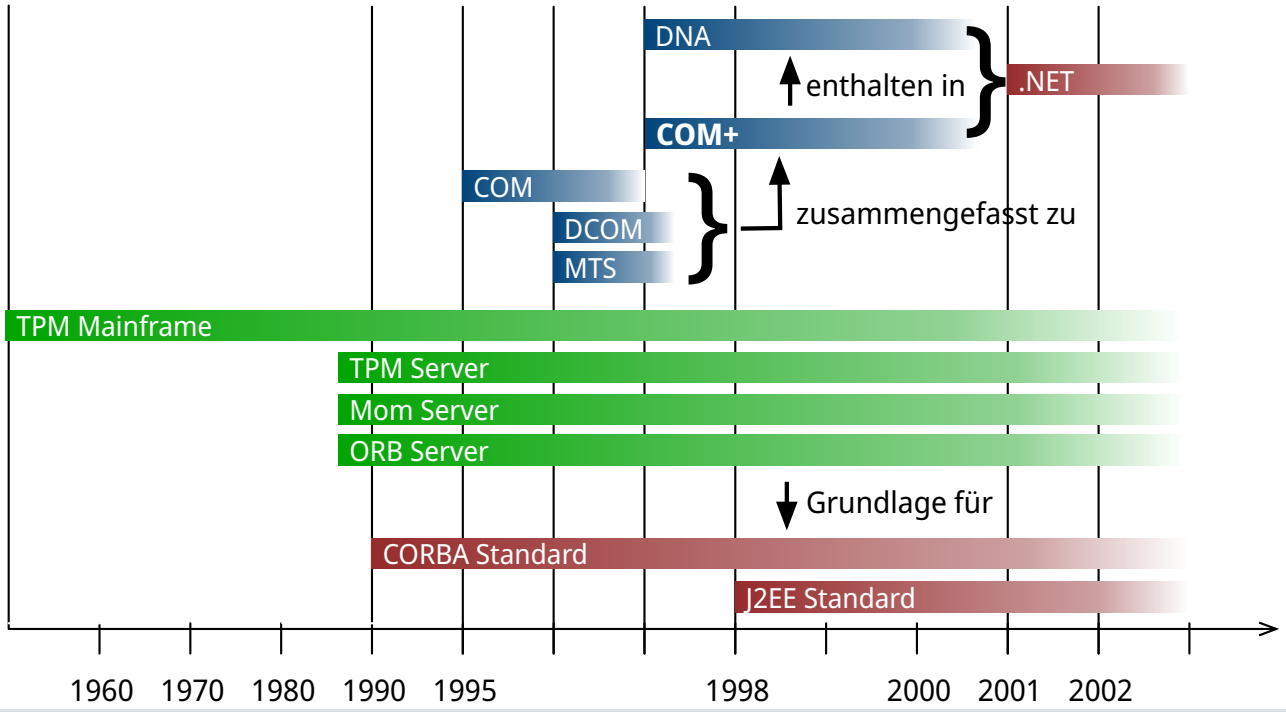
Middleware - konzeptionelle (historisch)

Darstellung nach: Alonso; Web services: Concepts, Architectures and Applications; Springer, 2004



Insbesondere die explizite Erzeugung von Stubs und Skeletons durch einen IDL Compiler erfolgt so in der heutigen Zeit nicht mehr. Die Erzeugung von Stubs und Skeletons - wenn überhaupt erforderlich - erfolgt heute automatisch durch die Middleware.

Historische Entwicklung von Middleware



Entwicklung von Middleware

- Middleware beabsichtigt die Details der Hardware, der Netze und der Verteilung auf niedriger Ebene zu verbergen.
- Anhaltender Trend zu immer leistungsfähigeren Primitiven (*Events*), die zusätzliche Eigenschaften haben oder eine flexiblere Nutzung des Konzepts ermöglichen.
- Die Entwicklung und das Erscheinungsbild für den Programmierer wird von den Trends in den Programmiersprachen diktiert:
  - RPC und C
  - CORBA und C++
  - RMI (Corba) und Java
  - „Klassische“ Webservices und XML
  - RESTful Webservices und JSON

Eine Middleware stellt eine umfassende Plattform für die Entwicklung und den Betrieb komplexer verteilter Systeme zur Verfügung.

2. Middleware-Technologien

Remote Procedure Call (RPC)

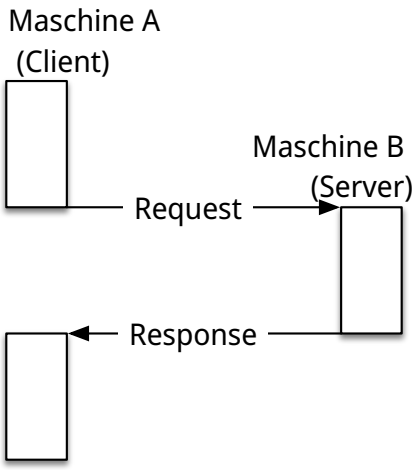
Schwerpunkt: verstecken der Netzkommunikation.

Ein Prozess kann eine Prozedur aufrufen deren Implementierung sich auf einem entfernten Rechner befindet:

- Programmierer von verteilten Systemen müssen sich nicht mehr um alle Details der Netzwerkprogrammierung kümmern (d. h. keine „expliziten“ Sockets mehr).
- Überbrückung der konzeptionellen Lücke zwischen dem Aufruf lokaler Funktionalität über Prozeduren und dem Aufruf entfernter Funktionalität über Sockets.

RPCs konzeptionell (synchrone Kommunikation)

- Ein Server ist ein Programm, das bestimmte Dienste implementiert.
- Cients möchten diese Dienste in Anspruch nehmen:
  - Die Kommunikation erfolgt durch das Senden von Nachrichten (kein gemeinsamer Speicher, keine gemeinsamen Festplatten usw.)
  - Einige minimale Garantien müssen gegeben werden (Behandlung von Fehlern, Aufrufsemantik, usw.)



RPCs - zentrale Fragestellungen und Herausforderungen

Sollen entfernte Aufrufe transparent oder nicht transparent für den Entwickler sein?

> Ein entfernter Aufruf ist etwas völlig anderes als ein lokaler Aufruf; sollte sich der Programmierer dessen bewusst sein?

Wie können Daten zwischen Maschinen ausgetauscht werden, die möglicherweise unterschiedliche Darstellungen für verschiedene Datentypen verwenden?
Komplexe Datentypen müssen linearisiert werden:

Marshalling: der Prozess des Aufbereitens der Daten in eine für die Übermittlung in einer Nachricht geeignete Form.

Unmarshalling: der Prozess der Wiederherstellung der Daten bei ihrer Ankunft am Zielort, um eine originalgetreue Repräsentation zu erhalten.

Wie findet und bindet man den Dienst, den man tatsächlich will, in einer potenziell großen Sammlung von Diensten und Servern?

Das Ziel ist, dass der Kunde nicht unbedingt wissen muss, wo sich der Server befindet oder sogar welcher Server den Dienst anbietet (Standorttransparenz).
Wie geht man mehr oder weniger elegant mit Fehlern um:

- Server ist ausgefallen
- Kommunikation ist gestört
- Server beschäftigt
- doppelte Anfragen ...

Je nach System ist die Reihenfolge der Bytes unterschiedlich:

- Intel-CPUs sind Little-Endian.
- PowerPC ist Big-Endian.
- ARM kann beides und ist meistens Little-Endian.

High-level View auf RPC

Für Programmierer sieht ein „entfernter“ Prozeduraufruf fast identisch aus wie ein „lokaler“ Prozeduraufruf und funktioniert auch so - auf diese Weise wird Transparenz erreicht.

Um Transparenz zu erreichen, führte RPC viele Konzepte von Middleware-Systemen ein:

- *Interface Description Language* (IDL)
- Verzeichnis- und Benennungsdienste
- Dynamische Bindung
- Marshalling und Unmarshalling
- *Opaque References*, um bei verschiedenen Aufrufen auf dieselbe Datenstruktur oder Entität auf dem Server zu verweisen.
  (Der Server ist für die Bereitstellung dieser undurchsichtigen Referenzen verantwortlich.)

RPC - Call Semantics

Nehmen wir an, ein Client stellt eine RPC-Anfrage an einen Dienst eines bestimmten Servers. Nachdem die Zeitüberschreitung abgelaufen ist, beschließt der Client die Anfrage erneut zu senden. Das finale Verhalten hängt von der Semantik des Aufrufs (🏴 *Call Semantics*) ab:

Maybe (vielleicht; keine Garantie)

Die Zielmethode kann ausgeführt worden sein und die Antwortnachricht(en) ging(en) verloren oder die Methode wurde gar nicht erst ausgeführt da die Anfrage verloren ging.

`XMLHTTPRequests` und `fetch()` in Webbrowsern verwenden diese Semantik.

At least once (mindestens einmal)

Die Prozedur wird ausgeführt werden solange der Server nicht endgültig versagt.

Es ist jedoch möglich, dass sie mehr als einmal ausgeführt wird wenn der Client die Anfrage nach einer Zeitüberschreitung erneut gesendet hatte.

At most once (höchstens einmal)

Die Prozedur wird entweder einmal oder gar nicht ausgeführt. Ein erneutes Senden der Anfrage führt nicht dazu, dass die Prozedur mehrmals ausgeführt wird.

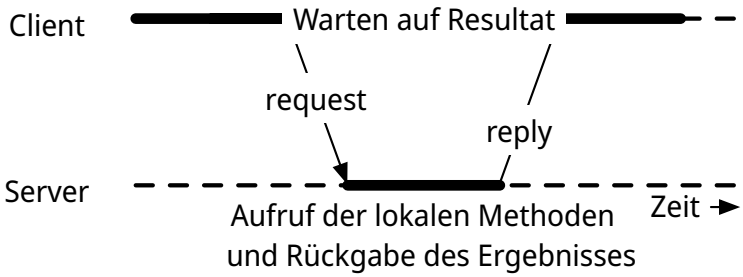
Exactly once (genau einmal)

Das System garantiert die gleiche Semantik wie bei lokalen Aufrufen unter der Annahme, dass ein abgestürzter Server irgendwann wieder startet.

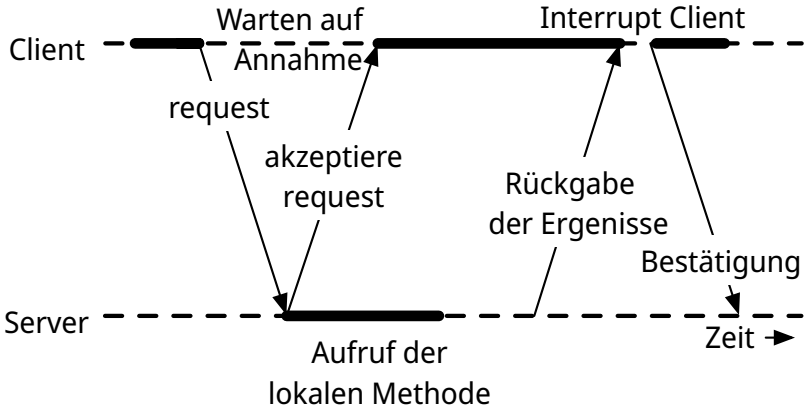
Verwaiste Aufrufe, d. h. Aufrufe auf abgestürzten Server-Rechnern, werden nachgehalten, damit sie später von einem neuen Server übernommen werden können.

Asynchrones RPC

Die Verbindung zwischen Client und Server in einem traditionellen RPC. Der Client wird blockiert und wartet.



Die Verbindung zwischen Client und Server bei einem asynchronen RPC. Der Client wird nicht blockiert.



Ein normaler Aufruf mittels `XMLHTTPRequest` (JavaScript) ist auch immer asynchron.

RPC - Bewertung

✓RPC bietet einen Mechanismus, um verteilte Anwendungen auf einfache und effiziente Weise zu implementieren.

✓RPC ermöglicht den modularen und hierarchischen Aufbau großer verteilter Systeme:

- Client und Server sind getrennte Einheiten
- Der Server kapselt und verbirgt die Details der Backend-Systeme (wie z. B. Datenbanken)

❗ RPC ist kein Standard, sondern wurde auf viele verschiedene Arten umgesetzt.

❗ RPC ermöglicht Entwicklern den Aufbau verteilter Systeme, löst aber nur ausgewählte Aspekte.

Wenn man moderne Ansätze wie RESTful WebServices mit RPC vergleicht, dann fällt auf, dass RPC eine deutlich bessere Tranzparenz bietet.

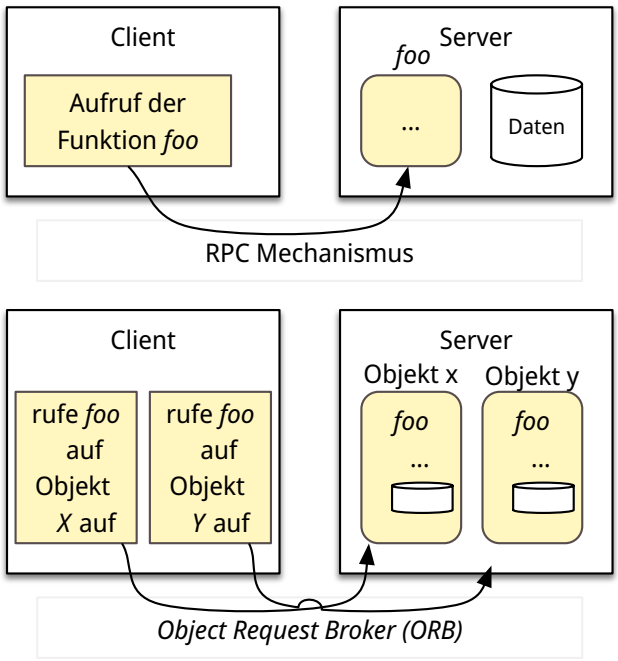
Das Network File System (NFS) und SMB sind bekannte RPC-basierte Anwendungen.

Java RMI (Remote Method Invocation)

Ermöglicht es einem Objekt, das in einer Java Virtual Machine (VM) läuft, Methoden eines Objekts aufzurufen, das in einer anderen Java VM läuft.

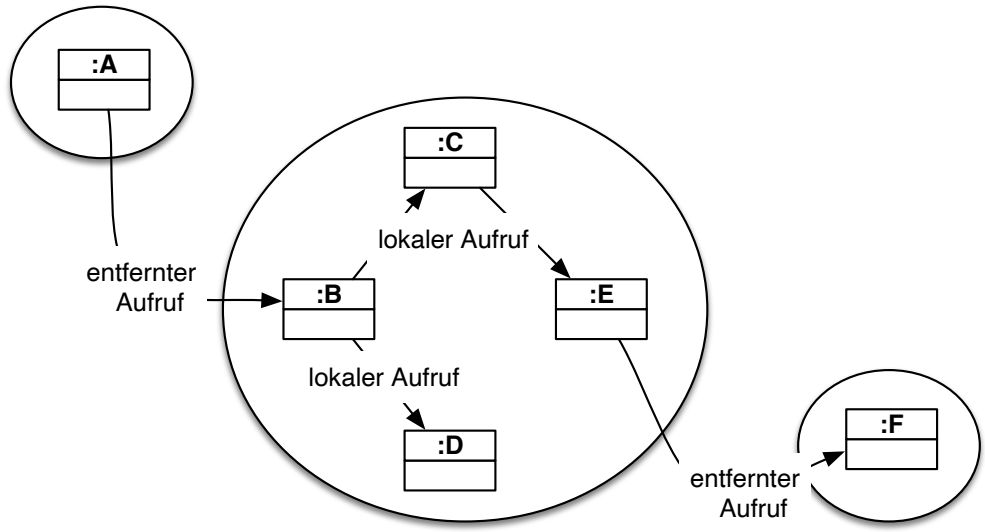
- Entfernte Objekte können ähnlich wie lokale Objekte behandelt werden.
- Übernimmt das Marshalling, den Transport und die Garbage Collection der entfernten Objekte.
- Teil von Java seit JDK 1.1

Java RMI vs. RPC



Java RMI ist eine spezielle Form von RPC, die in Java implementiert wurde. Der Unterschied ergibt sich im Prinzip aus dem Unterschied zwischen einem Prozeduraufruf und einem Methodenaufruf auf ein Objekt

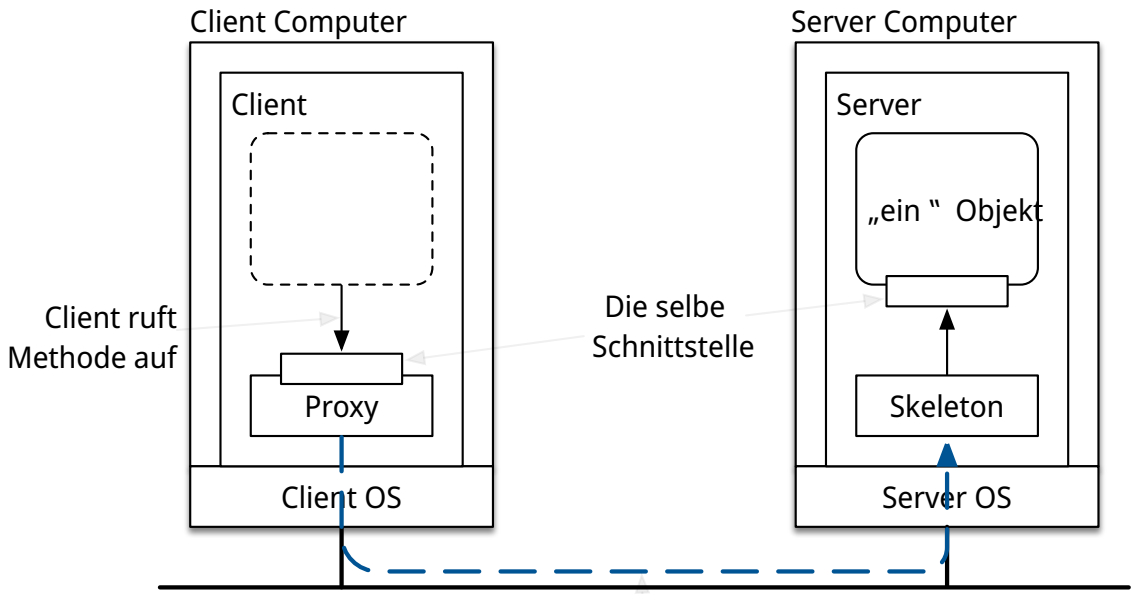
Java RMI implementiert ein *Distributed Object Model*



- Jeder Prozess enthält sowohl Objekte die entfernte Aufrufe empfangen können als auch solche, die nur lokale Aufrufe empfangen können.

  (Objekte die entfernte Aufrufe empfangen können, werden *Remote Objects* genannt).
- Objekte müssen die Remote-Objektreferenz eines Objekts in einem anderen Prozess kennen, um dessen Methoden aufrufen zu können (Remote Method Invocation; Remote Object References)

Anatomie eine Java RMI Aufrufs



Der Aufruf (in Form eines „Marshalled Object“) wird über das Netz weitergereicht

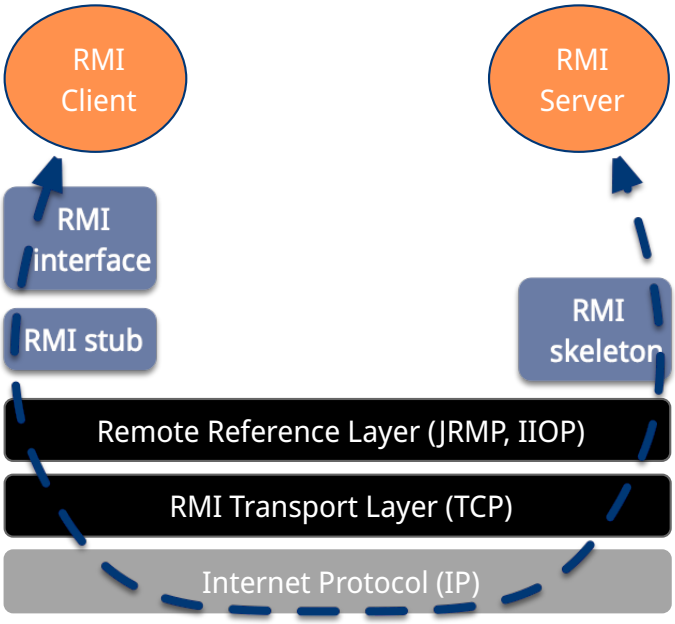
---

Der Proxy versteckt für den Client, dass es sich um einen entfernten Aufrufe handelt. Er implementiert die Remote-Schnittstelle und kümmert sich um das Marshalling und Unmarshalling der Parameter und des Ergebnisses.

Der Skeleton ist für die Entgegennahme der Nachrichten verantwortlich und leitet die Nachricht an das eigentliche Objekt weiter. Er sorgt für die Transparenz auf Serverseite.

Referenzen auf *Remote Objects* sind systemweit eindeutig und können frei zwischen Prozessen weitergegeben werden (z. B. als Parameter). Die Implementierung der entfernten Objektreferenzen wird von der Middleware verborgen (*Opaque-Referenzen*).

RMI Protocol Stack



- *Remote Reference Layer*: RMI-spezifische Kommunikation über TCP/IP, Verbindungsinitialisierung, Serverstandort, Verarbeitung serialisierter Daten
- *RMI Transport Layer (TCP)*: Verbindungsverwaltung, Bereitstellung einer zuverlässigen Datenübertragung zwischen Endpunkten
- Internetprotokoll in IP-Paketen enthaltene Datenübertragung (unterste Ebene)

Einfacher RMI Dienst und Aufruf

Schnittstelle des Zeitservers

```
import java.rmi.Remote;
import java.rmi.RemoteException;
import java.util.Date;

public interface Time extends Remote {
  public Date getTime() throws RemoteException;
}
```

Implementierung der Schnittelle durch den Zeitserver

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class TimeServer extends UnicastRemoteObject implements Time {
  public TimeServer() throws RemoteException {
    super();
  }

  public Date getTime() {
    return new Date();
  }
}
```

Registrierung des Zeitservers

```
import java.rmi.Naming;

public class TimeRegistrar {

  /** @param args args[0] has to specify the hostname. */
  public static void main(String[] args) throws Exception {
    String host = args[0];
    TimeServer timeServer = new TimeServer();
    Naming.rebind("rmi://" + host + "/ServerTime", timeServer);
  }
}
```

Client des Zeitservers

```
import java.rmi.Naming;
import java.util.Date;

public class TimeClient {
  public static void main(String[] args) throws Exception {
    String host = args[0];
    Time timeServer = (Time) Naming.lookup("rmi://" + host + "/ServerTime");
    System.out.println("Time on " + host + " is " + timeServer.getTime());
  }
}
```

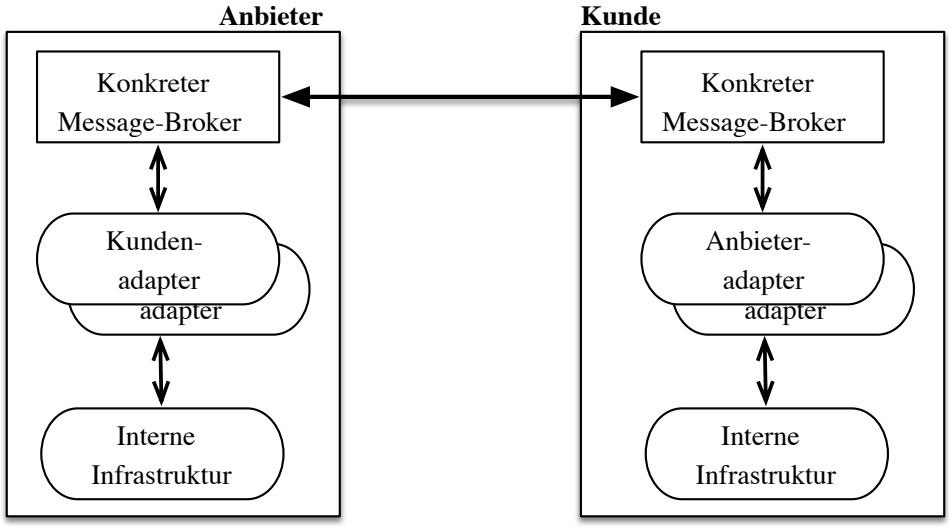
Java RMI - Tidbits

- RMI verwendet einen referenzzählenden Garbage-Collection-Algorithmus. Netzwerkprobleme können dann zu einer verfrühten GC führen was wiederum bei Aufrufen zu Ausnahmen führen kann.
- Die Aufrufsemantik (*Call Semantics*) von RMI ist *at most once*.
- (Un)Marshalling ist in Java RMI automatisch und verwendet Java Object Serialization.

  Der Overhead kann leicht ~25%-50% der Zeit für einen entfernten Aufruf ausmachen.

Integration von Unternehmensanwendungen

Die Probleme unternehmensübergreifende Punkt-zu-Punkt-Integration zu ermöglichen führten zur Entwicklung der nächsten Generation von Middleware-Technologien.



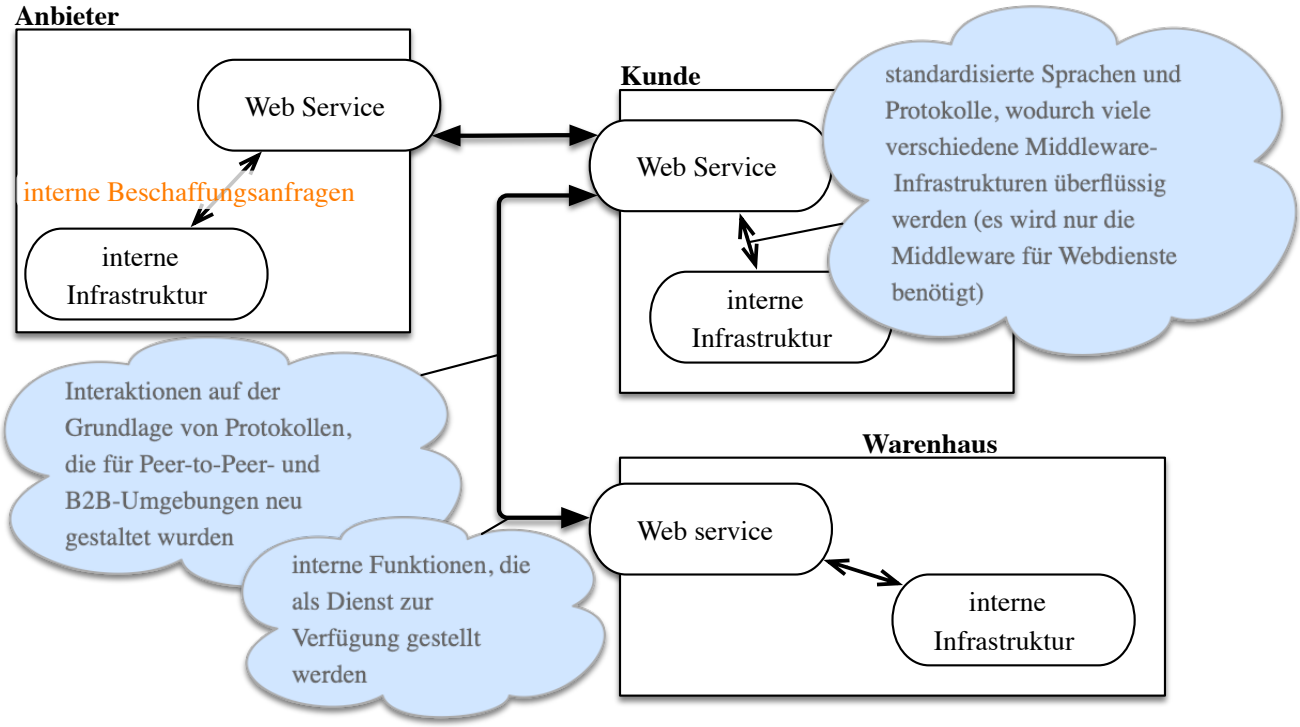
Darstellung nach *Web Services - Concepts, Architectures and Applications; Alonso et al.; Springer 2004*

Jedes Unternehmen verwendet(e) seinen eigenen „konkreten“` Message-Broker - wenn wir mit mehreren Unternehmen kommunizieren wollen, müssen wir mehrere Adapter/Lösungen implementieren und pflegen.

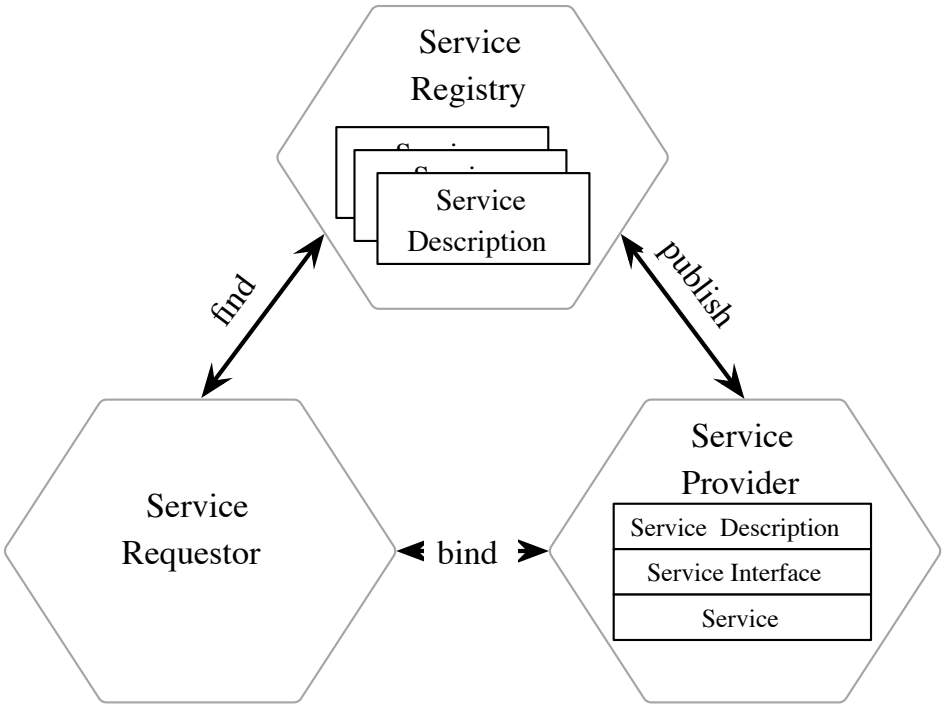
Webservices are self-contained, modular business applications that have open, internet-oriented, standards-based interfaces.

—UDDI Konsortium

Web Services - konzeptionell



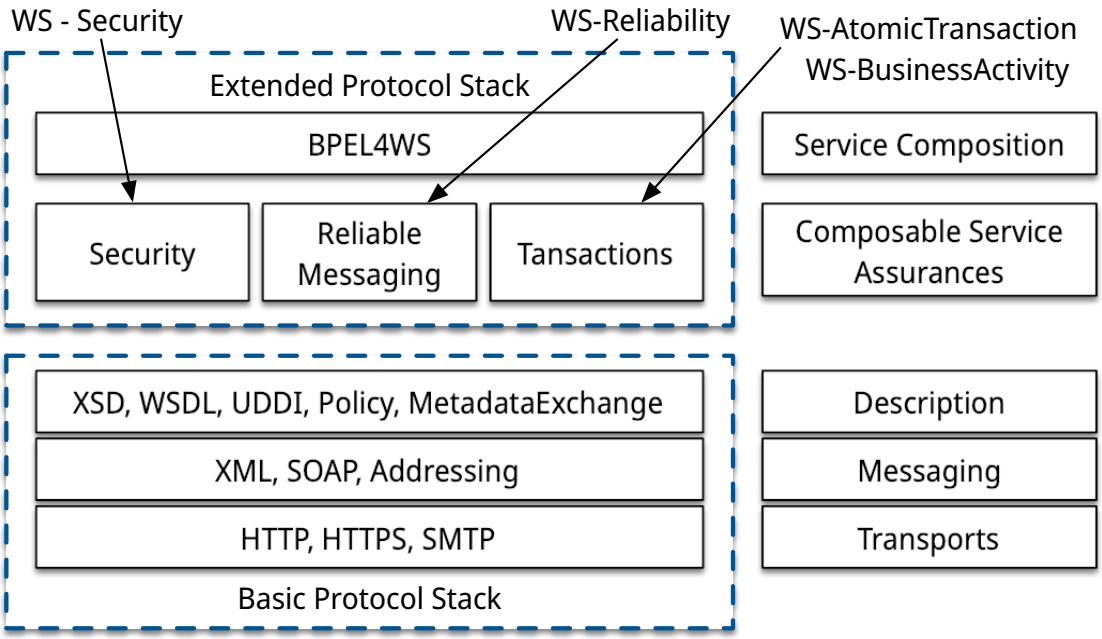
Web Services - wesentliche Bestandteile



- *Service Provider*: Die Einheit, die den Dienst implementiert und anbietet ihn im Namen des Anforderers auszuführen.
- *Service Requestor*: Der potenzielle Nutzer eines Dienstes.
- *Service Registry*: Auflistung der verfügbaren Dienste.

Konzeptionell hat sich somit im Vergleich zur RPC-Welt nicht viel geändert.

Web Services - Protokoll Stack



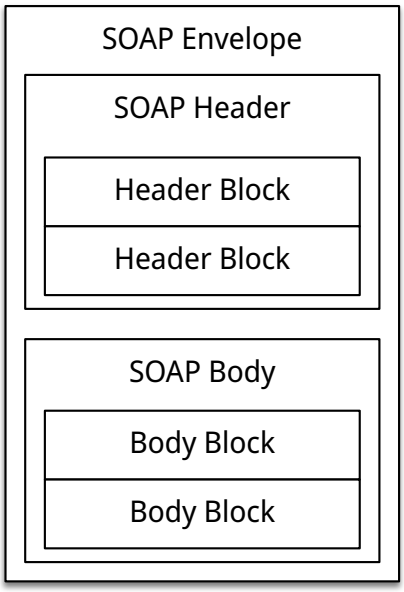
SOAP

- SOAP ist das Protokoll klassischer Web Services und ermöglicht die Kommunikation zwischen Anwendungen.
- SOAP umfasst die folgenden Teile:
  - Ein Nachrichtenformat, das beschreibt, wie eine Nachricht in ein XML-Dokument verpackt werden kann (Umschläge, Header, Body...)
  - Ein Satz von Kodierungsregeln für Daten
  - Eine Beschreibung wie eine SOAP-Nachricht mit dem zugrundeliegenden Transportprotokoll (HTTP oder SMTP) transportiert werden sollte. Wie eine SOAP-Nachricht in eine HTTP-Anfrage oder in eine E-Mail (SMTP) eingebettet werden kann.
  - Eine Reihe von Regeln, die bei der Verarbeitung einer SOAP-Nachricht zu befolgen sind, und die an dieser Verarbeitung beteiligten Stellen; welche Teile der Nachrichten von wem gelesen werden sollten und welche Maßnahmen diese Stellen ergreifen sollten, wenn sie den Inhalt nicht verstehen.

SOAP ist eine Weiterentwicklung von XML-RPC und stand ursprünglich für Simple Object Access Protocol.

SOAP (ab Version 1.2) ist ein Standard des W3C.

Aufbau einer SOAP-Nachricht



Nachrichten sind Umschläge, in die die Nutzdaten der Anwendung eingeschlossen werden.

Eine Nachricht hat zwei Hauptbestandteile:

Header (optional): Für infrastrukturelle Daten wie Sicherheit oder Zuverlässigkeit vorgesehen.

Body (obligatorisch): Für Daten auf Anwendungsebene vorgesehen. Jeder Teil kann in Blöcke unterteilt werden.

Beispiel einer SOAP-Nachricht

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />

<SOAP-ENV:Header>
  <t:Transaction xmlns:t="ws-transactions-URI" SOAP-ENV:mustUnderstand="1">
    57539
  </t:Transaction>
</SOAP-ENV:Header>

<SOAP-ENV:Body>
  <m:GetLastTradePrice xmlns:m="Some-URI">
    <symbol>DEF</symbol>
  </m:GetLastTradePrice>
</SOAP-ENV:Body>

</SOAP-ENV:Envelope>
```

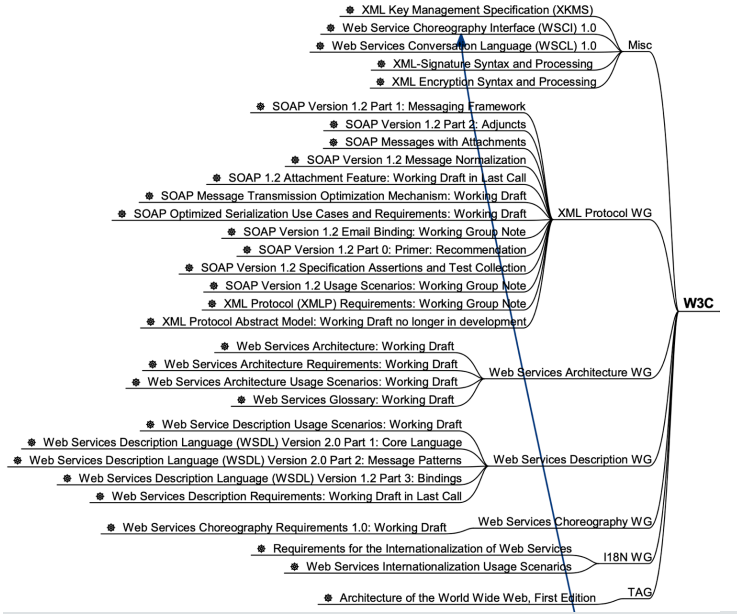
Beispiel eines SOAP-Aufrufs

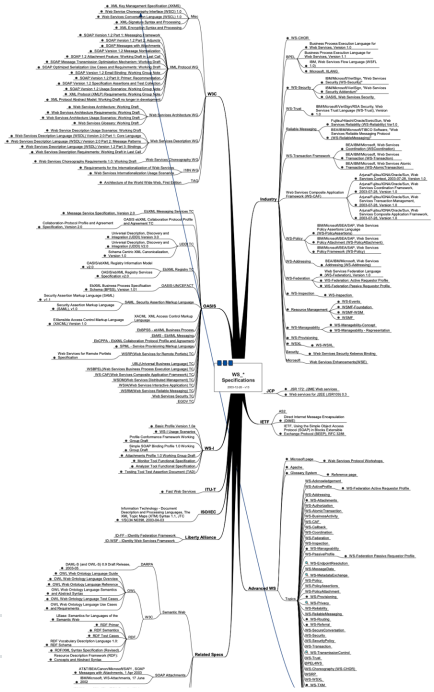
```
POST /StockQuote HTTP/1.1
Host: www.stockquoteserver.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">

  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>DIS</symbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Beispiel einer SOAP-Antwort

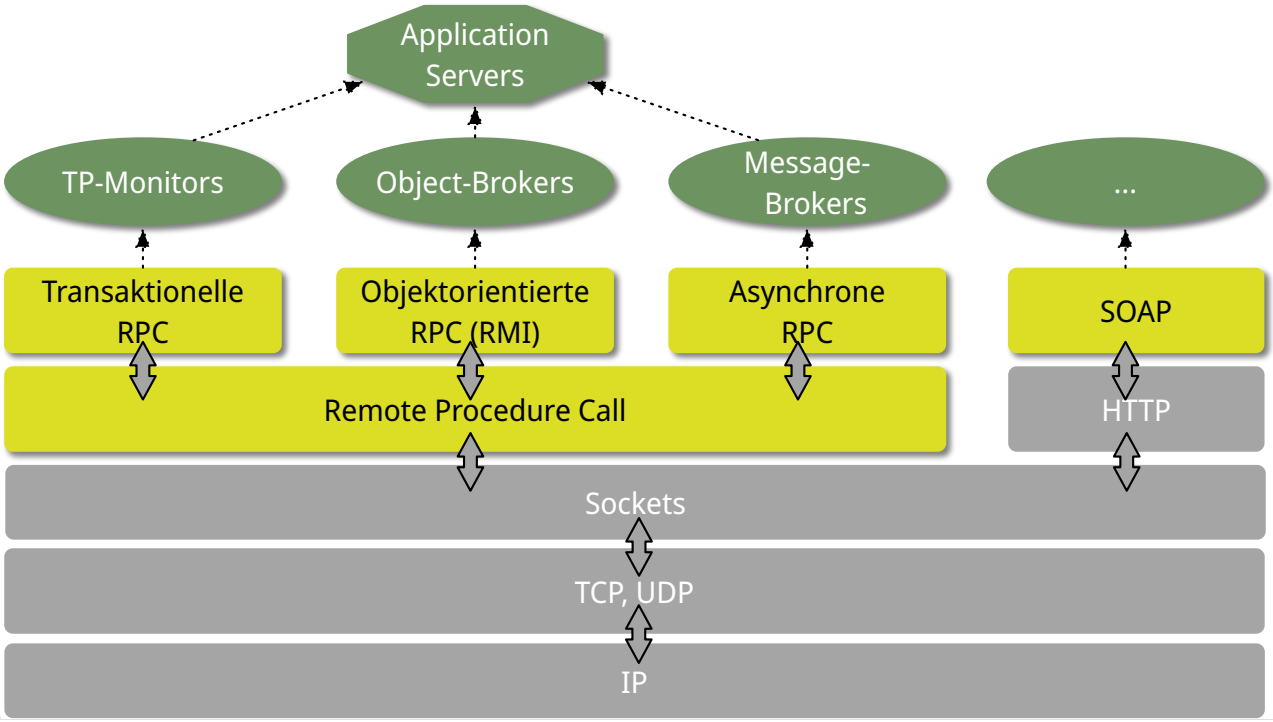
```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />

<SOAP-ENV:Body>
  <m:GetLastTradePriceResponse xmlns:m="Some-URI">
    <Price>34.5</Price>
  </m:GetLastTradePriceResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Web Services - Standardisierung





Überblick


Application Servers
TP-Monitors
Object-Brokers
Message-Brokers
...
Transaktionelle RPC
Objektorientierte RPC (RMI)
Asynchrone RPC
SOAP
Remote Procedure Call
HTTP
Sockets
TCP, UDP
IP

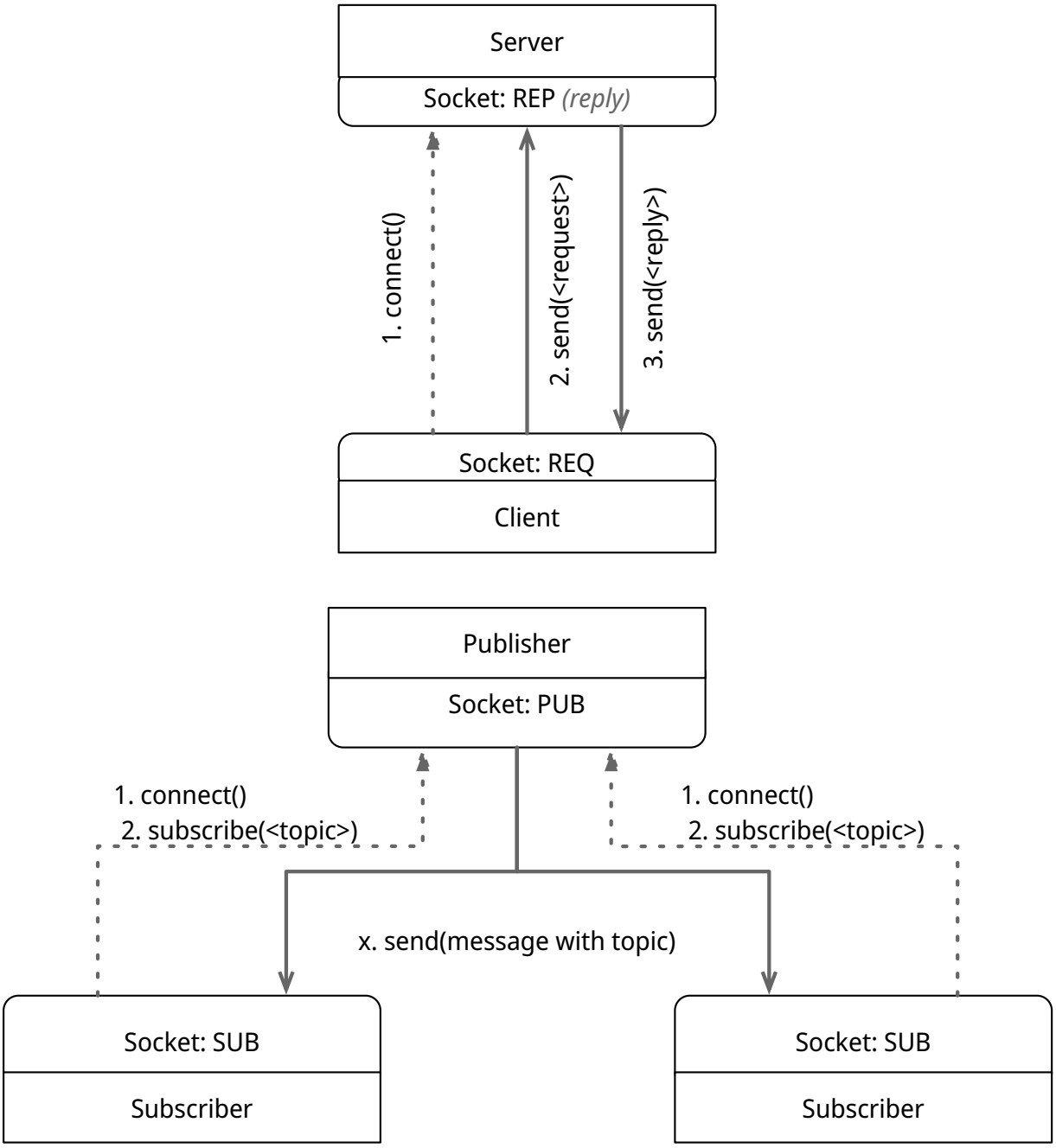
3. Messaging and Message-oriented Communication/Middleware

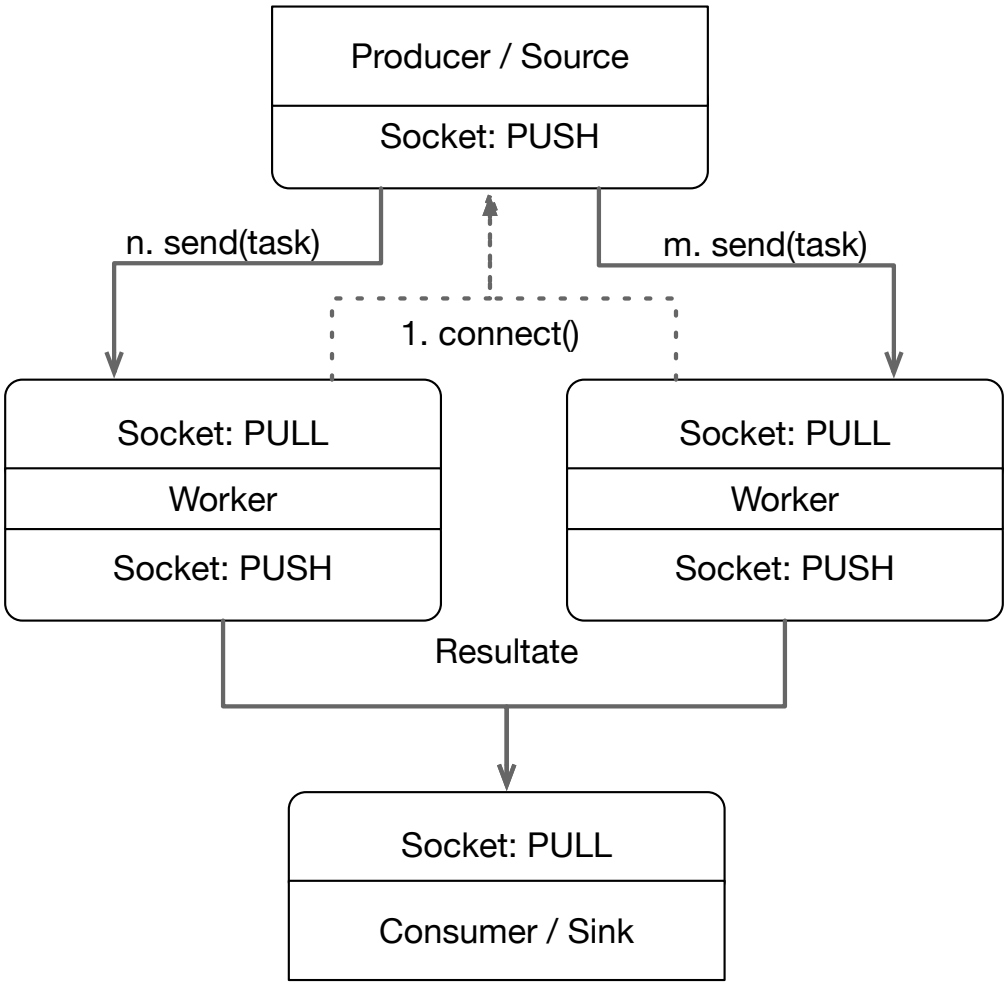
ZeroMQ

- ZeroMQ ist eine Messaging-Infrastruktur ohne explizite Server („Broker“).
- ZeroMQ unterstützt verbindungsorientierte aber asynchrone Kommunikation.
- ZeroMQ basiert auf klassischen Sockets, fügt aber neue Abstraktionen hinzu, um folgende Messaging Patterns zu ermöglichen:
  - *request-reply*
  - *pub-sub* (*publish-subscribe*)
  - pipeplining (*parallele Verarbeitung*)
- ZeroMQ ermöglicht N-zu-N Kommunikation.
- ZeroMQ unterstützt sehr viele Programmiersprachen; der Nutzer ist für das passend Marshalling bzw. Unmarshalling verantwortlich.

Sollte zum Beispiel der Server in Java und der Client in C geschrieben sein, dann ist ggf. das Verständnis darüber wie ein String übertragen wird unterschiedlich (z. B. mit `null` terminiert oder mit einer Länge versehen).

ZeroMQ - Messaging Patterns





Client-Server:	Ermöglicht die „übliche“ Kommunikation zwischen einem Client und einem Server. Allerdings findet ggf. eine Pufferung statt, wenn der Server nicht erreichbar ist.
Publish-Subscribe:	Ermöglicht es den Clients, sich für ein bestimmtes Thema zu registrieren und dann alle Nachrichten zu erhalten, die zu diesem Thema veröffentlicht werden. Ein Nachricht mit einem bestimmten Thema wird an alle dafür registrierten Clients gesendet.
Pipeline:	Ermöglicht die Versendung einer Aufgabe an genau einen beliebigen Worker aus einer Menge von (homogenen) Workern.

ZeroMQ - Beispiel *Publish-Subscribe* (Java)

Publisher

```
import static java.lang.Thread.currentThread
import org.zeromq.SocketType;
import org.zeromq.ZMQ;
import org.zeromq.ZContext;

public class Publisher {
  public static void main(String[] args)
      throws Exception {
    try (ZContext context = new ZContext()) {
      ZMQ.Socket publisher =
          context.createSocket(SocketType.PUB);
      publisher.bind("tcp://*:5556");
      publisher.bind("ipc://" + <endpoint>);

      while (!currentThread().isInterrupted()) {
        int zipcode = <some zipcode>
        //  Send to all subscribers
        String update = String.format("%05d %s",
            zipcode, <some msg>);
        publisher.send(update, 0);
      }
} } }
```

ZeroMQ - Beispiel *Publish-Subscribe* (Java)

Subscriber

```java
import java.util.StringTokenizer;

import org.zeromq.SocketType;
import org.zeromq.ZMQ;
import org.zeromq.ZContext;

public class Subscriber{
  public static void main(String[] args) {
    try (ZContext context = new ZContext()) {
      ZMQ.Socket subscriber =
          context.createSocket(SocketType.SUB);
      subscriber.connect("tcp://localhost:5556");
      subscriber.subscribe(
          <zipcode(Str)>.getBytes(ZMQ.CHARSET));
      while(true) {
        String string = subscriber.recvStr(0);
        // e.g. take string apart:
        //   part1: zipcode
        //   part2: message
        System.out.println(string);
      }
} } }
```

ZeroMQ - Beispiel *Publish-Subscribe* (Python)

```python
import signal
import time
import zmq

signal.signal(signal.SIGINT,
              signal.SIG_DFL)

context = zmq.Context()
socket = context.socket(zmq.PUB)
socket.bind('tcp://*:5555')

for i in range(5):
    socket.send(b'status 5')
    socket.send(b'All is well')
    time.sleep(1)
```

```python
import signal
import zmq


signal.signal(signal.SIGINT,
              signal.SIG_DFL)

context = zmq.Context()
socket = context.socket(zmq.SUB)
socket.connect('tcp://localhost:5555')
socket.setsockopt(zmq.SUBSCRIBE, b'status')

while True:
    message = socket.recv_multipart()
    print(f'Received: {message}')
```

Bzgl. des Handlings von Signalen in Python siehe auch: https://docs.python.org/3/library/signal.html#signal.signal

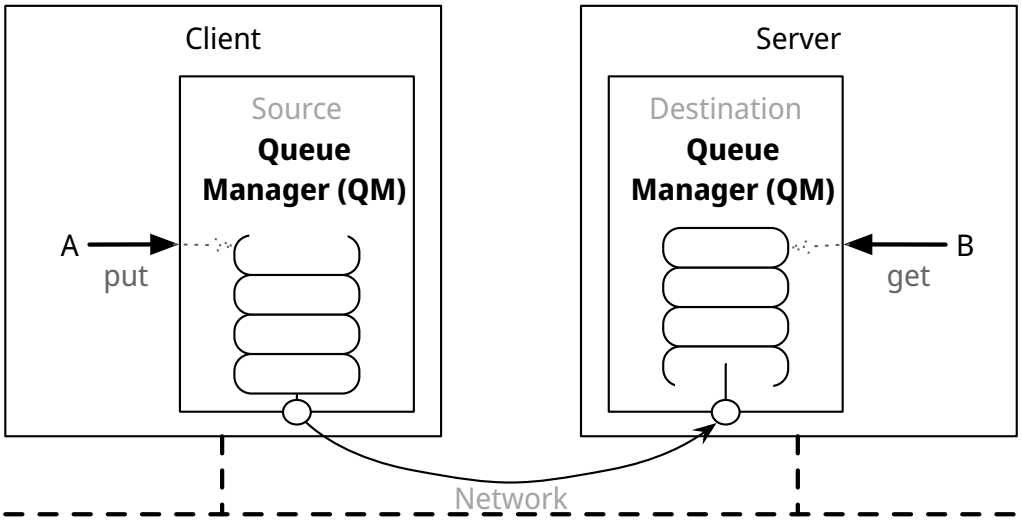
MOM - Message Oriented Middleware

- MOM bzw. Message-queueing Systems unterstützen persistente asynchrone Kommunikation.
- Sehr große Nachrichten werden unterstützt.
- Es wird nur die Garantie gegeben, dass Nachrichten letztendlich in die Warteschlange des Empfängers gelegt werden und die Nachrichten in der richtigen Reihenfolge ankommen.
  (Insbesondere wird keine Garantie gegeben, dass die Nachricht gelesen wird.)
- Der Sender und Empfänger sind nicht notwendigerweise gleichzeitig aktiv.
- Nachrichten haben immer einen eindeutigen Empfänger und quasi beliebigen Inhalt.

MOM - Grundlegendes Interface

Operation	Beschreibung
PUT	Legt eine Nachricht in eine bestimmte Warteschlange.
GET	Blockiert an einer bestimmten Warteschlange bis eine Nachricht verfügbar ist. Entfernt die erste Nachricht.
POLL	Prüft, ob eine Nachricht in einer bestimmten Warteschlange verfügbar ist. Entfernt ggf. die erste Nachricht. POLL blockiert niemals
NOTIFY	Registriert einen Handler (*Callback*) der aufgerufen wird, wenn eine Nachricht einer bestimmten Warteschlange hinzugefügt wird.

MOM - Queue Managers



Queue Managers sind der zentrale Baustein von Message-queueing Systemen. Im Allgemeinen gibt es (mindestens konzeptionell) einen lokalen *Queue Manager* pro Prozess. Ein *Queue Manager* ist ein Prozess, der Nachrichten in Warteschlangen speichert und verwaltet. Bei Bedarf kann er mehrere Warteschlangen verwalten und an andere *Queue Manager* weiterleiten.

Übung - Java

3.1. Asynchrone, verbindungsorientierte Kommunikation

Entwickeln Sie einen Client für einen „Logging Server“, der Lognachrichten (Strings) an den Server sendet. Im Fehlerfall, z. B. wenn der Server nicht verfügbar ist oder es zu einer Netzwerkpartitionierung kam, sollen die Nachrichten zwischengepuffert werden und bei Serververfügbarkeit wieder zugestellt werden. Mit anderen Worten: Im Fehlerfall soll der Client nicht blockieren, sondern weiter funktionieren. Der Client stellt stattdessen die Nachrichten dann zu, wenn der Server wieder verfügbar wird.

Stellen Sie sicher, dass Nachrichten immer in der richtigen Reihenfolge am Server ankommen. D. h. stellen Sie zum Beispiel sicher, dass eine gepufferte Nachricht nie nach einer neueren Nachricht ankommt.

Verwenden Sie den Code im Anhang als Schablone.

Einfacher TCP basierter SyslogServer in Java

```
import java.net.*;
import java.io.*;

public class SyslogServer {
  public static void main(String[] args) {
    ServerSocket server = new ServerSocket(9999);
    try {
      while (true) {
        try (Socket con = server.accept()) {
            var in = con.getInputStream();
            var ir = new InputStreamReader(in);
            var br = new BufferedReader(ir);
            System.out.println("[Logging] " + br.readLine());
        } catch (IOException e) {
            System.err.println(e);
        }
      }
    } catch (IOException e) {
        System.err.println(e);
    } finally {
      if (server != null) {
        server.close();
      }
    }
  }
}
```

Schablone für den Client in Java

```
import java.net.*;
import java.io.*;

public class Client {

  /**
   * Versendet die Nachricht an den Server (wenn möglich).
   */
  private static void sendMsg(String msg) throws IOException{
    try (Socket s = new Socket("localhost", 9999)) {
      BufferedReader networkIn =
          new BufferedReader(
              new InputStreamReader(s.getInputStream()));
      PrintWriter networkOut =
```

```
            new PrintWriter(s.getOutputStream());
        networkOut.println(msg);
        networkOut.flush();
      }
    }

    > Datenstruktur zum Zwischenspeichern der
    > bisher nicht erfolgreich versendeten Nachrichten!

    public static void log(String msg) {
        > Schicke Nachricht an den Server (wenn möglich).
        > Blockiert nicht, wenn der Server nicht verfügbar ist.
    }

    public static void startThread() throws Exception {
        Thread.ofVirtual().start(() -> {
            while (true) {
                try {
                  // Alle 5 Sekunden prüfen wir ob wir noch
                  // nicht versendete Nachrichten haben:
                  Thread.sleep(5000);
                } catch (InterruptedException e) { }
                > Versende Nachrichten,
                > die noch nicht versendet wurden
            }
        });
    }

    public static void main(String[] args) throws Exception {
        startThread();
        BufferedReader userIn =
            new BufferedReader(
                new InputStreamReader(System.in));
        while (true) {
            String theLine = userIn.readLine();
            if (theLine == null)
                break;
            log(theLine);
        }
    }
}
```

Übung - Python

3.2. Asynchrone, verbindungsorientierte Kommunikation

Entwickeln Sie sowohl einen Client (bzw. eine Clientkomponente) als auch einen Server für das zentralisierte Loggen von Nachrichten.

Im Fehlerfall, z. B. wenn der Server nicht verfügbar ist oder es zu einer Netzwerkpartitionierung kam, sollen die Nachrichten, die der Client an den Server senden will/wollte, zwischengepuffert werden und bei Serververfügbarkeit wieder zugestellt werden. Mit anderen Worten: die Methode des Clients zum senden von Nachrichten sollte nicht blockieren, sondern immer weiter funktionieren - auch im Fehlerfall.

Anforderungen

- Stellen Sie sicher, dass Nachrichten immer in der richtigen Reihenfolge am Server ankommen. D. h. stellen Sie zum Beispiel sicher, dass eine gepufferte Nachricht nie nach einer neueren Nachricht ankommt.
- Der Client nimmt (hier) die Nachrichten über die Konsole entgegen und sendet sie direkt an den Server. Der Server sollte diese dann sofort ausgeben!
- Stellen Sie sicher, dass keine Nachrichten verloren gehen, wenn der Server unkontrolliert beendet wird.
- Bevor Sie versuchen eine Nachricht wieder zu versenden, warten Sie X Sekunden (z. B. 5 Sekunden).

Hinweise

- Orientieren Sie sich an dem Code auf den Folien.
- Nutzen Sie ggf. die Möglichkeit Sockets in `File`-Objekte zu verwandeln, um die Nachrichten zu senden. Vergessen Sie ggf. nicht `flush()` aufzurufen, damit die Nachricht auch wirklich versendet wird.
- Prüfen Sie explizit, dass - wenn Sie Ihren Server abrupt beenden (CTRL+C) - und dann ganz schnell mehrere kleine Nachrichten senden, dass diese auch später *alle* ankommen. Falls dies nicht der Fall ist, überlegen Sie sich, wie Sie das Problem lösen können und implementieren Sie die Lösung.

Keine Anforderungen

- Duplikate von Nachrichten müssen nicht erkannt werden.

Schablone für die Serverseite

```
import queue
import socket
import queue
import threading

HOST = "localhost"
PORT = 5678

PRINT_QUEUE = queue.Queue()

def print_queue_handler():
    while True:
        try:
            msg = PRINT_QUEUE.get()
            print(msg, end="")
        finally:
            PRINT_QUEUE.task_done()


def ts_print(msg):
    PRINT_QUEUE.put(msg)

# implement the server logic here...
```

```
if __name__ == "__main__":
    threading.Thread(target=print_queue_handler, daemon=True).start()
    # start/run server
    PRINT_QUEUE.join()
```