

Verteilte Systeme und Komponenten

# Serverprozesse mittels Sockets und RPC

Martin Bättig

Letzte Aktualisierung: 23. September 2022

FH Zentralschweiz



# Inhalt

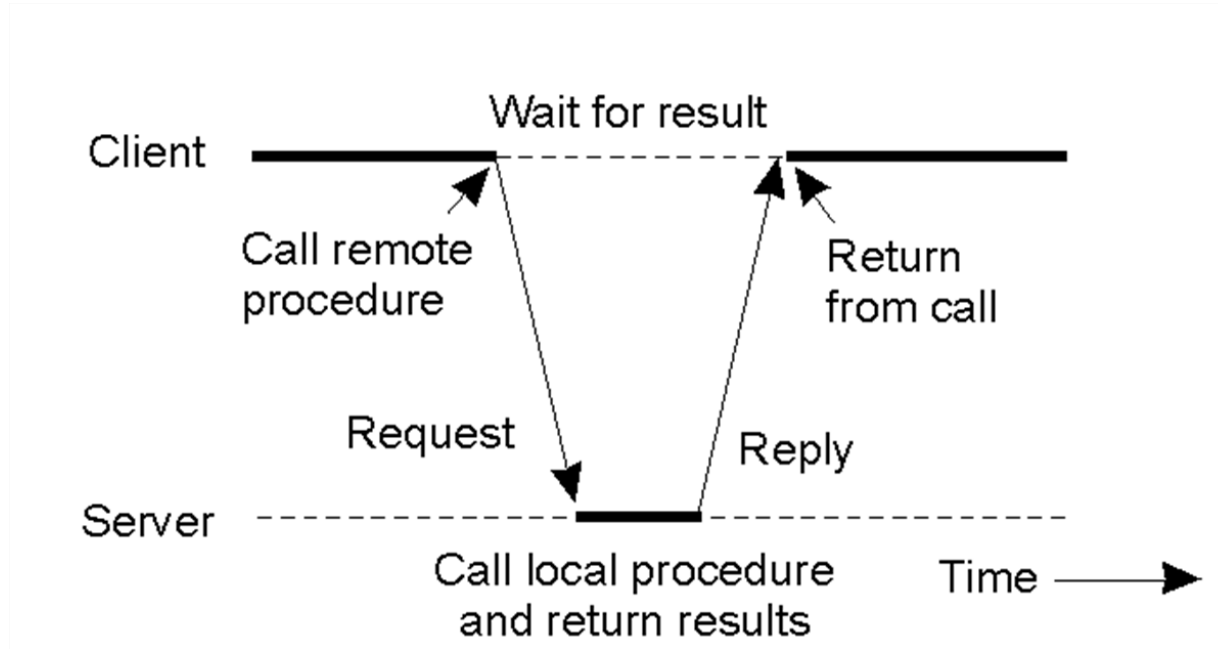
- Remote-Procedure-Call (RPC)
- TCP Client- und Serverprogramme
- gRPC
- Zusammenfassung

# Lernziele

- Sie verstehen das Prinzip des Remote-Procedure-Calls.
- Sie kennen die verschiedenen Arten um ein Serverprogramm zu implementieren und wissen, wann welche einzusetzen.
- Sie wissen, welche Aktionen nötig sind, um Daten verbindungsorientiert senden zu können.
- Sie wissen was ein Kommunikationsprotokoll ist und können ein einfaches Protokoll in Java umsetzen können.
- Sie kennen den Lebenszyklus eines TCP-Servers und können die einzelnen Elemente mit einem Java-Programm in Beziehung stellen.
- Sie können sowohl Java Client-Programme sowie Java Server-Programme analysieren und implementieren.
- Sie kennen die generelle Funktionsweise von gRPC.

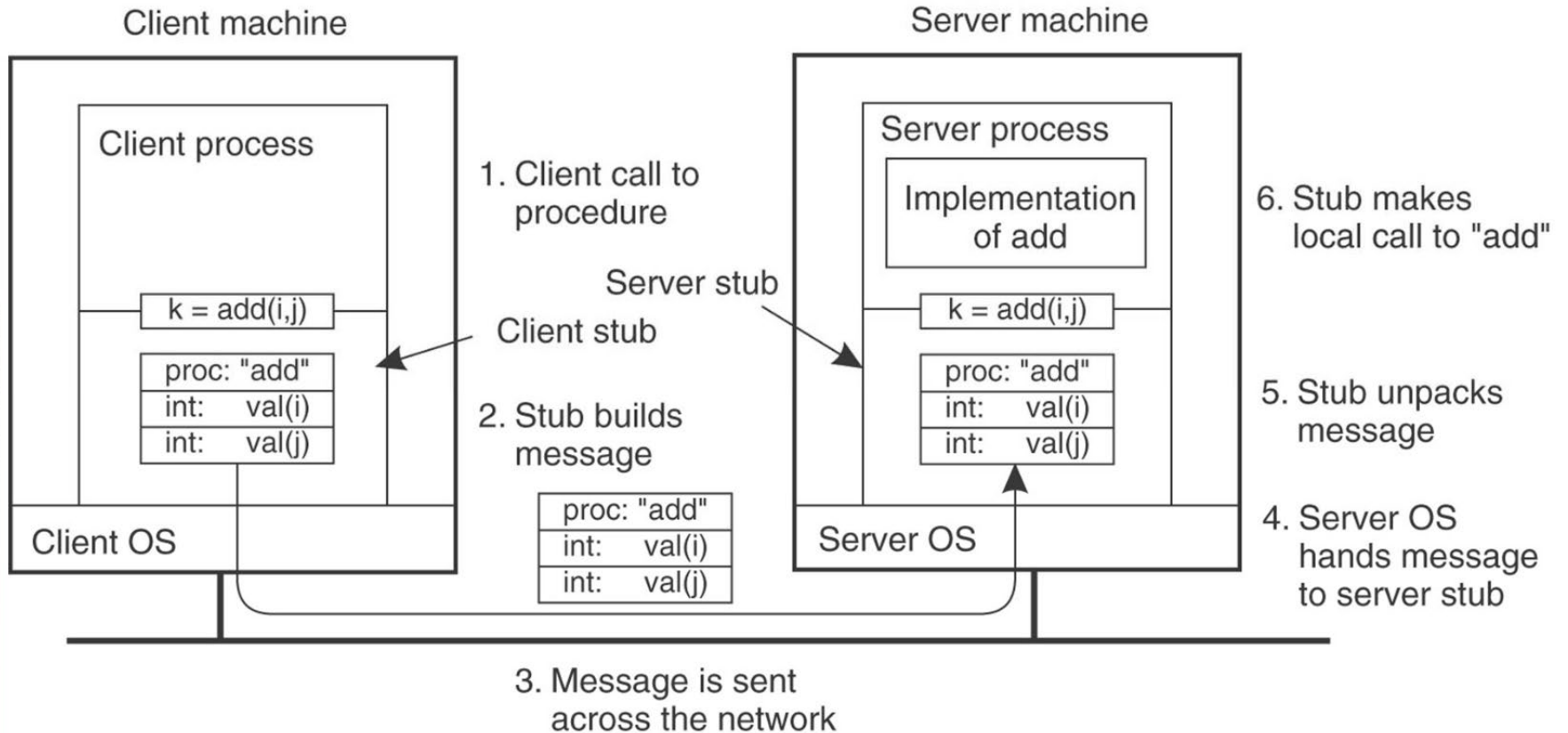
# Remote-Procedure-Call (RPC)

**Ziel:** Remote Funktionsaufruf analog zu lokalem Funktionsaufruf:



**Beispiel:** `result = doIt(a, b);`

# Remote-Procedure-Call: Funktionsprinzip

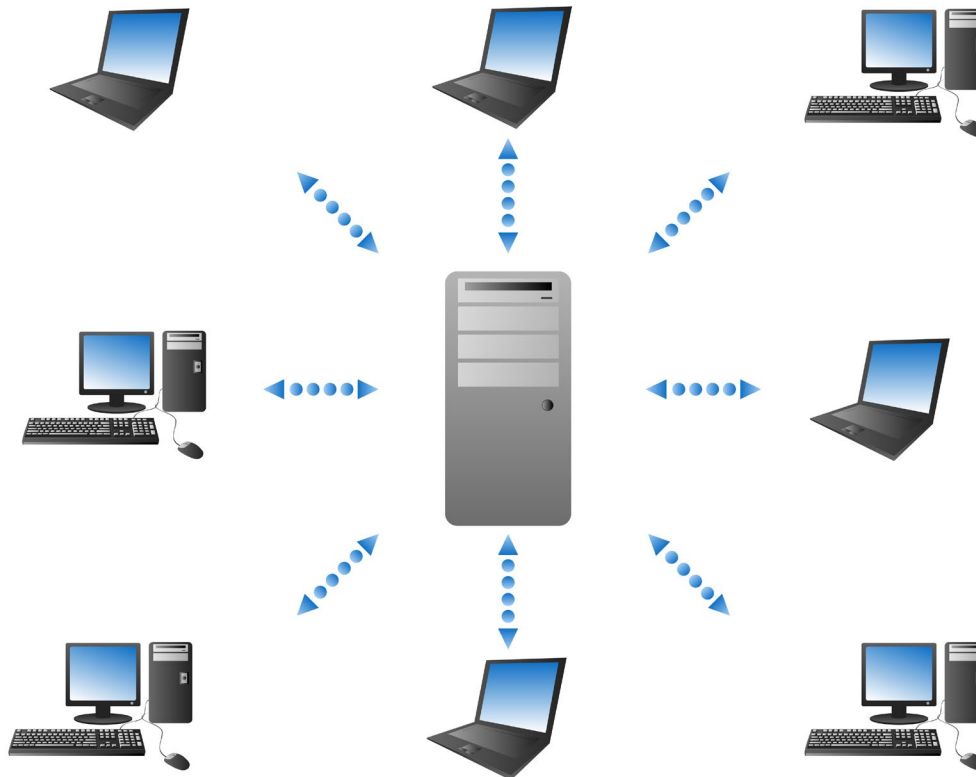


# Remote-Procedure-Call

- Vielfach verwendetes Konzept (gRPC, DCE, RMI, etc.).
- Was sind die Herausforderungen?

# Parallele (nebenläufige) Verarbeitung

- Serverprogramm muss ggf. mehr als einen Client **parallel** bedienen.



# Varianten der parallelen Verarbeitung

Variante	Einsatzgebiet
<b>Blocking I/O, Single-Threaded</b>	Eine langlebige Verbindung oder wenige kurzlebige Verbindungen.
<b>Blocking I/O, Mehrere Prozesse (Fork)</b>	Wenige langlebige Verbindungen.
<b>Blocking I/O, Mehrere Threads</b>	Wenige langlebige Verbindungen.
<b>Blocking I/O, Thread-Pools</b>	Viele kurzlebige Verbindungen.
<b>Non-Blocking I/O, Single-Threaded</b>	Viele Verbindungen (kurz- oder langlebig) mit wenig Bearbeitungszeit.
<b>Non-Blocking I/O, mehrere Threads</b>	Viele Verbindungen (kurz- oder langlebig) mit wenig bis viel Bearbeitungszeit.



# Parameterübergabe

**Beispiel:** Anhängen eines Datensatzes (data) an eine Liste (dbListe) auf einem entfernten System, Rückgabe als neue Liste (newlist).

```
newlist = append(data, dbList)
```

## Parameterübergabe:

### Lokal:

- By-value: Erstelle Kopie
- By-reference: Übergebe Referenz

### Remote:

- By-value: Erstelle Kopie
- By-reference: ???

## Fragen:

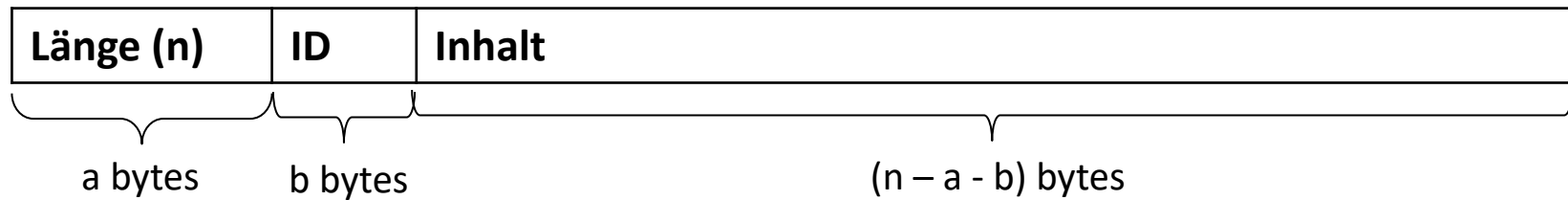
- Kosten?
- Wo sind Primitive oder Strukturen gespeichert?
- Wie identifiziere ich Primitive oder Strukturen?

# Kommunikationsprotokoll

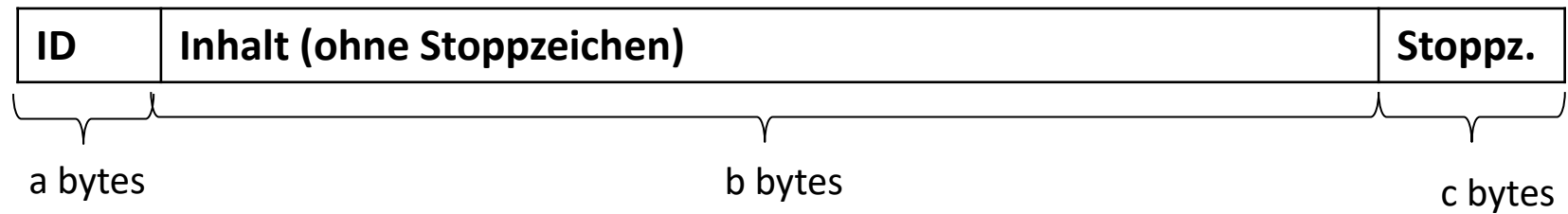
- Notwendig zur Verständigung zwischen zwei Maschinen (oft Client und Server).
- Hat Anforderungen an den Übertragungskanal (zuverlässig vs. unzuverlässig)
- Definiert generelle Nachrichtenstruktur.
  - Bspw.: Binär/Text, Länge, Stoppzeichen, Nachrichten ID, usw.
- Umfasst eine bestimmte Anzahl Nachrichten.

# Generelle Nachrichtenstruktur: Beispiele

Mit Längenangabe:



Mit Stoppzeichen:



# Nachrichten

- Teil eines Kommunikationsprotokolls.
- gesendet über Kommunikationskanal.
- enthalten Elemente bestimmter Datentypen.
  - ID:
    - identifiziert die Nachricht (z.B. GET, POST, ... bei HTTP)
    - Nicht immer benötigt: Bei strikten Interaktionen zwischen Kommunikationspartnern ist die ID unnötig. (z.B. Schach)
  - Argumente, oft abhängig von der ID:
    - Einfache Datentypen (Integer, Strings, Floating-point).
    - Strukturen oder Arrays
    - können zusätzliche Informationen enthalten, basierend auf der Art der Nachricht, z.B. eine Aktion oder Anfrage (Wandle X in Y und gibt Resultat zurück oder Berechne X mit Hilfe von A,B,C).

# Aufbau einer HTTP-Nachricht

## ■ HTTP Request:

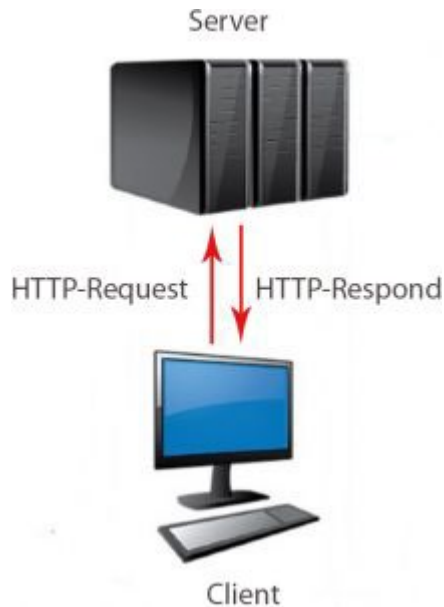
ID	METHOD
Argumente	URL
	HTTP/version
	General Header
	Request Headers
	Entity Header (optional)
	Leerzeile
	Request Entity (falls vorhanden)

## ■ HTTP Response:

ID	HTTP/version
Argumente	Status Code
	Reason Phrase
	General Header
	Response Header
	Entity Header (optional)
	Leerzeile
	Resource Entity (falls vorhanden)

# Beispiel: HTTP-Kommunikation

- Anforderung des HTML-Dokuments demo/picture.html



```
GET /demo/picture.html HTTP/1.1
Accept: */*
Accept-Language: de-ch
Accept-Encoding: deflate, gzip
User-Agent: Mozilla/4.0 (compatible; MSIE 5.01; Windows NT
5.0)
Host: localhost
Connection: Keep-Alive
```

Request

```
HTTP/1.1 200 OK
Server: ExperimentalWebServer 1.0
Content-type: text/html
Content-length: 143
```

Response

```
<html>
<head><title>Anzeige von Bildern</title></head>
<body background="pictures/bg.gif">

</body>
</html>
```

# Beispiel: SMTP-Kommunikation

## SMTP-Client

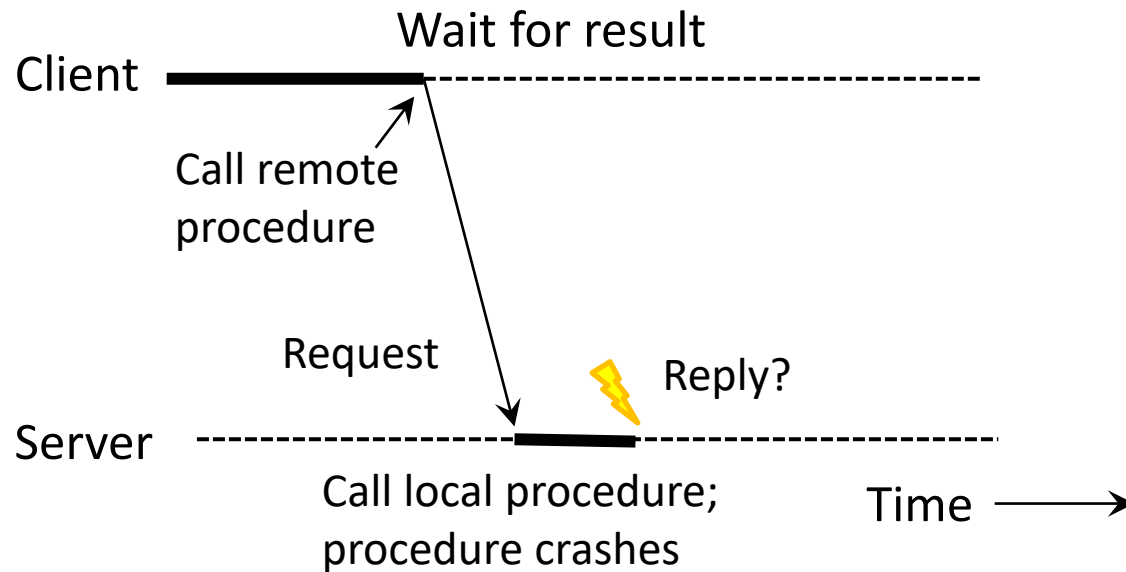
## SMTP-Server

telnet mail.example.com 25	→	
	←	220 service ready
HELO foobar.example.net	→	
	←	250 OK
MAIL FROM:<sender@example.org>	→	
	←	250 OK
RCPT TO:<receiver@example.com>	→	
	←	250 OK
DATA	→	
	←	354 start mail input
From: <sender@example.org>	→	
To: <receiver@example.com>		
Subject: Testmail		
Date: Thu, 26 Oct 2006 13:10:50 +0200		
Lorem ipsum dolor sit amet, consectetur		
ut labore et dolore magna aliqua.		
.		
	←	250 OK
QUIT	→	
	←	221 closing channel

# Fehlerbehandlung

- Wie erfolgt die Fehlerbehandlung?
  - Versuch wiederholen?
  - Aktion abbrechen?
  - Programm abbrechen?
  - etc.

## Beispiel:

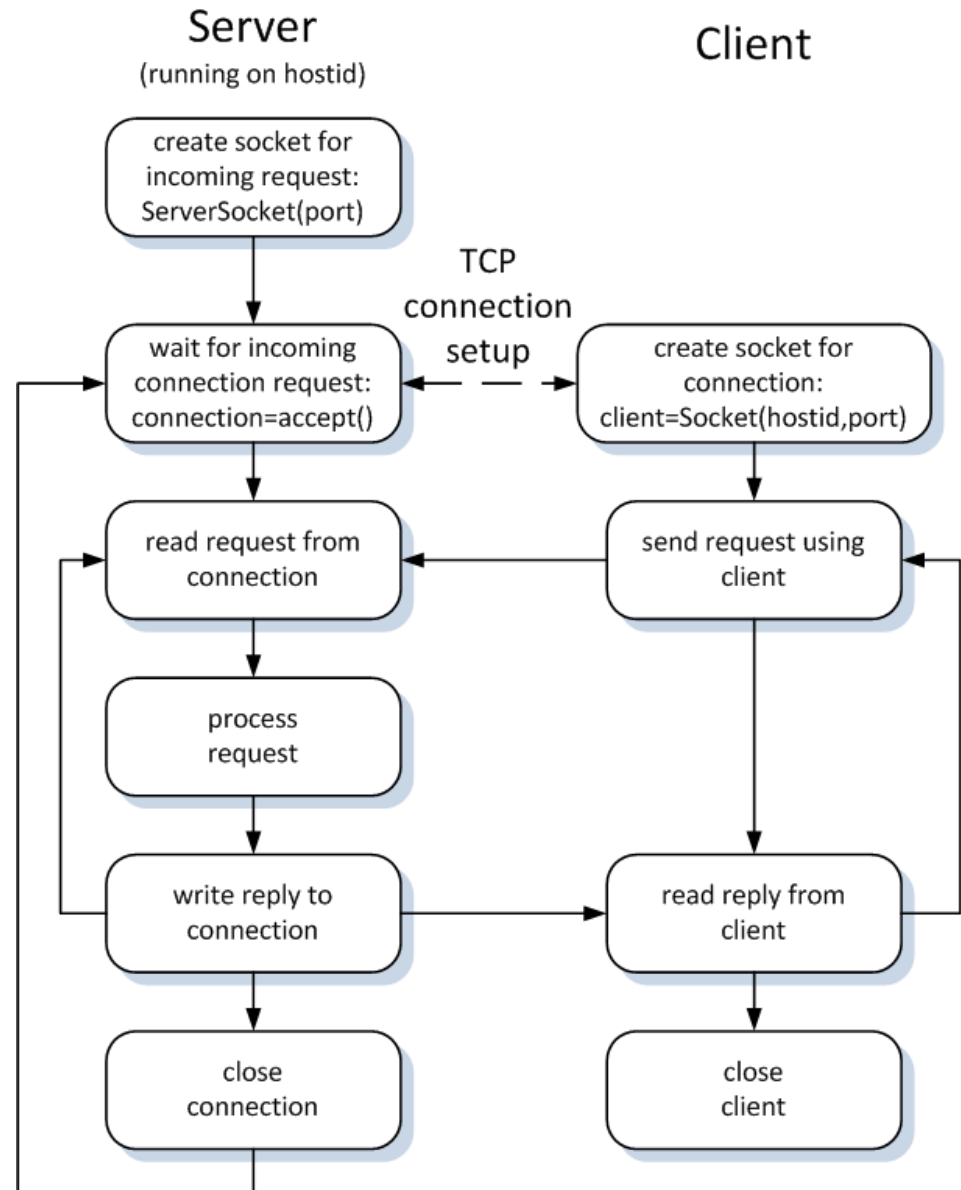




# **TCP Client- und Serverprogramme**

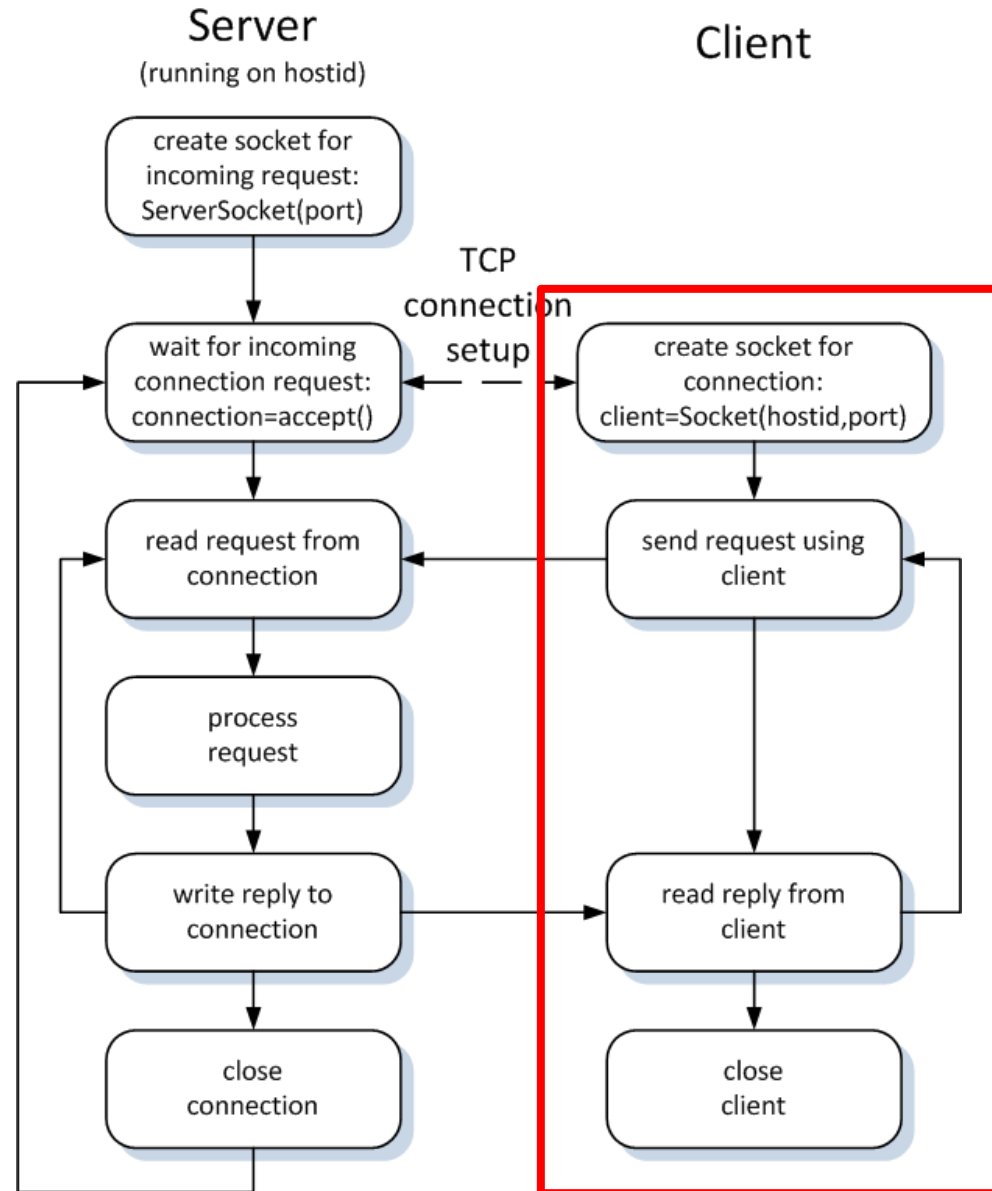
# Definitionen und Ablaufübersicht

- **Socket:** Kommunikationsendpunkt in TCP/IP-Netzwerk (IP, Port).
- **ServerSocket:** Spezieller Socket um auf eingehende Verbindungen zu warten.



# Ablauf beim Client

- Um Daten verbindungsorientiert zu versenden, sind folgende Aktionen nötig:
  1. Socket erzeugen
  2. Socket an einen lokalen Port binden (Server-seitig)
  3. Verbindung mit Zieladresse herstellen
  4. Daten über Socket lesen/schreiben
  5. Socket schliessen

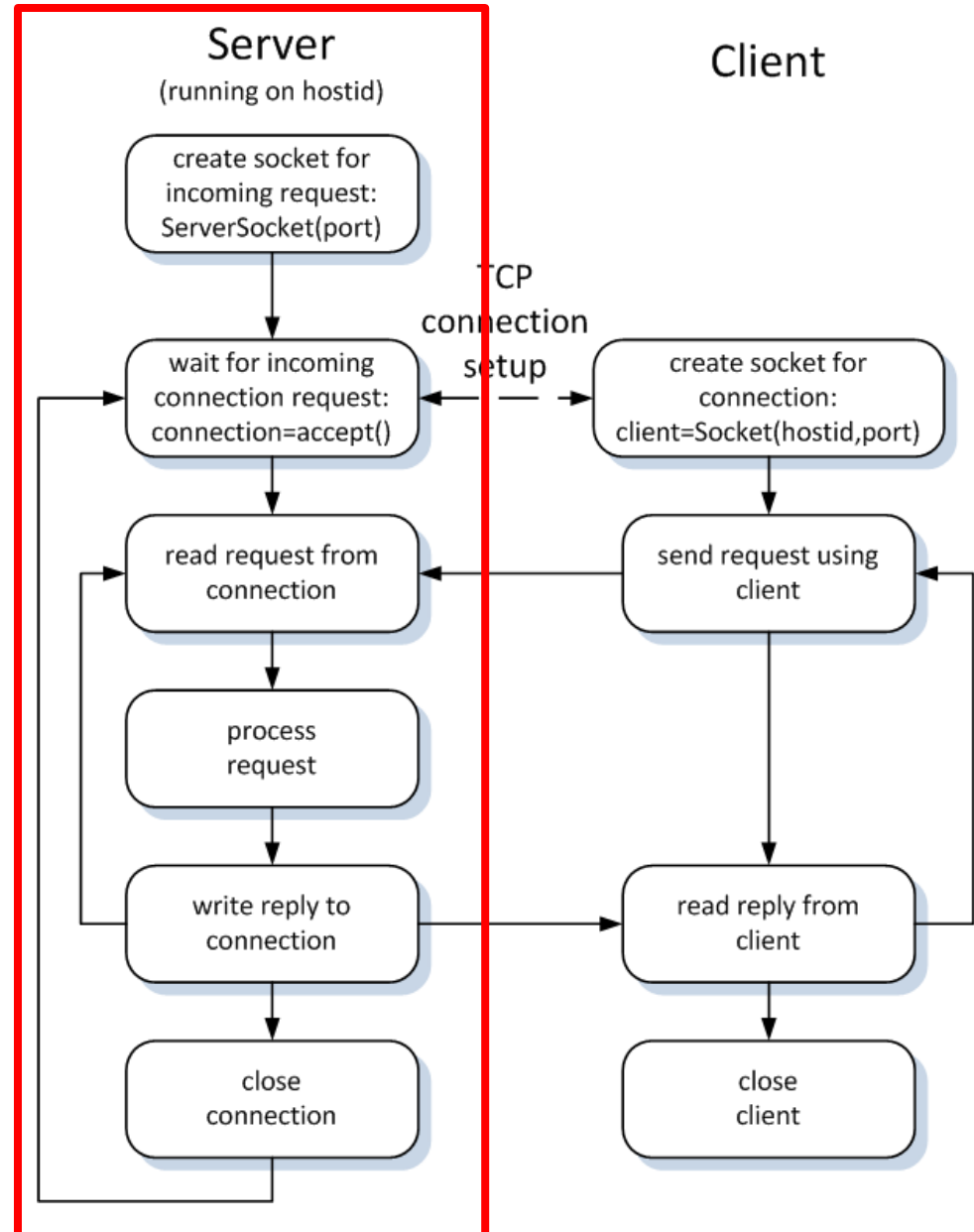


# Socket für den Server

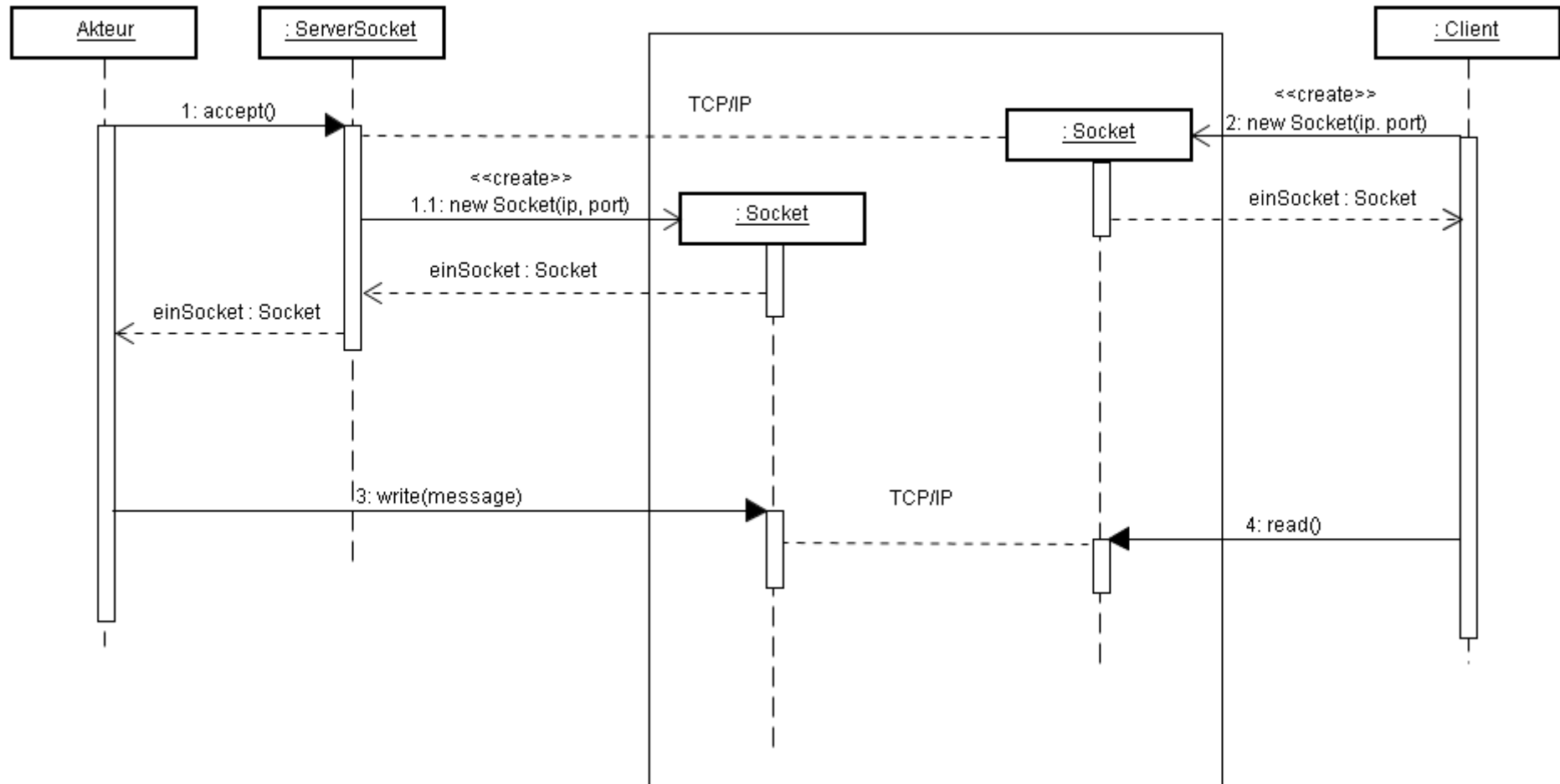
- Server hören an ihrem zugewiesenen Port auf Anfragen.
- Sockets bekommen als Argument Portnummer, zu der sich Clients verbinden können.
- Regeln für die Portnummer:
  - darf nicht in Benutzung sein,
  - kann nach Benutzung z.T. für einen bestimmten Zeitraum nicht verwendet werden (typisch: 4 Minuten)
  - falls  $< 1024$  nur nutzbar durch Rootbenutzer bei Unix-Systemen (Linux, MacOS, etc.)

# Lebenszyklus eines TCP Servers

1. Server-Socket erzeugen.
2. Mit accept-Methode auf Verbindung warten.
3. Ein- und Ausgabestrom mit erhaltenem Socket verknüpfen.
4. Daten lesen und schreiben, entsprechend dem Kommunikationsprotokoll.
5. Stream von Client und Socket schliessen.
6. Bei Schritt 2 weitermachen oder Server-Socket schliessen.



# Sequenzdiagramm TCP mit Sockets



# Einfacher DayTime-Client mit Blocking I/O

```
static void getTime(String host, int port) throws IOException {  
    Socket socket = new Socket(host, port);  
  
    DataInputStream is;  
    is = new DataInputStream(socket.getInputStream());  
  
    byte[] bytes = is.readAllBytes();  
    socket.close();  
  
    String time = new String(bytes);  
    System.out.println(time);  
}
```

Socketverbindung  
starten

Eingabestrom für  
öffnen

Alle Bytes bis Ende  
der Verbindung  
einlesen

Socket schliessen

# Einfacher DayTime-Server (Blocking I/O, Single-Threaded)

```
public class SimpleDayTimeServer {  
    //...  
    public static void main(final String[] args) {  
        try {  
            final ServerSocket listen = new ServerSocket(1300);  
  
            while (true) {  
                try (final Socket client = listen.accept()) {  
                    final DataOutputStream dout =  
                        new DataOutputStream(client.getOutputStream());  
                    final Date date = new Date();  
                    dout.write((date.toString()).getBytes());  
                }  
            }  
        } catch (IOException ex) {  
            LOG.debug(ex.getMessage());  
        }  
    }  
}
```

Server Socket an Port 1300 binden

Warten auf Client

Zeit wird dem Client gesendet.

Try-with-resources wird geschlossen. Verbindung zum Client wird beendet.



# Warten auf Verbindungen

- Nur die accept-Methode der **ServerSocket** Klasse nimmt eine wartende Verbindung an und zwar genau eine Verbindung.
- Die accept-Methode blockiert den Programmablauf.
- Die Kommunikation läuft nicht über den Server-Socket, sondern über den von der accept-Methode zurückgegebenen Socket.
- Zur schnellen Wiederverfügbarkeit (neue Verbindungen) muss das Programm so schnell wie möglich zur accept-Methode zurückkehren.
- Warten auf Verbindungen und die Kommunikation mit dem Client sollte daher nebenläufig ausgeführt werden.

# Nicht-blockierender EchoServer

```
public class EchoServer {  
  
    private static final Logger LOG = LogManager.getLogger(EchoServer.class);  
  
    public static void main(final String[] args) throws IOException {  
        final ServerSocket listen = new ServerSocket(7777);  
        final ExecutorService executor = Executors.newFixedThreadPool(5);  
        while (true) {  
            try {  
                LOG.info("Waiting for connection...");  
                final Socket client = listen.accept();  
                final EchoHandler handler = new EchoHandler(client);  
                executor.execute(handler);  
            } catch (Exception ex) {  
                LOG.debug(ex.getMessage());  
            }  
        }  
    }  
}
```

Executor für das parallele Handling.

Erstellung eines Echo Handlers, der die Kommunikation zum Client übernimmt.

Das Programm kann sofort auf den nächsten Verbindungsaufbau warten.

Der EchoHandler wird in einem eigenen Thread ausgeführt.

# EchoHandler – Erzeugung


- Der EchoServer erstellt ein EchoHandler-Objekt und übergibt den Socket zur Clientverbindung.
- Sobald der Executor den EchoHandler mit einem Thread gestartet hat, läuft der Handler unabhängig von anderen laufenden Threads.
- Der Client bestimmt das Ende des Echo Handlings.

```
public class EchoHandler implements Runnable {
```

```
    private static final Logger LOG =  
        LogManager.getLogger(EchoHandler.class);
```

```
    private final Socket client;
```

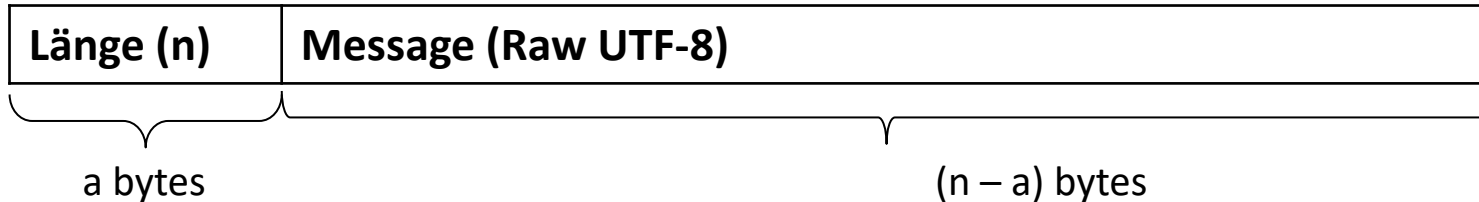
```
    public EchoHandler(final Socket client) {  
        this.client = client;  
    }
```



Übergabe des Client Socket an  
den Echo Handler

# Kommunikationsprotokoll für den EchoServer

## ▪ Message mit Länge



```
private static void sendMessage(DataOutputStream os, String msg) throws IOException {  
    byte[] bytesOut = msg.getBytes(StandardCharsets.UTF_8);  
    os.writeInt(bytesOut.length);  
    os.write(bytesOut);  
}
```

Schreibe Integer (plattformunabhängig)

Schreibe Bytes

```
private static String getMessage(DataInputStream is) throws IOException {  
    int length = is.readInt();  
    byte[] bytesIn = new byte[length];  
    is.readFully(bytesIn);  
  
    return new String(bytesIn, StandardCharsets.UTF_8);  
}
```

Lese Integer (plattformunabhängig)

Lese Bytes

# EchoHandler – Ausführung

```
@Override
public void run() {
    LOG.info("Connection to " + client);
    try (OutputStream out = client.getOutputStream();
        InputStream in = client.getInputStream()) {
        DataInputStream dataIn = new DataInputStream(in);
        DataOutputStream dataOut = new DataOutputStream(out);
        while (true) {
            String message = getMessage(dataIn);
            sendMessage(dataOut, message);
            dataOut.flush();
        }
    } catch (IOException ex) {
        LOG.debug(ex.getMessage());
    }
}
```

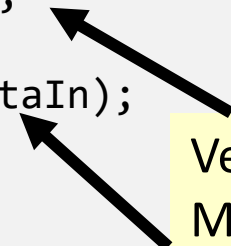
Wartet auf Daten vom Client.

Das Programm wird nicht blockiert, da der EchoHandler in einem eigenen Thread läuft.

Wichtig: flush – damit die Daten den Prozess/Host verlassen.

# EchoClient

```
public static void main(final String[] args) {  
    BufferedReader userIn  
        = new BufferedReader(new InputStreamReader(System.in));  
    try (Socket socket = new Socket("localhost", PORT);  
        OutputStream out = socket.getOutputStream();  
        InputStream in = socket.getInputStream()) {  
  
        DataInputStream dataIn = new DataInputStream(in);  
        DataOutputStream dataOut = new DataOutputStream(out);  
  
        while (true) {  
            String messageOut = userIn.readLine();  
            sendMessage(dataOut, messageOut);  
            dataOut.flush();  
            String messageIn = getMessage(dataIn);  
            System.out.println(messageIn);  
        }  
    } catch (Exception ex) {  
        LOG.debug(ex.getMessage());  
    }  
}
```



Verwendet selbe  
Message-Parsing  
Methoden wie  
Server

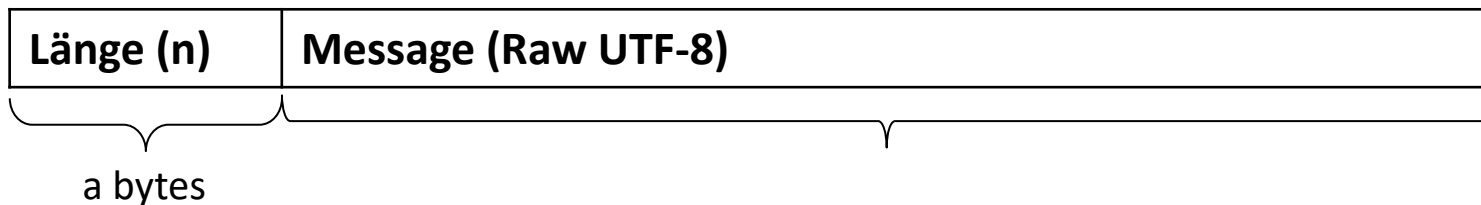
# Klassenraumübung: EchoJava

Übung mittels Online-Programmierungsumgebung:

- <https://replit.com/@mbaettig/EchoJava>
  - **Kein HSLU-Dienst (benötigt separates Konto), erstellen Sie einen Fork.**
- Verwendung:
  - Server starten: Click auf RUN.
  - Client starten: Shell öffnen und eingeben «java EchoClient».

## Aufgabe:

- EchoMessages haben folgende Struktur und Inhalt:



- Message so erweitern, dass noch eine Integer (32-Bit) mitgesendet werden kann (nur Protokoll, API nicht anpassen. Sie können eine Konstante senden).
- Wie muss das Kommunikationsprotokoll angepasst werden?

# Exkurs: Non-Blocking I/O – Selector und Channel

## Ablauf:

- Erstelle einen Selector.
- Öffne einen Channel (ServerSocket oder Socket).
- Setze Sockets auf Non-Blocking und registriere gewünschte Events (z.B. accept).
- Warte auf Events und behandle diese.

## Initialisierung vom Server:

```
selector = Selector.open();  
ServerSocketChannel socket = ServerSocketChannel.open();  
ServerSocket serverSocket = socket.socket();  
serverSocket.bind(new InetSocketAddress("localhost", 7777));  
socket.configureBlocking(false);  
socket.register(selector, SelectionKey.OP_ACCEPT);
```



# Exkurs: Non-Blocking I/O – Multiplexing

- Auf gleichem Selektor verschiedene Channels registrieren.

## Beispiel die Behandlung von Accept:

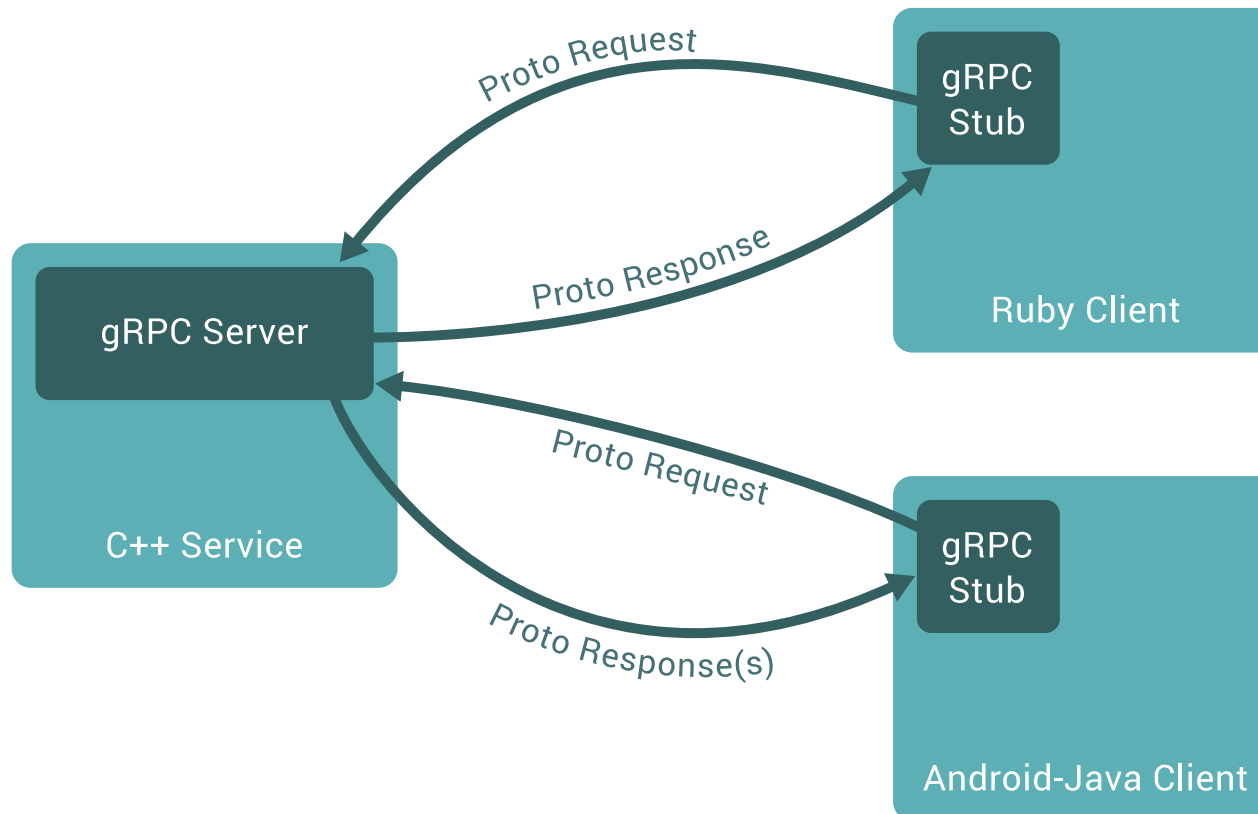
```
SocketChannel client = mySocket.accept();
client.configureBlocking(false);
client.register(selector, SelectionKey.OP_READ, new ClientState());
```

- Anschliessend auf Ereignisse warten («Event-Loop»):

```
while (true) {
    selector.select();
    Set<SelectionKey> selectedKeys = selector.selectedKeys();
    Iterator<SelectionKey> i = selectedKeys.iterator();
    while (i.hasNext()) {
        SelectionKey key = i.next();
        if (key.isAcceptable()) { handleAccept(socket); }
        else if (key.isReadable()) { handleRead(key); }
        else if (key.isWritable()) { handleWrite(key); }
        i.remove();
    }
}
```

# Remote-Procedure-Calls mit gRPC

- Plattform- und sprachübergreifendes RPC-Framework.
  - Java, C#, Python, Java, C++, Go, Node, ...
- Verwendet Google Protocol Buffers



# Definition von Messages mittels Protocol Buffers

- Protocol-Buffers sind Sprach- und Plattform neutral.
- Einsatzzweck: Serialisierung von strukturierten Daten.
- Dateiendung: .proto

```
syntax = "proto3";
```

Version: Falls nicht angegeben wird Version 2 verwendet.

```
message SearchRequest {
```

Definition einer Messagestruktur.

```
  string query = 1;
```

```
  int32 page_number = 2;
```

Typisierte Felder.

```
  int32 result_per_page = 3;
```

Unique Ids für binäres Format.

```
}
```

```
message SearchResult {
```

```
  // weitere Message-Definitionen
```

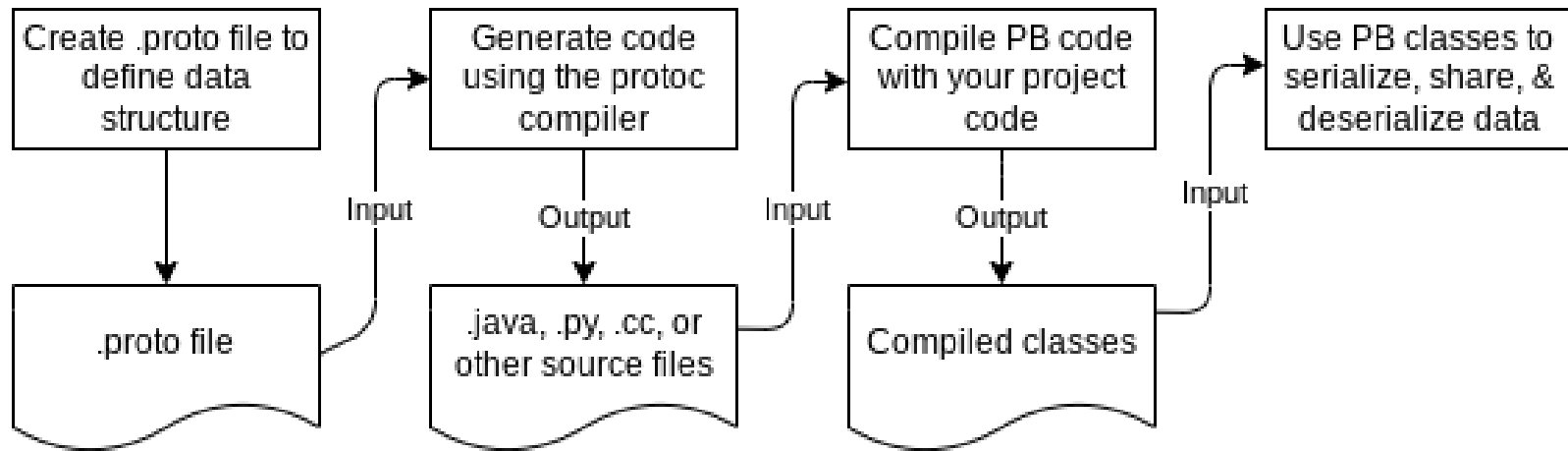
```
}
```

# Type-Mapping: Protocol Buffer zu Zielsprache

.proto Type	C++ Type	Java/Kotlin Type	Python Type	Go Type	C# Type	PHP Type	Dart Type
double	double	double	float	float64	double	float	double
float	float	float	float	float32	float	float	double
int32	int32	int	int	int32	int	integer	int
int64	int64	long	int/long	int64	long	integer/string	Int64
uint32	uint32	int	int/long	uint32	uint	integer	int
uint64	uint64	long	int/long	uint64	ulong	integer/string	Int64
sint32	int32	int	int	int32	int	integer	int
sint64	int64	long	int/long	int64	long	integer/string	Int64
fixed32	uint32	int	int/long	uint32	uint	integer	int
fixed64	uint64	long	int/long	uint64	ulong	integer/string	Int64
sfixed32	int32	int	int	int32	int	integer	int
sfixed64	int64	long	int/long	int64	long	integer/string	Int64
bool	bool	boolean	bool	bool	bool	boolean	bool
string	string	String	str/unicode	string	string	string	String

# Arbeiten mit Protocol Buffers

- Verwendung von `protoc` um Messages in Zielsprache zu generieren:



```
protoc --java_out=DST_DIR path/to/file.proto
```

Für **Java** generiert der `protoc`-Compiler:

- ⇒ eine `.java`-Datei mit einer Klasse pro Messagetyp.
- ⇒ sowie eine speziellen Builder-Klasse zur Erzeugung von Messageinstanzen.

# Service-Definitionen


```
syntax = "proto3";
```

```
service SearchService {  
  rpc Search(SearchRequest) returns (SearchResponse);  
}
```

```
message SearchRequest {  
  ...  
}
```

```
message SearchResponse {  
  ...  
}
```

Service: besteht aus einem oder mehreren RPC.



RPC: Name mit Input-Parameter und Rückgabewert.

# Echo mit gRPC: Service und Nachrichten

- Definition eines Kommunikationsprotokolls mit zwei Nachrichten

```
syntax = "proto3";

service EchoService {
    rpc echo(EchoRequest) returns (EchoResponse);
}

message EchoRequest {
    string message = 1;
}

message EchoResponse {
    string message = 1;
}
```

# Echo mit gRPC: Client

```
public class EchoClient {  
    private static final int PORT = 5001;  
  
    public static void main(String[] args) {  
        ManagedChannel channel =  
            ManagedChannelBuilder.forAddress("localhost", PORT)  
                .usePlaintext()  
                .build();  
  
        EchoServiceGrpc.EchoServiceBlockingStub stub =  
            EchoServiceGrpc.newBlockingStub(channel);  
  
        Echo.EchoResponse echo = stub.echo(Echo.EchoRequest.newBuilder()  
            .setMessage("test").build());  
        System.out.println(echo.getMessage());  
        channel.shutdown();  
    }  
}
```



# Echo mit gRPC: Server

```
public class EchoServer {  
    private static final int PORT = 5001;  
  
    public static class EchoService extends EchoServiceGrpc.EchoServiceImplBase {  
        public void echo(Echo.EchoRequest request,  
                        StreamObserver<Echo.EchoResponse> observer) {  
            Echo.EchoResponse response = Echo.EchoResponse.newBuilder()  
                .setMessage(request.getMessage()).build();  
            observer.onNext(response);  
            observer.onCompleted();  
        }  
    }  
  
    public static void main(String[] args) throws IOException, InterruptedException {  
        Server srv = ServerBuilder.forPort(PORT).addService(new EchoService()).build();  
        srv.start();  
  
        System.out.println("Server started, listening on " + PORT);  
  
        srv.awaitTermination();  
    }  
}
```

onNext: gibt Response zurück

onComplete: RPC ist fertig

Starte Server

Füge Service dem Server hinzu

Warte auf Ende

# Exkurs: Customized-Echo mit gRPC

**Ziel:** Pro Verbindung soll der Client ein Prefix setzen können, welches der Server jeweils der Antwort voranstellt.

**Variante 1:** Streaming gRPC mit genereller Request-Struktur:

```
service CustomizedEcho {  
    rpc myCall(stream Request) returns (stream Response);  
}  
  
message Request {  
    int32 action = 1;    // 01: set prefix | 02:echo  
    string content = 2;  // message or rprefix  
}
```

# Exkurs: Customized-Echo mit gRPC

**Ziel:** Pro Verbindung soll der Client ein Prefix setzen können, welches der Server jeweils der Antwort voranstellt.

**Variante 2:** Mittels einer «Session»

```
service CustomizedEcho {  
  
    // open connection  
    rpc open(SessionRequest) returns (SessionResponse);  
  
    // send new prefix  
    rpc setPrefix(SetPrefixRequest) returns (SetPrefixResponse);  
  
    // request a prefixed echo  
    rpc echo(EchoRequest) returns (EchoResponse);  
  
    // close connection  
    rpc close(CloseRequest) returns (CloseResponse);  
  
}
```

# Zusammenfassung

- Remote-Procedure-Calls sind synchrone entfernte Aufrufe.
- Umsetzung mittels Kommunikationsprotokoll.
  - Definiert Anforderungen an Kanal, generelle Struktur und Nachrichten.
- Viele verschiedene Möglichkeiten Verbindungen parallel zu verarbeiten.
- ServerSocket können Verbindungen akzeptieren (accept).
- Der Server selbst ist nicht aktiv, sondern horcht an seinem zugewiesenen Port und der IP-Adresse auf Client-Anfragen.
- gRPC ist ein modernes Framework für RPC.

**Fragen?**