TU Braunschweig
Institut für Betriebssysteme
und Rechnerverbund

Verteilte Systeme

Prof. Dr. Stefan Fischer

Kapitel 4: Interprozesskommunikation

Übersicht

- Interprozess-Kommunikation
- Direkte Nutzung des Netzwerk-API vs. Middleware
- TCP und UDP Sockets
- TCP und UDP Clients und Server in Java
- Generelle Mechanismen aufsetzend auf dieser Schnittstelle
  - Request-Reply-Protokolle
  - Gruppenkommunikation
  - Kodierung von Daten im Netz

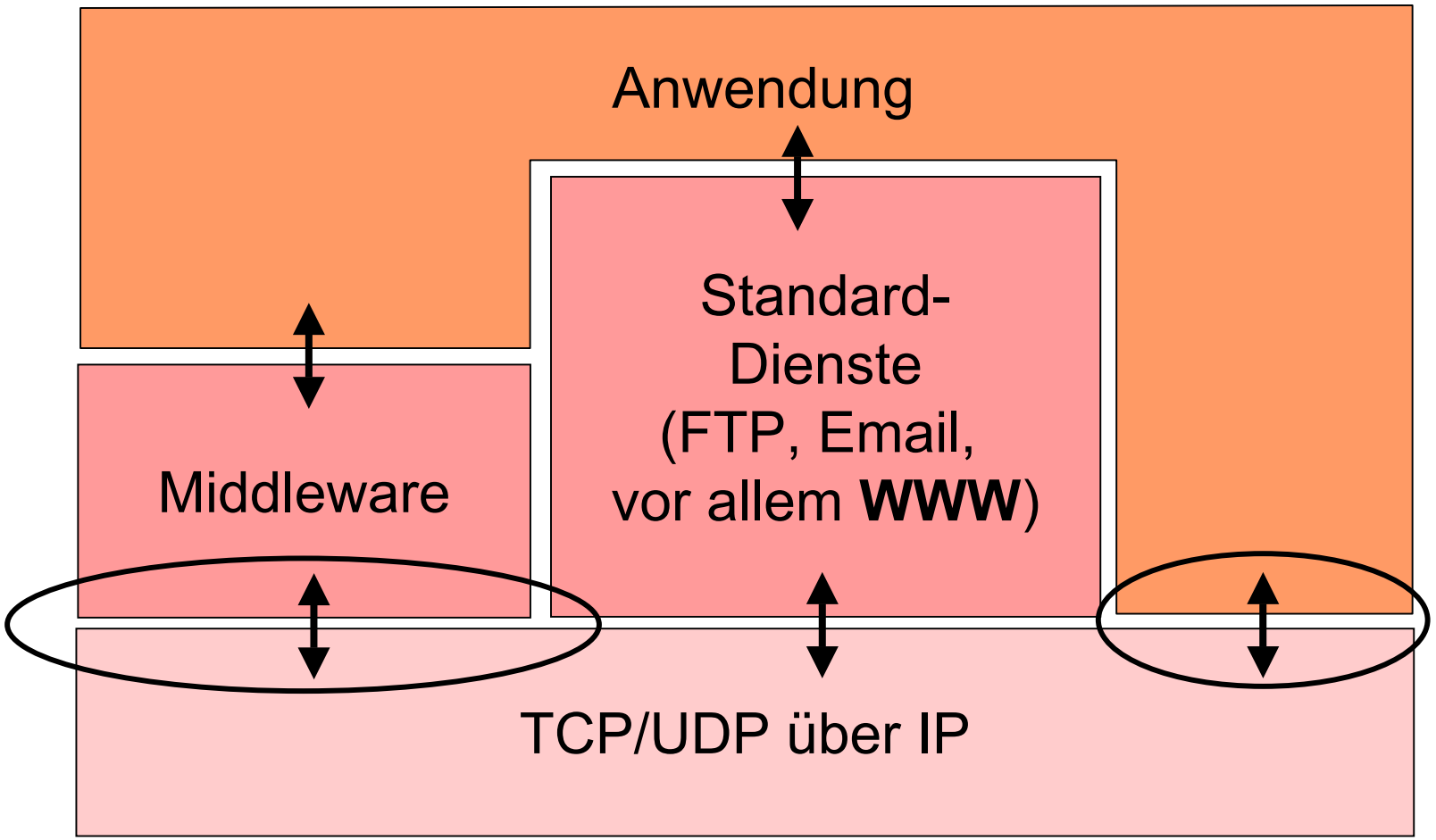
Interprozess-Kommunikation

- Anwendungsprogramme „leben“ in *Prozessen*.
- Ein Prozess ist ein Objekt des Betriebssystems, durch das Anwendungen sicheren Zugriff auf die Ressourcen des Computers erhalten. Einzelne Prozesse sind dazu gegeneinander isoliert.
- Damit zwei Prozesse Informationen austauschen können, müssen sie Interprozesskommunikation (*interprocess communication, IPC) einsetzen*.
- IPC basiert auf dem Austausch von Nachrichten (= Bitfolgen)
  - Eine Nachricht wird von dem einem Prozess geschickt (dem *Sender*).
  - Sie wird von einem anderen Prozess empfangen (dem *Empfänger*).

Synchroner vs. Asynchroner IPC

- Synchron:
  - Sender und Empfänger blockieren beim Senden bzw. Empfangen, d.h., wenn ein „Senden“ ausgeführt wird, kann der Prozess nur weiterarbeiten, nachdem das zugehörige „Empfangen“ im anderen Prozess ausgeführt wurde.
  - Wenn ein „Empfangen“ ausgeführt wird, wartet der Prozess so lange, bis eine Nachricht empfangen wurde.
  - Weniger effizient, aber leicht zu implementieren (Synchronisation beim Ressourcenzugriff wird gleich miterledigt).
- Asynchron:
  - Das „Senden“ findet nicht-blockierend statt, d.h., der Prozess kann nach dem Senden der Nachricht sofort weiterarbeiten.
  - Empfangen kann blockierend oder nicht-blockierend sein.
  - Etwas komplizierter zu implementieren (Warteschlangen), aber effizienter.

Implementierung vert. Anwendungen

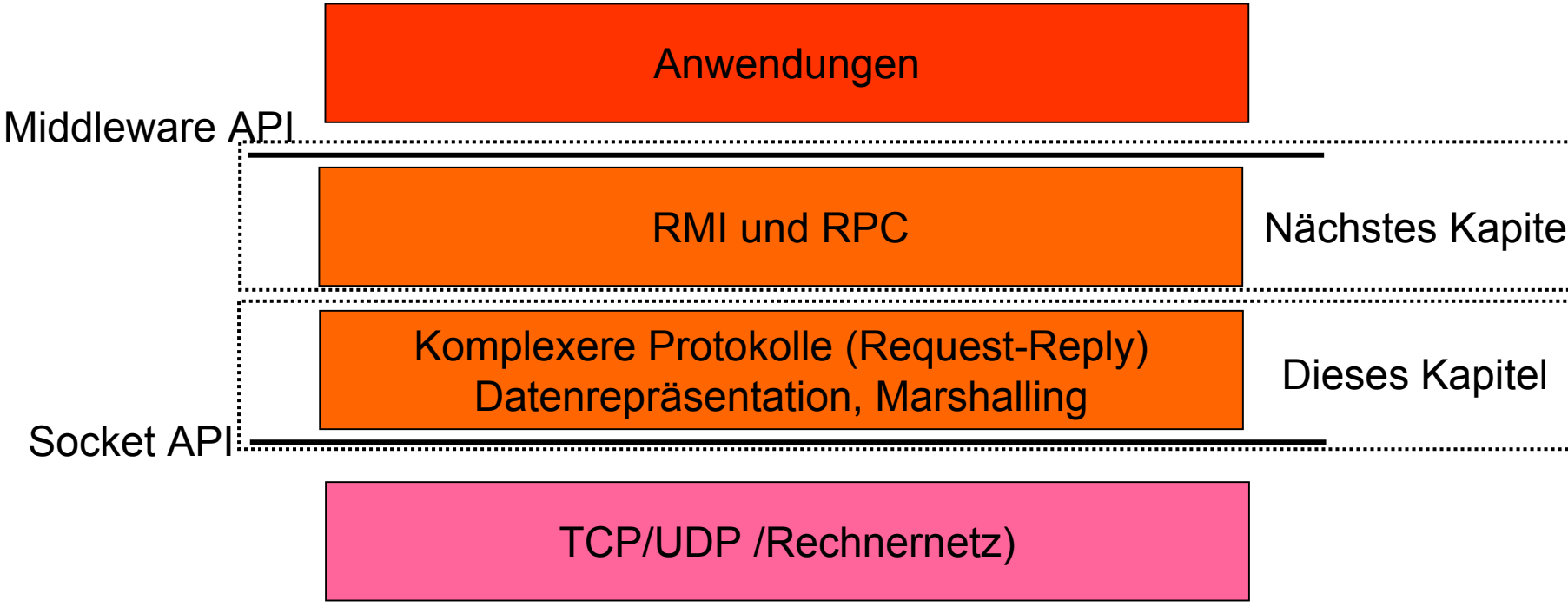


Netzprogrammierung vs. Middleware

- Direkte Netzprogrammierung
  - Direkte Kontrolle aller Transportparameter
  - größere Flexibilität bei der Entwicklung neuer Protokolle
  - Kann in vielen Fällen bessere Performance bringen
  - Grosses Problem: Datenrepräsentation
- Middleware
  - Sehr bequemer Weg zur Entwicklung von Anwendungen
  - Datenrepräsentation, Objektlokalisierung etc. muss nicht von der Anwendung gemacht werden
  - Oft viel Overhead

Netzwerkprogrammierung ist Thema dieses Kapitels.
Middleware betrachten wir im nächsten Kapitel.

Schichten des Kommunikationssystems

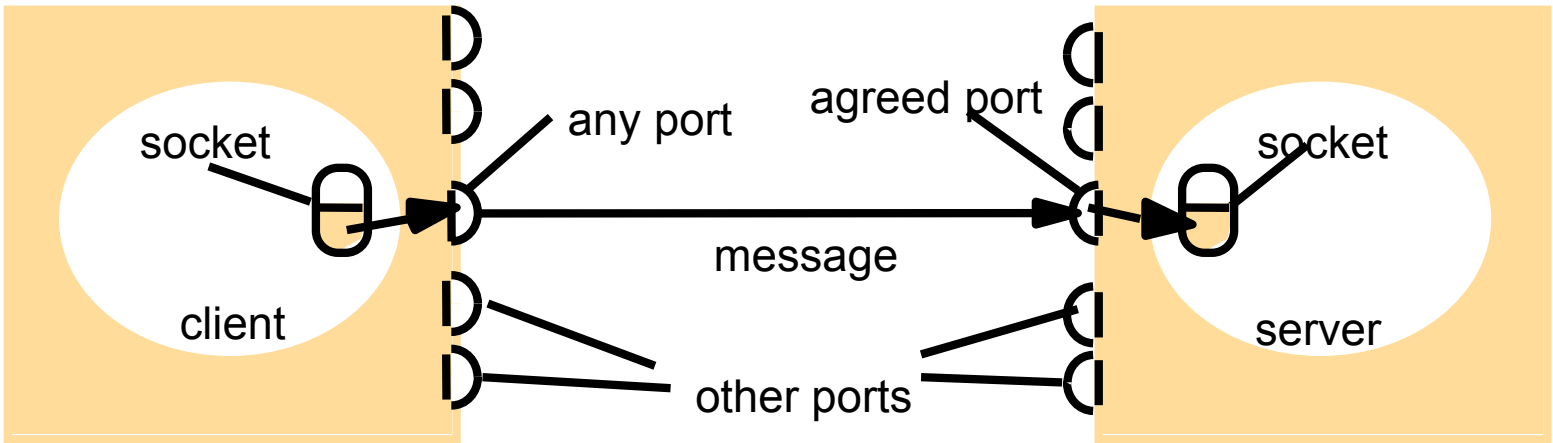


Das TCP/UDP API: Sockets

- Das Internet-API über der Transportschicht wird als *Sockets* bezeichnet.
- Ein Socket kann als Endpunkt einer Kommunikationsbeziehung betrachtet werden.
- Daten werden in Sockets abgelegt und durch Sockets empfangen.
- Es gibt in der Socket-Schnittstelle zwei grundlegende Typen von Sockets:
  - Ein Socket, der wie das Ende einer Telefonverbindung funktioniert
  - Ein Socket, der wie ein Briefkasten funktioniert
  - Was bedeutet das?
- Sockets sind mehr oder weniger in jeder Programmiersprache erhältlich, u.a. in Java.

Sockets und TCP/UDP-Ports

Ein Socket wird vor der Kommunikation mit einer TCP/UDP-Portnummer und einer IP-Adresse assoziiert.
Dadurch kann ein bestimmter Prozess auf einem Rechner identifiziert werden.



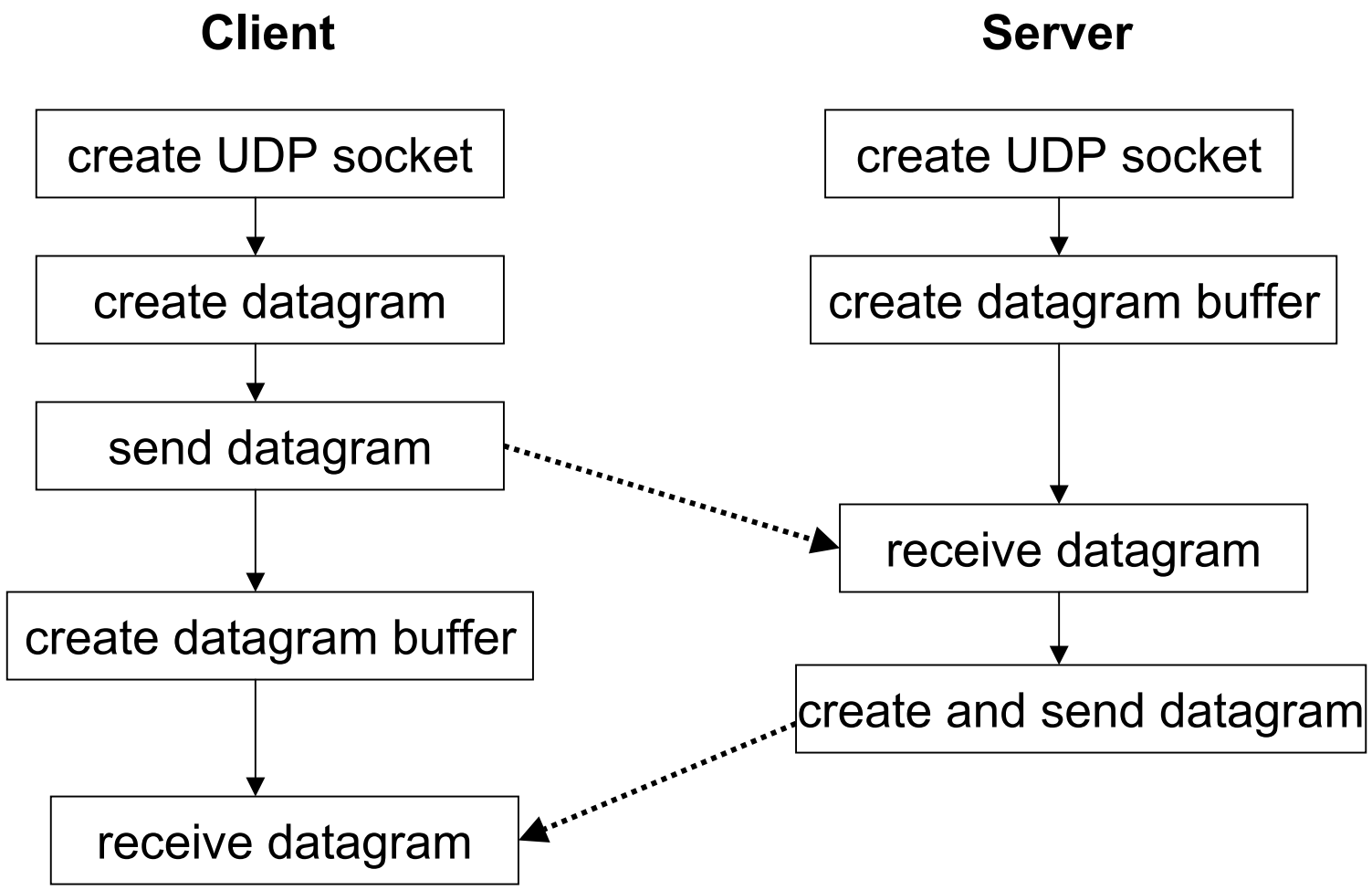
UDP oder „Datagram“ Sockets

- Eine Nachricht, die durch einen UDP-Socket geschickt wird, wird nicht bestätigt bzw. bei Verlust automatisch erneut geschickt.
- Paketfehler können deshalb zum Verlust der Nachricht führen, ohne dass es die Anwendung bemerkt.
- Maximale Grösse der Nachricht: 8 KByte; grössere Nachrichten müssen in der Anwendung segmentiert bzw. re-assembliert werden!
- UDP-Sockets verwenden nicht-blockierendes Senden und blockierendes Empfangen.

Java API für UDP-Sockets

- Zwei wichtige Klassen:
  - `DatagramPacket`
  - `DatagramSocket`
- `DatagramPacket` enthält die zu sendende Information.
- `DatagramSocket` besitzt vor allem die Methoden
  - `send(DatagramPacket)`
  - `receive(DatagramPacket)`
  - mit der offensichtlichen Funktionalität.
- Mehr Informationen finden sich in der API-Beschreibung unter http://java.sun.com/j2se/1.4.1/docs/api/ in der `net`-Package.

Typische Struktur von UDP-Programmen



Ein UDP-Client

```
import java.net.*;
import java.io.*;
public class UDPClient{
   public static void main(String args[]){
            try {
                    DatagramSocket aSocket = new DatagramSocket();
                    byte [] m = args[0].getBytes();
                    InetAddress aHost = InetAddress.getByName(args[1]);
                    int serverPort = 6789;
                    DatagramPacket request = new DatagramPacket(m,  args[0].length(),
                                                                aHost, serverPort);
                    aSocket.send(request);
                    byte[] buffer = new byte[1000];
                    DatagramPacket reply = new DatagramPacket(buffer, buffer.length);

                    aSocket.receive(reply);
                    System.out.println("Reply: " + new String(reply.getData()));
                    aSocket.close();
          }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
          }catch (IOException e){System.out.println("IO: " + e.getMessage());}
   }
}
```

Ein UDP-Server

```
import java.net.*;
import java.io.*;
public class UDPServer{
        public static void main(String args[]){
            try{
                    DatagramSocket aSocket = new DatagramSocket(6789);
                    byte[] buffer = new byte[1000];
                    while(true){
                        DatagramPacket request = new DatagramPacket(buffer, buffer.length);
                        aSocket.receive(request);
                        DatagramPacket reply = new DatagramPacket(request.getData(),
                                request.getLength(), request.getAddress(), request.getPort());
                        aSocket.send(reply);
                    }
            }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
            }catch (IOException e) {System.out.println("IO: " + e.getMessage());}
    }
}
```

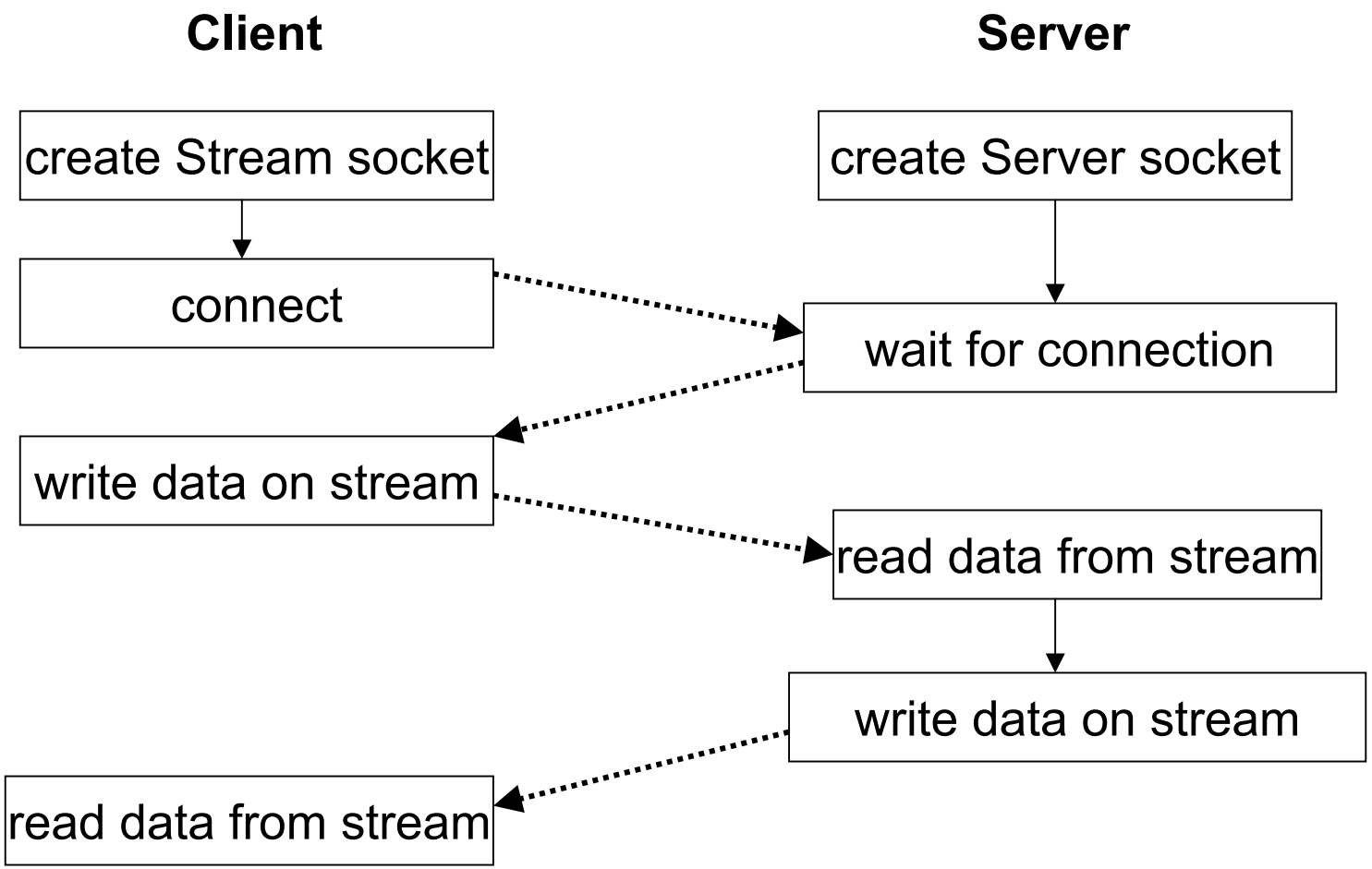
TCP oder „Stream“ Sockets

- TCP-Sockets gestatten eine Datenstrom-orientierte Kommunikation.
- Daten werden in einem Strom geschrieben, der vom Partner in derselben Reihenfolge empfangen wird.
- Paketverluste werden von TCP abgefangen, d.h., Anwendungen erhalten Daten erst, wenn Sie wirklich korrekt (in der Reihenfolge) sind.
- Vor dem eigentlichen Datenaustausch muss zunächst eine Verbindung zwischen Client und Server aufgebaut werden.
- Verbindungen werden über die Port/IP-Adressinformation identifiziert.

Das TCP-Socket-API in Java

- Zwei wichtige Klassen:
  - `ServerSocket`
  - `Socket`
- Ein `ServerSocket` ist passiv und wartet nach dem Aufruf der `accept`-Methode auf Verbindungsaufbauwünsche von Clients.
- Ein `Socket` wird vom Client genutzt. Mittels des Konstruktors wird implizit eine Verbindung zu einem Server aufgebaut.
- Bei beiden Klassen kann man sich eine Referenz auf den *Ein- bzw. Augabedatenstrom* geben lassen. Datenströme können dann nach dem ganz normalen Java-Schema genutzt werden.

Typische Struktur von TCP-Programmen



Ein TCP-Client

```
import java.net.*;
import java.io.*;
public class TCPClient {
        public static void main (String args[]) {
        // arguments supply message and hostname of destination
          try{
                int serverPort = 7896;
                Socket s = new Socket(args[1], serverPort);
                DataInputStream in = new DataInputStream( s.getInputStream());
                DataOutputStream out =
                        new DataOutputStream( s.getOutputStream());
                out.writeUTF(args[0]);          // UTF is a string encoding see Sn 4.3
                String data = in.readUTF();
                System.out.println("Received: "+ data) ;
                s.close();
          }catch (UnknownHostException e){
                        System.out.println("Sock:"+e.getMessage());
          }catch (EOFException e){System.out.println("EOF:"+e.getMessage());
          }catch (IOException e){System.out.println("IO:"+e.getMessage());}
        }
}
```

Ein TCP-Server

```
import java.net.*;
import java.io.*;
public class TCPServer {
   public static void main (String args[]) {
            try{
                        int serverPort = 7896;
                        ServerSocket listenSocket = new ServerSocket(serverPort);
                        while(true) {
                                    Socket clientSocket = listenSocket.accept();
                                    Connection c = new Connection(clientSocket);
                        }
            } catch(IOException e) {System.out.println("Listen :"+e.getMessage());}
   }
}

// this figure continues on the next slide
```

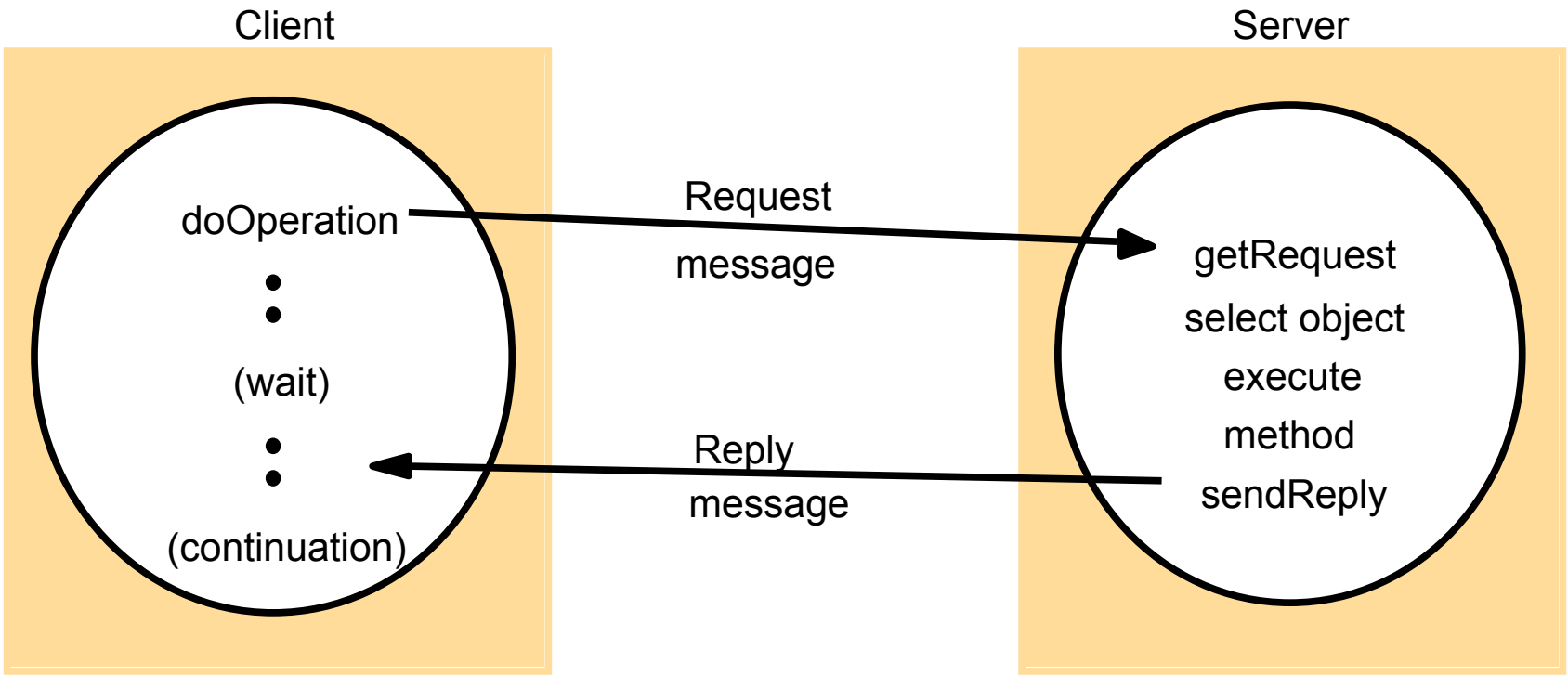
TCP Server (Forts.)

```
class Connection extends Thread {
        DataInputStream in;
        DataOutputStream out;
        Socket clientSocket;
        public Connection (Socket aClientSocket) {
          try {
                clientSocket = aClientSocket;
                in = new DataInputStream( clientSocket.getInputStream());
                out =new DataOutputStream( clientSocket.getOutputStream());
                this.start();
           } catch(IOException e)  {System.out.println("Connection:"+e.getMessage());}
        }
        public void run(){
          try {                                          // an echo server
                String data = in.readUTF();
                out.writeUTF(data);
                clientSocket.close();
          } catch(EOFException e) {System.out.println("EOF:"+e.getMessage());
          } catch(IOException e) {System.out.println("IO:"+e.getMessage());}
        }
}
```

Weitere Aufgaben des Programmierers

- Sockets stellen nur die grundlegenden Mechanismen zur Verfügung, es bleibt noch einiges zu tun:
  - Implementierung komplexerer Systemmodelle wie Request-Reply (bei Client-Server) oder Gruppenkommunikation
  - Vor allem aber die Notwendigkeit der homogenen Datenrepräsentation in heterogenen Umgebungen
- Dies sind die grundlegenden Techniken für komplexere Middleware wie
  - RPC
  - Java RMI
  - CORBA

Request-Reply-Kommunikation



Operationen des Request-Reply-Protokolls

public byte[] doOperation (RemoteObjectRef o, int methodId, byte[] arguments)

sends a request message to the remote object and returns the reply.
The arguments specify the remote object, the method to be invoked and the arguments of that method.

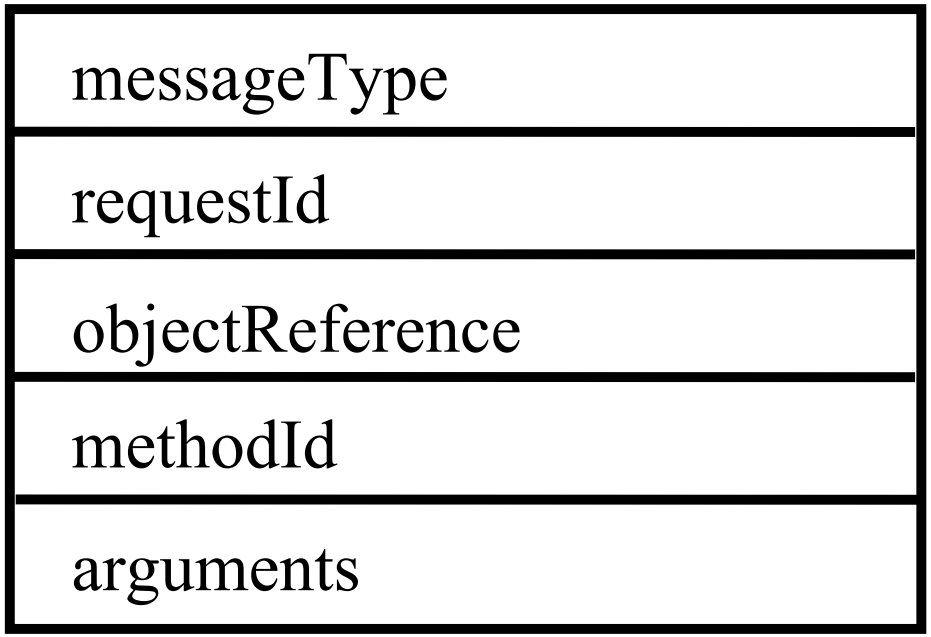
public byte[] getRequest ();

acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

sends the reply message reply to the client at its Internet address and port.

Nachrichtenstruktur bei Request-Reply



int (0=Request, 1= Reply)

int

RemoteObjectRef

int or Method

array of bytes

HTTP: ein typisches Request-Reply-Protokoll

- HTTP ist ein text-orientiertes Protokoll, das auf TCP-Sockets basiert.
- Wenn Sie auf einen Link in Ihrem Browser klicken,
  - wird zunächst eine TCP-Verbindung zum entsprechenden Web-Server aufgebaut und
  - dann eine HTTP-Request-Nachricht generiert und an den Server geschickt.
  - Der Server liest die Nachricht, führt die entsprechenden Arbeiten aus und schickt eine HTTP-Reply-Nachricht an den Cleint, typischerweise inclusive einer HTML-Datei.
  - Der Client (Browser) liest die Nachricht und stellt die Datei im Browserfenster dar.

HTTP-Nachrichtenformate

Request Message

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	//www.dcs.qmw.ac.uk/index.html	HTTP/ 1.1		

Reply Message

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

Multicast-Kommunikation

- Bei modernen Multimediaanwendungen wie Videokonferenzen oder gemeinsamem Editieren von Dokumenten wird eine effiziente Gruppenkommunikation immer wichtiger.
- Gruppenkommunikation zwischen n Mitgliedern könnte durch den Aufbau von (n-1) 1-zu-1-Verbindungen realisiert werden. Das ist sehr ineffizient – warum?
- Die Standardlösung heißt **Multicast** und stellt eine 1-zu-n-Kommunikation zur Verfügung:
  - Die Anwendung muss nur eine Verbindung pro Gruppe verwalten.
  - Ressourcen im Netz werden effizienter genutzt.

IP Multicast

- Bereits seit Anfang der 90er Jahre gibt es Vorschläge für eine Multicast-Lösung im Internet (S. Deering).
- IP Multicast verwendet eine spezielle Form von IP-Adressen, um Gruppen zu identifizieren (eine Klasse-D-Adresse).
- Wenn eine Nachricht an diese Adresse geschickt wird, erhalten sie alle Mitglieder, die ihre Zugehörigkeit zu der dazugehörigen Gruppe erklärt haben.
- Java stellt ein API für IP multicast zur Verfügung. Es erlaubt
  - Beitreten (join) und Verlassen (leave) der Gruppe
  - Senden von Paketen an die Gruppe
  - Empfang von Paketen, die an die Gruppe geschickt wurden

Beispiel für Java Multicast

```
import java.net.*;
import java.io.*;
public class MulticastPeer{
        public static void main(String args[]){
         // args give message contents & destination multicast group (e.g. "228.5.6.7")
           try {
                InetAddress group = InetAddress.getByName(args[1]);
                MulticastSocket s = new MulticastSocket(6789);
                s.joinGroup(group);
                byte [] m = args[0].getBytes();
                DatagramPacket messageOut =
                        new DatagramPacket(m, m.length, group, 6789);
                s.send(messageOut);



         // this figure continued on the next slide
```

Java Multicast (Forts.)

```
        // get messages from others in group
                byte[] buffer = new byte[1000];
                for(int i=0; i< 3; i++) {
                    DatagramPacket messageIn =
                            new DatagramPacket(buffer, buffer.length);
                    s.receive(messageIn);
                    System.out.println("Received:" + new String(messageIn.getData()));
                }
                s.leaveGroup(group);
        }catch (SocketException e){System.out.println("Socket: " + e.getMessage());
        }catch (IOException e){System.out.println("IO: " + e.getMessage());}
    }
}
```

Datenrepräsentation und Marshalling

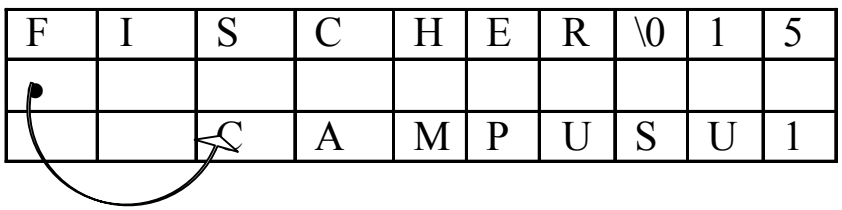
- Die meisten Anwendungen bzw. Middleware-Ansätze nutzen ein gemeinsames Datenformat.
- Notwendig wegen der Heterogenität der Umgebungen
  - Unterschiedliche Hardwarearchitektur
  - Verschiedene Betriebssysteme
  - Verschiedene Programmiersprachen
- Unter „Marshalling“ versteht man den Prozess der Transformation einer beliebigen Datenstruktur in eine übertragbare Nachricht:
  - „planieren“ der Datenstruktur (in eine zusammenhängende Nachricht)
  - Übersetzung in das gemeinsame Format

Abbildung von Datenstrukturen auf Nachrichten

- Ein Nachricht steht zusammenhängend im Speicher und kann so übertragen werden.

F	I	S	C	H	E	R	\0	1	5	C	A	M	P	U	S	U	1	

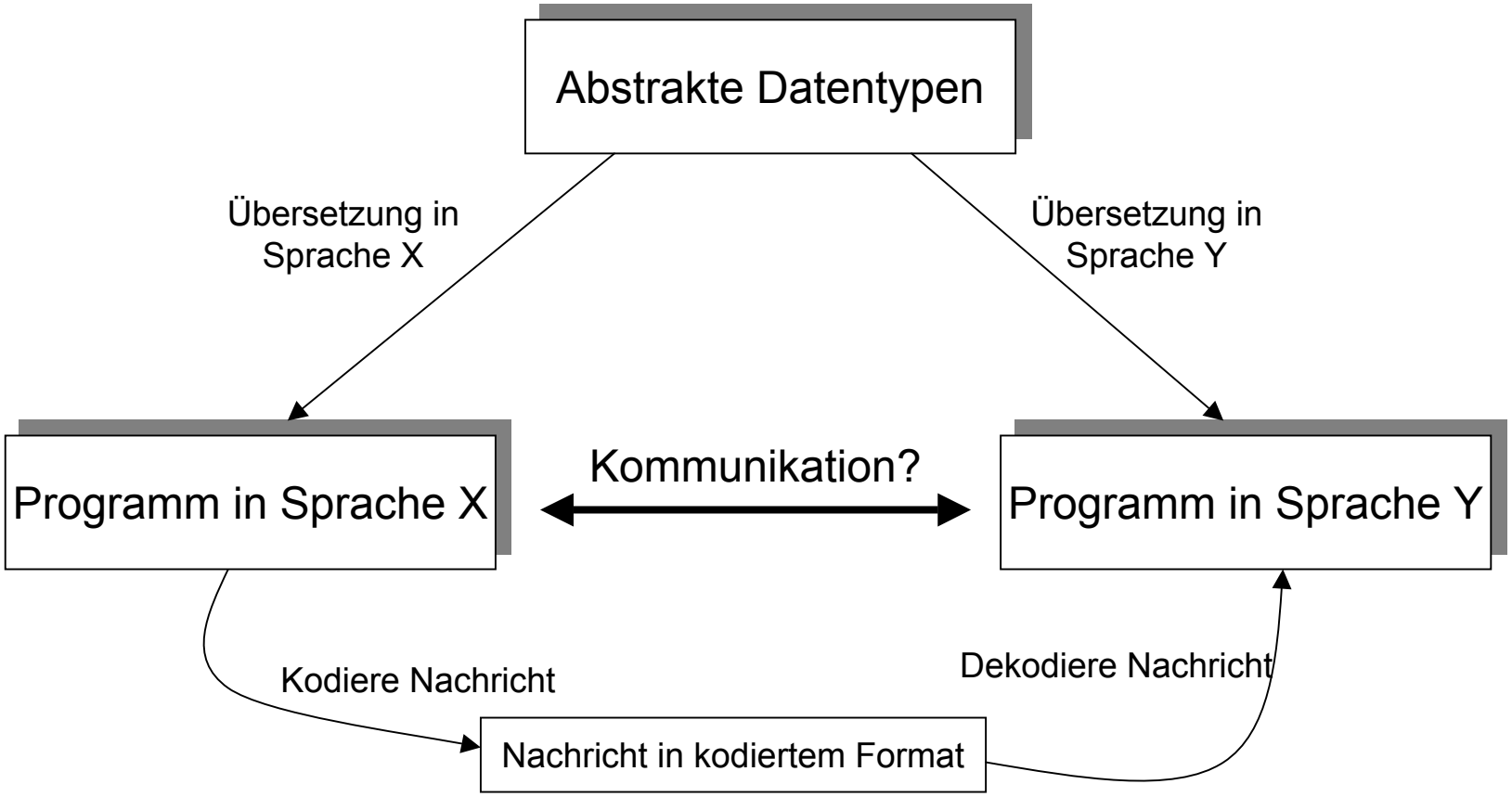
- Eine Datenstruktur kann über den Speicher verteilt sein und so nicht übertragen werden.



Datenrepräsentation

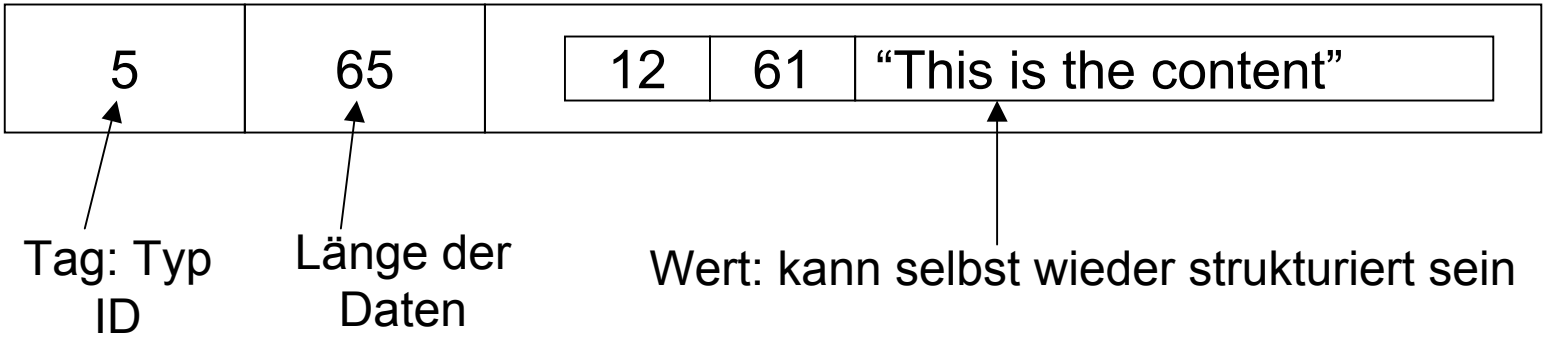
- Es gibt eine Reihe bekannter Ansätze für ein gemeinsames Netzdatenformat.
- Idee:
  - Definiere eine Menge von abstrakten Datentypen und eine Kodierung (ein genaues Bit-Format) für jeden dieser Typen
  - Stelle Werkzeuge zur Verfügung, die die abstrakten Datentypen in Datentypen der verwendeten Programmiersprache übersetzen
  - Stelle Prozeduren zur Verfügung, die die lokalen Darstellungen der Datentypen in das kodierte Format übersetzen
  - Zur Laufzeit: wenn ein bestimmter Datentyp übertragen werden soll, rufe die Kodierfunktion auf und übertarge das Ergebnis
  - Auf der Empfängerseite: dekodiere den Bit-String und erzeuge eine neue lokale Repräsentation des empfangenen Typs

Abbiildungsfunktionen



Bekannte Formate

- ASN.1 (ISO OSI)
- XDR (Internet RPC)
- CDR (CORBA)
- Java object serialization
- XML
- Zwei unterschiedliche Ansätze:
  - Übertragung in binärer Form
  - Übertragung als ASCII
- Beispiel ASN.1:



Die Realität

- Zuerst die schlechte Nachricht: das sieht alles ziemlich kompliziert aus, und es ist es auch. Als Socket-Programmierer muss man sich um diese Dinge selbst kümmern.

- Die gute Nachricht: die Aufgabe einer Middleware ist es, genau diese komplizierten Mechanismen automatisch zu erledigen. Der Anwendungsprogrammierer sieht davon nichts mehr.

- Mehr dazu im nächsten Kapitel.

Weitere Literatur

- Douglas Comer et al.: Internetworking with TCP/IP, 3. Band, Prentice Hall, 2000.
- Dick Steflik, Prashant Sridharan: *Advanced Java Networking*, 2nd ed., Prentice Hall, 2000
- Stefan Fischer, Walter Müller: *Netzwerkprogrammierung mit Linux und Unix*, 2. Auflage, Carl Hanser Verlag, 1999.