Script **generated by TTT**

Title: Eingebettete_Systeme (13.11.2012)

Date: Tue Nov 13 16:11:56 CET 2012

Duration: 62:49 min

Pages: 24

Nachrichtenbasierte Kommunikation

Bei nachrichtenbasierter Prozessinteraktion tauschen Prozesse gezielt Informationen durch Verschicken und Empfangen von Nachrichten aus; ein Kommunikationssystem unterstützt an der Schnittstelle wenigstens die Funktionen `send` und `receive` .

Bisher bestand zwischen Sender und Empfänger eine feste Beziehung, die über Prozessidentifikatoren (z.B. Namen oder Nummer) hergestellt wurde. Nachteile

- Prozessnummern ändern sich mit jedem Neustart.
- Prozessnamen sind nicht eindeutig, z.B. falls Programm mehrmals gestartet wurde.

⇒ Deshalb Senden von Nachrichten an ***Ports*** . Sie stellen Endpunkte einer Kommunikation dar. Sie können bei Bedarf dynamisch eingerichtet und gelöscht werden. Dazu existieren folgende Funktionen:

```
portID = createPort();
deletePort(portID);
send(E.portID, message);
receive(portID, message);
```

- ein Port ist mit dem Adressraum des zugehörigen Prozesses verbunden.
- der Empfängerprozess kann sender-spezifische Ports einrichten.
- ein Rechner mit einer IP-Adresse unterstützt mehrere tausend Ports.
- der Name des Ports ist für einen Rechner eindeutig.
- die Portnummern 1 - 1023 sind fest reserviert für bestimmte Protokolle (bzw. deren Applikationen).

Übersicht: fest zugeordnete Ports

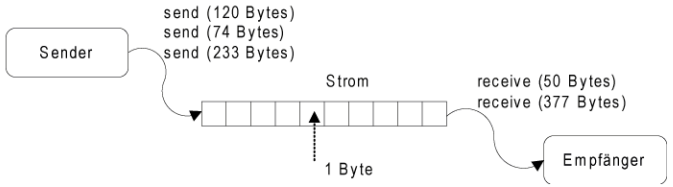
Übersicht: fest zugeordnete Ports

Protokoll	*Port*	*Beschreibung*
FTP	21	Kommandos für Dateitransfer (get, put)
Telnet	23	interaktive Terminal-Sitzung mit entferntem Rechner
SMTP	25	Senden von Email zwischen Rechnern
time	37	Time-Server liefert aktuelle Zeit
finger	79	liefert Informationen über einen Benutzer
HTTP	80	Protokoll des World Wide Web
POP3	110	Zugang zu Email durch einen sporadisch verbundenen Client
RMI	1099	Zugang zum Registrieren von entfernten Java Objekten.

Nachrichtenbasierte Kommunikation

Bei nachrichtenbasierter Prozessinteraktion tauschen Prozesse gezielt Informationen durch Verschicken und Empfangen von Nachrichten aus; ein Kommunikationssystem unterstützt an der Schnittstelle wenigstens die Funktionen `send` und `receive`.

Ströme (engl. streams) sind eine Abstraktion von Kanälen. Sie verdecken die tatsächlichen Nachrichtengrenzen.



BS-Dienste: Verbindungsauf- und -abbau, schreiben in Strom, lesen aus Strom.

Dienste für Dateizugriffe oder Zugriffe auf Geräte: spezielle Ausprägung der stromorientierten Kommunikation.

I/O in Java basiert auf Ströme.

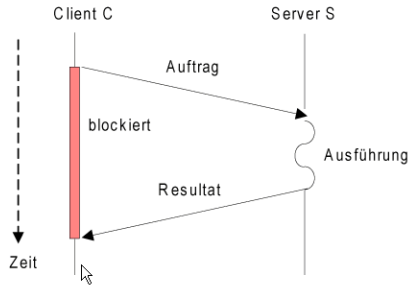
Klasse `java.io.OutputStream` zum Schreiben von Daten

Klasse `java.io.InputStream` zum Lesen von Daten

Spezialisierungen z.B. durch FileOutputStream, BufferedOutputStream oder FileInputStream.

Client-Server Modell basiert i.a. auf der Kommunikationsklasse der synchronen Aufträge. ***Server*** stellen Dienste zur Verfügung, die von vorher unbekannten ***Clients*** in Anspruch genommen werden können.

Client-Server Architektur



Beispiele für Dienste (Services), die mittels Server realisiert werden: Dateidienst, Zeitdienst, Namensdienst.

Definition: Client

Ein Client ist eine Anwendung, die auf einer Clientmaschine läuft und i.a. einen Auftrag initiiert, und die den geforderten Dienst von einem Server erhält.

Clients sind meist a-priori nicht bekannt.

Definition: Server

Ein Server ist ein Subsystem, das auf einer Servermaschine läuft und einen bestimmten Dienst für a-priori unbekannte Clients zur Verfügung stellt.

Server sind dedizierte Prozesse, die kontinuierlich folgende Schleife abarbeiten.

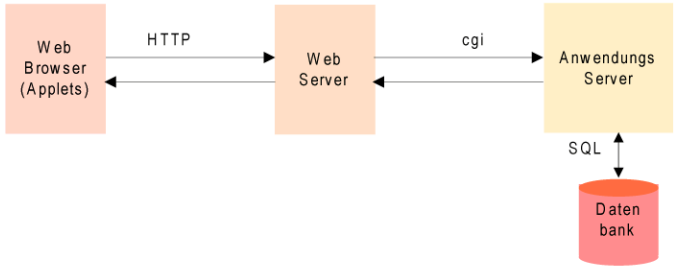
```
while (true) {
   receive (empfangsport, auftrag);
   führe Auftrag aus und erzeuge Antwortnachricht;
   bestimme sendeport für Antwortnachricht;
   send (sendeport, resultat);
}
```

Client und Server kommunizieren über Nachrichten, wobei beide Systeme auf unterschiedlichen Rechnern ablaufen können und über ein Netz miteinander verbunden sind.

Multi-Tier

ein System kann sowohl Client als auch Server sein.

Beispiel



Eine ***verteilte Anwendung*** ist eine Anwendung A, dessen Funktionalität in eine Menge von kooperierenden Teilkomponenten A_1 , .., A_n , $n \in IN$, $n > 1$ zerlegt ist;

> Jede Teilkomponente umfasst Daten (interner Zustand) und Operationen, die auf den internen Zustand angewendet werden.
>
> Teilkomponenten A_i sind autonome Prozesse, die auf verschiedenen Rechensystemen ausgeführt werden können. Mehrere Teilkomponenten können demselben Rechensystem zugeordnet werden.
>
> Teilkomponenten A_i tauschen über das Netz untereinander Informationen aus.

Die Teilkomponenten können z.B. auf der Basis des Client-Server Modells realisiert werden.

Einführung

Server Protokoll

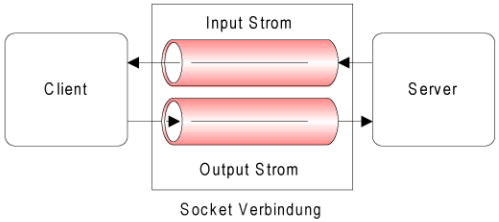
Client Protokoll

Bidirektionale Stromverbindung

Java Socket Class

Einführung

Ein ***Socket*** definiert einen einfachen, bidirektionalen Kommunikationskanal zwischen 2 Rechensystemen, mit Hilfe dessen 2 Prozesse über ein Netz miteinander kommunizieren können.



Socket Grundlagen

Sockets abstrahieren von den technischen Details eines Netzes, z.B. Übertragungsmedium, Paketgröße, Paketwiederholung bei Fehlern, Netzadressen.

Ein Socket kombiniert 2 Ströme, einen Input- und einen Output-Strom.

Ein Socket unterstützt die folgenden Basisoperationen:

- richte Verbindung zu entferntem Rechner ein ("connect").
- sende Daten.
- empfange Daten.
- schließe Verbindung.
- assoziiere Socket mit einem Port.
- warte auf eintreffende Daten ("listen").
- akzeptiere Verbindungswünsche von entfernten Rechnern (bzgl. assoziiertem Port).

Ein Server kommuniziert mit einer Menge von Clients, die a priori nicht bekannt sind. Ein Server benötigt eine Komponente (z.B. ein Verteiler-Thread), die auf eintreffende Verbindungswünsche reagiert.

Informeller Ablauf aus Serversicht

1. Erzeugen eines SocketServer und Binden an einen bestimmten Port.
2. Warten auf Verbindungswünsche von Clients.
3. Austausch von Daten zwischen Client und Server entsprechend einem wohldefinierten Protokoll (z.B. HTTP).
4. Schließen einer Verbindung (durch Server, durch Client oder durch beide); weiter bei Schritt 2.

Programmstück

```
Socket socket; //reference to socket
ServerSocket port; //the port the server listens to
try {
    port = new ServerSocket(10001, ...);
    socket = port.accept(); //wait for client call
    // communicate with client
    socket.close()
}
catch (IOException e) {e.printStackTrace();}
```

Für das Abhören des Ports kann ein eigener Verteiler-Thread spezifiziert werden; die Bearbeitung übernehmen sogenannte Worker-Threads.

Client Protokoll

Der Client initiiert eine Socket-Verbindung durch Senden eines Verbindungswunsches an den Port des Servers.

Informeller Ablauf aus Clientsicht

1. Erzeugen einer Socket Verbindung.
2. Austausch von Daten zwischen Client und Server über die Duplex-Verbindung entsprechend einem wohldefinierten Protokoll (z.B. HTTP).
3. Schließen einer Verbindung (durch Server, durch Client oder durch beide).

Programmstück

```
Socket connection; //reference to socket
try {
    connection = new Socket("www11.in.tum.de", 10001);
    ........ // communicate with client
    connection.close()
}
catch (IOException e) {e.printStackTrace();}
```


Bidirektionale Stromverbindung

Sockets bestehen aus 2 Strömen für die Duplexverbindung zwischen Client und Server.

Schreiben auf Socket

```
void writeToSocket(Socket sock, String str) throws IOException {
    oStream = sock.getOutputStream();
    for (int k = 0; k < str.length(); k++)
       oStream.write(str.charAt(k));
}
```

Lesen von Socket

```
String readFromSocket(Socket sock) throws IOException {
    iStream = sock.getInputStream();
    String str = "";
    char c;
    while ( (c = (char) iStream.read()) != '\n')
       str = str + c;
    return str;
}
```


Client Protokoll

Der Client initiiert eine Socket-Verbindung durch Senden eines Verbindungswunsches an den Port des Servers.

Informeller Ablauf aus Clientsicht

1. Erzeugen einer Socket Verbindung.
2. Austausch von Daten zwischen Client und Server über die Duplex-Verbindung entsprechend einem wohldefinierten Protokoll (z.B. HTTP).
3. Schließen einer Verbindung (durch Server, durch Client oder durch beide).

Programmstück

```
Socket connection; //reference to socket
try {
    connection = new Socket("www11.in.tum.de", 10001);
    ........ // communicate with client
    connection.close()
}
catch (IOException e) {e.printStackTrace();}
```

Generated by Targeteam

Server Protokoll

Ein Server kommuniziert mit einer Menge von Clients, die a priori nicht bekannt sind. Ein Server benötigt eine Komponente (z.B. ein Verteiler-Thread), die auf eintreffende Verbindungswünsche reagiert.

Informeller Ablauf aus Serversicht

1. Erzeugen eines SocketServer und Binden an einen bestimmten Port.
2. Warten auf Verbindungswünsche von Clients.
3. Austausch von Daten zwischen Client und Server entsprechend einem wohldefinierten Protokoll (z.B. HTTP).
4. Schließen einer Verbindung (durch Server, durch Client oder durch beide); weiter bei Schritt 2.

Programmstück

```
Socket socket; //reference to socket
ServerSocket port; //the port the server listens to
try {
    port = new ServerSocket(10001, ...);
    socket = port.accept(); //wait for client call
    // communicate with client
    socket.close()
}
catch (IOException e) {e.printStackTrace();}
```

Für das Abhören des Ports kann ein eigener Verteiler-Thread spezifiziert werden; die Bearbeitung übernehmen sogenannte Worker-Threads.

Bidirektionale Stromverbindung

Sockets bestehen aus 2 Strömen für die Duplexverbindung zwischen Client und Server.

Schreiben auf Socket

```
void writeToSocket(Socket sock, String str) throws IOException {
    oStream = sock.getOutputStream();
    for (int k = 0; k < str.length(); k++)
       oStream.write(str.charAt(k));
}
```

Lesen von Socket

```
String readFromSocket(Socket sock) throws IOException {
    iStream = sock.getInputStream();
    String str = "";
    char c;
    while ( (c = (char) iStream.read()) != '\n')
       str = str + c;
    return str;
}
```


Java unterstützt die beiden grundlegenden Klassen:

`java.net.Socket` zur Realisierung der Client-Seite einer Socket.

`java.net.ServerSocket` zur Realisierung der Server-Seite einer Socket.

Client-Seite einer Socket

Server-Seite einer Socket

Client-Seite einer Socket

Constructor

```
public Socket(String host, int port)
    throws UnknownHostException, IOException
```

Der Parameter host ist ein Rechnername, z.B. www11.in.tum.de.

Information über eine Socket

```
public InetAddress getInetAddress();
```

liefert als Ergebnis den Namen und IP-Adresse des entfernten Rechners, zu dem die Socket-Verbindung existiert.

```
public int getPort();
```

liefert als Ergebnis die Nummer des Ports, mit dem die Socket-Verbindung am entfernten Rechner assoziiert ist.

```
public int getLocalPort();
```

liefert als Ergebnis die Nummer des Ports, mit dem die Socket-Verbindung am lokalen Rechner assoziiert ist.

Ein-/Ausgabe

```
public InputStream getInputStream() throws IOException;
```

liefert den InputStream, von dem Daten gelesen werden können.

```
public OutputStream getOutputStream() throws IOException;
```

liefert den OutputStream, in dem Daten geschrieben werden können.

Constructor

```
public ServerSocket(int port)
    throws IOException, BindException
```

erzeugt eine Socket auf Server-Seite und assoziiert sie mit dem Port.

Einrichten/Schließen einer Verbindung

```
public Socket accept() throws IOException
```

diese Methode blockiert und wartet auf Verbindungswünsche von Clients.

```
public void close() throws IOException
```

Ein-/Ausgabe

```
public InputStream getInputStream() throws IOException;
```

liefert den InputStream, von dem Daten gelesen werden können.

```
public OutputStream getOutputStream() throws IOException;
```

liefert den OutputStream, in dem Daten geschrieben werden können.

Netzwerkprogrammierung

Eine ***verteilte Anwendung*** ist eine Anwendung A, dessen Funktionalität in eine Menge von kooperierenden Teilkomponenten A_1 , .., A_n , $n \in IN$, $n > 1$ zerlegt ist;

Jede Teilkomponente umfasst Daten (interner Zustand) und Operationen, die auf den internen Zustand angewendet werden.

Teilkomponenten A_i sind autonome Prozesse, die auf verschiedenen Rechensystemen ausgeführt werden können. Mehrere Teilkomponenten können demselben Rechensystem zugeordnet werden.

Teilkomponenten A_i tauschen über das Netz untereinander Informationen aus.

Die Teilkomponenten können z.B. auf der Basis des Client-Server Modells realisiert werden.

Server-Seite einer Socket

Constructor

```
public ServerSocket(int port)
    throws IOException, BindException
```

erzeugt eine Socket auf Server-Seite und assoziiert sie mit dem Port.

Einrichten/Schließen einer Verbindung

```
public Socket accept() throws IOException
```

diese Methode blockiert und wartet auf Verbindungswünsche von Clients.

```
public void close() throws IOException
```

Ein-/Ausgabe

```
public InputStream getInputStream() throws IOException;
```

liefert den InputStream, von dem Daten gelesen werden können.

```
public OutputStream getOutputStream() throws IOException;
```

liefert den OutputStream, in dem Daten geschrieben werden können.

Generated by Targeteam

Dieser Vorlesungsteil beschäftigte sich mit Aspekten des Betriebssystems und der systemnahen Programmierung. Insbesondere wurden folgende Aspekte behandelt:

- ein allgemeiner Überblick über Betriebssysteme, Betriebssystemarchitekturen.
- Synchronisation von Prozessen beim Zugriff auf gemeinsame Ressourcen.
- Verwaltung von Prozessen und deren Zuteilung an die CPU, um sie auszuführen.
- einfache Verwaltung des Arbeitsspeichers aus der Sicht des Betriebssystems.
- Prozesskommunikation in lokalen und verteilten Systemen.

Eingebettete vernetzte Systeme (IN8014)

Teil I: Betriebssysteme und Systemsoftware

- Prof. J. Schlichter
  - Lehrstuhl für Angewandte Informatik / Kooperative Systeme, Fakultät für Informatik, TU München
  - Boltzmannstr. 3, 85748 Garching

    Email: schlichter@in.tum.de

    Tel.: 089-289 18654

    URL: http://www11.informatik.tu-muenchen.de/