

Tutorium zur Client - Server Kommunikation mit Java Object Streams - V1.0

Mattias Gärtner

April 10, 2007

Abstract

Begriffe, Techniken und Java Klassen und Methoden zur Client/Server Kommunikation.

1 Das Client/Server Modell

Es gibt verschiedene Kommunikationsmodelle zwischen Rechnern. Das hier beschriebene ist das Client/Server Modell, bei dem es einen wartenden Server gibt und Clients, die sich einzeln mit dem Server verbinden.

2 IP Adressen und Ports

Im Internet wird jeder Rechner über eine eindeutige Nummer, der IP-Adresse, identifiziert. Um auf einem Server mehrere Dienste unterscheiden zu können, kommt noch eine Port Nummer hinzu. Mit einer solchen Adresse, wie zum Beispiel 134.95.10.193:7475, ist ein TCP/UDP Verbindungsziel eindeutig bestimmt.

3 UDP, TCP

Nachdem man weiss, mit welchem Programm auf welchem Rechner man kommunizieren will, muss ein Protokoll definiert sein. Das UDP - User Datagram Protocol ist ein sehr einfaches, mit dem man Datenpakete senden und empfangen kann. Da durch technische Einschränkungen Pakete verloren gehen können, oder Pakete wegen verschiedener Routen mit Verzögerung eintreffen können, ist weder Reihenfolge noch Ankommen der Pakete garantiert. TCP setzt auf UDP auf und garantiert Ankommen und Reihenfolge der Pakete. Dies kostet Bandbreite.

4 Verbindungsprobleme

Das Internet ist eine Datenautobahn auf der Staus normal sind. Das heisst Daten kommen nicht kontinuierlich, sondern bleiben einige Zeit aus oder kommen verstärkt. Auch können Verbindungen durch Programmabstürze, Hardware Schäden oder dynamische IP Adressen jederzeit verloren gehen. Clients und

besonders Server müssen daher Verspätungen und Verbindungsabbrüche erkennen und korrekt behandeln. Um Verbindungsabbrüche zu erkennen hat jede Verbindung einen Timeout. Wird innerhalb dieser Zeitspanne nichts gesendet, wird die Verbindung beendet. Gegen Verzögerungen muss die Applikation selbst gefeit sein.

5 Java - Kommunikation über Socket-Objekte

- datagram sockets: Abwicklung über UDP (user datagram protocol), verbindungslos, schnell aber unzuverlässig
- stream sockets: Abwicklung über TCP (transmission control protocol), funktioniert analog zu File-I/O

Mit diesen Sockets kann man sehr einfach UDP und TCP Verbindungen aufbauen. Konkreter Beispielcode folgt weiter unten. Im Rahmen dieser Veranstaltung werden nur TCP Streams benutzt. Zur Erinnerung: Streams sind Datenquellen, bzw. Datenziele. Es gibt zum Beispiel Filestreams, mit denen man Daten in Dateien schreiben kann, oder lesen kann. Im Gegensatz zu Dateien kann man bei Netzwerkstreams nicht wieder an eine andere Stelle der Daten springen, sondern kann immer nur sequentiell lesen bzw. schreiben.

6 Java - Objectstreams

Mit TCP Streams kann man sicher Daten verschicken. Als nächstes muss man ein Datenformat definieren. Da Java die Möglichkeit bietet Objekte zu serialisieren, also in einen Stream zu schreiben und wieder zu lesen, kann man sich das Schreiben eines Parsers sparen. Damit ein Objekt serialisierbar wird, muss es lediglich das Interface *Serializable* implementieren. Beispiel:

```
import java.io.*;
public class DMessage implements Serializable {
    public static final long serialVersionUID = 1;
    public String MyString;
}
```

Alle Java Basistypen wie String, int, char, long sind per se serialisierbar. Auch Arrays davon sind serialisierbar. Alle Unterobjekte müssen ebenfalls serialisierbar sein. Für Klassen wie Vector und Map ist dies der Fall. Das Lesen und Schreiben von Objekten geschieht mit wenigen Zeilen Code und weiter unten gibt es Beispiele.

6.1 Objectstreams - Fire and forget

Bei Netzwerkverbindungen wird mit mehreren Threads gearbeitet. Übergibt man dem Sende-Thread ein Objekt, so wird es nicht sofort gesendet, sondern gelangt erst in einen Puffer. Das bedeutet, zu sendende Objekte müssen immer neu erzeugt werden, und sobald sie dem Sende-Thread übergeben wurden, nicht mehr verändert werden. Dies gilt insbesondere für die Unterobjekte.

6.2 Objectstreams - Referenzen

Man kann mit Referenzen arbeiten. Beispiel:

```
import java.io.*;
public class DMSGGameState extends DMessage {
    ...
    public DMSField[] [] Fields;
    ...
}
```

Wenn `Fields[0][0]` auf dasselbe Objekt wie `Fields[0][1]` verweist, wird das Objekt auch nur einmal gesendet.

7 Java - Serverseite

7.1 ServerSocket-Objekt für den Serverport erstellen

```
int PortNumber = 2000;
int QueueLength = 100; // 0 für unendlich lang
ServerSocket GameSocket;
try {
    GameSocket = new ServerSocket(PortNumber, QueueLength);
} catch (IOException e) {
    System.out.println(e.toString());
}
```

Die `QueueLength` ist nur wichtig bei professionellen Servern, um sich vor DOS Attacken zu schützen. Wenn der Port bereits durch ein anderes Programm belegt ist, kann eine `IOException` auftreten, die abgefangen werden muss.

7.2 Server wartet auf Verbindungen

Ein Server wartet normalerweise endlos. Die Methode `accept` wartet ohne CPU Last zu erzeugen auf eine eingehenden Verbindung eines Clients:

```
try {
    while (true) {
        // wait for a new connection ...
        Socket NewClientSocket = GameSocket.accept();
        CreateGameClient(NewClientSocket);
    }
} catch (IOException e) {
    System.out.println(e.toString());
    ...
}
```

Das `NewClientSocket` Objekt stellt die TCP Verbindung dar. Dabei können `IOExceptions` auftreten, wenn zum Beispiel die Queue volllief.

7.3 Für jede Client-Verbindung einen Empfangsthread starten

Zuerst wird für die neue Verbindung der Empfangstimeout gesetzt. Jeder Client wartet auf Daten. Damit der Hauptthread des Servers nicht warten muss, wird pro Client ein Empfangsthread gestartet:

```
private void CreateGameClient(Socket NewClientSocket) {
    try{
        System.out.println("new client "+NewClientSocket.getInetAddress().toString()+":"+NewClientSocket.getPort());
        NewClientSocket.setSoTimeout(Timeout);
    } catch (IOException e) {
        System.out.println(e.toString());
        ...
    }
    // create new client thread
    GameClient NewGameClient = new GameClient(NewClientSocket);
    // start the thread. The new thread will execute the run() method.
    NewGameClient.start();
}
```

7.4 Der Empfangsthread

Der Empfangsthread erzeugt einen `ObjectInputStream` auf dem `ClientSocket` und liest Objekte, bis ein Fehler auftritt oder die Verbindung geschlossen wird. Dabei können verschiedene Exceptions auftreten. Einige sind normal, einige sind Warnungen und andere bedeuten Fehler:

```
public class GameClient extends Thread {
    public Socket ClientSocket;
    private ObjectInputStream ois;
    ...
    public void run() {
        ClientOk = true;
        try{
            try {
                ois = new ObjectInputStream(new BufferedInputStream(ClientSocket.getInputStream()));
            } catch (EOFException e) {
                // Client closed connection
                return;
            } catch (IOException e) {
                System.out.println(e.toString());
                return;
            }
        }
        CreateSendThread();
        while (ClientOk) {
            Object NewObject = null;
            try {
                NewObject = ois.readObject();
            } catch (EOFException e) {
                // Client closed connection
                break;
            }
        }
    }
}
```

```

    } catch (ClassNotFoundException e) {
        // invalid class => invalid protocol => instant disconnection
        break;
    } catch (SocketTimeoutException e) {
        // timeout => send DMSKeepAlive. When already sent => close
        ....
    } catch (IOException e) {
        // tcp connection broken => instant connection end
        break;
    }
    if (NewObject != null){
        if (!(NewObject instanceof DMessage)){
            // invalid class => invalid protocol => instant disconnection
            break;
        } else {
            HandleDMessage((DMessage)NewObject);
        }
    }
} finally {
    // close connection
    try{
        ClientSocket.close();
    } catch (EOFException e) {
        // ok
    } catch (IOException e) {
        System.out.println(e.toString());
    }
}
}
}
}

```

8 Senden

8.1 Sende-Stream erzeugen

Das Senden ähnelt dem Empfang, lediglich die Klassennamen enthalten Output statt Input.

```

ObjectOutputStream oos;
try {
    oos = new ObjectOutputStream(Client.ClientSocket.getOutputStream());
} catch (IOException e) {
    System.out.println(e.toString());
    ...
}

```

8.2 Ein Objekt senden

Mit der Methode `writeObject` wird ein `Object` in einem Datenstrom verwandelt und irgendwann gesendet. Um die Queue nicht volllaufen zu lassen, kann man mit der Methode `flush` warten, bis die Queue wieder leer ist.

```
try{
    oos.writeObject(Message);
    oos.flush();
} catch (IOException e) {
    System.out.println(e.toString());
    ...
}
```

9 Java - Clientseite

- wie Server-Kommunikation
- allerdings direkte Instanziierung des `Socket`-Objekts, da Verbindung aufgebaut wird
- es ist nur einen Sende- und ein Empfangsthread erforderlich, da nur ein Server

9.1 Beispiel: Serverhostname in IP Adresse auflösen

```
String ServerHostname = "progprak.scale.uni-koeln.de";
try{
    ServerIP = InetAddress.getByName(ServerHostname);
} catch (UnknownHostException e){
    System.out.println("resolving "+ServerHostname+" "+e.toString());
    ...
}
```

9.2 Beispiel: Client baut TCP Verbindung zu Server auf

```
try{
    ClientSocket = new Socket(ServerIP.getHostAddress(),ServerPort);
} catch(IOException e){
    System.out.println("Creating Socket "+ e.toString());
    ...
}
```

Der Rest läuft nun genau wie beim Server ab. Man setzt den Timeout und startet einen Empfangs- und Sendethread.