Einführung Netzwerk

1. Lesehinweise

- Java API
  - Socket
  - Input-/OutputStream
  - Reader/Writer
  - Scanner
  - Thread
  - BlockingQueue
- GASP
- Tutorials auf www.inf.fu-berlin.de/~rosanwo
  - Java IO
  - Praktische Java Klassen

2. Streams

2.1. Technisches

Streams sind gerichtete Kommunikationskanäle, d.h. ein Ende sendet, das andere empfängt Daten. Dabei "fließen" die Daten vom Sender zum Empfänger, d.h. sowohl Lese- als auch Schreiboperationen sind streng sequentiell. Ein mehrfaches Lesen, oder wahlfreier Zugriff sind nicht möglich.

2.2. Umgang mit Streams in Java

Alle IO Operationen in Java laufen über Streams. Das Senderende heißt hier OutputStream, das Leserende InputStream. Auf unterster Ebene werden alle Daten als byte Array übertragen, das heißt aber nicht, dass man nur solche lesen/schreiben kann. Zu diesem Zweck gibt es spezielle Unterklassen, z.b. **PrintStream** ist ein spezieller OutputStream, der alle primitiven Datentypen und Strings schreiben kann. Erzeugt werden Streams wie alle anderen Objekte mittels new. Oftmals werden dabei Streamobjekte geschachtelt. Ein PrintStream, der in eine Datei schreibt wird z.B. so erzeugt:

```
PrintStream ps = new PrintStream(new FileOutputStream(new File(...)));
ps.println("...");
ps.close();
```

Neben den speziellen Streams gibt es noch die Reader/Writer Klassen, um den Zugriff auf Streaminhalte komfortabler zu machen. Typischerweise liest man eine Datei folgendermaßen aus:

```
BufferedReader br = new BufferedReader(new InputStreamReader(new
FileInputStream(new File(...)));
while (...)
        String line = br.readLine();
```

3. Internet

3.1. TCP vs UDP

Es gibt grundsätzlich 2 Protokolle zur Übertragung von Daten über das internet: TCP und UDP.
UDP ist das einfachere der beiden. Es beschreibt einzelne Datenpakte, die unabhängig voneinander und evtl über verschiedene Wege vom Sender zum Empfänger gelangen. Die maximale Größe der Pakete ist jedoch beschränkt, daher besteht die Kommunikation zwischen Sender und Empfänger aus vielen kleinen solcher sog. Datagramme. Die Unabhängigkeit der Pakete und deren geringe Größe machen UDP zum schnelleren der beiden Protokolle, es ergeben sich jedoch auch Probleme. So können Datagramme in einer anderen Reihenfolge beim Empfänger ankommen, als sie losgeschickt wurden, oder es können welche verloren gehen. Für diese Fälle übernimmt UDP keine Garantien.
Daher verwenden wir für unsere Anwendung die Übertragung per TCP. TCP ist ein vermittlungsgebundenes Protokoll, d.h. zu Beginn der Kommunikation wird zwischen den beiden Teilnehmern eine feste Verbindung aufgebaut und am Ende wieder abgebaut. Danach können Daten beliebiger Größe zwischen den Teilnehmern ausgetauscht werden.
Hinter den Kulissen werden natürlich auch die in kleine Pakete aufgeteilt, doch davon braucht man als Anwender nichts mitzubekommen.
TCP übernimmt für die gesamte Kommunikation die Garantie, dass Daten in der Reihenfolge empfangen werden, wie sie gesendet wurden und dass keine Daten verloren gehen.

3.2. Sockets

Ein Teilnehmer einer Kommunikation identifiziert man mit 2 Werten:
• IP Adresse, bzw URL
• Port
Die IP-Adresse kann entweder direkt angegeben werden, oder es wird ein symbolischer Name verwendet, der von einem DNS-Server zu einer IP aufgelöst wird.
Der Turnier Server ist z.B. unter cali.mi.fu-berlin.de unter Port 42523 zu erreichen.

3.3. TCP Verbindung in Java

Folgender Code stellt eine Verbindung zum Turnier-Server her und lässt sich den InputStream vom Server, sowie den OutputStream zum Server geben.

```
Socket socket = new Socket();
socket.connect(new InetSocketAddress("cali.mi.fu-berlin.de", 42523), 10000);
InputStream in = socket.getInputStream();
OutputStream out = socket.getOutputStream();
...
socket.close();
```

Im obigen Bsp. wird außerdem ein Timeout von 10s verwendet. Kann in dieser Zeit keine Verbindung hergestellt werden, bricht der Aufruf ab und wirft eine Exception.

4. GASP

GASP ist ein Zustandsbasiertes Protokoll, d.h. nicht jeder Befehl ist zu jeder Zeit gültig, sondern nur in bestimmten Zuständen.

4.1. Befehls-Formate

Alle Befehle sind String basiert. D.h. es findet keinerlei Kodierung statt, sondern die Befehle werden direkt als Java Strings versendet und empfangen. Jeder Befehl wird mit einem Zeilenumbruch beendet. Befehle, die sich über mehrere Zeilen erstrecken, werden mit

COMPLETE abgeschlossen.
Der Großteil der Befehle läuft nach request-reply Prinzip. Der Client teilt dem Server mit, was er als nächstes tun will und der Server reagiert darauf.
Es gibt aber auch Befehle, die der Server unaufgefordert versendet. Diese sollten speziell behandelt werden.

4.2. Bsp Login

State : PROTOCOL EXPECTED
Client : PROTOCOL GASP 1.0
Server : OK
State : USER EXPECTED

5. Synchronisation

5.1. Threads

Es ist quasi unmöglich, alle Kommunikation und deren Verarbeitung sequentiell abzuarbeiten. Daher sollte reichlich gebrauch von Threads gemacht werden. Threads sind Programmabarbeitungsstränge. Laufen 2 Threads parallel, wird auch die Arbeit, die in diesen 2 Threads gemacht wird, gleichzeitig erledigt. Thread Objekte werden wie folgt erzeugt:

```
Thread t = new Thread(new Runnable() {
        public void run() {
                // Code der im Thread ausgefuehrt wird
        }
});
t.start(); //startet den Thread
...
t.join(); //wartet auf die Beendigung des Thread
```

Typische Grobstruktur:
1. Thread Lesen des Serveroutputs
2. Thread Senden von Befehlen an den Server
3. Thread Verarbeitung der Kommunikation

```
// Thread 1
while(true) {
        String nextServerCmd = reader.readLine();
        incomingCmdQueue.offer(nextServerCmd);
}

// Thread 2
while (true) {
        String cmd = outgoingCmdQueue.take();
        writer.println(cmd);
}

// Thread 3
while (true) {
        String cmd = incomingCmdQueue.take();
        String reply = process(cmd);
```

```
        outgoingCmdQueue.offer(reply);
}
```

5.2. Fehlendes flush()

Jede Schreiboperation muss mit einem flush() abgeschlossen werden. Ansonsten kommt am anderen Ende des Streams nichts an und es passiert einfach gar nichts.

5.3. Deadlocks

Bei der Verwendung mehrerer Threads muss darauf geachtet werden, dass sie sich nicht gegenseitig blockieren. Ein typisches Bsp. wäre ein simpler Entwurf, bei dem sich Lese- und Schreiboperationen stets abwechseln. Hier reicht ein asynchrones Kommando vom Server und der Lese-Thread hat nichts mehr zu lesen und der Schreib-Thread nichts zu schreiben. Das Programm hängt sich auf.

5.4. Request-reply

Leseoperationen auf Streams blockieren i.A. solange, bis wieder Daten gelesen werden können. Daher sollten sie auf keinen Fall im gleichen Thread, wie GUI-Methoden, oder sonstigen zeitkritischen Operationen passieren.
Auch ist darauf zu achten, Lese- und Schreiboperationen sauber zu trennen. D.h. jedes Kommando muss erst vollständig empfangen werden und dann kann die Antwort gesendet werden. Dies ist besonders kritisch, da es sowohl synchrone, wie auch asynchrone Kommunikation mit dem Server gibt.

6. Tipps zur Implementation

- Aufgabentrennung mit versch. Threads
- Trennung Verarbeitung - Lesen/Schreiben
- Konstanten für Protokollschlüsselworte. Evtl Struktur für einzelne Befehle
- Tests
- Transparenz
- Enge Absprache mit anderen Gruppen