Lehrstuhl Informatik 4
Andreas Blume, Egor Kudrjaschow
Nils Dunker, Bastian Korte, Minsu So

technische universität dortmund

Wintersemester 2020/2021

Rechnernetze und verteilte Systeme
Programmieraufgabe 2: Client-Server-Anwendung

Ausgabe: Fr. 4.12.2020; Abgabe: Di. 5.1.2021

1 Allgemeine Hinweise zur Bearbeitung

Die Programmieraufgaben sollen in Gruppen mit **maximal fünf Studierenden** bearbeitet werden. Die Implementierung soll in **Java 11** geschehen. Machen Sie sich also zunächst mit der Programmierung in Java (siehe z. B. offizielle Dokumentation der Java API) vertraut.

1.1 Bibliotheken

Zur Lösung der Programmieraufgaben soll ausschließlich auf die durch die Java-Standardbibliothek zur Verfügung gestellten Klassen zurückgegriffen werden, die sich in den folgenden Paketen (oder Subpaketen davon) befinden:

- `java.io.*`
- `java.lang.*`
- `java.net.*`
- `java.nio.*`
- `java.security.*`
- `java.util.*`
- `java.text.*`

Klassen aus den Pakten `com.sun.net.*` bzw. `sun.net.*` oder Bibliotheken von Drittanbietern dürfen nicht eingebunden werden. Ebenfalls ist das Kopieren von Quellcode aus Bibliotheken oder anderen Fremdquellen nicht erlaubt. **Verstöße führen zum Nichtbestehen der Studienleistung!**

1.2 Fehlerbehandlung

Im Programm sollten Exceptions (bzw. Fehlerzustände), die beispielsweise bei arithmetischen Operationen (Division durch 0) oder bei Zugriff auf noch nicht erzeugte Objekte entstehen können, verarbeitet werden. Eine unzureichende Fehlerbehandlung führt zu Punktabzug.

1.3 Kommentare und Dokumentation

Kommentieren Sie Ihren Quelltext ordentlich! Für Abgaben ohne ausreichende Kommentare werden Punkte abgezogen. Zusätzlich sollten Sie alle über die Aufgabenstellung hinausgehenden Bedienungsaspekte der Applikation (Befehle, Kommandozeilenparameter, ...) dokumentieren, um eine reibungslose Korrektur zu ermöglichen. Neben einer Textdatei ist es auch zulässig, diese Dokumentation von der Anwendung selbst (z. B. direkt nach dem Starten) ausgeben zu lassen.

1.4 Abgabe

Die Abgabe findet im Moodle statt. Die Abgabe kann bis zum Abgabetermin beliebig oft geändert werden, wobei die zuletzt abgegebene Version bewertet wird.

Die Abgabe muss vom Compiler übersetzbar sein und unter **Java 11** laufen. Abzugeben ist ein zip-Archiv mit allen zum Projekt gehörenden Quelltexten und eine ausführbare jar-Datei.

2 Tipps zur Programmierumgebung

2.1 IDE

Ein hilfreiches Werkzeug bei der Entwicklung von Software ist eine *integrierte Entwicklungsumgebung* (IDE). Bekannte Java-IDEs sind:

- **IntelliJ IDEA**
- **Eclipse**
- NetBeans IDE
- Editoren mit Java Plugin wie
  - Visual Studio Code
  - Atom
  - viele weitere

Alle genannten IDEs verfügen über detaillierte Anleitungen zur Installation und dem Import von Projekten.

2.2 Git Repositories

Git ist eine freie Software zur verteilten Versionsverwaltung von Dateien, die die gemeinsame Arbeit an den Programmieraufgaben unterstützen kann. Eine kurze Anleitung zu Git finden Sie z. B. hier. Einige Remote Git Server sind:

- (Fakultät/IRB) Gitea

- (Fachschaft) GitLab
- (Microsoft) GitHub
- (Atlassian) Bitbucket

Wichtig: Erstellen Sie *private* Repositories um Plagiate zu vermeiden! Öffentliche Repositories werden von Internet-Suchmaschinen (z. B. Google) indexiert und können somit von anderen Gruppen gefunden werden.

3 Aufgabe

In der Vorlesung wurden die beiden typischen Anwendungsarchitekturen Client-Server-Modell und Peer-To-Peer-Modell vorgestellt. Das Client-Server-Modell besteht aus Client und Server. Der Client initiiert zunächst den Kontakt zum Server und kann anschließend einen vom Server angebotenen Dienst in Anspruch nehmen (siehe Abbildung 1).

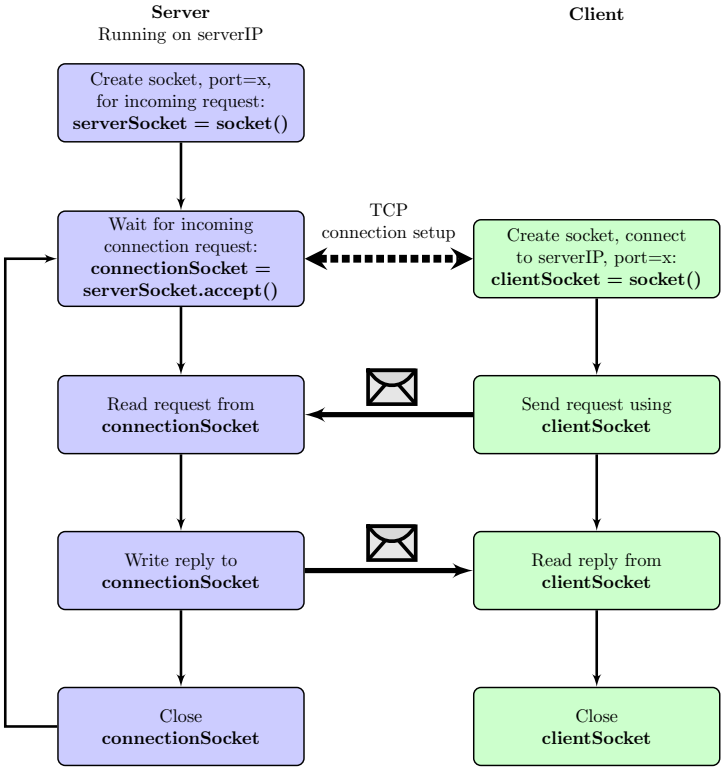

Abbildung 1: Client-Server-Anwendung mit TCP. [KR17]

In dieser Aufgabe sollen Sie einmal selbst eine einfache Client-Server-Anwendung entwickeln, wobei Nachrichten zwischen Client und Server per **TCP** ausgetauscht werden. Dazu soll ein Server implementiert werden, der verschiedene Dienste bereitstellt und am **Port 2020** auf Anfragen von einem, ebenfalls zu implementierenden, Client reagiert.

3.1 Dienste

Der Server soll die folgenden Dienste bereitstellen. Die Anfragen vom Client und die Antworten vom Server sind jeweils **UTF-8** kodiert.

Zeit und Datum

Der Client kann die aktuelle Zeit und das heutige Datum anfragen. Die Antwort des Servers basiert auf der Serverzeit. Die Bedeutung vom angegebenen Zeit- und Datumsformat können Sie zum Beispiel hier nachlesen.

Anfrage	Anfrageformat des Clients	Antwort des Servers
aktuelle Zeit	`GET Time`	`TIME HH:mm:ss`
heutiges Datum	`GET Date`	`DATE dd.MM.yyyy`

Rechner

Der Server stellt verschiedene einfache Rechenoperationen bereit, die vom Client genutzt werden können. Während das Ergebnis von Addition, Subtraktion und Multiplikation ein Integer ist, wird bei der Division eine Gleitkommazahl (float) berechnet und zurückgeschickt. Die Division durch Null beantwortet der Server mit `QUOTIENT undefined`.

Anfrage	Anfrageformat des Clients	Antwort des Servers
Addition	`ADD <Integer1> <Integer2>`	`SUM <Ergebnis>`
Subtraktion	`SUB <Integer1> <Integer2>`	`DIFFERENCE <Ergebnis>`
Multiplikation	`MUL <Integer1> <Integer2>`	`PRODUCT <Ergebnis>`
Division	`DIV <Integer1> <Integer2>`	`QUOTIENT <Ergebnis>`

Echo und Discard

Der Server soll die beiden Netzwerkdienste Echo und Discard unterstützen. Die Aufgabe des Echo-Dienstes ist es, alle empfangenen Daten unverändert zum Client zurückzusenden. Der Discard-Dienst verwirft alle empfangenen Daten und sendet keine Antwort.

Anfrage	Anfrageformat des Clients	Antwort des Servers
Echo	`ECHO <String>`	`ECHO <String>`
Discard	`DISCARD <String>`	–

Ping-Pong

Damit der Nutzer des Clients die Round Trip Time (RTT) zwischen Client und Server messen kann, soll ein `PING` vom Client mit `PONG` vom Server beantwortet werden.

Anfrage	Anfrageformat des Clients	Antwort des Servers
Ping-Pong	`PING`	`PONG`

3.2 Historie

Die Verbindung zwischen Client und Server ist **persistent**. Das heißt, nachdem Client und Server miteinander verbunden sind, kann der Client nacheinander mehrere Anfragen an den Server stellen und irgendwann die Verbindung zum Server wieder schließen.

Mit `HISTORY` soll der Client die Möglichkeit bekommen, beim Server alle bisher gestellten Anfragen abfragen zu können. Die Historie bezieht sich dabei jeweils nur auf die aktuelle Verbindung. Eine Historie über mehrere aufeinanderfolgende Verbindungen zwischen Client und Server gibt also nicht.

Die zu implementierenden Client-Anfragen sind:

- `HISTORY`
  → Server antwortet mit allen bisher vom Client gestellten Anfragen.
- `HISTORY <Integer>`
  → Server antwortet mit den letzten <Integer> (z. B. den letzten 5 oder 10) vom Client gestellten Anfragen. Wenn bisher weniger als <Integer> Anfragen vom Client an den Server geschickt wurden, dann gibt der Server nur alle Anfragen zurück, die bisher gestellt wurden.

Das Antwortformat des Servers können Sie selbst wählen und ist deshalb auch zu dokumentieren. Existiert noch keine Historie, dann antwortet der Server mit `ERROR Keine Historie vorhanden!`.

3.3 Fehlerhafte Anfrage vom Client

Wenn die vom Client gesendete Anfrage nicht dem geforderten Format entspricht (z. B. Float statt Integer), dann ist der Client vom Server darüber zu informieren.

```
ERROR Falsches Format!
```

Bei einer unbekannten Anfrage antwortet der Server wie folgt.

```
ERROR Unbekannte Anfrage!
```

4 Implementierung

Client und Server sollen in dieser Programmieraufgabe als zwei eigenständige Java-Programme entwickelt werden, die beide auf dem selben End-System laufen. Trotzdem sollen beide Programme so implementiert werden, dass Client und Server durch wenige Anpassungen auch auf zwei unterschiedlichen End-Systemen laufen könnten.

Wird vom Nutzer eine fehlerhafte IP-Adresse oder einer falscher Port eingegeben, dann soll eine aussagekräftige Fehlermeldung ausgegeben werden und sich das Programm beenden. Das Gleiche gilt bei Fehlern während des Verbindungsaufbaus.

4.1 Server

Das Java-Programm für den Server soll zunächst den Port vom Nutzer entgegennehmen. Wenn der Port korrekt eingegeben wurde (korrekter Datentyp und Port 2020), dann wird der Server mit seinen Diensten gestartet. Nun wartet der Server auf eine Verbindung zum Client und beantwortet anschließend die Anfragen.

Der Server kann durch die Eingabe "J" beendet werden.

```
An welchem Port soll der Server gestartet werden? // Programmausgabe

Port: abc   // "Port: " ist eine Programmausgabe, "abc" eine Nutzereingabe

Kein korrekter Port! Aktuell ist nur Port 2020 möglich.

Der Server wurde beendet.
```

Beispiel 1: Eine falsche Porteingabe führt zu einer Fehlermeldung.

```
An welchem Port soll der Server gestartet werden? // Programmausgabe

Port: 2020   // "Port: " ist eine Programmausgabe, "2020" eine Nutzereingabe

Der Server mit den Diensten:
  * Zeit und Datum
  * Rechner
  * Echo und Discard
  * Ping-Pong
wurde gestartet und wartet am Port 2020 auf Anfragen vom Client.

Wenn Sie den Server beenden wollen, dann geben Sie "J" ein: J

Der Server wurde beendet.
```

Beispiel 2: Nach der korrekten Porteingabe startet der Server mit seinen Diensten.

4.2 Client

Das Java-Programm für den Client soll vom Nutzer die IP-Adresse und den Port des Servers entgegennehmen. Da der Server ebenfalls lokal läuft, sind ausschließlich Localhost-Adressen erlaubt. Wir beschränken uns hier auf die IPv4-Adresse **127.0.0.1** und die Eingabe **localhost**. Andere Eingaben sind ungültig und führen zu einer entsprechenden Fehlermeldung (siehe Beispiel 3).

Der eingegebene Port soll ebenfalls überprüft werden (korrekter Datentyp und Port 2020; siehe Beispiel 4). Wurden die IP-Adresse und der Port korrekt eingegeben, dann baut der Client eine **TCP-Verbindung** zum Server auf.

Nach einem erfolgreichen Verbindungsaufbau kann der Client Anfragen an den Server stellen. Einige Anfragen und deren Antworten sind in Beispiel 6 angegeben. Das Java-Programm gibt vor jeder Anfrage "`$ `" aus, anschließend folgt die Eingabe des Nutzers. Aus der Antwort des Servers werden jeweils die eigentlichen Informationen extrahiert und ausgegeben. Einzige Ausnahme bildet Ping-Pong, da die Serverantwortet keine weiteren Informationen enthält.

Die Verbindung zum Server kann mit dem Kommando `EXIT` beendet werden.

```
Mit welchem Server wollen Sie sich verbinden? // Programmausgabe

IP-Adresse: abc // "IP-Adresse: " ist eine Programmausgabe, "abc" eine Eingabe

Falsche IP-Adresse! Aktuell ist nur die IPv4-Adresse 127.0.0.1 und die Eingabe
localhost möglich.

Der Client wurde beendet.
```

Beispiel 3: Eine falsche Adresseeingabe führt zu einer Fehlermeldung.

```
Mit welchem Server wollen Sie sich verbinden? // Programmausgabe

IP-Adresse: 127.0.0.1 // "IP-Adresse: " und "Port: " sind Programmausgaben
Port: abc             // "127.0.0.1" und "abc" Nutzereingaben

Kein korrekter Port! Aktuell ist nur Port 2020 möglich.

Der Client wurde beendet.
```

Beispiel 4: Eine falsche Porteingabe führt zu einer Fehlermeldung.

```
Mit welchem Server wollen Sie sich verbinden? // Programmausgabe

IP-Adresse: localhost  // "IP-Adresse: " und "Port: " sind Programmausgaben
Port: 2020             // "localhost" und "2020" Nutzereingaben

Fehler beim Verbindungsaufbau! Es konnte keine TCP-Verbindung zum Server mit
IP-Adresse localhost (Port: 2020) hergestellt werden.

Der Client wurde beendet.
```

Beispiel 5: Fehler beim Verbindungsaufbau mit entsprechender Fehlermeldung. Tritt zum Beispiel auf wenn der Server noch nicht gestartet wurde.

```
Mit welchem Server wollen Sie sich verbinden? // Programmausgabe

IP-Adresse: localhost   // "IP-Adresse: " und "Port: " sind Programmausgaben
Port: 2020              // "localhost" und "2020" Nutzereingaben

Eine TCP-Verbindung zum Server mit IP-Adresse localhost (Port: 2020) wurde
hergestellt. Sie können nun Ihre Anfragen an den Server stellen.

$ GET Time   // "$ " ist eine Programmausgabe, "GET Time" eine Nutzereingabe
12:00:00     // extrahierte Information aus der Serverantwort TIME 12:00:00

$ GET Date   // "$ " ist eine Programmausgabe, "GET Date" eine Nutzereingabe
01.12.2020   // extrahierte Information aus der Serverantwort DATE 01.12.2020

$ HISTORY    // "$ " ist eine Programmausgabe, "HISTORY" eine Nutzereingabe
GET Date     // alle bisher
GET Time     // gestellten Anfragen

$ DIV 1 2    // "$ " ist eine Programmausgabe, "DIV 1 2" eine Nutzereingabe
0.5          // extrahierte Information aus der Serverantwort QUOTIENT 0.5

$ DIV 5 0    // "$ " ist eine Programmausgabe, "DIV 5 0" eine Nutzereingabe
undefined    // extrahierte Information aus der Serverantwort QUOTIENT undefined

$ ECHO Spieglein, Spieglein.
Spieglein, Spieglein.

$ DISCARD Bitte ignorieren. // Achtung: Keine Serverantwort!

$ PING
PONG         // Achtung: Bei PONG wird nichts extrahiert!

$ MUL 3.14159 2.71828
Falsches Format!     // Grund: Float statt Integer

$ CALC 2+3
Unbekannte Anfrage!  // Grund: Kommando CALC unterstützt der Server nicht

$ abc231
Unbekannte Anfrage!  // Grund: Anfrage versteht der Server nicht

$ EXIT
Die Verbindung zum Server wurde beendet.

Der Client wurde beendet.
```

Beispiel 6: Nach dem erfolgreichen Verbindungsaufbau kann der Client Anfragen an den Server stellen. Verbindung zum Server beenden mit dem Kommando `EXIT`.

5 Sockets in Java

Um zu vermeiden, dass jeder Programmierer, der über eine Netzwerkschnittstelle kommunizieren will, alle Protokolle, die dafür nötig sind (z. B. Ethernet, IP, TCP, UDP, etc.) selbst implementieren muss, gibt es eine Software-Abstraktion mit dem Namen *Socket*.

In Java werden zwei unterschiedliche Arten von Sockets unterschieden: *Sockets* und *ServerSockets*. Beide verhalten sich unterschiedlich.

Sockets sind diejenigen Sockets, über die tatsächlich Nachrichten verschickt werden können. Um eine Verbindung aufzubauen dient folgender Code:

```
Socket mySocket = new Socket();
mySocket.connect(
    new InetSocketAddress( "42.42.42.42", 12345 )
);
```

Dadurch wird eine Verbindung mit dem Rechner aufgebaut, der die Adresse `42.42.42.42` hat. Zur Adresse gehört noch ein Port. Da ein Rechner mehrere Verbindungen verwalten kann, kann mit Hilfe des Ports auf der Gegenstelle das zugehörige Programm ausgewählt werden.

Um textbasiert zu kommunizieren, werden in Java Helferobjekte benötigt. Das ist zum Senden der *PrintWriter* und für den Empfang der *BufferedReader*. Die *read*-Funktionen des *BufferedReader*s blockieren. Das heißt, dass sie warten, bis zu lesende Inhalte verfügbar sind. Dazu wieder ein Beispiel:

```
Socket mySocket = new Socket();
mySocket.connect( new InetSocketAddress( "42.42.42.42", 12345 ) );

PrintWriter out = new PrintWriter(
    new OutputStreamWriter(
        mySocket.getOutputStream(), StandardCharsets.UTF_8
    ), true
);

BufferedReader in = new BufferedReader(
   new InputStreamReader(
       mySocket.getInputStream(), StandardCharsets.UTF_8
   )
);
```

ServerSockets dienen dazu auf ankommende Verbindungen zu warten. Ein ServerSocket muss an einen lokalen Port gebunden werden:

```
ServerSocket ssocket = new ServerSocket();
ssocket.bind( new InetSocketAddress( 12345 ) );
```

Die Methode *accept* ist eine blockierende Methode, die wartet, bis eine Verbindung aufgebaut wird. Wird eine Verbindung aufgebaut, so kehrt die Methode zurück und übergibt als Ergebnis einen neuen Socket. Mit Hilfe dieses neuen Sockets kann dann mit der Gegenstelle kommuniziert werden.

```
Socket client = ssocket.accept();
```

Da die Netzwerkschnittstelle ein Bauteil des Rechners ist, ist auch die Interaktion mit dem Betriebssystem nötig. Daher „lebt“ jeder Socket im Betriebssystem und es muss dem Betriebssystem mitgeteilt werden, dass man die Verbindung nicht länger benötigt. Alle Sockets – also auch ServerSockets – müssen daher geschlossen werden. Ansonsten bleiben sie auch nach Ende des Programms offen. Ein erneutes Öffnen ist dann nicht mehr möglich – auch dann nicht, wenn das Programm neu gestartet wird. Denken Sie also immer an

```
mySocket.close();
```

beziehungsweise

```
ssocket.close();
```

Weitere Informationen über Sockets finden Sie in “The Java Tutorials”, Abschnitt “All About Sockets”, oder weiterer Java-Literatur Ihrer Wahl.

Literatur

[KR17] J. Kurose and K. Ross. *Computer Networking: A Top-Down Approach, Global Edition*. Pearson Education Limited, 2017.