Hausübung 2: Datei-Transfer als verteilte Anwendung

Letzte Änderung 13.12.20

Es soll eine einfache verteilte Anwendung zum Transfer von Dateien realisiert werden. Sie besteht aus einem Server-Prozess und mehreren Client-Prozessen. Der Datei-Server empfängt Nachrichten verschiedener Art von Clients und antwortet darauf. Zum Beispiel kann ein Client in einer Nachricht eine Datei anfordern, die der Server in seiner Antwort an den Client ausliefert,

Der Client hat eine eigene GUI, für den Server können Sie optional auch eine vorsehen. Die GUI des Clients soll Sitzungsaufbau, -abbau und Kommunikation nach dem unten beschriebenen *Dateitransfer-Protokoll* ermöglichen. Mit der GUI des Clients können die Dateitransfer-Operationen angestoßen werden und die Liste der verfügbaren Dateien angezeigt werden. Als Klassenbibliothek für die GUI können Sie Swing oder JavaFX wählen. Die GUI des Servers, wenn vorhanden, ermöglicht es

- den Server zu starten und zu stoppen,
- das Protokoll der empfangenen und gesendeten Nachrichten anzuzeigen,
- die Liste der ausgelieferte Dateien anzeigen: an wen, wann, ...

Beide GUIs fangen *alle* Fehleingaben des Benutzers auf, insbesondere können die Protokollvorgaben nicht verletzt werden.

Transportschicht: TCP/IP-Verbindung über Sockets

Die Prozesse sollen mittels TCP/IP-Verbindungen über Sockets miteinander kommunizieren. Zur Herstellung solcher Verbindungen erzeugt der Server ein Objekt der Klasse `java.net.ServerSocket`. Dabei wird ein Socket geöffnet und an die lokale IP-Adresse gebunden. Danach wartet der Server in der blockierenden Methode `accept()` auf Clients.

Der Client erzeugt ein Objekt der Klasse `java.net.Socket`. Dabei wird dieses mit dem Socket des Servers verbunden. Im Erfolgsfall ist jetzt eine TCP/IP-Verbindung zwischen Client und Server hergestellt: Serverseitig wird dabei für die Kommunikation mit dem Client ein zusätzlicher Kommunikationssocket erzeugt.

Anwendungsschicht: Dateitransfer-Protokoll

Server und Client tauschen jetzt Nachrichten über diese TCP/IP-Verbindung und folgen dabei dem hier definierten Protokoll.

Nachrichten und ihr Format

Das Protokoll erlaubt nur Nachrichten von der Art, wie sie im Anhang angegeben sind. Jede Nachricht wird durch eine Zeichenkette dargestellt, die mit einem Zeilentrenner-Zeichen (*end line, '\n')* endet. Der String wird in ein Byte-Array umgewandelt (ggf. implizit durch `PrintWriter`) und über den Ausgabestrom des Sockets gesendet. Beim Empfang wird ebenfalls *zeilenweise* aus dem Eingabestrom des Sockets gelesen, wobei auch die Rückumwandlung des Byte-Stroms nach String stattfindet (ggf. implizit durch `BufferedReader`).

Phase 1. Sitzungsaufbau

Ein Client sendet ein `CON`. Der Server antwortet im Erfolgsfall mit `ACK`, im Fehlerfall mit `DND`. Dieser Fall tritt ein, wenn beim Server bereits drei Sitzungen bestehen. Der Server baut dann auch die TCP/IP-Verbindung mit dem Client ab, und schließt den Kommunikationssocket.

Phase 2. Sitzung

Ein Client kann jetzt die Dienste des Datei-Servers nutzen und Dateien transferieren, indem er folgende Nachrichten versendet:

- `LST` fordert eine Liste der beim Datei-Server vorhandenen Dateien an
- `PUT` übertragt eine lokale, beim Client vorhandene Datei zum Server, der die Datei bei sich ablegt
- `GET` fordert eine Datei vom Server an, die nach Auslieferung beim Client abgelegt wird.
- `DEL` an löscht eine Datei auf dem Server.
- `DAT` enthält als Bytefolge den Inhalt der zu übertragenden Datei.

Phase 3. Sitzungsabbau

Der Client sendet `DSC`, der Server antwortet mit `DSC`, um den Abbau zu bestätigen. Die TCP/IP-Verbindung zwischen Client und Server wird abgebaut, insbesondere werden die Kommunikationssockets auf beiden Seiten geschlossen.

Hinweise und Entwurfsvorgaben

1. Ein Team besteht immer aus **zwei** Personen, die **eine** gemeinsame Lösung abgeben. Zwingend vorgeschrieben ist, dass **pro Team ein gemeinsames, privates git-Repository** verwendet wird, während der gesamten Bearbeitungsdauer des Projektes.
2. Der Entwurf ist in zwei **Klassendiagrammen** (Server und Client) zu dokumentieren. Darüber hinaus muss der Quelltext vollständig mit sinnvollen **Dokumentationskommentaren** (für Klassen, Schnittstellen, Methoden und Attribute) erläutert sein. Die API-Beschreibungen müssen mit `javadoc` erzeugt werden.
3. Sehen Sie auf jeden Fall zwei Pakete vor: `pis.hue2.client`, `pis.hue2.server` mit Klassen `LaunchClient` und `LaunchServer`. Sinnvoll ist ein drittes Paket `pis.hue2.common`, in dem genau die Klassen vorkommen, die sowohl vom Server als auch vom Client verwendet werden. Dies könnten z.B. die Klassen der Objekte sein, die die ausgetauschten Nachrichten repräsentieren.
4. Der Server soll jeden Client in einem eigenen Thread bedienen.
5. Clients und Server müssen nicht unbedingt auf verschiedenen Rechnern laufen. Zum Testen empfiehlt es sich, alle Prozesse auf dem gleichen Rechner zu starten. IP-Adresse des lokalen Rechners ist 127.0.0.1.
6. Ein Client mit grafischer Oberfläche ist für Testzwecke nicht unbedingt erforderlich. Mit einem gewöhnlichen `nc`-Client oder `putty`-Client lässt sich ein Datei-Server und das vorgegebene Protokoll ebenfalls gut testen.
7. Die GUI des Clients beauftragt den tatsächlichen Dateitransfer nur, z.B. über einen

SwingWorker Thread, da es sich hier um eine potenziell lang dauernde Operation handelt.

8. Das oben beschriebene Protokoll ist *genau, so* zu implementieren. Ein guter Test ist es, Client- und Server-Implementierungen aus verschiedenen Lösungen zu mischen.
9. Der Datei-Server muss dafür sorgen, dass der gleichzeitige Zugriff durch zwei oder mehr Clients auf die gleiche Datei nicht zu Inkonsistenzen führt. Grundsätzlich gibt es dazu mehrere Lösungen, z.B.:
   1. *threadsicher*: Sie erstellen eine mit Sperren gesicherte Datenstruktur, die den nebenläufigen Zugriff auf die vorhandenen Dateien steuert.
   2. durch *Beauftragung*: Es wird nur einem einzigem Bediener-Thread erlaubt, auf die vorhandenen Dateien zuzugreifen. Andere Threads, die auf eine Datei zugreifen wollen, müssen diesen Bediener-Thread beauftragen.
10. Wenn ein Client eine Nachricht empfängt, wird dies in einem eigenen Empfänger-Thread geschehen. Die empfangene Nachricht muss in der GUI angezeigt werden, es muss also eine ***nicht threadsichere*** Swing (oder JavaFX-) Datenstruktur geändert werden. Dies darf der Empfanger-Thread bekanntlich **nicht** tun, sondern er muss den Event-Dispatch-Thread (Swing) bzw. den JavaFX-Application-Thread mit dieser Aufgabe beauftragen. Dies kann mit den Methoden `SwingUtilities.invokeLater(…)` bzw. `Platform.runLater(..)`) erreicht werden.
11. In Client und Server kann man die Empfangskomponente von der GUI entkoppeln, indem man eine Schnittstelle namens `Ausgabe` einführt mit allen Methoden, die den Zustand der GUI verändern: Also z.B. `zeigeNachricht(...)`, `zeigeListe(...)` , u.a. .

Codebeispiele und Literatur (alle online verfügbar)

(1) Client/Server-Programmierung in Netzwerken ,
Ratz, Dietmar / Scheffler, Jens / Seese, Detlef / Wiesenberger, Jan
In: Grundkurs Programmieren in Java (über die THM-Bibliothek)

(2) Netzwerkprogrammierung mit Sockets, Kap. 24 in
Heinisch, Cornelia / Müller-Hofmann, Frank / Goll, Joachim
Java als erste Programmiersprache, 6.Aufl. (In späteren Auflagen fehlt dieses Kapitel)

(3) Netzwerkprogrammierung, Kap 11 in Java 7 - Mehr als eine Insel :
http://openbook.rheinwerk-verlag.de/java7/ (Aufl. 7 reicht für diese HÜ.)

(4) im Trail Custom Networking, Kapitel All About Sockets

Anhang

```
/**
* Available instructions of the protocol.
*/
public enum Instruction {

 /**
 * CONNECT
 * connection request
 * usage: CON
 */
 CON,

 /**
 * DISCONNECT
 * disconnect notification
 * usage: DSC
 */
 DSC,

 /**
 * ACKNOWLEDGED
 * operation acknowledgement
 * usage: ACK
 */
 ACK,

 /**
 * DENIED
 * negative operation acknowledgement
 * usage: DND
 */
 DND,

 /**
 * LIST
 * list a directory
 * usage: LST
 **/
 LST,

 /**
 * UPLOAD
 * upload a file
 * usage: PUT <filename : string>
 */
 PUT,
```

```
 /**
 * DOWNLOAD
 * download a file
 * usage: GET <filename : string>
 */
 GET,

 /**
 * DELETE
 * delete a file
 * usage: DEL <filename : string>
 */
 DEL,

 /**
 * DATA
 * encapsulates the data to be transmitted
 * usage: DAT <length : string (long)> <data : byte[]>
 */
 DAT,

}
```