Übungsaufgabe #2: Library - Client/Server

05.11.2003

In dieser Aufgabe soll die Bibliothek aus der vorherigen Aufgabe so erweitert werden, dass Ausleihstationen auf mehreren Rechnern betrieben werden können. Kopieren Sie dazu alle Dateien aus Aufgabe 1 in das Verzeichnis `aufgabe2`. Zur Lösung der Aufgabe werden die aus der Tafelübung bekannten Datenströme (Streams), die Netzwerkkommunikation über Sockets und Threads benötigt.

Um auf die Bücher von verschiedenen Rechnern aus zugreifen zu können, soll die Bibliothek in einen Server-Teil und einen Client-Teil aufgeteilt werden. Der Server verwaltet die Datenbank, während die Clients den Zugriff auf die Medien ermöglichen sollen.

Die Aufgabe ist zur einfacheren Bearbeitung in folgende Teilaufgaben untergliedert:

a) Bevor mit der Implementierung begonnen wird, soll zunächst ein Design der Anwendung erstellt werden, um die anschließende Implementierung zu erleichtern. Es sollten die geplanten Klassen, Interfaces und Aufgaben festgelegt werden, benötigte Threads und die Synchronisation gemeinsamer Datenstrukturen.
Lesen Sie sich zuerst die folgenden Teilaufgaben durch und erstellen sie dann Ihre Lösung. Eine einfache ASCII-Textdatei reicht, bei umfangreicheren Grafiken kann auch ein PDF-Dokument erstellt werden.

b) Die Klassen sollen für die Aufgabe auf Pakete aufgeteilt werden. Die Aufteilung soll wie folgt geschehen:
   - Das Paket `library` soll alle Klassen enthalten, die sowohl vom Server als auch von den Clients genutzt werden.
   - Das Paket `library.server` soll alle Klassen enthalten, welche nur vom Server genutzt werden.
   - Das Paket `library.client` enthält die Klasse, die nur von den Clients genutzt werden.

   Für die folgenden Teilaufgaben muss das Interface `LibraryDB` um eine Methode erweitert werden:

   `void update(int id, Item item)`
   Ähnlich zur Methode `register()` kann mit Hilfe dieser Methode ein neues Objekt in der Datenbank angelegt werden, jedoch muss hierzu schon eine ID bekannt sein. Ausserdem kann diese Methode dazu verwendet werden ein existierendes Objekt zu aktualisieren.

c) Die Kommunikation zwischen den Clients und dem Server soll über Socket-Verbindungen geschehen. Implementieren Sie hierzu eine Klasse `LibraryServer`, die an einem bestimmten Port Verbindungen entgegen nimmt. Als Portnummer soll Ihre Benutzerkennung aus dem CIP-Pool dienen (diese können Sie mit dem Programm `id` ermitteln). Für jede geöffnete Verbindung soll der Server jeweils einen Thread erzeugen, welcher Anfragen vom Client entgegen nimmt und die Änderungen in die interne Datenbank übernimmt.

   Über die Verbindung werden Anfragen in der From von `Request`-Objekten ausgetauscht. Der jeweilige Thread liest vom Socket über einen Objektstream die `Request`-Objekte ein und ruft an diesem die Methode `Result perform(LibraryDB)` auf. Diese führt die entsprechende Aktion auf der Datenbank durch und liefert das Ergebnis des Aufrufes als `Result`-Objekt zurück. Das Ergebnis schickt der Server zum Client zurück. Hinweis: auch eine Exception ist ein Ergebnis.

   Um alle Clients über Änderungen der Datenbank zu informieren dient ein weiterer Thread. Dieser wird von allen anderen Threads über eingehende Aufrufe informiert und sendet die `Request`-Objekte an alle Clients weiter. Eine Ausnahme stellen die `register`-Requests dar: an ihrer Stelle sollen `update`-Requests an die Clients verschickt werden. Wenn sich ein neuer Client mit dem Server verbindet, so bekommt er von diesem Thread alle Items durch `update`-Requests übermittelt.

d) Die Klasse `LibraryFrontend` soll zunächst um eine zusätzliche Methode erweitert werden, welche die gespeicherten Titel zurückgibt: `Vector list(void)`.

   Verändern Sie die Klasse `LibraryFrontend` nun so, dass die Klasse anstelle von `LibraryDBImpl` ein Objekt der Klasse `LibraryDBProxy` benutzt. Diese Klasse implementiert ebenfalls das Interface `LibraryDB`. Beim Erzeugen einer Instanz soll eine Socket-Verbindung zum Server aufgebaut werden. Über diese Verbindung werden Veränderungen in Form von `Request`-Objekten weitergeleitet.

   Die Klasse `LibraryDBProxy` soll als Cache der Server-DB dienen. Das heißt, `get`-Anfragen können lokal beantwortet werden. Die Methoden `register()`, `lock()` und `unlock()` sollen so implementiert werden, dass jeweils eine Anfrage an den Server geschickt wird. Wird ein Rückgabewert erwartet, soll der Client blockieren bis die Antwort zur Verfügung steht.

   Alle vom Server eingehenden Objekte sollen durch einen eigenen Thread entgegengenommen werden. Durch `Request`-Objekte soll der Datenbank-Cache aktualisiert werden. `Result`-Objekte müssen auf geeignete Weise dem blockierten Aufrufer zugestellt werden.

   Hinweis: Durch eine `register`-Anfrage würde der Client selbst eine ID erzeugen die sich ggf. von der Server-ID unterscheiden würde. Aus diesem Grund wandelt der Server `register`-Anfragen in `update`-Anfagen um, bevor er sie an die Clients weiterreicht.

Noch ein Tipp zum Schreiben der Client/Server Anwendung:

- Ein Objektstream überträgt den Zustand eines Objektes nur einmal. Anschließend wird nur noch eine symbolische Referenz übertragen. Um die Zustandsänderung eines Objektes zu übertragen kann entweder jeweils eine Objektkopie (`clone()`) übertragen, oder mit der Methode `reset()` der Objektstrom zurückgesetzt werden.

<u>Bearbeitung:</u> bis zum 20.11.2003/18:00 Uhr
Alle Dateien sollen im Verzeichnis `/proj/i4mw/`*`loginname`*`/aufgabe2/` abgelegt werden.
Die Bearbeitung ist in 2er Gruppen möglich.