Verteilte Systeme

Übung A1

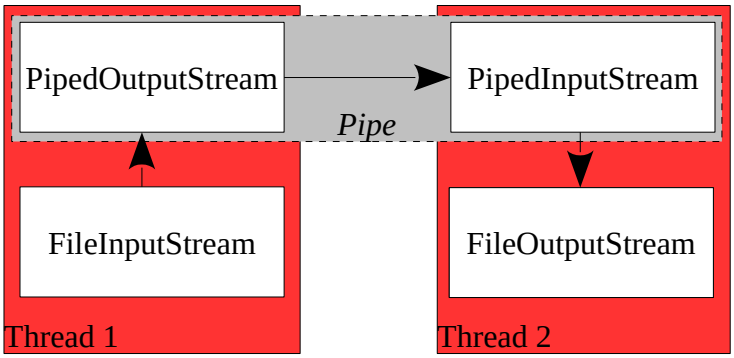
Sommersemester 2020

Sascha Baumeister

Diese Übung adressiert Grundlagen zum File-I/O (Java NIO.2) und zu den MIMD Technologien Multi-Threading bzw. Multi-Programming.

1 Intraprozess-Kommunikation mittels Pipes

Die Klasse **`FileCopySingleThreaded`** realisiert ein Programm zum Kopieren von Dateien mittels des neuen Java NIO.2 Filesystem-APIs aus Java 1.7. Ersetzt den bisherigen direkten Datentransfer durch zwei Transporter-Threads welche die Daten zum einen von der Quell-Datei in einen `PipedInputStream`, und zum zweiten aus einem `PipedOutputStream` in die Ziel-Datei kopieren.



Kopiert dazu die Klasse **`FileCopySingleThreaded`** nach **`FileCopyMultiThreaded`**, und geht folgendermaßen vor:

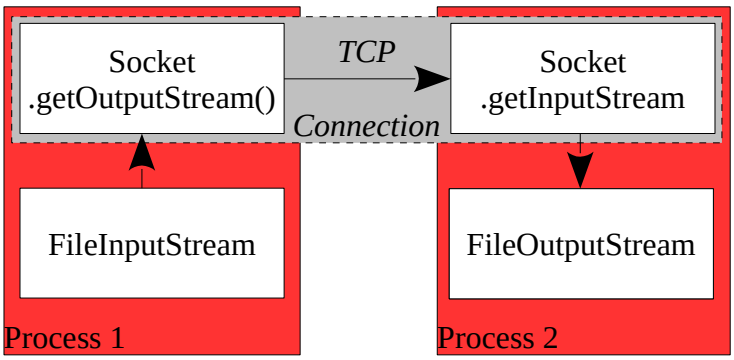
Erzeugt zwei Transporter Instanzen als Instanzen von Subtypen des Interfaces `Runnable`. Erstellt dazu entweder

- zwei Lambda-Expressions welche `Runnable` implementieren
- zwei Instanzen anonymer Subklassen welche `Runnable` passend implementieren
- eine statische innere Klasse Transporter welche `Runnable` implementiert, mit den Instanzvariablen inputStream und outputStream, vom Typ `InputStream` und `OutputStream`. Dazu einen passenden Konstruktor, welcher genutzt wird um die Instanzvariablen zu setzen.

Ändert die main()-Methode der Klasse `FileCopyMultiThreaded` nun so ab dass sie zwei Transporter-Instanzen in zwei Threads (i.e. asynchron, mittels `thread.start()`), sowie eine Pipe zum Datentransfer verwendet. Beachtet dabei dass das Schließen des einen Endes einer Pipe automatisch auch das andere Ende schließt! Eure Lösung sollte ohne Maßnahme zur Synchronisierung nach dem einmaligen Kopieren des Datei-Inhalts terminieren.

Ein Skalierungserfolg ist durch diese Maßnahme auf normalen Systemen nicht zu erwarten, da nun jedes Byte der zu kopierenden Datei zweimal gelesen und geschrieben werden muss. Auf Systemen mit mehreren Festplatten dagegen kann dies (bei geeignet großer Wahl der Pipe-Puffergröße) ganz anders aussehen: Falls Quelle und Ziel auf verschiedenen Festplatten beheimatet sind, kann hier die Reduzierung von Bewegungen der mechanischen Lese/Schreib-Köpfe leicht den zusätzlichen Aufwand beim RAM-Datentransfer überwiegen!

2 Interprozess-Kommunikation mittels TCP-Sockets



Kopiert dazu die Klasse **`FileCopyMultiThreaded`** nach `FileCopySend` sowie `FileCopyReceive`, und geht folgendermaßen vor:

- Ändert die main()-Methode der Klasse `FileCopySend` so ab dass nur noch der Lese-Teil des Kopiervorgangs ausgeführt wird. Das Argument soll dabei der Quell-Pfad sowie der zu öffnende TCP-Port sein. Einen Server-Port öffnet ihr mittels `ServerSocket service = new ServerSocket(port)`, und eine TCP-Verbindung akzeptiert ihr mittels `Socket connection = service.accept()`. Try-with-resources Blöcke helfen Euch dabei die Ressourcen wieder korrekt zu schließen. Das Programm soll nach dem einmaligen Versenden des Datei-Inhalts terminieren.
- Ändert die main()-Methode der Klasse `FileCopyReceive` so ab dass nur noch der Schreib-Teil des Kopiervorgangs ausgeführt wird. Das Argument soll dabei der Ziel-Pfad sowie die zu öffnende Socket-Adresse (TCP-Host und Port) sein. Die Socket-Adresse könnt ihr entweder zweiteilig übernehmen, oder mittels der Helper-Methode `InetAddress.toSocketAddress()` aus einem String parsen lassen (dazu muss die sb-toolbox Bibliothek konfiguriert sein). Die TCP-Verbindung initiiert ihr dann mittels Konstruktor `Socket connection = new Socket(host, port)`. Das Programm soll nach dem einmaligen Empfangen des Datei-Inhalts terminieren.
- Testet Eure Implementierung indem ihr eine Datei von einem Computer zu einem anderen übertragt. Eure Lösung sollte ohne Maßnahme zur Synchronisierung nach dem einmaligen Kopieren des Datei-Inhalts terminieren. Beachtet dass TCP-Verbindungen durch ihren hohen Abstraktionsgrad (verglichen mit UDP) als bidirektionale Interprozess-Pipes genutzt werden können.