HOCHSCHULE KONSTANZ TECHNIK, WIRTSCHAFT UND GESTALTUNG
UNIVERSITY OF APPLIED SCIENCES

Transparentes Austauschen von Socket-Verbindungen in Java

Ellen Wieland

Konstanz, 31. Januar 2011

BACHELORARBEIT

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

an der

Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Fakultät Informatik
Studiengang Wirtschaftsinformatik

Thema: **Transparentes Austauschen von Socket-Verbindungen in Java**

Bachelorkandidat: Ellen Wieland, Ringstraße 16, 78465 Konstanz

1. Prüfer: Prof. Dr.-Ing. Oliver Haase
2. Prüfer: Prof. Dr.-Ing. Jürgen Wäsch

Ausgabedatum: 01. November 2010
Abgabedatum: 31. Januar 2011

Zusammenfassung (Abstract)

Thema:	Transparentes Austauschen von Socket-Verbindungen in Java
Bachelorkandidat:	Ellen Wieland
Firma:	HTWG
Betreuer:	Prof. Dr.-Ing. Oliver Haase Prof. Dr.-Ing. Jürgen Wäsch
Abgabedatum:	31. Januar 2011
Schlagworte:	Java, TCP Sockets, Socket Library Erweiterung, Half-Close

Um das Problem des Verbindungsaufbaus zwischen zwei Peers, die sich in verschiedenen privaten Netzwerken hinter NATs befinden, zu lösen, existieren verschiedene Lösungen. Diese unterscheiden sich jedoch sehr bezüglich ihrer Skalierbarkeit und der Zeit, die für den Verbindungsaufbau benötigt wird. Die Technik des Relaying ist recht einfach zu realisieren, aber jeder neue Peer verursacht neue Last auf dem Server. Im Gegensatz dazu kann über Hole-Punching eine „günstige“, direkte Verbindung zwischen zwei Peers aufgebaut werden. Allerdings nimmt der Verbindungsaufbau aufgrund der Komplexität des Verfahrens viel Zeit in Anspruch.

In dieser Arbeit wird nun eine Erweiterung der Java Socket Library vorgestellt, die es ermöglichen soll, dass zuerst eine „teure“ Verbindung über eine Technik wie Relaying aufgebaut und dann, sobald eine „günstigere“ Verbindung, beispielsweise durch Hole-Punching, verfügbar ist, im Hintergrund ausgetauscht wird. Dieser Prozess soll für die Anwendung, welche über die Socket-Verbindung Daten verschickt, vollkommen transparent sein. Den Austausch der Verbindung nimmt eine übergeordnete Kontrollinstanz vor. Für die Anwendung ist daher nicht ersichtlich, ob sie mit der Erweiterung oder der Original-Implementierung von Java arbeitet.

Ehrenwörtliche Erklärung

Hiermit erkläre ich *Ellen Wieland, geboren am 05.12.1986 in Konstanz*, dass ich

(1) meine Bachelorarbeit mit dem Titel

Transparentes Austauschen von Socket-Verbindungen in Java

bei der HTWG unter Anleitung von Prof. Dr.-Ing. Oliver Haase selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 31. Januar 2011

(Unterschrift)

Inhaltsverzeichnis

Kapitel 1

Einleitung

In immer mehr Anwendungsgebieten kommt Peer-to-Peer (P2P) Technologie zum Einsatz. Nicht nur File Sharing, Instant Messaging, Grid Computing oder IP-Telefonie sind ohne P2P undenkbar, auch im Bereich von verteilten Suchmaschinen, Audio- und Video-Streaming oder E-Commerce können durch den Einsatz von P2P Technologien die Lastverteilung des Datenverkehrs verbessert, die Ausfallsicherheit des Gesamtsystems erhöht und, auf Seiten der Anbieter von Daten, Kosten gespart werden.

Eines der größten Probleme von P2P-Anwendungen ist allerdings, dass viele Peers keine exklusive öffentliche IP-Adresse mehr besitzen, sondern sich in einem privaten Adressraum hinter Routern mit NAT-Funktionalität (Network Address Translation) befinden. Der Router übersetzt für ausgehende Verbindungen die Kombination aus öffentlicher IP-Adresse und Portnummer in private IP-Adresse und Portnummer. Aus Sicherheitsgründen lassen die meisten NATs keine eingehenden Verbindungen zu. Das heißt, der NAT verwirft alle ankommenden Pakete, es sei denn, er kann sie einer Verbindung zuordnen, die aus dem privaten Netzwerk heraus initiiert wurde. Daher ist der Aufbau einer direkten Verbindung zwischen zwei Peers, die sich in zwei verschiedenen privaten Netzwerken befinden, schwierig.

Um dieses Problem zu umgehen, werden verschiedene Techniken eingesetzt. Eine Möglichkeit besteht darin, die Kommunikation über einen zentralen Relay-Server mit öffentlicher IP-Adresse zu leiten. Der Server erhält Daten von einem Peer und leitet sie an den anderen Peer weiter. Das Problem beim Relaying besteht darin, dass die Skalierbarkeit bezüglich der Größe begrenzt ist. Jeder neue Peer verbraucht Serverressourcen und Netzwerk Bandbreite. Ein großer Vorteil ist allerdings, dass diese Technik immer eingesetzt werden kann, da sie dem Prinzip der klassischen Client-Server-Verbindung entspricht. [Epp05]

Eine andere Möglichkeit ist das „Hole Punching“. Dabei wird ein Vermittlungsserver dazu benutzt, die öffentlichen IP-Adressen und Portnummern der Peers auszutauschen. Danach versuchen die beiden Peers simultan eine Verbindung zueinander aufzubauen. Da die NATs dann annehmen, dass die eingehende Verbindung eine Antwort auf die eigene Verbindungsanfrage ist, kann eine direkte Verbindung zwischen den beiden Peers aufgebaut werden. Der Vorteil dieser Technik besteht in der guten Skalierbarkeit, da der Vermittlungsserver

nur zum Austausch der IP-Adressen benötigt wird. Ein Nachteil ist, dass der Verbindungsaufbau komplex ist und daher einige Zeit in Anspruch nimmt bevor eine Verbindung aufgebaut ist. [FSK05]

1.1 Motivation

Im Rahmen einer Masterarbeit zum Thema TCP Hole Punching in Java kam die Frage auf, wie es realisiert werden kann, dass zuerst eine „teure“ Verbindung über eine Technik wie beispielsweise Relaying aufgebaut wird und später, sobald eine Verbindung über Hole Punching vorhanden ist, die alte „teure“ Verbindung gegen eine neue „günstigere“ Verbindung ausgetauscht wird. Das Austauschen der zugrundeliegenden Verbindung sollte dabei für die Anwendung, die über die Sockets kommuniziert, vollkommen transparent sein. Dadurch soll erreicht werden, dass die Peers so schnell wie möglich miteinander kommunizieren können und gleichzeitig auf lange Sicht eine möglichst „günstige“ Verbindung dafür benutzen.

1.2 Zielsetzung

In dieser Bachelorarbeit soll eine Erweiterung der Socket Library in Java erstellt werden, die es ermöglicht im Hintergrund die Socket-Verbindung auszutauschen. Dies soll vollkommen transparent für die Anwendung, welche die Sockets verwendet, geschehen. Der Fokus der Arbeit liegt dabei auf der Realisierung des Vorgehens für TCP, also für die `Socket`-Klasse, da TCP in letzter Zeit auch in Anwendungsszenarien, die traditionell UDP nutzen, immer mehr Bedeutung zukommt. Die Umsetzung für UDP und welche Auswirkung die Verwendung von SSL auf das Vorgehen hat, könnte in weiterführenden Arbeiten ermittelt werden.

1.3 Aufbau

Diese Arbeit ist in fünf Kapitel gegliedert. Im zweiten Kapitel werden die theoretischen Grundlagen für die Arbeit vorgestellt, wobei davon ausgegangen wird, dass der Leser mit Socket-Programmierung in Java bereits vertraut ist. Im dritten Kapitel wird die Implementierung der Aufgabenstellung in Java vorgestellt. Das vierte Kapitel bietet einen Überblick über die Validierung der im vorherigen Kapitel vorgestellten Lösung. Abschließend wird im fünften Kapitel eine Zusammenfassung und Bewertung vorgenommen, sowie eine Vorstellung weiterführender Forschungsthemen gegeben.

Kapitel 2

Theoretische Grundlagen

In diesem Kapitel werden wichtige Konzepte eingeführt, welche die Grundlage für diese Arbeit bilden. Als erstes wird das Verfahren des Half-Close, welches über das Schließen des Outputs funktioniert, vorgestellt. Dann wird auf die Möglichkeit eingegangen, durch sogenannte Socket Options das Verhalten von Socket-Verbindungen zu beeinflussen. Abschließend wird erläutert, wo und wie in der Java Socket Library angesetzt werden kann, um das Ziel dieser Arbeit, eine Erweiterung der Socket Library zum transparenten Austauschen von Socket-Verbindungen, zu realisieren.

2.1 Output shutdown (Half-Close)

Um eine bestehende Socket-Verbindung zu schließen, bietet die Klasse `java.net.Socket` nicht nur die Methode `close()`, welche die Verbindung in beide Richtungen gleichzeitig schließt, sondern auch zwei Methoden, mit welchen es möglich ist, den `InputStream` und den `OutputStream` unabhängig von einander zu beenden, `shutdownInput()` und `shutdownOutput()`.

Die `shutdownOutput()`-Methode wird oft in fortgeschrittener Netzwerk-Programmierung genutzt. Mit ihr ist es möglich, dem anderen Ende eine EOF-Meldung (End of File) zu schicken und trotzdem gleichzeitig noch von dem `InputStream` zu lesen.

Lokal hat der Aufruf von `shutdownOutput()` den Effekt, dass von dem `InputStream` des zugrundeliegenden Sockets weiterhin ganz normal gelesen werden kann, während weitere Versuche Daten auf den `OutputStream` zu schreiben eine `IOException` auslösen. Außerdem wird nach der erfolgreichen Übertragung aller restlichen Daten aus der Warteschlange der TCP Verbindungsabbau dieser Seite eingeleitet: es wird ein FIN-Paket geschickt, dessen Erhalt von der Gegenseite mit einem ACK-Paket bestätigt wird. Auf der Gegenseite der Verbindung ist das Verhalten gerade umgekehrt: es kann weiterhin auf den `OutputStream` geschrieben werden, während beim Lesen vom `InputStream` ein EOF beobachtet wird. Je nachdem welche Methode aufgerufen wurde, ist dies entweder ein `read count` von `-1` oder eine `EOFException`.

Mit der `shutdownOutput()`-Methode lässt sich deshalb vor dem Schließen einer Verbindung eine Synchronisation der Kommunikationspartner realisieren.

Bevor die Verbindung geschlossen wird, rufen dafür beide Kommunikationspartner `shutdownOutput()` auf und warten dann in einem blockierenden `read()` auf eine EOF-Meldung, wie in Abbildung 2.1 dargestellt. Wenn eine Seite die EOF-Meldung erhält, ist sichergestellt, dass die andere Seite `shutdownOutput()` aufgerufen hat. [Pit06]

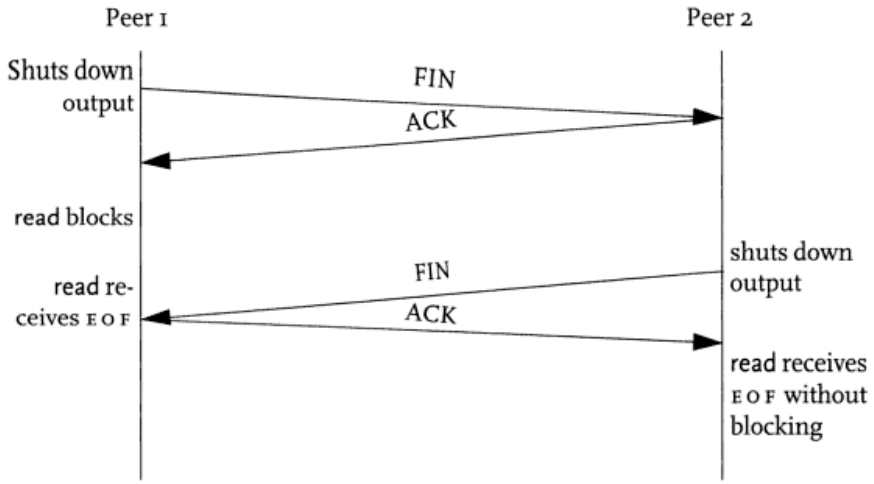

Abbildung 2.1: Synchronisation zweier Peers mithilfe von `shutdownOutput()` entnommen aus [Pit06]

2.2 Socket Options

Über Socket Options lassen sich fortgeschrittene Funktionen des TCP-Protokolls steuern. In Java können diese Optionen über Methoden in den Klassen `java.net.Socket` oder `java.net.ServerSocket` beeinflusst werden. Einige dieser Optionen, wie beispielsweise die Größe des Receive-Buffers, können dabei nur gesetzt werden, bevor eine Verbindung aufgebaut worden ist. Die meisten lassen sich aber jederzeit ändern.

Wichtige Optionen sind vor allem Socket Timeouts, über die festgelegt werden kann, wie lange in der `accept()`-, der `read()`- oder der `receive()`-Methode auf Daten gewartet werden soll und die Größen von Send- und Receive-Buffer. Außerdem kann über die Methode `setTcpNoDelay()` festgelegt werden, dass Daten sofort gesendet werden, anstatt den Versand hinauszuzögern und größere Pakete zu versenden. Die Methode `setSoLinger()` legt fest, wie sich die zugrundeliegende TCP-Implementierung verhält, wenn eine Socket-Verbindung geschlossen wird: Normalerweise wird nach dem Schließen der Verbindung der Socket nicht sofort zerstört, sondern kommt, nachdem alle Daten übertragen wurden, in den TIME-WAIT-Status. Solange der Socket sich im TIME-WAIT-Status befindet, kann der Port nicht für eine neue Verbindung verwendet werden. Dies dient dazu, dass verspätete Datensegmente nicht fälschlicherweise an eine neue Verbindung weitergereicht werden. Mit der Methode `setKeepalive()` wird festgelegt, dass durch das Versenden von Test-Nachrichten geprüft wird,

ob die Verbindung auf der Gegenseite noch aktiv ist. Damit „urgent data“, welche über die Methode `sendUrgentData()` verschickt wurde, empfangen werden kann, muss durch `setOOBInline()` festgelegt werden, dass diese Daten „inline“ mit den normalen Daten empfangen werden. [CD08]

Weitere Einstellungen zu der Art von Service, über welchen die Daten verschickt werden, können mit den Methoden `setTrafficClass()` oder `setPerformancePreferences()` vorgenommen werden. [Pit06]

2.3 Erweiterung der Java Socket Library

In Java repräsentiert die Klasse `Socket` aus dem `java.net`-Paket den clientseitigen Endpunkt einer TCP-Verbindung. Um das transparente Austauschen einer Socket-Verbindung zu ermöglichen, muss eine eigene Implementierung dieser `Socket`-Klasse erstellt werden.

Da über die Methoden `getInputStream()` und `getOutputStream()` externe Referenzen auf die mit dem Socket assoziierten Streams für Input und Output erzeugt werden können, muss auch jeweils für die Klassen `InputStream` und `OutputStream` eine eigene Implementierung bereitgestellt werden, welche für den internen und nach außen hin transparenten Austausch des jeweiligen Streams sorgt. Diese Beziehungen sind im folgenden Klassendiagramm aus Abbildung 2.2 dargestellt. Das nächste Kapitel befasst sich mit der Implementierung dieser Komponenten.

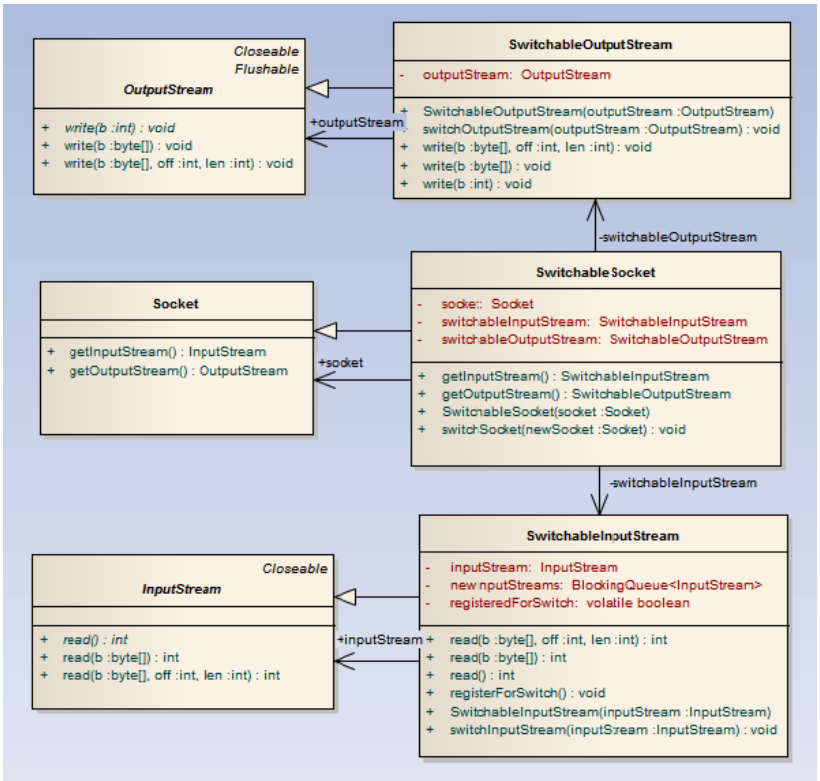

Abbildung 2.2: Klassendiagramm der Socket Library Erweiterung

Kapitel 3

Implementierung

In diesem Kapitel wird die Implementierung der vorgestellten Komponenten beschrieben. Für detailliertere Einblicke kann der Quellcode herangezogen werden.

3.1 `SwitchableSocket` mit `shutdownOutput()`

Dieser Ansatz nutzt das im vorherigen Kapitel vorgestellte Verfahren des Half-Close. Zusammengefasst funktioniert dies folgendermaßen: Wenn die Verbindung getauscht werden soll, wird zuerst der `OutputStream` getauscht, beide Seiten schicken durch den Aufruf von `shutdownOutput()` auf der alten Verbindung eine EOF-Meldung und sobald in einer der `read()`-Methoden `-1` gelesen wird, wird auch der `InputStream` getauscht.

3.1.1 Die Klasse `SwitchableSocket`

Die Klasse `SwitchableSocket` bildet das Kernstück der Implementierung. Sie ist eine Unterklasse der Klasse `java.net.Socket` und kann somit für eine Anwendung transparent als Socket-Implementierung verwendet werden.

Intern hält die Klasse `SwitchableSocket` eine Referenz auf das `Socket`-Objekt der aktuellen Verbindung, an welches alle von der Oberklasse geerbten Methoden (außer die Methoden zum Erhalt einer Referenz auf den `Input-` und den `OutputStream` des Sockets) weitergeleitet werden. Obwohl in dem Szenario, welches dieser Arbeit zugrunde liegt, immer bereits verbundene Sockets verwendet werden, und es deshalb nicht möglich ist, auf einem `SwitchableSocket`-Objekt beispielsweise eine der `connect()`-Methoden aufzurufen, wurde davon abgesehen explizit eine `Exception` zu werfen, dass diese Methode nicht unterstützt wird, weil es in einem anderen Anwendungsfall durchaus sinnvoll sein kann mit unverbundenen Sockets zu arbeiten.

Neben einer Referenz auf ein `Socket`-Objekt enthält die Klasse `SwitchableSocket` jeweils eine Referenz auf einen `SwitchableInputStream` und einen `SwitchableOutputStream`. Diese Referenzen werden auch von den Methoden `getInputStream()` und `getOutputStream()` zurückgeliefert. Die beiden Klassen werden in den nächsten beiden Unterkapiteln im Detail vorgestellt.

Die Methode, in welcher der Tausch der Socket-Verbindung vorgenommen wird, ist die Methode `switchSocket()`. Als erstes wird dort die Methode `registerForSwitch()` an dem internen `SwitchableInputStream` aufgerufen, um durch das Setzen des Flags `registeredForSwitch` auf `true` dem `SwitchableInputStream` zu signalisieren, dass es eine neue Verbindung gibt und der `InputStream` ausgetauscht werden soll, sobald der neue `InputStream` verfügbar ist und in einer der `read()`-Methoden eine EOF-Meldung gelesen wird. Um zu gewährleisten, dass beide Seiten bereit sind die neue Verbindung zu verwenden, das heißt, dass sich beide Seiten in der `switchSocket()`-Methode befinden, wird dann eine Synchronisation der beiden Kommunikationspartner über die neue Verbindung vorgenommen, wie in Listing 3.1 zu sehen ist. Die Seite, welche zuerst die `write()`-Methode aufgerufen hat, blockiert in der darauf folgenden `read()`-Methode, bis auch die andere Seite bereit ist, die neue Verbindung zu verwenden und an dieser Stelle in der `switchSocket()`-Methode angekommen ist.

```
1  // synchronization: send test message over new connection to
       test, if the other side is ready to use new connection
2  newSocket.getOutputStream().write(1);
3  // try to receive data from new connection
4  newSocket.getInputStream().read();
```

Listing 3.1: Synchronisation der Peers in `switchSocket()`-Methode

Nach der Synchronisation werden die Methoden `switchInputStream()` und `switchOutputStream()` aufgerufen, die den Tausch der jeweiligen internen Streams initialisieren. Sobald die Streams getauscht wurden, kann die neue Verbindung verwendet werden. Dies wird durch den Aufruf der Methode `shutdownOutput()`, welche den „alten“ `OutputStream` schließt und eine EOF-Meldung auf der Gegenseite auslöst, dem Kommunikationspartner mitgeteilt.

Nach der Initialisierung des Tauschprozesses der `Input-` und `OutputStreams` wird schließlich versucht, die Einstellungen, welche nach dem Verbinden eines Sockets gesetzt werden können, von der alten Verbindung auf die neue Verbindung zu übertragen. Je nach Betriebssystem und zugrundeliegender Socket-Implementierung werden aber nicht alle Optionen unterstützt [Har04]. Wenn versucht wird, eine Option zu setzen, die nicht unterstützt wird, wird eine `SocketException` geworfen.

Damit der Tausch der Socket-Verbindung für die Anwendung transparent bleibt, ist das Übertragen jeder Option in einen `try-catch`-Block eingebettet. Um möglichst alle Optionen zu übertragen und nicht nach dem ersten Fehler abzubrechen, ist jede `set()`-Operation wiederum von einem eigenen `try-catch`-Block umgeben wie in Listing 3.2 dargestellt. Im `catch`-Block wird des Weiteren nichts gemacht, er dient nur der Erhaltung der Transparenz.

```
  try {
      newSocket.setKeepAlive(socket.getKeepAlive());
  } catch (Exception e) {
      // DO NOTHING
  }
  try {
      newSocket.setSendBufferSize(socket.getSendBufferSize());
  } catch (Exception e) {
      // DO NOTHING
  } ...
```

Listing 3.2: Übertragen der Einstellungen auf die neue Socket-Verbindung

Als letztes wird dann in der `switchSocket()`-Methode die interne Referenz auf die neue Socket-Verbindung gesetzt.

3.1.2 Die Klasse `SwitchableInputStream`

Die Klasse `SwitchableInputStream` ist eine Wrapper-Klasse für den mit einer Socket-Verbindung assoziierten `InputStream`. Sie ist eine Unterklasse der Klasse `InputStream` und hält intern eine Referenz auf das `InputStream`-Objekt, das zu der aktuellen Socket-Verbindung gehört. Alle öffentlichen von `InputStream` geerbten Methoden werden an dieses Objekt weitergeleitet, wobei in den `read()`-Methoden, wenn eine EOF-Meldung beobachtet wird, zusätzlich getestet wird, ob das interne `InputStream`-Objekt getauscht werden soll. Dies wurde wie in Listing 3.3 gezeigt realisiert. Wenn `registeredForSwitch` = `true` ist, blockiert der Aufruf von `newInputStreams.take()` bis ein Eintrag in der Queue vorhanden ist. Nach dem Tausch wird das Flag dann wieder auf `false` gesetzt.

```
public int read() throws IOException {
  int data = 0;
  if ((data = inputStream.read()) == -1) {
    // test if EOF was caused because other side wants to switch
         the connection
    if (registeredForSwitch) {
      try {
        inputStream = newInputStreams.take();
        registeredForSwitch = false;
      } catch (InterruptedException e) {
        e.printStackTrace();
      }
      return this.read();
    // not caused by switch connection
    } else {
      return -1;
    }
  } else {
    return data;
  }
}
```

Listing 3.3: `read()`-Methode aus `SwitchableInputStream`

Das Flag `registeredForSwitch` wird auf `true` gesetzt, wenn die Methode `registerForSwitch()` aufgerufen wird und zeigt an, dass in einer der `read()`-Methoden die Referenz auf das alte `InputStream`-Objekt gegen ein neues `InputStream`-Objekt aus der `BlockingQueue` `newInputStreams` getauscht werden soll. Als Datentyp für die Liste der neuen `InputStreams` wurde eine `BlockingQueue` gewählt, weil sie threadsicher ist.

Dass zuerst das Flag `registeredForSwitch` gesetzt wird und nicht gleich vor der Synchronisation in der `switchSocket()`-Methode in der Klasse `SwitchableSocket` das neue `InputStream`-Objekt in die Liste der neuen `InputStreams` eingefügt wird, liegt daran, dass bei der Synchronisation ebenfalls Daten über die neue Verbindung geschickt werden. Je nachdem wie der Scheduler zwischen den verschiedenen Threads, die in einer `read()`-Methode auf Daten warten, umschaltet, könnte es passieren, dass die Daten, welche ausschließlich zur Synchronisation innerhalb der `switchSocket()`-Methode verschickt wurden, von dem „falschen“ Thread gelesen werden, wenn der neue `InputStream` schon vor der Synchronisation verfügbar ist.

Eine Liste von neuen `InputStreams` und nicht nur eine Variable `newInputStream` gibt es deshalb, weil der Austausch des Streams immer erst in einer der `read()`-Methoden erfolgt. Bei kurz aufeinander folgenden Aufrufen der `switchSocket()`-Methode, zwischen denen kein Aufruf einer `read()`-Methode erfolgt, würde bei der Verwendung einer einzelnen Variablen niemals von einer Verbindung gelesen werden, weil die Variable bei jedem Aufruf der `switchSocket()`-Methode überschrieben würde.

3.1.3 Die Klasse `SwitchableOutputStream`

Die Klasse `SwitchableOutputStream` ist die entsprechende Wrapper-Klasse für den `OutputStream`, der zu der aktiven Socket-Verbindung gehört. Die von der Oberklasse von `OutputStream` geerbten Methoden werden alle an die interne Referenz auf den `OutputStream` des Sockets weitergeleitet.

In der zusätzlichen Methode `switchOutputStream()` wird zuerst auf dem alten Stream die Methode `flush()` aufgerufen und dann die interne Referenz ausgetauscht. Wichtig ist außerdem, dass alle Methoden von `SwitchableOutputStream` `synchronized` sind, damit nicht durch einen unpassenden Context-Switch des Schedulers vor dem Tausch der internen Referenz auf den `OutputStream` noch auf den alten `OutputStream` geschrieben wird, obwohl er schon mit `shutdownOutput()` geschlossen wurde.

3.1.4 Ablauf

Im Folgenden werden anhand von UML-Sequenzdiagrammen die Abläufe beim Austauschen von `Input-` und `OutputStream` verdeutlicht.

Das Sequenzdiagramm in Abbildung 3.1 zeigt die Verwendung von `SwitchableSockets` in einer Anwendung und vereinfacht wie der Tausch des `InputStreams` funktioniert. Initialisiert wird der Tausch der Socket-Verbindung dabei von einer übergeordneten Kontrollinstanz.

Wie in dem Diagramm deutlich wird, ist der gesamte Austausch-Prozess für die Anwendung transparent: Anfangs wird über die Methode `getInputStream()` eine Referenz auf den `SwitchableInputStream` geholt, dann wird an diesem `InputStream` die Methode `read()` aufgerufen. Die Klasse `SwitchableSocket` kann also von der Anwendung genau so verwendet werden, wie die Standard-Socket-Implementierung `java.net.Socket`.

Im Hintergrund erzeugt die Kontrollinstanz beispielsweise über TCP-Hole-Punching eine neue Socket-Verbindung und startet den Austauschprozess. Über den Aufruf der Methode `registerForSwitch()` wird dabei dem `SwitchableInputStream` signalisiert, dass er beim nächsten Aufruf der `read()`-Methode und, nachdem durch die Methode `switchInputStream()` die Referenz auf den neuen `InputStream` an den `SwitchableInputStream` übertragen wurde, den zugrundeliegenden `InputStream` austauscht, sobald er eine EOF-Meldung erhält.

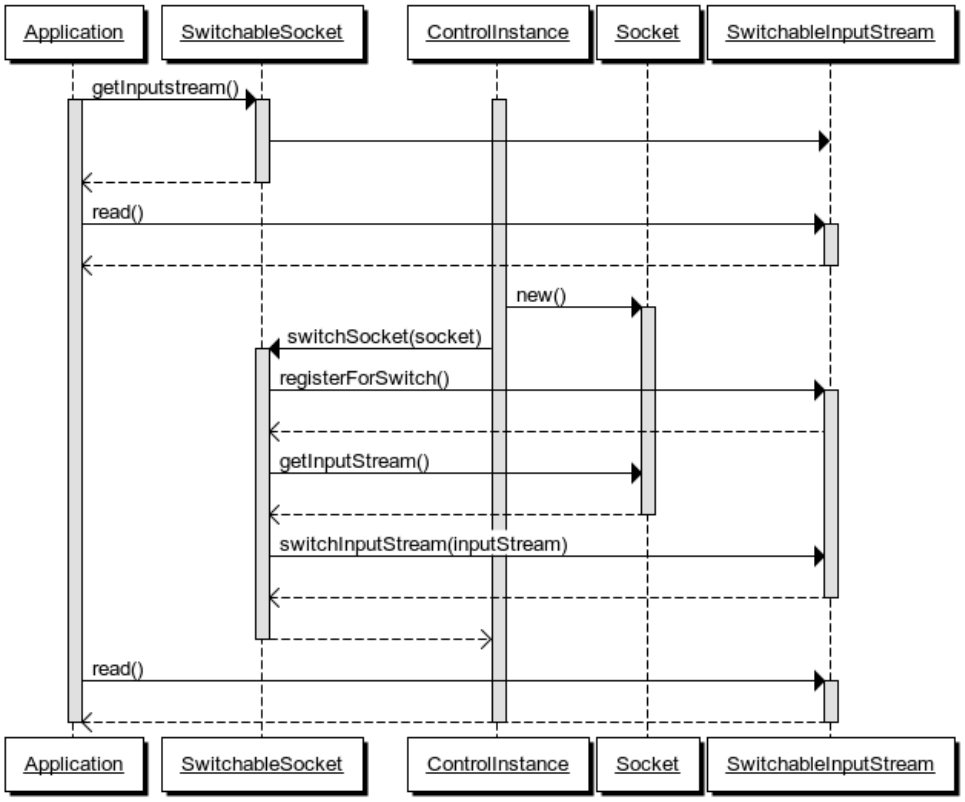

Abbildung 3.1: Ablauf des Austauschen des `InputStreams`

Analog dazu funktioniert der Austauschprozess des `OutputStreams` wie in Abbildung 3.2 dargestellt. Allerdings ist der Prozess weniger komplex, da der Tausch des zugrundeliegenden `OutputStreams` direkt in der Methode `switchOutputStream()` erfolgt. Der nächste Aufruf einer `write()`-Methode schreibt die Daten dann schon auf den neuen `OutputStream`, während beim nächsten Aufruf einer `read()`-Methode bis zum Erhalt einer EOF-Meldung noch von dem alten `InputStream` gelesen wird.

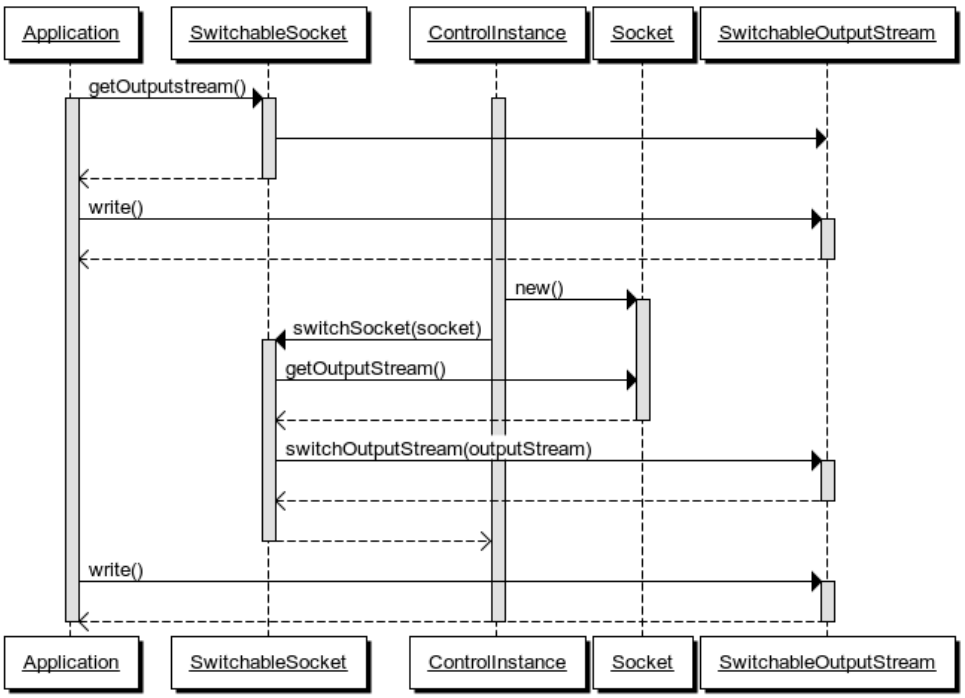

Abbildung 3.2: Ablauf des Austauschen des `OutputStreams`

3.2 `SwitchableSocket` unter Verwendung von Zählern

Da sich während des Verlaufs der Arbeit durch das Testen der vorgestellten Implementierung herausgestellt hat, dass unter bestimmten äußeren Umständen unter Windows der erste Ansatz, welcher das Verfahren des Half-Close nutzt, nicht immer zuverlässig funktioniert, und außerdem das Konzept, welches dem ersten Ansatz zu Grunde liegt, nicht anwendbar ist, wenn `SSLSockets` (Secure Socket Layer Sockets) verwendet werden sollen, wurde über einen alternativen Ansatz nachgedacht. Dass der erste Ansatz mit SSL nicht verwendet werden kann, liegt daran, dass die Klasse `SSLSocket` die Methode `shutdownOutput()` nicht unterstützt, da das SSL Protokoll ein Half-Close nicht erlaubt [Pit06].

Dieser neue Ansatz setzt auf die Verwendung von Zählern, um zu ermitteln, ob alle Daten von der alten Verbindung gelesen wurden, bevor die Socket-Verbindung ausgetauscht wird. Es wird also jedes Mal, wenn etwas auf den `SwitchableOutputStream` geschrieben wird, ein interner Zähler `numberOfBytesSent` um die Anzahl an geschriebenen Bytes erhöht. Wenn nun die Verbindung ausgetauscht werden soll, wird bei der Synchronisation der Kommunikationspartner nicht ein beliebiger Wert über die neue Verbindung geschickt, sondern der Wert des Zählers wird an die Gegenseite übermittelt. Im `SwitchableInputStream` gibt es ebenfalls einen internen Zähler `numberOfBytesReceived`. Dieser wird bei jedem Aufruf einer der `read()`-Methoden um die Anzahl der bereits gelesenen Bytes erhöht. Bei der Synchronisation der Kommunikationspartner in der `switchSocket()`-Methode wird dann das Attribut `numberOfBytesToRead` auf den Wert des auf der Ge-

genseite geführten Zählers `numberOfBytesSent` gesetzt. Ob die bestehende Verbindung gegen die neue Verbindung ausgetauscht werden soll, beziehungsweise, ob alle Daten von der „alten“ Verbindung gelesen wurden und auch der `InputStream` getauscht werden kann, wird in den `read()`-Methoden durch den Vergleich der Variablen `numberOfBytesReceived` und `numberOfBytesToRead` ermittelt. Wenn sie übereinstimmen, wird der `InputStream` getauscht.

Die größte Schwierigkeit bei der Implementierung dieses Ansatzes besteht darin, dass eine Möglichkeit gefunden werden muss, um aus den blockierten `read()`-Methoden zurückzukommen, den Austausch des `InputStreams` einzuleiten und danach die alte Verbindung zu schließen, damit die nicht mehr benötigten Verbindungen nicht weiter bestehen.

Mehrere Möglichkeiten zur Lösung des Problems wurden angedacht: Wenn am Ende der `switchSocket()`-Methode eine zusätzliche Nachricht über die alte Verbindung geschickt würde, könnte in einer der `read()`-Methoden die Prüfung erfolgen, ob alle Daten gelesen wurden. Allerdings ist nicht gewährleistet, dass bei einem Aufruf einer der `read()`-Methoden genau die Daten gelesen werden, die in einem einzelnen `write()` geschrieben wurden. In einem Aufruf einer der `read()`-Methoden können die Daten aus mehreren Aufrufen von `write()` gelesen werden. Das Problem dabei ist nun, dass sich die zusätzliche Nachricht mit den wirklichen Daten, die von der Anwendung versendet werden, vermischen könnte. Eine andere Möglichkeit könnte sein, dass am Ende der `switchSocket()`-Methode der „alte“ Socket und somit auch die mit der Verbindung assoziierten Streams geschlossen werden. Dadurch wird auf der Gegenseite eine EOF-Meldung beobachtet und lokal wird in allen blockierten `read()`-Operationen eine `SocketException` geworfen. Allerdings muss jeweils unterschieden werden, ob die `SocketException` von der `switchSocket()`-Methode ausgelöst wurde oder ob wirklich ein Fehler aufgetreten ist.

Außerdem muss bei diesem Ansatz gewährleistet werden, dass nachdem der Wert des Zählers `numberOfBytesSent` zur Gegenseite übertragen wurde, keine Daten mehr auf den „alten“ `OutputStream` geschrieben werden, sondern erst nach dem Tausch auf den neuen Stream geschrieben wird.

Kapitel 4

Validierung

Da das strukturierte Testen von verteilten Anwendungen meist kompliziert ist, war auch die Validierung der realisierten Socket Library Erweiterung schwierig. Um auf verschiedenen Umgebungen eine vergleichbare, reproduzierbare und automatisierte Validierung durchführen zu können, wurden JUnit-Tests und eine Beispielanwendung erstellt.

4.1 Testumgebung

Weil es bei verteilten Anwendungen und besonders bei P2P-Systemen wichtig ist, dass sie auf verschiedenen Betriebssystemen funktionieren, wurden alle Komponenten auf folgenden Betriebssystemen getestet:

- Windows 7 Professional
- Windows XP Professional SP3
- Ubuntu 10.10 „Maverick Meerkat“
- Mac OS X 10.6.5

Dabei wurde in den verschiedensten Konstellationen getestet: Client und Server befinden sich auf demselben Rechner oder auf verschiedenen Rechnern mit verschiedenen Betriebssystemen.

4.2 Realisierte Tests

Zum Testen der in dieser Arbeit realisierten Erweiterung der Java Socket Library wurden sowohl Komponententests, als auch ein Systemtest erstellt. Für die Klassen `SwitchableInputStream` und `SwitchableOutputStream` wurden Komponententests entwickelt. Für das Kernstück der Anwendung, die Klasse `SwitchableSocket`, wurden keine Komponententests geschaffen, da die Methode `switchSocket()` nicht ohne ein komplexes Testsystem getestet werden kann. Deshalb wurde ein Systemtest entworfen, der die korrekte Funktionsweise der `switchSocket()`-Methode und das Zusammenspiel aller Komponenten prüft.

4.2.1 Komponententests

In den JUnit-Tests für `SwitchableInputStream` und `SwitchableOutputStream` werden die einzelnen Methoden getestet, welche die zusätzliche Funktionalität zum transparenten Austauschen der Streams realisieren. Die Methoden, welche nur unverändert an den internen Stream weitergeleitet werden, wurden nicht explizit getestet. Um die Änderungen an privaten Attributen beobachten zu können, wurde die Klasse `PrivateAccessor` erstellt. Über die Methode `getPrivateField()`, welche Reflexion nutzt um das gewünschte Attribut zu finden, ist der Zugriff auf beliebige Attribute möglich.

In dem `SwitchableInputStreamTest` werden die `read()`-Methoden, sowie die Methoden `registerForSwitch()` und `switchInputStream()` getestet. Der Test der `read()`-Methoden besteht darin, dass von einem Test-`String` gelesen und dann überprüft wird, dass die gelesenen Daten mit den Original-Daten übereinstimmen. Bei den beiden anderen Methoden wird sichergestellt, dass ihr Aufruf die Attribute des getesteten `SwitchableInputStream`-Objekt verändert. In dem `SwitchableOutputStreamTest` wird die Methode `switchOuputStream()` getestet. Dabei wird kontrolliert, dass das interne `OutputStream`-Objekt ausgetauscht wurde.

4.2.2 Systemtest

Um die Funktionsfähigkeit des Gesamtsystems zu testen, wurde eine Beispielanwendung entwickelt, welche aus einer Server- und einer Clientkomponente, sowie einer Kontrollinstanz und einem JUnit-Test besteht. Die Serverkomponente entspricht dabei einem einfachen ECHO-Server, welcher die Daten, die er erhält, ausgibt und unverändert an den Client zurückschickt, siehe Listing 4.1.

```
1  while ((recvMsgSize = in.read(buffer)) != -1) {
2    String message = new String(buffer).trim();
3    System.out.println(message.substring(0, recvMsgSize));
4    out.write(buffer, 0, recvMsgSize);
5    out.flush();
6  }
```

Listing 4.1: ECHO-Server für Systemtest

Die Clientkomponente verfügt über eine Methode `sendMessage()`. Diese Methode verschickt eine Nachricht des Typs String an den Server und liefert die vom Server erhaltene Antwort zurück. Die Kontrollinstanz erzeugt in unterschiedlichen Abständen eine neue Socket-Verbindung zum Server und initiiert den Tausch der Verbindung auf der Seite des Clients wie aus Listing 4.2 ersichtlich ist. Auf der Server-Seite wird die `switchSocket()`-Methode aufgerufen, sobald eine neue Verbindung akzeptiert wurde.

```
1  Thread.sleep(sleepTimeInMillis);
2  System.out.println("SWITCH");
3  Socket socket = new Socket(SERVER_ADDRESS, SERVER_PORT);
4  client.getSocket().switchSocket(socket);
```

Listing 4.2: Die Kontrollinstanz initiiert den Tausch der Verbindung

Der JUnit-Test `SwitchableSocketTest` ist kein klassischer Komponententest, sondern ein Systemtest. Er startet die benötigten Komponenten, also den Server, die Kontrollinstanz und den Client, und erzeugt in einer Schleife eine konfigurierbare Anzahl von Test-Nachrichten, die von dem Client an den Server geschickt werden. Eine Test-Nachricht besteht aus einer festen Test-Nachricht kombiniert mit dem aktuellen Zeitstempel. Erfolgskriterium für das Bestehen des Tests in der Methode `testCorrectDataExchange()` ist hierbei, dass die vom Client verschickten Nachrichten mit den vom Server erhaltenen Antworten übereinstimmen. Die Methode, welche den korrekten Datenaustausch testet ist in Listing 4.3 dargestellt.

```
1  @Test
2  public void testCorrectDataExchange() {
3    long startTime = System.currentTimeMillis();
4    for(int i = 0; i < NUMBER_OF_MESSAGES; i++) {
5      try {
6        String message = TEST_MESSAGE + System.currentTimeMillis
             ();
7        String answer = client.sendMessage(message);
8        assertEquals(message, answer);
9      } catch (Exception e) {
10       fail("The following Exception occured: " + e.getMessage
             ());
11     }
12   }
13   long endTime = System.currentTimeMillis();
14   System.out.println("The test took: " + (endTime - startTime)
         + " milliseconds.");
15 }
```

Listing 4.3: Systemtest mit JUnit

Über die Variablen `startTime` und `endTime` ist es darüber hinaus möglich die Dauer des Tests zu bestimmen. Dies kann dazu genutzt werden, Performance-Unterschiede zwischen den verschiedenen Plattformen zu ermittlen.

4.3 Testergebnisse

Die Komponententests liefen auf allen Testsystemen fehlerfrei. Bei dem Systemtest gab es nur unter Windows 7 teilweise Probleme.

Auf Windows 7 Systemen, auf denen gleichzeitig auch die Firewall-Software ZoneAlarm installiert war, kam es manchmal zu einer Deadlock-Situation. Das heißt, die EOF-Meldung, welche signalisiert, dass die Verbindung getauscht werden kann, kam bei der Gegenseite nicht an und somit wurde der zugrunde-

liegende `InputStream` nie getauscht.

Dieses fehlerhafte Verhalten konnte auch auf einer Virtual Machine reproduziert werden. Auf der Virtual Machine waren nur Windows 7, Eclipse, Java und ZoneAlarm installiert. Der Fehler trat dabei nur bei dem Test direkt nach dem Neustart auf. Da das Problem vor der Installation von ZoneAlarm, also als nur Windows 7, Eclipse und Java installiert waren, nicht bestand, scheint es in direktem Zusammenhang mit ZoneAlarm zu stehen. Eine Recherche zu Problemen mit Java TCP Sockets im Zusammenhang mit ZoneAlarm brachte allerdings keine Resultate.

Weil ZoneAlarm aber unter Umständen auf sehr vielen Systemen installiert ist, wurde über eine alternative Version der Socket Library Erweiterung nachgedacht, die nicht die Technik des Half-Close nutzt, sondern über Zähler ermittelt, wann alle Daten von der alten Verbindung gelesen wurden und sie gegen die neue Verbindung getauscht werden kann, wie im vorherigen Kapitel beschrieben.

Kapitel 5

Fazit

Zusammengefasst kann festgehalten werden, dass das Ziel erreicht wurde, eine Erweiterung der Java Socket Library zu erstellen, die es ermöglicht für die Anwendung transparent die Socket-Verbindung auszutauschen. Mit der in dieser Arbeit entwickelten Erweiterung ist es möglich, P2P-Anwendungen zu programmieren, die anfangs eine „teure“ Verbindung nutzen und im Hintergrund durch eine übergeordnete Kontrollinstanz versuchen beispielsweise über Hole-Punching eine „günstigere“ Verbindung aufzubauen. Sobald die „günstigere“ Verbindung besteht, kann über die Kontrollinstanz dann der Tausch der Verbindung initiiert werden. In Kombination mit Techniken wie etwa Hole-Punching bietet die realisierte Erweiterung der Java Socket Library somit die Grundlage für erweiterte Funktionalität und Kostenersparnisse bei P2P-Anwendungen.

Die Transparenz ist bei der vorgestellten Implementierung ebenfalls vollständig gegeben, da die erstellten Klassen dasselbe öffentliche Interface bereitstellen wie die Klassen aus den Paketen `java.net` und `java.io`. Wenn die übergeordnete Kontrollinstanz, welche für den Austausch der Verbindung sorgt, nur Objekte vom Typ `java.net.Socket` an die Anwendung weitergibt, ist es aus Sicht der Anwendung nicht möglich festzustellen, ob es mit der Erweiterung oder der Original-Implementierung der Java Socket Library arbeitet.

Im Rahmen von weiterführende Arbeiten wäre es interessant, wie sich der ebenfalls kurz vorgestellte Ansatz, welcher mithilfe von Zählern anstatt der Technik des Half-Close arbeitet, umsetzen lässt. Bei P2P-Anwendungen ist meistens nicht im Voraus bekannt, welche anderen Anwendungen ein Peer installiert hat. Deshalb ist es besonders wichtig, dass die Erweiterung der Java Socket Library mit anderen Anwendungen kompatibel ist. In dieser Hinsicht weist die in dieser Arbeit erstellte Implementierung noch Defizite auf, wie im Rahmen der Validierung der Implementierung erkannt wurde.

Auch gerade für die Umsetzung des Ansatzes bei der Verwendung von SSL eignet sich der Ansatz mithilfe von Zählern. Dies und ebenso die Umsetzung des Verfahrens für UDP wären also mögliche Themen für weiterführende Arbeiten.

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[CD08] Kenneth L. Calvert and Michael J. Donahoo. *TCP/IP sockets in Java: Practical guide for programmers*. The Morgan Kaufmann practical guides series. Elsevier, Amsterdam u.a., 2. ed. edition, 2008.

[Epp05] Jeffrey L. Eppinger. Tcp connections for p2p apps: A software approach to solving the nat problem, 2005. Abgerufen am 23.01.2011 unter: `http://reports-archive.adm.cs.cmu.edu/anon/isri2005/CMU-ISRI-05-104.pdf`.

[FSK05] Bryan Ford, Pyda Srisuresh, and Dan Kegel. Peer-to-peer communication across network address translators. In *Proceedings of the 2005 USENIX Annual Technical Conference*. 2005.

[Har04] Elliotte Rusty Harold. *Java network programming: [developing networked applications]*. O'Reilly, Beijing, 3. ed. edition, 2004.

[Pit06] Esmond Pitt. *Fundamental Networking in Java*. Springer, London, 2006.