Praktikumsversuch
Socket-Programmierung mit Java

Max Helbig

16. November 2017

Inhaltsverzeichnis

Kapitel 1

Ziele des Versuches

Das Internet, in den 1950er Jahren vom amerikanischen Verteidigungsministerium in Auftrag gegeben, wurde es in den 70ern für die Forschung geöffnet und damit zivilen Institutionen zugänglich. In Jahr 1990 ging „World“ als erster kommerzieller ISP (Internet Service Provider) online, und ebnete somit den Weg das Internet auch für die Bevölkerung direkt zugänglich zu machen. Heute ist das Internet das mit Abstand wichtigste Kommunikationsnetz, welches aus dem Alltag der Menschen nicht wegzudenken ist. In Jahr 2013 setzte der Bundesgerichtshof den Wert eines Internetzugangs faktisch sogar mit einem Grundrecht gleich.

In der heutigen Zeit, in der die Kommunikation von Menschen oder Maschinen über Datennetze unerlässlich ist, spiegelt sich die Nutzbarkeit selbiger natürlich in der Funktionalität moderner Programmiersprachen wieder, die somit eine simple Implementierung von Software mit Netzzugang ermöglicht. In diesem Praktikumsversuch sollen Sie mit ein paar simplen Java-Programmen eben diese Funktionalitäten kennenlernen und aufzeigen, wie diese die eigentlichen Vorgänge im Netzwerk abstrahieren.

Kapitel 2

Grundlagen

2.1 Einführung

Hier wird ihnen zunächst eine kurzer Einblick in den Aufbau und die Funktionsweise des Internets gegeben. Diese Einführung ist kurz gehalten, für genauere Informationen können die Lehrmittel der Vorlesung „Die Internet-Protokollwelt“ aus dem Fachgebiet Kommunikationsnetze genutzt werden.

Die Struktur des Internets basiert auf dem ISO-OSI Schichtenmodell, dass alle Funktionalitäten, die für eine Kommunikation notwendig sind, in Einzelfunktionen, die Schichten einteilt (Siehe 2.1).

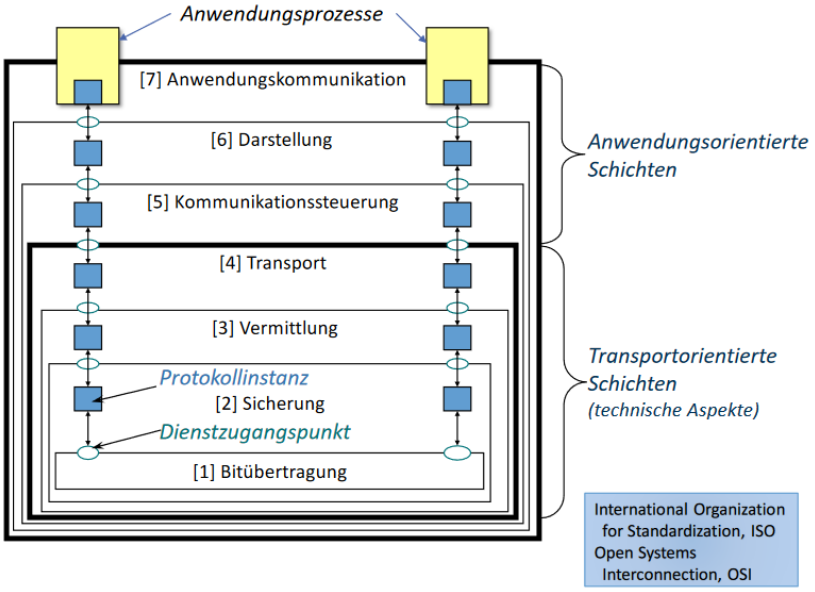

Abbildung 2.1: ISO-OSI Schichten

Hier ist zu beachten, dass nur auf der Bitübertragungsschicht physikalisch Daten ausgetauscht werden, und somit in allen anderen Schichten die Verbindungen nur logisch existieren. Die Bitübertragungsschicht überträgt die ihr gegebenen Daten, indem sie diese auf einen physikalischen Träger moduliert. Darüber befindet sich die Sicherungsschicht. Sie teilt

die Daten in Frames ein und fügt zu den Nutzdaten Redundanz hinzu, um Übertragungsfehler der Bitübertragungsschicht ausgleichen zu können. Schicht drei ist die Vermittlungsschicht, hier werden durch Routing Pfade zwischen Endgeräten geschaffen, welche für gewöhnlich keine direkte Verbindung haben, sondern über mehrere „Hops“ kommunizieren. In der vierten Schicht, der Transportschicht, können nun einzelne Prozesse in Endgeräten adressiert, sowie eine Übertragungsgarantie gegeben werden. Außerdem enthält sie Mechanismen zur Laststeuerung des Netzwerks.

Die folgende Abbildung2.1 zeigt, wie die Daten durch die Schichten weitergegeben werden.

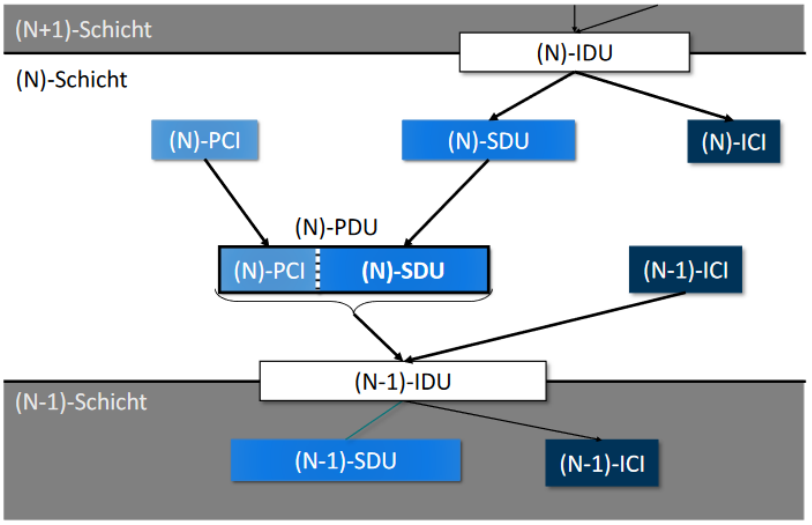

Abbildung 2.2: Interface

Eine Schicht ist für die Schicht über ihr immer der Dienstgeber. Um ihren Dienst ausführen zu können, benötigt die Schichtinstanz neben den Nutzdaten noch weitere Informationen (z.B. Adressen). Die Schnittstellendateneinheit (IDU) enthält somit eine Schnittstellenkontrollinformation (ICI) und die Dienstdateneinheit (SDU), welche die Nutzdaten darstellt, die beim Kommunikationspartner abgeliefert werden sollen. Die Schicht fügt nun noch eigene Protokollkontrolldaten (PCI) und wiederum ICI hinzu und übergibt die Daten der unter ihr liegenden Schicht.

Um die Abläufe des Netzwerkes zu abstrahieren, werden sogenannte Sockets eingesetzt. Sockets sind Teil der Betriebssystem-API und ermöglichen es somit Anwendungsprogrammierern den Netzwerkstack des Betriebssystems zu nutzen. Sockets können in zwei Gruppen eingeteielt werden: Stream- und Datagramm-Sockets. Die Funktionsweise von beiden wird in den folgenden Absätzen bezogen auf Java erläutert, ist jedoch auch in anderen Sprachen ähnlich.

2.2 Java UDP Sockets

Um das UDP-Protokoll mit einem Java-Programm zu nutzen, stehen die sogenannten „Datagram-Sockets“ zu Verfügung. Um eine Kommunikation zu ermöglichen, muss zunächst ein „DatagramSocket“-Objekt erzeugt und an einen Port gebunden werden, in dem

der Konstruktor mit einer Portnummer als Parameter aufgerufen wird. Sowohl für das Senden wie auch das Empfangen wird ein „DatagramPacket“-Objekt benötigt, das vorher angelegt werden muss. Zum Empfangen von Paketen wird die Methode `receive` des Sockets benutzt und ein Paketobjekt übergeben. Der Aufruf von `receive` blockiert, bis ein Paket empfangen wird.

```
DatagramSocket socket = new datagramSocket(2017);
byte[] packetdata = new byte[56];
DatagramPacket packet = new DatagramPacket(packetdata, packetdata.length)
socket.receive( packet );
```

Hierbei ist zu beachten, dass die Datengröße des Paketes vor dem Empfang festgelegt werden muss. In dem empfangenen Paket können nun die Daten sowie die Absender-Adresse und der Port ausgelesen werden.

```
packet.getData();
packet.getAddress();
packet.getPort();
```

Um ein Paket zu senden, wird die Methode „send“ des Sockets verwendet und wieder ein Paketobjekt übergeben. In diesem Fall werden die Felder für IP-Adresse und Port genutzt, um den Empfänger festzulegen.

Hinweis: Das DatagramPacket-Objekt besitzt nur jeweils ein Feld für die Portnummer bzw. die IP-Adresse, dies unterscheidet es von dem auf Netzwerkebene gesendeten Paketformat, welches Felder für Sender und Empfänger bietet.

Für detailliertere Informationen zu allen beschriebenen Java-Themen, kann das Buch „Java ist auch eine Insel“ (Christian Ullenboom; Rheinwerk Verlag) genutzt werden, das unter openbook.rheinwerk-verlag.de zum freien Lesen verfügbar ist.

2.3 Java TCP Sockets

Soll mit Java verbindungsorientiert gearbeitet werden, werden zwei Typen von Sockets benötigt. Diese werden durch die Klassen „Socket“ und „ServerSocket“ abgebildet. Wobei die Klasse ServerSocket nicht direkt an der Kommunikation beteiligt ist, sondern lediglich eingehende Verbindungen annimmt. Die eigentliche Kommunikation erfolgt dann über die Klasse Socket.

Um eingehende Verbindungen anzunehmen, wird ein Objekt der Klasse ServerSocket erzeugt und an einen Port gebunden. Danach wird die Methode `accept` verwendet, um die bereits angenommene Verbindung zu bearbeiten. Dieser socket ist nun die Schnittstelle mit der TCP-Verbindung. Die Methode `server.accept` ist blockierend, bis eine TCP-Verbindung aufgebaut wird. Zu beachten ist auch, dass der ServerSocket jeden TCP-Verbindungswunsch, den er auf dem angegebenen Port erhält, annimmt, noch bevor die `accept` Methode ausgeführt wird. Diese liefert dann immer einen Socket für die älteste angenommene und noch nicht abgerufene TCP-Verbindung. Sollen keine Verbindungen angenommen werden, muss der ServerSocket geschlossen werden.

```
server = new ServerSocket(port);
socket = server.accept();
server.close();
```

Steht eine TCP-Verbindung und wurde ein Socket-Objekt durch die `accept`-Methode erzeugt, kann man damit nun senden und empfangen. Im Gegensatz zu den Datagramm-Sockets ist die Kommunikation nicht paket basiert. Es gibt je einen Bitstrom für eine Richtung, diese Ströme erhält man durch `get`-Methoden auf das Socket-Objekt.

```
scanner = new Scanner(socket.getInputStream());
writer = new PrintWriter(socket.getOutputStream(), true);
```

Scanner und PrintWriter sind Klassen, die das Zugreifen auf die Datenströme ermöglichen. Je nach Anwendungsfall stehen hier in Java aber noch weitere Klassen zur Verfügung die auf unterschiedliche Weise den Zugriff ermöglichen. Die Klasse `scanner` teilt zum Beispiel den Datenstrom in Tokens auf, die durch ein konfigurierbares Zeichen von einander getrennt sind. Die Methode `scanner.hasNext` liefert den Wert False, wenn der Stream geschlossen wurde. Ist dies nicht der Fall, liefert sie True und blockiert solange, bis noch nicht abgerufene Tokens verfügbar sind.

```
while(scanner.hasNext()){
        System.out.println(scanner.next());
}
```

Das angegebene Codefragment würde solange einzelne Tokens untereinander ausgeben, bis der Socket und damit auch der Datenstrom geschlossen werden.

2.4 Threads in Java

Um in Java Threads einsetzen zu können, benötigt man zunächst erstmal Programmcode, der in einem neuen Thread ausgeführt werden soll. Wenn nun der Code in einer Klasse von einem Thread ausgeführt werden soll, muss diese das Interface `Runnable` implementieren, welches vorschreibt, dass eine Methode „run“ existieren muss.

```
public class UDPReceiver implements Runnable{
        DatagramSocket msocket;
        @Override public void run(){
```

Es gibt im Wesentlichen zwei Methoden, den geschriebenen Code parallelisiert auszuführen: Man kann die einzelnen Threads selbst verwalten, oder dies einem Executor überlassen.

```
private Thread udpReceiver;

        udpReceiver = new Thread( new UDPReceiver(socket));
        udpReceiver.setDaemon(true);
        udpReceiver.start();
```

In ersten Fall benötigt der Konstruktor des Threads ein Objekt einer Klasse, die Runnable implementiert, und muss danach noch gestartet werden. Der Thread existiert nun bereits im Betriebssystem, durch den Aufruf von `start` wird die `run`-Methode der eingebetteten Klasse ausgeführt. Daemon gibt an, ob ein Thread beendet werden soll wenn es keine nicht-Deamon-Threads (zum Beispiel das Hauptprogramm) mehr gibt. Um einen Thread zu beenden, kann er entweder direkt beendet werden oder man setzt in dem Thread lediglich das interrupt-Flag und lässt ihn darauf reagieren. Das Flag kann wie unten gezeigt innerhalb des Threads abgefragt werden.

```
udpReceiver.interrupt();
//innerhalb des Threads
@Override        public void run() {
                 while(!Thread.interrupted()){
                 ...
```

Die zweite Möglichkeit MUltithreading einzusetzen, besteht darin, die Threadverwaltung einem Executor zu überlassen. Diesem übergibt man wieder seine Runnable und muss sich um nichts weiter kümmern, der Executer entscheidet selbst, wie viele Threads erstellt werden und kann auch bereits beendete Threads wieder mit neuen eingebettenten Klassen starten. So können Executors auch Threads erzeugen, bevor diese benötigt werden, um dann zur Laufzeit performanter zu sein.

```
private ExecutorService mThreadPool = Executors.newCachedThreadPool();
mThreadPool.execute(new TCPClientLoop(socket));
```

2.5 Wireshark

Wireshark ist eine Software, die es ermöglicht sämtliche Pakete mitzuschneiden, die auf einem Netzwerkinterface transportiert werden, und die gesammelten Daten anschließend zu analysieren. In dieser Anleitung wird jedoch nur sehr grob auf die wichtigsten Funktionen eingegangen, die Sie für die Versuchsdurchführung benötigen.

Einen Mitschnitt starten Sie, in dem Sie im Hauptfenster von Wireshark ein oder mehrere Interfaces auswählen und den Startbutton drücken (siehe Screenshot). Die hier benötigten Netzwerkinterfaces sind „eth0“ und „Loopback: IO“

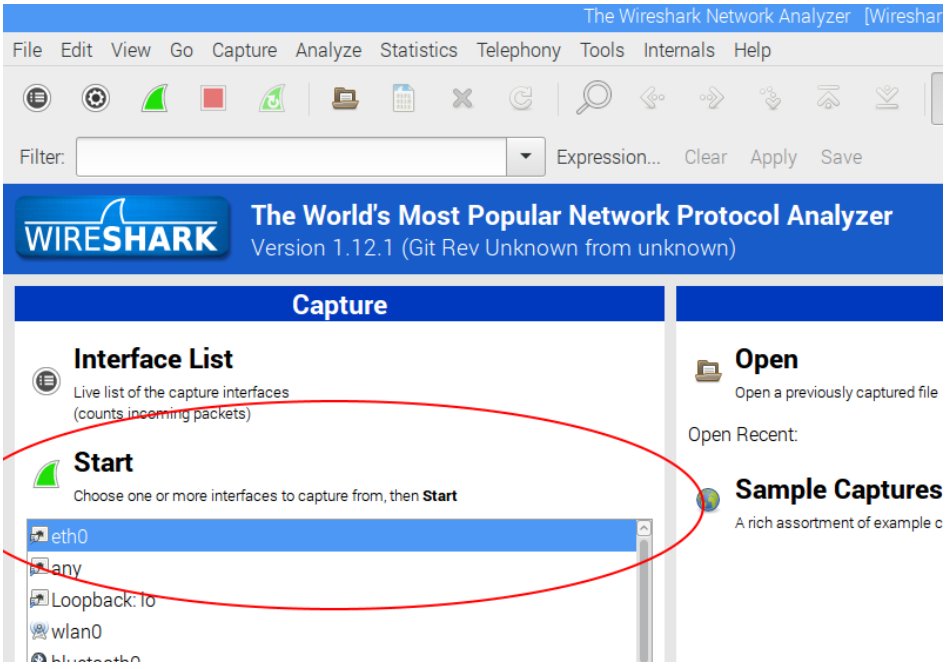

Abbildung 2.3: Auswahl Interface

Wireshark kann die mitgelesenen Pakete in einer Datei abspeichern oder aber auch direkt mit der Analyse beginnen. In diesem Praktikum ist ein simpler Filter und das nachträgli-

che Betrachten der Ergebnisse ausreichend. Um Pakete zu filtern, geben Sie einfach Ihre gewünschte Filterung in das Feld „Filter“ ein.

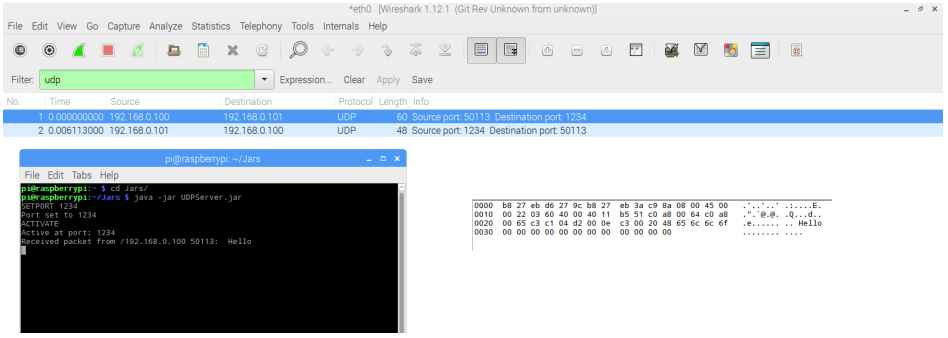

Abbildung 2.4: gefilterter Mitschnit

Somit können sie bereits sehr effektiv die Menge an zu betrachtenden Paketen auf das notwendigste beschränken. Im unteren Bereich des Fensters sehen Sie die gesamten Bytes des Paketes in Hexadezimal sowie ASCII-Schreibweise. (Der obere Screenshot zeigt den unteren Fensterbereich neben dem Terminal)

Hinweis: Die in Wireshark angegebenen Sequenznummern werden in der Standardeinstellung relativ dargestellt. Das heißt, es wird bei einer TCP-Verbindung mit Sequenznummer 0 begonnen um die Lesbarkeit zu vereinfachen. Beachten Sie, dass die tatsächlichen Sequenznummern meist mit einer Zufallszahl beginnen.

Kapitel 3

Versuchsaufbau

3.1 Hardware

Folgende Komponenten gehören zum Versuchsaufbau:

- Router (D-Link DIR-600)
- 4x Rasberry Pi 3 (Model B)
- KVM-Switch (CubiQ)
- Bildschirme und Eingabegeräte

Auf der rechten Seite befindet sich ein Rasberry Pi mit Eingabe- und Ausgabegeräten, im folgenden Server genannt. Auf der linken Seite sind drei Rasberry Pis über den KVM-Switch (Eingänge 1, 2, 3) mit den Eingabe- und Ausgabegeräten verbunden, im Weiteren Clients eins bis drei genannt. Um einen der Clients bedienen zu können, drücken Sie die entsprechende Zahl auf dem KVM-Switch.

Abbildung 3.1: Versuchsaufbau

Kapitel 4

Hausaufgaben

Die folgenden Fragen dienen dazu festzustellen, ob Sie die für den Versuch notwendigen Grundlagen verstanden haben. Wenn Sie mit der Beantwortung dieser Fragen Probleme haben, sollten Sie sich noch vor dem Versuchstermin mit den betreffenden Sachverhalten vertraut machen. Der Versuchsbetreuer wird diese Fragen vor dem Versuch mit ihnen durchgehen.

1. Beschreiben Sie kurz das ISO/OSI-Schichtenmodell
2. Was sind die Unterschiede zwischen TCP und UDP?
3. Beschreiben Sie mit eigenen Worten, was Sie unter einem Socket verstehen. Wenn Sie die Sockets einer Netzwerksschicht zuordnen müssten, welche wäre das?
4. Was sind well known ports? Geben sie einige Beispiele.
5. Um was kümmert sich DNS?
6. Im Kapitel Grundlagen wird der Empfang eines UDP-Paketes beschrieben. Wie groß sollte das Datenfeld des Paketes gewählt werden? Benennen Sie Vor- und Nachteile Ihrer Entscheidung.
7. Wie kann man den Zielport eines empfangenen UDP-Paketes in Java bestimmen?
8. Was passiert, wenn ein Java-Serversocket mehrere Verbindungsanfragen bekommt, aber die zugehörige `accept`-Methode nie ausgeführt wird?

Bonusfrage: Warum beginnen die Sequenznummern bei einer TCP-Verbindung meistens bei einer Zufallszahl?

Kapitel 5

Versuchsaufgaben

Starten Sie die Rechner und melden Sie sich als Benutzer „SocketPrak“ mit dem Passwort „Socket“ an. Bitte beachten Sie, dass die Eclipseprojekte bei jedem Start zurückgesetzt werden. Sie sollten die Rechner also während ihres Versuchs nicht neu starten.

5.1 UDP

Programmierung Starten Sie Eclipse auf einem Client und dem Server, und navigieren Sie zu den entsprechenden Projekten (UDPClient; UDPServer). Der UDP-Server soll eingehende Pakete spiegeln, also mit gleichem Inhalt an den Absender zurückschicken. Die Implementierung der Servers ist unvollständig und muss von Ihnen ergänzt werden. Im aktuellen Zustand empfängt er Pakete und gibt eine Empfangsbestätigung auf der Kommandozeile aus, dies können sie bereits ausprobieren. Die Aufgabe besteht darin, den Server so zu erweitern, dass er die empfangenen Pakete zurück an den Sender schickt. Hierzu können Sie die Implementierung des Clients als Anhaltspunkt verwenden.

Hinweis: Verändern Sie hierfür die Klasse „UDPServerLoop“.

Analyse Wenn sie Ihre eigene Implementierung benutzen, klicken sie in Eclipse auf „run“ und wählen, falls sie gefragt werden, „Java Application“ als Ausführungstyp aus.

Für den folgenden Schritt wird die vollständige Implementierung des Servers benötigt. Sollten Sie die Implementierung nicht geschafft haben, starten Sie hierfür auf dem Server die UDPServer.jar.

Die Jar-Dateien aller benötigten Programme befinden sich unter: /home/SocetPrak/Jars. Um die Applikation auszuführen, starten Sie ein Terminal und navigieren zum oben genannten Verzeichnis. Mit folgendem Aufruf starten sie das Programm:

```
java -jar UDPServer.jar
```

Ist der Server gestartet, setzen Sie einen Port, auf dem der Server auf Pakete warten soll, und starten ihn anschließend. bspw.

```
SETPORT 1234
ACTIVATE
```

Eine Liste mit allen in den Programmen nutzbaren Befehlen finden Sie im Anhang.

Nun wartet der Server auf Pakete und schickt diese an den Empfänger zurück.

Auf den Clients starten Sie entsprechend die UDPClient.jar. Hier müssen Sie den Port und den Host eingeben, zu dem Pakete geschickt werden sollen, bspw.

```
SETPORT 1234
SETHOST 192.168.0.100
```

Jetzt können Sie mit dem Befehl SEND Daten an den Server senden, bspw.

```
SEND Dies ist ein Test.
```

Sind alle Einstellungen korrekt, sollten sie sowohl auf dem Server als auch auf dem Client eine Empfangsbestätigung bekommen. Nachdem Sie die Verbindung getestet haben, starten Sie auf Server und Client nun Wireshark und beginnen Sie einen Mitschnitt auf dem Interface „eth0“ (Ethernetverbindung)

Jetzt senden Sie von unterschiedlichen Clients beliebige Daten an den Server (Sie können auch mehrere Instanzen des Clients auf einem Rechner starten)

Nachdem Sie nun ein paar Datenpakete gesendet haben, können Sie den Mitschnitt auswerten. Dazu filtern Sie Ihre Pakete entsprechend der Anforderungen, um die Übersichtlichkeit zu bewahren. Identifizieren Sie in den Datenpaketen die IP-Adressen und Portnummern sowie Ihre gesendeten Nutzdaten. Beschreiben Sie ein Datenpaket, das ein Client an den Server gesendet hat, sowie dessen direkte Antwort. Versuchen sie Client und Server auf dem gleichen Rechner auszuführen und einen Mitschnitt zu machen. Können Sie die von Ihnen gesendeten Pakete auf dem „eth0“-Interface mitlesen?

5.2 TCP Single Connection

Programmierung Starten Sie Eclipse auf einem Client und dem Server, und navigieren sie zu den entsprechenden Projekten (TCPClient; TCPSingleConServer). Der TCP-Server soll immer genau eine eingehende Verbindung annehmen. Während die Verbindung steht, soll der Server alle ankommenden Daten spiegeln. Im aktuellen Projekt, nimmt der Server mehrere Verbindungen an, kann jedoch nur eine bedienen. Ihre Aufgabe besteht darin, den Server so umzuschreiben, dass er nur eine Verbindung gleichzeitig annimmt.

Hinweis: verändern sie dafür die Klasse „TCPConnectionLoop“.

Analyse Starten Sie mehrere TCP-Clients und den TCP-Server. Sollte Ihre Implementierung der Servers nicht korrekt funktionieren, können Sie die TCPSingleConServer.jar verwenden. Im Server muss wieder mit SETPORT konfiguriert werden, auf welchem Port er auf Verbindungen warten soll. Ist dies geschehen, kann er mit ACTIVATE gestartet werden. Auf den Clients müssen wieder Port und IP-Adresse gesetzt werden. Bevor jedoch Daten gesendet werden können, müssen Sie mit dem Befehl CONNECT eine Verbindung zum Server aufbauen. Entsprechend können Sie die Verbindung mit DISCONNECT trennen.

Starten Sie Wireshark und beginnen sie einen Mitschnitt, wie bereits im Unterkapitel UDP5.1 beschrieben. Senden Sie nun einige Daten und beenden den Mitschnitt. Filtern Sie die für Sie wichtigen Pakete heraus und analysieren Sie sie. Legen Sie besonderes Augenmerk auf die Acknowledgements und dazugehörigen Sequenznummern. Erläutern Sie, wie diese mit Ihren Nutzdaten in Zusammenhang stehen.

5.3 TCP Multi Connection

Programmierung Starten Sie Eclipse auf einem Client und dem Server, navigieren sie zu den entsprechenden Projekten (TCPClient; TCPMultiConServer). Der TCP-Server soll alle eingehenden Verbindungen annehmen. Während die Verbindung steht, soll der Server alle ankommenden Daten spiegeln. Im aktuellen Projekt, nimmt der Server mehrere Verbindungen an, kann jedoch keine bedienen. Ihre Aufgabe besteht darin, den Server so umzuschreiben, dass er mehrere Clients gleichzeitig bearbeiten kann.

Hinweis: Verändern Sie hierfür die Klasse „TCPMultiConServer“

Analyse Starten Sie mehrere TCP-Clients und den TCP-Server. Sollte Ihre Implementierung der Servers nicht korrekt funktionieren, können Sie die TCPMultiConServer.jar verwenden. Im Server muss wieder mit SETPORT konfiguriert werden, auf welchem Port er auf Verbindungen warten soll. Ist dies geschehen, kann er mit ACTIVATE gestartet werden. Auf den Clients müssen wieder Port und IP-Adresse gesetzt werden. Bevor jedoch Daten gesendet werden können, müssen Sie mit dem Befehl CONNECT eine Verbindung zum Server aufbauen. Entsprechend können Sie die Verbindung mit DISCONNECT trennen.

Starten Sie Wireshark und beginnen sie einen Mitschnitt wie bereits im Unterkapitel UDP5.1 beschrieben. Senden Sie nun einige Daten und beenden den Mitschnitt. Filtern Sie die für sie wichtigen Pakete heraus und analysieren Sie sie. Legen Sie besonderes Augenmerk auf das Aufbauen und Abbauen von Verbindungen und die Unterscheidung unterschiedlicher TCP-Verbindungen. Erläutern Sie anhand des Mitschnittes einen TCP Handshake für einen Verbindungsaufbau und Abbau.

5.4 URL

Starten Sie eine Aufzeichnung mit Wireshark.

Nutzen Sie ihr Programm, um die unter „www.tu-ilmenau.de“ zu Verfügung stehende HTML-Datei zu laden und auf der Kommandozeile auszugeben.

Starten Sie das Programm URL (entweder mit der Jar-Datei oder aus Eclipse heraus).

Um ein URL-Objekt zu erstellen können, Sie folgende Befehle Verwenden:

- SETHOST
- SETPORT
- SETPROTOCOL
- CREATE
- REQUEST

Nachdem Sie die Parameter gesetzt haben, können Sie mit CREATE das URL-Objekt erzeugen und mit REQUEST das Ziel Ihrer Anfrage auf der Konsole ausgeben.

Beenden Sie den Mitschnitt und identifizieren Sie die HTTP- sowie DNS-Anfrage.

Kapitel 6

Verzeichnis der verwendeten Abkürzungen

TCP	Transmission Control Protocol
API	Application Programming Interface
UDP	User Datagram Protocol
DNS	Domain Name System
URL	Uniform Resource Locator
IDU	Interface Data Unit
ICI	Interface Control Information
SDU	Service Data Unit
PCI	Protocol Control Information
PDU	Protocol Data Unit

Verfügbare Befehle in den Programmen

UDPClient:

Befehl	Beschreibung
SETPORT	Setzt den Port auf die nachfolgend eingegebene Zahl
SETHOST	Setzt den Host auf den eingegebenen Wert (IP oder Name)
SEND	Sendet den nachfolgend eingegebenen Text in einen UDP-Paket
EXIT	Beendet das Programm

UDPServer:

Befehl	Beschreibung
SETPORT	Setzt den Port auf die nachfolgend eingegebene Zahl
ACTIVATE	Startet den Serverdienst
DEACTIVATE	Beendet den Serverdienst
EXIT	Beendet das Programm

TCPClient:

Befehl	Beschreibung
SETPORT	Setzt den Port auf die nachfolgend eingegebene Zahl
SETHOST	Setzt den Host auf den eingegebenen Wert (IP oder Name)
CONNECT	Baut eine Verbindung zu einem Server auf
DISCONNECT	Baut die Verbindung zum Server ab
SEND	Sendet den nachfolgend eingegebenen Text über einen TCP-Datenstrom
EXIT	Beendet das Programm

TCPServer:

Befehl	Beschreibung
SETPORT	Setzt den Port auf die nachfolgend eingegebene Zahl
ACTIVATE	Startet den Serverdienst
DEACTIVATE	Beendet den Serverdienst
EXIT	Beendet das Programm

URL:

Befehl	Beschreibung
SETPORT	Setzt den Port auf die nachfolgend eingegebene Zahl
SETHOST	Setzt den Host auf den eingegebenen Wert (IP oder Name)
SETPROTOCOL	Setzt das Protokoll auf das nachfolgend eingegebene
CREATE	Erzeugt aus den eingegebenen Parametern ein URL-Objekt
REQUEST	Gibt die Ressource aus auf die die URL zeigt (url.openStream())
EXIT	Beendet das Programm