TECHNISCHE UNIVERSITÄT MÜNCHEN
Institut für Informatik
Lehrstuhl für Netzwerkarchitektur
Prof. Dr. Anja Feldmann,
Vinay Aggarwal, Manfred Jobmann, Nils Kammenhuber, Jörg Wallerich, Arne Wichmann

5. Übung zur Internet-Praktikum

Auf dem zweiten Übungsblatt hatten wir einen Dateitransfer-Client, -server und -proxy programmiert. In dieser Aufgabe soll nun der Proxy so erweitert / verändert[1] werden, dass daraus ein einfaches Chat-System entsteht. Dieses System soll wie folgt funktionieren:

- Nachrichten können von jedem Client versendet werden
- und werden an alle anderen teilnehmenden Clients verteilt.
- Die Nachrichten werden durch die Server an die Clients verteilt.
- Ein Client nimmt am System teil, wenn er sich zu einem Server verbindet.
- Ein Server kann auch als Client fungieren (siehe Teil b) und ist daher eine Obermenge (vom Funktionsumfang gesehen) des Clients. Es genügt also ein Programm zu schreiben, welches die Funktionen des Servers implementiert, sich aber auch als normaler Client verhalten kann.

Protokollspezifikation

- Unser Protokoll benutzt TCP.
- Jede Protokollnachricht besteht aus einer Zeile. Diese Zeile beginnt mit einem dreistelligen Zahlencode, gefolgt von einem Kommando und einer kommandoabhängigen Anzahl von Parametern.
- Alle Zeilen (Protokoll- und Nachrichtenzeilen) enden mit $\backslash r\backslash n$.
- Alle Kommandos und Response-Nachrichten bestehen nur aus Großbuchstaben.
- Aufbau einer Peer-Verbindung (*handshake*):
  - Derjenige Knoten, der eine Verbindung anfordert (normalerweise ein Client), sendet folgendes Kommando:
    `100 OPEN <client-ID>`[2]
  - Der andere Knoten antwortet darauf mit
    `110 OPEN <client-ID>`
    oder im Fehlerfall mit
    `400 INVALID COMMAND`
  - Der erste Knoten schließt den Handshake ab mit
    `200 OK`
    bzw. mit
    `410 HANDSHAKE FAILED`

[1] Wenn euer Proxy vom zweiten Übungsblatt nicht so gut erweiterbar war, dass man ihn leicht verändern kann, könnte es sich durchaus lohnen, für dieses Blatt neu anzufangen und ein strukturiertes Programm zu schreiben, da noch spätere Übungsblätter auf diesem aufbauen werden.

[2] Mit `<client-ID>` usw. ist natürlich jeweils der Wert *ohne* die spitzen Klammern <> gemeint.

  - Sollte der erste Knoten nicht mit Codes 200 oder 410 antworten, so wird die Verbindung mit
    `420 HANDSHAKE FAILED`
    abgebrochen.

- Versenden von Text-Nachrichten:
  - Eine Textnachricht wird durch folgende Kommandozeile eingeleitet:
    `100 MESSAGE ID <client-ID>:<message-ID> CHANNEL <channel-ID> LINES <Anzahl Zeilen>`
  - Danach kommt die Text-Nachricht, die aus genau der Anzahl Zeilen bestehen muß, wie in der Kommandozeile angegeben.
  - Diese Nachricht wird an alle Nachbarn weitergeleitet.
  - Jeder Knoten, der eine Text-Nachricht empfängt, muß entscheiden, ob er diese Nachricht selbst haben will, und leitet dann diese Nachricht an alle Nachbarn, ausser dem Nachbarn, von dem die Nachricht empfangen wurde, weiter (*flooding*).

- Regulärer Verbindungsabbau:
  - Um sich bei seinem Nachbarn abzumelden, dient das Kommando
    `100 EXIT <client-ID>`
  - Ein solches Kommando wird mit
    `200 OK`
    beantwortet, die Verbindung beendet und alle Statusinformation für diesen Knoten gelöscht.

Dein Server und Dein Client sollten in der Lage sein, von der Standardeingabe einfache Anweisungen zu akzeptieren und auszuführen. Diese Anweisungen dienen zum Versenden von Nachrichten und dem Verbinden zu anderen Knoten. Folgende Liste von Anweisungen ist ein **Beispiel**!

- `connect <IP> <Port>`
- `disconnect <IP> <Port>`
- `send <Channel-Nummer>`

In diesem **Beispiel** würde man nach einer `send`-Anweisung eine beliebige Anzahl von Nachrichten-Zeilen eingeben und das Ende der Eingabe durch eine Zeile kennzeichnen, die nur aus einem Punkt besteht.

Aufgabe 1: (80 Punkte) Ein einfaches Chat-System

Wichtiger Hinweis: Diese Aufgabe ist in drei Teile (a,b und c) gegliedert. Diese Gliederung soll euch nur helfen, schrittweise eine Implementation zu entwickeln.

(a) **Ein Server, mehrere Clienten**

Der einfachste Fall besteht darin, dass ein Server S mehrere Clienten {a,b,c,...} hat (vergleiche *Abbildung 1*). Alle Clienten müssen sich zum Server S verbinden, damit sie am Chat-System teilnehmen können. Eine Nachricht die einer der Clienten (z.B. a) über seine Verbindung zum Server an S sendet, wird von S an alle anderen Clienten (b,c,...) verteilt, so dass alle die Nachricht lesen können.

Implementiere einen Chat-Server und einen Chat-Client, die obiges Protokoll miteinander sprechen können. Der Server muss dabei mehrere bestehende Verbindungen zu Clients verwalten können und dabei gleichzeitig neue Verbindungsanfragen beantworten können. Der Client soll sowohl Nachrichen vom Server erhalten und anzeigen können, als auch im Stande sein, Kommandos über die Tastatur zu lesen und gegebenenfalls eine Nachricht an seinen zuständigen Server senden zu können.

Für das Protokoll gilt für diese Aufgabe folgende (vereinfachende) Einschränkung:

Bei Nachrichten vom Typ `100 MESSAGE` soll als Client-ID `CLIENTID` und als Channel-ID `UNIVERSE` benutzt werden.

Abzugeben ist:

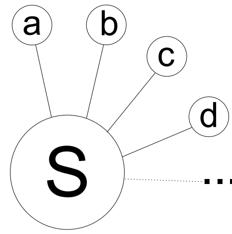

Abbildung 1: Erster Fall: ein Server, viele Clients

- Der Quelltext Deines Programmes

(b) **Ketten von Servern**

Ein etwas komplizierterer Fall ist gegeben, wenn ein Client eines Servers S selbst ein Server S' ist (siehe Abbildung 2). Beide Server haben ihrerseits Clienten ({a,b,c,...} und {a',b',c',...}).

Passt gegebenenfalls euren Server so an, dass er sich mit anderen Servern verbinden kann. Wieder sollen, sobald ein Client eine Nachricht an einen der Server sendet (im Beispiel S oder S'), alle anderen Clienten (z.B. b, a' und b', wenn a die Nachricht gesendet hat) diese Nachrichten sehen.

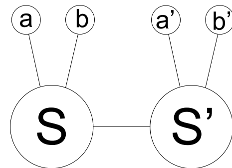

Abbildung 2: Zweiter Fall: Serverkette

Zum Testen eurer Implementation solltet ihr nur Serverketten und *keine* „Ringverbindungen" in eure Servertopologie einbauen (siehe Teil c).

Für das Protokoll gilt dieselbe Einschränkung wie für Teil a).

Abzugeben ist:

- Der Quelltext Deines Programmes

(c) **Vermeidung von Loops**

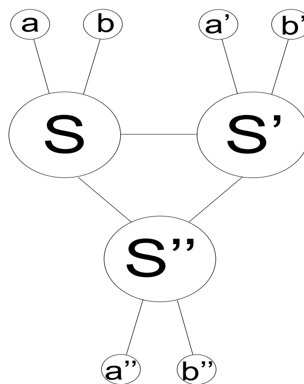

Abbildung 3: Dritter Fall: Ringtopologie

Das größte Problem bei dem bis jetzt beschriebenen System ist, dass es nicht mit Schleifen in der Topologie umgehen kann (siehe Abbildung 3). Ein Server hat Verbindung zu mehreren anderen Servern, die untereinander auch verbunden sind. Sendet ein Server alle Nachrichten, wie bisher beschrieben, an alle Verbindungen außer an diejenige, von der die Nachricht kommt, so wird das Packet unendlich oft an alle Clients ausgesendet, denn es läuft unendlich lange im Kreis herum (*loop*).

Erweitere deinen Chat-Server um eine Schleifenerkennung (*loop detection*).

Dazu werden jetzt bei `100 MESSAGE`-Nachrichten die Client-ID und die Message-ID benötigt und sind deshalb auch auszufüllen. Durch diese beiden IDs soll eine Nachricht im gesamten Netzwerk eindeutig wiederzuerkennen sein. Dazu soll eine Methode angegeben werden, wie man diese IDs so erstellen kann, dass die Eindeutigkeit garantiert wird.

Der Server muss sich jetzt merken, welche Nachricht er von welchem Client schon gesehen hat. Wenn eine Nachricht zum wiederholten mal empfangen wird, dann wird sie einfach verworfen (*drop*). Dazu sollen die IDs aus der `100 MESSAGE`-Nachricht genutzt werden.

Die Information, welche Nachrichten schon einmal gesehen wurden, soll platzsparend verwaltet werden, also nicht einfach ewig in einer Tabelle abgelegt werden. Dazu soll in regelmäßigen Zeitabständen alle Informationen verworfen werden, die ein Alter von 1 Minute überschritten haben.

Abzugeben sind:

- Spezifikation für die Erstellung eindeutiger Nachrichten-IDs (als Textdatei)
- Der Quelltext Deines Programmes

Details zur Abgabe der Aufgaben: siehe FAQ
(unterhalb `http://www.net.in.tum.de/teaching/SS05/inetprak/`**).**
Abgabedatum: 17.5.2005 23:59h s.t.